

# درک الگوریتم

راهنمای تصویری  
برای برنامه‌نویس‌ها  
و افراد کنجکاو

آدیتیا بهارگاوا

مترجم: مهران افشارزادری

عنوان و نام پدیدآور: درک الگوریتم (راهنمای تصویری برای برنامه‌نویس‌ها و افراد کنجکاو) // آدیتیا بهارگاو؛

مترجم: مهران افشارنادری

مشخصات نشر: تهران، ۱۴۰۲-

مشخصات ظاهری: دیجیتال، ۲۸۸ ص، مصور

موضوع: الگوریتم

صفحه‌آرایی: نفیسه عطاران

# درک الگوریتم

راهنمای تصویری  
برای برنامه‌نویس‌ها  
و افراد کنجکاو

آدیتیا بهارگاوا

مترجم: مهران افشارنادر

.....

تقدیم به پدر و مادرم، یُگِش و سَنگینا

قدردانی مترجم :

سپاس از جادی عزیز که کتاب را پیش از انتشار خواند و پیشنهادهایی در جهت بهتر شدن کیفیت ترجمه داد. با این حال هرگونه اشکال در متن نهایی متوجه مترجم است.





## فهرست

i	پیش‌گفتار
ii	قدردانی
iii	درباره‌ی این کتاب

## ۱ درآمدی بر الگوریتم

۱	مقدمه
۲	آن چه درباره‌ی کارایی خواهید آموخت
۲	آن چه در مورد حل مسئله یاد می‌گیرید
۴	جست‌وجوی دودویی
۶	یک راه بهتر برای جستجو
۱۳	زمان اجرا
۱۴	نماد $O$ بزرگ
۱۴	زمان اجرای الگوریتم با سرعت‌های متفاوتی رشد می‌کند
۱۷	به تصویر کشیدن چند زمان اجرا با $O$ بزرگ متفاوت
۱۹	$O$ بزرگ برای بدترین زمان اجرا
۲۰	برخی از زمان‌های معمول $O$ بزرگ
۲۲	فروشنده‌ی دوره‌گرد
۲۵	جمع‌بندی

## ۲ مرتب‌سازی انتخابی

---

۲۷	
۲۸	شیوه‌ی عملکرد حافظه
۳۰	آرایه‌ها و لیست‌های پیوندی
۳۱	لیست‌های پیوندی
۳۳	آرایه‌ها
۳۵	اصطلاح‌شناسی
۳۶	درج در وسط یک لیست
۳۷	حذف
۴۱	مرتب‌سازی انتخابی
۴۶	جمع‌بندی

## ۳ بازگشت

---

۴۷	
۴۸	بازگشت
۵۱	صورت پایه و صورت بازگشتی
۵۳	پشته
۵۴	پشته فراخوانی
۵۸	پشته فراخوانی با بازگشت
۶۳	جمع‌بندی

## ۴ مرتب‌سازی سریع

---

۶۵	
۶۶	تقسیم و حل
۷۴	مرتب‌سازی سریع
۸۱	بازبینی نماد O بزرگ
۸۲	مرتب‌سازی ادغامی در برابر مرتب‌سازی سریع

۸۴	حالت متوسط در مقابل بدترین حالت
۸۸	جمع‌بندی

## ۵ جدول‌های هش

---

۹۲	توابع هش
۹۷	موارد کاربرد
۹۷	استفاده از جدول‌های هش برای جست‌وجو
۹۹	جلوگیری از ورودی‌های تکراری
۱۰۲	استفاده از جدول هش برای کش
۱۰۵	جمع‌بندی
۱۰۵	تصادم‌ها
۱۰۸	کارایی
۱۱۱	ضریب بار
۱۱۳	تابع هش مناسب
۱۱۵	جمع‌بندی

## ۶ جست‌وجوی سطح اول

---

۱۱۸	مقدمه‌ای بر گراف‌ها
۱۲۱	گراف چیست؟
۱۲۲	جست‌وجوی سطح اول
۱۲۵	یافتن کوتاه‌ترین مسیر
۱۲۷	صف
۱۲۹	پیاده‌سازی گراف
۱۳۱	پیاده‌سازی الگوریتم
۱۳۶	زمان اجرا

۱۴۰ جمع‌بندی

۱۴۱ ۷ الگوریتم دیجسترا

---

۱۴۳ کار با الگوریتم دیجسترا

۱۴۸ اصطلاح‌شناسی

۱۵۱ معامله بر سر یک پيانو

۱۵۸ يال‌های با وزن منفي

۱۶۱ پياده‌سازی

۱۷۲ جمع‌بندی

۱۷۳ ۸ الگوریتم حریشانه

---

۱۷۴ مسئله‌ی برنامه‌ریزی برای کلاس درس

۱۷۷ مسئله‌ی کوله‌پشتی

۱۷۹ مسئله‌ی پوشش مجموعه

۱۸۱ الگوریتم‌های تقریبی

۱۸۸ تشخیص مسئله‌ی ان‌پی کامل

۱۹۰ فروشنده‌ی دوره‌گرد، گام به گام

۱۹۵ تشخیص مسئله‌ی ان‌پی کامل

۱۹۸ جمع‌بندی

۱۹۹ ۹ برنامه‌نویسی پویا

---

۱۹۹ مسئله‌ی کوله‌پشتی

۲۰۰ راه‌حل ساده

۲۰۱ برنامه‌نویسی پویا

۲۱۱	پرسش‌های متداول در مسئله‌ی کوله‌پشتی
۲۱۱	با اضافه کردن یک آیتم دیگر چه اتفاقی می‌افتد؟
۲۱۴	چه اتفاقی می‌افتد اگر ترتیب ردیف‌ها را تغییر بدهید؟
۲۱۴	آیا می‌توان شبکه را به جای ردیفی ستونی پر کرد؟
۲۱۵	چه اتفاقی می‌افتد اگر یک آیتم کوچک‌تر اضافه کنید؟
۲۱۵	آیا می‌توان کسری از یک آیتم را سرقت کرد؟
۲۱۶	بهینه‌سازی برنامه‌ی سفر
۲۱۸	بررسی آیتم‌های وابسته به یکدیگر
۲۱۸	آیا ممکن است که این راه حل نیاز به بیش از دو زیر-کوله‌پشتی داشته باشد؟
۲۱۹	آیا ممکن است با بهترین راه حل، کوله‌پشتی به طور کامل پر نشود؟
۲۲۰	طولانی‌ترین زیررشته‌ی مشترک
۲۲۱	ساخت شبکه
۲۲۲	پرکردن شبکه
۲۲۴	راه حل
۲۲۵	بزرگ‌ترین زیردنباله‌ی مشترک
۲۲۷	بزرگ‌ترین زیردنباله‌ی مشترک - راه حل
۲۲۹	جمع‌بندی

## ۱۰ الگوریتم K - نزدیک‌ترین همسایه

---

۲۳۱	طبقه‌بندی پرتقال در مقابل گریپ‌فروت
۲۳۴	ایجاد سیستم پیشنهاد
۲۳۵	استخراج ویژگی
۲۴۰	رگرسیون
۲۴۳	انتخاب ویژگی‌های خوب
۲۴۴	مقدمه‌ای بر یادگیری ماشین
۲۴۶	ساخت فیلتر اسپم

۲۴۷ پیش‌بینی بازار سهام  
۲۴۷ جمع‌بندی

## ۲۴۹ ۱۱ گام بعدی

---

۲۴۹ درخت  
۲۵۳ ایندکس‌های معکوس  
۲۵۴ تبدیل فوریه  
۲۵۵ الگوریتم‌های موازی  
۲۵۷ نگاشت کاهش  
۲۵۷ چرا الگوریتم‌های توزیع‌شده مفید هستند؟  
۲۵۸ تابع map  
۲۵۹ تابع reduce  
۲۶۰ فیلترهای بلوم و هایپرلاگ لاگ  
۲۶۱ فیلترهای بلوم  
۲۶۲ هایپرلاگ لاگ  
۲۶۳ الگوریتم‌های SHA  
۲۶۳ مقایسه‌ی فایل‌ها  
۲۶۵ چک‌کردن رمزهای عبور  
۲۶۶ هیشنگ حساس به مکان  
۲۶۷ تبادل کلید دیفی هلمن  
۲۶۹ برنامه‌ریزی خطی  
۲۷۱ پاسخ تمرین‌ها









## پیشگفتار

در ابتدا، برنامه‌نویسی برای من تنها یک سرگرمی بود. مبانی برنامه‌نویسی را از کتاب *Visual Basic 6 for Dummies* آموختم و برای فهم بیشتر به خواندن کتاب‌های دیگر ادامه دادم. اما موضوع الگوریتم سخت و غیر قابل درک باقی ماند. به خاطر دارم که فهرست مطالب اولین کتاب الگوریتمی را که داشتم، مزه‌مزه می‌کردم و به این فکر می‌کردم که «بالاخره این موضوعات را درک خواهم کرد!» اما مطالب سنگین بود و پس از چند هفته منصرف شدم. تا بعد اولین استاد الگوریتم خوبم را پیدا کردم و فهمیدم این ایده‌ها چقدر ساده و زیبا هستند.

چند سال پیش، اولین پست تصویری وبلاگم را نوشتم. اصولاً موضوعات را در قالب دیداری بهتر یاد می‌گیرم و سبک مصور را خیلی دوست دارم. از آن زمان، چند پست مصور در مورد برنامه‌نویسی فانکشنال<sup>۱</sup>، گیت<sup>۲</sup>، یادگیری ماشین<sup>۳</sup> و هم‌زمانی<sup>۴</sup> نوشتم. راستی زمانی که شروع به کار کردم نویسنده‌ی متوسطی بودم. توضیح مفاهیم فنی سخت است و ارائه‌ی مثال‌های خوب و توضیح یک مفهوم دشوار، زمان‌بر. بنابراین راحت‌ترین کار نادیده‌گرفتن مفاهیم سخت است. فکر می‌کردم چقدر کارم را خوب انجام می‌دهم، ولی بعد از محبوبیت یکی از پست‌هایم، یکی از همکارانم گفت: «پست تو را خواندم ولی چیزی سر در نیاوردم.» فهمیدم هنوز راه درازی در مسیر یادگیری نویسنده‌گی در پیش دارم.

در میانه‌ی نوشتن این پست‌های وبلاگ، انتشارات مینینگ به سراغ من آمد و از من پرسید که تمایلی دارم که یک کتاب مصور بنویسم؟ خوب، معلوم شد که ویراستاران مینینگ در مورد توضیح مفاهیم فنی چیزهای زیادی می‌دانند و به من آموختند که چگونه تدریس کنم. این کتاب را نوشتم تا این وسوسه را عملی کنم: می‌خواستم کتابی

1. functional programming
2. git
3. machine learning
4. concurrency

بنویسم که موضوعات فنی سخت را به خوبی توضیح بدهد، و کتاب الگوریتمی می‌خواستیم بنویسم که آسان خوان باشد. نوشته‌های من با اولین پست وبلاگ فاصله‌ی زیادی دارد، و امیدوارم این کتاب برای شما آسان و آموزنده باشد.

## قدردانی

از انتشارات منینگ تشکر می‌کنم که فرصت نوشتن این کتاب را در اختیار من قرار داد و دستم را برای خلاقیت باز گذاشت. از ناشر کتاب مرجان بیس و مایک استفنز برای همراهی، برت بیتس برای آموزش نویسندگی، و جنیفر استات چون ویراستاری فوق‌العاده پاسخگو و یاری‌رسان بود، تشکر می‌کنم. همچنین از افراد تیم تولید منینگ: کوین سالیوان، مری پی‌ریگز، تیفانی تیلور، لزی هایمز، و بقیه‌ی افراد دست‌اندرکار متشکرم. علاوه بر این، می‌خواهم از کسانی که دست‌نوشته‌ی کتاب را خواندند و پیشنهادهای ارائه کردند تشکر کنم: کارن بنسدون، راب گرین، مایکل همرا، اوزن هارلوویچ، کالین هستی، کریستوفر هاپت، چاک هندرسون، پاول کوزلوفسکی، آمیت لامبا، ژان فرانسوا مورین، رابرت موریسون، سانکار. راماناتان، ساندر راسل، داگ اسپارلینگ و دیمین وایت.

با تشکر از افرادی که به من کمک کردند تا به این نقطه برسیم: رفقای Flaskhit game board که به من کدنویسی را یاد دادند؛ دوستان زیادی از جمله بن سرکه، کارل پوزون، الکس منینگ، استر چان، آنیش بات، مایکل گلس، نیکراد مهدی، چارلز لی، جرد فریدمن، هما مانیکاوا و اساکام، هاری راجا، مورالی گودیپاتی، سرینیواس وارادان و دیگران که با مرور فصل‌ها، مشاوره دادند و فرصتی برای محک زدن اشکال گوناگون ارائه‌ی توضیحات را برایم فراهم کردند و سپاس از گری بردی، برای آموزش الگوریتم‌ها. یک تشکر اساسی دیگر از اساتید CLRS<sup>۱</sup>، کنوت و استرنگ، به راستی که روی شانه‌های غول‌ها ایستاده‌ام.

بابا، مامان، پریانکا و بقیه‌ی اعضای خانواده: از حمایت همیشگی شما متشکرم. و یک تشکر بزرگ از همسر مگی. ماجراهای زیادی در پیش روی ما وجود دارد، که البته محدود به آخر هفته‌ها در خانه ماندن و بازنویسی پاراگراف‌ها نمی‌شود.

در نهایت، سپاس فراوان از خوانندگانی که کتاب را مطالعه کردند و آنانی که به من در انجمن این کتاب بازخورد دادند تشکر می‌کنم. شما واقعاً به بهتر شدن این کتاب کمک کردید.

۱. توماس اچ کورمن چارلز ای لایسرسان رونالد ریوست کلیفورد استین نویسندگان کتاب مقدمه‌ای بر الگوریتم.

## درباره‌ی این کتاب

این کتاب به گونه‌ای طراحی شده که آسان‌خوان باشد. از جهش‌های فکری بزرگ اجتناب کرده‌ام. هر زمان که مفهوم جدیدی مطرح می‌شود، آن را توضیح می‌دهم یا به شما می‌گویم که چه زمانی آن را توضیح خواهیم داد. مفاهیم اصلی با تمرین‌ها و توضیحات متعدد تقویت می‌شوند تا بتوانید مفروضات خود را بررسی کنید و مطمئن شوید که موضوعات را متوجه شده‌اید.

هر موضوع را با مثال شرح می‌دهم. به جای نامفهوم‌نویسی هدف من این است که برای شما تجسم این مفاهیم را آسان کنم. از طرفی فکر می‌کنم با یادآوری آن چه از قبل می‌دانیم، یادگیری بهتری خواهیم داشت، و مثال‌ها یادآوری را آسان‌تر می‌کنند. بنابراین وقتی سعی می‌کنید تفاوت بین آرایه‌ها و لیست‌های پیوندی را به خاطر بیاورید (فصل ۲)، می‌توانید صرفاً به نشستن افراد در یک سالن برای تماشای فیلم فکر کنید. هرچند شاید بدیهی به نظر برسد، خودم مطالب را به شکل تصویری بهتر یاد می‌گیرم و این کتاب هم سرشار از تصاویر است.

مطالب کتاب به دقت تنظیم شده است. نیازی به نوشتن کتابی نیست که همه‌ی الگوریتم‌های مرتب‌سازی را پوشش بدهد - به همین دلیل است که ما ویکی‌پدیا و خان‌آکادمی را داریم. تمام الگوریتم‌هایی که در این کتاب گنجانده‌ام کاربردی هستند و من از آن‌ها در شغلم به عنوان یک مهندس نرم‌افزار استفاده می‌کنم. این‌ها پایه‌ی خوبی برای موضوعات پیچیده‌تر فراهم می‌کنند.

به سلامتی خواندن!

## نقشه‌ی راه

در سه فصل اول به بنیان الگوریتم می‌پردازیم:

• **فصل ۱-** اولین الگوریتم عملی خود را خواهید آموخت: جست‌وجوی دودویی<sup>۱</sup>. همچنین یاد می‌گیرید که سرعت یک الگوریتم را با استفاده از نماد  $O$  بزرگ<sup>۲</sup> تجزیه و تحلیل کنید. نماد  $O$  بزرگ در سراسر کتاب برای تجزیه و تحلیل سرعت یک الگوریتم استفاده شده است.

• **فصل ۲-** با دو ساختمان داده‌ی اساسی آشنا خواهید شد: آرایه و لیست پیوندی. این ساختمان داده‌ها در سراسر کتاب مورد استفاده قرار می‌گیرند، و از آنها برای ساختن ساختمان داده‌های پیشرفته‌تر مانند جدول‌های هَش<sup>۳</sup> استفاده می‌شود (فصل ۵).

• **فصل ۳-** در مورد بازگشت<sup>۴</sup>، یک تکنیک مفید که توسط بسیاری از الگوریتم‌ها استفاده می‌شود (مانند مرتب‌سازی سریع<sup>۵</sup>، که در فصل ۴ پوشش داده شده است) می‌آموزید.

به تجربه‌ی من، نماد  $O$  بزرگ و تابع‌های بازگشتی موضوعاتی چالش‌برانگیز برای مبتدیان است. بنابراین در این فصل با سرعتی آهسته‌تر زمان بیشتری را به این بخش‌ها اختصاص داده‌ام.

در بخش‌های دیگر کتاب به بررسی الگوریتم‌هایی با کاربردهای گسترده می‌پردازیم: در فصل‌های ۴، ۸، و ۹ **تکنیک‌های حل مسئله** - پوشش داده شده است. اگر به مشکلی برخورد کردید و مطمئن نبودید که چگونه آن را به طور موثر حل کنید، تقسیم و حل<sup>۶</sup> (فصل ۴) یا برنامه نویسی پویا<sup>۷</sup> (فصل ۹) را امتحان کنید. یا ممکن است متوجه بشوید که هیچ راه حل کارآمدی وجود ندارد و به جای آن با استفاده از الگوریتم حریصانه<sup>۸</sup> به یک پاسخ تقریبی دست پیدا کنید (فصل ۸).

• **جدول هَش** در فصل ۵ پوشش داده شده است. جدول هَش ساختمان داده‌ای

1. Binary search
2. Big O notation
3. Hash tables
4. recursion
5. quicksort
6. Divide and conquere
7. Dynamique programming
8. Greedy algorithm

بسیار مفید است. این جدول‌ها شامل مجموعه‌ای از جفت‌های کلید<sup>۱</sup> و مقدار<sup>۲</sup> است. مانند نام شخص و آدرس ایمیل او، یا نام کاربری و رمز عبور آن. جدول‌های هش واقعاً مفید هستند. وقتی می‌خواهم مسئله‌ای را حل کنم، دو استراتژی که در شروع در نظر می‌گیرم عبارتند از: «آیا می‌توانم از جدول هش استفاده کنم؟» و «آیا می‌توانم این را به عنوان یک گراف<sup>۳</sup> مدل‌سازی کنم؟»

• **الگوریتم گراف** در فصل‌های ۶ و ۷ پوشش داده شده است. گراف راهی برای مدل‌سازی یک شبکه است: یک شبکه‌ی اجتماعی، یا شبکه‌ای از مسیرها، یا نورون‌ها، یا هر مجموعه‌ی دیگر از ارتباط‌ها. جست‌وجوی سطح اول<sup>۴</sup> (فصل ۶) و الگوریتم دایجسترا<sup>۵</sup> (فصل ۷) راه‌هایی برای یافتن کوتاه‌ترین فاصله بین دو نقطه در یک شبکه هستند: می‌توانید از این روش برای محاسبه‌ی درجات جدایی بین دو نفر یا کوتاه‌ترین مسیر به مقصد استفاده کنید.

• **الگوریتم K-نزدیکترین همسایه<sup>۶</sup> (KNN)** - این یک الگوریتم ساده برای استفاده در یادگیری ماشینی است و در فصل ۱۰ به آن پرداخته می‌شود. می‌توانید از KNN برای ایجاد یک سیستم پیشنهادات، یک موتور OCR، سیستم پیش‌بینی ارزش سهام و به طور کلی هر چیزی که شامل پیش‌بینی یک مقدار باشد، استفاده کنید («ما فکر می‌کنیم آدیت به این فیلم ۴ ستاره می‌دهد») یا می‌توانید به کمک آن یک شیء را طبقه‌بندی کنید. («این حرف یک Q است.»).

• **گام‌های بعدی** در فصل ۱۱ به ده الگوریتم دیگر پرداخته می‌شود که برای مطالعه‌ی بیشتر مناسب هستند.

## شیوه‌ی خواندن این کتاب

ترتیب و محتوای این کتاب با دقت طراحی شده است. اگر به موضوع خاصی علاقه‌مند هستید، با خیال راحت به آن بخش بروید. در غیر این صورت، فصل‌ها را به ترتیب بخوانید. هر فصل زیربنای فصل بعدی است.

1. key
2. value
3. graph
4. Breadth-first search
5. dijkstra algorithm
6. k-nearest neighbors algorithm

اکیداً توصیه می‌کنم کد مثال‌ها را خودتان اجرا کنید. در این مورد هر چه بگویم کم گفته‌ام. نمونه کدها را کلمه به کلمه تایپ کنید (یا آنها را از [www.manning.com/books/grok-king-algorithms](http://www.manning.com/books/grok-king-algorithms) یا [https://github.com/egonschiele/grokking\\_algorithms](https://github.com/egonschiele/grokking_algorithms) دانلود کنید)، و آن‌ها را اجرا کنید. با پیروی از این شیوه چیزهای بیشتری را به خاطر خواهید سپرد.

انجام تمرین‌های این کتاب را هم توصیه می‌کنم. تمرینات کوتاه هستند - حل آن‌ها معمولاً فقط یک یا دو دقیقه، گاهی اوقات ۵ تا ۱۰ دقیقه زمان می‌برند. این تمرین‌ها به شما کمک می‌کنند تا از شرایط یادگیری خود مطلع بشوید، بنابراین قبل از اینکه دیر شود متوجه می‌شوید که از مسیر خارج شده‌اید یا خیر.

این کتاب برای چه کسانی مناسب است

این کتاب برای کسانی است که اصول کدنویسی را می‌دانند و می‌خواهند الگوریتم‌ها را درک کنند. شاید قبلاً در کدنویسی به مشکل برخورد کرده باشید و در تلاش برای یافتن یک راه حل الگوریتمی باشید. یا شاید می‌خواهید بدانید که الگوریتم‌ها به چه کاری می‌آیند. در اینجا یک لیست کوتاه و ناقص از افرادی است که احتمالاً از این کتاب بهره می‌برند:

- کدنویسان تفریحی
- دانش‌آموزان دوره‌های کدنویسی
- فارغ‌التحصیلان علوم کامپیوتر که به دنبال بازآموزی هستند
- فارغ‌التحصیلان فیزیک، ریاضی و دیگر رشته‌ها که به برنامه‌نویسی علاقه‌مند هستند

## قراردادهای کدنویسی و آدرس فایل‌ها برای دانلود

تمام نمونه کدهای این کتاب از Python 2.7 استفاده می‌کنند. تمام کدهای موجود در کتاب با فونت با عرض ثابت مانند این (fixed-width font like this) ارائه شده است تا از متن معمولی قابل تفکیک باشد. حاشیه‌نویسی کد برخی از لیست‌ها را همراهی می‌کند و مفاهیم مهم را برجسته می‌کند.

می‌توانید کد مثال‌های کتاب را از وبسایت ناشر به نشانی [www.manning.com/books/grok-king-algorithms](http://www.manning.com/books/grok-king-algorithms) یا از [https://github.com/egonschiele/grokking\\_algorithms](https://github.com/egonschiele/grokking_algorithms) دانلود کنید.

تنها زمانی بیشترین میزان یادگیری را تجربه می‌کنید که واقعاً از یادگیری لذت ببرید

پس لذت ببرید و نمونه کدها را اجرا کنید!

## درباره‌ی نویسنده

آدیتیا بهارگوا<sup>۱</sup> مهندس نرم‌افزار در Etsy، (بازار آنلاین برای کالاهای دست‌ساز) است. او دارای مدرک کارشناسی ارشد در رشته‌ی علوم کامپیوتر از دانشگاه شیکاگو است. او همچنین یک وبلاگ فناوری مصور محبوب و پربازدید به آدرس [adit.io](http://adit.io) دارد.





## یک | درآمدی بر الگوریتم



### در این فصل

- مبانی و اساس مباحث مطرح شده در کتاب را یاد می‌گیرید.
- اولین الگوریتم جست و جوی خودرामी نویسد (جست و جوی دودویی).
- یاد می‌گیرید چگونه درباره‌ی زمان اجرای یک الگوریتم صحبت کنید (علامت O بزرگ).
- یک تکنیک رایج برای طراحی الگوریتم به شما معرفی می‌شود (تابع بازگشتی).

### مقدمه

الگوریتم مجموعه‌ای از دستورالعمل‌ها برای انجام یک کار است. هرچند هر قطعه کد را می‌توان یک الگوریتم نامید، با این حال در این کتاب به موارد جالب‌تر پرداخته می‌شود. الگوریتم‌هایی را در این کتاب آورده‌ام که عملکرد سریعی دارند یا به واسطه‌ی آن‌ها مسائل قابل توجهی را می‌توان حل کرد یا هر دو این ویژگی‌ها را شامل می‌شوند. در ادامه مباحث مطرح شده در این کتاب را مرور می‌کنیم:

• در فصل ۱ در مورد جست و جوی دودویی صحبت می‌شود و می‌بینیم که چگونه یک الگوریتم می‌تواند سرعت کد شما را افزایش بدهد. در یک مثال، تعداد مراحل مورد نیاز از ۴ میلیارد مرحله به ۳۲ مرحله کاهش می‌یابد!

یک دستگاه GPS از الگوریتم گراف (همان‌طور که در فصل‌های ۶ و ۷ یاد می‌گیرید) برای محاسبه‌ی کوتاه‌ترین مسیر به مقصد استفاده می‌کند.

• می‌توانید از برنامه‌نویسی پویا (در فصل ۹ به آن پرداخته شده است) برای نوشتن یک الگوریتم هوش مصنوعی برای بازی چکرز استفاده کنید.

در هر مورد، الگوریتم را شرح می‌دهم و مثالی برای شما می‌زنم. سپس در مورد زمان اجرای الگوریتم در نماد O بزرگ صحبت می‌کنم. در انتها، به بررسی مسئله‌های دیگری که می‌توان با همان الگوریتم حل کرد، می‌پردازم.

## آن چه درباره‌ی کارایی<sup>۲</sup> خواهید آموخت

خبر خوب این که احتمالاً پیش از این نسخه‌ی پیاده‌سازی شده‌ی هر الگوریتمی که در این کتاب آمده، در زبان برنامه‌نویسی مورد علاقه‌ی شما در دسترس باشد، در نتیجه نیاز نخواهد شد خود شما کد هر الگوریتم را بنویسید! اما اگر این کدها درک نشوند، پیاده‌سازی آن‌ها بیهوده خواهد بود. در این کتاب، می‌آموزید الگوریتم‌های مختلف را مقایسه کنید: باید از مرتب‌سازی ادغامی<sup>۳</sup> استفاده کرد یا مرتب‌سازی سریع<sup>۴</sup>؟ از آرایه<sup>۵</sup> یا لیست<sup>۶</sup>؟ استفاده از یک ساختمان داده‌ی<sup>۷</sup> متفاوت می‌تواند تفاوت اساسی ایجاد کند.

## آن چه در مورد حل مسئله یاد می‌گیرید

شما تکنیک‌هایی را برای حل مسئله‌هایی که ممکن است تاکنون از درک شما خارج بوده‌اند، یاد می‌گیرید. مثلاً:

1. checkers
2. Performance
3. merge sort
4. quicksort
5. array
6. list
7. Data structure

- اگر به ساخت بازی‌های ویدیویی علاقه دارید، می‌توانید یک سیستم هوش مصنوعی بنویسید که حرکات بازیکن را به کمک الگوریتم گراف دنبال بکند.
  - می‌آموزید که با استفاده از  $K$  نزدیک‌ترین همسایه یک سیستم پیشنهاد بسازید.
  - برخی از مسائل قابل حل نیستند! بخشی از این کتاب که در مورد مسائل NP کامل صحبت می‌کند و به شما نشان می‌دهد که چگونه آن مسئله‌ها را شناسایی کنید و الگوریتمی ارائه بدهید که به پاسخ تقریبی برسید.
- در پایان این کتاب، با برخی از کاربردی‌ترین الگوریتم‌ها آشنا می‌شوید. سپس می‌توانید از دانش جدید خود برای یادگیری الگوریتم‌های خاص‌تر برای هوش مصنوعی، پایگاه‌های داده و غیره استفاده کنید. یا می‌توانید با چالش‌های بزرگ‌تری در محل کار روبرو بشوید.

### پیش‌نیاز

پیش از شروع این کتاب دانش مقدماتی از جبر مورد نیاز است. در این حد که در تابع  $f(x) = x \times 2$ ، پاسخ  $f(5)$  را بدانید. اگر جواب شما عدد ۱۰ است، از دانش کافی ریاضی برخوردارید.

به‌علاوه، دنبال‌کردن این فصل (و این کتاب) اگر با زبان برنامه‌نویسی آشنا باشید آسان‌تر خواهد بود. تمامی مثال‌های این کتاب به زبان پایتون نوشته شده‌اند. اگر با هیچ زبان برنامه‌نویسی آشنایی ندارید و می‌خواهید یاد بگیرید، پایتون را انتخاب کنید - برای تازه‌کارها حرف ندارد. اگر با زبان دیگری مثل روبی هم آشنایی دارید، باز مشکلی نخواهید داشت.

## جست وجوی دودویی



فرض کنید در یک دفترچه تلفن دنبال نام فردی می‌گردید (چه کار از مُدافتاده‌ای) اسم آن شخص با حرف K شروع می‌شود. می‌توانید از اول شروع کرده و صفحات را ورق بزنید تا به اسم‌هایی برسید که با حرف k شروع می‌شوند. با این حال احتمال این‌که از صفحه‌ای از وسط دفترچه شروع به جست‌وجو کنید بیشتر است، چون می‌دانید که Kها جایی در همان میانه‌ی دفترچه‌ی تلفن هستند.

یا فرض کنید به دنبال کلمه‌ای در دیکشنری می‌گردید که با حرف O شروع شده است. این بار هم، از صفحات وسط شروع می‌کنید.

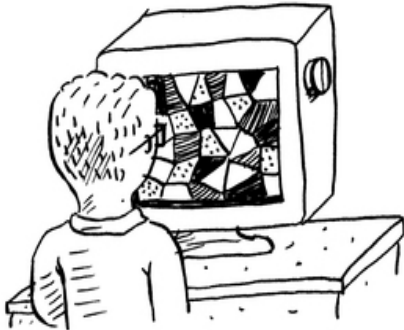
حالا فرض کنید که وارد فیس‌بوک می‌شوید.

فیس‌بوک باید تأیید کند که در این سایت حساب کاربری دارید. پس، باید در پایگاه داده‌ی خود دنبال نام شما بگردد. فرض کنید که نام کاربری شما Karmageddon است.

فیس‌بوک می‌تواند از حرف A شروع کرده و به دنبال نام شما بگردد- با این حال برایش منطقی‌تر است که از جایی در میانه‌ی فهرست شروع کند.

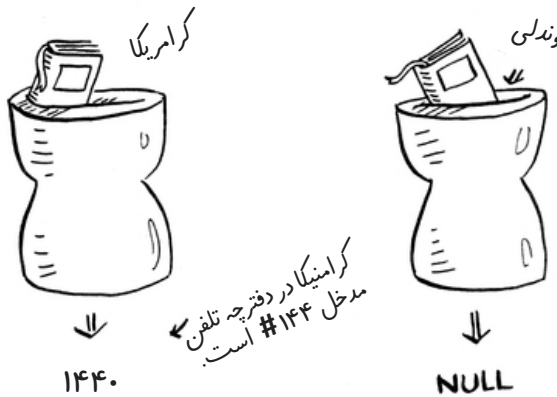
این یک مسئله‌ی جست‌وجو است. و در تمامی این موارد از الگوریتم یکسانی برای حل مسئله استفاده می‌شود.

جست‌وجوی دودویی الگوریتمی است



که ورودی آن یک لیست مرتب شده از عناصر است (بعدها توضیح خواهیم داد که چرا نیاز است تا این لیست مرتب بشود). اگر عنصری که به دنبال آن می‌گردید در آن لیست باشد، جست‌وجوی دودویی موقعیت مورد نظر را بر می‌گرداند. در غیر این صورت جواب جست‌وجوی دودویی null خواهد بود.

به عنوان مثال:



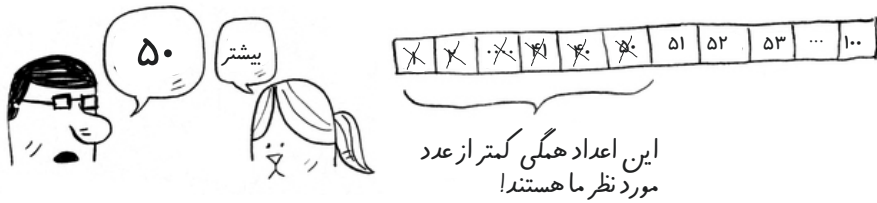
به دنبال شماره تلفن شرکت‌ها در یک دفترچه تلفن از طریق جست‌وجوی دودویی

در اینجا مثالی داریم از این‌که جست‌وجوی دودویی چگونه کار می‌کند. عددی بین ۱ تا ۱۰۰ را انتخاب می‌کنم.



باید بتوانید با کمترین دفعات ممکن عدد مورد نظر را حدس بزنید. در هر نوبت، به شما خواهیم گفت که آیا حدس شما کمتر یا بیشتر از عدد مورد نظر است، یا اینکه عدد را درست حدس زده‌اید.





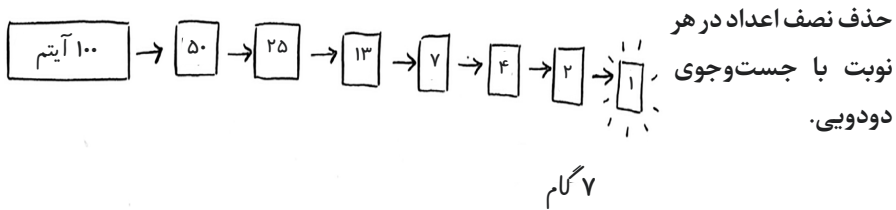
کوچک‌تر از عدد مورد نظر است، با این حال نصف اعداد حذف شدند! حالا می‌دانید که اعداد ۱-۵۰ همگی کمتر هستند. حدس بعدی: ۷۵.



بزرگ‌تر است، اما باز هم نصف اعداد باقی‌مانده را حذف کرده‌اید! با جست‌وجوی دودویی، عدد وسطی را حدس می‌زنید و نصف اعداد باقی‌مانده را در هر نوبت حذف می‌کنید. عدد بعدی ۶۳ است (عدد میانی ۵۰ و ۷۵).



همین الان اولین الگوریتم را یاد گرفتید. به این راه حل جست و جوی دودویی گفته می‌شود. در اینجا تعداد اعدادی که در هر نوبت می‌توان حذف کرد به نمایش در آمده است:

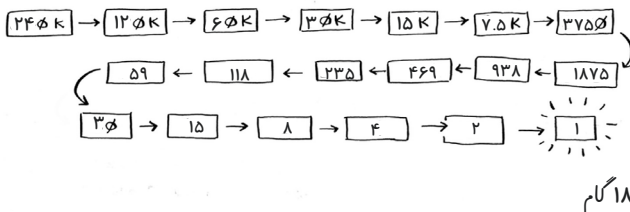


به هر عددی می‌توان با حداکثر ۷ حدس دست یافت - چون با هر حدس اعداد بسیاری را حذف می‌کنید!

فرض کنید که به دنبال یک کلمه در دیکشنری می‌گردید. این دیکشنری از ۲۴۰۰۰۰ کلمه تشکیل شده است. در بدترین وضعیت، فکر می‌کنید چند گام برای هر جست و جو مورد نیاز است؟

گام \_\_\_\_\_ : جست و جوی ساده  
 گام \_\_\_\_\_ : جست و جوی دودویی

جست و جوی ساده در شرایطی که کلمه‌ی مورد نظر آخرین کلمه‌ی دیکشنری باشد ۲۴۰۰۰۰ مرحله خواهد داشت. در حالی که در هر مرحله از جست و جوی دودویی، تعداد کلمات را به نصف کاهش می‌دهید تا تنها یک کلمه باقی بماند.



پس جست و جوی دودویی ۱۸ مرحله طول می‌کشد- تفاوت قابل توجه!



به طور معمول، برای هر فهرستی از  $n$ ، جست و جوی دودویی  $\log_2 n$  مرحله برای بدترین وضعیت نیاز دارد، در حالی که برای جست و جوی ساده  $n$  مرحله مورد نیاز است.

## لگاریتم

شاید مفهوم لگاریتم را فراموش کرده باشید، اما احتمالاً می دانید که نمایی (توان) چیست.  $\text{LOG}_{10} 100$  مثل این است که بپرسیم، «چندتا ده تا باید در هم ضرب کنیم تا به عدد ۱۰۰ برسیم؟» جواب ۲ است:  $10 \times 10 = 2$ . بنابراین  $\text{LOG}_{10} 100 = 2$ . در واقع لگاریتم برعکس توان است.

$$\begin{array}{l} 10^2 = 100 \leftrightarrow \text{Log}_2 100 = 2 \\ 10^3 = 1000 \leftrightarrow \text{Log}_2 1000 = 3 \\ \hline 2^3 = 8 \leftrightarrow \text{Log}_2 8 = 3 \\ \hline 2^4 = 16 \leftrightarrow \text{Log}_2 16 = 4 \\ \hline 2^5 = 32 \leftrightarrow \text{Log}_2 32 = 5 \end{array}$$

لگاریتم برعکس توان است.

در این کتاب هرگاه از زمان اجرا با علامت  $O$  بزرگ صحبت می کنم (کمی بعد توضیح می دهم)،  $\log$  همیشه به معنای  $\log_2$  است. وقتی که از راه جست و جوی ساده به دنبال عضوی می گردید، در بدترین حالت باید به هر عضو نگاه کنید. پس برای یک فهرست ۸ عددی، حداکثر باید ۸ عدد را چک کنید. برای جست و جوی دودویی، باید در بدترین حالت،  $\log n$  عضورا چک کنید. برای یک فهرست ۸ عضوری،  $\log 8 = 3$ ، چون  $2^3 = 8$ . پس برای یک فهرست ۸ عضوی، حداکثر سه عدد را باید چک کنید. برای لیستی با ۱۰۲۴ عضو،  $\log 1024 = 10$  مورد، چون  $2^{10} = 1024$ . پس برای یک فهرست ۱۰۲۴ عددی، باید در بدترین حالت ۱۰ عدد را چک کنید.

**نکته**

در این کتاب به دفعات درباره‌ی  $\log$  time یا زمان لگاریتمی صحبت می‌کنم، پس باید مفهوم لگاریتم را بفهمید. اگر اطلاعات کافی درباره‌ی آن ندارید، خان آکادمی ([khanacademy.org](http://khanacademy.org)) ویدئوی خوبی در مورد لگاریتم دارد که درک این مبحث را ممکن می‌کند.

**نکته**

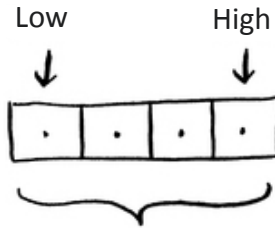
تنها زمانی می‌توان از جست‌وجوی دودویی استفاده کرد که لیست شما مرتب‌شده باشد. برای مثال، اسامی دفتر تلفن به ترتیب حروف الفبا ردیف شده است، پس می‌توانید از جست‌وجوی دودویی برای پیدا کردن یک نام استفاده کنید. اگر اسامی مرتب نشده بودند چه اتفاقی می‌افتاد؟

بیا بیا با هم نگاهی بیندازیم به اینکه چگونه باید جست‌وجوی دودویی را به زبان پایتون نوشت. در این کد نمونه از آرایه استفاده شده است. اگر نمی‌دانید که آرایه‌ها به چه شکل عمل می‌کنند، نگران نباشید؛ در فصل بعدی به آن می‌پردازیم. فقط باید بدانید که می‌توانید سکانسی از اعضا را در یک ردیف از باکت یا سطل‌های متوالی که آرایه نامیده می‌شود ذخیره کنید. باکت‌ها از شماره‌ی صفر شماره‌گذاری شده‌اند: باکت اول در موقعیت #۰ و باکت دوم #۱ و باکت سوم #۲ است و به همین ترتیب. تابع `binary_search` به عنوان ورودی یک آرایه‌ی مرتب‌شده و یک آیتیم می‌گیرد. اگر آیتیم در آرایه بود، تابع موقعیت آن را برمی‌گرداند. بخشی از آرایه را که در آن جست‌وجو

انجام می شود را زیر نظر می گیریم. در گام نخست، آرایه به این شکل است:

```
low = 0
```

```
high = len(list) - 1
```



اعدادی که در میان آن ها جست و جو می کنیم.

در هر نوبت، عضو میانی را چک می کنید:

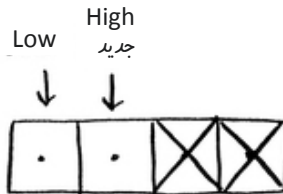
```
mid = (low + high) // 2
```

اگر حاصل (low + high) زوج نباشد، پایتون ..... ←  
mid را کاهش می کند.

```
guess = list[mid]
```

اگر حدس مورد نظر کم تر از عدد مورد نظر ما باشد، مطابق زیر مقدار low را تغییر می دهید:

```
if guess < item:  
    low = mid + 1
```



و اگر حدس مورد نظر بیشتر باشد، high را تغییر می دهیم. کد کامل به این شکل است:

```

def binary_search(list, item):
    low = 0
    high = len(list) - 1

    while low <= high:
        mid = (low + high) / 2
        guess = list[mid]

        if guess == item:
            return mid

        if guess > item:
            high = mid - 1

        else:
            low = mid + 1

    return None

my_list = [1, 3, 5, 7, 9]

print binary_search(my_list, 3) # => 1
print binary_search(my_list, -1) # => None

```

high, low محدوده‌ی جست‌وجوی شمارا مشخص می‌کند.

وقتی دامنه را به یک عنصر محدود نکرده باشید...

عنصر میانی را بررسی می‌کند

عدد مورد نظر پیدا شد!

از عدد مورد نظر بزرگ‌تر است.

از عدد مورد نظر کوچک‌تر است.

آیتم وجود ندارد.

بیا باید که بالا را تست کنیم!

به یاد داشته باشید، لیست‌ها از ایندکس صفر شروع می‌شوند. دومین اسلات ایندکس یک دارد.

None در پایتون به معنای هیچ است. در این حالت آیتم مورد نظر پیدا نشده است.

## تمرین

۱.۱ فرض کنید یک لیست مرتب شده از ۱۲۸ نام دارید، و با جست‌وجوی دودویی در میان آن لیست به جست‌وجو می‌پردازید. حداکثر تعداد مراحل چه مقدار خواهد بود؟

۱.۲ تعداد اعضای لیست را دو برابر مقدار قبلی در نظر بگیرید. این بار حداکثر تعداد مراحل چقدر خواهد بود؟

## زمان اجرا



هرگاه از الگوریتم صحبت می‌کنم، به زمان اجرای آن هم اشاره می‌کنم. به طور معمول می‌خواهید کارآمدترین الگوریتم را برای بهینه‌سازی زمان یا فضا انتخاب کنید.

به جست‌وجوی دودویی برگردیم. زمان صرفه‌جویی شده با استفاده از این الگوریتم به چه میزان است؟ خب، اولین رویکرد این بود که هر عدد را یک‌به‌یک بررسی کنیم. اگر فهرستی از ۱۰۰ عدد باشد تا ۱۰۰ حدس نیاز دارد. اگر فهرستی از ۴ میلیارد عدد باشد، تا ۴ میلیارد

حدس نیاز دارد. بنابراین حداکثر تعداد حدس‌ها برابر با اندازه‌ی لیست است. به چنین حالتی زمان خطی می‌گویند.

زمان اجرا برای جست‌وجوی دودویی متفاوت است. اگر لیست ۱۰۰ داشته باشد، حداکثر ۷ حدس نیاز است. اگر لیست ۴ میلیارد آیتم داشته باشد، حداکثر ۳۲ دفعات ۳۲ مرتبه می‌شود. عجب الگوریتم کارا و قدرتمندی، مگر نه؟ جست‌وجوی دودویی در زمان لگاریتمی (یا به قول بومی‌ها لگ تایم) اجرا می‌شود. در جدول زیر خلاصه‌ای از یافته‌های امروز ما آمده است.

جست‌وجوی ساده	جست‌وجوی دودویی	
۱۰۰ آیتم	۱۰۰ آیتم	بزرگ کاهش یافته
↓	↓	
۱۰۰ حدس	۷ حدس	
<hr/>		
۴۰۰۰۰۰۰ آیتم	۴۰۰۰۰۰۰ آیتم	بزرگ کاهش یافته
↓	↓	
۴۰۰۰۰۰۰ حدس	۳۲ حدس	
<hr/>		
$O(n)$	$O(\log n)$	
↑ زمان خطی	↑ زمان لگاریتمی	

زمان اجرای الگوریتم‌های جست‌وجو

## نماد O بزرگ



O بزرگ یک نماد ویژه است که به شما می‌گوید الگوریتم مورد نظر تا چه اندازه سریع است. چه اهمیتی دارد؟ خوب، شما اغلب از الگوریتم‌هایی که دیگران نوشته‌اند استفاده می‌کنید - و وقتی این کار را انجام می‌دهید، خوب است که بدانید این الگوریتم‌ها چقدر سریع یا کند هستند. در این بخش توضیح می‌دهم نماد O بزرگ چیست و فهرستی از متداول‌ترین زمان‌های اجرای الگوریتم‌ها را به شما ارائه می‌دهم.

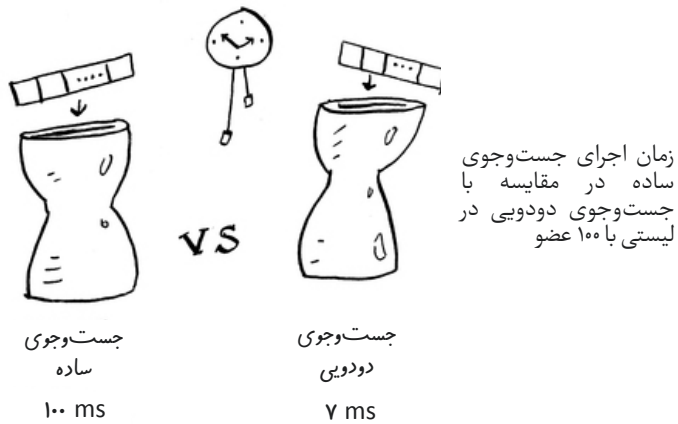
### زمان اجرای الگوریتم‌ها با سرعت‌های متفاوتی رشد می‌کند

باب در حال نوشتن یک الگوریتم جست‌وجو برای ناسا است. الگوریتم او در آستانه‌ی فرود موشک بر روی ماه آغاز می‌شود و به محاسبه‌ی محل فرود کمک می‌کند. این نمونه‌ای است که نشان می‌دهد چگونه زمان اجرای دو الگوریتم می‌تواند با نسبت متفاوت رشد کند.

باب در تلاش است میان جست‌وجوی ساده و جست‌وجوی دودویی تصمیم بگیرد. الگوریتم باید هم‌زمان سریع و صحیح باشد. از یک طرف، جست‌وجوی دودویی سریع‌تر است و باب فقط ۱۰ ثانیه فرصت دارد تا بفهمد کجا باید فرود بیاید - در غیر این صورت، موشک از مسیر خود خارج خواهد شد. از سوی دیگر، نوشتن جست‌وجوی ساده آسان‌تر است و احتمال بروز مشکلات کمتر است و باب واقعاً نمی‌خواهد باگ‌هایی در کد برای فرود موشک ایجاد شود! برای دقت بیشتر، باب تصمیم می‌گیرد هر دو الگوریتم را با لیستی از ۱۰۰ عنصر زمان‌بندی کند.

بیایید فرض کنیم برای بررسی یک عنصر ۱ میلی‌ثانیه طول می‌کشد. با جست‌وجوی ساده، باب باید ۱۰۰ عنصر را بررسی کند، بنابراین جست‌وجو ۱۰۰ میلی‌ثانیه طول می‌کشد تا اجرا شود. از طرف دیگر، او فقط باید ۷ عنصر را با جست‌وجوی دودویی بررسی کند ( $\log_2 100$  تقریباً برابر عدد ۷ است)، بنابراین جست‌وجو ۷ میلی‌ثانیه طول می‌کشد تا

اجرا شود. اما به طور واقع بینانه، این لیست بیش از یک میلیارد عنصر خواهد داشت. اگر این طور باشد، جست و جوی ساده چقدر طول می کشد؟ جست و جوی دودویی چقدر؟ قبل از خواندن ادامه ی مطلب، مطمئن شوید که برای هر یک از پرسش ها پاسخی دارید.



باب جست و جوی دودویی را با ۱ میلیارد عنصر اجرا می کند و ۳۰ میلی ثانیه طول می کشد ( $\log_2 1,000,000,000$  حدوداً برابر است با ۳۰). با خودش فکر می کند « ۳۲ Ms تقریباً ۳۰ است. جست و جوی دودویی حدود ۱۵ برابر سریع تر از جست و جوی ساده است، زیرا جست و جوی ساده ۱۰۰ میلی ثانیه با ۱۰۰ عنصر و جست و جوی دودویی ۷ میلی ثانیه طول کشید. بنابراین جست و جوی ساده  $30 \times 15 = 450$  میلی ثانیه طول می کشد، درست است؟ بسیار کم تر از آستانه ی ۱۰ ثانیه ای.» باب تصمیم می گیرد با یک جست و جوی ساده به آن ادامه بدهد. این انتخاب درستی است؟

نه. معلوم شد، باب اشتباه می کند. آن هم چه جور. زمان اجرای جست و جوی ساده با ۱ میلیارد آیت ۱ میلیارد میلی ثانیه خواهد بود که ۱۱ روز می شود! مشکل این است که زمان اجرا برای جست و جوی دودویی و جست و جوی ساده با سرعت یکسانی رشد نمی کند.

جست و جوی ساده	جست و جوی دودویی	
۱۰۰ عنصر ۱۰۰ ms	۷ ms	زمان اجرا با سرعت‌های بسیار متفاوتی رشد می‌کند!
۱۰۰۰۰ عنصر ۱۰ ثانیه	۱۴ ms	
۱۰۰۰۰۰۰۰۰۰ عنصر ۱۱ روز	۳۲ ms	

به عبارتی با افزایش تعداد آیت‌ها، جست و جوی دودویی برای اجرا اندکی زمان بیشتر برای اجرا نیاز دارد. اما جست و جوی ساده زمان بسیار بیشتری را برای اجرا نیاز دارد. بنابراین با بزرگ‌تر شدن فهرست اعداد، جست و جوی دودویی ناگهان بسیار سریع‌تر از جست و جوی ساده می‌شود. باب گمان می‌کرد جست و جوی دودویی ۱۵ برابر سریع‌تر از جست و جوی ساده است، اما چنین تصویری درست نیست. اگر لیست دارای ۱ میلیارد آیت‌ها باشد، ۳۳ میلیون بار سریع‌تر است. به همین دلیل است که دانستن مدت زمان اجرای یک الگوریتم کافی نیست - باید بدانید با افزایش اندازه‌ی لیست زمان اجرا چگونه افزایش می‌یابد. اینجاست که نماد  $O$  بزرگ وارد می‌شود.

نماد  $O$  بزرگ به شما می‌گوید که یک الگوریتم چقدر سریع است. برای مثال، فرض کنید فهرستی با اندازه‌ی  $n$  دارید. جست و جوی ساده باید هر عنصر را بررسی کند، بنابراین  $n$  عملیات طول خواهد کشید. زمان اجرا در نماد  $O$  بزرگ  $O(n)$  است. واحد زمان و ثانیه‌ی آن چه شد؟ خبری ازش نیست -  $O$  بزرگ سرعت را بر حسب ثانیه به شما نمی‌گوید. نماد  $O$  بزرگ به شما امکان می‌دهد تعداد عملیات را مقایسه کنید. به شما می‌گوید که الگوریتم با چه سرعتی رشد می‌کند.





در اینجا مثال دیگری داریم. جست و جوی دودویی برای بررسی لیستی با اندازه  $n$  به  $\log n$  عملیات نیاز دارد. زمان اجرا در نماد  $O$  بزرگ چقدر است؟  $O(\log n)$ . به طور کلی نماد  $O$  بزرگ به شکل زیر نوشته می شود.

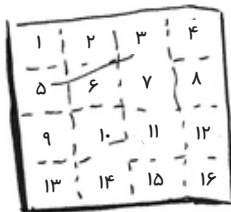
شکل نماد  $O$  بزرگ

تعداد عملیات  $O(n)$  "  $O$  بزرگ "

این نماد  $O$  بزرگ نامیده می شود زیرا شما یک «  $O$  بزرگ » را در جلوی تعداد عملیات قرار می دهید (به نظر شوخی می رسد، اما حقیقت دارد!). و تعداد عملیاتی را که یک الگوریتم انجام می دهد به شما می گوید. چند مثال را مرور کنیم. ببینید می توانید زمان اجرای این الگوریتم ها را مشخص کنید.

### به تصویر کشیدن چند زمان اجرا با $O$ بزرگ متفاوت

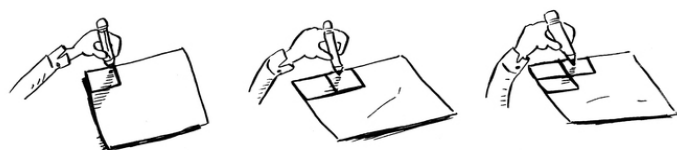
در اینجا یک مثال عملی داریم که می توانید در خانه با چند تکه کاغذ و یک مداد دنبال کنید. فرض کنید باید یک چهارخانه ی ۱۶ تایی بکشید.



### الگوریتم یک

یکی از راه های انجام این کار، کشیدن به نوبت ۱۶ چهارخانه، است. به یاد داشته باشید که نماد  $O$  بزرگ تعداد عملیات را می شمارد. در این مثال، رسم هر چهارخانه برابر با یک عملیات است. شما باید ۱۶ چهارخانه بکشید.

کدام الگوریتم برای کشیدن یک چهارخانه در هر بار چند عملیات طول می کشد؟ این جدول مناسب است؟



کشیدن یک  
چهارخانه در هر نوبت

برای کشیدن ۱۶ چهارخانه ۱۶ مرحله طول می‌کشد. زمان اجرای این الگوریتم چقدر است؟

### الگوریتم دو

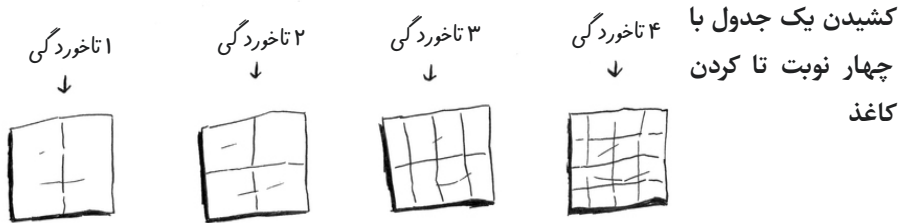
حالا این الگوریتم را امتحان کنید. کاغذ را از وسط تا کنید.



در این مثال، هر بار تا کردن کاغذ یک عملیات است. شما فقط با این عملیات دو چهارخانه درست کردید!  
کاغذ را دوباره و دوباره و دوباره تا کنید.



بعد از چهار بار تا زدن آن را باز کنید و یک جدول زیبا خواهید داشت! هر تا زدن تعداد چهارخانه‌ها را دو برابر می‌کند. شما با ۴ عملیات ۱۶ چهارخانه درست کردید!



شما می‌توانید با هر تا زدن تعداد چهارخانه را دو برابر کنید. در نتیجه می‌توانید ۱۶ چهارخانه را در ۴ مرحله رسم کنید. زمان اجرای این الگوریتم چقدر است؟ قبل از حرکت، زمان‌های اجرای هر دو الگوریتم را مشخص کنید.  
 پاسخ: الگوریتم اول زمان اجرای  $O(n)$  و الگوریتم دوم زمان اجرای  $O(\log n)$  دارد.

## O بزرگ برای بدترین زمان اجرا

فرض کنید از جست‌وجوی ساده برای جست‌وجوی شخصی در دفترچه تلفن استفاده می‌کنید. می‌دانید که جست‌وجوی ساده زمان اجرای  $O(n)$  دارد، به این معنی که در بدترین حالت، باید تک تک ورودی‌های دفترچه تلفن خود را بررسی کنید. در این مورد، شما به دنبال نام Adit هستید. این فرد اولین نام در دفترچه‌ی تلفن شما است. بنابراین لازم نبود تا تمام نام‌ها را مرور کنید - در اولین تلاش آن را پیدا کردید. آیا این الگوریتم زمان  $O(n)$  را صرف کرده است؟ یا به دلیل اینکه در اولین تلاش فرد مورد نظر را پیدا کردید، زمان  $O(1)$  طول کشید؟

جست‌وجوی ساده همچنان به زمان  $O(n)$  نیاز دارد. در این صورت، فوراً چیزی را که دنبالش می‌گشتید پیدا کردید. این بهترین سناریو است. اما نماد  $O$  بزرگ در مورد بدترین سناریو است. بنابراین می‌توانید بگویید که در بدترین حالت، باید هر نام وارد شده در دفترچه تلفن را یک بار نگاه کنید. زمان آن  $O(n)$  است. این برای آگاهی از بدترین حالت ممکن است - می‌دانید که جست‌وجوی ساده هرگز کندتر از زمان  $O(n)$  نخواهد بود.

**نکته**

همراه با بدترین زمان اجرا، توجه به زمان اجرای متوسط<sup>۱</sup> هم مهم است. در فصل ۴، بدترین حالت در مقابل حالت متوسط مورد بحث قرار گرفته است.

**برخی از زمان اجرای معمول O بزرگ**

در ادامه پنج زمان اجرای O بزرگ که به دفعات با آن‌ها مواجه می‌شوید، از سریع‌ترین به کندترین حالت فهرست شده است:

•  $O(\log n)$ ، همچنین به عنوان log time شناخته می‌شود. به عنوان مثال: جست‌وجوی دودویی.

•  $O(n)$ ، همچنین به عنوان زمان خطی شناخته می‌شود. به عنوان مثال: جست‌وجوی ساده

•  $O(n * \log n)$ ، یک الگوریتم مرتب‌سازی سریع، مانند مرتب‌سازی سریع (که در فصل ۴ آمده است).

•  $O(n^2)$ ، یک الگوریتم مرتب‌سازی آهسته، مانند مرتب‌سازی انتخابی (در فصل ۲ آمده است).

•  $O(n!)$ ، یک الگوریتم بسیار کند، مانند فروشنده‌ی دوره‌گرد (مورد بعدی که به آن پرداخته می‌شود!).

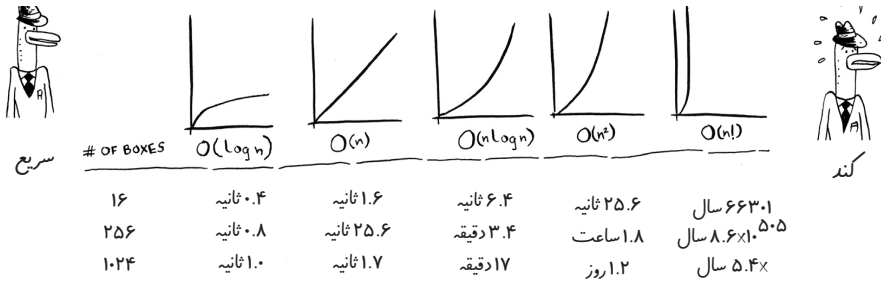
فرض کنید یک بار دیگر جدولی با ۱۶ چهارخانه ترسیم می‌کنید و می‌توانید از ۵ الگوریتم مختلف برای این کار استفاده کنید. در صورت انتخاب الگوریتم اول، برای ترسیم شبکه زمان  $O(\log n)$  طول می‌کشد و می‌توانید ۱۰ عملیات در ثانیه انجام بدهید.

با زمان  $O(\log n)$ ، ۴ عملیات طول می‌کشد تا یک جدول ۱۶ چهارخانه‌ای بکشید (۱۶ برابر ۴ است). بنابراین ۰.۴ ثانیه طول می‌کشد تا جدول را بکشید. اگر بخواهید

۱۰۲۴ چهارخانه رسم کنید چه؟  $\log 1024 = 10$  عملیات طول می‌کشد. یا ۱ ثانیه برای رسم جدولی از ۱۰۲۴ چهارخانه. این اعداد از الگوریتم اول استفاده می‌کنند.

الگوریتم دوم کندتر است: زمان  $O(n)$  طول می‌کشد. برای رسم ۱۶ چهارخانه ۱۶

عملیات و برای رسم ۱۰۲۴ چهارخانه ۱۰۲۴ عملیات طول می کشد. در واقع چند ثانیه؟  
در ادامه زمان لازم برای ترسیم کردن جدول از طریق دیگر الگوریتم ها، از سریع ترین به کندترین، آمده است:



زمان اجراهای دیگری نیز وجود دارد، اما این پنج مورد رایج ترین آن ها هستند. البته در واقعیت نمی توانید زمان اجرای  $O$  بزرگ را به این سراسستی به تعدادی عملیات تبدیل کنید، اما فعلا تا همین جا کافی است. پس از اینکه چند الگوریتم دیگر را یاد گرفتید، در فصل ۳ به نماد  $O$  بزرگ باز خواهیم گشت. نکات اصلی تا به این جا به شرح زیر است:

- سرعت الگوریتم نه بر اساس ثانیه بلکه بر مبنای رشد تعداد عملیات اندازه گیری می شود.
- در واقع، مقصود این است که با افزایش اندازه ی داده های ورودی زمان اجرای یک الگوریتم با چه سرعتی افزایش می یابد.
- زمان اجرای الگوریتم ها با نماد  $O$  بزرگ بیان می شود.
- $O(\log n)$  سریع تر از  $O(n)$  است، اما با رشد لیست مواردی که جست و جو می کنید، سرعت آن بسیار بیشتر می شود.

### تمرین

زمان اجرا برای هر یک از این سناریوها را بر حسب  $O$  بزرگ مشخص کنید.  
**۱.۳** نام یک نفر را در نظر دارید و می خواهید شماره تلفن آن شخص را در دفترچه تلفن پیدا کنید.

**۱.۴** شما یک شماره تلفن دارید و می خواهید نام شخص را در دفترچه تلفن پیدا کنید.  
 (نکته: باید تمام صفحات دفترچه را برگردید!)

**۱.۵** شما می خواهید شماره‌ی هر فرد نوشته شده در دفترچه تلفن را بخوانید.  
**۱.۶** شما می خواهید شماره‌ی افرادی را بخوانید که نام آن‌ها با حرف  $A$  شروع شده است. (این مسئله پیچیده است! مفاهیمی را دربرمی‌گیرد که در فصل ۴ بیشتر به آن پرداخته می‌شود. احتمالاً با دیدن پاسخ آن متعجب خواهید شد.)

### فروشنده‌ی دوره‌گرد

ممکن است با خواندن بخش آخر فکر کرده باشید، «امکان ندارد با الگوریتمی مواجه شوم که  $O(n!)$  زمان نیاز داشته باشد.» خوب، اجازه دهید تا به شما ثابت کنم که اشتباه می‌کنید! در ادامه نمونه‌ای از یک الگوریتم با زمان اجرای بسیار بد آمده است. این یک مسئله‌ی معروف در علوم کامپیوتر است، زیرا رشد آن وحشتناک است و بنا بر نظر افراد خبره و باهوش راهی برای بهبود زمان آن وجود ندارد به این مسئله فروشنده‌ی دوره‌گرد می‌گویند.

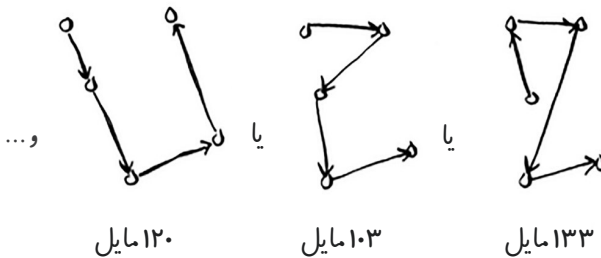


یک فروشنده را فرض کنید.

این فروشنده باید به پنج شهر سفر کند.



این فروشنده، که آپوس صدایش می‌کنم، می‌خواهد به هر پنج شهر برود با این شرط که حداقل مسافت را طی کند. یک روش این است: مشاهده‌ی تمامی حالت‌های ممکن که می‌توان به شهرها سفر کرد.



او کل مسافت را جمع می‌کند و سپس مسیری را که کمترین فاصله را دارد انتخاب می‌کند. ۱۲۰ جایگشت<sup>۱</sup> با ۵ شهر وجود دارد، بنابراین برای حل مسئله‌ی ۵ شهر، به ۱۲۰ عملیات نیاز است. برای ۶ شهر، ۷۲۰ عملیات انجام می‌شود (۷۲۰ جایگشت وجود دارد). برای ۷ شهر، ۵۰۴۰ عملیات طول می‌کشد!

تعداد شجر	تعداد عملیات
۶	۷۲۰
۷	۵۰۴۰
۸	۴۰۳۲۰
...	...
۱۵	۱,۳۰۷,۶۷۴,۳۶۸,۰۰۰
...	...
۳۰	۲,۶۵۲,۵۲,۸۵۹,۸۱۲,۱۹۱,۰۵۸,۶۳۶,۳۰۸,۴۸۰,۰۰۰,۰۰۰

تعداد عملیات  
به شدت افزایش می یابد.

به طور کلی، برای  $n$  آیت،  $n!$  (فاکتوریل) عملیات برای محاسبه‌ی نتیجه طول می‌کشد. در نتیجه  $O(n!)$  یا مرتبه اجرایی فاکتوریل است. تعداد بالای عملیات در تمامی شرایط (به استثنای ارقام بسیار کوچک) صرف خواهد شد. هنگامی که با بیش از ۱۰۰ شجر سر و کار دارید، محاسبه‌ی پاسخ در زمان مورد نظر ناممکن است و پیش از اینکه به جواب برسید دنیا به پایان رسیده است.

این یک الگوریتم وحشتناک است! اپوس باید از الگوریتم دیگری استفاده کند، درست است؟ اما اطلاعی از آن ندارد. چون این یکی از مسائل حل نشده در علوم کامپیوتر است. هیچ الگوریتم سریع شناخته شده‌ای برای آن وجود ندارد و متخصص‌ها بر این باورند که دسترسی به یک الگوریتم هوشمند برای این مسئله ناممکن است. بهترین کاری که می‌توان انجام داد این است که یک راه حل تقریبی ارائه بدهیم. برای اطلاعات بیشتر به فصل ۱۰ مراجعه کنید.

یک نکته‌ی پایانی: اگر به مسئله‌های پیچیده‌تر علاقه‌مند هستید، درخت‌های جست‌وجوی دودویی را بررسی کنید! شرح مختصری از آن‌ها در فصل آخر آمده است.



## جمع بندی

- جست و جوی دودویی بسیار سریع تر از جست و جوی ساده است.
- $O(\log n)$  سریع تر از  $O(n)$  است، اما زمانی که لیست آیتیم هایی که در آن جستجو می کنید بزرگ شود، بسیار سریع تر هم خواهد شد.
- سرعت الگوریتم به ثانیه اندازه گیری نمی شود.
- زمان الگوریتم بر حسب رشد یک الگوریتم اندازه گیری می شود.
- زمان الگوریتم با نماد  $O$  بزرگ نوشته می شود.



## ۲ | مرتب‌سازی انتخابی



در این فصل

• درباره‌ی آرایه‌ها و لیست‌های پیوندی - دو تا از اساسی‌ترین ساختمان‌های داده که در همه‌جا از آن‌ها استفاده می‌شوند - یاد می‌گیرید. پیش از این از آرایه‌ها در فصل ۱ استفاده کرده‌اید، و تقریباً در هر یک از فصل‌های این کتاب از آن‌ها استفاده می‌کنید. آرایه‌ها موضوعی حیاتی هستند، پس به آن‌ها توجه کنید! اما گاهی اوقات بهتر است از یک لیست پیوندی استفاده کنید. در این فصل مزایا و معایب هر دو توضیح داده می‌شود تا بتوانید تصمیم بگیرید کدام یک برای الگوریتم شما مناسب است.

• اولین الگوریتم مرتب‌سازی خود را یاد می‌گیرید. بسیاری از الگوریتم‌ها تنها در صورتی کار می‌کنند که داده‌های شما مرتب شده باشند. جست‌وجوی دودویی را به یاد دارید؟ جست‌وجوی دودویی را تنها می‌توانید بر یک لیست مرتب‌شده از عناصر اجرا کنید. در این فصل مرتب‌سازی انتخابی را یاد می‌گیرید. اکثر زبان‌ها دارای یک الگوریتم مرتب‌سازی داخلی و از پیش نوشته‌شده هستند، بنابراین به‌ندرت نیاز است تا نسخه‌ای از این الگوریتم را از صفر بنویسید. اما مرتب‌سازی انتخابی مقدمه‌ای است بر مرتب‌سازی سریع که در فصل بعدی به آن می‌پردازیم. مرتب‌سازی سریع، الگوریتمی مهم است و اگر از قبل یک الگوریتم مرتب‌سازی را بشناسید، درک آن آسان‌تر خواهد شد.

### پیش‌نیازها:

برای درک تحلیل کارایی<sup>۱</sup> در این فصل، باید نماد O بزرگ و لگاریتم‌ها را بشناسید. اگر چیزی درباره‌ی آن‌ها نمی‌دانید، پیشنهاد می‌کنم به عقب برگشته و فصل ۱ را بخوانید. از نماد O بزرگ در سراسر کتاب استفاده می‌شود.

### شیوه‌ی عملکرد حافظه

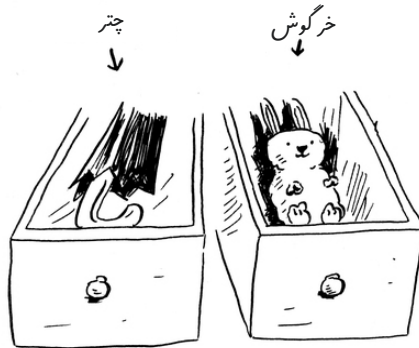
تصور کنید به یک نمایش می‌روید و باید وسایلی را که همراه خود دارید تحویل بدهید. یک گنجی در آنجا قرار گرفته است.



هر یک از کشوها می‌تواند یک وسیله را در خود جای بدهد. دو وسیله‌ی همراه خود را می‌خواهید امانت بدهید، پس دو کشو لازم دارید.

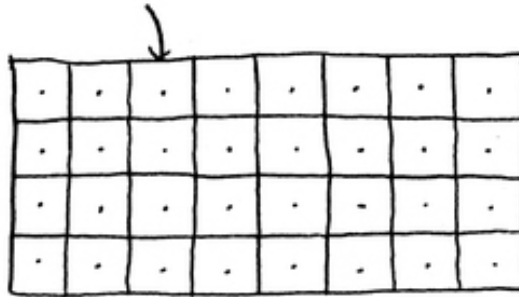


وسایل همراه خود را در این دو کشو قرار می‌دهید.



حالا برای نمایش آماده هستید! اساساً حافظه‌ی کامپیوتر شما این‌گونه عمل می‌کند. کامپیوتر شما شبیه یک مجموعه‌ی بزرگ از کشوها است و هر کشو دارای یک آدرس است.

آدرس : fe0ffeeb



fe0ffeeb آدرس یک روزنه یا اسلات<sup>۱</sup> در حافظه است.

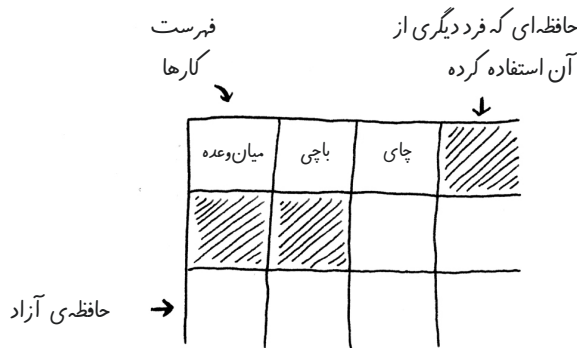
هر بار که می‌خواهید یک آیتم را در حافظه ذخیره کنید، از کامپیوتر درخواست مقداری فضا می‌کنید و کامپیوتر آدرسی به شما می‌دهد که می‌توانید آیتم خود را در آنجا ذخیره کنید. اگر می‌خواهید چندین آیتم را ذخیره کنید، دو راه اساسی برای این کار وجود دارد: آرایه‌ها و لیست‌ها. در ادامه در مورد آرایه‌ها و لیست‌ها و همچنین مزایا و معایب هر کدام صحبت خواهیم کرد. یک راه واحد برای ذخیره‌ی آیتم‌ها در همه‌ی شرایط وجود ندارد، بنابراین مهم است که از تفاوت آن‌ها آگاه باشید.

1. slot

## آرایه‌ها و لیست‌های پیوندی

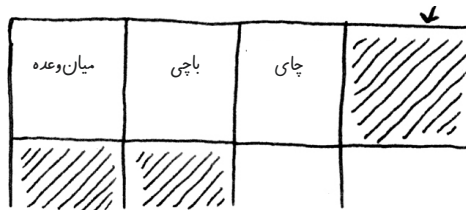


گاهی نیاز است لیستی از عناصر را در حافظه ذخیره کنید. فرض کنید در حال نوشتن یک برنامه برای مدیریت کارهای روزمره‌ی خود هستید. شما می‌خواهید کارهای انجام شده را به عنوان یک لیست در حافظه ذخیره کنید. باید از یک آرایه استفاده کنید یا یک لیست پیوندی؟ بیایید برای درک آسان‌تر فهرست کارهای روزانه را در یک آرایه ذخیره کنیم. استفاده از آرایه به این معنی است که تمام تسک‌های شما به طور پیوسته (در کنار یکدیگر) در حافظه ذخیره می‌شوند.



حالا فرض کنید می‌خواهید یک کار چهارم اضافه کنید. اما کشوی بعدی با وسایل شخص دیگری اشغال شده است!

در این محل که پیش از این اشغال شده، نمی‌توانید تسک جدیدی اضافه کنید.



مثل این است که با دوستان خود به سینما بروید و جایی برای نشستن پیدا کنید - اما دوست دیگری به شما می‌پیوندد و جایی برایش باقی نمانده است. شما باید به یک مکان دیگر بروید که در آن برای همه‌ی شما جا باشد. در این مورد، باید از کامپیوتر خود یک تکه حافظه‌ی متفاوت بخواهید که بتواند چهار تسک را در خود جای بدهد. سپس باید تمام تسک‌های خود را به آنجا منتقل کنید.

اگر دوست دیگری سر برسد، شما دوباره با مشکل جا مواجه می‌شوید - و همگی باید برای بار دوم جابه‌جا بشوید! چه مکفاتی. به همین ترتیب، افزودن موارد جدید به یک آرایه می‌تواند دردسر بزرگی باشد. اگر فضا ندارید و هر بار نیاز دارید که به یک نقطه‌ی جدید در حافظه بروید، افزودن یک آیتیم جدید واقعا کند خواهد بود. یکی از راه‌حل‌های آسان این است که «صندلی‌های خالی را برای خود نگه دارید»: حتی اگر فقط ۳ آیتیم در لیست تسک خود داشته باشید، می‌توانید محض احتیاط از کامپیوتر ۱۰ اسلات بخواهید. سپس می‌توانید ۱۰ آیتیم را بدون نیاز به جابه‌جایی به فهرست وظایف خود اضافه کنید. این یک راه حل خوب است، اما باید از چند جنبه‌ی منفی آن آگاه باشید:

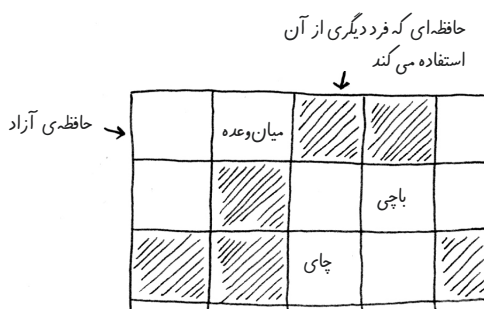
- ممکن است به اسلات‌های اضافی درخواستی نیاز پیدا نکنید و در نتیجه آن حافظه هدر برود. شما از آن استفاده نمی‌کنید، اما هیچ کس دیگری نیز نمی‌تواند از آن استفاده کند.

- ممکن است بیش از ۱۰ آیتیم را به لیست تسک خود اضافه کنید و به هر حال مجبور به حرکت شوید.

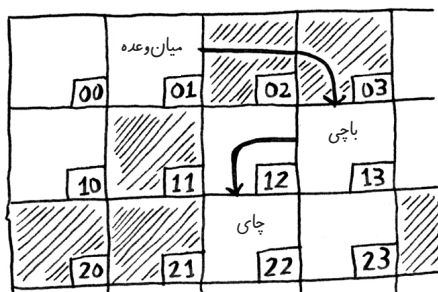
بنابراین این روش راه‌حل بدی نیست، اما چندان راه‌حل بی‌نقصی هم نیست. لیست‌های پیوندی این مشکل اضافه کردن آیتیم‌ها را حل می‌کند.

## لیست‌های پیوندی

با لیست‌های پیوندی، آیتیم‌های شما می‌توانند در هر جایی از حافظه باشند.



هر آیتم آدرس بعدی لیست را ذخیره می‌کند. دسته‌ای از آدرس‌های حافظه‌ی تصادفی<sup>۱</sup> به هم پیوند داده شده‌اند.



آدرس‌های حافظه‌ی پیوندی

درست مثل گنج‌یابی است. به آدرس اول می‌روید، و گفته می‌شود: «آیتم بعدی را می‌توان در آدرس ۱۲۳ پیدا کرد.» پس به آدرس ۱۲۳ می‌روید و می‌گویید: «آیتم بعدی را می‌توان در آدرس ۸۴۷ پیدا کرد»، و غیره. افزودن یک آیتم به لیست پیوندی آسان است: آن را در هر جایی در حافظه قرار می‌دهید و آدرس را با آیتم قبلی ذخیره می‌کنید. با لیست‌های پیوندی، هرگز مجبور نیستید آیتم‌های مورد نظر خود را جابه‌جا کنید. همچنین از یک مشکل دیگر جلوگیری می‌کنید. فرض کنید با پنج نفر از دوستانتان به تماشای یک فیلم محبوب می‌روید. شما شش نفر سعی می‌کنید جایی برای نشستن پیدا کنید، اما سالن پر از تماشاگر شده است. شش صندلی خالی در کنار هم در دسترس نیست. خب، گاهی اوقات این اتفاق با آرایه‌ها می‌افتد. فرض کنید در تلاش برای یافتن

1. Random memory



۱۰۰۰۰ روزنه برای یک آرایه هستید. حافظه‌ی شما ۱۰۰۰۰ روزنه دارد، اما این ۱۰۰۰۰ روزنه در کنار یکدیگر نیستند. شما نمی‌توانید فضایی برای آرایه‌ی خود داشته باشید! یک لیست پیوندی مانند این است که بگویید: «بیایید جدا جدا بنشینیم و فیلم را تماشا کنیم.» اگر در حافظه‌ی فضای خالی موجود باشد، فضایی برای لیست پیوندی خود خواهید داشت. اگر لیست‌های پیوندی برای درج عملکرد بسیار بهتری دارند، پس آرایه‌ها برای چه کاری مناسب هستند؟

## آرایه‌ها

وب‌سایت‌هایی که ۱۰ تا از برترین‌ها را فهرست می‌کنند، از یک تاکتیک شیطانی برای بازدید بیشتر صفحه‌های وب‌سایت استفاده می‌کنند.



گره‌ی شیطانی

بعدی

به جای اینکه لیست را در یک صفحه به شما نشان بدهند، هر آیت‌م را در یک صفحه قرار می‌دهند و شما را مجبور می‌کنند روی گزینه‌ی بعدی کلیک کنید تا به مورد بعدی در لیست برسید. به عنوان مثال، برای نمایش لیست ۱۰ تا از شرورترین شخصیت‌های تلویزیونی کل لیست در یک صفحه به شما نمایش داده نمی‌شود. در عوض، از شماره‌ی ۱۰ (نیومن)

شروع می‌شود و باید در هر صفحه روی گزینه‌ی بعدی کلیک کنید تا به شماره‌ی ۱ (گاس فرینگ) برسید. با این تمهید هر مطلب ۱۰ صفحه برای نمایش تبلیغات خواهد داشت. البته که ۹ بار کلیک روی گزینه‌ی بعدی برای رسیدن به سر فهرست کاری خسته‌کننده است و اگر کل لیست در یک صفحه باشد و بتوانید برای اطلاعات بیشتر روی نام هر فرد کلیک کنید، بسیار بهتر است.

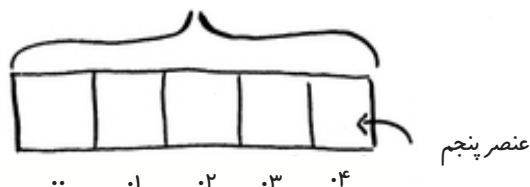
لیست‌های پیوندی نیز مشکل مشابهی دارند. فرض کنید می‌خواهید آخرین آیت‌م را در یک لیست پیوندی بخوانید. شما نمی‌توانید یک سراغ آن رفته و آن را بخوانید، زیرا نمی‌دانید در چه آدرسی قرار گرفته است. در عوض، باید به آیت‌م شماره‌ی ۱ رفته تا

آدرس آیتم شماره‌ی ۲ را به دست بیاورید.

سپس برای دریافت آدرس آیتم شماره‌ی ۳ باید به آیتم شماره‌ی ۲ بروید. و به همین ترتیب، تا زمانی که به آخرین آیتم برسید. اگر بخواهید همه‌ی آیتم‌ها را در یک نوبت بخوانید، لیست‌های پیوندی عالی هستند: می‌توانید یک آیتم را بخوانید، آدرس آیتم بعدی را دنبال کنید و غیره. اما اگر بخواهید مدام از یک آیتم به آیتمی دیگر بروید، لیست‌های پیوندی اصلاً مناسب نیستند.

آرایه‌ها متفاوت هستند. شما آدرس هر آیتم را در آرایه‌ی خود می‌دانید. برای مثال، فرض کنید آرایه‌ی شما پنج آیتم را شامل می‌شود، و می‌دانید که از آدرس ۰۰ شروع می‌شود. آدرس آیتم شماره‌ی ۵ چیست؟

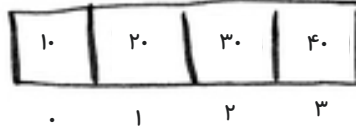
آرایه‌ای از پنج آیتم



با یک جمع و تفریق ساده می‌دانید جواب ۰۴ است. اگر می‌خواهید عناصر تصادفی را بخوانید، آرایه‌ها عالی هستند، زیرا فوراً می‌توانید هر عنصری را در آرایه‌ی خود جست‌وجو کنید. با یک لیست پیوندی، عناصر در کنار یکدیگر نیستند، بنابراین شما نمی‌توانید درجا موقعیت عنصر پنجم را در حافظه محاسبه کنید - باید به عنصر اول رفته تا آدرس عنصر دوم را دریافت کنید، سپس به عنصر دوم بروید تا آدرس عنصر سوم را دریافت کنید، و به همین ترتیب. تا زمانی که به عنصر پنجم برسید.

## اصطلاح‌شناسی

عناصر موجود در یک آرایه شماره‌گذاری شده‌اند. این شماره‌گذاری به جای ۱ از ۰ شروع می‌شود. برای مثال در این آرایه ۲۰ در موقعیت ۱ قرار دارد.



و ۱۰ در موقعیت صفر است. این موضوع معمولاً باعث آشفتگی برنامه‌نویسان تازه‌کار است. شروع از صفر، نوشتن انواع کدهای مبتنی بر آرایه را آسان‌تر می‌کند، بنابراین برنامه‌نویسان به آن پایبند هستند. تقریباً هر زبان برنامه‌نویسی که استفاده می‌کنید، عناصر آرایه‌ای را که از ۰ شروع می‌شوند شماره‌گذاری می‌کنند. به زودی به آن عادت خواهید کرد.

موقعیت یک عنصر را ایندکس<sup>۱</sup> یا اندیس آن می‌نامند. بنابراین به جای گفتن «۲۰ در جایگاه ۱ است»، اصطلاح صحیح این است که «۲۰ در ایندکس ۱ است.» در این کتاب از ایندکس به معنای موقعیت یا جایگاه<sup>۲</sup> استفاده می‌کنم. در اینجا زمان اجرای عملیات رایج در آرایه‌ها و لیست‌ها آمده است.

	آرایه	لیست
خواندن (READING)	$O(1)$	$O(n)$
درج (INSERTION)	$O(n)$	$O(1)$

$$O(n) = \text{زمان خطی}$$

$$O(1) = \text{زمان ثابت}$$

1. index
2. position

سؤال: چرا زمان  $O(n)$  برای وارد کردن یک عنصر در یک آرایه نیاز است؟ فرض کنید می‌خواهید عنصری را در ابتدای یک آرایه وارد کنید. چگونه این کار را انجام می‌دهید؟ چه مدت طول می‌کشد؟ پاسخ این سؤالات را در بخش بعدی بخوانید!

### تمرین

۲.۱ فرض کنید در حال ساخت یک برنامه برای پیگیری امور مالی خود هستید.

۱. خورد و خوراک روزمره

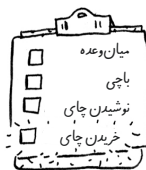
۲. سینما

۳. حق عضویت SFBC

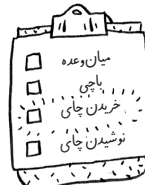
هرروز خرج‌های خود را یادداشت می‌کنید. در پایان ماه، هزینه‌ها را بررسی کرده و میزان هزینه‌های خود را جمع می‌زنید. بنابراین، شما تعداد زیادی درج‌کردن و مقدار کمی خواندن دارید. برای این برنامه باید از میان آرایه و لیست کدام را استفاده کنید؟

### درج در وسط یک لیست

فرض کنید می‌خواهید لیست کارهای روزانه‌ی شما بیشتر شبیه یک تقویم باشد. پیش از این عنوان کارهای روزانه را به انتهای لیست اضافه می‌کردید. اما حالا می‌خواهید آن‌ها را با اولویت انجام دادن اضافه کنید.

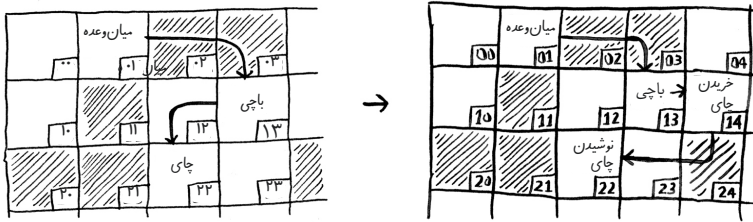


بدون ترتیب

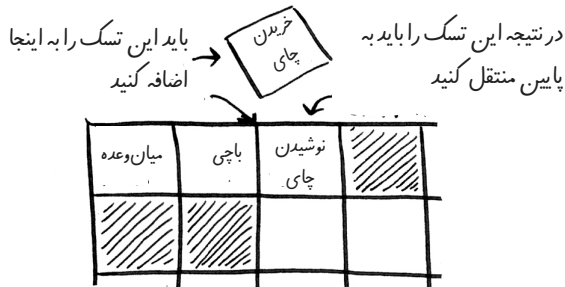


مرتب شده

اگر بخواهید عناصر را در وسط لیست وارد کنید، کدام روش بهتر است: آرایه یا لیست؟ از طریق لیست، آسان است و تنها کافی است آنچه عنصر قبلی به آن اشاره می‌کند، تغییر دهید.



اما با آرایه‌ها، باید بقیه‌ی عناصر را به پایین جابه‌جا کنید.



و اگر فضایی وجود نداشته باشد، ممکن است مجبور شوید همه چیز را در یک مکان جدید کپی کنید! اگر می‌خواهید عناصر را در وسط قرار دهید، لیست‌ها بهتر هستند.

## حذف

اگر بخواهید یک عنصر را حذف کنید چه؟ باز هم، لیست‌ها بهتر هستند، زیرا فقط باید آنچه را که عنصر قبلی به آن اشاره می‌کند تغییر دهید. با آرایه‌ها، وقتی یک عنصر را حذف می‌کنید، همه چیز باید به بالای خود منتقل شود.

برخلاف درج، حذف همیشه کارساز است. درج ممکن است گاهی اوقات که فضایی در حافظه باقی‌مانده امکان‌پذیر نباشد. اما همیشه می‌توانید یک عنصر را حذف کنید.

### 1. Deletions

در اینجا زمان اجرا برای عملیات رایج<sup>۱</sup> در آرایه‌ها و لیست‌های پیوندی آمده است.

	آرایه‌ها	لیست‌ها
خواندن	$O(1)$	$O(n)$
درج کردن	$O(n)$	$O(1)$
حذف کردن	$O(n)$	$O(1)$

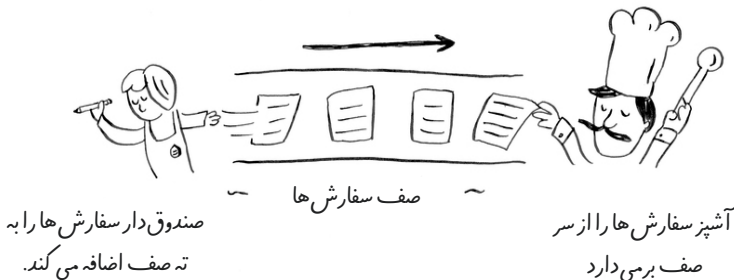
شایان ذکر است که درج و حذف فقط زمانی  $O(1)$  هستند که فوراً بتوانید به عنصری که باید حذف بشود دسترسی پیدا کنید. پیگیری اولین و آخرین آیت‌ها در یک لیست پیوندی یک روش معمول است، بنابراین حذف آن‌ها تنها به زمان  $O(1)$  نیاز خواهد داشت.

کدام بیشتر استفاده می‌شود: آرایه یا لیست؟ بدیهی است که بستگی به مورد استفاده دارد. آرایه‌ها کاربرد زیادی دارند زیرا امکان دسترسی تصادفی<sup>۲</sup> را فراهم می‌کنند. دو نوع دسترسی وجود دارد: دسترسی تصادفی و دسترسی ترتیبی<sup>۳</sup>. دسترسی ترتیبی به معنای خواندن یک به یک عناصر، از اولین عنصر است. از لیست‌های پیوندی فقط می‌توان برای دسترسی ترتیبی استفاده کرد. اگر می‌خواهید دهمین عنصر یک لیست پیوندی را بخوانید، باید ۹ عنصر اول را بخوانید و پیوندها را تا عنصر دهم دنبال کنید. دسترسی تصادفی به این معنی است که می‌توانید مستقیماً به عنصر دهم بروید. به دفعات از زبان من می‌شنوید که می‌گویم آرایه‌ها برای عمل خواندن سریع‌تر هستند. به این دلیل که دسترسی تصادفی از ویژگی‌های آن است. بسیاری از موارد استفاده نیاز به دسترسی تصادفی دارند، بنابراین از آرایه در موارد بسیاری استفاده می‌شود. از آرایه و لیست برای پیاده‌سازی سایر ساختمان‌های داده هم استفاده می‌شوند (در ادامه‌ی کتاب آمده است).

1. common operations
2. Random accesss
3. sequential access

## تمرین

**۲.۲** فرض کنید در حال ساخت برنامه‌ای برای رستوران‌ها هستید تا سفارش مشتری‌ها در آن ثبت بشود. برنامه‌ی شما باید فهرستی از سفارشات را ذخیره کند. گارسون‌ها مدام به این لیست سفارش‌هایی را اضافه می‌کنند، و آشپزها سفارشات را از لیست می‌بینند و آن‌ها را آماده می‌کنند. این یک صف سفارش است: گارسون‌ها سفارشات را به ته صف اضافه می‌کنند، و آشپزها اولین سفارش را از سر صف بر می‌دارند و آن را حاضر می‌کنند.

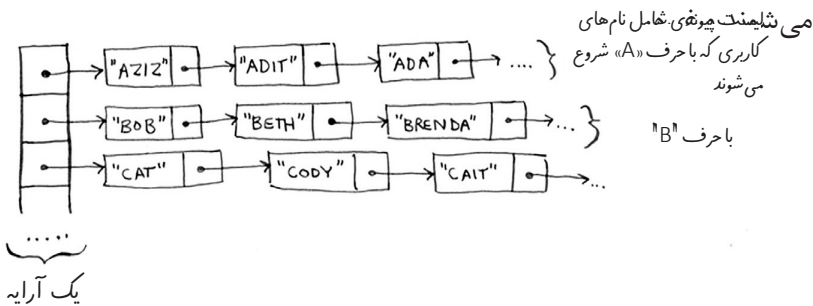


برای پیاده‌سازی این صف از میان آرایه و لیست پیوندی کدام را استفاده می‌کنید؟ (راهنمایی: لیست‌های پیوندی برای درج/حذف و آرایه‌ها برای دسترسی تصادفی مناسب هستند. کدام یک را در اینجا به کار می‌برید؟)

**۲.۳** بیایید یک آزمایش نظری انجام بدهیم. فرض کنید فیس‌بوک لیستی از نام‌های کاربری را نگه می‌دارد. هنگامی که شخصی سعی می‌کند به فیس‌بوک وارد بشود، نام کاربری او جست‌وجو می‌شود. اگر نام آن‌ها در لیست نام‌های کاربری باشد، می‌توانند وارد شوند. به دفعات کاربران وارد فیس‌بوک می‌شوند، بنابراین جست‌وجوهای زیادی از طریق این لیست از نام‌های کاربری انجام می‌شود. فرض کنید فیس‌بوک از جست‌وجوی دودویی برای جست‌وجوی لیست استفاده می‌کند. جست‌وجوی دودویی نیاز به دسترسی تصادفی دارد. شما باید بتوانید فوراً به وسط لیست نام‌های کاربری برسید. با دانستن این موضوع، آیا لیست را به شکل یک آرایه یا یک لیست پیوندی پیاده‌سازی می‌کنید؟

**۲.۴** همچنین کاربرها بیشتر اوقات در فیس بوک ثبت نام می کنند. فرض کنید تصمیم گرفتید از یک آرایه برای ذخیره ی لیست کاربران استفاده کنید. استفاده از آرایه برای درج چه جنبه های منفی به دنبال دارد؟ به طور خاص، فرض کنید از جست و جوی دودویی برای لاگین<sup>۱</sup> استفاده می کنید. وقتی کاربران جدیدی به آرایه اضافه می کنید چه اتفاقی می افتد؟

**۲.۵** در واقعیت، فیس بوک از آرایه و لیست پیوندی برای ذخیره ی اطلاعات کاربر استفاده نمی کند. بیا ببینیم ساختمان داده ی ترکیبی را در نظر بگیریم: آرایه ای از لیست های پیوندی. شما یک آرایه با ۲۶ اسلات دارید. هر اسلات به یک لیست پیوندی اشاره می کند. به عنوان مثال، اولین اسلات در آرایه به یک لیست پیوندی اشاره می کند که شامل تمامی نام های کاربری است که با حرف a شروع می شوند. اسلات دوم به یک لیست پیوندی اشاره می کند که شامل همه ی نام های کاربری است که با حرف b شروع



فرض کنید Adit B در فیس بوک ثبت نام کرده است، و شما می خواهید او را به لیست اضافه کنید. به اسلات ۱ آرایه می روید، به لیست پیوندی برای اسلات ۱ رفته و Adit B را به پایان لیست اضافه می کنید. حال، فرض کنید می خواهید Zakhir H را جست و جو کنید. به اسلات ۲۶ می روید، که به لیست پیوندی از همه ی نام های Z اشاره می کند. سپس در آن لیست جست و جو می کنید تا Zakhir H را پیدا کنید.

این ساختمان داده ی ترکیبی را با آرایه ها و لیست های پیوندی مقایسه کنید. به نسبت آرایه و فهرست پیوندی برای جست و جو و درج کندتر است یا سریع تر؟ نیازی



نیست زمان اجرای O بزرگ را محاسبه کنید، فقط اینکه ساختمان داده‌ی جدید سریع‌تر است یا کندتر؟

## مرتب‌سازی انتخابی



بباید تمامی این موارد را در کنار هم قرار دهیم تا الگوریتم دوم را یاد بگیریم: مرتب‌سازی انتخابی. برای درک مطالب این بخش، باید آرایه‌ها و لیست‌ها و همچنین نماد O بزرگ را بلد باشید.

فرض کنید تعدادی آهنگ در کامپیوتر خود دارید. در کنار نام هر هنرمند، تعداد دفعات پخش قطعات موسیقی هر هنرمند فهرست شده است.

دفعات پخش	
۱۵۶	RADIOHEAD
۱۴۱	KISHORE KUMAR
۳۵	THE BLACK KEYS
۹۴	NEUTRAL MILK HOTEL
۸۸	BECK
۶۱	THE STROKES
۱۱۱	WILCO

می‌خواهید این فهرست را از بیشترین به کمترین تعداد پخش مرتب کنید تا بتوانید هنرمندان مورد علاقه‌ی خود را رتبه‌بندی کنید. چگونه چنین کاری امکان‌پذیر است؟ یکی از راه‌ها این است که لیست را مرور کرده و هنرمندی را که بیشترین پخش را داشته، پیدا کنید. آن هنرمند را به یک لیست جدید اضافه کنید.

دفعات پخش	🎵
۱۵۶	RADIOHEAD
۱۴۱	KISHORE KUMAR
۳۵	THE BLACK KEYS
۹۴	NEUTRAL MILK HOTEL
۸۸	BECK
۶۱	THE STROKES
۱۱۱	WILCO

→

دفعات پخش	🎵 به ترتیب
۱۵۶	RADIOHEAD

۱. ریدوهد از همه بیشتر پخش شده...

۲. به یک لیست جدید اضافه کنید

دوباره این کار را انجام دهید تا هنرمندی که بیشترین بار پخش شده است را پیدا

کنید.

دفعات پخش	🎵
۱۴۱	KISHORE KUMAR
۳۵	THE BLACK KEYS
۹۴	NEUTRAL MILK HOTEL
۸۸	BECK
۶۱	THE STROKES
۱۱۱	WILCO

→

دفعات پخش	🎵 به ترتیب
۱۵۶	RADIOHEAD
۱۴۱	KISHORE KUMAR

۱. کیشور کومار در فهرست پرشونده ترین خواننده رتبه‌ی دوم را دارد

۲. پس هنرمند بعدی است که به لیست جدید اضافه می‌شود

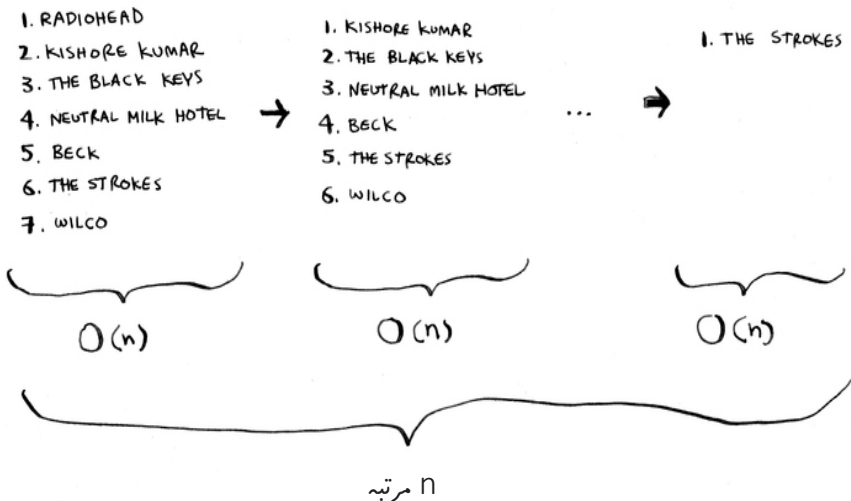
این کار را ادامه دهید و در نهایت یک لیست مرتب شده خواهید شد.

🎵	PLAY COUNT
RADIOHEAD	۱۵۶
KISHORE KUMAR	۱۴۱
WILCO	۱۱۱
NEUTRAL MILK HOTEL	۹۴
BECK	۸۸
THE STROKES	۶۱
THE BLACK KEYS	۳۵

بیا یاد عینک علوم کامپیوتری خود را به چشم بزنیم و ببینیم این کار چقدر طول می‌کشد. به یاد داشته باشید که زمان  $O(n)$  به این معنی است که شما با هر عنصر لیست یک بار کار دارید. به عنوان مثال، جست‌وجوی ساده روی لیست هنرمندان به معنای یک‌بار بررسی هر هنرمند است.

1. RADIOHEAD
  2. KISHORE KUMAR
  3. THE BLACK KEYS
  4. NEUTRAL MILK HOTEL
  5. BECK
  6. THE STROKES
  7. WILCO
- }  $n$   
آیتم

برای پیدا کردن هنرمندی که بیشترین تعداد پخش را دارد، باید هر کدام را بررسی کنید. همان‌طور که مشاهده کردید، زمان  $O(n)$  طول می‌کشد. بنابراین شما عملیاتی دارید که زمان  $O(n)$  نیاز دارد و باید آن را  $n$  بار انجام دهید:



این زمان  $O(n \times n)$  یا زمان  $O(n^2)$  طول می‌کشد.

الگوریتم‌های مرتب‌سازی بسیار مفید هستند. حالا می‌توانید موارد زیر را مرتب کنید:

- نام‌ها در دفترچه تلفن
- تاریخ سفرها
- ایمیل‌ها (جدیدترین به قدیمی‌ترین)

### هر بار عناصر کمتری را بررسی کنید

شاید با خودتان فکر کنید: همین که پیش می‌روید، تعداد عناصری که باید بررسی کنید به طور دائم کاهش می‌یابد. در نهایت، تنها باید یک عنصر را بررسی کنید. پس چگونه زمان اجرا همچنان  $O(n^2)$  است؟ پرسش خوبی است و پاسخ آن به ثابت‌ها در نماد  $O$  بزرگ برمی‌گردد. در فصل ۴ بیشتر به این موضوع خواهیم پرداخت، اما اصل ماجرا به این صورت است:

درست است که لازم نیست هر بار لیستی از  $n$  عنصر را بررسی کنید. شما  $n$  عنصر را بررسی می‌کنید، سپس  $n-1$ ،  $n-2$ ، ...،  $2$ ،  $1$ . به طور متوسط، لیستی را بررسی می‌کنید که دارای  $n/2$  عنصر است. زمان اجرا  $O(n \times n/2)$  است. اما ثابت‌هایی مانند  $n/2$  در نماد  $O$  بزرگ نادیده گرفته می‌شوند (تکرار می‌کنم برای توضیحات کامل به فصل ۴ مراجعه کنید)، بنابراین فقط  $O(n \times n)$  یا  $O(n^2)$  را می‌نویسید.

مرتب‌سازی انتخابی الگوریتمی منظم است، اما چندان سریع نیست. مرتب‌سازی سریع، الگوریتم مرتب‌سازی سریع‌تری است که فقط زمان  $O(n \log n)$  خواهد داشت. در فصل بعدی به این الگوریتم پرداخته می‌شود!

### نمونه کد لیست

کد مرتب‌سازی مثال فهرست موزیک‌ها را نوشتیم، اما کدی که در زیر آمده است کار بسیار مشابهی را انجام می‌دهد: مرتب‌سازی آرایه از کوچک‌ترین به بزرگ‌ترین. بیایید تابعی بنویسیم تا کوچکترین عنصر یک آرایه را پیدا کند:

```

def findSmallest(arr):
    smallest = arr[0] <----- کوچک‌ترین مقدار را ذخیره می‌کند
    smallest_index = 0 <----- ایندکس کوچک‌ترین مقدار را ذخیره می‌کند
    for i in range(1, len(arr)):
        if arr[i] < smallest:
            smallest = arr[i]
            smallest_index = i
    return smallest_index

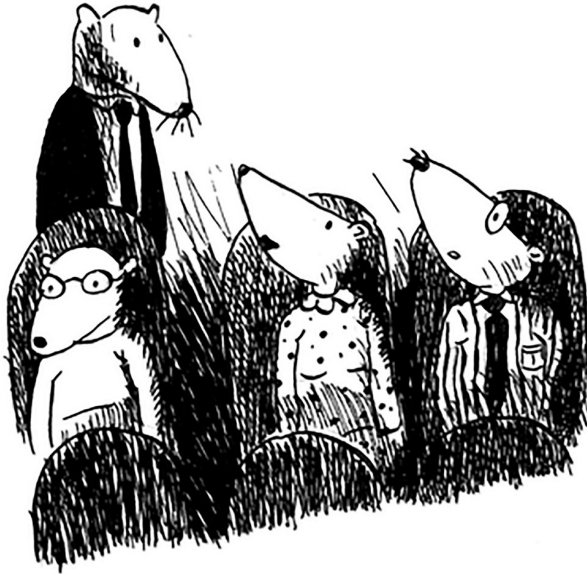
```

اکنون می‌توانید از این تابع برای نوشتن مرتب‌سازی انتخابی استفاده کنید:

```

def selectionSort(arr): <-----
    newArr = [] <----- یک آرایه را مرتب می‌کند
    for i in range(len(arr)):
        smallest = findSmallest(arr)
        newArr.append(arr.pop(smallest)) <----- کوچک‌ترین عنصر آرایه را پیدا کرده و آن را به آرایه‌ی جدید اضافه می‌کند.
    return newArr
print selectionSort([5, 3, 6, 2, 10])

```



## جمع بندی

- حافظه‌ی کامپیوتر شما مانند مجموعه‌ی بزرگی از کشوها است.
- برای ذخیره‌ی چند عنصر، از یک آرایه یا یک لیست استفاده کنید.
- با یک آرایه، تمام عناصر شما درست در کنار یکدیگر ذخیره می‌شوند.
- با یک لیست، عناصر در همه‌جا پراکنده می‌شوند و یک عنصر آدرس عنصر بعدی را ذخیره می‌کند.
- آرایه‌ها اجازه‌ی خواندن سریع را می‌دهند.
- لیست‌های پیوندی اجازه‌ی درج و حذف سریع را می‌دهند.
- همه‌ی عناصر موجود در آرایه باید از یک نوع<sup>۱</sup> باشند (همه‌ی آن‌ها اعداد صحیح<sup>۲</sup> یا دابل<sup>۳</sup> یا موارد دیگر باشند).

1. type
2. integer
3. double

## بازگشت<sup>۱</sup>



در این فصل

• بازگشت را یاد می‌گیرید. بازگشت یک تکنیک کدنویسی است که در بسیاری از الگوریتم‌ها استفاده می‌شود. بازگشت یک زیربنا و پیش‌نیاز برای درک فصل‌های بعدی این کتاب است.

• یاد می‌گیرید که چگونه یک مسئله را به یک صورت پایه<sup>۲</sup> و یک صورت بازگشتی<sup>۳</sup> تقسیم کنید. استراتژی تقسیم و حل (فصل ۴) از این مفهوم ساده برای حل مسئله‌های دشوار استفاده می‌کند.

برای این فصل هیجان‌زده هستیم چون موضوع آن تابع بازگشتی است؛ روشی زیبا برای حل مسئله. بازگشت یکی از موضوعات مورد علاقه‌ی من است، با این حال کدنویسی به روش بازگشت همیشه محل اختلاف نظر بوده است. برنامه‌نویس‌ها یا آن را دوست دارند یا از آن متنفرند، و یا از آن چند سالی متنفرند تا این‌که متوجه می‌شوند دوستش دارند. من شخصاً از جبهه‌ی سوم هستیم. برای اینکه کار را برای شما راحت‌تر کنم، چند توصیه دارم:

• در این فصل نمونه کدهای زیادی آورده شده است. این کدها را برای خودتان اجرا

1. recursion
2. Base case
3. Recursive case

کنید تا ببینید چگونه کار می‌کند.

• همچنین درباره‌ی توابع بازگشتی صحبت خواهیم کرد. برای حداقل یک بار، با قلم و کاغذ به یک تابع بازگشتی پا بگذارید: کاری شبیه به: «خب، من ۵ را به factorial می‌دهم، و سپس حاصل ضرب ۵ در factorial با ورودی ۴ را برمی‌گردانم که می‌شود ...»، و به همین ترتیب. بررسی گام‌به‌گام یک تابع به این شکل به شما می‌آموزد عملکرد یک تابع بازگشتی<sup>۱</sup> به چه شکل است.

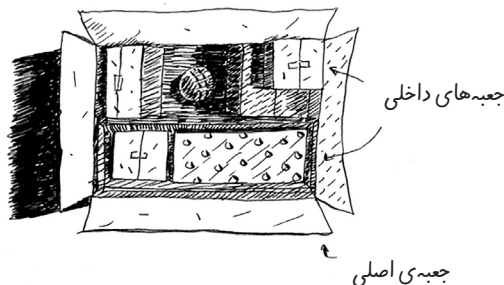
این فصل شامل تعداد متعددی شبه‌کد<sup>۲</sup> است. شبه‌کد یک توضیح سطح بالا<sup>۳</sup> از مسئله‌ای است که در تلاش برای حل آن هستید. مانند کد نوشته می‌شود، اما به گفتار انسان نزدیک‌تر است.

## بازگشت

فرض کنید مشغول زیر و رو کردن وسایل اتاق زیر شیروانی مادر بزرگتان هستید و با یک چمدان قفل شده‌ی مرموز روبه‌رو می‌شوید.



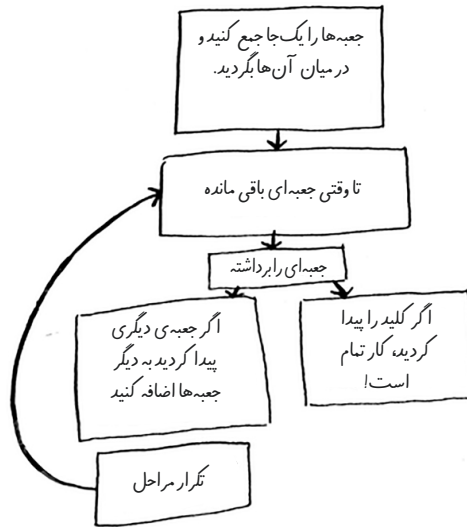
مادر بزرگ به شما می‌گوید که احتمالاً کلید چمدان در این یکی جعبه است.



1. Recursive function
2. Pseudocode
3. High-level

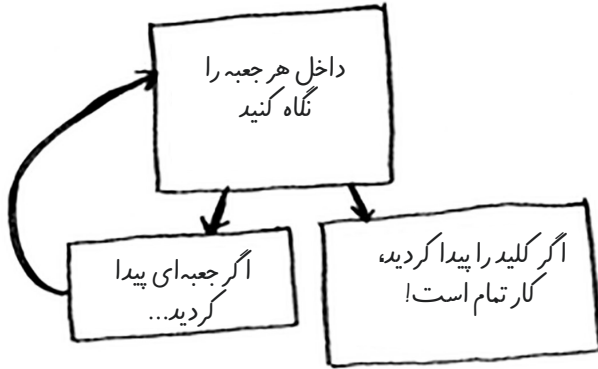


این جعبه حاوی جعبه‌های بیشتری است، و داخل آن جعبه‌ها هم جعبه‌های دیگری هست. کلید داخل یکی از جعبه‌ها است. الگوریتم شما برای جست‌وجوی کلید چیست؟ قبل از خواندن ادامه‌ی کتاب به دنبال یک الگوریتم باشید و به آن فکر کنید. یک رویکرد به این شکل است:



۱. جعبه‌ها را یک جا جمع کنید تا در میان آن‌ها بگردید.
۲. یک جعبه را بردارید و داخل آن را نگاه کنید.
۳. اگر جعبه‌ی دیگری پیدا کردید به دیگر جعبه‌ها اضافه کنید تا بعداً بررسی کنید.
۴. اگر کلیدی پیدا کردید، کار شما تمام شده است!
۵. تکرار مراحل.

در ادامه رویکرد جایگزین آمده است:



۱. داخل جعبه را نگاه کنید.

۲. اگر یک جعبه پیدا کردید به مرحله‌ی اول بروید.

۳. اگر کلیدی پیدا کردید، کار شما تمام شده است!

به نظر شما کدام رویکرد راحت‌تر است؟ در رویکرد اول از حلقه‌ی `while` استفاده می‌شود. تا زمانی که جعبه‌ای باقی مانده، یک جعبه را بردار و داخلش را بگرد:

```
def look_for_key(main_box):
    pile = main_box.make_a_pile_to_look_through()

    While pile is not empty:
        box = pile.grab_a_box()

        for item in box:
            if item.is_a_box():
                pile.append(item)

            elif item.is_a_key():
                print "found the key"!
```

در روش دوم از بازگشت استفاده می‌شود. بازگشت نقطه‌ای است که یک تابع خود را فراخوانی می‌کند. در ادامه روش دوم و در قالب شبه‌کد نوشته شده است:

```
def look_for_key(box):
    for item in box:
        if item.is_a_box():
            look_for_key(item)  ← بازگشت!
        elif item.is_a_key():
            print"found the key"
```

هر دو رویکرد به یک نتیجه ختم می‌شوند، اما رویکرد دوم برای من گویاتر و خواناتر است. کاربرد بازگشت در واضح‌تر شدن راه حل است. هیچ مزیتی در عملکرد استفاده از بازگشت وجود ندارد. در واقع، حلقه‌ها گاهی اوقات عملکرد بهتری دارند. من این نقل قول لی کالدول در Stack Overflow را دوست دارم: «به‌کارگیری حلقه‌ها می‌تواند کارایی برنامه‌ی شما را بالا ببرد. به‌کارگیری بازگشت می‌تواند کارایی برنامه‌نویسی شما را افزایش دهد. انتخاب با شماست که در موقعیت شما کدام اهمیت بیشتری دارد!»  
 بسیاری از الگوریتم‌های مهم از بازگشت استفاده می‌کنند، بنابراین درک مفهوم آن مهم است.

## صورت پایه و صورت بازگشتی



از آنجا که یک تابع بازگشتی خودش را فراخوانی می‌کند، نوشتن یک تابع که به اشتباه به یک حلقه‌ی بی‌نهایت ختم بشود محتمل است.

به عنوان مثال، فرض کنید می‌خواهید تابعی بنویسید که یک شمارش معکوس چاپ کند، مانند این:

> ۳...۲...۱

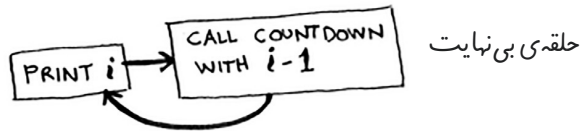
به صورت زیر می‌توان آن را بازگشتی نوشت:

```
def countdown(i):
    print i
    countdown(i-1)
```

این کد را بنویسید و اجرا کنید. متوجه مشکلی خواهید شد: این تابع تا ابد اجرا

1. <http://stackoverflow.com/a/72694/139117>.

خواهد شد!



> ۳...۲...۱...۰...-۱...-۲...

(برای توقف اسکریپت Ctrl-C را فشار بدهید.)

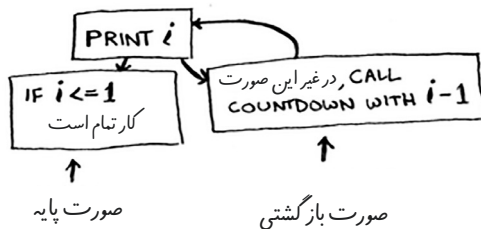
هنگام نوشتن یک تابع بازگشتی، باید مشخص کنید که تابع چه زمانی بازگشت را متوقف کند. به همین دلیل است که هر تابع بازگشتی دارای دو بخش است: صورت پایه و صورت بازگشتی. صورت بازگشتی زمانی است که تابع خود را فراخوانی می‌کند. صورت پایه زمانی است که تابع فراخوانی خود را متوقف می‌کند. بنابراین در یک حلقه‌ی بی‌پایان گرفتار نمی‌شود.

بیا یک صورت پایه به تابع شمارش معکوس اضافه کنیم:

```

def countdown(i):
    print i
    if i <= 1: <----- صورت پایه
        return
    else: <----- صورت بازگشتی
        countdown(i-1)
  
```

اکنون تابع همان طور که انتظار می‌رود کار می‌کند. چیزی شبیه به این:



## پُشته<sup>۱</sup>

در این بخش به پشته‌فراخوانی<sup>۲</sup> می‌پردازیم. پشته‌فراخوانی یک مفهوم مهم در برنامه‌نویسی است و درک آن هنگام استفاده از بازگشت نیز مهم است.



فرض کنید در تدارک برگزاری یک مهمانی باریکیو هستید. شما لیستی از مجموعه کارهایی که برای این مهمانی باید انجام داده شود به شکل پشته‌ای از کاغذهای یادداشت چسب‌دار آماده کرده‌اید.

زمانی را به خاطر بیاورید که در مورد آرایه‌ها و لیست‌ها صحبت می‌کردیم، و شما لیستی از کارهای روزانه داشتید؟ می‌توانستید آیتم کارهای روزانه را در هر نقطه از لیست اضافه یا آیتمی را تصادفی از آن حذف کنید. روش پشته از کاغذهای یادداشت چسب‌دار بسیار ساده‌تر است. هنگامی که یک آیتم را وارد می‌کنید، آن آیتم به بالای لیست اضافه می‌شود. وقتی آیتمی را می‌خوانید، فقط آیتم رویی را می‌خوانید و از لیست حذف می‌شود. بنابراین بر لیست کارها فقط دو عمل اعمال می‌شود: *push* (درج) و *pop* (حذف و خواندن).



PUSH

(اضافه کردن یک آیتم بر روی بقیه‌ی آیتم‌ها)

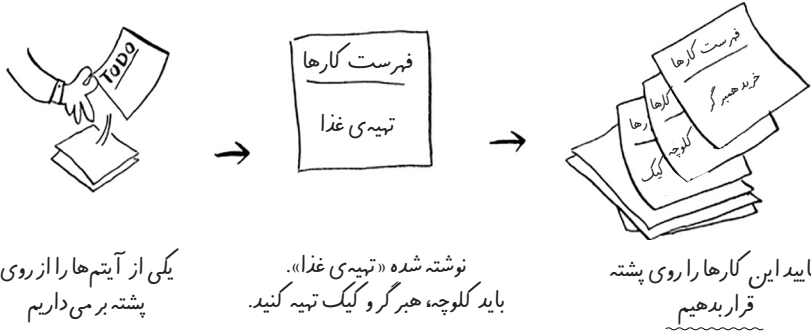


POP

(برداشتن بالاترین آیتم و خواندن آن)

1. The stack
2. Call stack

بیا یاد لیست کارها را در عمل ببینیم.



به این ساختمان داده، پشته می‌گویند. پشته یک ساختمان داده‌ی ساده است. شما تا الان بدون اینکه متوجه باشید از پشته استفاده کرده‌اید!

## پشته‌فراخوانی

کامپیوتر شما از پشته‌ای به نام پشته‌فراخوانی استفاده می‌کند. بیا یاد آن را در عمل ببینیم. در اینجا تابع ساده‌ای وجود دارد:

```
def greet(name):
    print "hello" + name + "!"
    greet2(name)
    print "getting ready to say bye..."
    bye()
```

کار این تابع خوش‌آمدگویی به شما و فراخوانی دو تابع دیگر است. این دو تابع در ادامه آمده است:

```
def greet2(name):
    print "how are you , " + name + "?"

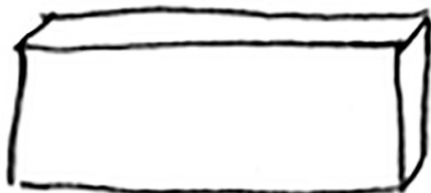
def bye():
    print "ok bye!"
```

بیا بید به آن چه هنگام فراخوانی یک تابع رخ می‌دهد، بپردازیم.

### نکته

print یک تابع در پایتون است، اما برای سادگی کار در این مثال، وانمود می‌کنیم که این طور نیست. فقط تظاهر می‌کنیم.

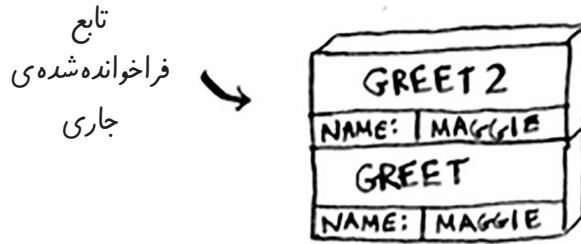
فرض کنید تابع (`"maggie"`) `greet` را فراخوانی می‌کنید. ابتدا، کامپیوتر شما یک جعبه حافظه برای آن فراخوانی تابع اختصاص می‌دهد.



حالا بیا بید از حافظه استفاده کنیم. به متغیر `name` مقدار `"Maggie"` داده شده است. این عمل باید در حافظه ذخیره شود.



هر بار که یک تابع را فراخوانی می‌کنید، کامپیوتر شما مقادیر همه‌ی متغیرهای آن فراخوانی را به این شکل در حافظه ذخیره می‌کند. بعد، شما `hello, maggie!` را پرینت می‌کنید. بعد (`"maggie"`) `greet2` را صدا می‌زنید. مجدداً، کامپیوتر شما یک جعبه حافظه برای این فراخوانی اختصاص می‌دهد.



کامپیوتر شما از یک پشته برای این جعبه‌ها استفاده می‌کند. جعبه‌ی دوم بر روی جعبه‌ی اول اضافه می‌شود. شما `how are you, maggie?` را پرینت می‌کنید سپس از فراخوانی تابع بر می‌گردید. وقتی این اتفاق می‌افتد، جعبه‌ی بالایی پشته بیرون گذاشته می‌شود.



اکنون بالاترین جعبه در پشته تابع `greet` است، به این معنی که شما به تابع `greet` بازگشته‌اید. وقتی تابع `greet2` را فراخوانی کردید، تابع `greet` تا حدودی تکمیل شده بود. این ایده‌ی اصلی این بخش است: وقتی تابعی را از تابع دیگری فراخوانی می‌کنید، تابع فراخوانی در حالت نسبتاً تکمیل شده متوقف می‌شود. تمام مقادیر متغیرهای آن تابع هنوز در حافظه ذخیره می‌مانند. اکنون که کار با تابع `greet2` تمام شد، به تابع `greet` بازگشته و از جایی که کار را متوقف کرده‌اید ادامه می‌دهید. ابتدا `getting ready to say bye` را پرینت می‌کنید... تابع `bye` را صدا می‌زنید.





جعبه‌ای برای آن تابع به بالای پشته اضافه می‌شود. سپس `ok bye!` را پرینت می‌کنید و از فراخوانی تابع باز می‌گردید.



و به تابع `greet` بازگشته‌اید. کار دیگری نمانده است، بنابراین از تابع `greet` هم برمی‌گردید. این پشته که برای ذخیره‌ی متغیرها برای چندین تابع استفاده می‌شود، پشته‌فراخوانی نامیده می‌شود.

### تمرین

۳.۱ فرض کنید من یک پشته‌فراخوانی مانند این را به شما نشان می‌دهم.



تنها بر اساس این پشته‌فراخوانی، چه اطلاعاتی می‌توانید بدهید؟ حالا بیایید پشته‌فراخوانی را در عمل و با یک تابع بازگشتی ببینیم.

## پشته فراخوانی با بازگشت

توابع بازگشتی از پشته فراخوانی هم استفاده می‌کنند! بیایید در عمل با تابع فاکتوریل به این موضوع نگاه کنیم.  $factorial(5)$  به صورت ۵! نوشته می‌شود و به این صورت تعریف می‌شود:  $۱ * ۲ * ۳ * ۴ * ۵ = ۵!$ . به همین شکل  $factorial(3)$  است  $۱ * ۲ * ۳$ . در اینجا یک تابع بازگشتی برای محاسبه‌ی فاکتوریل یک عدد وجود دارد:

```
def fact(x):
    if x == 1:
        return 1
    else:
        return x * fact(x-1)
```

حالا شما  $fact(3)$  را صدا می‌زنید. بیایید خط به خط این فراخوانی را بررسی کنیم و ببینیم که چگونه پشته تغییر می‌کند. به یاد داشته باشید، بالاترین جعبه در پشته به شما می‌گوید که در حال حاضر در داخل کدام فراخوانی از  $fact$  هستید.

پشته فراخوانی (CALL STACK) که

fact(3)

FACT
X   3

فراخوانی اول fact.  
X برابر است با ۳.

if x == 1:

FACT
X   3

else:

FACT
X   3

فراخوانی بازگشتی!

return x \* fact(x-1)

FACT
X   2
FACT
X   3

حالا ما در دومین فراخوانی  
fact هستیم. X برابر با عدد  
۲ است

if x == 1:

FACT
X   2
FACT
X   3

بالاترین فراخوانی تابع همان  
است که در حال حاضر در آن  
هستیم.

else:

FACT
X   2
FACT
X   3

توجه داشته باشید که هر دو فراخوانی  
تابع متغیری با نام X دارند و ارزش X  
در این دو مورد متفاوت است.

return x \* fact(x-1)

FACT
X   1
FACT
X   2
FACT
X   3

از فراخوانی پایین به X بالا و  
از فراخوانی بالا به X پایین  
نمی توان دسترسی داشت

if x == 1:

FACT
X   1
FACT
X   2
FACT
X   3

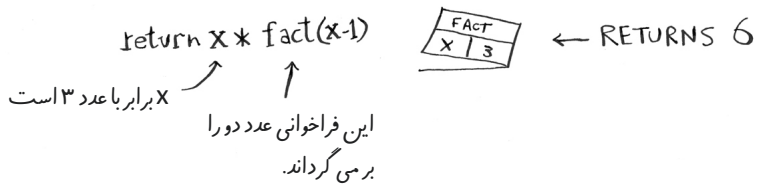
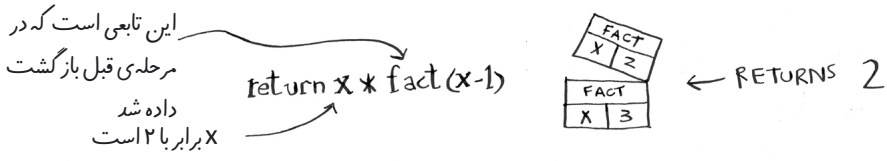
سه مرحله از فراخوانی fact را  
به پیش رفته ایم، اما هنوز یک  
مورد فراخوانی هم تا به اینجا به  
اتمام نرسیده!

return 1

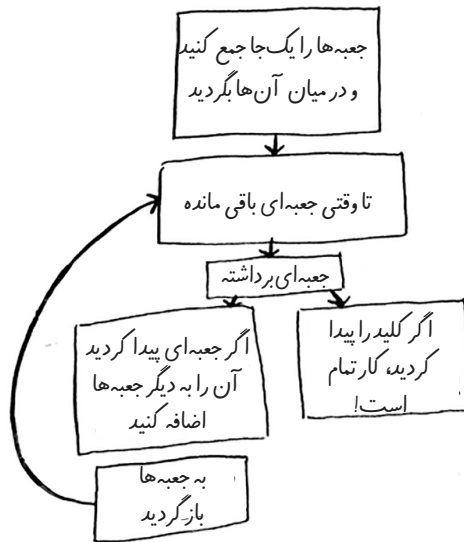
FACT
X   1
FACT
X   2
FACT
X   3

این اولین جعبه ای است که از  
روی پشته برداشته می شود و در نتیجه  
اولین مورد فراخوانی شده است  
که return آن به دست می آید.

RETURNS 1

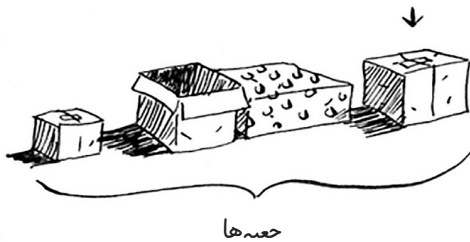


توجه داشته باشید که هر تماس یا فراخوانی fact کپی خود از x را دارد. شما نمی‌توانید به کپی دیگری از تابع x دسترسی پیدا کنید. پشته نقش مهمی در بازگشت بازی می‌کند. در مثال ابتدایی، دو رویکرد برای یافتن کلید وجود داشت. نگاهی دوباره به روش اول بیندازیم:

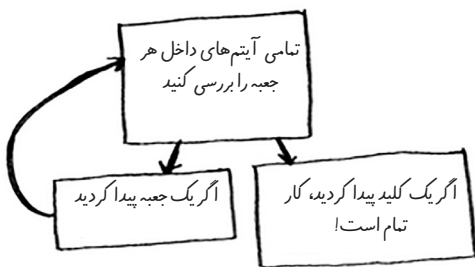


به این ترتیب، دسته‌ای از جعبه‌ها را برای جست‌وجو میان آن‌ها در کنار هم قرار می‌دهید، در نتیجه می‌دانید که چه جعبه‌هایی برای جست‌وجو باقی مانده است.

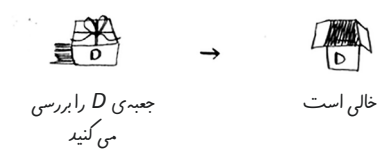
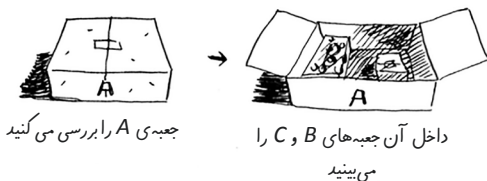
جعبه‌ی بعدی برای جست‌وجو



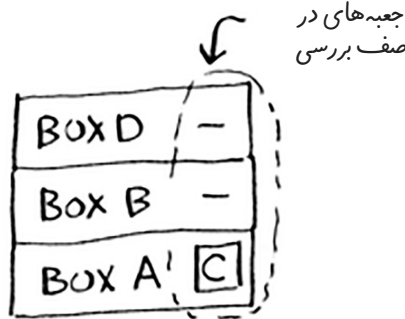
اما در رویکرد بازگشتی، خبری از دسته‌ای از جعبه‌ها نیست.



اگر دسته‌ای از جعبه‌ها وجود نداشته باشد، الگوریتم شما از کجا می‌تواند بفهمد که هنوز باید در میان چه جعبه‌هایی جست‌وجو کند؟ مثال:



در این مرحله، پشته فراخوانی به این شکل است:



«جعبه‌ها» در پشته ذخیره می‌شوند! این پشته‌ای از فراخوانی‌های تابع نیمه تکمیل شده است که هرکدام لیست نیمه کامل خود را از جعبه‌هایی دارد که باید آن‌ها را بگردد. استفاده از پشته راحت است زیرا نیازی نیست حساب مجموعه‌ی جعبه‌ها را داشته باشید - پشته این کار را برای شما انجام می‌دهد.

استفاده از پشته راحت است، اما به یک قیمت: ذخیره همه‌ی آن اطلاعات می‌تواند حافظه‌ی زیادی را اشغال کند. هر یک از این فراخوانی‌های تابع مقداری از حافظه را اشغال می‌کند، و وقتی پشته‌ی شما خیلی بزرگ است، به این معنی است که کامپیوتر شما اطلاعات را برای بسیاری از تماس‌های عملکرد ذخیره می‌کند. در آن مرحله، شما دو گزینه دارید:

- می‌توانید کد خود را بازنویسی کرده و به جای آن از یک حلقه استفاده کنید.
- می‌توانید از چیزی به نام *tail recursion* استفاده کنید. این یک مبحث پیشرفته‌ی بازگشتی است که خارج از محدوده‌ی این کتاب است. از طرفی تنها برخی از زبان‌ها از آن پشتیبانی می‌کنند.

## تمرین

**۳.۲** فرض کنید به طور تصادفی یک تابع بازگشتی نوشتید که تا ابد اجرا می‌شود. همان‌طور که دیدید، کامپیوتر شما برای هر فراخوانی تابع، حافظه را در پشته تخصیص

می دهد. وقتی تابع بازگشتی شما تا ابد اجرا بشود چه اتفاقی برای پشته می افتد؟

### جمع بندی



- بازگشت به زمانی گفته می شود که یک تابع، خود را فراخوانی می کند.
- هر تابع بازگشتی دو صورت دارد: صورت پایه و صورت بازگشتی.
- یک پشته دو عملیات دارد: push و pop.
- همه ی فراخوانی های تابع به پشته فراخوانی می روند.
- فراخوانی تماس می تواند بسیار بزرگ شود و منجر به اشغال حافظه ی زیادی بشود.





## ۴ | مرتب‌سازی سریع



در این فصل

• تقسیم و حل را یاد می‌گیرید. گاهی اوقات با مسئله‌ای روبه‌رو می‌شوید که با هیچ‌یک از الگوریتم‌هایی که آموخته‌اید قابل حل نیست. وقتی یک الگوریتمیست خبره با چنین مشکلی روبرو می‌شود، تسلیم نمی‌شود. این افراد جعبه ابزاری دارند که مملو از تکنیک‌های گوناگون برای حل مسئله است و سعی می‌کنند راه‌حلی برای مسئله بیابند. تقسیم و حل<sup>۱</sup> اولین تکنیک عمومی است که یاد می‌گیرید.

• در مورد مرتب‌سازی سریع، یک الگوریتم مرتب‌سازی خوب که اغلب اوقات از آن استفاده می‌شود، یاد می‌گیرید. مرتب‌سازی سریع از تقسیم و حل استفاده می‌کند.

در فصل پیش همه چیز را درباره‌ی بازگشت آموختید. این فصل بر استفاده از مهارت تازه‌ی شما برای حل مسئله‌ها تمرکز دارد. تقسیم و حل (D&C)، یک تکنیک بازگشتی مشهور برای حل مسئله را بررسی خواهیم کرد.

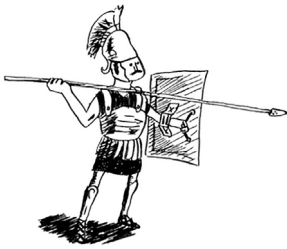
در این فصل به مهم‌ترین بخش الگوریتم‌ها می‌پردازیم. با این همه، یک الگوریتم اگر تنها برای حل یک نوع مسئله به کار بیاید، چندان به درد نمی‌خورد.

1. Divide and conquer

در عوض، D&C راه جدیدی برای فکر کردن در مورد حل مسئله‌ها در اختیار می‌گذارد. D&C ابزار دیگری در جعبه ابزار شماست. وقتی با یک مشکل جدید مواجه می‌شوید، لازم نیست سرگردان شوید. در عوض، می‌توانید بپرسید: «آیا می‌توانیم در صورت استفاده از تقسیم و حل این مسئله را حل کنیم؟»

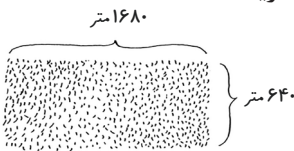
در پایان فصل، اولین الگوریتم مهم D&C را می‌آموزید: مرتب‌سازی سریع. مرتب‌سازی سریع، یک الگوریتم مرتب‌سازی است و بسیار سریع‌تر از مرتب‌سازی انتخابی (که در فصل ۲ یاد گرفتید) است. مرتب‌سازی سریع مثال خوبی از کدنویسی درجه‌ی یک است.

## تقسیم و حل

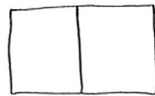


درک D&C یا تقسیم و حل ممکن است کمی طول بکشد. بنابراین، ما سه مثال خواهیم داشت. ابتدا مثالی تصویری را به شما نشان می‌دهم. بعد نمونه کدی را می‌نویسم که اگرچه چندان جالب نیست اما احتمالاً آسان باشد. در نهایت، مرتب‌سازی سریع را بررسی می‌کنیم، یک الگوریتم مرتب‌سازی که از D&C استفاده می‌کند.

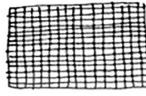
فرض کنید شما یک کشاورز هستید و یک قطعه زمین دارید.



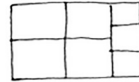
قصد دارید این مزرعه را به طور مساوی به قطعات مربعی شکل تقسیم کنید. همچنین می‌خواهید نقشه‌ها تا حد امکان بزرگ باشند. بنابراین هیچ کدام از موارد زیر به‌کار نمی‌آید.



زمین به قطعات مربعی شکل تقسیم نشده است



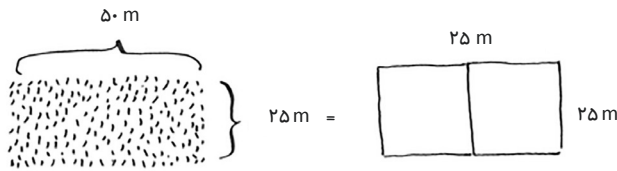
قطعه‌ها زیادی کوچک هستند



تمام قطعه‌ها باید هم‌اندازه باشند

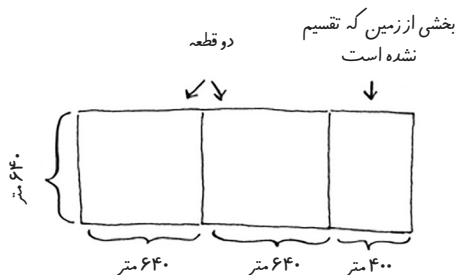
چگونه می‌توانید اندازه‌ی بزرگ‌ترین مربعی را که می‌توان برای یک قطعه زمین استفاده کرد، مشخص کنید؟ از استراتژی D&C استفاده کنید! الگوریتم‌های D&C الگوریتم‌های بازگشتی هستند. برای حل یک مشکل با استفاده از D&C، دو مرحله وجود دارد:

۱. صورت پایه را مشخص کنید. این باید ساده‌ترین حالت ممکن باشد.
۲. مسئله‌ی خود را تقسیم یا کوچک کنید تا زمانی که به صورت پایه تبدیل شود. بیایید از D&C برای یافتن راه حل این مسئله استفاده کنیم. اندازه‌ی بزرگ‌ترین مربعی که می‌توانید استفاده کنید چقدر است؟ ابتدا صورت پایه را در نظر بگیرید. ساده‌ترین حالت این است که یک طرف ضریبی از طرف دیگر باشد.

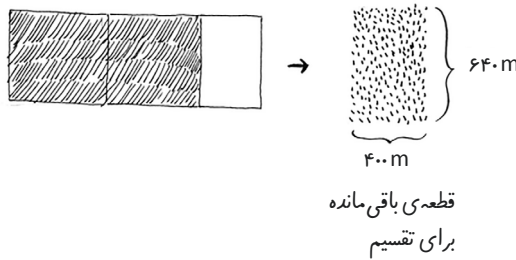


فرض کنید یک طرف ۲۵ متر و طرف دیگر ۵۰ متر باشد. پس بزرگ‌ترین مربعی که می‌توان استفاده کرد ۲۵ متر x ۲۵ متر است. برای تقسیم زمین به دو تا از آن مربع‌ها نیاز دارید.

اکنون باید صورت بازگشتی را پیدا کنید. اینجاست که D&C وارد می‌شود. بر اساس روش تقسیم و حل، با هر تماس بازگشتی، باید مسئله‌ی خود را کوچک کنید. چگونه مسئله را در اینجا کوچک می‌کنید؟ بیایید با علامت‌گذاری بزرگ‌ترین مربع‌های قابل استفاده شروع کنیم.



می‌توان دو مربع  $۶۴۰ \times ۶۴۰$  در آن جا داد، و هنوز مقداری زمین برای تقسیم باقی بماند. دیگر همه چیز مشخص است. یک بخش زمین برای تقسیم باقی مانده است. چرا برای این بخش هم الگوریتم مشابهی را اعمال نمی‌کنید؟

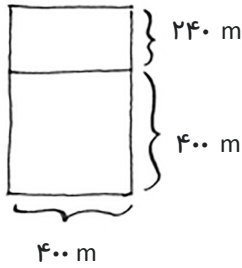


پس مسئله را با یک مزرعه‌ای با قطع  $۶۴۰ * ۱۶۸۰$  شروع کردید که باید تقسیم می‌شد. اما اکنون باید یک بخش کوچک‌تر،  $۴۰۰ \times ۶۴۰$  را تقسیم کنید. اگر بزرگ‌ترین مربعی را پیدا کنید که برای این اندازه مناسب است، آن بزرگ‌ترین مربعی است که برای کل مزرعه کارساز است. شما فقط ابعاد مسئله را از مزرعه‌ای به اندازه‌ی  $۶۴۰ \times ۱۶۸۰$  به مزرعه‌ای به اندازه‌ی  $۴۰۰ \times ۴۰۰$  کاهش داده‌اید!

### لگوریتم اقلیدسی<sup>۱</sup>

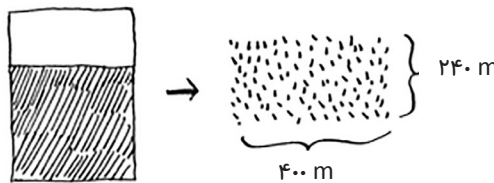
«اگر بزرگ‌ترین قطعه‌ای را که برای این اندازه مناسب است بیابید، به بزرگ‌ترین جعبه‌ای خواهد رسید که برای کل مزرعه کارساز باشد.» اگر دلیل درستی این جمله را متوجه نمی‌شوید، نگران نباشید. چون متأسفانه واضح نیست. اثبات این‌که چرا این‌گونه است مفصل‌تر از آن است که در این کتاب بیاید، بنابراین فقط باید به من باور داشته باشید که از این قاعده پیروی می‌کند. اگر می‌خواهید اثبات آن را بفهمید، الگوریتم اقلیدسی را جست‌وجو کنید. خان‌آکادمی در این آدرس توضیح خوبی ارائه کرده است:

<https://www.khanacademy.org/computing/computer-science/cryptography/modarithmetic/a/the-euclidean-algorithm>.

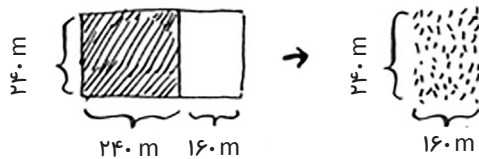


بیا بید دوباره همان الگوریتم را اعمال کنیم. با شروع یک مزرعه  $640 \times 400$  متر، بزرگ‌ترین مربعی که می‌توانید ایجاد کنید  $400 \times 400$  متر است.

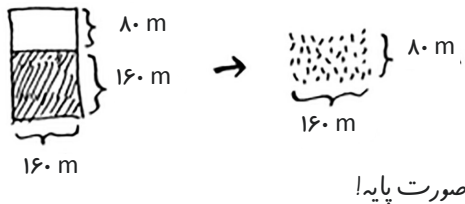
در این مرحله یک قطعه‌ی کوچک‌تر  $400 \times 240$  متری در اختیار شما قرار می‌گیرد.



و در آن می‌توانید یک مربع دیگر ترسیم کنید تا یک بخش کوچک‌تر به ابعاد  $240 \times 160$  متر به دست آورید.

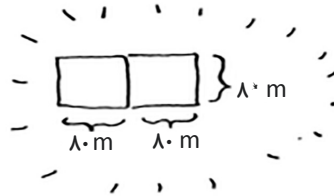


و سپس یک مربع بر روی آن بکشید تا یک بخش کوچک‌تر به دست آورید.

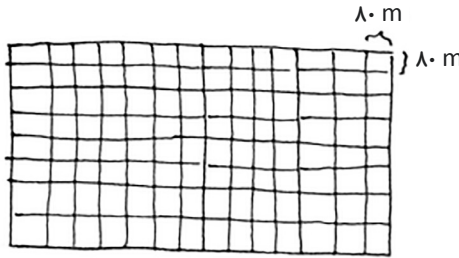


صورت پایه!

خب، شما در صورت پایه هستید:  $۸۰$  ضرب  $۱۶۰$  است. اگر این بخش را با استفاده از مربع‌ها تقسیم کنید، دیگر چیزی باقی نمی‌ماند!



بنابراین، برای قطعه زمین اصلی، بزرگ‌ترین قطعه‌ای که می‌توانید استفاده کنید  $۸۰ \times ۸۰$  متر است.



برای جمع بندی، در اینجا نحوه‌ی کار D&C آمده است:  
 ۱. یک صورت ساده را به عنوان صورت پایه در نظر بگیرید.  
 ۲. چگونه می‌توان مسئله‌ی خود را کوچک کرد و به صورت پایه رسید.  
 D&C یک الگوریتم ساده نیست که بتوانید آن را برای یک مسئله اعمال کنید. در عوض، راهی برای فکر کردن به راه حل یک مسئله است. بیایید یک مثال دیگر انجام دهیم.



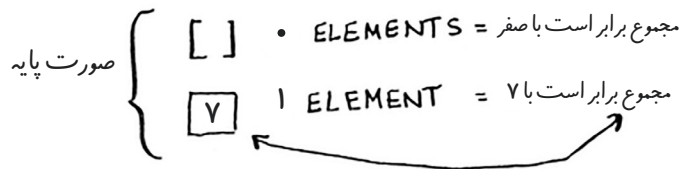
آرایه‌ای از اعداد به شما داده می‌شود.

باید همه‌ی اعداد را جمع زده و مجموع را برگردانید. انجام این کار به کمک یک حلقه خیلی ساده است:

```
def sum(arr):
    total = 0
    for x in arr:
        total += x
    return total
print sum([1, 2, 3, 4])
```

اما چگونه می‌توان این کار را با یک تابع بازگشتی انجام داد؟

**مرحله ۱:** صورت پایه را مشخص کنید. ساده‌ترین آرایه‌ای که می‌توان داشت چیست؟ ساده‌ترین صورت را در نظر گرفته و بعد ادامه بدهید. اگر آرایه‌ای با صفر یا یک عنصر داشته باشید، جمع آن بسیار آسان خواهد بود.



در نتیجه این صورت پایه خواهد بود.

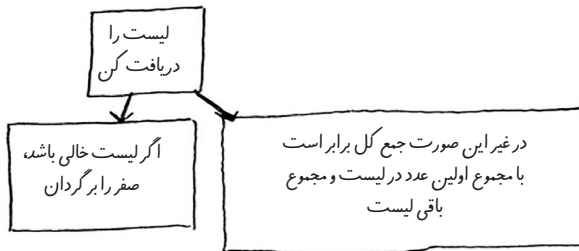
**مرحله ۲:** شما باید با هر تماس بازگشت به یک آرایه‌ی خالی نزدیک‌تر بشوید. چگونه ابعاد مسئله‌ی خود را کوچک می‌کنید؟ یک راه آن به شکل زیر است:

$$\text{Sum}(\boxed{2 \quad 4 \quad 6}) = 12$$

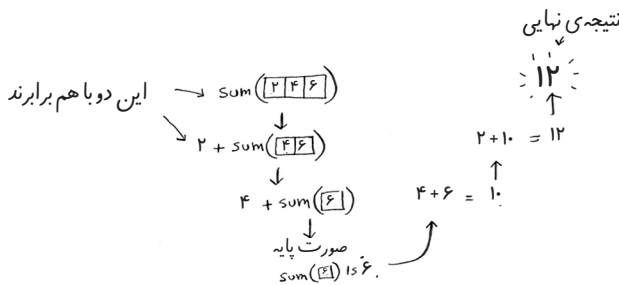
مثل این می‌ماند که بنویسیم:

$$2 + \text{SUM}(\boxed{4 \quad 6}) = 2 + 10 = 12$$

در هر صورت، نتیجه ۱۲ می‌شود. اما در نسخه‌ی دوم، آرایه‌ی کوچکتری را به تابع sum ارسال می‌کنید. یعنی ابعاد مسئله را کوچک کرده‌اید!  
تابع sum شما می‌تواند به این صورت عمل کند.



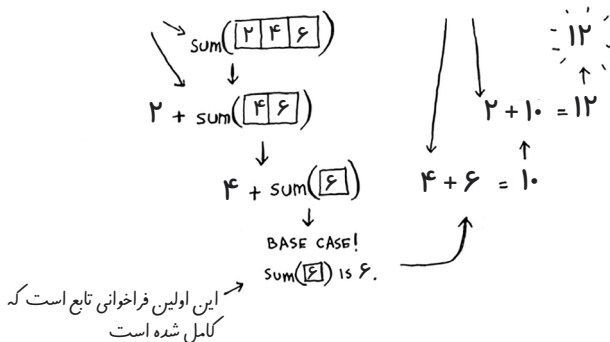
در پایین مراحل آن به تصویر درآمده است.



به یاد داشته باشید، تابع بازگشتی، موقعیت را دنبال می‌کند.

فراخوانی‌های تابع تا مرحله‌ی صورت پایه کامل نمی‌شوند.

به یاد داشته باشید که بازگشت جایگاه این فراخوانی یا تماس‌های تابع را در خود نگه می‌دارد.



این اولین فراخوانی تابع است که کامل شده است



### نکته

هنگامی که یک تابع بازگشتی می‌نویسید که شامل یک آرایه است، صورت پایه اغلب یک آرایه‌ی خالی یا آرایه‌ای با یک عنصر است. اگر نمی‌دانید چاره چیست، اول از همه آن را امتحان کنید.

### نگاهی به برنامه نویسی فانکشنال

شاید با خودتان فکر کنید «وقتی می‌توان مسئله را به راحتی با یک حلقه انجام داد، چرا باید این کار را به صورت بازگشتی انجام بدهم؟» خب، این نگاهی گذرا به برنامه‌نویسی فانکشنال است! زبان‌های برنامه‌نویسی فانکشنال مانند هسکل<sup>۱</sup> حلقه ندارند، بنابراین برای نوشتن توابعی مانند این باید از بازگشت استفاده کنید. اگر درک خوبی از بازگشت داشته باشید، یادگیری زبان‌های فانکشنال آسان‌تر خواهد بود. برای مثال، در اینجا نحوه‌ی نوشتن یک تابع `sum` در هسکل آمده است:

```
sum [] = 0          <----- صورت پایه
sum (x:xs) = x + (sum xs) <----- صورت بازگشتی
```

توجه داشته باشید که به نظر می‌رسد دو تعریف برای تابع دارید. اولین تعریف در صورت پایه و تعریف دوم در صورت بازگشتی اجرا می‌شود. همچنین می‌توانید این تابع را در هسکل با استفاده از دستور `if` بنویسید:

```
sum arr = if arr == []
           then 0
           else (head arr) + (sum (tail arr))
```

اما خواندن تعریف اول ساده‌تر است. به دلیل استفاده‌ی زیاد هسکل از توابع بازگشتی، ظرافت‌های فراوانی مانند این را در خود دارد تا آن را آسان‌تر کند. اگر به تابع بازگشتی علاقه دارید، یا علاقه‌مند به یادگیری یک زبان جدید هستید، نگاهی به زبان هسکل بیندازید.

## تمرین



۴.۱ کد را برای تابع `sum` قبلی بنویسید.

۴.۲ یک تابع بازگشتی برای شمارش تعداد آیت‌های یک لیست

بنویسید.

۴.۳ حداکثر عدد را در یک لیست بیابید.

۴.۴ جست‌وجوی دودویی را از فصل ۱ به خاطر دارید؟ این یک

الگوریتم تقسیم و حل نیز هست. آیا می‌توانید برای جست‌وجوی

دودویی، صورت پایه و صورت بازگشتی را پیدا کنید؟

## مرتب‌سازی سریع



مرتب‌سازی سریع یک الگوریتم مرتب‌سازی است. این الگوریتم بسیار

سریع‌تر از مرتب‌سازی انتخابی است و اغلب در پروژه‌های روزمره از

آن استفاده می‌شود. به عنوان مثال، کتابخانه‌ی استاندارد C تابعی

به نام `qsort` دارد که در واقع اجرای الگوریتم مرتب‌سازی سریع در

این زبان برنامه‌نویسی است. مرتب‌سازی سریع هم از D&C استفاده

می‌کند.

بیابید از مرتب‌سازی سریع برای مرتب کردن یک آرایه استفاده

کنیم. ساده‌ترین آرایه‌ای که یک الگوریتم مرتب‌سازی می‌تواند از

عهده‌ی آن بریاید چیست (توصیه‌ای را که در بخش قبل کردم به خاطر دارید)؟ خب،

برخی از آرایه‌ها اصلاً نیازی به مرتب‌سازی ندارند.

نیازی به مرتب‌سازی آرایه‌ها  
به این شکل نیست.

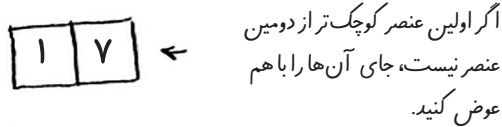
$$\left\{ \begin{array}{l} [ ] \leftarrow \text{آرایه‌ی خالی} \\ [ ۲۰ ] \leftarrow \text{آرایه‌ای با یک عنصر} \end{array} \right.$$

آرایه‌ها و آرایه‌های خالی تنها با یک عنصر، صورت پایه خواهند بود. شما فقط می‌توانید

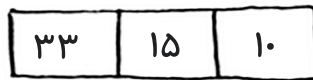
آن آرایه‌ها را به همان شکل که هستند برگردانید - چیزی برای مرتب‌سازی وجود ندارد:

```
def quicksort(array):
    if len(array) < 2:
        return array
```

بیا ببینیم به آرایه‌های بزرگ‌تر نگاه بیندازیم. مرتب‌سازی آرایه‌ای با دو عنصر بسیار آسان است.



آرایه‌ای با سه عنصر چطور؟



به یاد داشته باشید، شما از D&C استفاده می‌کنید. بنابراین می‌خواهید این آرایه را تا زمانی که در صورت پایه قرار بگیرید، تجزیه کنید. در اینجا نحوه‌ی کار الگوریتم مرتب‌سازی سریع آمده است. ابتدا یک عنصر از آرایه را انتخاب کنید. این عنصر محور نامیده می‌شود.

بعداً در مورد چگونگی انتخاب یک محور مناسب صحبت خواهیم کرد. در حال حاضر، فرض کنید اولین آیت‌م در آرایه محور است. حالا عناصر کوچک‌تر از محور و عناصر بزرگ‌تر از محور را پیدا کنید.



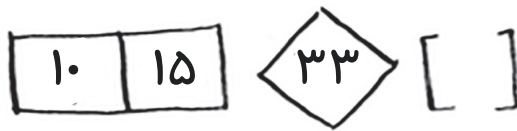
به این کار پارتیشن بندی<sup>۱</sup> می گویند. حالا شما موارد زیر را در اختیار دارید:

• زیرآرایه ای<sup>۲</sup> از همه ی اعداد کوچک تر از محور

• محور

• یک زیرآرایه از تمام اعداد بزرگ تر از محور

این دو زیرآرایه مرتب نشده اند. آن ها فقط پارتیشن بندی شده اند اما اگر مرتب شده باشند، مرتب سازی کل آرایه بسیار آسان خواهد بود.



اگر زیرآرایه ها مرتب شده اند، می توانید کل آن را به این شکل ترکیب کنید: آرایه ی سمت راست + محور + آرایه ی سمت چپ - و یک آرایه مرتب شده به دست آورید. در این نمونه آرایه ی مرتب شده ی زیر را خواهیم داشت:

$$[10, 15] + [33] + [] = [10, 15, 33]$$

چگونه زیر آرایه ها را مرتب می کنید؟ مرتب سازی سریع صورت پایه از قبل می داند که چگونه آرایه های دو عنصر (زیرآرایه ی سمت چپ) و آرایه های خالی (زیرآرایه سمت راست) را مرتب کند. بنابراین اگر مرتب سازی سریع را روی دو زیرآرایه فراخوانی کنید و سپس نتایج را ترکیب کنید، یک آرایه ی مرتب شده به دست می آورید!

```
quicksort([15, 10]) + [33] + quicksort([])
```

```
> [10, 15, 33]
```

با انتخاب هر عنصر دیگری به عنوان محور، نتیجه همچنان حاصل می شود. فرض

کنید این بار ۱۵ را به عنوان محور انتخاب کنید.



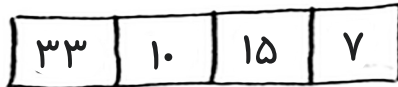
هر دو زیرآرایه فقط یک عنصر دارند و شما می دانید که چگونه باید آن ها را مرتب کنید. بنابراین اکنون می دانید که چگونه یک آرایه از سه عنصر را مرتب کنید. مراحل به این شکل است:

1. partitioning
2. Sub-array

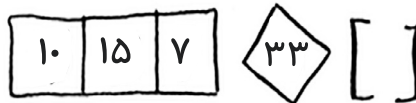
۱. یک محور را انتخاب کنید.

۲. آرایه را به دو آرایه‌ی فرعی تقسیم کنید: عناصر کوچک‌تر از محور و عناصر بزرگ‌تر از محور.

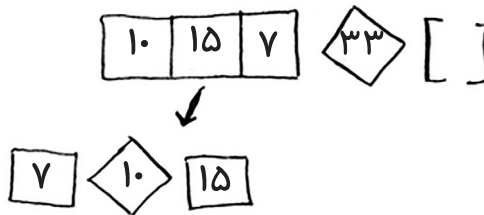
۳. مرتب‌سازی سریع را به صورت بازگشتی در دو آرایه‌ی فرعی فراخوانی کنید. آرایه‌ای از چهار عنصر چطور؟



فرض کنید دوباره ۳۳ را به عنوان محور انتخاب کنید.



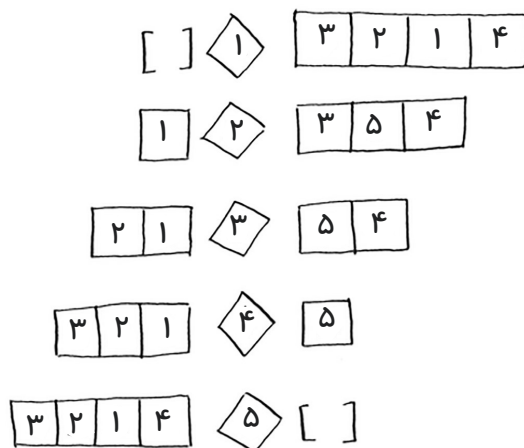
آرایه‌ی سمت چپ دارای سه عنصر است. از پیش می‌دانید که چگونه یک آرایه از سه عنصر را مرتب کنید: مرتب‌سازی سریع بر روی آن را به شکل بازگشتی فراخوانی کنید.



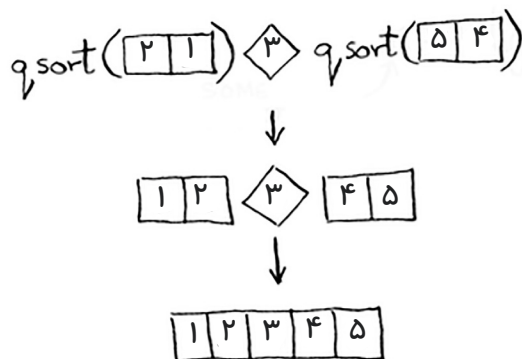
بنابراین می‌توانید یک آرایه از چهار عنصر را مرتب کنید. و اگر می‌توانید یک آرایه از چهار عنصر را مرتب کنید، می‌توانید یک آرایه از پنج عنصر را مرتب کنید. چرا؟ فرض کنید این آرایه‌ی پنج عنصری را دارید.



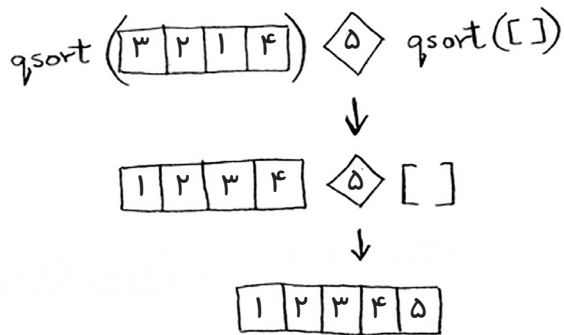
در اینجا همه‌ی روش‌هایی که می‌توانید این آرایه را تقسیم‌بندی کنید، بسته به محوری که انتخاب می‌کنید، آورده شده است.



توجه داشته باشید که تمام این زیرآرایه‌ها بین ۰ تا ۴ عنصر دارند. و شما از پیش می‌دانید که چگونه یک آرایه از ۰ تا ۴ عنصر را با استفاده از الگوریتم مرتب‌سازی سریع مرتب کنید! بنابراین مهم نیست که چه محوری را انتخاب می‌کنید، می‌توانید مرتب‌سازی سریع را به صورت بازگشتی در دو زیرآرایه فراخوانی کنید. برای مثال، فرض کنید عدد ۳ را به عنوان محور انتخاب کنید. شما مرتب‌سازی سریع را در زیر آرایه‌ها فرا می‌خوانید:



زیرآرایه‌ها مرتب می‌شوند و سپس تمام آن‌ها را با هم ترکیب می‌کنید تا یک آرایه‌ی مرتب‌شده به دست بیاورید. حتی اگر عدد ۵ را به عنوان محور انتخاب کنید، باز هم نتیجه حاصل می‌شود.



این روش با هر عنصری به عنوان محور جواب می‌دهد. بنابراین می‌توانید یک آرایه از پنج عنصر را مرتب کنید. با همین منطق، می‌توانید یک آرایه از شش عنصر و غیره را مرتب کنید.

## برهان‌های استقرایی<sup>۱</sup>

همین الان به برهان‌های استقرایی نیم‌نگاهی داشتید! اثبات‌های استقرایی یکی از راه‌های اثبات کارایی الگوریتم شما هستند. هر اثبات استقرایی دو مرحله دارد: صورت پایه و صورت استقرایی. آشنا به نظر می‌رسند؟ به عنوان مثال، فرض کنید می‌خواهم ثابت کنم که می‌توانم تا بالای یک نردبان بروم. در حالت استقرایی، اگر پاهایم روی یک پله باشد، می‌توانم پاهایم را روی پله‌ی بعدی بگذارم. بنابراین اگر در پله‌ی ۲ هستم، می‌توانم به پله‌ی ۳ بروم. این حالت استقرایی است. برای صورت پایه، می‌گویم که پاهایم روی پله‌ی ۱ قرار گرفته‌اند. بنابراین می‌توانم با یک پله یک پله بالا رفتن، به بالای نردبان بروم. شما از استدلال مشابهی برای مرتب‌سازی سریع استفاده می‌کنید. در صورت پایه، نشان دادم که الگوریتم برای صورت پایه جواب می‌دهد: آرایه‌هایی با اندازه‌ی ۰ و ۱. در صورت استقرایی، نشان دادم که اگر مرتب‌سازی سریع برای آرایه‌ای با اندازه‌ی ۱ مناسب باشد، برای آرایه‌ای با اندازه‌ی ۲ هم مناسب خواهد بود. و اگر برای آرایه‌ای با اندازه‌ی ۲ نتیجه‌بخش باشد، برای آرایه‌های با اندازه‌ی ۳ و اندازه‌های دیگر هم نتیجه‌بخش است. پس می‌توانم بگویم که مرتب‌سازی سریع برای همه‌ی آرایه‌ها با هر اندازه‌ای مناسب است. من در اینجا بیش از این و با جزئیات بیشتر به اثبات‌های استقرایی نمی‌پردازم، اما آن‌ها جالب هستند و با D&C ارتباط تنگاتنگی دارند

کد مرتب‌سازی سریع در ادامه آمده است:

```
def quicksort(array):
```

```
    if len(array) < 2:
```

```
        return array
```

صورت پایه: آرایه‌هایی با صفر یا یک عنصر «مرتب» شده‌اند.

```
    else :
```

```
        pivot = array[0]
```

صورت بازگشتی

```
        less = [i for i in array[1:] if i <= pivot]
```

زیر آرایه‌ای از تمامی عناصر کمتر از محور (pivot)



### 1. Inductive proofs

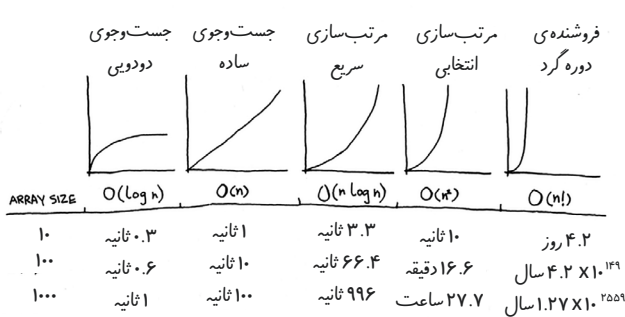


```

greater = [i for i in array[1:] if i > pivot] ← زیر آرایه‌ای از تمامی عناصر
return quicksort(less) + [pivot] + quicksort(greater)    بزرگ‌تر از محور
print quicksort([10, 5, 2, 3])
    
```

## بازبینی نماد O بزرگ

مرتب‌سازی سریع منحصر به فرد است زیرا سرعت آن به محوری که انتخاب می‌کنید بستگی دارد. قبل از این که در مورد مرتب‌سازی سریع صحبت کنم، اجازه دهید دوباره به رایج‌ترین زمان‌های اجرای O بزرگ نگاه بیندازیم.



تخمین‌ها بر اساس کامپیوتری با سرعت کم است که ۱۰ عملیات در ثانیه انجام می‌دهد.

مثال‌های زمانی این نمودار تخمینی با این فرض هستند که ۱۰ عملیات در ثانیه انجام بدهید. این نمودارها دقیق نیستند و تنها برای این است که به شما درکی بدهد که این زمان‌های اجرا تا چه اندازه متفاوت از یکدیگر هستند. در حقیقت، کامپیوتر شما می‌تواند بیش از ۱۰ عملیات در ثانیه انجام دهد.

هر زمان اجرا یک الگوریتم نمونه دارد. مرتب‌سازی انتخابی را که در فصل ۲ یاد گرفتید، بررسی کنید. زمان اجرای آن  $O(n^2)$  است. این الگوریتمی بسیار کند است. الگوریتم مرتب‌سازی دیگری به نام مرتب‌سازی ادغامی است که  $O(n \log n)$  است. خیلی سریع‌تر! مرتب‌سازی سریع یک نمونه‌ی چالش برانگیز است. در بدترین حالت، مرتب‌سازی سریع زمان  $O(n^2)$  خواهد داشت. به اندازه‌ی مرتب‌سازی انتخابی کند است! البته برای بدترین حالت است. در حالت

متوسط، مرتب‌سازی سریع به زمان  $O(n \log n)$  نیاز دارد. بنابراین ممکن است با خودتان فکر کنید:

- بدترین حالت<sup>۱</sup> و حالت متوسط<sup>۲</sup> در اینجا به چه معناست؟
- اگر مرتب‌سازی سریع به طور متوسط  $O(n \log n)$  است، اما مرتب‌سازی ادغامی همیشه  $O(n \log n)$  است، چرا از مرتب‌سازی ادغامی استفاده نکنید؟ مگر سریع‌تر نیست؟

## مرتب‌سازی ادغامی در برابر مرتب‌سازی سریع

فرض کنید این تابع ساده را برای چاپ هر آیتم در یک لیست دارید:

```
def print_items(list):
    for item in list:
        print item
```

این تابع تمام موارد موجود در لیست را مرور کرده و آن را چاپ می‌کند. از آنجا که یک بار روی کل لیست حلقه می‌زند، این تابع در زمان  $O(n)$  اجرا می‌شود. حالا، فرض کنید این تابع را طوری تغییر دهید که قبل از چاپ یک آیتم، ۱ ثانیه توقف کند:

```
from time import sleep
def print_items2(list):
    for item in list:
        sleep(1)
        print item
```

قبل از اینکه یک آیتم را پرینت کند، ۱ ثانیه مکث می‌کند. فرض کنید لیستی از پنج آیتم را با استفاده از هر دو تابع پرینت می‌کنید.

```
2 4 6 8 10
```

```
print_items: 2 4 6 8 10
```

```
print_items2: <sleep> 2 <sleep> 4 <sleep> 6 <sleep> 8 <sleep> 10
```

1. Worst case
2. Average case

هر دو تابع یک بار در لیست حلقه می‌زنند، بنابراین هر دو زمان  $O(n)$  هستند. کدامیک در عمل سریع‌تر خواهد بود؟ فکر می‌کنم `print_items` بسیار سریع‌تر خواهد بود، چون قبل از پرینت یک آیت، مکث ۱ ثانیه‌ای ندارد. بنابراین باوجود این که هر دو تابع در نماد  $O$  بزرگ سرعت یکسانی دارند، `print_items` در عمل سریع‌تر است. وقتی نماد  $O$  بزرگ را مانند  $O(n)$  می‌نویسید، در واقع به این معنی است:

$$c * n$$

↑  
یک زمان ثابت

$c$  مقدار ثابتی از زمانی است که الگوریتم شما طول می‌کشد. به آن ثابت<sup>۱</sup> می‌گویند. برای مثال، ممکن است  $10 * n$  milliseconds برای `print_items` در مقابل  $1 \text{ second} * n$  برای `print_items2` باشد. معمولاً آن ثابت را نادیده می‌گیرید، زیرا اگر دو الگوریتم زمان‌های  $O$  بزرگ متفاوتی داشته باشند، ثابت اهمیتی ندارد. برای مثال جست‌وجوی دودویی و جست‌وجوی ساده را در نظر بگیرید. فرض کنید هر دو الگوریتم این ثابت‌ها را داشته باشند.

$10 \text{ ms} * n$	$1 \text{ sec} * \log n$
جست‌وجوی ساده	جست‌وجوی دودویی

ممکن است بگویید: «چه جالب! ثابت جست‌وجوی ساده ۱۰ میلی‌ثانیه است، اما جست‌وجوی دودویی ثابت ۱ ثانیه دارد. پس جست‌وجوی ساده خیلی سریع‌تر است!» حال فرض کنید در حال جست‌وجو در میان لیستی از ۴ میلیارد عنصر هستید. زمان آن به این شکل است:

جست‌وجوی ساده	}	$\times$	۱۰ میلی‌ثانیه	=	۴ میلیارد	=	۴۶۳ روز
جست‌وجوی دودویی		$\times$	۱ ثانیه	=	۳۲	=	۳۲ ثانیه

1. constant

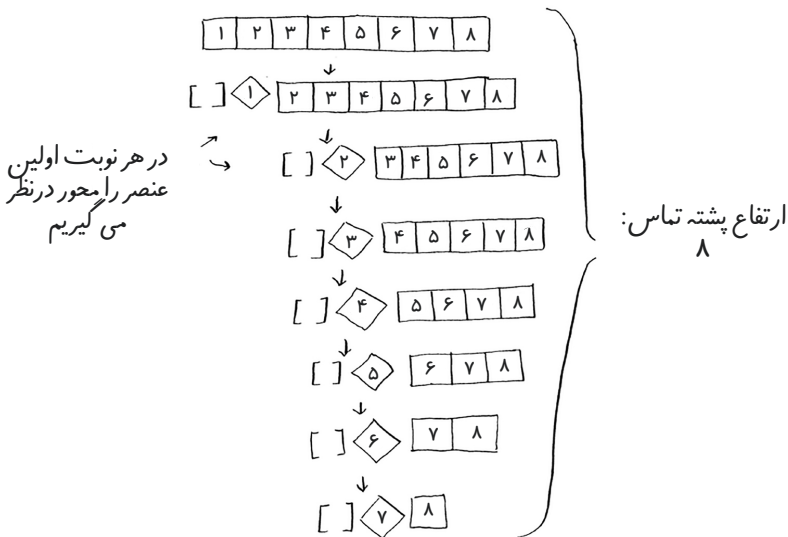
همان طور که می بینید، جست و جوی دودویی همچنان بسیار سریع تر است. آن ثابت اصلاً تفاوتی ایجاد نکرد.

اما گاهی اوقات ثابت می تواند تفاوت ایجاد کند. مرتب سازی سریع در مقابل مرتب سازی ادغامی تنها یک مثال است. مرتب سازی سریع ثابت کوچک تری نسبت به مرتب سازی ادغامی دارد. بنابراین اگر هر دو در زمان  $O(n \log n)$  باشند، مرتب سازی سریع، پرسرعت تر است. مرتب سازی سریع در عمل هم سریع تر است، زیرا اغلب اوقات در حالت متوسط و نه بدترین حالت خواهد بود.

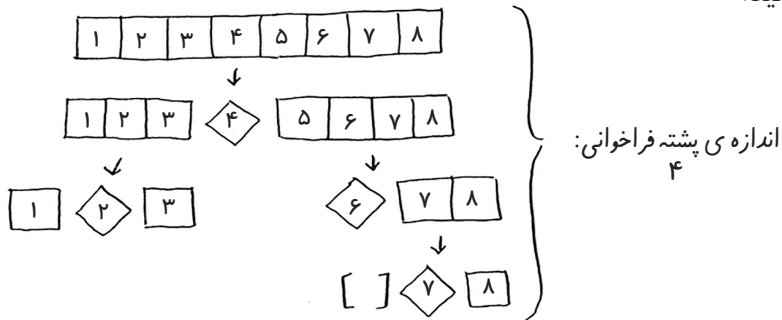
با خودتان فکر می کنید: حالت متوسط در مقابل بدترین حالت چیست؟

### حالت متوسط در مقابل بدترین حالت

عملکرد مرتب سازی سریع به مقدار زیادی به محوری که انتخاب می کنید بستگی دارد. فرض کنید همیشه اولین عنصر را به عنوان محور انتخاب می کنید. و شما مرتب سازی سریع را با آرایه ای که از پیش مرتب شده است فراخوانی می کنید. مرتب سازی سریع بررسی نمی کند که آیا آرایه ی ورودی از قبل مرتب شده است یا خیر. بنابراین همچنان سعی خواهد کرد آن را مرتب کند.



توجه کنید که آرایه به دو نیمه تقسیم نمی‌شود. در عوض، یکی از زیرآرایه‌ها همیشه خالی است. بنابراین پشته‌فراخوانی بسیار طولانی است. حالا در عوض، فرض کنید همیشه عنصر میانی را به عنوان محور انتخاب می‌کنید. اکنون به پشته‌فراخوانی نگاه کنید.

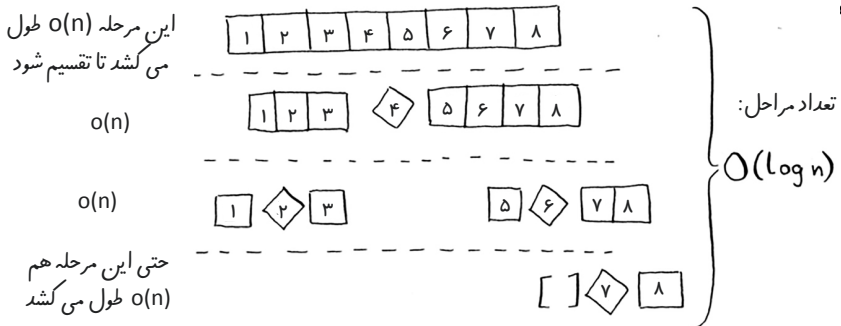


خیلی کوتاه شد! از آنجایی که هر بار آرایه را به نصف تقسیم می‌کنید، نیازی به برقراری تماس‌های بازگشتی زیادی ندارید. شما زودتر به حالت پایه می‌رسید و پشته‌فراخوانی بسیار کوتاه‌تر است.

مثال اول بدترین سناریو و مثال دوم بهترین سناریو است. در بدترین حالت، اندازه‌ی پشته  $O(n)$  است. در بهترین حالت، اندازه‌ی پشته  $O(\log n)$  است.

اکنون به سطح اول پشته نگاه کنید. شما یک عنصر را به عنوان محور انتخاب می‌کنید و باقی عناصر به زیرآرایه تقسیم می‌شوند. شما تمام هشت عنصر آرایه را بررسی می‌کنید. در نتیجه اولین عملیات به زمان  $O(n)$  نیاز دارد. تمام هشت عنصر را در این سطح از پشته‌فراخوانی مرور کردید. اما در واقع، عناصر  $O(n)$  را در هر سطح از پشته‌فراخوانی بررسی می‌کنید.





در این مثال، سطوح  $O(\log n)$  وجود دارد (به عبارت فنی «ارتفاع پشته‌فراخوانی  $O(\log n)$  است.»). و هر سطح به زمان  $O(n)$  نیاز دارد. کل الگوریتم زمان  $O(n \log n) = O(n) * O(\log n)$  نیاز دارد. این سناریو برای بهترین حالت است. در بدترین حالت،  $O(n)$  مرحله وجود دارد، بنابراین الگوریتم زمان  $O(n^2) = O(n) * O(n)$  خواهد داشت.

یک خبر خوب! بهترین حالت همان حالت متوسط است. اگر همیشه یک عنصر تصادفی در آرایه را به عنوان محور انتخاب کنید، مرتب‌سازی سریع به طور متوسط در زمان  $O(n \log n)$  تکمیل می‌شود. مرتب‌سازی سریع یکی از سریع‌ترین الگوریتم‌های مرتب‌سازی موجود است و نمونه‌ی بسیار خوبی از D&C است.

## تمرین

هر یک از این عملیات بر اساس نماد  $O$  بزرگ چقدر طول می‌کشد؟

۴.۵ چاپ مقدار هر عنصر در یک آرایه.

۴.۶ دو برابر کردن مقدار هر عنصر در یک آرایه.

۴.۷ دو برابر کردن مقدار فقط اولین عنصر در یک آرایه.

۴.۸ ایجاد جدول ضرب با تمام عناصر موجود در آرایه. بنابراین اگر آرایه‌ی شما  $[2, 3, 7, 8, 10]$  باشد، ابتدا هر عنصر را در ۲ ضرب کنید، سپس هر عنصر را در ۳، بعد در ۷ و غیره ضرب کنید.

## جمع‌بندی

• D&C با تجزیه‌ی یک مسئله به قطعات کوچک‌تر و کوچک‌تر کار می‌کند. اگر از D&C در یک لیست استفاده می‌کنید، احتمالاً صورت پایه یک آرایه‌ی خالی یا یک آرایه با یک عنصر است.

• اگر مرتب‌سازی سریع را اجرا می‌کنید، یک عنصر تصادفی را به عنوان محور انتخاب کنید. میانگین زمان اجرای مرتب‌سازی سریع  $O(n \log n)$  است!

• ثابت در نماد  $O$  بزرگ می‌تواند گاهی اوقات مهم باشد. به همین دلیل است که مرتب‌سازی سریع، سریع‌تر از مرتب‌سازی ادغام است.

• ثابت‌ها تقریباً هرگز در مقایسه‌ی میان جست‌وجوی ساده و جست‌وجوی دودویی مهم نیست، چون زمانه که لیست شما بزرگ می‌شود  $O(\log n)$  بسیار سریع‌تر از  $O(n)$  است.





## ۵ | جدول‌های هش



در این فصل

- در مورد جدول‌های هش، یکی از مفیدترین ساختمان‌های داده، یاد می‌گیرید. جدول‌های هش کاربردهای زیادی دارند. در این فصل به موارد رایج استفاده‌ی آن پرداخته می‌شود.

- در مورد اجزای داخلی جدول‌های هش: پیاده‌سازی، تصادم‌ها<sup>۱</sup> و توابع هش یاد می‌گیرید. این به شما کمک می‌کند تا نحوه‌ی تجزیه و تحلیل عملکرد جدول هش را درک کنید.



فرض کنید در یک فروشگاه مواد غذایی کار می‌کنید. وقتی مشتری محصولی را خریداری می‌کند، باید قیمتش را در یک کتابچه جست‌وجو کنید. اگر این کتابچه بر اساس حروف الفبا مرتب نشده باشد، ممکن است زمان زیادی طول بکشد تا تک‌تک سطرها را جست‌وجو کنید تا سیب را پیدا کنید. شما می‌توانید مطابق فصل ۱ در جایی که باید هر خط را نگاه کنید، از جست‌وجوی ساده استفاده کنید، به خاطر دارید چه مقدار طول می‌کشید؟ زمان  $O(n)$ . اگر کتاب بر اساس حروف الفبا باشد، می‌توانید از جست‌وجوی دودویی برای یافتن قیمت یک سیب استفاده کنید. این فقط به زمان  $O(\log n)$  نیاز دارد.

1. implementation
2. collisions

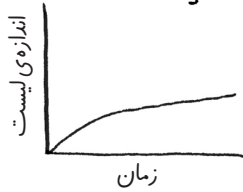
تخم مرغ ...	۲.۴۹ ₺
شیر ....	۱.۹۹ ₺
گلایه ....	۷۹ ₺

لیست مرتب شده

گلایه ..	۷۹ ₺
تخم مرغ ...	۲.۴۹ ₺
شیر ....	۱.۹۹ ₺

لیست مرتب نشده

$O(\log n)$



$O(n)$



محض یادآوری، تفاوت زیادی میان زمان  $O(n)$  و  $O(\log n)$  وجود دارد! فرض کنید می‌توان ۱۰ خط از کتاب را در هر ثانیه نگاه کرد. در ادامه زمان لازم برای جست‌وجوی ساده و جست‌وجوی دودویی آمده است:

آیتم‌های داخل کتاب	$O(n)$	$O(\log n)$	تعداد خطوط لازم برای بررسی کردن:
۱۰۰	۱۰ ثانیه	۱ ثانیه ←	$\log_2 100 = 7$ خط
۱۰۰۰	۱.۶۶ دقیقه	۱ ثانیه ←	$\log_2 1000 = 10$ خط
۱۰۰۰۰	۱۶.۶ دقیقه	۲ ثانیه ←	$\log_2 10000 = 14$ خط = ۲ ثانیه

از پیش می‌دانید که جست‌وجوی دودویی بسیار سریع است. اما به عنوان یک صندوق‌دار، جست‌وجو در تمام یک کتابچه مکافات است، حتی اگر محتوای کتابچه مرتب شده باشد. می‌توانید حال و روز بی‌اعصاب مشتری را هنگام جست‌وجوی آیتم‌ها در کتاب تصور کنید. چیزی که واقعاً به آن نیاز دارید، ریفیقی است که تمامی اسم‌ها و قیمت‌ها را حفظ کرده باشد. در نتیجه نیاز نخواهد بود چیزی را جست‌وجو کنید: از او می‌پرسید، و او بلافاصله پاسخ را به شما می‌گوید.



رفیق شما مگی می‌تواند قیمت هر آیتمی را در زمان  $O(1)$  به شما بگوید، مهم نیست کتابچه چقدر حجیم باشد. او حتی از جست‌وجوی دودویی هم سریع‌تر است.

# آیتم‌های داخل کتاب	جست‌وجوی ساده	جست‌وجوی دودویی	مگی
	$O(n)$	$O(\log n)$	$O(1)$
۱۰۰	۱۰ ثانیه	۱ ثانیه	درجا
۱۰۰۰	۱.۶ دقیقه	۱ ثانیه	درجا
۱۰۰۰۰	۱۶.۶ دقیقه	۲ ثانیه	درجا

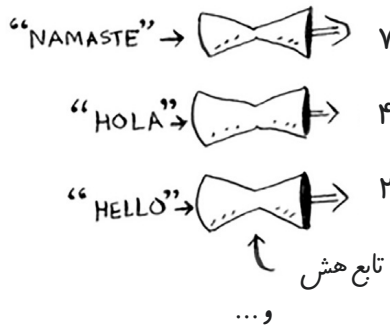
چه انسان نازنینی! چگونه یک «مگی» داشته باشیم؟  
 بیاید عینک ساختمان داده‌ای خود را به صورت بزنیم. شما تاکنون دو ساختمان داده می‌شناسید: آرایه و لیست (در مورد پشته‌ها صحبت نمی‌کنم چون نمی‌توانید چیزی را در یک پشته «جست‌وجو» کنید). این کتابچه را می‌توانید به صورت یک آرایه پیاده‌سازی کنید.

( ۰.۷۹، گلابی )	( ۱.۴۹، شیر )	( ۲.۴۹، تخم‌مرغ )
-----------------	---------------	-------------------

هر آیتم در آرایه در واقع دو آیتم است: یکی نام یک محصول و دیگری قیمت آن است. اگر این آرایه را بر اساس نام مرتب کنید، می‌توانید جست‌وجوی دودویی را روی آن اجرا کنید تا قیمت یک آیتم را بیابید. بنابراین می‌توانید آیتم‌ها را در زمان  $O(\log n)$  بیابید. اما می‌خواهید آیتم‌ها را در زمان  $O(1)$  پیدا کنید. یعنی می‌خواهید یک «مگی» بسازید. اینجاست که توابع هش از راه می‌رسند.

## توابع هش

تابع هش تابعی است که در آن یک رشته قرار می‌دهید و آن یک عدد را به شما برمی‌گرداند.

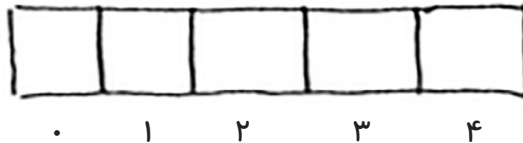


در اصطلاح فنی، ما می‌گوییم که یک تابع هش «رشته‌ها را به اعداد نگاشت» می‌کند. ممکن است فکر کنید هیچ الگوی قابل فهمی برای عددی که با قرار دادن یک رشته به دست می‌آورید وجود ندارد. اما برخی از الزامات برای یک تابع هش وجود دارد:

- باید اصولی و یکپارچه باشد. به عنوان مثال، فرض کنید «سیب» را وارد کرده و «۴» را برگرداند. هر بار که «سیب» را وارد می‌کنید، باید «۴» را دریافت کنید. غیر از این باشد، جدول هش شما کار نخواهد کرد.

- باید کلمات مختلف را به اعداد مختلف نگاشت کند. به عنوان مثال، یک تابع هش اگر همیشه برای هر کلمه‌ای که وارد می‌کنید «۱» را برگرداند فایده ندارد. در بهترین حالت، هر کلمه‌ی متفاوت باید به عدد متفاوتی نگاشت شود.

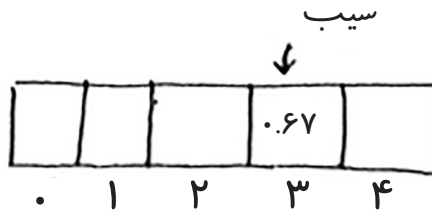
بنابراین یک تابع هش رشته ها را به اعداد نگاشت می کند. به چه کار می آید؟ خب، شما می توانید از آن برای خلق «مگی» خود استفاده کنید!  
با یک آرایه ی خالی شروع کنید:



شما تمام قیمت های مورد نظر خود را در این آرایه ذخیره می کنید. بیایید قیمت یک سیب را اضافه کنیم. «سیب» را به تابع هش وارد کنید.



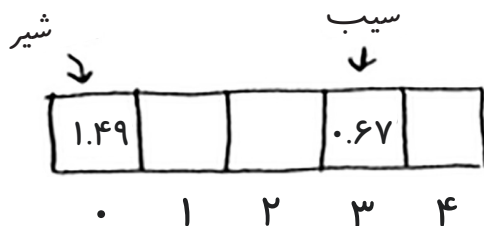
خروجی تابع هش «۳» خواهد بود. پس بیایید قیمت یک سیب را در ایندکس ۳ آرایه ذخیره کنیم.



اجازه بدهید شیر را هم اضافه کنیم. به تابع هش «شیر» بدهید.



تابع هش «۰» را ارائه می‌کند. بیایید قیمت شیر را در ایندکس صفر ذخیره کنیم.



ادامه دهید، و در نهایت کل آرایه از قیمت‌ها پر خواهد شد.

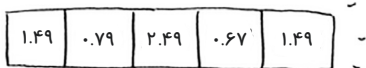


حالا می‌پرسید: «هی، آووکادو چنده؟» شما نیازی به جست‌وجوی آن در آرایه ندارید. فقط «آووکادو» را وارد تابع هش می‌کنید.



این به شما می‌گوید که قیمت در ایندکس ۴ ذخیره می‌شود. و همان‌طور که انتظار می‌رفت نتیجه را مشاهده می‌کنید.

آووکادو = ۱.۴۹



تابع هش به شما می‌گوید که قیمت دقیقاً در چه نقطه‌ای ذخیره شده است، بنابراین اصلاً نیازی به جست‌وجو نخواهید داشت! چنین چیزی با توجه به موارد زیر امکان‌پذیر است:

• تابع هش به طور پیوسته یک نام را به یک ایندکس یکسان نگاشت می‌کند. هر بار که «آووکادو» را وارد می‌کنید، همان عدد را برمی‌گرداند. بنابراین می‌توانید برای اولین بار از آن برای یافتن محل ذخیره‌ی قیمت آووکادو استفاده کنید و بعد می‌توانید از آن برای پیدا کردن جایی که آن قیمت را ذخیره کرده‌اید، استفاده کنید.

• تابع هش رشته‌های مختلف را به ایندکس‌های مختلف نگاشت می‌کند. «آووکادو» به ایندکس ۴ نگاشت می‌شود. «شیر» به ایندکس صفر نگاشت می‌شود. همه چیز به یک روزنه‌ی متفاوت در آرایه که می‌توانید قیمت آن را ذخیره کنید نگاشت می‌شود.

• تابع هش می‌داند که آرایه‌ی شما چه اندازه‌ای دارد و فقط ایندکس‌های معتبر<sup>۱</sup> را برمی‌گرداند. بنابراین اگر آرایه‌ی شما ۵ آیتم باشد، تابع هش ۱۰۰ را بر نمی‌گرداند ... چون ۱۰۰ یک ایندکس معتبر در آرایه نیست.

خب شما یک «مگی» ساختید! یک تابع هش و یک آرایه را کنار هم قرار دهید و ساختمان داده‌ای به نام جدول هش دریافت می‌کنید. جدول هش اولین ساختمان داده با منطق مضاعف است که یاد می‌گیرید. آرایه‌ها و لیست‌ها مستقیماً به حافظه نگاشت می‌شوند، اما جدول‌های هش هوشمندتر هستند. آن‌ها از یک تابع هش برای تشخیص هوشمندانه‌ی محل ذخیره‌ی عناصر استفاده می‌کنند.

جدول‌های هش احتمالاً مفیدترین ساختمان داده‌ی پیچیده‌ای هستند که یاد می‌گیرید. آن‌ها همچنین به عنوان hash maps، maps، دیکشنری‌ها و آرایه‌های ربطی یا انجمنی<sup>۲</sup> شناخته می‌شوند. و نکته‌ی دیگر اینکه جدول‌های هش سریع هستند! بحث ما در مورد آرایه‌ها و لیست‌های پیوندی در فصل ۲ را به خاطر دارید؟ شما می‌توانید یک آیتم را فوراً از یک آرایه دریافت کنید. و جدول‌های هش از یک آرایه برای ذخیره‌ی داده‌ها استفاده می‌کنند، بنابراین جداول هش به همان اندازه سریع هستند.

احتمالاً هرگز مجبور نخواهید شد که جدول‌های هش را خودتان پیاده‌سازی کنید. هر زبان خوب جدول هش پیاده‌سازی شده‌ی خود را دارد. پایتون هم دارای جدول هش است و به آن‌ها دیکشنری می‌گویند. با استفاده از تابع dict می‌توانید یک جدول هش جدید بسازید:

1. valid
2. associative arrays

```
>>> book = dict()
```



یک جدول هش  
خالی



جدول هش  
قیمت محصولات

book یک جدول هش جدید است. بیایید چند قیمت  
به book اضافه کنیم:

```
>>> book["apple"] = 0.67 ←----- قیمت یک سیب
>>> book["milk"] = 1.49 ←----- قیمت شیر
>>> book["avocado"] = 1.49
>>> print book
{'avocado': 1.49, 'apple': 0.67, 'milk': 1.49}
```

به همین سادگی! حالا بیایید قیمت آووکادو را بررسی کنیم:

```
>>> print book["avocado"]
1.49 ←----- قیمت آووکادو
```

جدول‌های هش دارای کلیدها و مقادیر<sup>۲</sup> است. در هش book، نام محصولات کلید، و قیمت آن‌ها مقادیرند. جدول هش کلیدها را به مقدار آن‌ها نگاشت می‌کند. در بخش بعدی، نمونه‌هایی را مشاهده خواهید کرد که جدول هش برای آن‌ها بسیار مناسب است.

1. keys
2. values



## تمرین

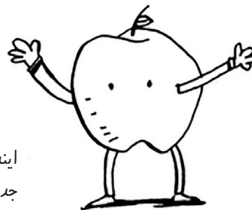
برای توابع هش مهم است که به طور مداوم همان خروجی را برای ورودی یکسان برگردانند. اگر این کار را نکنند، بعد از اینکه آیتم خود را در جدول هش قرار دادید، نمی توانید آن را پیدا کنید! کدام یک از این توابع هش معتبر هستند؟

5.1  $f(x) = 1$  ← ..... برای تمامی ورودی ها «۱» را برمی گرداند

5.2  $f(x) = \text{rand}()$  ← ..... هر دفعه یک عدد تصادفی برمی گرداند

5.3  $f(x) = \text{next\_empty\_slot}()$  ← ..... ایندکس اسلات خالی بعدی جدول هش را برمی گرداند

5.4  $f(x) = \text{len}(x)$  ← ..... از طول رشته به عنوان ایندکس استفاده می کند



## موارد کاربرد

جدول های هش در همه جا کاربرد دارد. این بخش چند مورد استفاده ی آن را به شما نشان می دهد.

### استفاده از جدول هش برای جست و جو

موبایل شما یک دفترچه تلفن دم دستی و به درد بخور دارد. همراه هر نام یک شماره تلفن هست.

BADE MAMA → ۵۸۱ ۶۶۰ ۹۸۲۰

ALEX MANNING → ۴۸۴ ۲۳۴ ۴۶۸۰

JANE MARIN → ۴۱۵ ۵۶۷ ۳۵۷۹



فرض کنید مطابق این شکل می خواهید یک دفترچه تلفن داشته باشید. شما در حال نگاشت نام افراد به شماره تلفن هستید. دفترچه تلفن شما باید این ویژگی ها را داشته باشد:

- نام یک فرد و شماره تلفن مرتبط با آن شخص را اضافه کنید.
  - نام شخص را وارد کنید و شماره تلفن مرتبط با آن نام را دریافت کنید.
- این یک کاربرد عالی برای جدول‌های هش است! جدول‌های هش برای موارد زیر فوق‌العاده هستند:

- ایجاد یک نگاشت از چیزی به چیز دیگر

- جست‌وجو به دنبال چیزی

ساخت دفترچه تلفن بسیار آسان است. ابتدا یک جدول هش جدید بسازید:

```
>>> phone_book = dict()
```

راستی، پایتون یک میان‌بر برای ساخت جدول هش دارد. می‌توانید از دو آکولاد استفاده کنید:

```
>>> phone_book = {} ← phone_book = dict() مشابه
```

بیا بید شماره تلفن برخی از افراد را به این دفترچه تلفن اضافه کنیم:

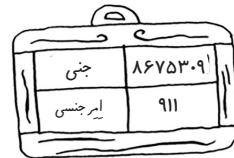
```
>>> phone_book["jenny"] = 8675309
```

```
>>> phone_book["emergency"] = 911
```

همین! حالا فرض کنید می‌خواهید شماره تلفن جنی را پیدا کنید. فقط کافی است کلید را به هش ارسال کنید:

```
>>> print phone_book["jenny"]
```

```
8675309 ← شماره تلفن جنی
```



یک جدول هش به عنوان دفترچه تلفن

تصور کنید اگر ناچار بودید این کار را با استفاده از یک آرایه انجام دهید. چگونه انجام

می دادید؟ جدول های هش مدل سازی رابطه از یک آیتم به دیگری را ساده می کند. جدول های هش برای جست و جو در مقیاس بسیار بزرگ تر استفاده می شود. به عنوان مثال، فرض کنید به وب سایتی مانند <http://adit.io> می روید. کامپیوتر شما باید [adit.io](http://adit.io) را به یک آدرس IP ترجمه کند.

ADIT.IO → 173.255.248.55

برای هر وب سایتی که بازدید می کنید، آدرس آن باید به یک آدرس IP ترجمه شود.

GOOGLE.COM → 74.125.239.133

FACEBOOK.COM → 173.252.128.6

SCRIBD.COM → 23.235.47.175

چه عالی، نگاشت یک آدرس وب به یک آدرس IP؟ به نظر می رسد یک کاربرد فوق العاده برای جدول های هش باشد! این فرآیند *DNS resolution* نامیده می شود. جدول های هش یکی از روش های دستیابی به این امکان است.



## جلوگیری از ورودی های تکراری

فرض کنید مسئول یکی از حوزه های اخذ رأی هستید. قاعداً هر فرد تنها یک بار می تواند رأی بدهد. چگونه مطمئن می شوید که افراد پیش تر رأی نداده اند؟ وقتی کسی برای رأی دهی وارد می شود، نام کامل او را می پرسید. سپس آن را با لیست افرادی که رأی داده اند چک می کنید.



اگر نام آن فرد در لیست باشد، این شخص قبلاً رأی داده است - او را از محل بیرون کنید! در غیر این صورت اسمش را به لیست اضافه می‌کنید و اجازه‌ی رأی دادن می‌دهید. حالا فرض کنید افراد زیادی برای رأی دهی آمده‌اند و لیست افرادی که رأی داده‌اند بسیار طولانی است.



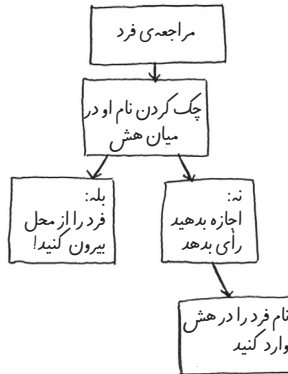
هر بار که فرد جدیدی برای رأی دادن وارد می‌شود، باید این فهرست بلندبالا را بررسی کنید تا ببینید آیا قبلاً رأی داده یا خیر. اما راه بهتری هم هست: از هش استفاده کنید! ابتدا یک هش برای افرادی که رأی داده‌اند ایجاد کنید:

```
>>> voted = {}
```

وقتی فردی جدید برای رأی دادن وارد می‌شود، بررسی کنید که آیا از قبل در هش فهرست شده است یا خیر:

```
>>> value = voted.get("tom")
```

اگر «tom» در جدول هش باشد، تابع `get` مقدار آن را برمی‌گرداند. در غیر این صورت، `None` را برمی‌گرداند. می‌توانید از این برای بررسی اینکه آیا کسی قبلاً رأی داده است یا خیر استفاده کنید!



کد آن به این شکل است:

```
voted = {}  
  
def check_voter(name):  
    if voted.get(name):  
        print("Kick them out");  
    else  
        voted[name] = True  
        print("let them vote!")
```

بیا بید چند بار آن را تست کنیم:

```
>>> check_voter("tom")  
!let them vote!  
>>> check_voter("mike")  
let them vote!  
>>> check_voter("mike")  
kick them out!
```

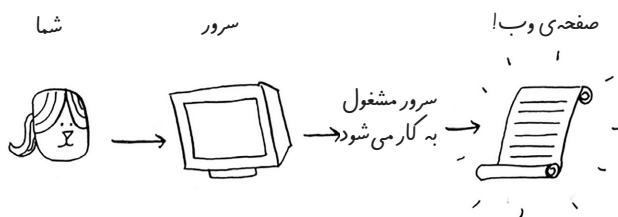
اولین باری که نام وارد می‌شود، این عبارت چاپ خواهد شد، "می‌تواند رأی دهد!" سپس مایک وارد می‌شود و چاپ می‌کند: «می‌تواند رأی دهد!» سپس مایک برای بار دوم تلاش می‌کند که برود، و چاپ می‌کند: «او را بیرون کنید!» به یاد داشته باشید، اگر این نام‌ها را در لیست افراد رأی‌داده ذخیره می‌کردید، این تابع در نهایت بسیار کند می‌شد، زیرا می‌بایست یک جست‌وجوی ساده در کل لیست انجام می‌داد. اما در عوض شما نام آن‌ها را در یک جدول هش ذخیره می‌کنید و یک جدول هش فوراً به شما می‌گوید که آیا نام این شخص در جدول هش وجود دارد یا خیر. بررسی موارد مشابه و تکراری با یک جدول هش بسیار سریع است.

## استفاده از جدول هش برای کش<sup>۱</sup>



و در نهایت کاربرد آن در کش کردن. اگر روی یک وبسایت کار می‌کنید، ممکن است قبلاً نام کش را به عنوان یک ویژگی خوب شنیده باشید. قضیه از این قرار است. فرض کنید از facebook.com بازدید می‌کنید:

۱. شما یک درخواست<sup>۲</sup> به سرور فیس بوک ارسال می‌کنید.
۲. سرور یک ثانیه فکر می‌کند و صفحه‌ی وب را برای شما ارسال می‌کند.
۳. شما یک صفحه‌ی وب دریافت می‌کنید.



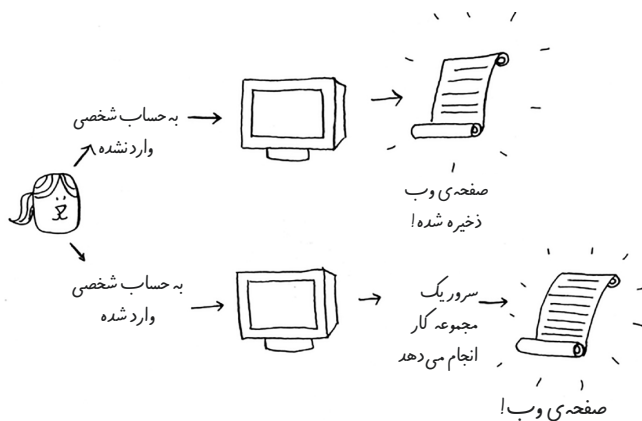
به عنوان مثال، در فیس بوک، سرور ممکن است تمام فعالیت‌های دوستان شما را جمع‌آوری کند تا به شما نشان بدهد. چند ثانیه طول می‌کشد تا تمام آن فعالیت‌ها جمع‌آوری شده و آن را به شما نشان بدهد. این چند ثانیه برای یک کاربر می‌تواند طولانی باشد. ممکن است فکر کنید، «چرا فیس بوک این قدر کند است؟» از طرف دیگر، سرورهای فیس بوک باید به میلیون‌ها نفر خدمات‌رسانی کنند و این چند ثانیه‌ها با هم جمع می‌شوند. سرورهای فیس بوک به سختی کار می‌کنند تا به همه‌ی این وبسایت‌ها سرویس دهند. آیا راهی برای سریع‌تر کردن فیس بوک و اینکه سرورهایش هم‌زمان کار کمتری انجام دهند وجود دارد؟

فرض کنید خواهرزاده‌ای دارید که مدام از شما در مورد سیارات سؤال می‌کند. «فاصله‌ی مریخ از زمین چقدره؟» «فاصله‌ی ماه چقدره؟» «فاصله‌ی مشتری چقدره؟» هر بار باید در گوگل سرچ کنید و جواب او را بدهید. چند دقیقه‌ای طول می‌کشد.

1. cache
2. request

حالا، فرض کنید دائم می‌پرسد: «فاصله‌ی ماه چقدره؟» خیلی زود، به یاد می‌آورید که ماه ۲۳۸۹۰۰ مایل دورتر است. نیازی نیست در گوگل دنبال جواب بگردید ... فقط به یاد می‌آورید و پاسخ می‌دهید. نحوه‌ی عملکرد کش به این صورت است: وب‌سایت‌ها به جای محاسبه‌ی مجدد داده، آن را به خاطر می‌آورند.

اگر در حساب کاربری خود در فیس‌بوک هستید، محتوایی که می‌بینید به شکل اختصاصی برای شما چیده شده است. هر بار که به [facebook.com](http://facebook.com) می‌روید، سرورهای آن باید به محتوای مورد علاقه‌ی شما فکر کنند. اما اگر حساب کاربری خود در فیس‌بوک نشده باشید، صفحه‌ی ورود را مشاهده می‌کنید. همه یک صفحه‌ی ورود واحد را می‌بینند. از فیس‌بوک بارها و بارها این درخواست یکسان ارسال می‌شود: «وقتی از حساب کاربری خارج شدم من را به صفحه‌ی اصلی منتقل کن.» بنابراین عوض آن‌که سرور را وادار کند تا شکل صفحه‌ی اصلی را بفهمد، شکل صفحه‌ی اصلی را در خاطر می‌سپارد و همان را برای شما ارسال می‌کند.



نام این فرآیند کش است. این کار دو مزیت دارد:

- صفحه‌ی وب را بسیار سریع‌تر دریافت می‌کنید، درست مانند زمانی که فاصله‌ی زمین تا ماه را به خاطر می‌سپارید. دفعه‌ی بعد که خواهرزاده‌ی شما سؤال کرد، نیازی نخواهد بود تا جواب آن را در گوگل جست‌وجو کنید. می‌توانید فوراً پاسخ دهید.

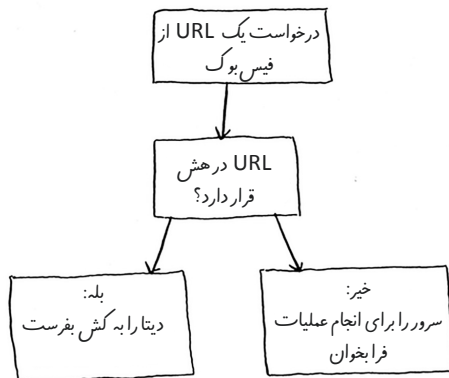
• از فیس بوک کمتر کار کشیده می شود.  
کش کردن یک راه رایج برای سرعت بخشیدن به کارها است. همه ی وبسایت های بزرگ از کش استفاده می کنند. و این داده ها در یک هش ذخیره می شوند!

فیس بوک فقط صفحه ی اصلی را کش نمی کند. صفحه ی درباره، صفحه ی تماس، صفحه ی شرایط و ضوابط و بسیاری موارد دیگر را هم در حافظه کش می کند. بنابراین لازم است از URL صفحه به داده های صفحه نگاشت کند.

facebook.com/about → DATA FOR THE ABOUT PAGE

facebook.com → DATA FOR THE HOME PAGE

هنگامی که از یک صفحه در فیس بوک بازدید می کنید، ابتدا بررسی می کند که آیا صفحه ی مورد نظر در هش ذخیره شده است یا خیر.



کد آن به شکل زیر است:

```

cache = {}
def get_page(url):
    if cache.get(url):
        return cache[url] ← دیتای کش شده را برمی گرداند
    
```



**else:**

```
data = get_data_from_server(url)
cache[url] = data ←----- ابتدا این داده را در کش ذخیره می‌کند
return data
```

در اینجا، تنها در صورتی از سرور کار می‌کشید که URL مورد نظر در کش موجود نباشد. با این حال، قبل از اینکه داده‌ها را برگردانید، آن‌ها را در کش ذخیره می‌کنید. دفعه‌ی بعد که شخصی این URL را درخواست کرد، به جای اینکه سرور را مجبور به انجام کار کنید، داده‌ها را می‌توانید از کش ارسال کنید.

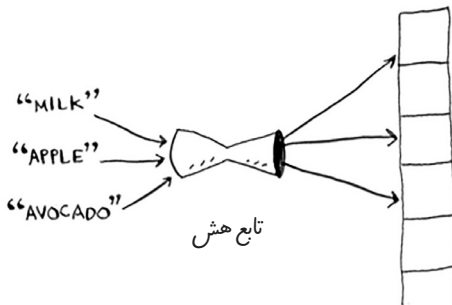
## جمع‌بندی

به طور خلاصه، هش برای این موارد مناسب هستند:

- مدل‌سازی رابطه‌ها از یک چیز به چیز دیگر
- فیلتر کردن موارد تکراری و مشابه
- کش کردن / به خاطر سپردن داده‌ها به جای اینکه سرور مشغول شود.

## تصادم‌ها

همان‌طور که قبلاً گفتم، بیشتر زبان‌ها دارای جدول هش هستند. نیازی نیست تا نوشتن آن را بلد باشید. بنابراین، چندان در مورد کدهای داخلی جدول‌های هش صحبت نمی‌کنم. اما کارایی آن همچنان برای شما اهمیت دارد! برای درک کارایی جدول‌های هش، ابتدا باید بدانید که

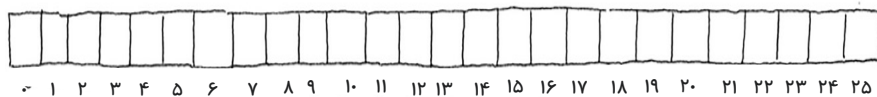


تصادم چیست. در دو بخش بعدی به تصادم‌ها و کارایی می‌پردازیم.

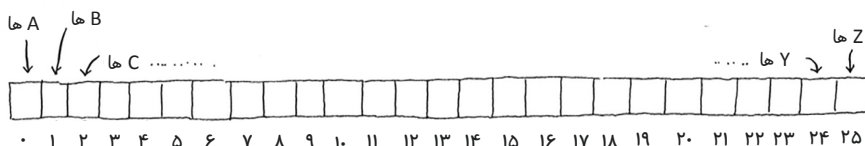
اول از همه اعتراف کنم که قبل‌تر به شما یک دروغ مصلحتی گفتم. به شما گفتم که یک تابع هش همیشه کلیدهای مختلف را به اسلات‌های مختلف در آرایه نگاشت می‌کند.

در واقعیت، نوشتن یک تابع هش که این کار را انجام بدهد تقریباً غیر ممکن است.

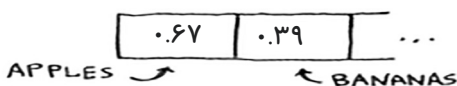
بیا یه یه مثال ساده بزنیم. فرض کنید آرایه‌ی شما ۲۶ اسلات دارد.



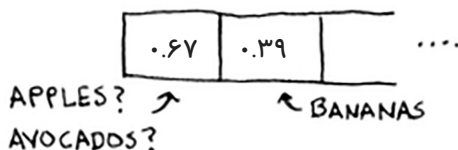
تابع هش شما بسیار ساده است: بر اساس حروف الفبا یک نقطه از آرایه را اختصاص می‌دهد.



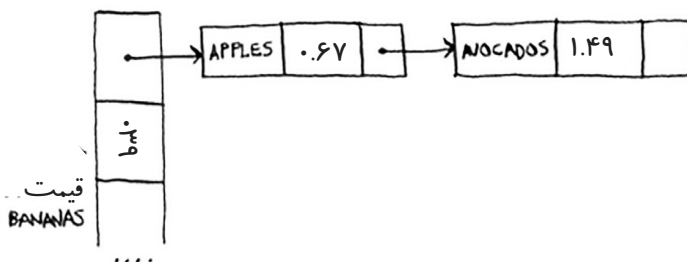
شاید متوجه مشکل شده باشید. قیمت سیب (apple) را می‌خواهید در هش وارد کنید. اولین روزنه به شما اختصاص داده می‌شود. سپس می‌خواهید قیمت موز (banana) را در هش قرار بدهید. روزنه‌ی دوم برای آن اختصاص می‌یابد.



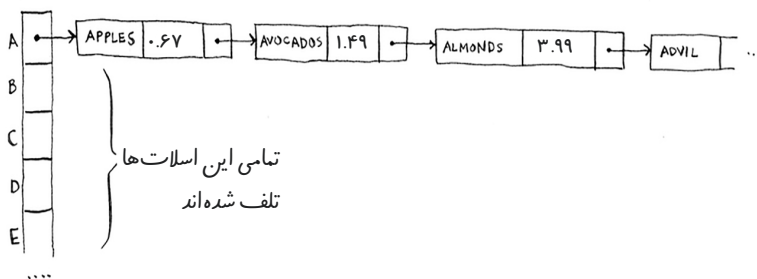
همه چیز به خوبی پیش می‌رود! اما اکنون می‌خواهید قیمت آووکادو (avocado) را در هش قرار بدهید. دوباره اولین روزنه به شما اختصاص داده می‌شود.



نه! سیب‌ها قبلاً آن اسلات را اشغال کرده‌اند! چه باید کرد؟ به این وضعیت تصادم می‌گویید: اختصاص دو کلید به یک اسلات. این یک مشکل است. اگر قیمت آووکادو را در آن جایگاه ذخیره کنید، قیمت سیب را بازنویسی می‌کنید. سپس نوبت بعدی که کسی قیمت سیب را می‌پرسد، به جای آن قیمت آووکادو را دریافت می‌کند! تصادم‌ها بد هستند و باید راه حل جایگزین برای آن در نظر گرفت. راه‌های مختلفی برای مقابله با تصادم وجود دارد. ساده‌ترین مورد این است: اگر چندین کلید به یک اسلات نگاشت می‌شوند، یک لیست پیوندی را در آن اسلات ایجاد کنید.



در این مثال، هر دو «apple» و «avocado» به یک اسلات نگاشت می‌شوند. بنابراین شما یک لیست پیوندی را در آن اسلات ایجاد می‌کنید. اگر بخواهید از قیمت موز با خبر شوید، همچنان سریع خواهد بود. اگر می‌خواهید قیمت سیب را بدانید، کمی کندتر است. برای یافتن «apple» باید از طریق این لیست پیوندی جست‌وجو کنید. اگر لیست پیوند داده‌شده کوچک باشد، مشکلی اساسی پیش نمی‌آید - باید سه یا چهار عنصر را جست‌وجو کنید. اما فرض کنید در یک فروشگاه مواد غذایی کار می‌کنید که در آن فقط محصولاتی را می‌فروشید که با حرف A شروع می‌شوند.



یک دقیقه صبر کنید! کل جدول هش به جز یک اسلات کاملاً خالی است. و آن اسلات یک لیست پیوندی بزرگی دارد! هر عنصر در این جدول هش در لیست پیوند داده شده است. این شرایط به همان اندازه بد است که برای شروع همه چیز را در یک لیست پیوندی قرار بدهید. جدول هش شما را کند می‌کند.

در اینجا به دو نکته پی می‌بریم:

• تابع هش شما بسیار مهم است. تابع هش همه‌ی کلیدها را در یک اسلات نگاشت کرده است. در حالت ایده‌آل، هش شما کلیدها را به طور یکنواخت در سراسر هش نگاشت می‌کند.

• اگر لیست‌های پیوندی بزرگ شوند، جدول هش شما بسیار کند می‌شود. اما اگر از یک تابع هش خوب استفاده کنید طول آن زیاد نخواهد شد!

توابع هش مهم هستند. در یک تابع هش خوب تصادم بسیار کمی رخ می‌دهد. پس چگونه یک تابع هش خوب را انتخاب می‌کنید؟ در بخش بعدی به این موضوع پرداخته می‌شود!

## کارایی

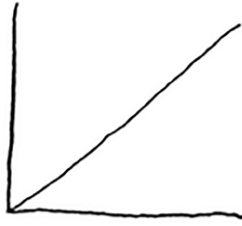
این فصل را در فروشگاه مواد غذایی شروع کردید. تصمیم داشتید سازوکاری ایجاد کنید که قیمت محصولات را فوراً به شما اعلام کند. خوب، جدول‌های هش واقعا سریع هستند. در حالت متوسط، جداول هش در همه‌ی شرایط زمان  $O(1)$  دارد.  $O(1)$  زمان ثابت

نامیده می‌شود. تا اینجا کار با زمان ثابت مواجه نشده‌اید. این اصطلاح به معنی فوری نیست بلکه به این معناست که صرف نظر از اینکه جدول هش چقدر بزرگ باشد زمان صرف شده ثابت می‌ماند. به عنوان مثال، می‌دانید که جست‌وجوی ساده زمان خطی دارد.

	حالت متوسط	بدترین حالت
جست‌وجو	$O(1)$	$O(n)$
درج	$O(1)$	$O(n)$
حذف	$O(1)$	$O(n)$

PERFORMANCE OF HASH TABLES

### 1. Constant time



$O(n)$   
زمان خطی  
(جست و جوی ساده)

جست و جوی دودویی با زمان لگاریتمی<sup>۱</sup> که دارد، سریع تر است:



$O(\log n)$   
زمان لگاریتمی  
(جست و جوی دودویی)

جست و جوی در جدول هش به زمان ثابت نیاز دارد.



$O(1)$   
زمان ثابت  
(جدول های هش)

می بینید که نمودار آن یک خط صاف است. به این معنی که فرقی نمی کند جدول هش شما ۱ عنصر داشته باشد یا ۱ میلیارد عنصر. دریافت هرچیز از جدول هش به زمان یکسانی نیاز دارد. در واقع، پیش از این زمان ثابت را دیده اید. خواندن یک آیتیم از آرایه زمان ثابتی نیاز دارد. مهم نیست که آرایه ی شما چقدر بزرگ باشد. برای به دست آوردن یک عنصر به زمان یکسانی نیاز است. در حالت متوسط، جداول هش واقعا سریع هستند.

در بدترین حالت، یک جدول هش برای همه چیز به  $O(n)$  - زمان خطی - نیاز دارد که واقعا کند است. بیایید جدول های هش را با آرایه ها و لیست ها مقایسه کنیم.

	لیست پیوندی	آرایه ها	جدول های هش (بدترین)	جدول های هش (متوسط)
جست و جو	$O(n)$	$O(1)$	$O(n)$	$O(1)$
درج	$O(1)$	$O(n)$	$O(n)$	$O(1)$
حذف	$O(1)$	$O(n)$	$O(n)$	$O(1)$

به حالت متوسط برای جدول های هش نگاه کنید. جدول های هش برای جست و جو به همان سرعت آرایه ها هستند (دریافت مقدار در یک ایندکس). و به همان سرعت لیست های پیوندی در درج و حذف هستند. خیر دنیا و آخرت! اما در بدترین حالت، جدول های هش در همه ی این موارد کند هستند. بنابراین مهم است که با استفاده از جدول های هش کارایی بدترین حالت را نداشته باشید. و برای انجام این کار، باید از تصادم جلوگیری کنید. برای جلوگیری از تصادم، به این دو مورد نیاز دارید:

- ضریب بار کم
- یک تابع هش خوب

## 1. load factor

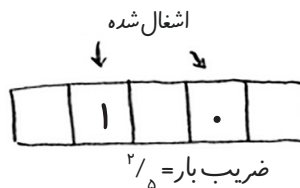
### نکته

پیش از آن‌که ادامه دهیم خوب است بدانید که خواندن این بخش ضروری نیست. در این قسمت می‌خواهم در مورد شیوه‌ی پیاده‌سازی جدول هش صحبت کنم، اما هیچ‌گاه نیاز پیدا نمی‌کنید که شخصاً کد آن را پیاده‌سازی کنید. از هر زبان برنامه‌نویسی که استفاده کنید، یک نسخه‌ی پیاده‌سازی شده از جدول‌های هش دارد. می‌توانید از جدول هش داخلی زبان برنامه‌نویسی استفاده کنید و مطمئن باشید که کارایی خوبی دارد. بخش بعدی برای آن است که از فرآیند اجرایی در پس‌زمینه آگاه شوید.

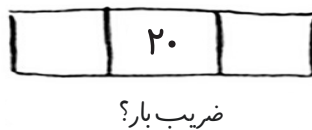
### ضریب بار (لود فاکتور)

محاسبه‌ی ضریب بار جدول هش آسان است. تعداد آیتم‌ها در جدول هش تقسیم بر تعداد کل اسلات‌ها.

جدول‌های هش از یک آرایه برای ذخیره‌سازی استفاده می‌کنند، به همین دلیل تعداد اسلات‌های اشغال شده در یک آرایه را می‌شمارید. به عنوان مثال، این جدول هش دارای ضریب بار  $2/5$  یا  $4$ .

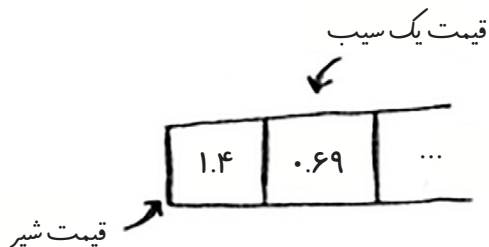


ضریب بار این جدول هش چیست؟

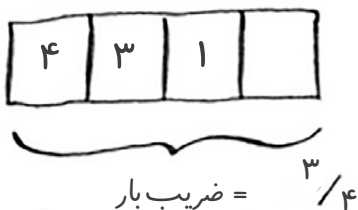


جواب صحیح  $1/3$  است. ضریب بار، تعداد اسلات‌های خالی در جدول هش را اندازه‌گیری می‌کند.

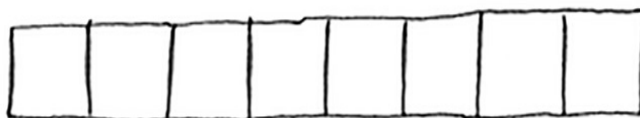
فرض کنید باید قیمت ۱۰۰ آیتم محصول را در جدول هش خود ذخیره کنید و جدول هش شما ۱۰۰ اسلات دارد. در بهترین حالت، هر آیتم جایگاه مخصوص به خود را خواهد داشت.



این جدول هشت دارای ضرب بار ۱ است. اگر جدول هشت شما فقط ۵۰ اسلات داشته باشد چه؟ ضرب بار ۲ خواهد داشت. هیچ راهی وجود ندارد که هر آیتیم اسلات مخصوص به خود را داشته باشد، زیرا اسلات‌های کافی وجود ندارد! داشتن ضرب بار بیشتر از ۱ به این معنی است که آیتیم‌های بیشتری نسبت به اسلات در آرایه‌ی خود دارید. هنگامی که ضرب بار شروع به رشد کند، باید اسلات‌های بیشتری را به جدول هشت خود اضافه کنید. به این عمل تغییر اندازه<sup>۱</sup> می‌گویند. برای مثال، فرض کنید جدول هشت زیر را دارید که در آستانه‌ی پر شدن است.



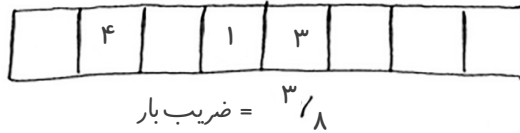
باید اندازه‌ی این جدول هشت را تغییر دهید. ابتدا یک آرایه‌ی جدید ایجاد می‌کنید که بزرگ‌تر باشد. قاعده‌ی سرانگشتی<sup>۲</sup> این است که آرایه‌ای با اندازه‌ی دوبرابر آن بسازید.



1. resizing
2. rule of thumb



اکنون باید تمامی آن آیتم‌ها را مجدداً با استفاده از تابع hash در جدول هش جدید وارد کنید:

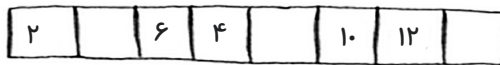


این جدول جدید دارای ضریب بار  $\frac{3}{8}$  است. خیلی بهتر از پیش! با ضریب بار کمتر، تصادم‌های کمتری خواهید داشت و جدول شما عملکرد بهتری خواهد داشت. یک قانون سرانگشتی خوب این است که وقتی ضریب بار شما بیشتر از  $0.7$  است، اندازه‌ی آن را تغییر دهید.

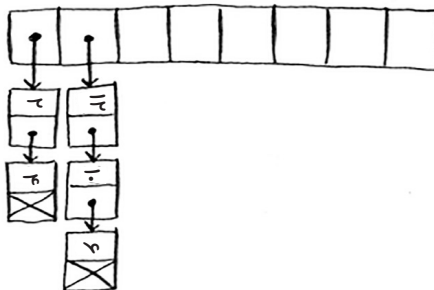
ممکن است فکر کنید، «این تغییر اندازه زمان‌بر است!» حق با شماست تغییر اندازه پرهزینه است و اغلب اوقات تمایل ندارید اندازه را تغییر دهید. اما به طور متوسط، جدول‌های هش حتی با تغییر اندازه، زمان  $O(1)$  دارند.

### تابع هش مناسب

تابع هش مناسب مقادیر را در آرایه به شکل مساوی توزیع می‌کند.



تابع هش بد مقادیر را با هم گروه‌بندی می‌کند و تصادم‌های بسیاری ایجاد می‌کند.



یک تابع هش خوب چه ویژگی‌هایی دارد؟ این مسئله هرگز دغدغه‌ی شما نخواهد بود - پیرمردهایی با ریش‌های بلند (و پیرزن‌هایی) در اتاق‌های تاریک می‌نشینند و دغدغه‌ی حل این مسائل را دارند. اگر همچنان کنجکاو هستید، تابع SHA را جست‌وجو کنید (توضیح کوتاهی درباره‌ی آن در فصل آخر وجود دارد). می‌توانید از آن به عنوان تابع هش خود استفاده کنید.

### تمرین

توابع هش باید توزیع خوبی داشته باشند. آن‌ها باید آیتیم‌ها را تا حد امکان هرچه گسترده‌تر نگاشت کنند. بدترین حالت برای یک تابع هش این است که همه‌ی آیتیم‌ها را در جدول هش به اسلات یکسانی نگاشت کند.

چهار تابع هش زیر را که با رشته‌ها کار می‌کنند، فرض کنید:

آ. «۱» را برای همه‌ی ورودی‌ها برگردانید.

ب. از طول رشته به عنوان ایندکس استفاده کنید.

پ. از اولین کاراکتر رشته به عنوان ایندکس استفاده کنید. بنابراین، تمام رشته‌هایی

که با  $a$  شروع می‌شوند با هم هش می‌شوند و غیره.

ت. هر حرف را با یک عدد اول نگاشت کنید:  $a = 2, b = 3, c = 5, d = 7, e = 11$  و

غیره. برای یک رشته، تابع هش مجموع تمام کاراکترهای باقی‌مانده‌ی اندازه‌ی هش

است. به عنوان مثال، اگر اندازه‌ی هش شما ۱۰ است و رشته‌ی آن «bag» است، ایندکس

$$۲ = ۱۰ \% ۲۲ = ۱۰ \% ۱۷ + ۲ + ۳ \text{ است.}$$

برای هر یک از این مثال‌ها، کدام تابع‌های هش توزیع خوبی ارائه می‌کنند؟ اندازه‌ی

جدول هش را با ۱۰ اسلات فرض کنید.

**۵.۵** دفترچه تلفنی که در آن کلیدها اسامی و مقادیر شماره‌ی تلفن هستند. اسامی

عبارتند از: Ben, Esther, Bob, و Dan.

**۵.۶** نگاهی از اندازه‌ی باتری به توان باتری. اندازه‌ها A, AA, AAA و AAAA هستند.

۵.۷ نگاشتی از عنوان کتاب به اسامی نویسنده‌ها. عناوین *Watchmen* و *Fun Home, Maus* هستند

هستند

## جمع‌بندی

تقریباً هرگز نیازی به پیاده‌سازی جدول هش پیدا نمی‌کنید. زبان برنامه‌نویسی که استفاده می‌کنید باید جدول هش پیاده‌سازی شده را برای شما فراهم کند. می‌توانید از جدول‌های هش پایتون استفاده کنید و فرض کنید که کارایی حالت میانگین را دریافت خواهید کرد: زمان ثابت.

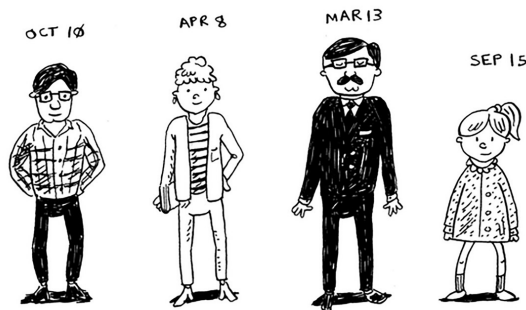
جدول هش یک ساختمان داده‌ی قدرتمند است زیرا بسیار سریع است و به شما امکان می‌دهند داده‌ها را به روشی متفاوت مدل‌سازی کنید. احتمالاً به زودی متوجه می‌شوید که همیشه در حال استفاده از آن‌ها بوده‌اید:

- می‌توانید با ترکیب یک تابع هش با یک آرایه یک جدول هش بسازید.
- تصادم‌ها بد هستند. شما به یک تابع هش نیاز دارید که تصادم‌ها را به حداقل برساند.

- جدول‌های هش در جست‌وجو، درج و حذف واقعاً سریع هستند.
- جدول‌های هش برای مدل‌سازی روابط از یک آیتم به آیتم دیگر مناسب هستند.
- هنگامی که ضریب بار شما از ۰.۷ بیشتر شد، زمان تغییر اندازه‌ی جدول هش فرا رسیده است.

• جدول‌های هش برای ذخیره‌سازی داده‌ها (به عنوان مثال، از طریق یک وب‌سرور) استفاده می‌شود.

- جدول‌های هش برای گرفتن موارد تکراری عالی هستند.





## ۶ | جست و جوی سطح اول



در این فصل

• یاد می‌گیرید یک شبکه را با استفاده از ساختمان داده‌ای جدید و انتزاعی مدل‌سازی کنید: گراف‌ها<sup>۱</sup>.

• جست و جوی سطح اول را یاد می‌گیرید، الگوریتمی که می‌توانید روی گراف‌ها اجرا کنید تا به سؤالاتی مانند «کوتاه‌ترین مسیر برای رفتن به X چیست؟» پاسخ دهید.

• در مورد گراف‌های جهت‌دار<sup>۲</sup> و گراف‌های غیرجهت‌دار<sup>۳</sup> یاد می‌گیرید.

• مرتب‌سازی توپولوژیکی<sup>۴</sup> را یاد می‌گیرید، نوع متفاوتی از الگوریتم مرتب‌سازی که وابستگی‌های بین گره‌ها را آشکار می‌کند.

در این فصل گراف‌ها معرفی می‌شوند. ابتدا در مورد اینکه گراف چیست صحبت می‌کنم (آن‌ها محور X یا Y ندارند). سپس اولین الگوریتم گرافی که یاد می‌گیرید را به شما نشان می‌دهم. این الگوریتم جست و جوی سطح اول (BFS) نام دارد.

1. Breadth-first search
2. graphs
3. Directed graph
4. Undirected graph
5. Topological sort

جست و جوی سطح اول به شما امکان می‌دهد کوتاه‌ترین فاصله‌ی میان دو چیز را پیدا کنید. اما کوتاه‌ترین فاصله می‌تواند معناهای گوناگونی داشته باشد! می‌توانید از جست و جوی سطح اول برای موارد زیر استفاده کنید:

- نوشتن کد هوش مصنوعی برای بازی چکرز که کمترین حرکت را برای پیروزی محاسبه کند.
- نوشتن یک تصحیح‌کننده‌ی غلط‌های املایی (کم‌ترین تغییر در حروف کلمه‌ای که غلط املایی دارد به یک کلمه‌ی درست برای مثال: READED -> READER)
- پیدا کردن نزدیک‌ترین پزشک

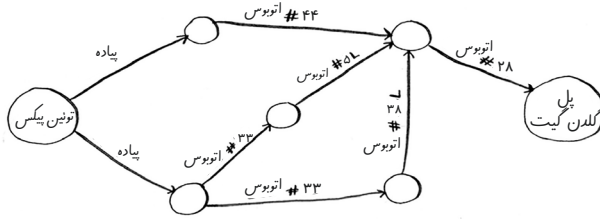
الگوریتم‌های گراف از مفیدترین الگوریتم‌هایی هستند که می‌شناسم. چند فصل بعدی را حتماً با دقت مطالعه کنید این‌ها الگوریتم‌هایی هستند که می‌توانید بارها و بارها از آن‌ها استفاده کنید.

## مقدمه‌ای بر گراف‌ها

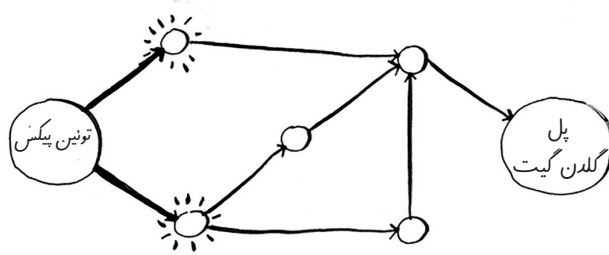


فرض کنید در سانفرانسیسکو هستید و می‌خواهید از توئین پیکس به پل گلدن گیت بروید. می‌خواهید با اتوبوس، با حداقل تغییر خط به مقصد برسید. گزینه‌های شما به شکل زیر است:

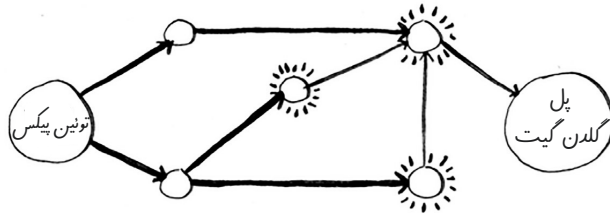
1. checkers



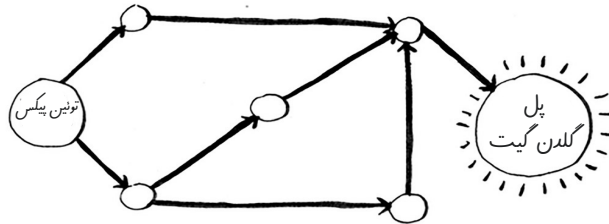
الگوریتم شما برای یافتن مسیر با کم‌ترین گام چیست؟  
 خب، آیا می‌توانید در تنها یک مرحله به آنجا برسید؟ در اینجا تمام مکان‌هایی  
 مشخص شده که می‌توانید در یک مرحله به آن‌ها برسید.



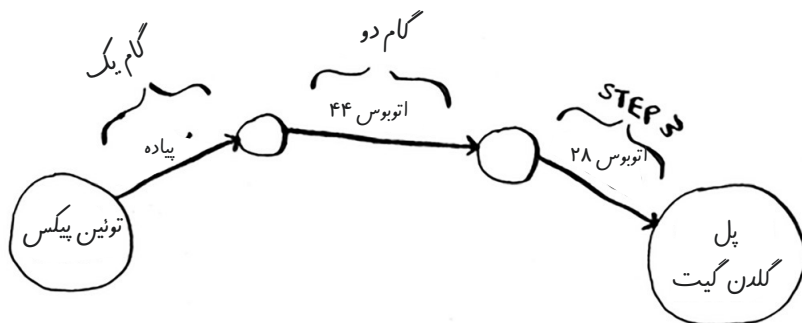
پل مورد نظر با علامت مشخص نشده است و نمی‌توانید در یک مرحله به آنجا  
 برسید. در دو مرحله چطور؟



باز هم، پل آنجا نیست، بنابراین نمی‌توانید در دو مرحله به پل برسید. سه مرحله  
 چطور؟



خب! بالاخره پل گلدن گیت نمایان می‌شود. بنابراین سه مرحله نیاز است تا مطابق این مسیر از توئین پیکس به مقصد برسید.



مسیرهای دیگری نیز وجود دارد که شما را به مقصد می‌رساند، اما طولانی‌تر هستند (چهار مرحله). الگوریتم تشخیص داد که کوتاه‌ترین مسیر به پل سه مرحله دارد. به این نوع مسئله، مسئله‌ی کوتاه‌ترین مسیر می‌گویند. همیشه در این نوع مسئله‌ها به دنبال کوتاه‌ترین چیز هستید. این می‌تواند کوتاه‌ترین مسیر برای رسیدن به خانه‌ی دوست شما باشد. یا می‌تواند کمترین تعداد حرکت برای مات کردن در یک بازی شطرنج باشد. الگوریتمی که برای حل یک مسئله کوتاه‌ترین مسیر انجام می‌شود، جست‌وجوی سطح اول نامیده می‌شود.

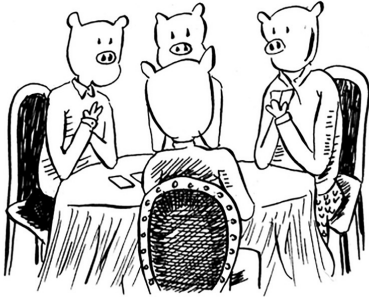
برای اینکه بفهمید چگونه از توئین پیکس به پل گلدن گیت بروید، به دو مرحله نیاز است:

۱. مسئله را به گراف مدل‌سازی کنید.
۲. مشکل را با استفاده از جست‌وجوی سطح اول حل کنید.

در ادامه به این می‌پردازیم که گراف چیست. سپس با جزئیات بیشتر وارد جست‌وجوی سطح اول خواهیم شد.

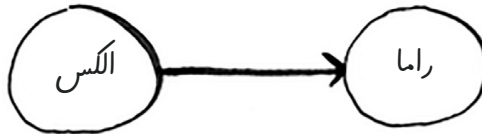
## 1. Shortest-path problem



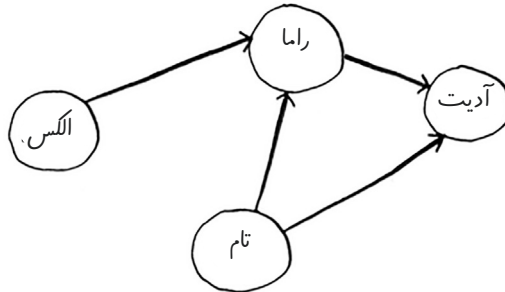


## گراف چیست؟

گراف مجموعه‌ای از اتصالات<sup>۱</sup> را مدل‌سازی می‌کند. برای مثال، فرض کنید شما و دوستانتان مشغول بازی پوکر هستید و می‌خواهید مدل‌سازی کنید که کی به کی بدهکار است. در اینجا می‌توانید بگویید: «الکس به راما بدهکار است.»



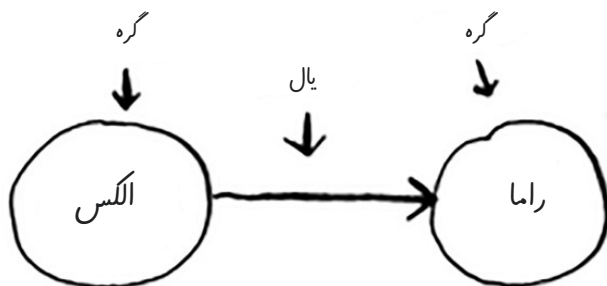
گراف کامل می‌تواند چیزی شبیه به این باشد.



گراف افرادی که به دیگران سر بازی پوکر پول بدهکار هستند.

الکس به راما بدهکار است، تام به آدیت بدهکار است و به همین ترتیب. هر گراف از گره‌ها<sup>۲</sup> و یال‌ها<sup>۳</sup> تشکیل شده است.

1. connections
2. nodes
3. edges



تمام اجزای سازنده‌ی آن همین است که می‌بینید! گراف‌ها از گره‌ها و یال‌ها تشکیل شده‌اند. یک گره می‌تواند مستقیماً به بسیاری از گره‌های دیگر متصل شود. آن گره‌ها همسایه‌های آن گره هستند. در این گراف، راما همسایه‌ی الکس است. آدیت همسایه‌ی الکس نیست، زیرا مستقیماً به همدیگر متصل نیستند. اما آدیت همسایه‌ی راما و تام است.

گراف‌ها روشی برای مدل‌سازی نحوه‌ی اتصال چیزهای مختلف به یکدیگر هستند. اکنون بیاید جست‌وجوی سطح اول را در عمل ببینیم.

## جست‌وجوی سطح اول

در فصل ۱ به یکی از الگوریتم‌های جست‌وجو نگاه کردیم: جست‌وجوی دودویی. جست‌وجوی سطح اول نوع متفاوتی از الگوریتم جست‌وجو است: الگوریتمی که بر روی گراف‌ها اجرا می‌شود. می‌تواند به ما در پاسخ دو دسته پرسش کمک کند:

- پرسش نوع ۱: آیا مسیری از گره‌ی A به گره‌ی B وجود دارد؟

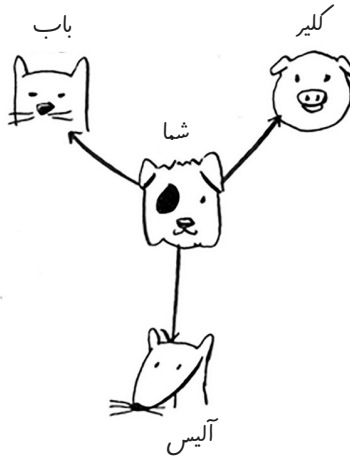
- پرسش نوع ۲: کوتاه‌ترین مسیر از گره‌ی A به گره‌ی B چیست؟

قبلاً زمانی که کوتاه‌ترین مسیر از توئین پیکس تا پل گلدن گیت را محاسبه می‌کردید، با جست‌وجوی سطح اول مواجه شدید. این یک پرسش از نوع دوم بود: «کوتاه‌ترین مسیر چیست؟» حالا بیایید الگوریتم را با جزئیات بیشتری بررسی کنیم. شما پرسشی از نوع

اولمی کنید: «آیا مسیری وجود دارد؟»

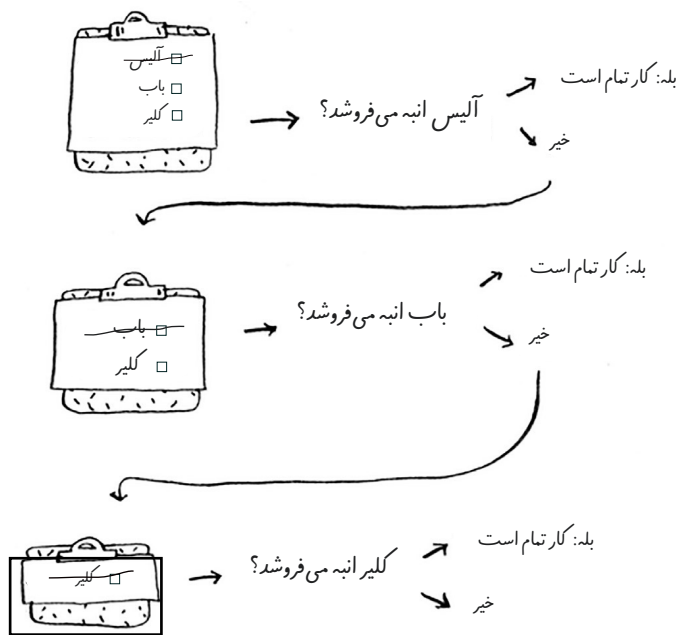


فرض کنید شما صاحب یک مزرعه‌ی انبه هستید و به دنبال فروشنده‌ای می‌گردید که بتواند انبه‌های شما را بفروشد. آیا به یک فروشنده‌ی انبه در فیس‌بوک ارتباط دارید؟ خوب، شما می‌توانید در میان دوستان خود جست‌وجو کنید.

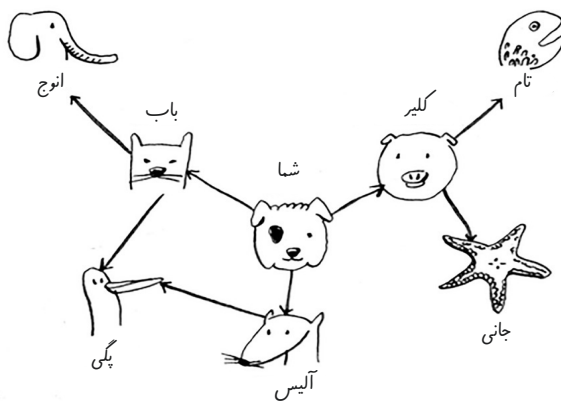


این جست‌وجوی بسیار ساده‌ای است. ابتدا لیستی از دوستان برای جست‌وجو تهیه کنید.

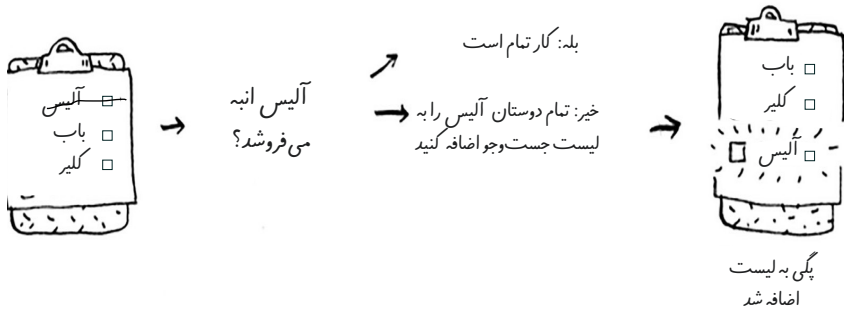
حالا به سراغ تک تک افراد لیست بروید و بررسی کنید که آیا آن شخص فروشنده‌ی انبه است یا خیر.



فرض کنید هیچ یک از دوستان شما فروشنده‌ی انبه نیست. اکنون باید از طریق دوستان دوستان خود جست‌وجو کنید.



هر بار که فردی از لیست را جست‌وجو می‌کنید، همه‌ی دوستان او را به این لیست اضافه می‌کنید.



به این ترتیب، نه تنها در میان دوستان خود، بلکه در میان دوستان این افراد نیز جست‌وجو می‌کنید. به یاد داشته باشید، هدف این است که یک فروشنده‌ی انبه در شبکه‌ی خود پیدا کنید. بنابراین اگر آلیس فروشنده‌ی انبه نیست، دوستان او را نیز به لیست اضافه کنید. این به این معنی است که شما در نهایت دوستان او را جست‌وجو خواهید کرد - و سپس دوستان این افراد و الی آخر. با این الگوریتم، تا زمانی که با فروشنده انبه روبه‌رو شوید، کل شبکه‌ی خود را جست‌وجو می‌کنید. این همان الگوریتم جست‌وجوی سطح اول است.

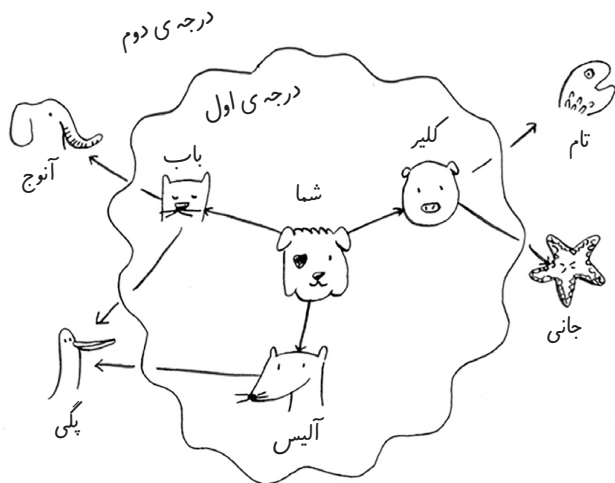
### یافتن کوتاه‌ترین مسیر

به طور خلاصه، این دو پرسشی است که جست‌وجوی سطح اول می‌تواند پاسخ بدهد:

- پرسش نوع ۱: آیا مسیری از گره‌ی A به گره‌ی B وجود دارد؟ (آیا فروشنده‌ی انبه در شبکه‌ی شما وجود دارد؟)

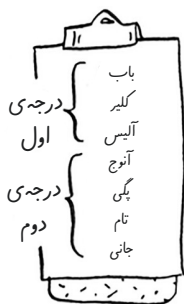
- پرسش نوع ۲: کوتاه‌ترین مسیر از گره‌ی A تا گره‌ی B چیست؟ (نزدیک‌ترین فروشنده‌ی انبه کیست؟)

نحوه‌ی پاسخ‌دادن به پرسش ۱ را دیدید. حالا بیاید تلاش کنیم به پرسش ۲ پاسخ دهیم. آیا می‌توانید نزدیک‌ترین فروشنده‌ی انبه را پیدا کنید؟ به عنوان مثال، دوستان شما رابطه‌ی درجه یک و دوستان آن‌ها رابطه‌ی درجه دو هستند.



ارتباط درجه اول را به ارتباط درجه دوم ترجیح می دهید و ارتباط درجه دوم را به ارتباط درجه سوم و غیره ترجیح می دهید. بنابراین، قبل از اینکه مطمئن شوید یک ارتباط درجه یک با فروشنده انبه ندارید، نباید هیچ ارتباط درجه دومی را جست و جو کنید. خوب، جست و جوی سطح اول این کار را انجام می دهد! روشی که جست و جوی سطح اول کار می کند، جست و جو از نقطه ی شروع به بیرون پخش می شود. بنابراین، ارتباطات درجه یک را قبل از ارتباطات درجه دوم بررسی می کنید. سؤال: کلر اول بررسی می شود یا آنوج؟ پاسخ: کلر یک ارتباط درجه یک است و آنوج یک ارتباط درجه دو است. بنابراین کلر قبل از آنوج بررسی می شود.

به عبارت دیگر ارتباطات درجه اول قبل از ارتباطات درجه دوم به لیست جست و جو اضافه می شوند.



شما فقط اسامی لیست را مرور می کنید و می بینید که آیا فروشنده ی انبه هستند یا خیر. ارتباطات درجه اول قبل از ارتباطات درجه دوم جست و جو می شوند، بنابراین نزدیک ترین انبه فروش را پیدا خواهید کرد. جست و جوی سطح اول نه تنها یک مسیر از A به B، بلکه کوتاه ترین مسیر را پیدا می کند.

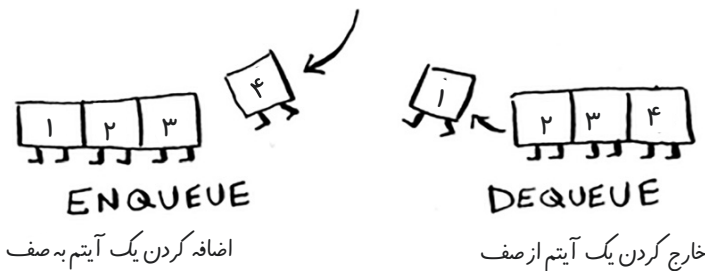
توجه داشته باشید که این شرایط تنها در صورتی صدق می‌کند که افراد را به همان ترتیبی که اضافه شده‌اند جست‌وجو کنید. یعنی اگر کلر قبل از آنوج به لیست اضافه شده باشد، باید او را قبل از آنوج جست‌وجو کنید. اگر آنوج را قبل از کلر جست‌وجو کنید و هر دو فروشنده‌ی انبه باشند، چه اتفاقی می‌افتد؟ خب، آنوج یک ارتباط درجه دو است و کلر یک ارتباط درجه یک. در نهایت با فروشنده‌ای مواجه می‌شوید که در شبکه‌ی شما نزدیک‌ترین فرد به شما نیست. بنابراین باید افراد را به ترتیبی که اضافه شده‌اند جست‌وجو کنید. یک ساختمان داده برای چنین موقعیتی وجود دارد: صف.

## صف<sup>۱</sup>



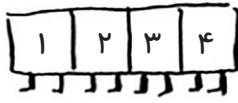
عملکرد صف دقیقاً مانند کاربرد آن در زندگی واقعی است. فرض کنید شما و دوستان در ایستگاه اتوبوس در صف ایستاده‌اید. اگر قبل از او در صف هستید، ابتدا سوار اتوبوس می‌شوید. یک صف به همین ترتیب کار می‌کند. صف‌ها شبیه به پشته‌ها هستند. شما نمی‌توانید به عناصر تصادفی در صف

دسترسی داشته باشید. در عوض، تنها دو عملیات وجود دارد، *enqueue* (به صف وارد کردن) و *dequeue* (از صف خارج کردن).



اگر دو آیتم را به لیست وارد کنید، اولین موردی که به لیست وارد کرده‌اید قبل از آیتم دوم قرار می‌گیرد. شما می‌توانید از این روش برای لیست جست‌وجوی خود استفاده کنید! افرادی که ابتدا به لیست اضافه می‌شوند اول از همه از صف خارج و جست‌وجو می‌شوند.

صف یک ساختمان داده‌ی FIFO نامیده می‌شود: First In, First Out. در مقابل، پشته یک ساختمان داده‌ی LIFO است: Last In, First Out.



FIFO

اولین ورودی، اولین خروجی



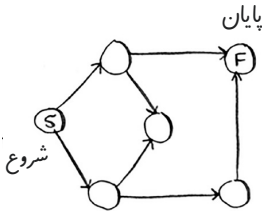
LIFO

آخرین ورودی، اولین خروجی

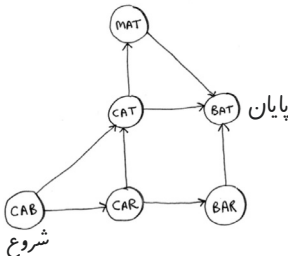
اکنون که می‌دانید یک صف چگونه کار می‌کند، بیایید جست‌وجوی سطح اول را اجرا کنیم!

### تمرین

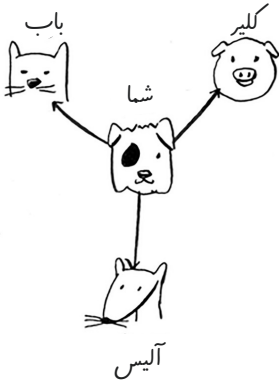
برای یافتن راه حل، الگوریتم جست‌وجوی سطح اول را روی هر یک از این گراف‌ها اجرا کنید.  
۶.۱ طول کوتاه‌ترین مسیر را از ابتدا تا انتها پیدا کنید.



۶.۲ طول کوتاه‌ترین مسیر را از «cab» تا «bat» بیابید.





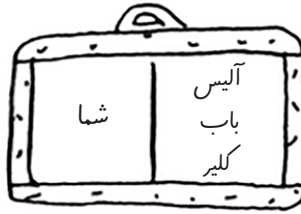


## پیاده‌سازی گراف

ابتدا باید کد گراف را پیاده‌سازی کنید. یک گراف از چندین گره تشکیل شده است.

و هر گره به گره‌های همسایه متصل است. چگونه رابطه‌ای مانند «you -> bob» را نشان می‌دهید؟ خوشبختانه، شما پیش از این با ساختمان داده‌ای آشنا شده‌اید که امکان می‌دهد رابطه‌ها را نشان دهید: جدول هش!

در نظر داشته باشید، جدول هش به شما امکان می‌دهد یک کلید را به یک مقدار نگاشت کنید. در این مورد، شما می‌خواهید یک گره را به تمام همسایگانش نگاشت کنید.

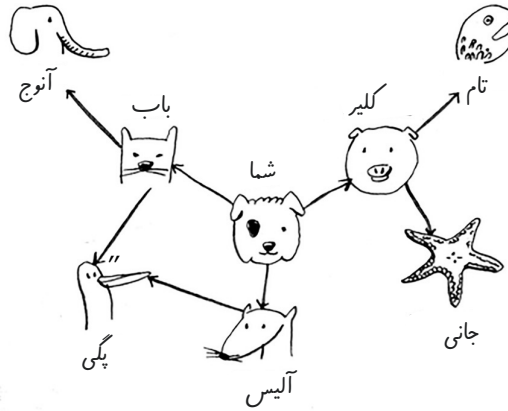


کد آن در پایتون به این صورت است:

```
graph = {}  
graph["you"] = ["alice", "bob", "claire"]
```

توجه داشته باشید که «you» به یک آرایه نگاشت شده است. بنابراین `graph["you"]` آرایه‌ای از همه‌ی همسایگان «you» را در اختیار شما می‌گذارد.

یک گراف تنها دسته‌ای از گره و یال است، در نتیجه این تنها چیزی است که برای داشتن یک گراف در پایتون نیاز دارید. در مورد یک گراف بزرگ‌تر مانند این چطور؟



کد پایتون آن در ادامه آمده است:

```
graph = {}
graph["you"] = ["alice", "bob", "claire"]
graph["bob"] = ["anuj", "peggy"]
graph["alice"] = ["peggy"]
graph["claire"] = ["thom", "jonn"]
graph["anuj"] = []
graph["peggy"] = []
graph["thom"] = []
graph["jonny"] = []
```

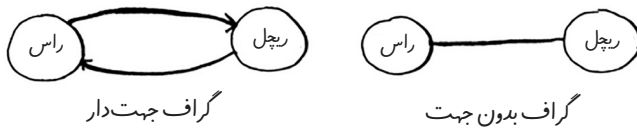
پرسش: آیا مهم است که جفت کلید/مقدار را به چه ترتیبی اضافه کنید؟ مشکلی خواهد بود اگر به جای:

```
graph["anuj"] = []
graph["claire"] = ["thom", "jonny"]
```

بنویسیم:

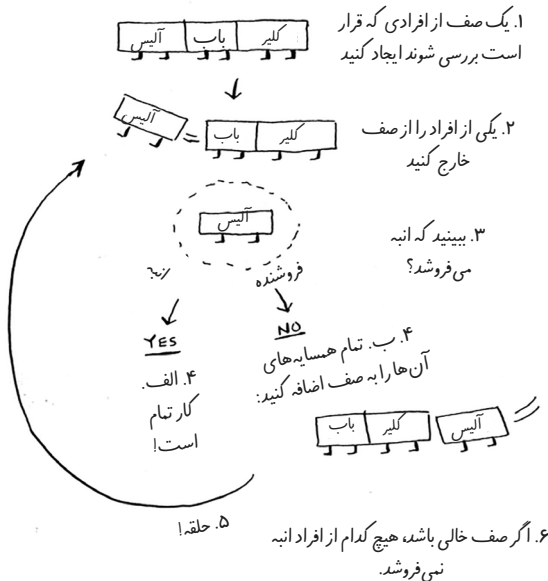
```
graph["claire"] = ["thom", "jonny"]
graph["anuj"] = []
```

فصل قبل را به خاطر بیاورید. پاسخ: تفاوتی ندارد. جدول های هش ترتیبی ندارند، بنابراین مهم نیست جفت کلید/مقدار را به چه ترتیبی اضافه می کنید. آنوج، پگی، تام و جانی هیچ همسایه ای ندارند. فلیش هایی به آن ها اشاره می کند، اما هیچ فلشی از سمت آن ها به شخص دیگری نیست. این یک گراف جهت دار است. - رابطه فقط یک طرفه است. بنابراین آنوج همسایه ی باب است، اما باب همسایه ی آنوج نیست. یک گراف بدون جهت هیچ فلشی ندارد و هر دو گره همسایه ی یکدیگر هستند. به عنوان مثال، هر دوی این گراف ها برابر هستند.



## پیاده سازی الگوریتم

برای جمع بندی، در ادامه شیوه ی اجرا و پیاده سازی آن آمده است.



### نکته

هنگام به روزرسانی صف ها، از اصطلاحات enqueue و dequeue استفاده می کنیم. همچنین با اصطلاحات push و pop آشنا می شوید. Push تقریباً همیشه همان enqueue است و pop تقریباً همیشه همان dequeue).

برای شروع یک صف درست کنید. در پایتون، از تابع صف دو طرفه (deque) برای این کار استفاده می‌کنید:

```
from collections import deque
search_queue = deque()
search_queue += graph["you"]
```

یک صف جدید ایجاد می‌کنیم. ←  
 تمام همسایگان را به صف جست‌وجو اضافه می‌کنیم. ←



به یاد داشته باشید، graph["you"] لیستی از تمامی همسایه‌های شما، مانند ["alice", "bob", "claire"] ارائه می‌کند. همه‌ی آن‌ها به صف جست‌وجو اضافه می‌شوند.

باقی کد به این صورت است:

```
while search_queue:
    person = search_queue.popleft()
    if person_is_seller(person):
        print person + " is a mango seller!"
        return True
    else:
        search_queue += graph[person]
return False
```

در شرایطی که صف خالی نباشد ←  
 نفر اول را از صف جدا می‌کنند. ←  
 آیا این شخص انبه می‌فروشد؟ ←  
 بله. انبه می‌فروشد. ←  
 خیر. تمام دوستان این شخص را به صف جست‌وجو اضافه کنید. ←  
 اگر به این مرحله رسیدید، هیچ‌کدام از افراد حاضر در صف انبه نمی‌فروشند. ←

نکته‌ی آخر: همچنان به یک تابع `person_is_seller` نیاز است تا هر زمان یک فروشنده‌ی انبه پیدا شد به شما اطلاع بدهد:

```
def person_is_seller(name):
    return name[-1] == 'm'
```

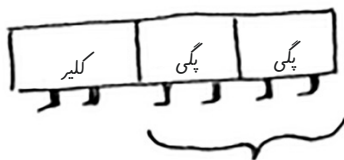
## 1. Double-ended queue

این تابع بررسی می کند که آیا نام شخص به حرف  $m$  ختم می شود یا خیر. اگر چنین باشد، این شخص یک انبه فروش است. روشی هرچند احمقانه اما کارساز برای این مثال. بیایید جست و جوی سطح اول را در عمل ببینیم.



- و به همین ترتیب. الگوریتم ادامه می یابد تا زمانی که:
- فروشنده ی انبه پیدا شود
- یا صف خالی شود که در این صورت فروشنده ای برای انبه وجود ندارد.

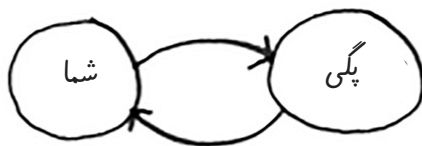
آلیس و باب یک دوست مشترک دارند: پگی. بنابراین پگی دو بار به صف اضافه می‌شود: یک بار وقتی دوستان آلیس را اضافه می‌کنید و بار دیگر زمانی که دوستان باب را اضافه می‌کنید. در نهایت با دو پگی در صف جست‌وجو مواجه خواهید شد.



عجب! پگی دو نوبت در  
صف جست‌وجو هست!

اما تنها یک بار باید پگی را بررسی کنید تا ببینید انبه می‌فروشد یا خیر. اگر دوبار بررسی کنید، کار غیر ضروری و اضافه‌ای انجام داده‌اید. بنابراین هنگامی که شخصی را برای جست‌وجو بررسی می‌کنید، باید به آن شخص تیک «بررسی شده» بزنید تا دوباره بررسی نشود.

اگر این کار را نکنید، ممکن است در یک حلقه بی‌نهایت قرار بگیرید. فرض کنید گراف انبه‌فروش به این شکل باشد:



برای شروع، صف جست‌وجو تمام همسایه‌های شما را شامل می‌شود.



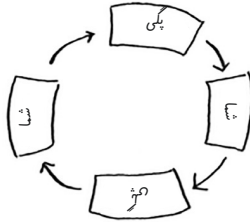
حالا شما پگی را بررسی می‌کنید. او فروشنده‌ی انبه نیست، بنابراین همه‌ی همسایگان او را به صف جست‌وجو اضافه می‌کنید.



بعد، خودتان را بررسی کنید. شما انبه‌فروش نیستید، بنابراین همه‌ی همسایگان خود را به صف جست‌وجو اضافه می‌کنید.



و غیره. این یک حلقه‌ی بی‌نهایت است، زیرا صف جست‌وجو میان شما و پگی تداوم خواهد داشت.



قبل از بررسی یک شخص، مهم است که مطمئن بشوید قبلاً بررسی نشده است. برای انجام این کار، فهرستی از افرادی که قبلاً بررسی کرده‌اید را نگه دارید. با در نظر گرفتن این نکته کد نهایی برای جست‌وجوی سطح اول به این صورت است:

```
def search(name) :  
    search_queue = deque()  
    search_queue += graph[name]
```

```

searched = []
while search_queue:
    person = search_queue.popleft()
    if not person in searched:
        if person_is_seller(person):
            print person + " is a mango seller!"
            return True
        else:
            search_queue += graph[person]
            searched.append(person)
return False
search("you")

```

آرایه‌ای برای نام افرادی که پیش از این جست‌وجو کرده‌اید. ←  
 تنها در صورتی جست‌وجو می‌کند که نام فرد مورد نظر پیش از این جست‌وجو نشده باشد. ←  
 این شخص را به عنوان جست‌وجو شده علامت‌گذاری می‌کند ←

کد را اجرا کنید. تابع `person_is_seller` را به چیزی بامعنی‌تر تغییر بدهید و ببینید چیزی را که انتظار دارید پیرینت می‌کند یا خیر.

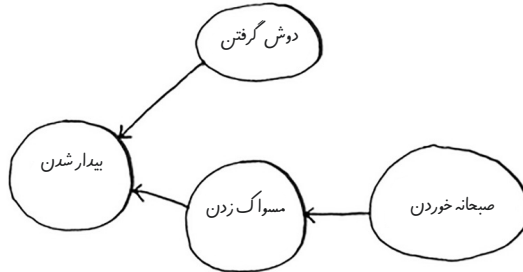
## زمان اجرا

اگر در سراسر شبکه‌ی خود به دنبال فروشنده‌ی انبه بگردید، به این معنی است که تمامی یال‌ها را دنبال خواهید کرد (به یاد داشته باشید یک یال، فلش یا ارتباط یک فرد به فردی دیگر است). بنابراین زمان اجرای حداقل (تعداد یال‌ها)  $O$  است. همچنین یک صف از هر فرد برای جست‌وجو نگه می‌دارید. افزودن یک نفر به صف زمان ثابت دارد:  $O(1)$ . انجام این کار برای هر فرد، (تعداد افراد)  $O$  نیاز دارد. جست‌وجوی سطح اول (تعداد یال‌ها + تعداد افراد)  $O$  را می‌گیرد و معمولاً به صورت  $O(V+E)$  (تعداد رئوس و  $E$  تعداد یال‌ها) نوشته می‌شود.



## تمرین

گراف کوچکی از برنامه‌ی صبحگاهی من به این شکل است:



این به شما می‌گوید که تا زمانی که دندان‌هایم را مسواک نزدم صبحانه نمی‌خورم. بنابراین «صبحانه خوردن» به «مسواک زدن» وابسته است.

از طرف دیگر، دوش گرفتن به مسواک زدن دندان‌هایم بستگی ندارد، زیرا می‌توانم قبل از مسواک زدن دوش بگیرم. از این گراف، می‌توانید فهرستی از ترتیب برنامه‌ی صبحگاهی من تهیه کنید:

۱. بیدار شدن.
۲. دوش گرفتن.
۳. مسواک زدن.
۴. صبحانه خوردن.

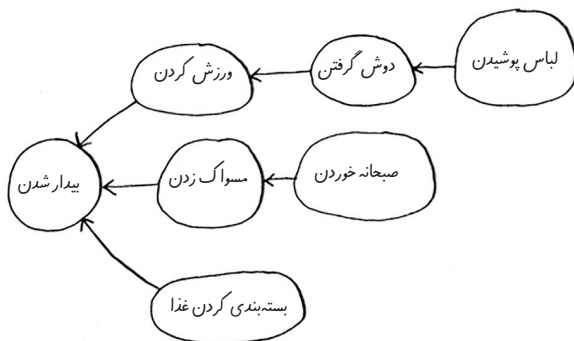
توجه داشته باشید که «دوش گرفتن» را می‌شود جابه‌جا کرد، بنابراین این لیست نیز درست است:

۱. بیدار شدن.
۲. مسواک زدن.
۳. دوش گرفتن.
۴. صبحانه خوردن.

۶.۳ درستی یا نادرستی هر یک از این سه لیست را مشخص کنید.

A.	B.	C.
۱. بیدار شدن	۱. بیدار شدن	۱. دوش گرفتن
۲. دوش گرفتن	۲. مسواک زدن	۲. بیدار شدن
۳. صبحانه خوردن	۳. صبحانه خوردن	۳. مسواک زدن
۴. مسواک زدن	۴. دوش گرفتن	۴. صبحانه خوردن

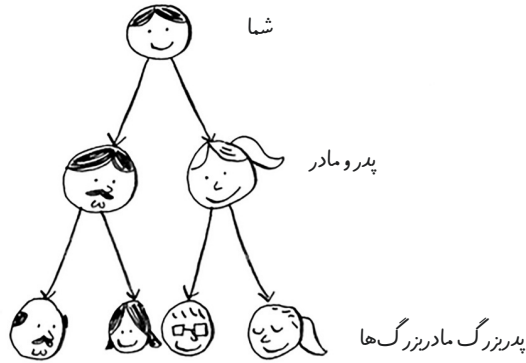
۶.۴ در اینجا یک گراف بزرگ‌تر وجود دارد. یک لیست صحیح برای این گراف تهیه کنید.



می‌توان گفت که این لیست مرتب‌شده است. اگر تسک A به تسک B وابسته باشد، تسک A بعدتر در لیست نشان داده می‌شود. به این مرتب‌سازی توپولوژیکی<sup>۱</sup> گفته می‌شود و راهی برای ایجاد لیست مرتب‌شده از یک گراف است. فرض کنید در تدارک یک عروسی هستید و یک گراف بزرگ سرشار از کارهایی که باید انجام شوند در اختیار دارید و نمی‌دانید از کجا شروع کنید. می‌توانید گراف را از نظر توپولوژیکی مرتب کنید و لیستی از کارهایی که باید انجام دهید را به ترتیب دریافت کنید.

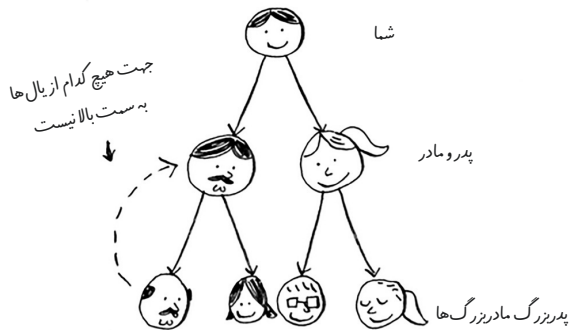
## 1. Topological sort

فرض کنید شما یک شجره‌نامه دارید.



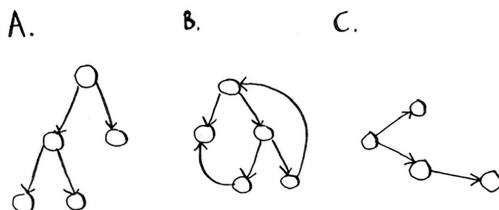
چون گره (افراد) و یال دارد در نتیجه این یک گراف است.

یال‌ها به والدین گره‌ها اشاره می‌کنند. اما همه‌ی یال‌ها به سمت پایین هستند - منطقی هم نیست در یک شجره‌نامه جهش یک یال به سمت بالا باشد! این بی‌معنی خواهد بود - پدر شما نمی‌تواند پدر پدر بزرگ شما باشد!



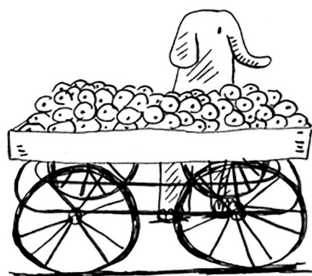
به این گراف درخت گفته می‌شود. درخت نوع خاصی از گراف است که در آن جهت هیچ یالی رو به عقب نیست.

۶.۵ کدام یک از تصاویری که در ادامه آمده علاوه بر گراف درخت هم هستند؟



## جمع بندی

- جست و جوی سطح اول به شما می گوید که آیا مسیری از A به B وجود دارد یا خیر.
- اگر مسیری وجود داشته باشد، جست و جوی سطح اول کوتاه ترین مسیر را پیدا می کند.
- اگر مشکلی مانند «یافتن کوتاه ترین X» دارید، سعی کنید مشکل خود را به صورت گراف مدل سازی کنید و از جست و جوی سطح اول برای حل آن استفاده کنید.
- یک گراف جهت دار دارای فلش است، و رابطه از جهت فلش پیروی می کند (adit -> rama به معنای این است که «راما به ادیت بدهکار است»).
- گراف های غیر جهت دار فلش ندارند و رابطه به دو طرف پیش می رود (ross - rachel این معناست که «ریچل با راس قرار می گذارد و راس با ریچل قرار می گذارد»).
- صف ها FIFO (اولین ورودی، اولین خروجی) هستند.
- پشته ها LIFO (آخرین ورودی، اولین خروجی) هستند.
- باید افراد را به ترتیبی که به لیست جست و جو اضافه شده اند بررسی کنید، بنابراین لیست جست و جو باید یک صف باشد. در غیر این صورت کوتاه ترین مسیر را نخواهید داشت.



- هنگامی که فردی را بررسی می کنید، مطمئن بشوید که او را دوباره بررسی نکنید. در غیر این صورت، ممکن است در یک حلقه ی بی نهایت قرار بگیرید.

## ۷ | الگوریتم دایجسترا<sup>۱</sup>



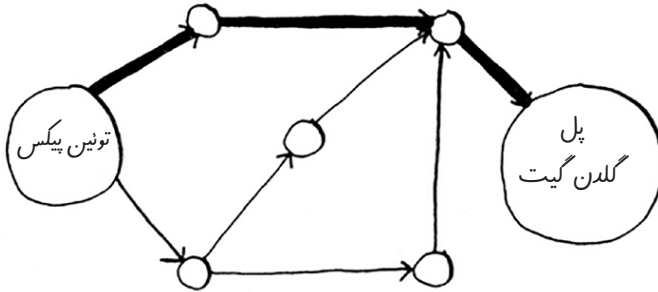
در این فصل

- مبحث گراف‌ها را ادامه می‌دهیم و شما با گراف‌های وزن‌دار<sup>۲</sup> آشنا می‌شوید: روشی برای اختصاص وزن کم‌تر یا بیشتر به برخی از یال‌ها.
- الگوریتم دایجسترا را یاد می‌گیرید. این الگوریتم به شما امکان می‌دهد برای گراف‌های وزن‌دار به پرسش «کوتاه‌ترین مسیر به X چیست؟» پاسخ دهید.
- در مورد چرخه<sup>۳</sup> در گراف، جایی که الگوریتم دایجسترا جواب‌گو نیست، یاد می‌گیرید.

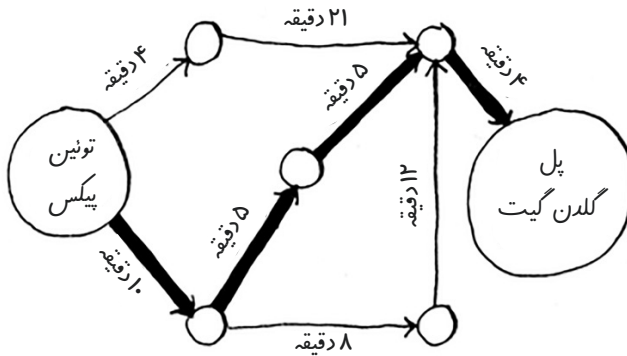
---

1. Dijkstra's algorithm  
2. weighted graphs  
3. cycles

در فصل قبل، مسیری برای رفتن از نقطه‌ی A به نقطه‌ی B پیدا کردید.



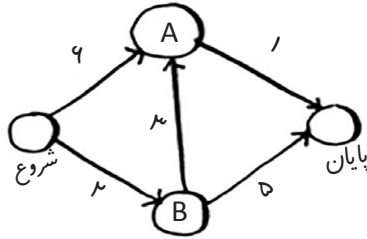
این لزوماً سریع‌ترین مسیر نیست بلکه کوتاه‌ترین مسیر است، زیرا کم‌ترین تعداد قطعه (سه قطعه) را دارد. اما فرض کنید زمان سفر را به آن قطعه‌ها اضافه کنید. اکنون می‌بینید که مسیر سریع‌تری وجود دارد.



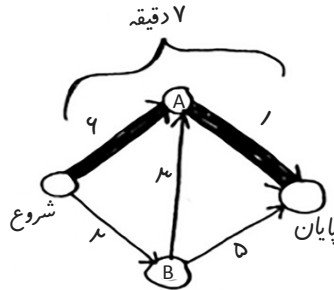
در فصل پیش از جست‌وجوی سطح اول استفاده کردید. جست‌وجوی سطح اول مسیری را که کمترین قطعه را دارد (گراف اول در اینجا نشان داده شده) پیدا می‌کند. اگر به جای آن سریع‌ترین مسیر را بخواهید (گراف دوم) چه می‌شود؟ شما می‌توانید این کار را به سریع‌ترین حالت با یک الگوریتم متفاوت به نام الگوریتم دایجسترا انجام بدهید.

## کار با الگوریتم دایجسترا

عملکرد آن را با یک گراف بررسی کنیم:



هر قطعه یک زمان سفر بر حسب دقیقه دارد. شما از الگوریتم دایجسترا استفاده می‌کنید تا در کوتاه‌ترین زمان ممکن از نقطه‌ی شروع تا نقطه‌ی پایان بروید. اگر در این گراف جست‌وجوی سطح اول انجام دهید، کوتاه‌ترین مسیر را به این شکل خواهید داشت:



این مسیر ۷ دقیقه طول می‌کشد. بیایید ببینیم می‌توان مسیری یافت که زمان کمتری طول بکشد! چهار گام برای الگوریتم دایجسترا وجود دارد:

۱. «ارزان‌ترین» گره را پیدا کنید. این همان گره‌ای است که می‌توانید در کم‌ترین زمان به آن برسید.

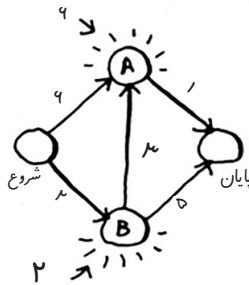
۲. هزینه‌های همسایگان این گره را به‌روز کنید. جلوتر توضیح می‌دهم منظورم

چیست.

۳. این کار را تا انجام آن برای تمامی گره‌های گراف تکرار کنید.

۴. مسیر نهایی را محاسبه کنید.

**گام ۱:** ارزان‌ترین گره را پیدا کنید. شما در نقطه‌ی شروع ایستاده‌اید و نمی‌دانید که آیا باید به گره‌ی A بروید یا گره‌ی B. چقدر طول می‌کشد تا به هر گره برسید؟



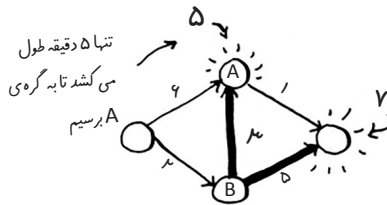
۶ دقیقه تا گره‌ی A و ۲ دقیقه برای رسیدن به گره‌ی B طول می‌کشد. در مورد بقیه‌ی گره‌ها، هنوز چیزی نمی‌دانید.

گره	زمان رسیدن به گره
A	۶
B	۲
پایان	$\infty$

از آنجا که هنوز نمی‌دانید چقدر طول می‌کشد تا به نقطه‌ی پایان برسید، بی‌نهایت را کنار می‌گذارید (به زودی دلیلش را می‌فهمید). گره‌ی B نزدیک‌ترین گره است ... ۲ دقیقه با آن فاصله دارد.

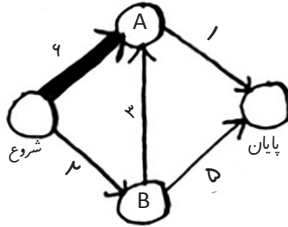
**گام ۲:** محاسبه کنید که چقدر طول می‌کشد تا از طریق یک یال از B به همه‌ی همسایگان گره‌ی B برسید.

گره	زمان
A	۵
B	۲
پایان	۷

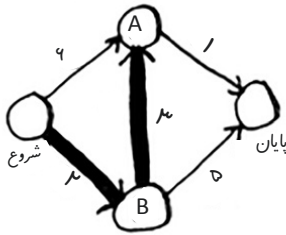




بسیار خوب. شما یک مسیر کوتاه‌تر برای گرهی A پیدا کردید! قبل‌تر ۶ دقیقه طول می‌کشید تا به گرهی A برسید.



اما اگر از گرهی B عبور کنید، مسیری وجود دارد که فقط ۵ دقیقه طول می‌کشد!



وقتی مسیر کوتاه‌تری برای همسایه‌ی B پیدا کردید، هزینه‌ی آن را به روز کنید. در این مورد، یافته‌ها عبارتند از:

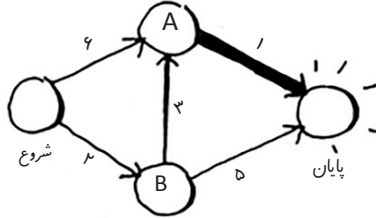
- یک مسیر کوتاه‌تر به A (از ۶ دقیقه به ۵ دقیقه کاهش می‌یابد)
- یک مسیر کوتاه‌تر تا پایان (از بی‌نهایت به ۷ دقیقه کاهش می‌یابد)

**گام ۳: تکرار کنید!**

گره	زمان
A	۵
B	۲
پایان	۷

**دوباره گام ۱:** گره‌ای را پیدا کنید که کم‌ترین زمان را برای رسیدن نیاز دارد. کار با گرهی B تمام شده است، بنابراین گرهی A کوچک‌ترین تخمین زمان بعدی را شامل می‌شود.

**دوباره گام ۲:** هزینه‌ها را برای همسایگان گرهی A به روز کنید.



چه خوب، اکنون ۶ دقیقه طول می‌کشد تا به نقطه‌ی پایان برسیم! شما الگوریتم دایجسترا را برای هر گره اجرا کرده‌اید (نیازی به اجرای آن برای گرهی پایانی ندارید). در این مرحله، شما می‌دانید

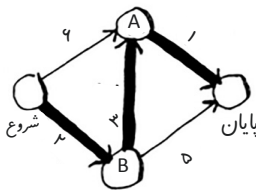
• ۲ دقیقه طول می‌کشد تا به گرهی B برسید.

• ۵ دقیقه طول می‌کشد تا به گرهی A برسید.

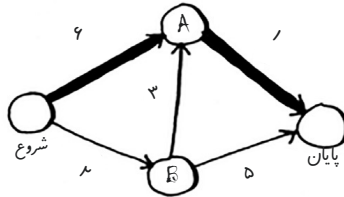
• ۶ دقیقه طول می‌کشد تا به پایان برسید.

گره	زمان
A	۵
B	۲
پایان	۶

آخرین مرحله یعنی محاسبه‌ی مسیر نهایی را برای بخش بعدی نگه می‌دارم. فعلاً فقط نشان می‌دهم که مسیر نهایی چیست.

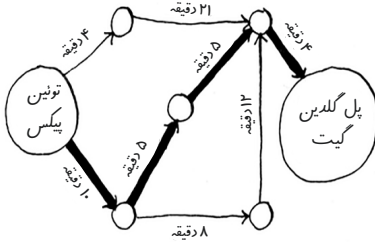


جست‌وجوی سطح اول این را به عنوان کوتاه‌ترین مسیر پیدا نمی‌کند، چون سه قطعه است. و راهی دو قطعه‌ای برای شروع تا پایان وجود دارد.

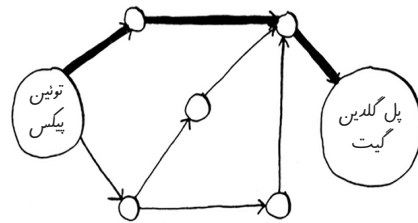


کوتاه‌ترین مسیر با جست‌وجوی سطح اول

در فصل پیش، شما از جست‌وجوی سطح اول برای یافتن کوتاه‌ترین مسیر بین دو نقطه استفاده کردید. در آن زمان، «کوتاه‌ترین مسیر» به معنای مسیری با کمترین قطعات بود. اما در الگوریتم دایجسترا، شما به هر بخش یک عدد یا وزن اختصاص می‌دهید. سپس الگوریتم دایجسترا مسیری با کمترین وزن کل را پیدا می‌کند.



گراف وزن دار  
(با استفاده از بلس فرد)



گراف بی‌وزن  
(با استفاده از BFS)

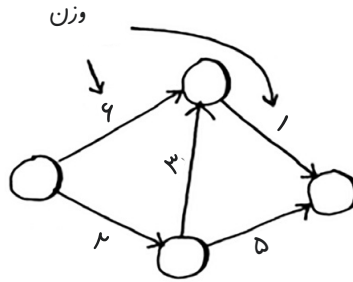
به طور خلاصه، الگوریتم دایجسترا دارای چهار گام است:

۱. ارزان‌ترین گره را پیدا کنید. این گره‌ای است که می‌توانید در کوتاه‌ترین زمان به آن برسید.
۲. بررسی کنید که آیا مسیر ارزان‌تری به همسایگان این گره وجود دارد یا خیر. اگر چنین است، هزینه‌های آن‌ها را به روز کنید.

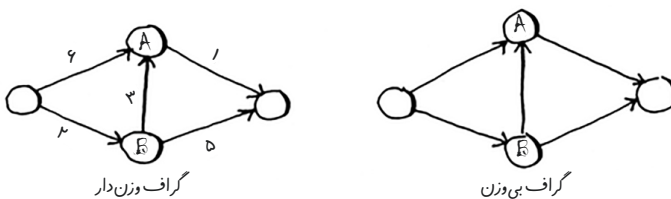
۳. این روند را تکرار کنید تا زمانی که این کار را برای هر گره در گراف انجام داده باشید.
۴. مسیر نهایی را محاسبه کنید. (در بخش بعدی آمده است!)

## اصطلاح‌شناسی

قرار است چند مثال دیگر از الگوریتم دایجسترا را به شکل عملی به شما نشان بدهم. اما ابتدا اجازه دهید به تشریح چند اصطلاح بپردازم. هنگامی که با الگوریتم دایجسترا کار می‌کنید، هر یال در گراف همراه با یک عدد است. به این اعداد وزن گفته می‌شود.

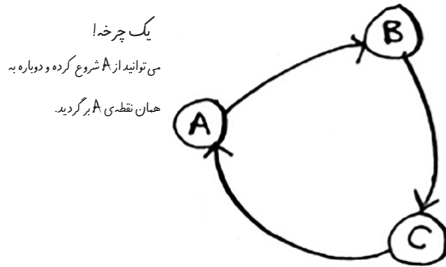


به گراف دارای وزن، گراف وزن دار<sup>۲</sup> می‌گویند. گراف بدون وزن را گراف بی‌وزن<sup>۳</sup> می‌نامند.

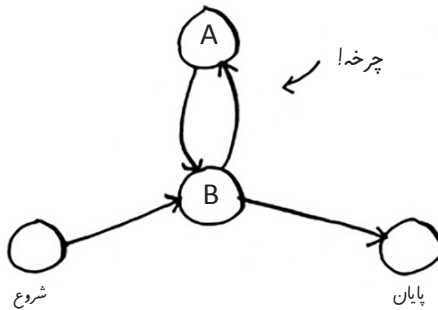


برای محاسبه‌ی کوتاه‌ترین مسیر در یک گراف بی‌وزن، از جست‌وجوی سطح اول استفاده کنید. برای محاسبه‌ی کوتاه‌ترین مسیر در یک گراف وزن دار، از الگوریتم دایجسترا استفاده کنید. گراف‌ها همچنین می‌توانند چرخه داشته باشند. یک چرخه به این شکل است.

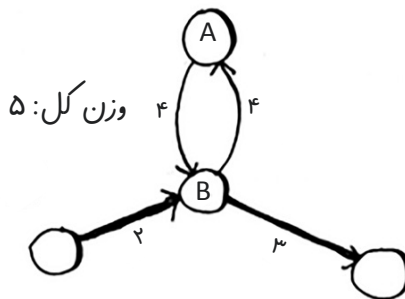
1. weights
2. Weighted graph
3. Unweighted graph



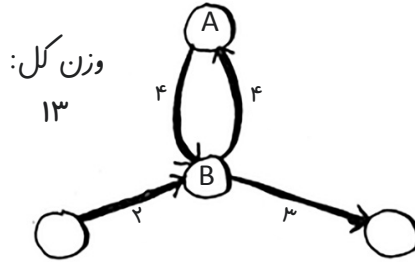
به این معنی که می‌توانید از یک گره شروع کرده، به اطراف حرکت کنید و در نهایت به همان گره برسید. فرض کنید می‌خواهید کوتاه‌ترین مسیر در این گراف را که دارای یک چرخه است پیدا کنید.



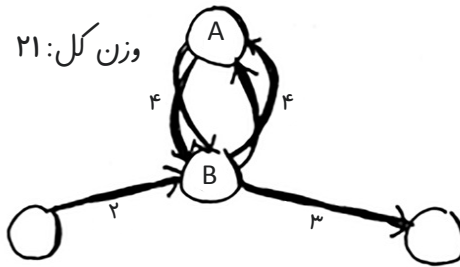
دنبال کردن چرخه منطقی است؟ خوب، شما می‌توانید از مسیری استفاده کنید که از چرخه اجتناب می‌کند.



یا می‌توانید چرخه را طی کنید.



در هر صورت به گره A ختم می‌شود، اما این چرخه وزن بیشتری دارد. حتی اگر بخواهید می‌توانید چرخه را دو بار طی کنید.

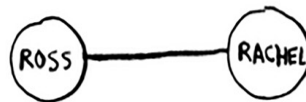


اما هر بار که چرخه را طی می‌کنید، مقداری معادل ۸ به وزن کل اضافه می‌کنید. در نتیجه با طی کردن چرخه هرگز به کوتاه‌ترین مسیر دست پیدا نمی‌کنید.

در آخر، به خاطر دارید در فصل ۶ درباره‌ی گراف‌های جهت‌دار و غیر جهت‌دار چه گفتیم؟

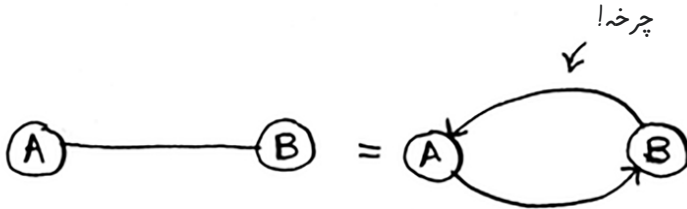


گراف جهت‌دار



گراف بدون جهت

گراف بدون جهت به این معنی است که هر دو گره به یکدیگر اشاره می‌کنند. در واقع همان چرخه است!



با یک گراف بدون جهت، هر یال چرخه‌ی دیگری اضافه می‌کند. الگوریتم دایجسترا فقط با گراف‌های جهت‌دار غیر مدور که به اختصار DAG نامیده می‌شوند کار می‌کند.

## معامله بر سر یک پیانو



صحبت از اصطلاحات کافی است، بیاید به مثال دیگری نگاه کنیم! این تصویر راما است. راما در تلاش است تا یک کتاب موسیقی را با پیانو عوض کند.

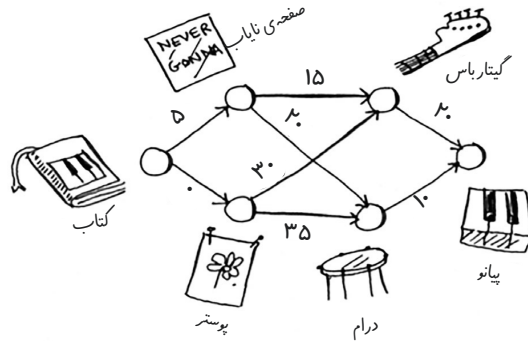


الکس می‌گوید: «این پوستر را با کتابت تاخت می‌زنم. این پوستر گروه محبوبم دیسترویره. کتابت رو با این ال پی کمیاب ریک استلی عوض می‌کنم و پنج دلار هم میذارم روش.» امی می‌گوید: «اوه، شنیده‌ام که صفحه‌ی آهنگ واقعاً محشری داره. من گیتار یا درامم را با پوستر یا صفحه‌ی تو عوض می‌کنم.»

بتهوون فریاد می‌زند: «گیتارو من می‌خواستم. پیانورو در عوض یکی از چیزهای امی به تو می‌دهم.»

بسیار عالی! راما با کمی پول می‌تواند یک کتاب پیانو را به یک پیانوی واقعی تبدیل کند. فقط باید بفهمد که چگونه برای انجام این معامله‌ها کمترین پول را خرج کند. بیایید آنچه را که به او پیشنهاد شده است، به گراف تبدیل کنیم.

### 1. Directed acyclic graph



در این گراف، گره‌ها تمام آیتم‌هایی را شامل می‌شوند که راما می‌تواند سر آن‌ها معامله کند. وزن روی یال‌ها مبلغی است که او باید برای انجام معامله بپردازد. بنابراین می‌تواند پوستر را با گیتار به قیمت ۳۰ دلار عوض کند، یا صفحه‌ها را با گیتار با ۱۵ دلار معاوضه کند. راما چگونه می‌تواند مسیر کتاب تا پیانو را که کمترین پول را در آن خرج می‌کند، بفهمد؟ الگوریتم دایجسترا نجات‌بخش است! به یاد داشته باشید، الگوریتم دایجسترا چهار مرحله دارد. در این مثال، شما هر چهار مرحله را انجام خواهید داد، بنابراین مسیر نهایی را در پایان نیز محاسبه خواهید کرد.

گره	هزینه
صفحه	۵
پوستر	۰
گیتار	$\infty$
درام	$\infty$
پیانو	$\infty$

هنوز به این گره‌ها نرسیده‌ایم

ابتدا نیاز به مقداری مقدمه‌چینی است. جدولی از هزینه‌ی هر گره تهیه کنید. هزینه‌ی یک گره به معنای آن است که رسیدن به آن چقدر هزینه‌بردار است.

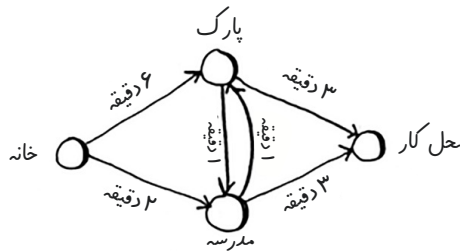
والد	گره
کتاب	صفحه
کتاب	پوستر
—	گیتار
—	درام
—	پیانو

با ادامه الگوریتم، این جدول را به تناوب به روز خواهید کرد. برای محاسبه‌ی مسیر نهایی، همچنین به یک ستون والد در این جدول نیاز دارید.

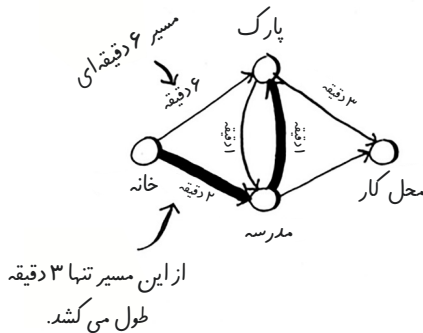


به زودی نحوه‌ی کار این ستون را به شما نشان خواهم داد. بیایید الگوریتم آن را بنویسیم.

**گام ۱:** ارزان‌ترین گره را پیدا کنید. در این مورد، پوستر ارزان‌ترین معامله است، صفر دلار. آیا راه ارزان‌تری برای معامله‌ی پوستر وجود دارد؟ این نکته‌ی بسیار مهمی است، پس راجع به آن فکر کنید. آیا می‌توانید یک سری معاملات را ببینید که پوستر راما را با کمتر از صفر دلار دریافت می‌کند؟ اگر به جوابی در ذهن خود رسیدید، ادامه‌ی مطلب را بخوانید. پاسخ: خیر. چون پوستر ارزان‌ترین گره‌ای است که راما می‌تواند به آن دسترسی پیدا کند، راهی برای ارزان‌تر کردن آن وجود ندارد. در اینجا یک راه متفاوت برای نگاه کردن به آن وجود دارد. فرض کنید از خانه به محل کار سفر می‌کنید.



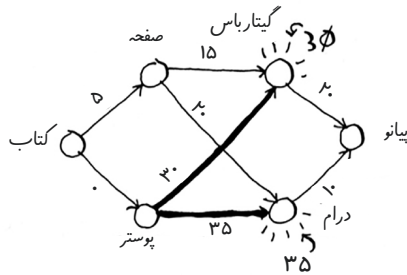
اگر مسیر را به سمت مدرسه طی کنید، ۲ دقیقه طول می‌کشد. اگر مسیر را به سمت پارک طی کنید، ۶ دقیقه طول می‌کشد. آیا راهی وجود دارد که بتوانید مسیر پارک را طی کنید و در کمتر از ۲ دقیقه به مدرسه برسید؟ غیرممکن است، زیرا فقط تا خود پارک بیشتر از ۲ دقیقه طول می‌کشد. از طرف دیگر آیا می‌توانید مسیر سریع‌تری به پارک پیدا کنید؟ بله.



این ایده‌ی کلیدی الگوریتم دایجسترا است: به ارزان‌ترین گره در گراف خود نگاه کنید. هیچ راه ارزان‌تری برای رسیدن به این گره وجود ندارد!

به مثال موسیقی برگردیم. پوستر ارزان‌ترین معامله است.

**گام ۲:** مشخص کنید که چقدر طول می‌کشد تا به همسایگانش برسید (هزینه).



والد	گره	هزینه
کتاب	صفحه	۵
کتاب	پوستر	۰
گیتار باس	گیتار	۳۰
پوستر	درام	۳۵
—	پیانو	∞

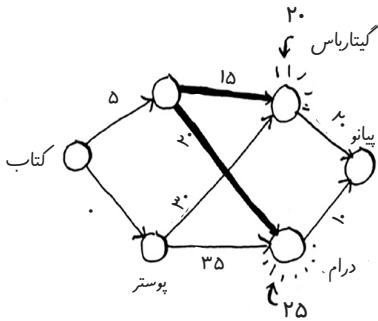
قیمت گیتار باس و درام را در جدول دارید. وقتی به سراغ پوستر رفتید ارزش آن‌ها مشخص شده بود، بنابراین پوستر به عنوان والد آن‌ها در نظر گرفته می‌شود. یعنی برای رسیدن به گیتار بیس، یال پوستر را دنبال می‌کنید، همین را برای درام‌ها هم می‌شود گفت.

والد	گره	هزینه
کتاب	صفحه	۵
کتاب	پوستر	۰
پوستر	گیتار	۳۰
پوستر	درام	۳۵
—	پیانو	∞

از «پوستر» که شروع کنیم به این گره‌ها می‌رسیم

**دوباره گام ۱:** صفحه ارزان‌ترین گره‌ی بعدی با قیمت ۵ دلار است.

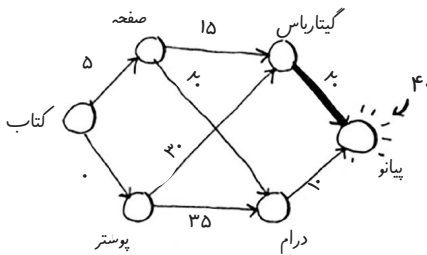
**دوباره گام ۲:** مقادیر همه‌ی همسایگانش را به روز کنید.



هزینه	گره	والد
۵	صفحه	کتاب
۰	پوستر	کتاب
۲۰	گیتار	صفحه
۲۵	درام	صفحه
∞	پیانو	—

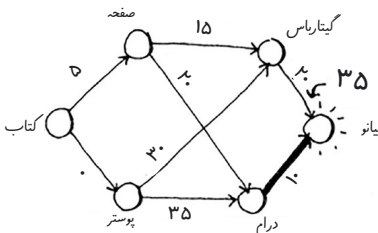
خب، قیمت درام و گیتار را به روز کردی! این به این معناست که با دنبال کردن یال های صفحه، رسیدن به درام و گیتار ارزان تر است. بنابراین صفحه را به عنوان والد جدید برای هر دو ساز در نظر می گیرید.

گیتار باس ارزان ترین کالای بعدی است. همسایگان آن را به روز کنید.



هزینه	گره	والد
۵	صفحه	کتاب
۰	پوستر	کتاب
۲۰	گیتار	صفحه
۲۵	درام	صفحه
۴۰	پیانو	گیتار

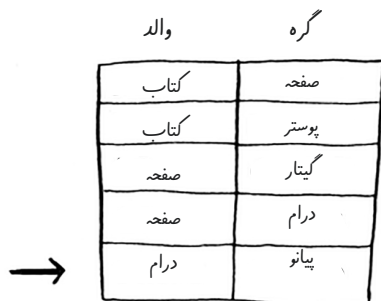
خوب، بالاخره با مبادله ی گیتار با پیانو، به یک قیمت برای پیانو دست پیدا کردید. بنابراین شما گیتار را به عنوان والد در نظر می گیرید. در نهایت، آخرین گره، درام است.



هزینه	گره	والد
۵	صفحه	کتاب
۰	پوستر	کتاب
۲۰	گیتار	صفحه
۲۵	درام	صفحه
۳۵	پیانو	درام

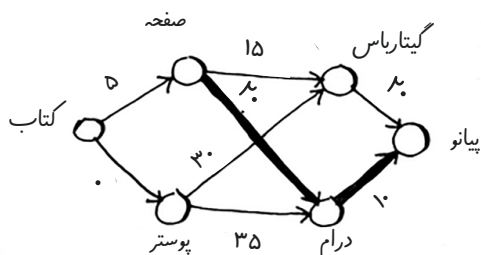
اما می تواند با معامله ی درام با پیانو، پیانو را حتی ارزان تر دریافت کند. بنابراین ارزان ترین مجموعه معاملات ۳۵ دلار برای اما هزینه در بر دارد.

حالا، همان طور که قول داده بودم، باید مسیر را مشخص کنید. تا اینجا، می دانید که کوتاه ترین مسیر ۳۵ دلار هزینه دارد، اما چگونه می توانید مسیر را مشخص کنید؟ برای شروع، به والد پیانو نگاه کنید.

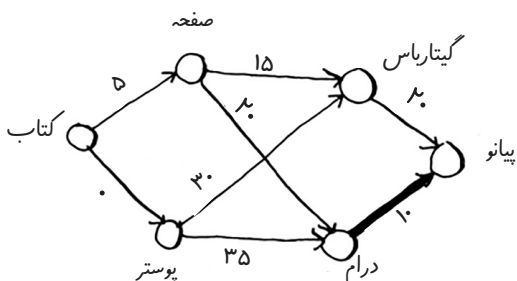


درام والد پیانو است. یعنی اما درام ها را با پیانو عوض می کند. پس شما این یال را دنبال می کنید.

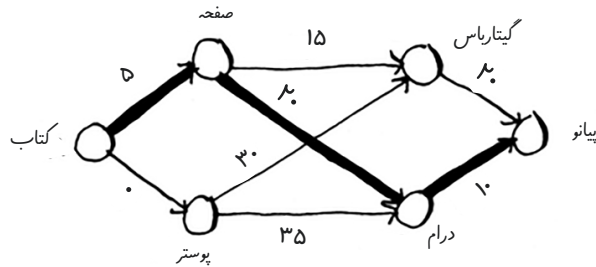
بیا بید ببینیم یال ها را چگونه دنبال می کنید. پیانو درام را به عنوان والد خود دارد.



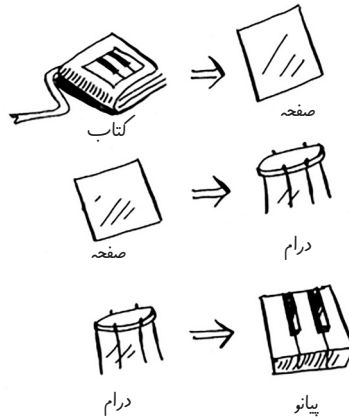
و صفحه والد درام ها است.



بنابراین راما صفحه را با درام‌ها معاوضه می‌کند. و البته، این کتاب را با صفحه عوض خواهد کرد. با عقب‌گرد از سمت والدین، اکنون مسیر کامل را در اختیار دارید.



مجموعه معاملاتی که راما باید انجام بدهد در زیر آمده است.

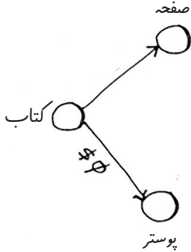


تا اینجا، از عبارت کوتاه‌ترین مسیر به معنای واقعی کلمه‌ی آن استفاده کرده‌ام؛ محاسبه‌ی کوتاه‌ترین مسیر بین دو مکان یا بین دو نفر. امیدوارم این مثال به شما نشان داده باشد که کوتاه‌ترین مسیر نباید درباره‌ی فاصله‌ی فیزیکی باشد. این می‌تواند در مورد به حداقل رساندن چیزی باشد. راما در این مورد می‌خواست مقدار پولی را که خرج می‌کند به حداقل برساند. با تشکر از دایجسترا!

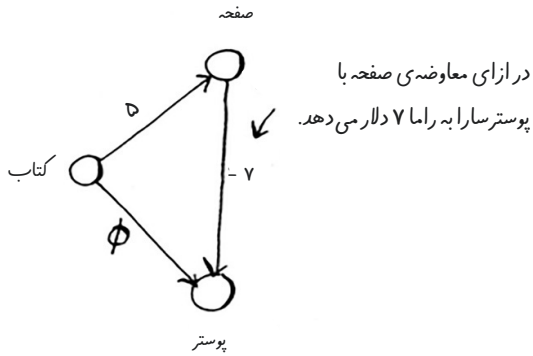
## 1. minimizing

## یال‌های با وزن منفی<sup>۱</sup>

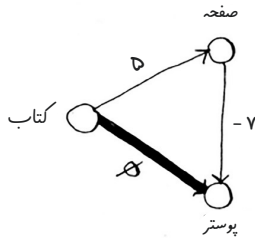
در این مثال، الکس پیشنهاد داد که کتاب را با دو آیتم معامله کند.



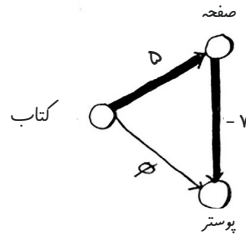
فرض کنید سارا پیشنهاد می‌دهد که صفحه را با پوستر معاوضه کند، و او به راما ۷ دلار دیگر می‌دهد. انجام این معامله برای راما هیچ هزینه‌ای ندارد. در عوض، او ۷ دلار به دست می‌آورد. در گراف چگونه باید نشان داد؟



یال میان صفحه و پوستر وزن منفی دارد! اگر راما این معامله را انجام بدهد، ۷ دلار دریافت می‌کند. حالا راما دو راه برای رسیدن به پوستر دارد.



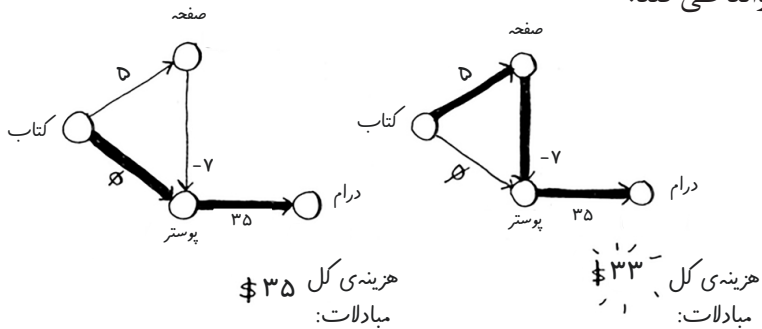
از این مسیر راما پولی گیرش نمی‌آید.



از این مسیر راما دو دلار به دست می‌آورد.

### 1. Negative-weight edges

بنابراین انجام معامله‌ی دوم منطقی است - راما از این طریق ۲ دلار پس می‌گیرد! حالا اگر یادتان باشد راما می‌تواند پوستر را با طبل عوض کند. دو مسیر هست که می‌تواند طی کند.

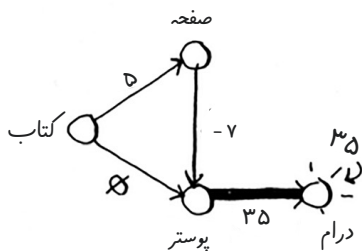


مسیر دوم برای او ۲ دلار کمتر هزینه دارد، بنابراین او باید آن مسیر را طی کند، درست است؟ خب حدس بزنید چه اتفاقی افتاده؟ اگر الگوریتم دایجسترا را روی این گراف اجرا کنید، راما مسیر اشتباهی را در پیش خواهد گرفت. مسیرش طولانی‌تر می‌شود. اگر یال‌هایی با وزن منفی دارید، نمی‌توانید از الگوریتم دایجسترا استفاده کنید. یال‌های با وزن منفی الگوریتم را خراب می‌کنند. بیایید بینیم وقتی الگوریتم دایجسترا را در این مورد اجرا می‌کنید چه اتفاقی می‌افتد. ابتدا جدولی از هزینه‌ها ایجاد کنید.

صفحه	۵
پوستر	۰
درام	∞

هزینه

در مرحله‌ی بعد، گره‌ی کم هزینه را پیدا کنید و هزینه‌ها را برای همسایگانش به روز کنید. در این مورد، پوستر کم هزینه‌ترین گره است. بنابراین، طبق الگوریتم دایجسترا، هیچ راه ارزان‌تری از پرداخت صفر دلار برای رسیدن به پوستر وجود ندارد (می‌دانید که چنین چیزی اشتباه است!). به هر حال، بیایید هزینه‌ها را برای همسایگانش به روز



صفحه	۵
پوستر	<del>۸</del>
درام	۳۵

کنیم.

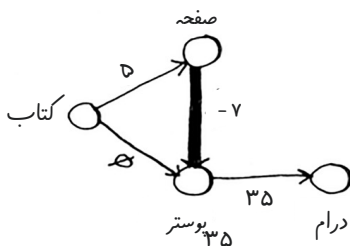
هزینه

خب، درام‌ها اکنون ۳۵ دلار قیمت دارند.

صفحه	۵
<del>پوستر</del>	<del>۸</del>
درام	۳۵

بیا باید ارزان‌ترین گرهی بعدی را که پیش از این پردازش نشده است، پیدا کنیم.

هزینه‌ها را برای همسایگان خود به روز کنید.



صفحه	۵
<del>پوستر</del>	<del>۸</del>
درام	۳۵

هزینه

علی‌رغم اینکه پیش از این گرهی پوستر را پردازش کرده‌اید، هزینه‌ی آن را به روز می‌کنید. این یک هشدار جدی است. هنگامی که یک گره را پردازش می‌کنید، به این معنی است که هیچ راه ارزان‌تری برای رسیدن به آن گره وجود ندارد. با این حال



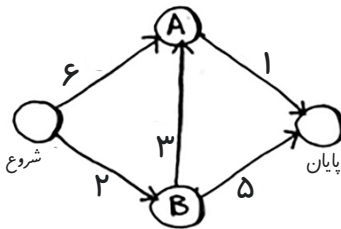
الان شما یک راه ارزان تر برای پوستر پیدا کردید! درام ها هیچ همسایه ای ندارند، بنابراین این نقطه پایان الگوریتم است. در تصویر زیر هزینه های نهایی آمده است.

صفحه	۵
پوستر	-۲
درام	۳۵

هزینه های نهایی

درام ها ۳۵ دلار هستند. مسیری را می شناسید که فقط ۳۳ دلار هزینه دارد، اما الگوریتم دایجسترا آن را پیدا نکرد. الگوریتم دایجسترا فرض می کند که چون شما گرهی پوستر را پردازش می کنید، راه سریع تری برای رسیدن به آن گره وجود ندارد. این فرض تنها در صورتی کارساز است که یال هایی با وزن منفی نداشته باشید. بنابراین نمی توانید از یال های وزن منفی با الگوریتم دایجسترا استفاده کنید. اگر می خواهید کوتاه ترین مسیر را در گرافی پیدا کنید که دارای یال هایی با وزن منفی است، چنین الگوریتمی برای آن وجود دارد! نام این الگوریتم بلمن-فورد نام دارد. الگوریتم بلمن-فورد خارج از محدوده ی این کتاب است، اما می توانید توضیحات خیلی خوبی درباره ی این الگوریتم در اینترنت پیدا کنید.

## پیاده سازی



بیباید ببینیم که چگونه الگوریتم دایجسترا را با کد پیاده سازی کنیم. از این گراف برای این مثال استفاده خواهیم کرد.

برای کدنویسی این مثال، به سه جدول هش نیاز دارید.

شروع	A	۶
	B	۲
A	FIN	۱
B	A	۳
	FIN	۵
پایان	-	

گراف

A	۶
B	۲
پایان	$\infty$

هزینه

A	شروع
B	شروع
FIN	-

والد

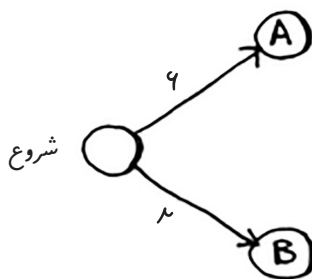
با پیشرفت الگوریتم، هزینه‌ها و جدول‌های والد‌ها را به‌روز می‌کنید. ابتدا باید گراف را پیاده‌سازی کنید. از جدول هش مانند آن چه در فصل ۶ انجام دادید استفاده می‌کنید:

```
graph = { }
```

در فصل پیش، تمام همسایگان یک گره را مانند کد زیر در جدول هش ذخیره کردید:

```
graph["you"] = ["alice", "bob", "claire"]
```

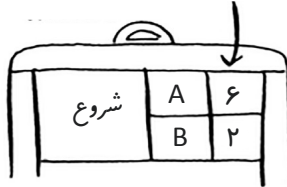
اما این بار باید همسایه‌ها و هزینه‌ی رسیدن به آن همسایه را ذخیره کنید. برای مثال، Start دو همسایه دارد: A و B.



وزن آن یال‌ها را چگونه نشان می‌دهید؟ چرا صرفاً از یک جدول هش دیگر استفاده نکنید؟

```
graph["start"]={}  
graph["start"]["a"] = 6  
graph["start"]["b"] = 2
```

این جدول هش جدول‌های هش  
بیشتری در خود دارد



بنابراین `graph["start"]` یک جدول هش است. می‌توانید همه‌ی همسایگان `Start` را به این شکل دریافت کنید:

```
>>>print graph["start"].keys()  
["a", "b"]
```

یک یال از `Start` به `A` و یک یال از `Start` به `B` وجود دارد. اگر بخواهید وزن آن یال‌ها را بیابید چه؟

```
>>> print graph["start"]["a"]  
2  
>>> print graph["start"]["b"]  
6
```

بیابید گره‌های دیگر و همسایگان آن‌ها را به گراف اضافه کنیم:  
`graph["a"] = {}`

```
graph["a"]["fin"] = 1
graph["b"] = {}
graph["b"]["a"] = 3
graph["b"]["fin"] = 5
graph["fin"] = {}
```

گره‌ی پایانی هیچ همسایه‌ای ندارد ←

جدول هش گراف کامل به این شکل است.

شروع	A	۶
	B	۲
A		۱
B	A	۳
		۵
پایان		—

این ها همگی جدول هش هستند

گراف

در مرحله‌ی بعد به یک جدول هش برای ذخیره‌ی هزینه‌های هر گره نیاز دارید.

A	۶
B	۲
پایان	∞

هزینه

هزینه‌ی یک گره مدت زمانی است که طول می‌کشد تا از نقطه‌ی شروع به آن گره برسید. شما می‌دانید که از شروع تا گره‌ی B ۲ دقیقه و تا گره‌ی A ۶ دقیقه طول می‌کشد. (اگرچه ممکن است مسیری را پیدا کنید که زمان کمتری می‌برد). از زمان مورد نیاز برای رسیدن به نقطه‌ی پایان اطلاعی ندارید. اگر هنوز هزینه را نمی‌دانید، بی‌نهایت را کنار می‌گذارید. آیا می‌توانید بی‌نهایت را به زبان پایتون بنویسید؟ بله می‌توانید:

```
infinity = float("inf")
```

در اینجا کدی برای ایجاد جدول هزینه‌ها آمده است:

```
infinity = float("inf")
costs = {}
costs["a"] = 6
costs["b"] = 2
costs["fin"] = infinity
```

همچنین به جدول هش دیگری برای والدها نیاز دارید:



A	شروع
B	شروع
پایان	-

والد

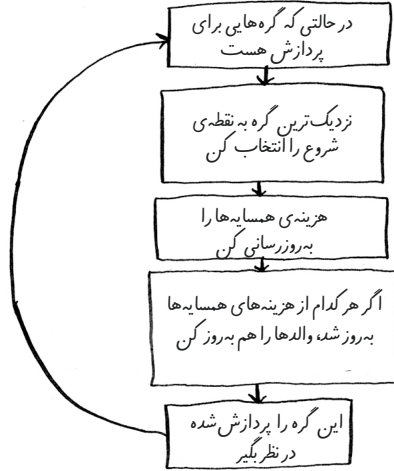
در اینجا کد برای ایجاد جدول هش والدها آمده است:

```
parents = {}
parents["a"] = "start"
parents["b"] = "start"
parents["fin"] = None
```

در نهایت، شما به یک آرایه نیاز دارید تا تمام گره‌هایی را که قبلاً پردازش کرده‌اید را در نظر داشته باشد، چرا که نیازی به پردازش بیش از یک بار یک گره ندارید:

```
processed = []
```

تمام تنظیمات همین است. حالا بیایید به الگوریتم نگاه کنیم.



ابتدا کد را به شما نشان می‌دهم و سپس آن را مرور می‌کنم.

```

node = find_lowest_cost_node(costs) <----- گره‌ای با کم‌ترین هزینه که پردازش نشده را پیدا می‌کند
while node is not None: <----- اگر تمام گره‌ها پردازش شده باشند، کار این حلقه به پایان می‌رسد.
    cost = costs[node]
    neighbors = graph[node]

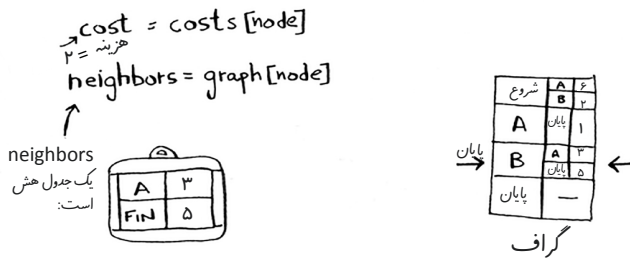
    for n in neighbors.keys(): <----- از تمام همسایه‌های این گره عبور می‌کند.
        new_cost = cost + neighbors[n] <----- اگر رسیدن به این همسایه کم‌هزینه‌تر است
        if costs[n] > new_cost: <----- با انتخاب این گره ...
            costs[n] = new_cost <----- هزینه‌ی این گره به‌روز می‌شود.
            parents[n] = node <----- این گره والد جدید برای این همسایه می‌شود.
    processed.append(node) <----- گره به عنوان پردازش شده علامت‌گذاری می‌شود.
    node = find_lowest_cost_node(costs) <----- بال بعدی را برای پردازش و ایجاد حلقه پیدا می‌کند.
  
```

این همان الگوریتم دایجسترا در پایتون است! من کد تابع را بعداً به شما نشان خواهم داد. ابتدا، بیایید کد الگوریتم `find_lowest_cost_node` را در عمل ببینیم.

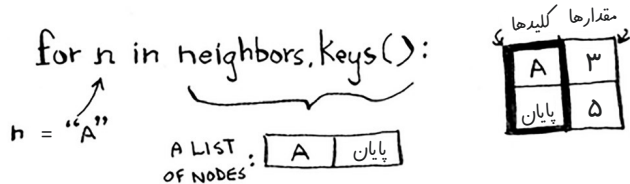
این گره را با کمترین هزینه پیدا کنید.



هزینه و همسایه های آن گره را پیدا کنید.



در همسایه ها حلقه بزیند.



هر گره هزینه ای دارد. هزینه به این معنی است که از ابتدا چقدر طول می کشد تا به آن گره برسید. در اینجا محاسبه می کنید که چقدر طول می کشد تا به گره A برسید به شرط آن که مسیر شما به جای Start > node A، به این ترتیب باشد:

$$\text{Start} > \text{node B} > \text{node A}$$

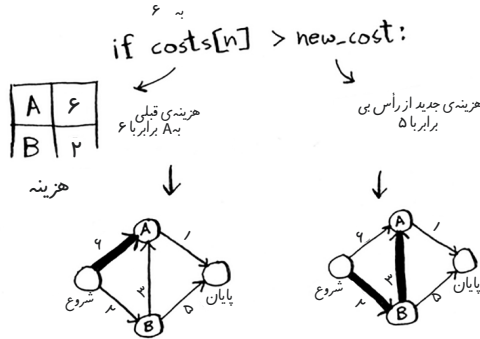
$$\text{new\_cost} = \text{cost} + \text{neighbors}[n]$$

↑  
COST OF "B", i.e. ۲

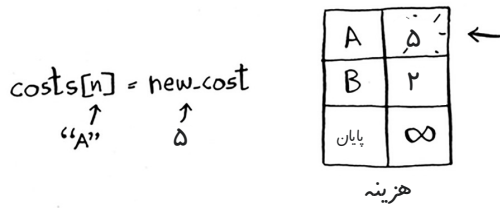
↓  
DISTANCE FROM B TO A: ۳

} new\_cost = ۲ + ۳ = ۵

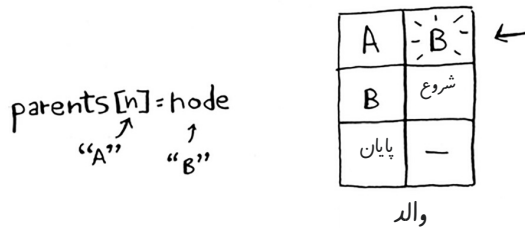
بیا ببینیم هزینه‌ها را با هم مقایسه کنیم.



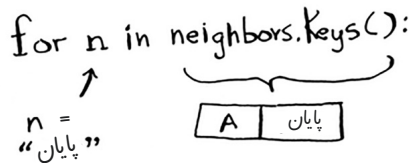
مسیر کوتاه‌تری برای گرهی A پیدا کردید! هزینه را به روز کنید.



مسیر جدید از گرهی B می‌گذرد، بنابراین B را به عنوان والد جدید در نظر بگیرید.



خوب، شما به بالای حلقه بازگشته‌اید. همسایه‌ی بعدی for گرهی Finish است.



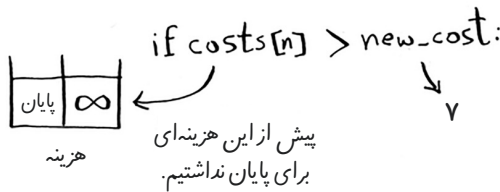


اگر از گرهی B عبور کنید چقدر طول می‌کشد تا به گرهی Finish برسید؟

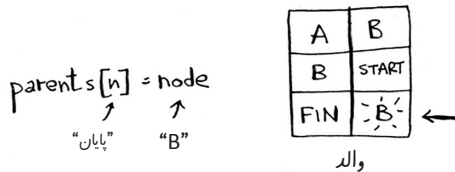
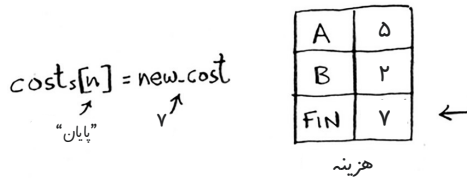
$$\text{new\_cost} = \text{cost} + \text{neighbors}[n]$$

$\downarrow$                        $\downarrow$                       }  $2+5=7$   
 ۲                      فاصله‌ی B تا پایان: ۵

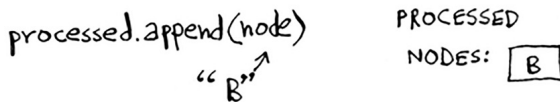
۷ دقیقه طول می‌کشد. هزینه‌ی قبلی بی‌نهایت دقیقه بود و ۷ دقیقه زمان کمتری از آن است.



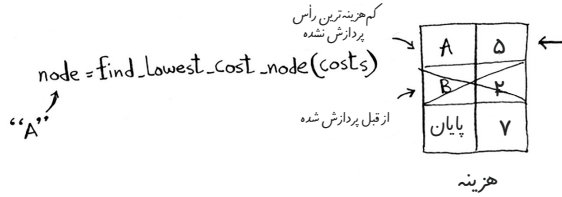
هزینه‌ی جدید و والد جدید را برای گرهی Finish در نظر بگیرید.



بسیار خوب، شما هزینه‌های همه‌ی همسایگان گرهی B را به روز کردید. این گره را به عنوان پردازش شده علامت بزنید.



گره‌ی بعدی را برای پردازش پیدا کنید.



هزینه و همسایگان گره‌ی A را به دست بیاورید.

cost = costs[node]

۵

neighbors = graph[node]



گره‌ی A فقط یک همسایه دارد: گره‌ی Finish.

for n in neighbors.keys():

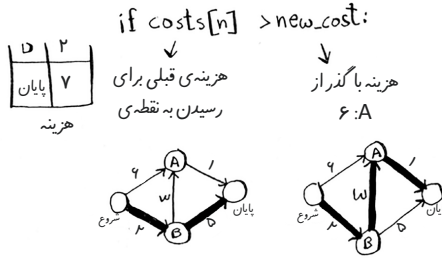
↑

“پایان”

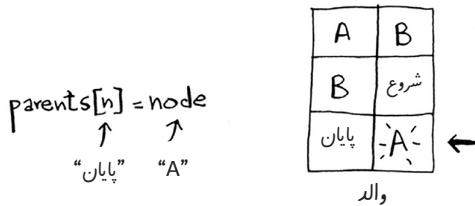
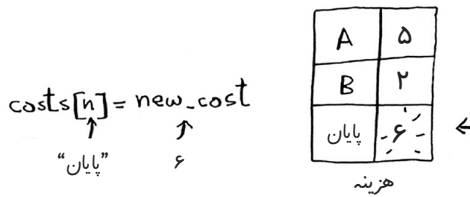
پایان

در حال حاضر ۷ دقیقه طول می‌کشد تا به گره‌ی Finish برسید. اگر از گره‌ی A عبور کنید چقدر طول می‌کشد تا به آنجا برسید؟

$$\left. \begin{array}{l} \text{new\_cost} = \text{cost} + \text{neighbors}[n] \\ \downarrow \qquad \qquad \downarrow \\ \text{هزینه‌ی رسیدن به A از نقطه‌ی شروع: ۵} \qquad \text{فاصله‌ی A تا پایان: ۱} \end{array} \right\} ۵+۱=۶$$



از گره‌ی A سریع‌تر می‌توان به Finish رسید! بیایید هزینه و والدها را به روز کنیم.



هنگامی که تمام گره‌ها را پردازش کردید، الگوریتم به پایان می‌رسد. امیدوارم این راهنما به شما کمک کند تا الگوریتم را کمی بهتر درک کرده باشید. یافتن کم‌هزینه‌ترین گره با تابع `find_lowest_cost_node` بسیار آسان است. کد آن در ادامه آمده است:

```
def find_lowest_cost_node(costs):
    lowest_cost = float("inf")
    lowest_cost_node = None

    for node in costs:
        cost = costs[node]

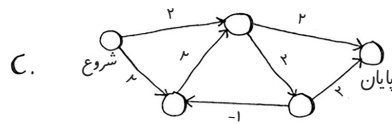
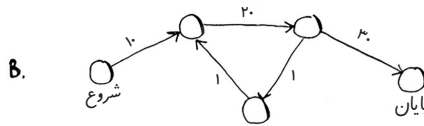
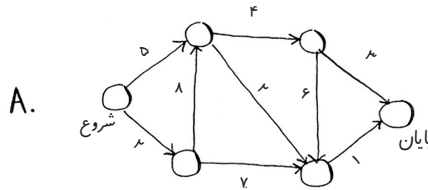
        if cost < lowest_cost and node not in processed:
            lowest_cost = cost
            lowest_cost_node = node
```

هر node را بررسی می‌کند ←  
اگر تا اینجای کار کمترین هزینه باشد و هنوز پروسس نشده باشد  
آن را کم‌ترین هزینه‌ی جدید ←  
در نظر می‌گیرد.

`return lowest_cost_node`

### تمرین

۷.۱ در هر یک از این گراف‌ها، وزن کوتاه‌ترین مسیر ابتدا تا انتها چقدر است؟



### جمع‌بندی

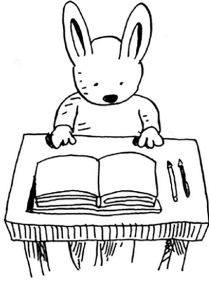
- جست‌وجوی سطح اول برای محاسبه‌ی کوتاه‌ترین مسیر برای یک گراف بدون وزن استفاده می‌شود.
- الگوریتم دایجسترا برای محاسبه‌ی کوتاه‌ترین مسیر برای یک گراف وزن‌دار استفاده می‌شود.
- الگوریتم دایجسترا زمانی مفید است که همه‌ی وزن‌ها مثبت باشند.
- اگر وزن‌های منفی دارید، از الگوریتم بلمن-فورد استفاده کنید.



- یاد می‌گیرید چگونه از پس غیرممکن‌ها بر بیایید: مسئله‌هایی که راه حل الگوریتمی سریعی ندارند (مسئله‌های ان‌پی کامل<sup>۲</sup>).
- یاد می‌گیرید در مواجهه با این مسئله‌ها چگونه آن‌ها را شناسایی کنید و وقت خود را برای یافتن یک الگوریتم سریع برای آنها تلف نکنید.
- شما در مورد الگوریتم‌های تقریبی یاد می‌گیرید که می‌توانید از آن‌ها برای یافتن سریع یک راه حل تقریبی برای یک مسئله NP-complete استفاده کنید.
- در مورد استراتژی حریصانه که یک استراتژی ساده‌ای برای حل مسئله‌ی است، یاد می‌گیرید.

---

1. greedy algorithm  
2. NP-complete



## مسئله‌ی برنامه‌ریزی برای کلاس درس

فرض کنید تنها یک اتاق برای کلاس درس دارید و می‌خواهید بیشترین تعداد ممکن کلاس را در آنجا برگزار کنید. لیستی از کلاس‌ها در اختیار دارید:

کلاس	شروع	پایان
هنر	۹ AM	۱۰ AM
انگلیسی	۹:۳۰ AM	۱۰:۳۰ AM
ریاضی	۱۰ AM	۱۱ AM
کامپیوتر	۱۰:۳۰ AM	۱۱:۳۰ AM
موسیقی	۱۱ AM	۱۲ PM

شما نمی‌توانید همه‌ی این کلاس‌ها را در آنجا برگزار کنید، چون برخی از آن‌ها با یکدیگر هم‌پوشانی زمانی دارند.



شما می‌خواهید تا جایی که ممکن است کلاس‌های بیشتری را در این اتاق برگزار کنید. چگونه مجموعه‌ای از کلاس‌ها را انتخاب می‌کنید به شرط آنکه بزرگ‌ترین تعداد کلاس‌های ممکن را داشته باشید؟

مسئله‌ی دشواری به نظر می‌رسد، درست است؟ در واقع، آن قدر الگوریتم آسانی دارد که ممکن است شما را شگفت‌زده کند. نحوه‌ی عملکرد آن در اینجا آمده است:

۱. کلاسی را انتخاب کنید که زودتر به اتمام می‌رسد. این اولین کلاسی است که در این اتاق برگزار می‌کنید.

۲. حالا باید کلاسی را انتخاب کنید که بعد از اولین کلاس شروع شود. دوباره کلاسی را انتخاب کنید که زودتر تمام می‌شود. این دومین کلاسی است که برگزار می‌کنید.

به این کار ادامه داده تا در نهایت به پاسخ برسید! بیایید آن را امتحان کنیم. کلاس هنر قبل از همه، در ساعت ۱۰ صبح به پایان می‌رسد، بنابراین این یکی از کلاس‌هایی است که شما انتخاب می‌کنید.

هنر	۹ AM	۱۰ AM	✓
انگلیسی	۹:۳۰ AM	۱۰:۳۰ AM	
ریاضی	۱۰ AM	۱۱ AM	
کامپیوتر	۱۰:۳۰ AM	۱۱:۳۰ AM	
موسیقی	۱۱ AM	۱۲ PM	

اکنون به کلاسی نیاز دارید که بعد از ساعت ۱۰ صبح شروع شده و زودتر از باقی کلاس‌ها به پایان می‌رسد.

هنر	۹ AM	۱۰ AM	✓
انگلیسی	۹:۳۰ AM	۱۰:۳۰ AM	✗
ریاضی	۱۰ AM	۱۱ AM	✓
کامپیوتر	۱۰:۳۰ AM	۱۱:۳۰ AM	
موسیقی	۱۱ AM	۱۲ PM	

زبان انگلیسی به دلیل تداخل با هنر از فهرست مورد نظر ما حذف می‌شود، اما کلاس ریاضی مناسب است. در نهایت، علوم کامپیوتر با ریاضی تداخل زمانی دارد، اما کلاس موسیقی مشکلی ندارد.







## مسئله‌ی کوله پشتی

فرض کنید شما یک دزد طمع‌کار هستید. با یک کوله‌پشتی در یک فروشگاه هستید، و همه‌ی اقلام زیر برای دزدی موجود است اما تنها می‌توانید چیزهایی را که در کوله‌پشتی جا می‌شوند بدزدید. کوله‌پشتی می‌تواند ۳۵ پوند را در خود جای بدهد.

شما سعی می‌کنید ارزش اقلامی را که در کوله‌پشتی خود قرار می‌دهید به حداکثر برسانید. از چه الگوریتمی استفاده می‌کنید؟

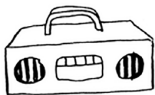


با استراتژی حریصانه باز هم بسیار ساده خواهد بود:

۱. گران‌ترین چیزی را که در کوله‌پشتی شما جا می‌شود انتخاب کنید.

۲. گران‌ترین چیز بعدی را که در کوله‌پشتی شما جا می‌شود انتخاب کنید. و به همین ترتیب.

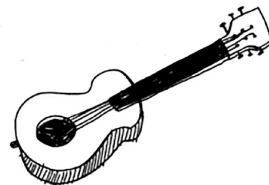
تنها مشکل این است که این الگوریتم اینجا به کار نمی‌آید! به عنوان مثال، فرض کنید سه چیز برای سرقت موجود است.



ضبط صوت  
۳۰۰۰ دلار  
۳۰ پوند

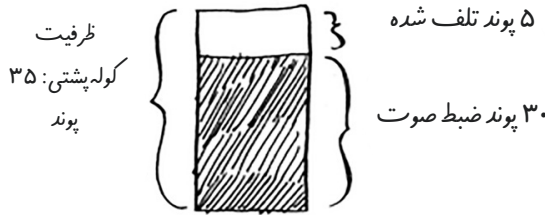


لپ‌تاپ  
۲۰۰۰ دلار  
۲۰ پوند



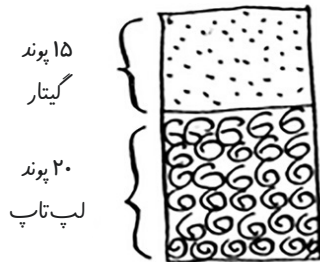
گیتار  
۱۵۰۰ دلار  
۱۵ پوند

در کوله‌پشتی ۳۵ پوند آیتم را می‌توان قرار داد. سیستم ضبط صوت از تمامی وسایل باارزش‌تر است، بنابراین شما آن را می‌دزدید. اکنون برای هیچ چیز دیگری جای خالی باقی نمانده است.



ارزش: ۳۰۰۰ دلار

شما ۳۰۰۰ دلار کالاگیرتان آمده است. اما صبر کنید! اگر به جای آن لپ‌تاپ و گیتار را انتخاب می‌کردید، می‌توانستید معادل ۳۵۰۰ دلار سرقت کنید!



ارزش: ۳۵۰۰ دلار

واضح است که در اینجا استراتژی حریصانه راه حل بهینه را به شما نمی‌دهد. با این حال بسیار نزدیک به آن است. در فصل بعدی نحوه‌ی محاسبه‌ی راه حل صحیح را توضیح می‌دهم. اما اگر سارقی در یک مرکز خرید باشید، وضعیت ایده‌آل برای شما اهمیت چندانی ندارد. در چنین شرایطی «به نسبت خوب» کفایت می‌کند. نتیجه‌ای که از مثال دوم می‌توان گرفت: گاهی، بی‌نقص بودن دشمن خوب بودن است. گاهی اوقات تنها چیزی که نیاز دارید یک الگوریتم است که مشکل را به خوبی حل کند. و اینجاست که الگوریتم‌های حریصانه می‌درخشند، زیرا نوشتن آن‌ها ساده است و

معمولاً با نتایج ایده‌آل چندان تفاوتی ندارند.

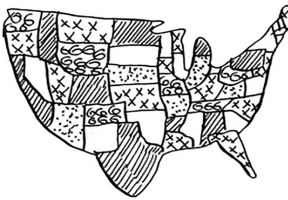
## تمرین

**۸.۱** شما برای یک شرکت مبلمان کار می‌کنید و باید تولیدات این شرکت را به سراسر کشور ارسال کنید. شما باید بسته‌ها را در کامیون بار بزنید. بسته‌ها اندازه‌های مختلفی دارند و شما سعی می‌کنید حداکثر استفاده را از فضای کامیون داشته باشید. بسته‌ها را چگونه انتخاب می‌کنید تا از فضای داخل کامیون به شکل بهینه استفاده کنید؟ از استراتژی حریصانه استفاده کنید. آیا این الگوریتم شما را به راه حل بهینه می‌رساند؟

**۸.۲** شما به اروپا می‌روید و هفت روز فرصت دارید تا جایی که زمان اجازه بدهد از مکان‌های گوناگون بازدید کنید. شما یک ارزش نقطه‌ای<sup>۱</sup> به هر مورد اختصاص می‌دهید (میزان اشتیاق به دیدن آن) و تخمین می‌زنید که بازدید هر کدام چقدر طول می‌کشد. چگونه می‌توانید مجموع ارزش را به حداکثر برسانید (بازدید از تمام مکان‌هایی که واقعاً دوست دارید از آن‌ها بازدید کنید)؟ با یک استراتژی حریصانه. آیا این راه حل بهینه به شما می‌دهد؟

بیا بید به آخرین نمونه نگاه کنیم. این مثالی است که در آن استفاده از الگوریتم‌های حریصانه کاملاً ضروری است.

## مسئله‌ی پوشش مجموعه<sup>۲</sup>



فرض کنید می‌خواهید برنامه‌ای رادیویی داشته باشید و این برنامه در هر ۵۰ ایالت برای شنوندگان در دسترس باشد. باید بررسی کنید که باید این برنامه را از کدام ایستگاه‌ها مخابره کرد تا به گوش همه‌ی مخاطب‌ها برسد. پخش در هر ایستگاه هزینه دارد، بنابراین سعی می‌کنید تعداد ایستگاه‌هایی را که از آن مخابره می‌کنید به حداقل برسانید. شما لیستی از ایستگاه‌ها دارید.

1. Point value
2. The set-covering problem

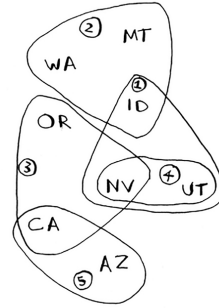
دسترسی در ایستگاه رادیو

KONE	ID,NV,UT
KTWO	WA,ID,MT
KTHREE	OR,NV,CA
KFOUR	NV,UT
KFIVE	CA,AZ

و غیره...

هر ایستگاه یک منطقه را پوشش می دهد و البته هم پوشانی هم وجود دارد.

چگونه می توانید کوچک ترین مجموعه ایستگاه را انتخاب کنید به شرط آنکه برنامه در تمامی ۵۰ ایالت مخابره شود؟ آسان به نظر می رسد، این طور نیست؟ در عمل کاری بسیار



دشوار است. در اینجا شیوه ی انجام آن آمده است:

۱. تمام زیرمجموعه های ممکن از ایستگاه ها را فهرست کنید. به این زیرمجموعه ها مجموعه ی توانی گفته می شود و تعداد آن ها  $2^n$  است.



و غیره...

و غیره...

و غیره...

۲. از میان آن ها، مجموعه ای را انتخاب کنید که کمترین تعداد ایستگاه را با فرض پوشش تمامی ۵۰ ایالت شامل می شود.

مشکل این است که محاسبه ی هر زیرمجموعه ی ممکن از ایستگاه ها زمان بر است. زمان  $O(2^n)$  طول می کشد، چون  $2^n$  ایستگاه وجود دارد. اگر مجموعه ی کوچکی از ۵ تا ۱۰ ایستگاه دارید، این کار ممکن است. اما با تمام مثال های اینجا، به این فکر کنید که اگر آیتم های زیادی داشته باشید چه پیش می آید. اگر ایستگاه های بیشتری داشته باشید خیلی بیشتر طول می کشد. فرض کنید بتوانید در هر ثانیه ۱۰ زیرمجموعه حساب کنید. هیچ الگوریتمی نیست که بتواند آن را با سرعتی مناسب حل کند! چاره چیست؟

## 1. Power set

تعداد ایستگاه	زمان لازم
۵	۳.۲ ثانیه
۱۰	۱۰۲.۴ ثانیه
۳۲	۱۳.۶ سال
۱۰۰	$10^4 \times 4$ سال

## الگوریتم‌های تقریبی<sup>۱</sup>

الگوریتم‌های حریصانه نجات‌بخش هستند! در اینجا یک الگوریتم حریصانه معرفی می‌شود که تقریباً به همین صورت است:

۱. ایستگاهی را انتخاب کنید که برای ایالت‌هایی که هنوز امکان دریافت برنامه‌ی رادیویی را ندارند پوشش حداکثری داشته باشد. مشکلی نیست اگر ایستگاهی برای برخی از ایالت‌ها با ایستگاه‌های قبلی هم‌پوشانی داشته باشد.

۲. این کار را تا زمانی که تمام ایالت‌ها تحت پوشش قرار بگیرند، ادامه بدهید.

به این الگوریتم تقریبی می‌گویند. در شرایطی که محاسبه‌ی راه حل دقیق زمان زیادی می‌برد، یک الگوریتم تقریبی کارساز خواهد بود. الگوریتم‌های تقریبی بر اساس معیارهای زیر داوری می‌شوند:

- چقدر سریع هستند
- چقدر به راه حل بهینه نزدیک هستند

الگوریتم‌های حریصانه انتخاب خوبی هستند زیرا نه تنها دستیابی به آن‌ها ساده است، بلکه به این معنی است که معمولاً سریع هم اجرا می‌شوند. در این حالت، الگوریتم

حریصانه در زمان  $O(n^2)$  اجرا می‌شود. در اینجا  $n$  تعداد ایستگاه‌های رادیویی است. بیا بید کد این مسئله را بررسی کنیم.

## کد

در این مثال، برای سادگی مسئله تنها از زیر مجموعه‌ی ایالت‌ها و ایستگاه‌ها استفاده می‌کنم.

ابتدا فهرستی تهیه کنید از ایالت‌هایی که قصد پوشش آن‌ها را دارید:

```
states_needed = set(["mt", "wa", "or", "id", "nv", "ut",
                    "ca", "az"])
```

یک آرایه وارد کرده و به‌شما set برمی‌گرداند.

برای این کار از یک set استفاده کردم. یک set مانند یک لیست است، با این تفاوت که در set هر آیتیم تنها یک بار در مجموعه ظاهر می‌شود. Setها نمی‌توانند تکراری باشند. برای مثال، فرض کنید این لیست را دارید:

```
>>> arr = [1, 2, 2, 3, 3, 3]
```

و شما آن را به یک set تبدیل کردید:

```
>>> set(arr)
set([1, 2, 3])
```

۱، ۲، و ۳ در set فقط یک‌بار نمایش داده می‌شوند.

$[1, 2, 2, 3, 3, 3] \xrightarrow{\text{تبدیل به set}} (1, 2, 3) \xrightarrow{\text{SET}}$

شما همچنین به لیست ایستگاه‌هایی که از بین آن‌ها انتخاب می‌کنید نیاز دارید. برای این منظور از هش استفاده کردم:

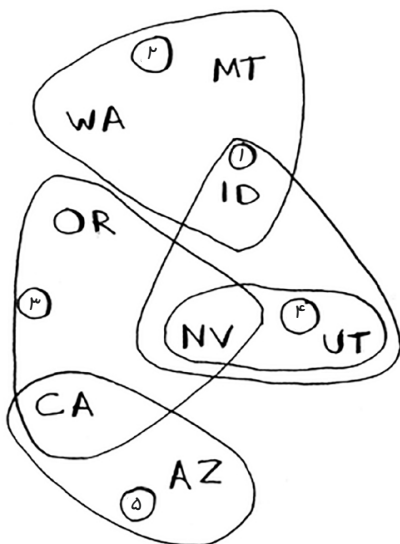
```
stations = {}
stations["kone"] = set(["id", "nv", "ut"])
stations["ktwo"] = set(["wa", "id", "mt"])
stations["kthree"] = set(["or", "nv", "ca"])
stations["kfour"] = set(["nv", "ut"])
stations["kfive"] = set(["ca", "az"])
```

کلیدها نام ایستگاه‌ها هستند و مقادارها ایالت‌هایی هستند که آن ایستگاه‌ها پوشش می‌دهند.

بنابراین در این مثال، ایستگاه kone آیداهو، نوادا و یوتا را پوشش می‌دهد. تمام مقادیر نیز set هستند. به زودی می‌بینید که در همه‌ی شرایط استفاده از set زندگی شما را آسان‌تر می‌کند.

در نهایت، نیاز است تا set نهایی ایستگاه‌هایی که استفاده می‌کنید در جایی نگهداری بشود:

```
final_stations = set()
```



### محاسبه‌ی پاسخ

اکنون باید حساب کنید که از چه ایستگاه‌هایی استفاده می‌کنید. به تصویر سمت چپ نگاه کرده و ببینید آیا می‌توانید پیش‌بینی کنید که از چه ایستگاه‌هایی باید استفاده کرد.

ممکن است بیش از یک راه حل صحیح وجود داشته باشد. شما باید به هر یک از ایستگاه‌ها نگاه کرده و ایستگاهی را انتخاب کنید که بیشتر ایالت‌های پوشش نیافته را شامل شود. نام آن را `best_station`

در نظر می‌گیرم:

```
best_station = None
states_covered = set()

for station, states_for_station in stations.items():
```

States\_covered، سستی است از تمام ایالت‌های هنوز پوشش داده نشده‌ای که ایستگاه مورد اشاره آن‌ها را پوشش می‌دهد. حلقه‌ی for به شما این امکان را می‌دهد که بر روی هر ایستگاهی حلقه اعمال کنید تا ببینید کدام یک مناسب‌ترین ایستگاه است. بیایید به کد حلقه‌ی for نگاه کنیم:

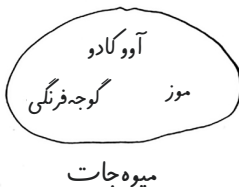
```
covered = states_needed & states_for_station <-----
if len(covered) > len(states_covered):
    best_station = station
    states_covered = covered
```

سینکس جدید به آن  
set intersection  
می‌گویند.

در اینجا یک خط کد عجیب و غریب می‌بینید:

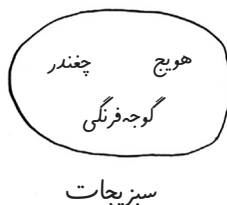
```
covered = states_needed & states_for_station
```

قضیه چیست؟



**بست**

فرض کنید یک بست از میوه‌ها دارید.

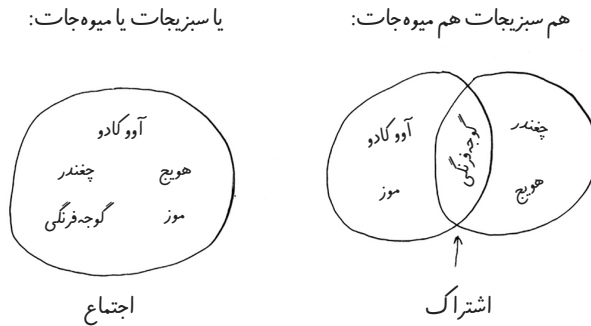


همچنین یک بست از سبزیجات دارید.

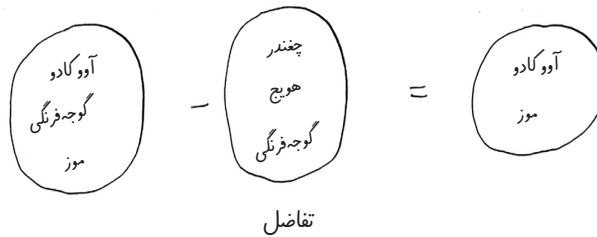


وقتی دو دسته بست داشته باشید، می‌توانید کارهای جالبی با آن‌ها انجام دهید.

در ادامه کارهایی که می‌توانید با ست‌ها انجام دهید، آمده است.



چیزهایی که میوه هستند اما  
سبزیجات نیستند



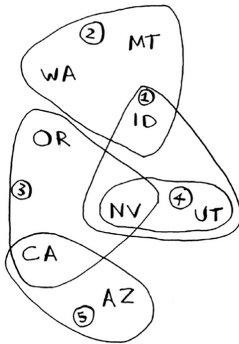
- اجتماع مجموعه<sup>۱</sup> به معنای «ترکیب هر دو مجموعه» است.
- اشتراک مجموعه<sup>۲</sup> به معنی «یافتن آیتم‌هایی است که در هر دو مجموعه ظاهر می‌شوند» (در این مورد، فقط گوجه‌فرنگی).
- تفاضل مجموعه<sup>۳</sup> به این معنی است که «اقدام یک مجموعه را از آیتم‌های مجموعه‌ی دیگر کم کنید.»

1. t  
2. Set intersection  
3. Set difference

به عنوان مثال:

```
>>> fruits = set(["avocado", "tomato", "banana"])
>>> vegetables = set(["beets", "carrots", "tomato"])
>>> fruits | vegetables <----- اجتماع مجموعه
set(["avocado", "beets", "carrots", "tomato", "banana"])
>>> fruits & vegetables <----- اشتراک مجموعه
set(["tomato"])
>>> fruits - vegetables <----- تفاضل مجموعه
set(["avocado", "banana"])
>>> vegetables - fruits <----- حاصل این عبارت چیست؟
```

### جمع بندی:



- سِت‌ها مانند لیست‌ها هستند، با این تفاوت که سِت‌ها نمی‌توانند عضو تکراری داشته باشند.
- می‌توانید چند عملیات جالب مانند اجتماع، اشتراک، تفاضل را روی سِت‌ها اعمال کنید.

### بازگشت به کد

بیا بید به مثال اصلی برگردیم.

این کد یک اشتراک مجموعه است:

```
covered = states_needed & states_for_station
```

covered مجموعه‌ای از ایالت‌ها است که در هر دوی states\_needed و states\_for\_station حضور دارد. بنابراین covered، set ایالت‌های بدون

پوشش است که این ایستگاه پوشش می‌دهد! بعد بررسی می‌کنید که آیا این ایستگاه ایالت‌های بیشتری را نسبت به `best_station` فعلی پوشش می‌دهد یا خیر:

```
if len(covered) > len(states_covered):
    best_station = station
    states_covered = covered
```

اگر چنین باشد، این ایستگاه `best_station` جدید است. در نهایت، پس از اتمام حلقه‌ی `for`، `best_station` را به لیست نهایی ایستگاه‌ها اضافه می‌کنید:

```
final_stations.add(best_station)
```

همچنین باید `states_needed` را به‌روز کنید. از آنجایی که این ایستگاه برخی از ایالت‌ها را پوشش می‌دهد، دیگر به این ایالت‌ها نیازی نیست:

```
states_needed -= states_covered
```

و شما حلقه می‌زنید تا `states_needed` خالی شود. در اینجا کد کامل این حلقه آمده است:

```
while states_needed:
    best_station = None
    states_covered = set()

    for station, states in stations.items():
        covered = states_needed & states

        if len(covered) > len(states_covered):
            best_station = station
            states_covered = covered

    states_needed -= states_covered
    final_stations.add(best_station)
```

در نهایت، می‌توانید `final_stations` را پرینت کنید، باید این نوشته را مشاهده کنید:

```
>>> print final_stations
set(['ktwo', 'kthree', 'kone', 'kfive'])
```

این همان چیزی است که انتظار داشتید؟ به جای ایستگاه‌های ۱، ۲، ۳ و ۵، می‌توانید ایستگاه‌های ۲، ۳، ۴ و ۵ را انتخاب کنید. بیایید زمان اجرای الگوریتم حریصانه را با الگوریتم دقیق مقایسه کنیم.

تعداد ایستگاه‌ها	$O(n!)$ الگوریتم دقیق	$O(n^2)$ الگوریتم حریصانه
۵	۳.۲ ثانیه	۲.۵ ثانیه
۱۰	۱۰۲.۴ ثانیه	۱۰ ثانیه
۳۲	۱۳.۶ سال	۱۰۲.۴ ثانیه
۱۰۰	$4 \times 10^{41}$ سال	۱۶.۶۷ دقیقه

### تمرین

هر یک از الگوریتم‌های زیر را بررسی کنید که آیا الگوریتم حریصانه هستند یا خیر.

۸.۳ مرتب‌سازی سریع

۸.۴ جست‌وجوی سطح اول

۸.۵ الگوریتم دایجسترا



### مسئله‌ی ان پی کامل

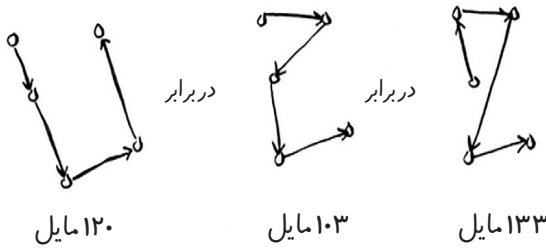
برای حل مسئله‌ی پوشش-مجموعه، باید تمامی مجموعه‌های ممکن را محاسبه می‌کردید.

1. Exact algorithm

شاید به یاد مسئله‌ی فروشنده‌ی دوره‌گرد فصل یک افتاده باشید. در این مسئله، یک فروشنده باید به پنج شهر مختلف سفر کند.



و او در تلاش است کوتاه‌ترین مسیری بیابد که از هر پنج شهر بگذرد. برای یافتن کوتاه‌ترین مسیر، ابتدا باید هر مسیر ممکن را محاسبه کنید.

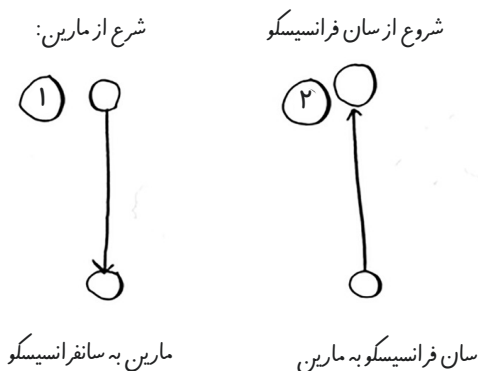


...

چند مسیر برای پنج شهر باید محاسبه کنید؟

## فروشنده‌ی دوره‌گرد، گام به گام

از جزء شروع می‌کنیم. تنها دو شهر را فرض کنید. دو مسیر برای انتخاب وجود دارد.



### مسیر یکسان یا متفاوت؟

ممکن است فکر کنید این باید همان مسیر باشد. در نهایت مگر مسیر  $SF > Marin$  با  $Marin > SF$  برابر نیست؟ نه لزوماً. برخی از شهرها (مانند سانفرانسیسکو) خیابان‌های یک طرفه‌ی زیادی دارند، بنابراین نمی‌توانید از راهی که آمده‌اید بازگردید. همچنین ممکن است مجبور شوید ۱ یا ۲ مایل از مسیر خود دور شوید تا یک ورودی به بزرگراه پیدا کنید. بنابراین این دو مسیر لزوماً یکسان نیستند.

ممکن است از خود بپرسید، «در مسئله‌ی فروشنده‌ی دوره‌گرد، آیا شهر خاصی وجود دارد که باید از آن شروع کنید؟» برای مثال، فرض کنید من یک فروشنده‌ی دوره‌گرد هستم و در سانفرانسیسکو زندگی می‌کنم و باید به چهار شهر دیگر سفر کنم. سانفرانسیسکو مبداء سفر من است.

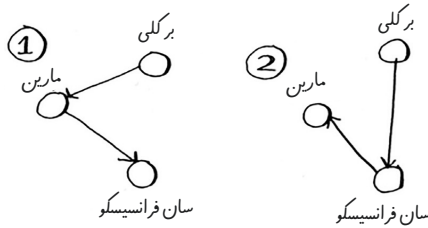
اما گاهی اوقات شهر مبداء تعیین نمی‌شود. فرض کنید شما در FedEx کار می‌کنید و

می‌خواهید بسته‌ای را در Bay Area تحویل دهید. این بسته از شیکاگو به یکی از ۵ مرکز می‌باشد. Bay Area منتقل می‌شود. سپس آن بسته بار کامیونی می‌شود تا برای تحویل بسته‌ها به محل‌های مختلف برود. به کدام مکان باید رفت؟ در اینجا مبدأ ناشناخته است. محاسبه‌ی مسیر بهینه و مبدأ فروشنده‌ی دوره‌گرد به عهده‌ی شماست. زمان اجرای هر دو نسخه یکسان است. اما اگر شهری به عنوان مبدأ مشخص نشود، مثال ساده‌تری می‌شود و برای همین با همین مثال کار را پیش می‌برم. دو شهر = دو مسیر ممکن.

### ۳ شهر

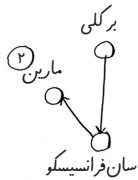
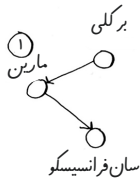
حالا فرض کنید یک شهر دیگر اضافه کنید. چند مسیر ممکن وجود دارد؟ اگر از برکلی شروع کنید، دو شهر دیگر هم برای بازدید دارید.

شروع از برکلی

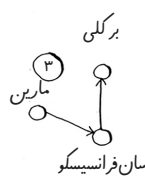


در مجموع شش مسیر وجود دارد، دو مسیر برای سفر از هر شهر.

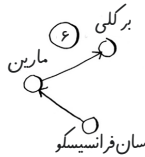
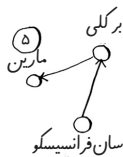
شروع از برکلی:



شروع از مارین:



شروع از سان فرانسیسکو:



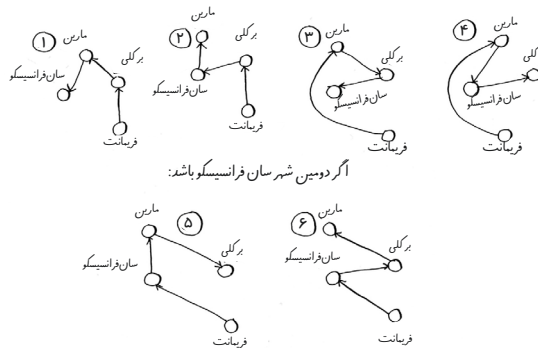
بنابراین سه شهر = شش مسیر ممکن.

#### ۴ شهر

بیا بید شهر فرمونت را اضافه کنیم. حالا فرض کنید سفر را از این شهر شروع می کنید.

شروع از

اگر دومین شهر مارین باشد: فریمانت: اگر دومین شهر برکلی باشد:



شش مسیر ممکن از مبداء فریمانت هست. نگاه کنید! آن ها بسیار شبیه به شش مسیری هستند که در زمانی که تنها سه شهر داشتید محاسبه کردید. با این تفاوت که در حال حاضر همه ی مسیرها یک شهر دیگر را هم شامل می شوند، فریمانت! یک الگو وجود دارد. فرض کنید چهار شهر هست، و فریمانت را شهر مبداء انتخاب می کنید. سه شهر باقی می ماند. و می دانید که اگر سه شهر باشد، شش مسیر مختلف برای رفت و آمد میان آن شهرها وجود دارد. اگر از فریمانت شروع کنید، شش مسیر ممکن است. همچنین می توانید از شهر دیگری شروع کنید.

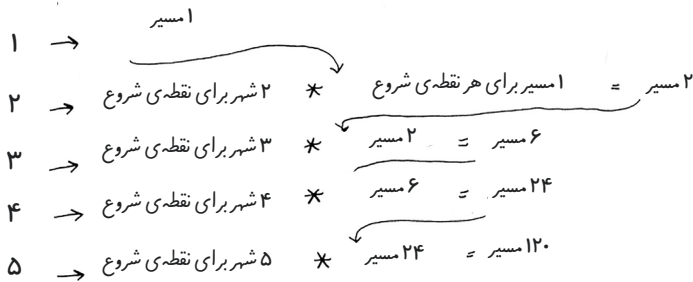
$$\begin{aligned}
 & \text{شروع از سان فرانسیسکو:} & \text{شروع از مارین:} \\
 & 6 \text{ مسیر ممکن} & = & 6 \text{ مسیر ممکن} \\
 & = & & = \\
 & \text{شروع از برکلی:} & & \\
 & 6 \text{ مسیر ممکن} & = & 6 \text{ مسیر ممکن}
 \end{aligned}$$



چهار گزینه برای شهر مبداء، با شش مسیر ممکن برای هر شهر مبداء  $4 * 6 = 24$  مسیر ممکن می‌شود.

الگوی مشخصی مشاهده می‌کنید؟ هر بار که شهر جدیدی اضافه می‌کنید، تعداد مسیرهایی را که باید حساب کنید افزایش می‌دهید.

تعداد شهرها



چند مسیر ممکن برای شش شهر وجود دارد؟  $720$  مسیر.  $5040$  مسیر برای ۷ شهر،  $40320$  مسیر برای ۸ شهر.

این همان تابع فاکتوریل است (به خاطر دارید در مورد آن در فصل ۳ خواندید؟). پس  $120 = 5!$  فرض کنید ۱۰ شهر باشد. چند مسیر ممکن وجود دارد؟  $3628800 = 10!$ . برای ۱۰ شهر باید بیش از ۳ میلیون مسیر ممکن را حساب کنید. همان‌طور که می‌بینید تعداد مسیرهای ممکن به سرعت زیاد می‌شود!

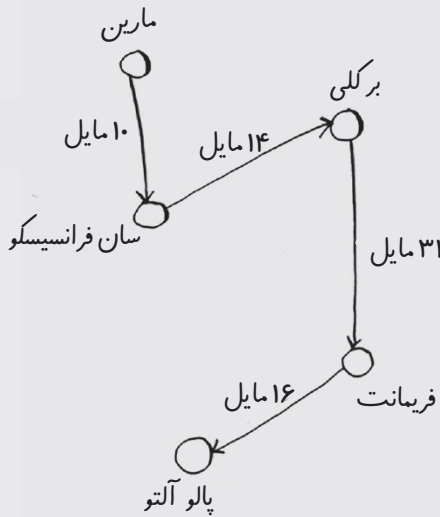
به همین دلیل در صورت داشتن تعداد زیادی شهر، محاسبه‌ی راه حل «صحیح» برای مسئله‌ی فروشنده‌ی دوره‌گرد غیرممکن است.

مسئله‌ی فروشنده‌ی دوره‌گرد و مسئله‌ی پوشش مجموعه در یک مورد مشترک هستند: شما هر راه حل ممکن را محاسبه می‌کنید و کوچک‌ترین/کوتاه‌ترین راه حل را انتخاب می‌کنید. هر دوی این مسئله‌ها ان‌پی کامل هستند.

### تقریب

یک الگوریتم تقریبی مناسب برای مسئله‌ی فروشنده‌ی دوره‌گرد دارای چه ویژگی‌هایی است؟ باید الگوریتم ساده‌ای باشد که مسیر کوتاهی پیدا کند. قبل از آنکه ادامه دهید، ببینید پاسخی می‌توانید بیابید.

روش کار من به این شکل است: به دلخواه یک شهر را به عنوان مبدأ انتخاب می‌کنم. سپس، هر بار که فروشنده باید شهر بعدی را برای سفر انتخاب کند، نزدیک‌ترین شهر بازدید نشده را انتخاب می‌کنم. فرض کنید مارین نقطه‌ی آغاز باشد.



مسافت کل: ۷۱ مایل. شاید این کوتاه‌ترین مسیر نباشد، با این حال مسیر بسیار کوتاهی است.

در ادامه توضیح کوتاهی درباره‌ی ان پی کامل آمده است: حل بعضی از مسئله‌ها حسابی سخت هستند. به عنوان مثال از مسئله‌های فروشنده‌ی دوره‌گرد و پوشش می‌توان نام برد. بر اساس نظر بسیاری از افراد باهوش و کاردرست الگوریتمی برای حل سریع این مسئله‌ها نمی‌توان نوشت.

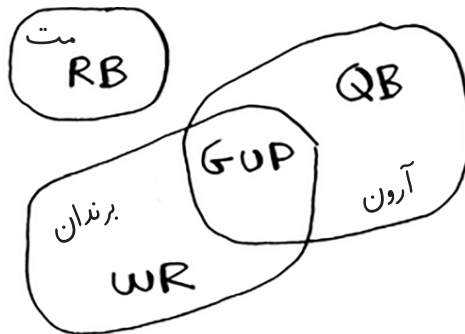
## تشخیص مسئله‌ی ان پی کامل



یونا در حال انتخاب بازیکن برای تیم فوتبال خیالی خود است. او توانایی‌هایی را که برای بازیکن‌ها در نظر دارد، لیست کرده است: مهاجم نوک حمله‌ی کاردرست، مهاجم توپ‌بر خوب، با توانایی بازی خوب در هوای بارانی، زیر فشار و نظایر آن. او لیستی از بازیکنان دارد که در آن توانایی هر بازیکن قید شده است.

بازیکن	ویژگی
مت فرت	RB
برندان مارشال	WR / تحت فشار عملکرد خوبی دارد.
آرون راجرز	QB / تحت فشار عملکرد خوبی دارد.
...	...

چوناً به تیمی نیاز دارد که تمام توانایی‌های مورد نظر او را برآورده کند. همچنین تعداد نفرات تیم محدود است. یونا متوجه ماجرا می‌شود: «یک لحظه صبر کنید. این یک مسئله‌ی پوشش مجموعه است!»

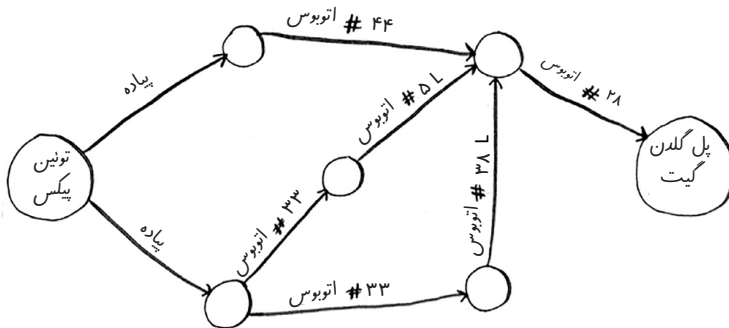


چوناً می‌تواند از همان الگوریتم تقریبی برای ایجاد تیم خود استفاده کند:

۱. بازیکنی را انتخاب کنید که از میان توانایی‌های مورد نیاز بتواند بیشترین ویژگی‌ها را شامل بشود.

۲. این کار را تا زمانی تکرار کنید که بازیکنان انتخابی تیم تمام توانایی‌های مورد نظر را داشته باشند (یا فضای تیم شما تمام شود).

مسئله‌های ان پی کامل همه جا حضور پیدا می‌کنند! خوب است که تشخیص بدهید مسئله‌ای که مشغول حل آن هستید ان پی کامل است یا خیر. در این مرحله، می‌توانید از تلاش برای حل کامل آن دست کشیده و به جای آن با استفاده از یک الگوریتم تقریبی به جواب برسید. اما تشخیص اینکه آیا مسئله‌ای که روی آن کار می‌کنید ان پی کامل است یا خیر، دشوار است. معمولاً بین مسئله‌ای که به راحتی قابل حل است و مسئله‌ی ان پی کامل تفاوت بسیار کمی وجود دارد. به عنوان مثال، در فصل‌های قبل در مورد کوتاه‌ترین مسیرها بسیار صحبت کردم. شما می‌دانید که چگونه باید کوتاه‌ترین راه را برای رسیدن از نقطه‌ی A به نقطه‌ی B محاسبه کنید.



اما اگر بخواهید کوتاه‌ترین مسیری را بیابید که چندین نقطه را به هم متصل کند، مثل مسئله‌ی فروشنده‌ی دوره‌گرد و ان پی کامل خواهد بود. پاسخ کوتاه: هیچ راه آسانی برای تشخیص اینکه آیا مسئله‌ای که روی آن کار می‌کنید ان پی کامل است وجود ندارد. چند نکته را با هم مرور کنیم:

- الگوریتم شما اگر تعداد آیتم‌ها انگشت شمار باشد به سرعت اجرا می‌شود، اما با آیتم‌های بیشتر اساسی کند می‌شود.

- «همه‌ی ترکیبات X» معمولاً به یک مسئله‌ی ان پی کامل اشاره می‌کنند.

- آیا باید «هر نسخه‌ی ممکن»  $X$  را محاسبه کنید چون نمی‌توانید آن را به مسئله‌های فرعی کوچک‌تر تقسیم کنید؟ ان پی کامل را امتحان کنید.
- اگر مسئله‌ی شما شامل یک توالی باشد (مانند دنباله‌ای از شهرها، مانند فروشنده‌ی دوره‌گرد)، و حل آن دشوار باشد، ان پی کامل را امتحان کنید.
- اگر مسئله‌ی شما شامل یک مجموعه (مانند مجموعه‌ای از ایستگاه‌های رادیویی) است و حل آن سخت است، ان پی کامل را امتحان کنید.
- می‌توانید مسئله‌ی خود را به عنوان مسئله‌ی پوشش مجموعه یا مسئله‌ی فروشنده‌ی دوره‌گرد بیان کنید؟ پس مسئله‌ی شما قطعاً ان پی کامل است.

## تمرین

- ۸.۶** یک پستچی باید به ۲۰ خانه بسته تحویل بدهد. او باید کوتاه‌ترین مسیری را که به تمام ۲۰ خانه می‌رود، پیدا کند. آیا این یک مسئله‌ی ان پی کامل است؟
- ۸.۷** پیدا کردن بزرگ‌ترین دسته در یک مجموعه از افراد (یک دسته به مجموعه افرادی گفته می‌شود که همه‌ی افراد آن مجموعه همدیگر را می‌شناسند). آیا این یک مسئله‌ی ان پی کامل است؟
- ۸.۸** شما مشغول رسم کردن نقشه‌ی ایالات متحده هستید و باید ایالت‌های مجاور را با رنگ‌های مختلف رنگ‌آمیزی کنید. باید حداقل تعداد رنگ مورد نیاز خود را پیدا کنید تا هیچ دو ایالت مجاور هم‌رنگ نباشند. آیا این یک مسئله‌ی ان پی کامل است؟

## جمع بندی

- الگوریتم‌های حریمانه به صورت محلی بهینه‌سازی می‌شوند، به این امید که در نهایت به یک بهینه‌ی سراسری برسند.
- مسئله‌های ان‌پی کامل راه حل سریع شناخته شده‌ای ندارند.
- اگر مسئله‌ای از ان‌پی کامل دارید، بهترین گزینه استفاده از یک الگوریتم تقریبی است.
- نوشتن الگوریتم‌های حریمانه آسان و سرعت اجرای آن زیاد است، بنابراین برای الگوریتم‌های تقریبی مناسب هستند.

## ۹ | برنامه‌نویسی پویا



در این فصل

• درباره‌ی برنامه‌نویسی پویا یاد می‌گیرید. برنامه‌نویسی پویا تکنیکی برای حل یک مسئله‌ی دشوار با تقسیم کردن آن به زیرمسئله<sup>۱</sup> و شروع کار با حل آن زیرمسئله‌ها است.

• با حل مثال، یاد می‌گیرید که یک راه حل برنامه‌نویسی پویا برای یک مسئله‌ی جدید ارائه کنید.



### مسئله‌ی کوله‌پشتی

بیابید دوباره مسئله‌ی کوله‌پشتی فصل ۸ را مرور کنیم. شما یک سارق هستید با یک کوله‌پشتی که ظرفیت ۴ پوندی دارد.

1. Dynamic programming
2. subproblem

این سه آیتم را می‌توان در کوله‌پشتی قرار داد:



ضبط صوت  
دلار ۳۰۰۰  
۴ پوند



لپ‌تاپ  
دلار ۲۰۰۰  
۳ پوند

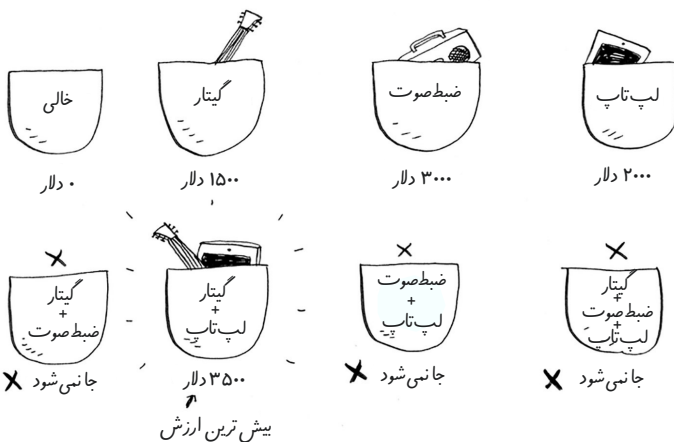


گیتار  
دلار ۱۵۰۰  
۱ پوند

کدام یک از این‌ها را باید بدزدید تا ارزش کالاهای سرقت‌شده حداکثر باشد؟

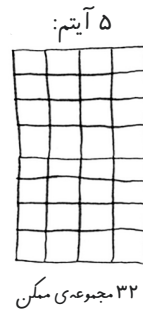
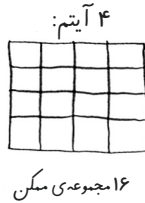
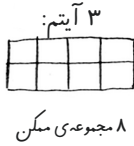
### راه حل ساده

ساده‌ترین الگوریتم این است: تمام مجموعه کالاهای ممکن را امتحان می‌کنید و مجموعه‌ای را پیدا می‌کنید که بیشترین ارزش را داشته باشد.



این روش هرچند نتیجه‌بخش است، اما بسیار کند است. برای ۳ آیتم باید ۸ مجموعه‌ی ممکن را حساب کنید. برای ۴ آیتم باید ۱۶ مجموعه حساب کنید. با هر آیتمی که اضافه می‌کنید، تعداد مجموعه‌هایی که باید حساب کنید دو برابر می‌شود! این الگوریتم به زمان  $O(2^n)$  نیاز دارد که خیلی خیلی کند است.





۳۲ آیتم =  
۴ میلیارد  
مجموعه‌ی ممکن !!

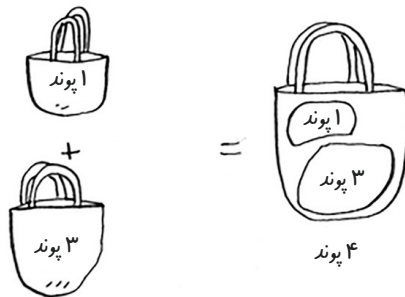
این روش برای تعداد متوسطی از کالا غیر عملی است. در فصل ۸ نحوه‌ی محاسبه‌ی یک جواب تقریبی را مشاهده کردید. آن راه حل اگر چه نزدیک به راه حل بهینه است، اما لزوماً می‌تواند راه حل بهینه نباشد.

پس چگونه جواب بهینه را محاسبه می‌کنید؟

### برنامه‌نویسی پویا

پاسخ: با برنامه نویسی پویا! بیایید ببینیم در اینجا الگوریتم برنامه‌نویسی پویا به چه شکل کار می‌کند. در برنامه‌نویسی پویا با حل مسئله‌های فرعی شروع کرده و آن را به حل مسئله‌ی بزرگ بسط می‌دهید.

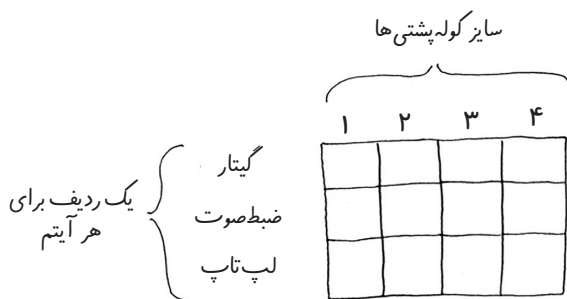
برای مسئله‌ی کوله پشتی، با حل مسئله‌ی کوله‌پشتی‌های کوچک‌تر (یا «کوله‌پشتی‌های فرعی») شروع می‌کنید و سپس تا حل مسئله‌ی اصلی پیش می‌روید.



برنامه‌نویسی پویا مفهوم سختی است، بنابراین نگران نشوید اگر فوراً آن را متوجه نشدید. مثال‌های زیادی را با هم بررسی می‌کنیم.

ابتدا با نشان دادن این الگوریتم و به شکل عملی شروع می‌کنم. بعد از آن که یک بار آن را در عمل دیدید، سؤال‌های زیادی در ذهن شما شکل می‌گیرد! تمام تلاشم را می‌کنم تا به تمامی این موارد پردازم.

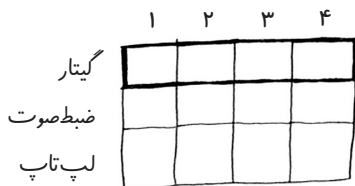
هر الگوریتم برنامه‌نویسی پویا با یک شبکه‌ی شروع می‌شود. در ادامه شبکه‌ای برای مسئله‌ی کوله‌پشتی آمده است.



ردیف‌های شبکه ایتِم‌ها و ستون‌ها وزن‌های کوله‌پشتی از ۱ پوند تا ۴ پوند هستند. شما به تمام آن ستون‌ها نیاز دارید زیرا به شما در محاسبه‌ی مقادیر کوله‌های فرعی کمک می‌کنند. شبکه در ابتدا تهی است. شما باید تمام سلول‌های شبکه را پر کنید. با پر شدن شبکه، به پاسخ این مسئله دست می‌یابید! لطفاً به همین شکل پیش بروید. شبکه‌ی خود را بسازید، تا با هم آن را پر کنیم.

### ردیف گیتار

فرمول دقیق محاسبه‌ی این شبکه را بعدتر به شما می‌گویم. ابتدا بیایید نگاهی به آن بیندازیم. از ردیف اول شروع کنید.



ردیف اول به گیتار اختصاص دارد، به این معنی که شما سعی می‌کنید گیتار را در کوله‌پشتی جای بدهید. در هر سلول، یک تصمیم‌گیری ساده وجود دارد: گیتار را بدزدید یا نه؟ به یاد داشته باشید، شما در تلاش برای یافتن مجموعه‌ای از اقلام هستید که بیشترین ارزش را از سرقت نصیب شما بکند.

سلول اول دارای یک کوله‌پشتی با ظرفیت ۱ پوند است. گیتار نیز ۱ پوند است، یعنی در کوله‌پشتی جا می‌شود! بنابراین ارزش این سلول ۱۵۰۰ دلار و حاوی یک گیتار است. بیایید شبکه را پر کنیم.

	۱	۲	۳	۴
گیتار	۱۵۰۰ دلار G			
ضبط‌صوت				
لپ‌تاپ				

مانند این، هر سلول در شبکه شامل لیستی از تمام مواردی است که در آن مقطع زمانی در کوله‌پشتی جا می‌شوند.

بیایید به سلول بعدی نگاه کنیم. در اینجا شما یک کوله‌پشتی با ظرفیت ۲ پوند دارید. خوب، گیتار قطعاً در آن جا می‌شود!

	۱	۲	۳	۴
گیتار	۱۵۰۰ دلار G	۱۵۰۰ دلار G		
ضبط‌صوت				
لپ‌تاپ				

برای بقیه‌ی سلول‌های این ردیف هم به همین شکل عمل می‌کنیم. به یاد داشته باشید، این ردیف اول است، بنابراین شما فقط گزینه‌ی گیتار را برای انتخاب دارید. این طور فرض می‌کنید که دو آیتم دیگر در حال حاضر برای سرقت در دسترس نیستند.

	۱	۲	۳	۴
گیتار	۱۵۰۰ دلار G	۱۵۰۰ دلار G	۱۵۰۰ دلار G	۱۵۰۰ دلار G
ضبط صوت				
لپ‌تاپ				

در این مرحله، احتمالاً گیج شده‌اید. چرا این کار را برای کوله‌پشتی‌هایی با ظرفیت ۱ پوند، ۲ پوند و غیره انجام می‌دهید، در حالی که مسئله از یک کوله‌پشتی ۴ پوندی صحبت می‌کند؟ به یاد دارید که چگونه به شما گفتم که برنامه‌نویسی پویا با یک مسئله‌ی کوچک شروع می‌شود و به مسئله‌ای بزرگ تبدیل می‌شود؟ شما در اینجا مسئله‌هایی فرعی را حل می‌کنید که به شما در حل مسئله‌ی بزرگ کمک می‌کند. ادامه دهید، در ادامه همه چیز واضح‌تر می‌شود.

در این مرحله، شبکه‌ی شما باید به این شکل باشد.

	۱	۲	۳	۴
گیتار	۱۵۰۰ دلار G	۱۵۰۰ دلار G	۱۵۰۰ دلار G	۱۵۰۰ دلار G
ضبط صوت				
لپ‌تاپ				

به یاد داشته باشید که هدف شما به حداکثر رساندن ارزش کوله پشتی است. این ردیف بهترین حدس فعلی برای ارزش حداکثری را نشان می‌دهد. بنابراین در حال حاضر، طبق این ردیف، اگر یک کوله پشتی با ظرفیت ۴ پوند داشته باشید، حداکثر ارزش کالا ۱۵۰۰ دلار خواهد بود.

	۱	۲	۳	۴
گیتار	۱۵۰۰ دلار G	۱۵۰۰ دلار G	۱۵۰۰ دلار G	۱۵۰۰ دلار G
ضبط صوت				
لپ‌تاپ				

← بهترین گزینه‌ی فعلی برای سرقت: گیتار ۱۵۰۰ دلاری

می‌دانید که این راه حل نهایی نیست. همانطور که الگوریتم را مرور می‌کنیم، تخمین خود را بهبود می‌بخشید.

### ردیف ضبط صوت

بیا یاد برویم سراغ ردیف بعدی. این ردیف برای ضبط صوت است. اکنون که در ردیف دوم هستید، می‌توانید ضبط صوت یا گیتار را سرقت کنید. در هر ردیف، می‌توانید آیتم آن ردیف یا آیتم‌های ردیف‌های بالاتر را بدزدید. بنابراین در حال حاضر امکان سرقت لپ‌تاپ وجود ندارد، اما می‌توانید ضبط صوت و/یا گیتار را بدزدید. بیا یاد با سلول اول شروع کنیم، یک کوله پشتی با ظرفیت ۱ پوند. حداکثر مقداری که در این مقطع می‌توانید در یک کوله پشتی ۱ پوندی قرار بدهید، ۱۵۰۰ دلار است.

ماکزیمم فعلی برای کوله پشتی یک پوندی

	۱	۲	۳	۴
گیتار	۱۵۰۰ دلار G	۱۵۰۰ دلار G	۱۵۰۰ دلار G	۱۵۰۰ دلار G
ضبط صوت				
لپ‌تاپ				

ماکزیمم جدید برای یک کوله پشتی ۱ پوندی

آیا باید این ضبط صوت را بدزدید یا نه؟

شما یک کوله پشتی با ظرفیت ۱ پوند دارید. آیا ضبط صوت در آن جا می شود؟ نه، خیلی بزرگ است! از آنجایی که ضبط صوت در کوله پشتی جا نمی شود، ۱۵۰۰ دلار حداکثر مقدار برای یک کوله پشتی ۱ پوندی است.

	۱	۲	۳	۴
گیتار	۱۵۰۰ دلار G	۱۵۰۰ دلار G	۱۵۰۰ دلار G	۱۵۰۰ دلار G
ضبط صوت	↓ ۱۵۰۰ دلار G			
لپ تاپ				

برای دو سلول بعدی هم به همین شکل عمل می کنیم. این کوله پشتی ها دارای ظرفیت ۲ پوند و ۳ پوند هستند. حداکثر مقدار قبلی برای هر دو ۱۵۰۰ دلار بود.

	۱	۲	۳	۴
گیتار	۱۵۰۰ دلار G	۱۵۰۰ دلار G	۱۵۰۰ دلار G	۱۵۰۰ دلار G
ضبط صوت	↓ ۱۵۰۰ دلار G	↓ ۱۵۰۰ دلار G	↓ ۱۵۰۰ دلار G	
لپ تاپ				

همچنان ضبط صوت جا نمی شود، بنابراین فرض شما بدون تغییر باقی می ماند. اگر یک کوله پشتی با ظرفیت ۴ پوند داشته باشید چه؟ خب: بالاخره ضبط صوت جا می شود! حداکثر مقدار قبلی ۱۵۰۰ دلار بود، اما اگر در عوض ضبط صوت را در آن قرار بدهید، ارزش آن ۳۰۰۰ دلار می شود! بیایید ضبط صوت را انتخاب کنیم.

	۱	۲	۳	۴
گیتار	\$1500 G	\$1500 G	\$1500 G	\$1500 G
ضبط صوت	↓ \$1500 G	↓ \$1500 G	↓ \$1500 G	↓ \$3000 G
لپ تاپ				

برآورد خود را به روز کردید! اگر یک کوله پشتی ۴ پوندی دارید، می توانید حداقل ۳۰۰۰ دلار کالا در آن قرار بدهید. با نگاه به شبکه متوجه می شوید که در حال به روزرسانی تدریجی تخمین خود هستید.

	۱	۲	۳	۴	
گیتار	۱۵۰۰ دلار G	۱۵۰۰ دلار G	۱۵۰۰ دلار G	۱۵۰۰ دلار G	← برآورد قبلی
ضبط صوت	↓ ۱۵۰۰ دلار G	↓ ۱۵۰۰ دلار G	↓ ۱۵۰۰ دلار G	↓ ۳۰۰۰ دلار G	← برآورد جدید
لپ تاپ					← راه حل نهایی

### ردیف لپ تاپ

بیاید همین کار را با لپ تاپ انجام بدهیم! وزن این لپ تاپ ۳ پوند است، بنابراین در یک کوله پشتی ۱ پوندی یا ۲ پوندی جای نمی گیرد. تخمین دو سلول اول ۱۵۰۰ دلار است.

	۱	۲	۳	۴
گیتار	۱۵۰۰ دلار G	۱۵۰۰ دلار G	۱۵۰۰ دلار G	۱۵۰۰ دلار G
ضبط صوت	↓ ۱۵۰۰ دلار G	↓ ۱۵۰۰ دلار G	↓ ۱۵۰۰ دلار G	↓ ۳۰۰۰ دلار G
لپ تاپ	↓ ۱۵۰۰ دلار G	↓ ۱۵۰۰ دلار G		

برای ۳ پوند، تخمین قدیمی ۱۵۰۰ دلار بود. اما می‌توانید به جای آن لپ‌تاپ را انتخاب کنید، و ارزش آن ۲۰۰۰ دلار است. بنابراین حداکثر برآورد جدید ۲۰۰۰ دلار است!

	۱	۲	۳	۴
گیتار	۱۵۰۰ دلار G	۱۵۰۰ دلار G	۱۵۰۰ دلار G	۱۵۰۰ دلار G
ضبط صوت	↓ ۱۵۰۰ دلار G	↓ ۱۵۰۰ دلار G	↓ ۱۵۰۰ دلار G	۳۰۰۰ دلار S
لپ‌تاپ	↓ ۱۵۰۰ دلار G	↓ ۱۵۰۰ دلار G	۲۰۰۰ دلار L	

برای ۴ پوند، اوضاع جالب می‌شود. این بخش مهم است. برآورد فعلی ۳۰۰۰ دلار است. شما می‌توانید لپ‌تاپ را در کوله‌پشتی قرار بدهید، اما ارزش آن فقط ۲۰۰۰ دلار می‌شود.

۲۰۰۰ دلار      یا      ۳۰۰۰ دلار  
لپ‌تاپ                      ضبط صوت

خب، نتیجه به خوبی تخمین قدیمی نیست. اما صبر کنید! وزن لپ‌تاپ فقط ۳ پوند است، بنابراین همچنان ۱ پوند فضای خالی دارید! می‌توانید چیزی در این ۱ پوند قرار بدهید.

۳۰۰۰ دلار      یا       $\left( \begin{array}{l} ۲۰۰۰ \text{ دلار} \\ \text{لپ‌تاپ} \end{array} + \frac{\text{???}}{\text{یک پوند از فضای خالی}} \right)$

حداکثر مقداری که می‌توانید در ۱ پوند فضا قرار بدهید چقدر است؟ خب، در طی این مدت همواره آن را محاسبه کرده‌اید.



	۲	۳	۴
	۱۵۰۰ دلار G	۱۵۰۰ دلار G	۱۵۰۰ دلار G
↓	۱۵۰۰ دلار G	۱۵۰۰ دلار G	۳۰۰۰ دلار S
↓	۱۵۰۰ دلار G	۱۵۰۰ دلار G	۲۰۰۰ دلار L

ماکزیم ارزش  
برای ۱ پوند →

طبق بهترین تخمین آخر، می‌توانید گیتار ۱۵۰۰ دلاری را در فضای ۱ پوندی قرار بدهید. بنابراین مقایسه‌ی واقعی به شرح زیر است.

$$۳۰۰۰ \text{ دلار} \quad \text{یا} \quad \left( \begin{array}{l} ۱۵۰۰ \text{ دلار} \\ + \\ ۲۰۰۰ \text{ دلار} \end{array} \right) \begin{array}{l} \text{گیتار} \\ \text{لپ‌تاپ} \end{array}$$

ضبط صوت

شاید با خودتان فکر می‌کردید که چرا حداکثر مقدار را برای کوله پشتی‌های کوچک‌تر محاسبه می‌کنید. حالا دیگر امیدوارم قابل فهم باشد! هنگامی که فضای خالی دارید، می‌توانید از پاسخ به آن مسئله‌های فرعی استفاده کنید تا بفهمید چه چیزی در آن فضا قرار می‌گیرد. بهتر است لپ‌تاپ + گیتار را با ارزش ۳۵۰۰ دلار انتخاب کنید.

شبکه‌ی نهایی به این صورت است.

	۱	۲	۳	۴
گیتار	۱۵۰۰ دلار G	۱۵۰۰ دلار G	۱۵۰۰ دلار G	۱۵۰۰ دلار G
↓	۱۵۰۰ دلار G	۱۵۰۰ دلار G	۱۵۰۰ دلار G	۳۰۰۰ دلار S
↓	۱۵۰۰ دلار G	۱۵۰۰ دلار G	۲۰۰۰ دلار L	۳۵۰۰ دلار L

لپ‌تاپ

پاسخ!

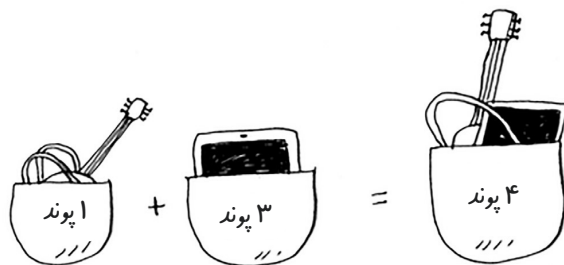
پاسخ: حداکثر ارزشی که در کوله‌پشتی قرار می‌گیرد ۳۵۰۰ دلار است شامل یک گیتار و یک لپ‌تاپ!

شاید فکر کنید که از فرمول دیگری برای محاسبه‌ی مقدار آخرین سلول استفاده کردم. به این دلیل که هنگام پرکردن مقادیر سلول‌های قبلی از پیچیدگی‌های غیرضروری صرف نظر کردم. مقدار هر سلول با فرمولی یکسان محاسبه می‌شود:

$$\text{ستون ردیف} \downarrow \downarrow \text{CELL}[i][j] = \text{ماکزیم} \left\{ \begin{array}{l} \text{۱. ماکزیم قبلی (ارزش در CELL}[i-1][j]) \\ \text{یا} \\ \text{۲. ارزش آیتم فعلی} \end{array} \right. + \text{ارزش فضای باقی‌مانده} \uparrow$$

[j-1][i] - وزن آیتم - CELL

شما می‌توانید از این فرمول برای هر سلول این شبکه استفاده کنید، و باید در نهایت شبکه‌ی شما برابر باشد با همان شبکه‌ای که من به آن رسیدم. به یاد دارید که چگونه در مورد حل زیر مسئله‌ها صحبت کردم؟ شما راه‌حل‌های دو زیرمسئله را برای حل مشکل بزرگ‌تر ترکیب کردید.



## پرسش‌های متداول در مسئله‌ی کوله‌پشتی

چون ممکن است همچنان ابهاماتی در مسئله‌ی کوله‌پشتی باقی مانده باشد، در این بخش به برخی از سؤالات رایج پاسخ داده می‌شود.



آیفون  
۳۰۰۰ دلار  
۱ پوند

### با اضافه‌کردن یک آیتم دیگر چه اتفاقی می‌افتد؟

فرض کنید متوجه آیتم قابل سرقت چهارمی بشوید که قبلاً به آن توجه نکردید! مثلاً می‌توانید یک آیفون هم بدزدید.

آیا برای حساب این آیتم جدید باید همه چیز را از اول حساب کنید؟ جواب منفی است. به یاد داشته باشید، برنامه‌نویسی پویا به تدریج و مستمر بر اساس برآورد شما ساخته می‌شود. تا اینجا، این آیتم‌ها حداکثر ارزش را دارند:

	۱	۲	۳	۴
گیتار	۱۵۰۰ دلار G	۱۵۰۰ دلار G	۱۵۰۰ دلار G	۱۵۰۰ دلار G
ضبط صوت	۱۵۰۰ دلار G	۱۵۰۰ دلار G	۱۵۰۰ دلار G	۳۰۰۰ دلار S
لپ‌تاپ	۱۵۰۰ دلار G	۱۵۰۰ دلار G	۲۰۰۰ دلار L	۳۵۰۰ دلار L G

یعنی برای یک کوله‌پشتی ۴ پوندی، می‌توانید کالاهایی به ارزش ۳۵۰۰ دلار را سرقت کنید. فرض بر این بود که این بیشترین ارزش نهایی است. حالا بیایید یک ردیف برای آیفون اضافه کنید.

	۱	۲	۳	۴
گیتار	۱۵۰۰ دلار G	۱۵۰۰ دلار G	۱۵۰۰ دلار G	۱۵۰۰ دلار G
ضبط صوت	۱۵۰۰ دلار G	۱۵۰۰ دلار G	۱۵۰۰ دلار G	۳۰۰۰ دلار S
لپ‌تاپ	۱۵۰۰ دلار G	۱۵۰۰ دلار G	۲۰۰۰ دلار L	۳۵۰۰ دلار L G
آیفون				

← پاسخ جدید

شما یک حداکثر ارزش جدید دارید! سعی کنید سلول‌های این ردیف جدید را پیش از خواندن باقی کتاب پر کنید.

بیا بید با سلول اول شروع کنیم. آیفون در یک کوله پشتی ۱ پوندی قرار می‌گیرد. پیش از این حداکثر قیمت ۱۵۰۰ دلار بود، اما آیفون ۲۰۰۰ دلار ارزش دارد. در عوض آیفون را انتخاب می‌کنیم.

	۱	۲	۳	۴
گیتار	۱۵۰۰ دلار G	۱۵۰۰ دلار G	۱۵۰۰ دلار G	۱۵۰۰ دلار G
ضبط صوت	۱۵۰۰ دلار G	۱۵۰۰ دلار G	۱۵۰۰ دلار G	۳۰۰۰ دلار S
لپ‌تاپ	۱۵۰۰ دلار G	۱۵۰۰ دلار G	۲۰۰۰ دلار L	۳۵۰۰ دلار LG
آیفون	۲۰۰۰ دلار I			

در سلول بعدی می‌توانید آیفون و گیتار را جای بدهید.

	۱	۲	۳	۴
گیتار	۱۵۰۰ دلار G	۱۵۰۰ دلار G	۱۵۰۰ دلار G	۱۵۰۰ دلار G
ضبط صوت	۱۵۰۰ دلار G	۱۵۰۰ دلار G	۱۵۰۰ دلار G	۳۰۰۰ دلار S
لپ‌تاپ	۱۵۰۰ دلار G	۱۵۰۰ دلار G	۲۰۰۰ دلار L	۳۵۰۰ دلار LG
آیفون	۲۰۰۰ دلار I	۳۵۰۰ دلار IG		

برای سلول ۳، گزینه‌ی بهتری از انتخاب دوباره‌ی آیفون و گیتار ندارید، پس در این مرحله کاری نمی‌کنید.

برای سلول آخر، وضعیت جالب می‌شود. حداکثر مقدار فعلی ۳۵۰۰ دلار است. به جای آن می‌توانید آیفون را بدزدید و ۳ پوند فضای باقیمانده در اختیار داشته باشید.

$$3500 \text{ دلار لپ‌تاپ + گیتار} \quad \text{vs} \quad \left( 2000 \text{ دلار آیفون} + \frac{???}{3 \text{ پوند فضای خالی}} \right)$$

آن ۳ پوند ۲۰۰۰ دلار ارزش دارد! ۲۰۰۰ دلار از آیفون + ۲۰۰۰ دلار از زیرمسئله‌ی قبلی: ۴۰۰۰ دلار می‌شود. یک حداکثر جدید!

نسخه‌ی جدید شبکه‌ی نهایی به شکل زیر است:

۱۵۰۰ دلار G	۱۵۰۰ دلار G	۱۵۰۰ دلار G	۱۵۰۰ دلار G
۱۵۰۰ دلار G	۱۵۰۰ دلار G	۱۵۰۰ دلار G	۳۰۰۰ دلار S
۱۵۰۰ دلار G	۱۵۰۰ دلار G	۲۰۰۰ دلار L	۳۵۰۰ دلار LG
۳۵۰۰ دلار I	۳۵۰۰ دلار IG	۳۵۰۰ دلار IG	۴۰۰۰ دلار IL

↑  
پاسخ جدید

پرسش: هیچ‌گاه ارزش یک ستون پایین هم می‌آید؟

ماکزیمم ارزش  
نزولی است.

۱۵۰۰ دلار	۱۵۰۰ دلار	۱۵۰۰ دلار	۱۵۰۰ دلار
∅	∅	∅	۳۰۰۰ دلار

لحظه‌ای تأمل کنید و به پاسخ این پرسش فکر کنید.

پاسخ: خیر. در هر تکرار، حداکثر برآورد فعلی را ذخیره می‌کنید. هرگز تخمین بدتر از قبل نمی‌شود!

### تمرین

۹.۱ فرض کنید می‌توانید یک آیتم دیگر را سرقت کنید: پخش‌کننده‌ی MP۳. وزن آن ۱ پوند است و ۱۰۰۰ دلار ارزش دارد. آیا باید آن را بدزدید؟

### چه اتفاقی می‌افتد اگر ترتیب ردیف‌ها را تغییر بدهید؟

آیا تغییری در پاسخ ایجاد می‌شود؟ فرض کنید ردیف‌ها را به این ترتیب پر می‌کنید: ضبط صوت، لپ‌تاپ، گیتار. شبکه‌ی آن به چه شکل است؟ قبل از خواندن ادامه‌ی کتاب، شبکه را برای خود پر کنید.  
شبکه به این شکل است:

	۱	۲	۳	۴
ضبط صوت	∅	∅	∅	۳۰۰۰ دلار S
لپ‌تاپ	∅	∅	۲۰۰۰ دلار L	↓ ۳۰۰۰ دلار S
گیتار	۱۵۰۰ دلار G	۱۵۰۰ دلار G	↓ ۲۰۰۰ دلار L	↖ ۳۵۰۰ دلار LG

پاسخ تغییر نمی‌کند. ترتیب ردیف‌ها اهمیتی ندارد.

### آیا می‌توان شبکه را به جای ردیفی، ستونی پر کرد؟

خودتان آن را امتحان کنید! در این مسئله فرقی نمی‌کند. اما می‌تواند در مسئله‌های دیگر تفاوت ایجاد کند.

## چه اتفاقی می‌افتد اگر یک آیتم کوچک‌تر اضافه کنید؟

فرض کنید بشود یک گردن بند دزدید. وزن آن ۰.۵ پوند است و ارزش آن ۱۰۰۰ دلار است. تا اینجا، شبکه‌ی شما فرض می‌کند که همه‌ی وزن‌ها اعداد صحیح هستند. حالا تصمیم می‌گیرید گردن‌بند را بدزدید. ۳.۵ پوند برای شما باقی مانده است. حداکثر مقداری که می‌توانید در ۳.۵ پوند قرار بدهید چقدر است؟ نمی‌دانید! شما فقط مقادیر را برای کوله‌های ۱ پوندی، ۲ پوندی، ۳ پوندی و ۴ پوندی محاسبه کرده‌اید. شما باید ارزش یک کوله پستی ۳.۵ پوندی را بدانید.

برای گردن‌بند، شما باید دانه‌بندی ریزتری را در نظر بگیرید، بنابراین شبکه باید تغییر کند.

	۰.۵	۱	۱.۵	۲	۲.۵	۳	۳.۵	۴
گیتار								
ضبط صوت								
لپ‌تاپ								
جواهرات								

## آیا می‌توان کسری از یک کالا را سرقت کرد؟

فرض کنید شما یک سارق هستید و در یک فروشگاه مواد غذایی ایستاده‌اید. می‌توانید کیسه‌های عدس و برنج را بدزدید. اگر یک کیسه‌ی کامل جا نشود، می‌توانید آن را باز کنید و تا جایی که می‌توانید بردارید. بنابراین دیگر همه یا هیچ نیست - می‌توانید کسری از یک آیتم را بردارید. چگونه با استفاده از برنامه‌نویسی پویا این موضوع را مدیریت می‌کنید؟

پاسخ: نمی‌توانید. در راه حل برنامه‌نویسی پویا، یا آیتم را برمی‌دارید یا خیر. انتخاب نیمی از آیتم جزو گزینه‌های برنامه‌نویسی پویا نیست.

اما این مورد نیز به راحتی با استفاده از یک الگوریتم حریصانه حل می‌شود! اول، تا جایی که می‌توانید از با ارزش‌ترین چیز بردارید. آن آیتم که تمام شد، تا آنجا که می‌توانید از با ارزش‌ترین کالای بعدی بردارید و غیره.

برای مثال، فرض کنید این آیتم‌ها را برای انتخاب دارید.



کینوآ

\$6/۱۶



دال

\$3/۱۶



برنج

\$2/۱۶

کوله‌پشتی پر از کینوآ



هر پوند کینوآ با ارزش‌تر از چیزهای دیگر است. بنابراین، تمام کینوآیی را که امکان حمل آن را دارید، بردارید! چه بهتر اگر تمام کوله‌پشتی شما با همین کینوآ پر شود.

اگر کینوآ تمام شد و کوله‌پشتی پر نشد، کالای با ارزش بعدی را بردارید و غیره.

### بهینه‌سازی برنامه‌ی سفر

فرض کنید برای تعطیلات فرح‌بخش به لندن می‌روید. دو روز فرصت دارید و کارهای زیادی برای انجام هست. اما فرصت انجام همه‌ی آن کارها را ندارید، بنابراین از آن‌ها یک لیست تهیه می‌کنید.

جاذبه	زمان	امتیاز
کلیسای وست‌مینستر	۱/۲ روز	۷
تئاتر گلوب	۱/۲ روز	۶
نگارخانه‌ی ملی	۱ روز	۹
نگارخانه‌ی ملی	۲ روز	۹
کلیسای جامع سنت پل	۱/۲ روز	۸



برای هر یک از آن‌ها، زمان مورد نیاز برای بازدید از آن محل و میزان اشتیافتان برای دیدن آن را یادداشت کنید. می‌توانید بر اساس این لیست درباره‌ی برنامه‌ی خود تصمیم بگیرید؟

همان مسئله‌ی کوله پشتی است! با این تفاوت که به جای کوله‌پشتی، زمان محدود است و عوض ضبط صوت و لپ‌تاپ، فهرست مکان‌هایی که می‌خواهید سر بزنید وجود دارد. قبل از حرکت، شبکه‌ی برنامه‌نویسی پویا را برای این لیست ترسیم کنید. شبکه به این شکل خواهد بود.

	۱/۲	۱	۱/۲	۲
کلیسای وست‌مینستر				
تئاتر گلوب				
نگارخانه‌ی ملی				
موزه‌ی بریتانیا				
کلیسای جامع سنت پل				

درست متوجه شدید؟ شبکه را پر کنید. در نهایت چه مکان‌هایی را باید ببینید؟ پاسخ آن به این شکل است:

	۱/۲	۱	۱/۲	۲
کلیسای وست‌مینستر	۷ ω	۷ ω	۷ ω	۷ ω
تئاتر گلوب	۷ ω	۱۳ ωω	۱۳ ωω	۱۳ ωω
نگارخانه‌ی ملی	۷ ω	۱۳ ωω	۱۶ ωω	۲۲ ωωω
موزه‌ی بریتانیا	۷ ω	۱۳ ωω	۱۶ ωω	۲۲ ωωω
کلیسای جامع سنت پل	۸ ω	۱۵ ωω	۲۱ ωωω	۲۴ ωωωω

↑  
پاسخ نهایی  
کلیسای وست‌مینستر  
نگارخانه‌ی ملی  
کلیسای جامع سنت پل

## بررسی آیتم‌های وابسته به یکدیگر

فرض کنید می‌خواهید به پاریس بروید، بنابراین چند آیتم را به لیست اضافه می‌کنید.

برج ایفل	۱ <sup>۱</sup> / <sub>۲</sub> روز	۸
لوور	۱ <sup>۱</sup> / <sub>۲</sub> روز	۹
نوتردام	۱ <sup>۱</sup> / <sub>۲</sub> روز	۷

برای بازدید از این مکان‌ها به زمان زیادی نیاز است، زیرا ابتدا باید از لندن به پاریس سفر کنید که نصف روز طول می‌کشد. اگر بخواهید هر سه آیتم را در برنامه‌ی خود داشته باشید، چهار روز و نیم طول می‌کشد.

صبر کنید، به این صورت نیست. لازم نیست برای هر مورد به پاریس سفر کنید. هنگامی که در پاریس هستید، برای هر کدام تنها یک روز نیاز است. پس باید یک روز برای هر آیتم + نصف روز سفر = سه و نیم روز باشد نه چهار و نیم روز. اگر برج ایفل را در کوله‌پشتی خود قرار بدهید، مورد «ارزان‌تر» موزه لوور خواهد بود - به جای یک و نیم روز فقط یک روز برای شما هزینه خواهد داشت. چگونه این را در برنامه نویسی پویا مدل‌سازی می‌کنید؟

چنین چیزی امکان‌پذیر نیست. برنامه‌نویسی پویا قدرتمند است زیرا می‌تواند زیرمسئله‌ها را حل کند و از پاسخ آن‌ها برای حل مسئله‌ی بزرگ استفاده کند. برنامه‌نویسی پویا فقط زمانی به کار می‌آید که هر زیرمسئله ناپیوسته باشد - زمانی که به سایر زیرمسئله‌ها وابسته نباشد. به این معنا که هیچ راهی برای محاسبه‌ی مسئله‌ی پاریس با استفاده از الگوریتم برنامه‌نویسی پویا وجود ندارد.

## آیا ممکن است که این راه حل نیاز به بیش از دو زیر-کوله‌پشتی داشته باشد؟

بهترین راه حل می‌تواند سرقت بیش از دو آیتم باشد. طبق این الگوریتم، حداکثر دو کوله‌پشتی را با هم ترکیب می‌کنید - هرگز بیشتر از دو کوله‌پشتی نخواهید داشت. اما این زیر-کوله‌پشتی‌ها می‌توانند زیر-کوله‌پشتی‌های خودشان را داشته باشند.



داشتن سه زیر - کوله پشتی امکان پذیر نیست.



امامی توان زیر - کوله پشتی هایی داشت که زیر - کوله پشتی های خود را داشته باشند.

## آیا ممکن است با بهترین راه حل، کوله پشتی به طور کامل پر نشود؟



الماس  
یک میلیون دلار  
۳.۵ پوند

بله. فرض کنید می توانید یک الماس را هم بدزدید.

این الماس بزرگ است: وزن آن ۵.۳ پوند است و یک میلیون دلار قیمت دارد، بسیار باارزش تر از هر چیز دیگری. حتماً باید آن را بدزدید! با سرقت آن نیم پوند فضا باقی می ماند و هیچ چیز در آن فضای باقی مانده جا نمی شود.

### تمرین

**۹.۲** فرض کنید می خواهید به پیک نیک بروید. یک کوله پشتی دارید که ۶ پوند را در خود جای می دهد و می توانید آیتم های زیر را بردارید. به هر کدام یک عدد اختصاص داده شده و هر چه مقدار آن بیشتر باشد، آیتم اهمیت بیشتری دارد:

• آب، ۳ پوند، ۱۰

• کتاب، ۱ پوند، ۳

• غذا، ۲ پوند، ۹

• ژاکت، ۲ پوند، ۵

• دوربین، ۱ پوند، ۶

مجموعه ی بهینه ای از آیتم هایی که باید در پیک نیک همراه خود داشته باشید

شامل چه چیزهایی می شود؟

## طولانی‌ترین زیررشته‌ی مشترک

تا به اینجا، با یک مسئله‌ی برنامه‌نویسی پویا مواجه شده‌اید. از آن چه نتیجه‌ای می‌توان گرفت؟



- برنامه‌نویسی پویا زمانی مفید است که بخواهید چیزی را با توجه به یک محدودیت بهینه کنید. در مسئله‌ی کوله‌پشتی، شما باید ارزش کالاهایی را که دزدیده‌اید، با توجه به اندازه‌ی کوله‌پشتی به حداکثر برسانید.

- کاربرد برنامه‌نویسی پویا در مواردی است که مسئله به زیرمسئله‌های مجزا تقسیم بشود و به یکدیگر وابسته نباشند.

دستیابی به یک راه حل از نوع برنامه‌نویسی پویا می‌تواند دشوار باشد. در این بخش بر روی این موضوع تمرکز می‌کنیم. تعدادی از نکات کلّی آن به این شرح است:

- هر راه حل برنامه‌نویسی پویا شامل یک شبکه است.
- مقدارهای موجود در سلول‌ها معمولاً همان چیزی است که می‌خواهید بهینه‌سازی کنید. برای مسئله‌ی کوله‌پشتی، مقدارها ارزش اجناس بود.
- هر سلول یک مسئله‌ی فرعی است، پس به این فکر کنید که چگونه می‌توانید مسئله‌ی خود را به چند زیرمسئله تقسیم کنید. این روش به شما کمک می‌کند تا محورها را انتخاب کنید.



بیاید به مثال دیگری نگاه کنیم. فرض کنید [dictionary.com](http://dictionary.com) را اداره می‌کنید. کاربری یک کلمه تایپ می‌کند و شما معنی آن کلمه را می‌نویسید. اما می‌خواهید که اگر کاربر کلمه را اشتباه نوشت، بتوانید

کلمه‌ی مورد نظر او را حدس بزنید. الکس در حال جست‌وجوی fish است، اما به طور تصادفی hish را تایپ کرده است. هرچند این کلمه در دیکشنری شما لیست نشده است، اما لیستی از کلمات مشابه دارید.

## مشابه "HISH"

- FISH
- VISTA

(این یک مثال بسیار ساده است، بنابراین شما لیست خود را به دو کلمه محدود می‌کنید. در واقعیت، این لیست احتمالاً هزاران کلمه خواهد بود.)  
الکس hish را تایپ کرد. منظور الکس تایپ کدام کلمه بود: fish یا vista؟

## ساخت شبکه

- شبکه برای این مسئله چگونه است؟ شما باید به این پرسش‌ها پاسخ بدهید:
- مقدار هر سلول چقدر است؟
  - چگونه این مسئله را به مسئله‌های فرعی تقسیم می‌کنید؟
  - محورهای شبکه کدام هستند؟

در برنامه‌نویسی پویا، تلاش شما در به حداکثر رساندن چیزی است. در این مورد، شما در تلاش برای یافتن طولانی‌ترین زیررشته‌ای هستید که دو کلمه‌ی مشترک دارند. hish و fish چه زیررشته‌ی مشترکی دارند؟ hish و vista چطور؟ این همان چیزی است که شما می‌خواهید آن را حساب کنید.

به یاد داشته باشید، مقدار سلول‌ها معمولاً همان چیزی است که می‌خواهید بهینه‌سازی کنید. در این مورد، مقادیر احتمالاً یک عدد خواهند بود: طول طولانی‌ترین زیررشته<sup>۱</sup> که در این دو رشته مشترک هستند.

چگونه این مسئله را به چند زیرمسئله تقسیم می‌کنید؟ می‌توانید زیر رشته‌ها را با هم مقایسه کنید. به جای مقایسه‌ی *fish* و *hish* ابتدا می‌توانید *his* و *fis* را مقایسه کنید. هر سلول دارای طول طولانی‌ترین زیررشته‌ای است که دو زیررشته‌ی مشترک دارند. به شما این سرخ را می‌دهد که احتمالاً محورها دو کلمه هستند. بنابراین احتمالاً شبکه چیزی شبیه به این باشد:

	H	I	S	H
F				
I				
S				
H				

اگر از نظر شما این یک جادو جنبل است، نگران نباشید. موضوع دشواری است - به همین دلیل است که آموزش این موضوع را در بخش‌های آخر این کتاب در نظر گرفته‌ام! بعداً یک تمرین به شما می‌دهم تا برنامه‌نویسی پویا را تمرین کنید.

### پر کردن شبکه

حالا درک درستی دارید از اینکه یک شبکه باید چگونه باشد. فرمول پر کردن هر سلول از شبکه چیست؟ می‌توانید کمی تقلب کنید، زیرا راه حل آن را می‌دانید - *fish* و *hish* یک زیررشته به طول مشترک ۳ دارند: *ish*.

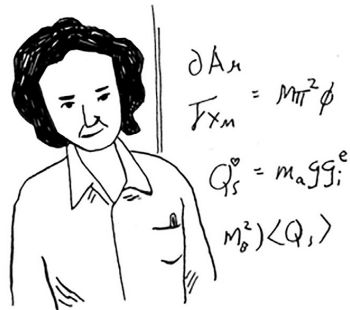
اما همچنان چیزی از فرمول قابل استفاده در اختیار شما نمی‌گذارد. دانشمندان علوم کامپیوتر گاهی درباره‌ی استفاده از الگوریتم فاینمن<sup>۱</sup> شوخی می‌کنند. نام الگوریتم فاینمن از فیزیک‌دان معروف ریچارد فاینمن گرفته شده و به این صورت است:

۱. مسئله را یادداشت کنید.

۲. به سختی فکر کنید.

۳. راه حل را یادداشت کنید.

1. Feynman algorithm



دانشمندان علوم کامپیوتر یک گوله نمک هستند!

حقیقت این است که در اینجا هیچ راه آسانی برای محاسبه‌ی فرمول وجود ندارد. شما باید آزمایش کنید و سعی کنید چیزی را پیدا کنید که کارساز باشد. گاهی اوقات الگوریتم‌ها دستورالعمل‌هایی دقیق نیستند. آن‌ها چارچوبی هستند که ایده‌ی خود را روی آن می‌سازید.

سعی کنید خودتان راه حلی برای این مسئله پیدا کنید. به شما سرنخی می‌دهم - بخشی از شبکه به این شکل است.

	H	I	S	H
F	•	•		
I				
S			۲	•
H				۳

مقادیر دیگر چیست؟ به یاد داشته باشید که هر سلول مقدار یک زیرمسئله است. چرا سلول (۳،۳) دارای مقدار ۲ است؟ چرا سلول (۳،۴) مقدار ۰ دارد؟ بعد از اینکه سعی کردید به فرمولی دست پیدا کنید، ادامه‌ی مطلب را بخوانید. حتی اگر به جواب درست هم نرسیدید، تلاش شما باعث می‌شود تا توضیح من برای شما بسیار قابل فهم‌تر باشد.

## راه حل

شبکه‌ی نهایی به این صورت است.

	H	I	S	H
F	۰	۰	۰	۰
I	۰	۱	۰	۰
S	۰	۰	۲	۰
H	۱	۰	۰	۳

فرمول مورد نظر من برای پر کردن هر سلول به این شکل است.

۱. اگر حروف با هم منطبق نباشند مقدار صفر خواهد شد.

	H	I	S	H
F	۰	۰	۰	۰
I	۰	۱	۰	۰
S	۰	۰	۲	۰
H	۱	۰	۰	۳

۲. اگر حروف منطبق باشند، ارزش آن برابر است با ارزش همسایه ۱ گوشه‌ی بالایی چپ

شبه‌کد این فرمول به این صورت است:

```

if word_a[i] == word_b[j]: ←..... حروف با هم سازگارند.
    cell[i][j] = cell[i-1][j-1] + 1
else: ←..... حروف با هم سازگار نیستند.
    cell[i][j] = 0
    
```



در این جدول *hish* در برابر *vista* درج شده است.

	V	I	S	T	A
H	.	.	.	.	.
I	.	۱	.	.	.
S	.	.	۲	.	.
H	.	.	.	.	.

↑ پاسخ نهایی  
 ↑ پاسخ نهایی نیست

توجه داشته باشید که در این مسئله، راه حل نهایی ممکن است در آخرین سلول نباشد! برای مسئله‌ی کوله پشتی، همیشه راه حل نهایی در سلول آخر بود. اما برای طولانی‌ترین زیررشته‌ی مشترک، راه حل بزرگترین عدد در شبکه است - و ممکن است آخرین سلول نباشد.

بیایید به سؤال اصلی برگردیم: کدام رشته با *hish* اشتراک بیشتری دارد؟ *fish* و *hish* زیررشته‌ای از سه حرف مشترک و *hish* و *vista* زیر رشته‌ای از دو حرف مشترک دارند. احتمالاً منظور الگس *fish* بوده است.

### بزرگ‌ترین زیردنباله‌ی مشترک<sup>۱</sup>

فرض کنید الگس اشتباهی *fosh* را جست‌وجو کرده است. منظورش کدام کلمه است: *fish* یا *fort*؟

	F	O	S	H
F	۱	.	.	.
O	.	۲	.	.
R	.	.	.	.
T	.	.	.	.

vs

	F	O	S	H
F	۱	.	.	.
I	.	.	.	.
S	.	.	۱	.
H	.	.	.	۲

بیایید آن‌ها را با استفاده از فرمول بزرگترین زیردنباله‌ی مشترک مقایسه کنیم. هر دو یکسان هستند: دو حرف!

1. Longest common subsequence

اما fosh به fish نزدیک تر است.

$$\begin{array}{cccc} F & O & S & H \\ \downarrow & & \downarrow & \downarrow \\ F & I & S & H \end{array} = ۳$$

$$\begin{array}{cccc} F & O & S & H \\ \downarrow & \downarrow & & \\ F & O & R & T \end{array} = ۲$$

شما در حال مقایسه‌ی بزرگ‌ترین زیررشته‌ی مشترک هستید، اما در واقع باید بزرگ‌ترین زیردنباله‌ی مشترک را مقایسه کنید: تعداد حروف در یک دنباله که در این دو کلمه مشترک است. چگونه بزرگترین زیردنباله‌ی مشترک را محاسبه می‌کنید؟

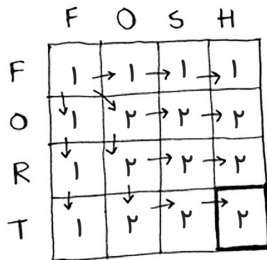
در اینجا بخشی از شبکه‌ی fish و fosh آمده است.

	F	O	S	H
F	1	1		
I	1			
S		1	۲	۲
H				

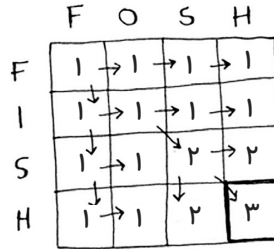
آیا می‌توانید فرمول این شبکه را پیدا کنید؟ طولانی‌ترین زیردنباله‌ی مشترک بسیار شبیه به طولانی‌ترین زیررشته‌ی مشترک است و فرمول آن‌ها نیز تقریباً مشابه هستند. پیش از آنکه به راه حل این مسئله بپردازم تلاش کنید آن را پیدا کنید.

## بزرگ‌ترین زیردنباله‌ی مشترک - راه حل

شبکه‌ی نهایی به این شکل است.



بزرگ‌ترین زیردنباله‌ی مشترک = ۲



بزرگ‌ترین زیردنباله‌ی مشترک = ۳

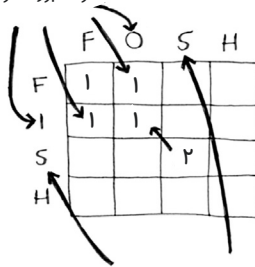
فرمول من برای پرکردن هر سلول:

از همسایگان گوشه‌ی بالایی چپ

(مقاومت با بزرگ‌ترین

زیررشته‌ی مشترک)

۱. اگر حروف با هم برابر نبودند،  
حرف بزرگ‌تر را انتخاب کنید



۲- اگر با هم منطبق باشند، مقدار آن برابر با همسایه‌ی گوشه‌ی بالایی چپ به علاوه‌ی یک خواهد بود.

و این شبه کد آن است:

حروف برابر باشند. <-----  
**if** word\_a[i] == word\_b[j]:  
 cell[i][j] = cell[i-1][j-1] + 1

حروف با هم برابر نباشند. <-----  
**else:**  
 cell[i][j] = max(cell[i-1][j], cell[i][j-1])

خب - موفق شدید! این قطعاً یکی از سخت‌ترین فصل‌های کتاب است. حالا واقعاً برنامه‌نویسی پویا کاربرد دارد؟ بله:

- زیست‌شناسان از بزرگ‌ترین زیردنباله‌ی مشترک برای یافتن تشابهات رشته‌های DNA استفاده می‌کنند. از این روش می‌توان برای تشخیص شباهت دو حیوان یا دو بیماری استفاده کرد. بزرگ‌ترین زیردنباله‌ی مشترک برای یافتن درمان بیماری ام اس استفاده می‌شود.

- آیا تا به حال از diff (مانند git diff) استفاده کرده‌اید؟ Diff تفاوت‌های میان دو فایل را به شما می‌گوید و برای انجام این کار از برنامه‌نویسی پویا استفاده می‌کند.

- در مورد تشابهات رشته‌ها صحبت کردیم. فاصله‌ی لون اشتاین<sup>۱</sup> میزان تشابهات دو رشته را اندازه‌گیری می‌کند و از برنامه‌نویسی پویا استفاده می‌کند. فاصله‌ی لون اشتاین در موارد متعددی کاربرد دارد، از بررسی درستی املاء گرفته تا فهمیدن اینکه آیا کاربران با داده‌هایی که بارگزاری می‌کنند کپی‌رایت را نقض می‌کنند یا خیر.

- آیا تا به حال از word wrap برنامه‌ای مانند ورد مایکروسافت استفاده کرده‌اید؟ چگونه می‌توان فهمید که کجا باید wrap بشود تا طول خط ثابت بماند؟ برنامه‌نویسی پویا!

## تمرین

**۹.۳** برای محاسبه‌ی بزرگ‌ترین زیردنباله‌ی مشترک، میان blue و clues شبکه را ترسیم و آن را پر کنید.

## جمع‌بندی

- برنامه‌نویسی پویا زمانی مفید است که بخواهید با توجه به یک محدودیت چیزی را بهینه کنید.
- زمانی می‌توان از برنامه‌نویسی پویا استفاده کرد که مسئله به زیرمسئله‌های ناپیوسته تقسیم شود.
- برای هر راه حل برنامه‌نویسی پویا یک شبکه وجود دارد.
- مقادیر موجود در سلول‌ها معمولاً همان چیزی هستند که می‌خواهید بهینه‌سازی کنید.
- هر سلول یک مسئله‌ی فرعی است، پس به این فکر کنید که چگونه می‌توانید مسئله‌ی خود را به مسئله‌های فرعی تقسیم کنید.
- هیچ فرمول واحدی برای محاسبه‌ی راه حل یک برنامه‌نویسی پویا وجود ندارد.



## ۱۰ | الگوریتم K - نزدیک ترین همسایه<sup>۱</sup>



در این فصل

• یاد می‌گیرید با استفاده از الگوریتم  $k$  نزدیک ترین همسایه، یک سیستم طبقه‌بندی بسازید.

• در مورد استخراج ویژگی<sup>۲</sup> یاد می‌گیرید.

• در مورد رگرسیون<sup>۳</sup> یاد می‌گیرید: پیش‌بینی یک عدد، مانند ارزش سهام در روز بعد، یا میزان رضایت کاربر از تماشای یک فیلم.

• موارد استفاده و محدودیت‌های الگوریتم K - نزدیک ترین همسایه را یاد می‌گیرید.

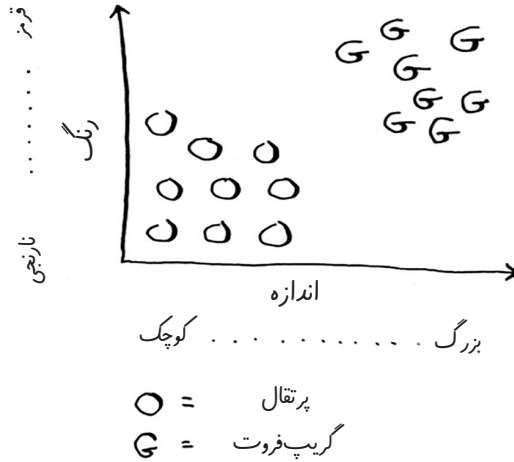
### طبقه‌بندی پرتقال در مقابل طبقه‌بندی گریپ فروت



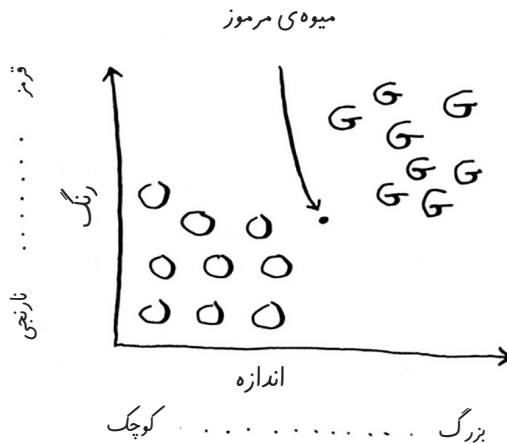
به این میوه نگاه کنید. پرتقال است یا گریپ فروت؟ خب، می‌دانیم که عموماً گریپ فروت‌ها بزرگ‌تر و قرمزتر هستند.

1. k-nearest neighbors
2. feature extraction
3. regression

روند فکر کردن من چیزی شبیه به این است: یک گراف در ذهنم دارم.

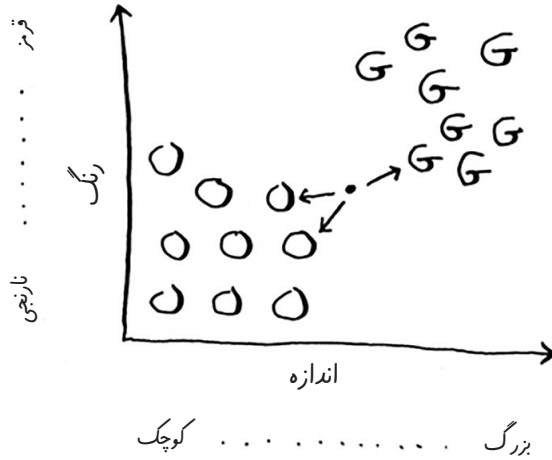


عموماً، میوه‌های بزرگ‌تر و قرمزتر گریپ فروت هستند. این میوه هم بزرگ است و هم قرمز. بنابراین احتمالاً گریپ فروت باشد. اما اگر میوه‌ای مشابه داشته باشید چه؟

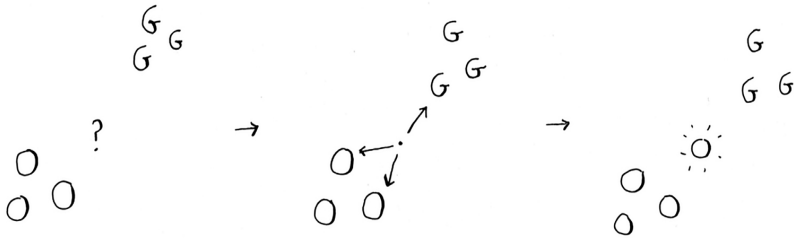


این میوه را چگونه طبقه‌بندی می‌کنید؟ یک راه این است که به همسایگان این نقطه نگاه کنید. به سه همسایه‌ی نزدیک‌تر این نقطه نگاهی بیندازید.





بیشتر همسایه‌ها پرتقال هستند. در نتیجه این میوه احتمالاً یک پرتقال است.  
 تبریک: شما الان از الگوریتم K - نزدیک‌ترین همسایه (KNN) برای طبقه‌بندی استفاده کردید! این الگوریتم بسیار ساده است.



۱. میوه‌ای جدید برای طبقه‌بندی در اختیار دارید

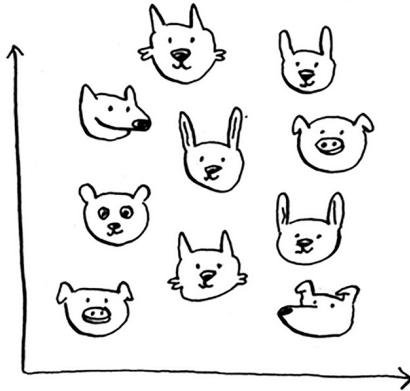
۲. به سه همسایه‌ی نزدیک آن نگاه می‌کنید

۳. پرتقال‌های بیشتر در همسایگی، پس احتمالاً پرتقال باشد.

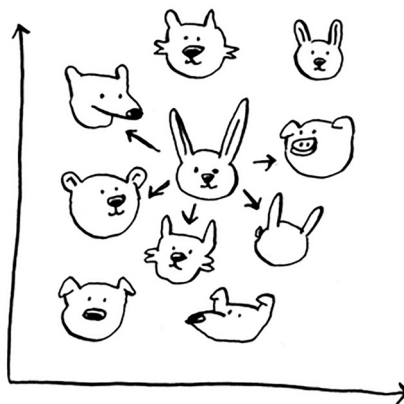
الگوریتم KNN ساده ولی مفید است! اگر چیزی را بخواهید طبقه‌بندی کنید، بد نیست ابتدا KNN را امتحان کنید. بیایید به یک مثال واقعی‌تر نگاه کنیم.

## ایجاد سیستم پیشنهاد

فرض کنید نتفلیکس هستید و می‌خواهید یک سیستم پیشنهاد فیلم برای کاربران بسازید. در ظاهر، چیزی شبیه مسئله‌ی گریپ فروت است! شما می‌توانید هر کاربر را روی یک گراف رسم کنید.



ترسیم این کاربران بر اساس تشابهات است، بنابراین کاربرهایی با سلیقه‌ی مشابه در فاصله‌ی نزدیک‌تری به هم رسم می‌شوند. فرض کنید می‌خواهید فیلم‌هایی به پریانکا پیشنهاد کنید. پنج کاربر نزدیک به او را پیدا می‌کنید.



جاستین، جی سی، جوی، لنس و کریس همگی در فیلم‌ها سلیقه‌ای مشابه دارند. بنابراین اگر آن‌ها فیلمی را دوست داشته باشند، احتمالاً پریانکا هم از آن فیلم خوشش خواهد آمد!

هنگامی که این گراف را دارید، ساختن یک سیستم پیشنهادات آسان است. اگر فیلمی هست که جاستین دوست داشته باشد، آن را به پریانکا توصیه می‌کنید.



۱. جاستین فیلم می‌بیند

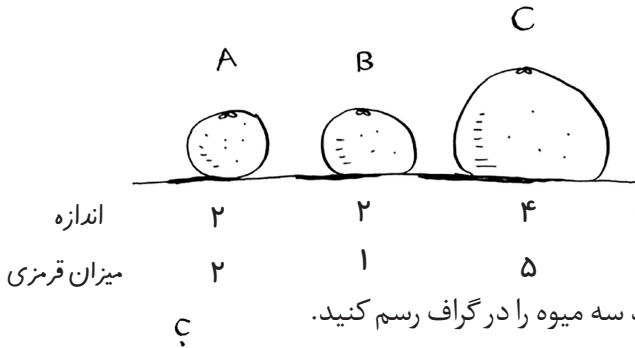
۲. از فیلم خوشش آمده

۳. فیلم را به پریانکا پیشنهاد می‌دهد

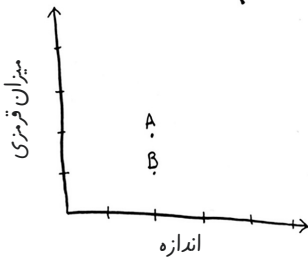
اما هنوز یک بخش مهم از این روش جا مانده است. شما کاربران را بر اساس تشابهات ترسیم کردید. چگونه متوجه می‌شوید که دو کاربر چقدر شبیه هم هستند؟

## استخراج ویژگی

در مثال گریپ‌فروت، میوه‌ها را بر اساس اندازه و میزان سرخی مقایسه کردید. اندازه و رنگ ویژگی‌هایی هستند که با هم مقایسه می‌کنید. حالا فرض کنید سه میوه دارید. می‌توانید ویژگی‌ها را استخراج کنید.



بعد می‌توانید سه میوه را در گراف رسم کنید.



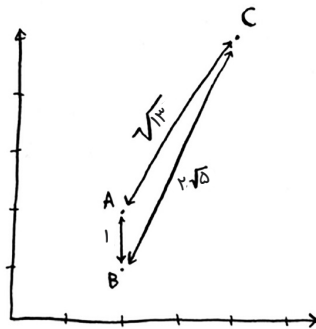
از گراف و به شکل تصویری می‌توان دریافت که میوه‌های A و B مشابه هستند. بیابید اندازه‌گیری کنیم که آن‌ها چقدر به هم نزدیک هستند. برای یافتن فاصله‌ی بین دو نقطه از فرمول فیثاغورث استفاده می‌کنیم.

$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

برای مثال، فاصله‌ی بین A و B به صورت زیر حساب می‌شود:

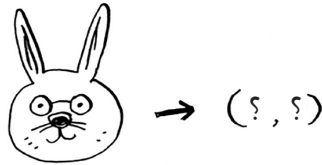
$$\begin{aligned} & \sqrt{(2-2)^2 + (2-1)^2} \\ &= \sqrt{0+1} \\ &= \sqrt{1} \\ &= 1 \end{aligned}$$

فاصله‌ی بین A و B یک است. شما می‌توانید بقیه‌ی فواصل را هم پیدا کنید.



فرمول فاصله تأییدی است بر آنچه پیش از این و به شکل تصویری دیدید: میوه‌های A و B مشابه یکدیگر هستند.

فرض کنید در عوض مقایسه را میان کاربران نتفلیکس انجام بدهید. شما به راهی برای ترسیم گراف کاربران نیاز دارید. بنابراین، باید هر کاربر را مطابق آن چه برای میوه‌ها انجام دادید به مجموعه‌ای از مختصات تبدیل کنید.



زمانی که بتوانید کاربران را در گراف در نظر بگیرید. می‌توانید فاصله‌ی میان آن‌ها را اندازه‌گیری کنید.

در اینجا شیوه‌ی تبدیل کاربران به مجموعه‌ای از اعداد آمده است. هنگامی که کاربران برای نتفلیکس ثبت نام می‌کنند، از آن‌ها خواسته می‌شود بر اساس میزان علاقه‌مندی به دسته‌بندی فیلم‌ها امتیاز بدهند. برای هر کاربر، مجموعه‌ای از رتبه‌بندی در اختیار دارید!

	پریانکا	جاستین	مرفئوس
کدی	۳	۴	۲
اکشن	۴	۳	۵
درام	۴	۵	۱
ترسناک	۱	۱	۳
عاشقانه	۴	۵	۱

پریانکا و جاستین عاشق فیلم‌های عاشقانه هستند و از فیلم‌های ترسناک متنفرند. مرفئوس فیلم اکشن دوست دارد و از فیلم‌های عاشقانه متنفر است (بیزار است از اینکه ببیند یک فیلم اکشن خوب با یک صحنه‌ی عاشقانه‌ی مسخره خراب می‌شود). به یاد

دارید در مسئله‌ی پرتقال و گریپ فروت، هر میوه با مجموعه‌ای از دو عدد نشان داده می‌شد؟ در اینجا، هر کاربر با مجموعه‌ای از پنج عدد نشان داده می‌شود.

$$\text{🍌} \rightarrow (2, 2)$$

$$\text{🐰} \rightarrow (3, 4, 4, 4, 4)$$

یک ریاضی‌دان خواهد گفت، به جای اینکه فاصله را در دو بعد محاسبه کنید، اکنون فاصله را در پنج بعد محاسبه می‌کنید. اما فرمول فاصله ثابت باقی می‌ماند.

$$\sqrt{(a_1 - a_p)^2 + (b_1 - b_p)^2 + (c_1 - c_p)^2 + (d_1 - d_p)^2 + (e_1 - e_p)^2}$$

تنها کافی است مجموعه‌ای از پنج عدد را به جای مجموعه‌ای از دو عدد در نظر بگیرید.

فرمول فاصله انعطاف‌پذیر است: شما می‌توانید مجموعه‌ای از یک میلیون عدد داشته باشید و همچنان از همان فرمول فاصله‌ی قدیمی برای پیدا کردن فاصله استفاده کنید. شاید از خود بپرسید: «با داشتن پنج عدد فاصله به چه معناست؟» فاصله به شما می‌گوید که این مجموعه اعداد چقدر شبیه هم هستند.

$$\begin{aligned} & \sqrt{(3-4)^2 + (4-3)^2 + (4-5)^2 + (1-1)^2 + (4-5)^2} \\ &= \sqrt{1+1+1+0+1} \\ &= \sqrt{4} \\ &= 2 \end{aligned}$$

این عدد، فاصله‌ی پریانکا و جاستین است.

پریانکا و جاستین خیلی شبیه هم هستند. تفاوت پریانکا و مورفیوس چیست؟ قبل از ادامه‌ی خواندن کتاب مسافت را محاسبه کنید. درست متوجه شدید؟ پریانکا و مورفیوس فاصله‌ای معادل ۲۴ دارند. فاصله به شما می‌گوید که سلیقه‌ی پریانکا بیشتر شبیه سلیقه‌ی جاستین است تا مورفیوس.

عالی! حالا پیشنهاد دادن فیلم به پریانکا آسان است: اگر جاستین فیلمی را دوست دارد، آن را به پریانکا توصیه کنید و برعکس. شما الان یک سیستم پیشنهاد فیلم ساختید!

اگر کاربر نتفلیکس هستید، نتفلیکس مدام از شما می‌خواهد: «لطفاً به فیلم‌های بیشتری امتیاز بدهید. هرچه به فیلم‌های بیشتری امتیاز بدهید، پیشنهادهایی که به شما داده می‌شود بهتر خواهد بود.» حالا دیگر می‌دانید چرا. هر چه تعداد فیلم‌های بیشتری را رتبه‌بندی کنید، نتفلیکس با دقت بیشتری می‌تواند مشاهده کند که شما به چه کاربران دیگری شبیه هستید.

## تمرین

**۱۰.۱** در مثال نتفلیکس، فاصله‌ی بین دو کاربر مختلف را با استفاده از فرمول فاصله محاسبه کردید. اما همه‌ی کاربران با یک معیار ثابت به فیلم‌ها امتیاز نمی‌دهند. فرض کنید دو کاربر با نام‌های یوگی و پینکی سلیقه‌ی یکسانی در فیلم دارند. یوگی به هر فیلمی که دوست داشته باشد ۵ امتیاز می‌دهد، در حالی که پینکی گزیده‌تر عمل می‌کند و امتیاز ۵ را فقط برای بهترین فیلم‌ها ننگه می‌دارد. آن‌ها سلیقه‌ی مشابهی دارند، اما طبق الگوریتم فاصله، همسایه نیستند. چه راهکاری برای استراتژی متفاوت آن‌ها در رتبه‌بندی در نظر می‌گیرید؟

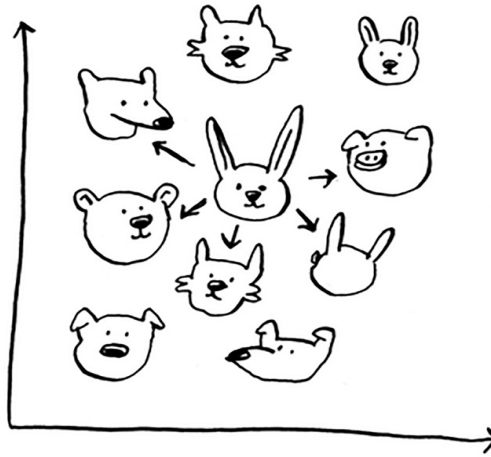
**۱۰.۲** فرض کنید نتفلیکس گروهی از «اینفلوئنسرها» را گزینش کرده است. به عنوان

مثال، کوئنتین تارانتینو و وس اندرسون اینفلوئنسرهای نتفلیکس هستند، بنابراین رتبه‌بندی این افراد بیشتر از یک کاربر عادی حساب می‌شود. چه تغییراتی برای سیستم پیشنهادات نیاز است تا نسبت به رتبه‌بندی اینفلوئنسرها جانبدارانه عمل کند؟

## رگرسیون

فرض کنید به دنبای چیزی بیش از صرفاً پیشنهاد فیلم باشید: می‌خواهید حدس بزنید که پریانکا به این فیلم چه امتیازی می‌دهد. پنج نفری که به هم نزدیک‌تر هستند را در نظر بگیرید.

راستی من مدام عدد ۵ را مثال می‌زنم. هیچ چیز خاصی در مورد عدد ۵ وجود ندارد:



شما می‌توانید ۲، یا ۱۰، یا ۱۰۰۰۰ نفر نزدیک‌تر را در نظر بگیرید. به همین دلیل است که الگوریتم  $k$ -نزدیک‌ترین همسایه نامیده می‌شود و نه پنج همسایه‌ی نزدیک‌تر! فرض کنید می‌خواهید رتبه‌بندی فیلم آوازخوان حرفه‌ای را حدس بزنید. خوب، جاستین، جی سی، جوی، لنس و کریس چه امتیازی به آن دادند؟



جاستین	:	۵
جی‌سی	:	۴
جونی	:	۴
لنس	:	۵
کریس	:	۳

می‌توانید میانگین رتبه‌بندی آن‌ها را حساب کنید و ۲.۴ ستاره بدهید. به این کار رگرسیون می‌گویند. طبقه‌بندی و رگرسیون دو کار اساسی است که با KNN انجام می‌دهید:

• طبقه‌بندی = دسته‌بندی در یک گروه

• رگرسیون = پیش‌بینی پاسخ (مانند یک عدد)

رگرسیون بسیار کاربردی است. فرض کنید یک نانواپی کوچک در برکلی دارید و هر روز نان تازه پخت می‌کنید. شما می‌خواهید تخمین بزنید که برای امروز باید چه تعداد نان درست کنید. شما مجموعه‌ای از ویژگی‌ها را در اختیار دارید:

• آب و هوا در مقیاس ۱ تا ۵ (۱ = بد، ۵ = عالی).

• آخر هفته یا تعطیلات؟ (۱ = آخر هفته یا تعطیلات است، صفر = روزهای دیگر هفته).

• آیا مسابقه‌ای برگزار می‌شود؟ (۱ = بله، صفر = خیر).



و از میزان فروش نان در مجموعه ویژگی مختلف آگاه هستید.

$$\text{A. } (5, 1, 0) = 300 \quad \text{B. } (3, 1, 1) = 225$$

قرص نان

$$\text{C. } (1, 1, 0) = 75 \quad \text{D. } (4, 0, 1) = 200$$

قرص نان

$$\text{E. } (4, 0, 0) = 150 \quad \text{F. } (2, 0, 0) = 50$$

قرص نان

امروز یک روز آخر هفته با هوای خوب است. بر اساس داده‌هایی که مشاهده کردید، چند قرص نان خواهید فروخت؟ بیا بید از KNN استفاده کنیم، در شرایطی که  $K = 4$ . ابتدا، چهار همسایه‌ی نزدیک‌تر را برای این نقطه مشخص کنید.

$$C(4, 1, 0) = ?$$

A, B, D, E نزدیکترین همسایه‌ها هستند.

- A. ۱ ←
- B. ۲ ←
- C. ۹
- D. ۲ ←
- E. ۱ ←
- F. ۵

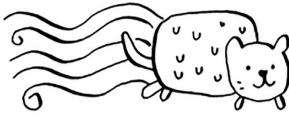
میانگین نان‌های فروخته شده در آن روزها ۲۱۸.۷۵ است. این تعداد نانی است که باید برای امروز پخت کنید!

### شباهت کسینوسی<sup>۱</sup>

تاکنون از فرمول فاصله برای مقایسه‌ی فاصله‌ی بین دو کاربر استفاده کرده‌اید. آیا این بهترین فرمول برای استفاده است؟ یکی از موارد رایج که در عمل مورد استفاده قرار می‌گیرد، شباهت کسینوسی است. فرض کنید دو کاربر شبیه هم هستند، اما یکی از آن‌ها در رتبه‌بندی محافظه‌کارتر است. هر دو عاشق فیلم آمریکان آنتونی از منموهان دسای هستند. پل به آن ۵ ستاره و روان ۴ ستاره داده است. اگر کماکان از فرمول فاصله استفاده کنید، این دو کاربر حتی اگر سلیقه‌ی مشابهی داشته باشند، می‌توانند همسایه‌ی یکدیگر نباشند، شباهت کسینوسی فاصله‌ی بین دو بردار<sup>۲</sup> را اندازه نمی‌گیرد. در عوض، زوایای دو بردار را با هم مقایسه می‌کند. در برخورد با چنین مواردی بهتر است چنین رویکردی را داشته باشیم. موضوع شباهت کسینوسی خارج از محدوده‌ی این کتاب است، اما اگر از KNN استفاده می‌کنید، درباره‌ی آن جست‌وجو کنید!

1. Cosine similarity
2. vectors

## انتخاب ویژگی‌های خوب



برای تعیین پیشنهادات، امتیاز کاربران به دسته‌بندی فیلم‌ها را در اختیار داشتید. در عوض اگر از کاربران می‌خواستید به عکس‌گره‌ها امتیاز بدهند چه؟ اطلاعات کاربران را بر اساس تشابه امتیازدهی این عکس‌ها خواهید داشت.

در این شرایط، قاعده‌تاً موتور پیشنهادات عملکرد بدتری خواهد داشت، چون این «ویژگی‌ها» ارتباط چندانی با سلیقه در فیلم‌ها ندارند! یا فرض کنید از کاربران می‌خواهید به فیلم‌ها امتیاز بدهند تا بر اساس آن بتوانید به این افراد پیشنهاداتی ارائه کنید - اما تنها از آن‌ها می‌خواهید به انیمیشن‌های داستان اسباب‌بازی، داستان اسباب‌بازی ۲ و داستان اسباب‌بازی ۳ امتیاز بدهند. این به شما اطلاعات چندانی درباره‌ی سلیقه‌ی فیلمی کاربران نمی‌دهد.

هنگامی که با KNN کار می‌کنید، انتخاب ویژگی‌های مناسب برای مقایسه بسیار مهم است. انتخاب ویژگی‌های مناسب یعنی:

- ویژگی‌هایی که مستقیماً با فیلم‌هایی که می‌خواهید توصیه کنید مرتبط باشد.
- ویژگی‌هایی که صرفاً یک سوگیری مشخص ندارند (برای مثال، اگر از کاربران بخواهید فقط به فیلم‌های کم‌دی امتیاز بدهند، به شما نمی‌گوید که آیا آن‌ها فیلم‌های اکشن را دوست دارند یا خیر)

به نظر شما رتبه‌بندی روش خوبی برای توصیه‌ی فیلم است؟ شاید من به سریال شونو بیشتر از برنامه‌ی تلویزیونی جویندگان خانه امتیاز دادم، اما در واقع زمان بیشتری را صرف تماشای جویندگان خانه کنم. چه پیشنهادی برای بهبود سیستم پیشنهادات نتفلیکس دارید؟

برگردیم به مثال نانوایی: آیا می‌توانید دو ویژگی خوب و دو ویژگی بد برای نانوایی انتخاب کنید؟ شاید بعد از آگهی در روزنامه یا دوشنبه‌ها لازم باشد نان‌های بیشتری پخت کنید.

وقتی نوبت به انتخاب ویژگی‌های خوب می‌رسد، هیچ پاسخ درستی وجود ندارد. شما باید تمام جوانب را در نظر بگیرید.

### تمرین

۱۰.۳ نتفلیکس میلیون‌ها کاربر دارد. در مثال قبلی به پنج همسایه‌ی نزدیک برای ساختن سیستم پیشنهادات نگاه کردیم. این تعداد خیلی کم است؟ یا خیلی زیاد است؟

### مقدمه‌ای بر یادگیری ماشینی



KNN الگوریتمی واقعاً مفید و مقدمه‌ای برای ورود به دنیای جادویی یادگیری ماشینی است! یادگیری ماشینی چیزی نیست جز هوشمندتر کردن کامپیوتر. قبل‌تر یک نمونه از یادگیری ماشینی را مرور کردید: ایجاد یک سیستم پیشنهادات. بیایید چند نمونه‌ی دیگر را بررسی کنیم.

### OCR

OCR مخفف تشخیص نوری حرف<sup>۱</sup> است. به این معنا که می‌توانید از یک صفحه متن عکس بگیرید و کامپیوتر به طور خودکار متن را برای شما بخواند. گوگل از OCR برای دیجیتالی کردن کتاب‌ها استفاده می‌کند. OCR چگونه کار می‌کند؟ برای مثال این عدد را در نظر بگیرید.

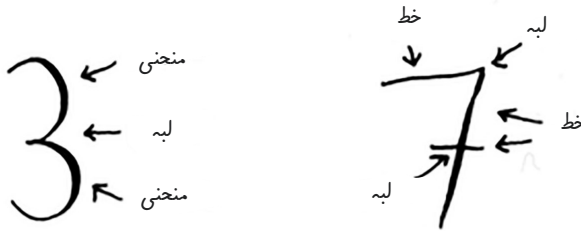
7

چگونه به طور خودکار می‌توان فهمید که این عدد چیست؟ برای این کار می‌توانید از KNN استفاده کنید:

۱. تصاویر زیادی از اعداد را مرور کرده و ویژگی‌های آن اعداد را استخراج کنید.

1. optical character recognition

۲. هنگامی که یک تصویر جدید دریافت کردید، ویژگی‌های آن تصویر را استخراج می‌کند، و نزدیک‌ترین همسایه‌های آن را پیدا می‌کند!  
این همان مسئله‌ی پرتقال و گریپ فروت است. به طور کلی، الگوریتم‌های OCR خطوط، لبه‌ها و منحنی‌ها را اندازه‌گیری می‌کنند.



بعد وقتی یک کاراکتر جدید دریافت کردید، می‌توانید همان ویژگی‌ها را از آن استخراج کنید.

استخراج ویژگی در OCR بسیار پیچیده‌تر از مثال میوه است. اما باید بدانید که فناوری‌های پیچیده هم بر اساس ایده‌های ساده مانند KNN ساخته می‌شوند. می‌توانید از همین ایده‌ها برای تشخیص گفتار یا تشخیص چهره استفاده کنید. وقتی عکسی را در فیس‌بوک بارگزاری می‌کنید، این پلتفرم گاهی آن‌قدر هوشمند است که افراد به تصویر درآمد در عکس را به طور خودکار تگ کند. این یک کاربرد واقعی یادگیری ماشینی است!

اولین مرحله‌ی OCR که در آن تصاویر اعداد را مرور می‌کنید و ویژگی‌ها را استخراج می‌کنید، آموزش نامیده می‌شود. اکثر الگوریتم‌های یادگیری ماشینی یک مرحله آموزش دارند: به کامپیوتر پیش از آنکه بتواند کاری انجام دهد، باید آموزش داده بشود. مثال بعدی شامل فیلترهای اسپم است و یک مرحله‌ی آموزش دارد.

## 1. training

## ساخت فیلتر اسپم

فیلترهای اسپم از الگوریتم ساده‌ی دیگری به نام دسته‌بندی‌کننده‌ی بیز ساده<sup>۱</sup> استفاده می‌کنند. در ابتدا، دسته‌بندی‌کننده‌ی بیز ساده را با تعدادی داده آموزش می‌دهید.

موضوع	اسپم؟
«پسورد خود را ریست کنید»	اسپم نیست
«شما برنده‌ی ۱ میلیون دلار شده‌اید»	اسپم
«پسوردت را برای من بفرست.»	اسپم
«شاهزاده‌ی نیجریه می‌خواهد ۱۰ میلیون دلار به شما منتقل کند»	اسپم
«تولدت مبارک»	اسپم نیست

فرض کنید ایمیلی با این مضمون دریافت می‌کنید که «یک میلیون دلار خود را دریافت کنید!» آیا این یک ایمیل اسپم است؟ می‌توانید این جمله را به کلمات تقسیم کنید. سپس، برای هر کلمه، ببینید احتمال نمایش آن کلمه در ایمیل اسپم چقدر است. برای مثال در این مدل بسیار ساده کلمه‌ی میلیون فقط در ایمیل‌های اسپم ظاهر می‌شود. بیز ساده احتمال اینکه چیزی اسپم باشد را تشخیص می‌دهد. عملکردی مشابه KNN دارد.

برای مثال، می‌توانید از بیز ساده برای دسته‌بندی میوه‌ها استفاده کنید: یک میوه‌ی بزرگ و قرمز دارید. احتمال اینکه این میوه گریپ‌فروت باشد، چقدر است؟ این یک الگوریتم ساده‌ی دیگر است که تا حد زیادی مؤثر است. ما این الگوریتم‌های ساده را دوست داریم!



## پیش‌بینی بازار سهام

یک مثال از مسئله‌ای که پیاده‌سازی یادگیری ماشین در آن دشوار است: پیش‌بینی تغییرات ارزش بازار سهام. چگونه ویژگی‌های خوبی را در بازار سهام انتخاب می‌کنید؟ فرض کنید شما می‌گویید اگر دیروز ارزش سهام افزایش پیدا کرده، امروز هم افزایش می‌یابد. آیا این ویژگی خوبی است؟ یا فرض کنید می‌گویید که ارزش سهام همیشه در ماه می کاهش می‌یابد. آیا این رویکرد کارساز است؟ هیچ راه تضمینی برای استفاده از ارقام گذشته برای پیش‌بینی عملکرد آینده وجود ندارد. پیش‌بینی آینده زمانی که متغیرهای زیادی در آن دخیل هستند سخت و تقریباً غیرممکن است.

## جمع‌بندی

امیدوارم با خواندن این بخش دیدی به دست آورده باشید از کارهایی که با KNN و یادگیری ماشین می‌توان انجام داد. یادگیری ماشین حوزه‌ی جالبی است که در صورت علاقه می‌توانید دانش خود را در آن گسترش دهید:

• KNN برای طبقه‌بندی و رگرسیون استفاده می‌شود و شامل نگاه‌کردن به k- نزدیک‌ترین همسایه است.

• طبقه‌بندی = دسته‌بندی<sup>۱</sup> در یک گروه.

• رگرسیون = پیش‌بینی پاسخ (مانند عدد).

• استخراج ویژگی به معنای تبدیل یک آیتم (مانند میوه یا کاربر) در لیستی از اعداد قابل مقایسه است.

• انتخاب ویژگی‌های خوب بخش مهمی از یک الگوریتم موفق KNN است.





## ۱۱ | گام‌های بعدی

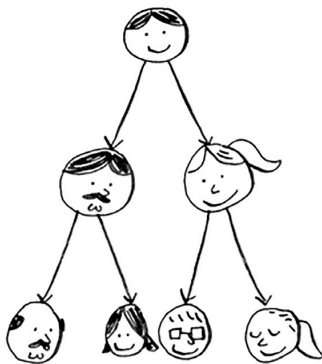


در این فصل

• مرور مختصری خواهیم کرد به ۱۰ الگوریتمی که در این کتاب به آن‌ها پرداخته نشده است و به دلایل مفید بودن آن‌ها اشاره می‌کنیم.

• به شما سرنخ‌هایی داده خواهد شد تا با توجه به علائق خود درباره‌ی آن‌ها مطالعات بیشتری داشته باشید.

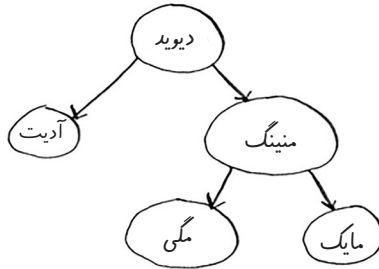
## درخت



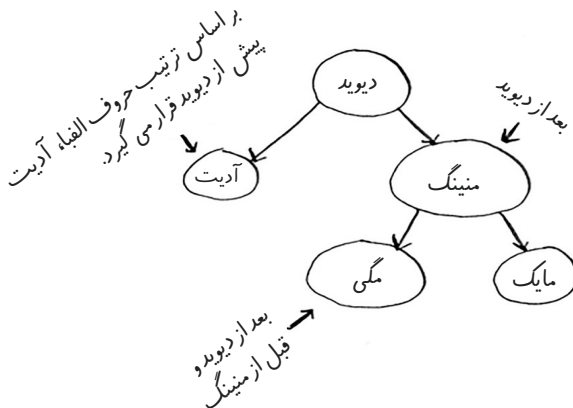
بیابید به مثال جست‌وجوی دودویی برگردیم. هنگامی که کاربر وارد فیس‌بوک می‌شود، فیس‌بوک باید در یک آرایه‌ی بزرگ نگاه کند و ببیند آیا نام کاربری مورد نظر وجود دارد یا خیر. گفتیم سریع‌ترین راه برای جست‌وجو در این آرایه، اجرای جست‌وجوی دودویی است. اما یک مشکل وجود دارد: هر بار که یک کاربر جدید ثبت‌نام می‌کند، نام کاربری او را در آرایه

وارد می‌کنید. سپس باید آرایه را دوباره مرتب کنید، زیرا جست‌وجوی دودویی فقط با آرایه‌های مرتب شده کار می‌کند. بهتر نخواهد بود اگر بتوانید نام کاربری را فوراً در روزنه‌ی سمت راست آرایه وارد کنید، تا بعد از آن مجبور نشوید آرایه را مرتب کنید؟ این ایده‌ی اصلی ساختمان داده‌ی درخت جست‌وجوی دودویی است.

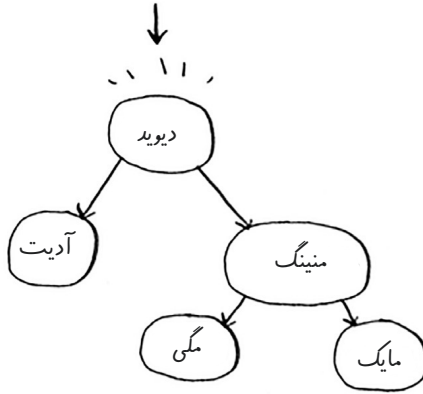
درخت جست‌وجوی دودویی به این شکل است:



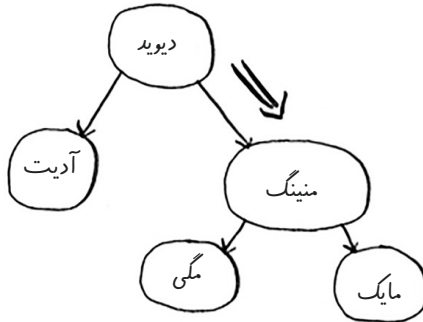
برای هر گره، گره‌های سمت چپ آن از نظر ارزش کوچک‌تر و گره‌های سمت راست از نظر ارزش بزرگ‌تر هستند.



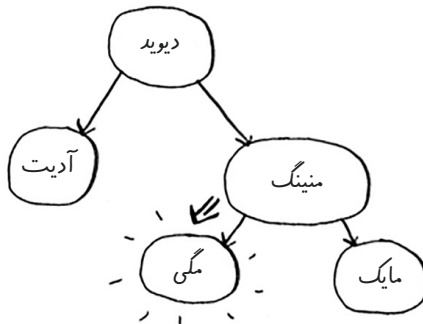
فرض کنید به دنبال نام کاربری مگی هستید. شما از گره‌ی ریشه شروع می‌کنید.



نام مگی بعد از نام دیوید می‌آید، پس به سمت راست می‌روید.



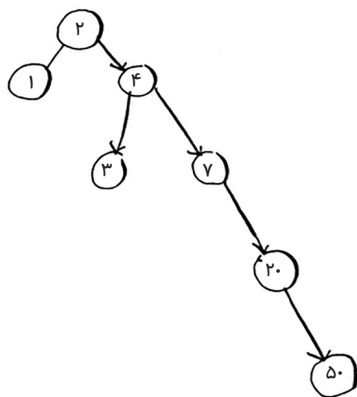
نام مگی قبل از نام منینگ می‌آید، پس به سمت چپ می‌روید.



مگی پیدا شد! تقریباً مانند اجرای یک جست‌وجوی دودویی است! جست‌وجوی یک عنصر در درخت جست‌وجوی دودویی به طور متوسط زمان  $O(\log n)$  و در بدترین حالت زمان  $O(n)$  دارد. جست‌وجوی یک آرایه‌ی مرتب شده در بدترین حالت به زمان  $O(\log n)$  نیاز دارد، ممکن است تصور کنید آرایه‌ی مرتب شده بهتر است. اما درخت جست‌وجوی دودویی به طور متوسط برای درج و حذف سرعت بسیار بیشتری دارد.

	آرایه	درخت جست‌وجوی دودویی
جست‌وجو	$O(\log n)$	$O(\log n)$
درج	$O(n)$	$O(\log n)$
حذف	$O(n)$	$O(\log n)$

درختان جست‌وجوی دودویی عیب‌هایی هم دارند: اول‌ازهمه، شما دسترسی تصادفی دریافت نمی‌کنید. شما نمی‌توانید بگویید «عنصر پنجم این درخت را به من بدهید.» این زمان‌های عملکرد نیز به طور متوسط و متگی به توازن درخت هستند. فرض کنید یک درخت نامتوازن مانند آنچه در ادامه آمده، دارید.

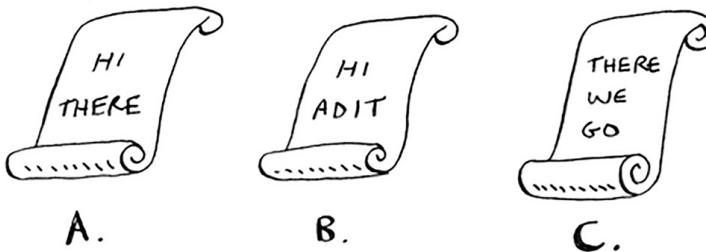


همان‌طور که مشاهده می‌کنید درخت به سمت راست متمایل است. این درخت عملکرد چندان خوبی ندارد، زیرا متوازن نیست. درخت‌های جست‌وجوی دودویی خاصی وجود دارند که خود را متوازن می‌کنند. یک مثال درخت قرمز-سیاه<sup>۱</sup> است. پس در چه شرایطی از درخت‌های جست‌وجوی دودویی استفاده می‌شود؟ درختان B، نوع خاصی از درخت دودویی، معمولاً برای ذخیره‌ی داده‌ها در پایگاه‌های داده<sup>۲</sup> استفاده می‌شوند. اگر به پایگاه‌های داده یا ساختارهای داده‌ی پیشرفته‌تر علاقه دارید، این موارد را بررسی کنید:

- درختان B
- درختان قرمز-سیاه
- هیپ‌ها<sup>۳</sup>
- درخت گسترده<sup>۴</sup>

## ایندکس‌های معکوس<sup>۵</sup>

در اینجا یک نسخه‌ی بسیار ساده از نحوه‌ی کار یک موتور جست‌وجو ذکر شده است. فرض کنید سه صفحه‌ی وب با این محتوای ساده دارید.



1. Red-black tree
2. databases
3. Heaps
4. Splay trees
5. Inverted indexes

HI	A, B
THERE	A, C
ADIT	B
WE	C
GO	C

بیا یاد بگیریم با این محتوا یک جدول هش بسازیم.

کلیدهای جدول هش کلمات هستند و مقادیر به شما می گویند که هر کلمه در چه صفحاتی ظاهر می شود. حال فرض کنید کاربری به دنبال hi است. بیا یاد بگیریم hi در چه صفحاتی نمایش داده می شود.

HI	A, B
----	------

درست است! در صفحات A و B ظاهر می شود. اجازه بدهید آن صفحات را به عنوان نتیجه به کاربر نشان بدهیم. یا فرض کنید کاربر there را جست و جو کرده است. می دانید که در صفحات A و C نشان داده می شود. خیلی آسان است، مگر نه؟ این یک ساختمان داده ی کاربردی و مفید است: یک هش که کلمات را به مکان هایی که در آن ظاهر می شوند نگاشت می کند. به این ساختمان داده، ایندکس معکوس می گویند و معمولاً برای ساخت موتورهای جست و جو استفاده می شود. اگر به جست و جو علاقه مند هستید، این نقطه ی خوبی برای شروع است.

## تبدیل فوریه<sup>۱</sup>

تبدیل فوریه یکی از آن الگوریتم های نادر است: درخشان، زیبا و با هزاران کاربرد. بتر اکسپلیند<sup>۲</sup> (وب سایتی عالی که ریاضیات را به سادگی توضیح می دهد) بهترین تشبیه را برای تبدیل فوریه می کند: اگر یک اسموتی به تبدیل فوریه بدهید، این الگوریتم، مواد تشکیل دهنده ی آن را به شما می گویند. یا اگر یک آهنگ تحویل بگیرد، فرکانس های آن را جدا می کند.

1. The Fourier transform
2. <https://betterexplained.com/>

به نظر می‌رسد که این ایده‌ی ساده کاربردهای بسیاری دارد. برای مثال، اگر بتوانید یک آهنگ را بر اساس فرکانس‌ها جدا کنید، می‌توانید فرکانس‌هایی را که برایتان مهم هستند تقویت کنید. می‌توانید باس را تقویت کنید و تریبل<sup>۱</sup> را مخفی کنید. تبدیل فوریه برای پردازش سیگنال‌ها محشر است. می‌توانید از آن برای فشرده‌سازی موسیقی استفاده کنید. ابتدا یک فایل صوتی را به یادداشت‌های تشکیل دهنده‌ی آن تقسیم می‌کنید. تبدیل فوریه دقیقاً به شما می‌گوید که هر نُت در کل آهنگ چندبار استفاده شده است.

بنابراین می‌توانید فقط از نت‌هایی که مهم نیستند خلاص شوید. فرمت MP۳ این‌گونه کار می‌کند!

موسیقی تنها نوع سیگنال دیجیتال نیست. فرمت JPG یکی دیگر از فرمت‌های فشرده است و به همین صورت عمل می‌کند. از تبدیل فوریه برای پیش‌بینی زلزله‌های آینده و تجزیه و تحلیل DNA استفاده می‌شود.

می‌توانید از آن برای ساخت اپلیکیشنی مانند Shazam استفاده کنید که حدس می‌زند چه آهنگی در حال پخش است. تبدیل فوریه کاربردهای زیادی دارد. احتمال اینکه با آن برخورد کنید زیاد است!

## الگوریتم‌های موازی<sup>۲</sup>

سه مورد بعدی در مورد مقیاس‌پذیری و کار با داده‌های عظیم است. در گذشته، کامپیوترها سریع‌تر و سریع‌تر می‌شدند. اگر می‌خواستید سرعت الگوریتم خود را بهتر کنید، می‌توانستید چند ماه صبر کنید تا خود کامپیوترها سریع‌تر شوند. اما ما به پایان آن دوره نزدیک شده‌ایم. در عوض، لپ‌تاپ‌ها و رایانه‌ها با چندین هسته عرضه می‌شوند. برای اینکه الگوریتم‌های خود را سریع‌تر کنید، باید آن‌ها را طوری تغییر بدهید که بی‌درنگ و به صورت موازی در تمام هسته‌ها اجرا بشوند!

اگر بخواهم با یک مثال ساده تشریح کنم در بهترین حالت یک الگوریتم مرتب‌سازی تقریباً زمان  $O(n \log n)$  دارد. روشن است که نمی‌توانید یک آرایه را در زمان  $O(n)$

1. treble
2. Parallel algorithms

مرتب کنید، مگر اینکه از یک الگوریتم موازی استفاده کنید! یک نسخه‌ی موازی از مرتب‌سازی سریع وجود دارد که یک آرایه را در زمان  $O(n)$  مرتب می‌کند.

طراحی الگوریتم‌های موازی، اطمینان از کارکرد صحیح آن‌ها و اطلاع از میزان افزایش سرعت سخت است. یک مورد را می‌توان با قطعیت گفت - افزایش سرعت خطی نیست. بنابراین اگر به جای یک هسته، دو هسته در لپ‌تاپ خود دارید، به این معنا نخواهد بود که الگوریتم شما به شکلی جادویی دو برابر سریع‌تر اجرا می‌شود. چند دلیل برای این مسئله وجود دارد:

- سربار مدیریت موازی<sup>۱</sup> - فرض کنید باید یک آرایه از ۱۰۰۰ آیتم را مرتب کنید. چگونه این وظیفه را بین دو هسته تقسیم می‌کنید؟ آیا به هر هسته ۵۰۰ آیتم می‌دهید تا مرتب بشوند و سپس دو آرایه‌ی مرتب شده را در یک آرایه‌ی مرتب شده‌ی بزرگ ادغام می‌کنید؟ ادغام این دو آرایه زمان‌بر است.

- متعادل‌سازی بار<sup>۲</sup> - فرض کنید ۱۰ کار برای انجام باشد، بنابراین به هر هسته ۵ کار می‌دهید. اما هسته‌ی A تمام کارهای آسان را انجام می‌دهد، بنابراین در ۱۰ ثانیه کار خود را به پایان می‌رساند، در حالی که هسته‌ی B همه‌ی کارهای سخت را انجام می‌دهد، بنابراین یک دقیقه طول می‌کشد. یعنی هسته‌ی A به مدت ۵۰ ثانیه بی‌کار مانده بود در صورتی که هسته‌ی B مشغول انجام تمام کارها بود! چگونه کار را به طور مساوی توزیع می‌کنید تا هر دو هسته به یک اندازه کار کنند؟

اگر به جنبه‌ی نظری عملکرد و مقیاس‌پذیری علاقه دارید، مطالعه‌ی الگوریتم‌های موازی می‌تواند برای شما جالب باشند!

1. Overhead of managing the parallelism
2. Load balancing



## نگاشت‌کاهش<sup>۱</sup>

نوع خاصی از الگوریتم موازی هست که روز به روز محبوب‌تر می‌شود: الگوریتم توزیع شده<sup>۲</sup>. اگر به دو تا چهار هسته نیاز داشته باشید، اجرای یک الگوریتم موازی روی لپ‌تاپ کفایت می‌کند، اما اگر به صدها هسته نیاز داشته باشید چه؟ پس می‌توانید الگوریتم خود را برای اجرا در چندین ماشین بنویسید. الگوریتم نگاشت‌کاهش یک الگوریتم توزیع شده‌ی محبوب است. می‌توانید به کمک ابزار منبع باز و محبوب Apache Hadoop از آن استفاده کنید.

### چرا الگوریتم‌های توزیع شده مفید هستند؟

فرض کنید جدولی با میلیارد‌ها یا تریلیون‌ها ردیف دارید و می‌خواهید یک کوئری SQL پیچیده را بر روی آن اجرا کنید. شما نمی‌توانید آن را در MySQL اجرا کنید، چرا که پس از چند میلیارد ردیف عملکرد خوبی نخواهد داشت. از نگاشت‌کاهش در Hadoop استفاده کنید!

یا فرض کنید باید فهرست طولانی از مشاغل را پردازش کنید. پردازش هر کار ۱۰ ثانیه طول می‌کشد و شما باید ۱ میلیون شغل را مانند این پردازش کنید. اگر این کار را روی یک دستگاه انجام بدهید، ماه‌ها طول می‌کشد! اگر بتوانید آن را روی ۱۰۰ کامپیوتر اجرا کنید، تنها در چند روز کار شما به انجام می‌رسد.

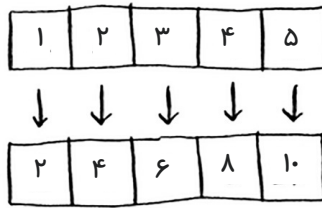
استفاده از الگوریتم‌های توزیع شده زمانی ایده‌آل است که کار زیادی برای انجام دادن دارید و می‌خواهید زمان لازم برای انجام آن را تسریع کنید. نگاشت‌کاهش به طور خاص از دو ایده‌ی ساده ساخته شده است: تابع map و تابع reduce.

1. MapReduce
2. distributed algorithm

## تابع map

تابع مپ ساده است: یک آرایه دریافت می‌کند و همان تابع را برای هر آیتیم در آرایه اعمال می‌کند. به عنوان مثال، در اینجا ما هر آیتیم در آرایه را دو برابر می‌کنیم:

```
>>> arr1 = [1, 2, 3, 4, 5]
>>> arr2 = map(lambda x: 2 * x, arr1)
[2, 4, 6, 8, 10]
```



arr2 شامل [2, 4, 6, 8, 10] است - هر عنصر در arr1 دو برابر شد! دو برابر کردن یک عنصر بسیار سریع است. حال فرض کنید تابعی را اعمال می‌کنید که پردازش آن به زمان بیشتری نیاز دارد. به این شبه کد نگاه کنید:

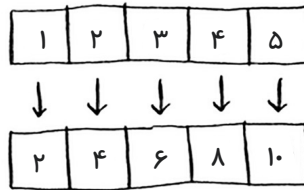
```
>>> arr1 = # A list of URLs
>>> arr2 = map(download_page, arr1)
```

در اینجا لیستی از URL ها دارید و می‌خواهید هر یک از این صفحات را دانلود کنید و محتویات آن را در arr2 ذخیره کنید. این فرآیند برای هر یک از صفحات می‌تواند چند ثانیه طول بکشد. اگر هزار URL داشته باشید، ممکن است چند ساعت طول بکشد!

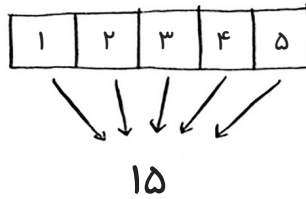
خوب نمی‌شد اگر ۱۰۰ کامپیوتر داشتید و مپ کار را به طور خودکار میان همه‌ی آن‌ها پخش می‌کرد؟ در نتیجه شما در هر نوبت ۱۰۰ صفحه را دانلود می‌کردید و کار بسیار سریع‌تر پیش می‌رفت! این ایده‌ی اصلی پشت «map» در MapReduce است.

## تابع reduce

تابع reduce گاهی می‌تواند گیج‌کننده باشد. طرح کلی این است که شما یک لیست کامل از آیتم‌ها را به یک مورد «کاهش» بدهید. با map، از یک آرایه به آرایه دیگر می‌روید.



با reduce، یک آرایه را به تنها یک آیتم تبدیل می‌کنید.



مثال:

```
>>> arr1 = [1, 2, 3, 4, 5]
>>> reduce(lambda x,y: x+y, arr1)
15
```

در این حالت، تمام عناصر آرایه را جمع می‌کنید:  $1 + 2 + 3 + 4 + 5 = 15$ ! من در اینجا جزئیات بیشتری از reduce ارائه نمی‌کنم، زیرا در این مورد آموزش‌های آنلاین زیادی در دسترس است.

نگاشت‌کاهش از این دو مفهوم ساده برای اجرای کوئری در مورد داده‌ها در چندین ماشین استفاده می‌کند. وقتی مجموعه داده‌ی بزرگی دارید (میلیاردها ردیف)، نگاشت‌کاهش می‌تواند در عرض چند دقیقه به شما پاسخ بدهد، جایی که پایگاه داده‌ی سنتی ممکن است ساعت‌ها طول بکشد.

## فیلترهای بلوم<sup>۱</sup> و هایپر لاگ لاگ<sup>۲</sup>

فرض کنید اپلیکیشن Reddit را مدیریت می‌کنید. می‌خواهید ببینید لینکی که کاربر پست می‌کند، آیا قبلاً در سایت فرستاده شده است یا خیر. پیام‌هایی که تکراری نباشند با ارزش‌تر در نظر گرفته می‌شوند. بنابراین باید بفهمید که آیا این لینک قبلاً ارسال شده است یا خیر.

یا فرض کنید شما گوگل هستید و در حال خزیدن<sup>۳</sup> یا به عبارتی پیمایش در صفحات وب هستید. تنها در صورتی می‌خواهید یک صفحه‌ی وب را پیمایش کنید که پیش از این پیمایش نشده باشند. بنابراین باید بفهمید که آیا این صفحه قبلاً بررسی شده است یا خیر.

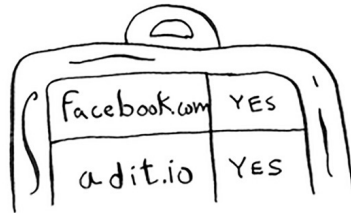
یا فرض کنید وب‌سایت bit.ly را اداره می‌کنید که یک کوتاه‌کننده‌ی URL است. نمی‌خواهید اجازه بدهید کاربران از طریق سایت شما به وب‌سایت‌های مخرب هدایت بشوند. شما مجموعه‌ای از URL‌ها دارید که مخرب در نظر گرفته می‌شوند. اکنون باید بفهمید که آیا کاربر را به یک URL در آن مجموعه هدایت می‌کنید یا خیر.

همه‌ی این نمونه‌ها مشکل یکسانی دارند. شما یک مجموعه‌ی بسیار بزرگ دارید.



1. Bloom filters
2. HyperLogLog
3. crawling

شما یک آیتم جدید دارید و می‌خواهید ببینید که آیا به آن مجموعه تعلق دارد یا خیر. شما می‌توانید این کار را به سرعت با هش انجام دهید. برای مثال، فرض کنید گوگل یک هش بزرگ دارد که کلیدهای آن تمام صفحاتی را شامل می‌شود که در آن‌ها پیمایش انجام شده است.



می‌خواهید ببینید آیا قبلاً `adit.io` را پیمایش کرده‌اید یا خیر. آن را در هش جست‌وجو کنید.

`adit.io → YES`

`adit.io` کلیدی است که در هش وجود دارد، بنابراین شما قبلاً آن را پیمایش کرده‌اید. میانگین زمان جست‌وجو برای جدول‌های هش  $O(1)$  است. `adit.io` این هش است، بنابراین شما قبلاً به آن سر زده‌اید. شما آن را با زمان ثابت پیدا کردید. بسیار خوب!

با این تفاوت که این هش باید بزرگ باشد. گوگل تریلیون‌ها صفحه‌ی وب را ایندکس می‌کند. اگر این هش شامل تمامی URL‌هایی باشد که گوگل ایندکس کرده است، فضای زیادی را اشغال می‌کند. `Reddit` و `bit.ly` مشکل فضای یکسانی دارند. وقتی با داده‌های زیادی سروکار دارید، باید خلاق باشید!

## فیلترهای بلوم

فیلترهای بلوم یک راه حل ارائه می‌کنند. فیلترهای بلوم ساختمان داده‌های احتمالی هستند و پاسخی به شما می‌دهند که هرچند امکان دارد اشتباه باشد اما احتمالاً صحیح باشد. به

جای هش، می‌توانید از فیلتر بلوم بپرسید که آیا قبلاً این URL را پیمایش کرده یا خیر. یک جدول هش به شما پاسخ دقیقی می‌دهد. فیلتر بلوم به شما پاسخی می‌دهد که احتمالاً درست است:

- موارد مثبت کاذب ممکن است. احتمال دارد گوگل اعلام کند: «شما قبلاً این سایت را پیمایش کرده‌اید»، حتی اگر این کار را نکرده باشید.

- منفی کاذب امکان‌پذیر نیست. اگر فیلتر بلوم اعلام کند: «شما این سایت را پیمایش نکرده‌اید»، پس حتماً همین‌طور است.

فیلترهای بلوم بسیار عالی هستند زیرا فضای بسیار کمی را اشغال می‌کنند. یک جدول هش باید هر آدرسی را که گوگل پیمایش کرده، ذخیره کند، اما فیلتر بلوم نیازی به انجام این کار ندارد. زمانی که به پاسخ دقیقی نیاز ندارید، بسیار خوب هستند، مانند تمام این مثال‌ها. برای bit.ly اشکالی ندارد که اعلام کند: «این سایت ممکن است مخرب باشد، بنابراین بیشتر مراقب باشید.»

## هایپرلاگ لاگ

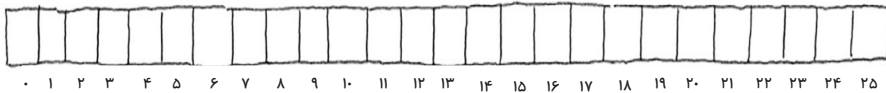
در همین رابطه الگوریتم دیگری با نام HyperLogLog وجود دارد. فرض کنید گوگل می‌خواهد تعداد جست‌وجوهای منحصر به فرد انجام شده به دست کاربران را بشمارد. یا فرض کنید آمازون می‌خواهد تعداد آیتم‌های منحصر به فردی را که امروز کاربران به آن‌ها نگاه می‌کنند، بشمارد. برای پاسخ به این پرسش‌ها فضای زیادی نیاز است! در مورد گوگل باید گزارشی از تمام جست‌وجوهای منحصر به فرد داشته باشید. وقتی کاربر چیزی را جست‌وجو می‌کند، باید ببینید که آیا از قبل در گزارش وجود دارد یا خیر. اگر نه، باید آن را به گزارش اضافه کنید. حتی برای یک روز هم لاگ بسیار بزرگی می‌شود!

هایپرلاگ لاگ تعداد عناصر منحصر به فرد در یک مجموعه را تخمین می‌زند. درست مانند فیلترهای بلوم، پاسخ دقیقی به شما نمی‌دهد، اما بسیار به پاسخ دقیق نزدیک است و برای عملیاتی مشخص به نسبت شرایط دیگر تنها کسری از حافظه را استفاده می‌کند.

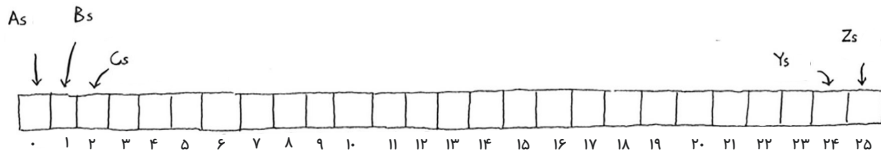
اگر داده‌های زیادی دارید و با پاسخ‌های تقریبی راضی هستید، الگوریتم‌های احتمالی را بررسی کنید!

## الگوریتم‌های SHA

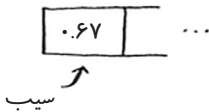
هش را از فصل ۵ به خاطر دارید؟ محض یادآوری، فرض کنید یک کلید دارید و می‌خواهید مقدار مرتبط را در یک آرایه قرار بدهید.



شما از یک تابع هش استفاده می‌کنید تا به شما بگوید که مقدار را در چه روزنه‌ای قرار بدهید.



و شما مقدار را در آن روزنه قرار می‌دهید.



این روش به شما امکان می‌دهد جست‌وجوهایی با زمان-ثابت انجام بدهید. هنگامی که می‌خواهید مقدار یک کلید را بدانید، می‌توانید دوباره از تابع هش استفاده کنید و در زمان  $O(1)$  به شما می‌گوید کدام روزنه را باید بررسی کنید.

در این حالت، شما می‌خواهید که تابع هش توزیع خوبی در اختیار شما بگذارد. بنابراین یک تابع هش یک رشته را می‌گیرد و شماره‌ی روزنه‌ی آن رشته را به شما بازمی‌گرداند.

## مقایسه‌ی فایل‌ها

“hello”  $\Rightarrow$  2cf24db...

یکی دیگر از توابع هش یک تابع الگوریتم هش ایمن<sup>۱</sup> (SHA) است. با توجه به یک رشته، SHA یک هش برای آن رشته به شما می‌دهد.

شاید این اصطلاح برای شما کمی گیج‌کننده باشد. SHA یک تابع هش است. یک هش تولید می‌کند که فقط یک رشته‌ی کوتاه است. تابع هش برای جدول‌های هش از یک رشته به ایندکس آرایه می‌رود، در حالی که SHA از رشته‌ای به رشته‌ی دیگر می‌رود. SHA یک هش متفاوت برای هر رشته تولید می‌کند.

“hello” ⇒ 2cf24db...

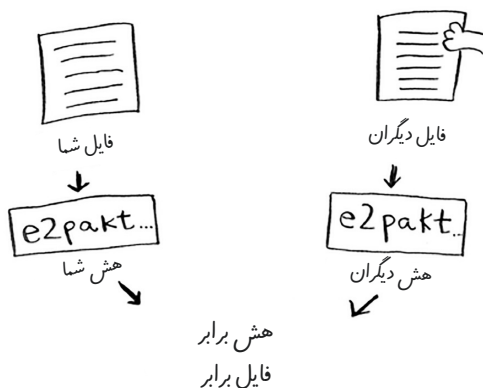
“algorithm” ⇒ b1eb2ec...

“password” ⇒ 5e88489...

### نکته

هش‌های SHA طولانی هستند و در اینجا کوتاه شده‌اند.

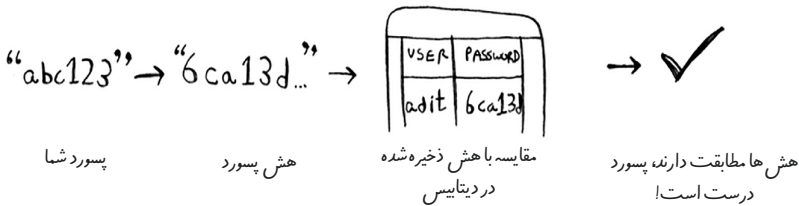
شما می‌توانید از SHA برای تشخیص یکسان بودن دو فایل استفاده کنید. این زمانی مفید است که فایل‌های بسیار بزرگی دارید. فرض کنید یک فایل ۴ گیگابایتی دارید. می‌خواهید بررسی کنید که آیا دوست شما همان فایل بزرگ را دارد یا خیر. لازم نیست فایل پر حجم خود را ایمیل کنید. در عوض، می‌توانید هش SHA را محاسبه کرده و آن را مقایسه کنید.





## چک کردن رمزهای عبور

SHA همچنین زمانی مفید است که بخواهید رشته‌ها را بدون اینکه نشان دهید رشته‌ی اصلی چیست مقایسه کنید. به عنوان مثال، فرض کنید سرویس جیمیل هک می‌شود و مهاجم تمام رمزهای عبور را سرقت می‌کند! آیا رمز عبور شما فاش شده است؟ نه، اینطور نیست. گوگل رمز عبور اصلی را ذخیره نمی‌کند، فقط هش SHA رمز عبور را ذخیره می‌کند! وقتی رمز عبور خود را تایپ می‌کنید، گوگل آن را هش می‌کند و آن را با هش موجود در پایگاه داده‌ی خود بررسی می‌کند.



بنابراین فقط هش‌ها را مقایسه می‌کند - لازم نیست رمز عبور شما را ذخیره کند! SHA معمولاً برای هش رمزهای عبور مانند این استفاده می‌شود. این یک هش یک طرفه است. می‌توانید هش یک رشته را دریافت کنید.

$abc123 \rightarrow 6ca13d$

اما با در اختیار داشتن هش نمی‌توانید رشته‌ی اصلی را به دست بیاورید.

$? \leftarrow 6ca13d$

در نتیجه اگر هکری هش‌های SHA را از جیمیل به دست بیاورد، نمی‌تواند آن هش‌ها را به رمزهای عبور اصلی تبدیل کند! شما می‌توانید رمز عبور را به هش تبدیل کنید، اما برعکس آن امکان پذیر نیست.

SHA در واقع خانواده‌ای از الگوریتم‌ها است: SHA-0، SHA-1، SHA-2 و SHA-3. در زمان نگارش این کتاب، SHA-0 و SHA-1 دارای نقاط ضعفی هستند. اگر از الگوریتم SHA برای هش رمز عبور استفاده می‌کنید، از SHA-2 یا SHA-3 استفاده کنید. استاندارد طلایی برای توابع هش رمز عبور در حال حاضر bcrypt است (اگرچه امنیت هیچ‌کدام کاملاً تضمین شده نیست).

## هشینگ حساس به مکان'

SHA یک ویژگی مهم دیگر نیز دارد: به محل حساس نیست. فرض کنید یک رشته دارید و یک هش برای آن ایجاد می‌کنید.

`dog` → `cd6357`

اگر فقط یک کاراکتر از رشته را تغییر دهید و هش را دوباره ایجاد کنید، کاملاً متفاوت می‌شود!

`dot` → `e392da`

این نکته‌ی مثبتی است چون مهاجم نمی‌تواند هش‌ها را مقایسه کند و ببیند به شکستن رمز عبور نزدیک شده است یا خیر.

گاهی اوقات، برعکس است: یک تابع هش حساس به مکان می‌خواهید. اینجاست که Simhash وارد می‌شود. اگر تغییر کوچکی در یک رشته ایجاد کنید، Simhash هش ایجاد می‌کند که تنها تفاوت اندکی دارد. این به شما امکان می‌دهد هش‌ها را مقایسه کنید و ببینید دو رشته چقدر شبیه هم هستند که ویژگی بسیار مفیدی است!

- گوگل از Simhash برای شناسایی موارد تکراری هنگام پیمایش وب استفاده می‌کند.
- یک معلم می‌تواند از Simhash استفاده کند تا ببیند آیا دانش‌آموزان مقاله‌های خود را از وب کپی کرده‌اند یا خیر.

- Scribd به کاربران امکان می‌دهد اسناد یا کتاب‌هایی را برای اشتراک‌گذاری با

### 1. Locality-sensitive hashing

دیگران بارگزاری کنند. اما Scribd نمی‌خواهد کاربران محتوای دارای حق چاپ را آپلود کنند! این سایت می‌تواند از Simhash استفاده کند تا ببیند که آیا یک فایل بارگزاری شده به طور مثال مشابه کتاب هری پاتر است یا خیر و اگر چنین است، به طور خودکار جلوی آن را بگیرد.

Simhash زمانی مفید است که می‌خواهید ایتیم‌های مشابه را بررسی کنید.

## تبادل کلید دیفی-هلمن<sup>۱</sup>

در اینجا خوب است به الگوریتم دیفی-هلمن اشاره کنیم، چرا که یک مسئله‌ی قدیمی را به روشی زیبا حل می‌کند. شیوه‌ی رمزگذاری یک پیام به نحوی که فقط در مقصد قابل خواندن باشد چیست؟

ساده‌ترین راه این است که یک رمز مانند  $a = 1$ ،  $b = 2$  و غیره ایجاد کنید. سپس اگر پیام «۴،۱۵،۷» را برای شما ارسال کنم، می‌توانید آن را به «d,o,g» ترجمه کنید. اما برای چنین کاری، هر دو طرف باید بر سر یک رمز به توافق برسیم. ما نمی‌توانیم در خود ایمیل به نتیجه برسیم، زیرا ممکن است شخصی ایمیل شما را هک کند، رمز را پیدا کرده، و پیام‌های ما را رمزگشایی کند. عجب، حتی اگر همدیگر را حضوری ببینیم، باز هم ممکن است کسی رمز را حدس بزند - کار چندان پیچیده‌ای نیست. پس رمز را باید هر روز تغییر بدهیم. اما برای این کار لازم است هر روز همدیگر را ملاقات کنیم!

حتی اگر هر روز موفق به تغییر آن شویم، رمز ساده‌ای مانند این به راحتی با یک حمله‌ی brute-force شکسته می‌شود. فرض کنید من پیام «۵، ۱۳، ۱۹، ۱۶، ۲۴، ۱۳، ۱۶، ۶، ۹» را می‌بینم. حدس من این است که در این مورد از  $a = 1$ ،  $b = 2$  و غیره استفاده می‌شود.

۹	۶	۱۳	۱۳	۱۶	۲۴	۱۶	۱۹	۱۳	۵
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
i	f	m	m	p	x	p	s	m	e

1. Diffie-Hellman key exchange

این کلمات معنی خاصی ندارند. بیایید  $a = 2$ ،  $b = 3$  و غیره را امتحان کنیم.

۹	۶	۱۳	۱۳	۱۶	۲۴	۱۶	۱۹	۱۳	۵
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
h	e	l	l	o	w	o	r	l	d

نتیجه داد! یک رمز ساده مانند این به راحتی شکسته می‌شود. آلمانی‌ها از رمز بسیار پیچیده‌تری در جنگ جهانی دوم استفاده کردند، اما آن هم شکسته شد. دیفی هلمن این دو مشکل را حل می‌کند:

- نیازی نیست هر دو از رمز اطلاع داشته باشند. بنابراین نیازی به ملاقات برای توافق بر سر رمز نیست..

- رمزگشایی پیام‌های رمزنگاری شده بسیار دشوار است.

دیفی هلمن دو کلید دارد: یک کلید عمومی و یک کلید خصوصی. کلید عمومی دقیقاً همین است: عمومی. می‌توانید آن را در وبسایت خود ارسال کنید، به دوستان خود ایمیل کنید، یا هر کار دیگری که می‌خواهید با آن انجام بدهید. نیازی نیست آن را پنهان کنید. وقتی شخصی می‌خواهد برای شما پیامی ارسال کند، آن را با استفاده از کلید عمومی رمزنگاری می‌کند. یک پیام رمزنگاری شده فقط با استفاده از کلید خصوصی قابل رمزگشایی است. تا زمانی که شما تنها فردی باشید که کلید خصوصی را در اختیار دارید، فقط شما می‌توانید این پیام را رمزگشایی کنید!

الگوریتم دیفی هلمن همچنان به همراه جایگزین آن RSA استفاده می‌شود. اگر به رمزنگاری علاقه دارید، دیفی هلمن نقطه شروع خوبی است: ساده و ظریف است و دنبال کردن آن چندان سخت نیست.

## برنامه‌ریزی خطی<sup>۱</sup>

بهترین الگوریتم را برای آخر ننگه داشتیم. برنامه‌ریزی خطی یکی از جالب‌ترین چیزهایی است که می‌شناسم.

برنامه‌ریزی خطی برای به حداکثر رساندن یک چیز مشخص با در نظر داشتن به برخی محدودیت‌ها استفاده می‌شود. برای مثال، فرض کنید شرکت شما دو محصول، پیراهن و ساک پارچه‌ای و شلوار تولید می‌کند. پیراهن به ۱ متر پارچه و ۵ دکمه نیاز دارد. ساک‌ها به ۲ متر پارچه و ۲ دکمه نیاز دارند. ۱۱ متر پارچه و ۲۰ دکمه دارید. شما به ازای هر پیراهن ۲ دلار و برای هر ساک ۳ دلار به دست می‌آورید. برای به حداکثر رساندن سود باید چه تعداد پیراهن و ساک تولید کنید؟

در اینجا ضمن آن‌که به اقلامی که در اختیار دارید محدود شده‌اید تلاش دارید سود خود را به حداکثر برسانید.

مثال دیگر: فرض کنید یک سیاستمدار هستید و می‌خواهید تعداد آرای خود را به حداکثر برسانید. تحقیقات شما نشان داده است که به طور متوسط یک ساعت کار (بازاریابی، تحقیق و غیره) برای هر رأی از ساکنان سانفرانسیسکو، یا یک و نیم ساعت / رأی از یک ساکن شیکاگو طول می‌کشد. به حداقل ۵۰۰ ساکن سانفرانسیسکایی و ۳۰۰ ساکن شیکاگو نیاز دارید. ۵۰ روز فرصت دارید. همچنین برای شما ۲ دلار / ساکن سانفرانسیسکو در مقابل ۱ دلار / ساکن شیکاگو هزینه دارد. کل بودجه‌ی شما ۱۵۰۰ دلار است. حداکثر تعداد کل آرای که می‌توانید کسب کنید (سانفرانسیسکو + شیکاگو) چقدر است؟

در اینجا شما سعی می‌کنید رأی‌ها را به حداکثر برسانید، و زمان و پول شما محدود است.

ممکن است تصور کنید، «در مورد بسیاری از موضوعات بهینه‌سازی در این کتاب صحبت شده است. ارتباط آن‌ها با برنامه‌ریزی خطی چیست؟ همه‌ی الگوریتم‌های گراف را می‌توان از طریق برنامه‌ریزی خطی انجام داد. برنامه‌ریزی خطی یک چارچوب بسیار کلی‌تر است و

1. Linear programming

مسئله‌های گراف زیرمجموعه‌ای از آن هستند. امیدوارم دود از کله‌ی شما بلند شده باشد! برنامه نویسی خطی از الگوریتم غیر مرکب<sup>۱</sup> استفاده می کند. این یک الگوریتم پیچیده است، به همین دلیل است که در این کتاب به آن پرداخته نشده است. اگر به بهینه‌سازی علاقه دارید، برنامه‌ریزی خطی را جست‌وجو کنید!

## سخن آخر

امیدوارم این مرور سریع ۱۰ الگوریتم به شما نشان داده باشد که چه موارد بیشتری برای کشف باقی مانده است. بهترین راه برای یادگیری این است که موضوع مورد علاقه‌ی خود را پیدا کنید و در آن شیرجه بزنید. این کتاب شالوده‌ای مستحکم برای چنین کاری فراهم کرده است.

## پاسخ تمرین‌ها



### فصل ۱

۱.۱ فرض کنید یک لیست مرتب شده از ۱۲۸ نام دارید، و با جست‌وجوی دودویی در میان آن لیست به جست‌وجوی می‌پردازید. حداکثر تعداد مراحل چه مقدار خواهد بود؟

پاسخ: ۷.

۱.۲ تعداد اعضای لیست را دو برابر مقدار قبلی در نظر بگیرید. این بار حداکثر تعداد مراحل چقدر خواهد بود؟

پاسخ: ۸.

۱.۳ نام یک نفر را در نظر دارید و می‌خواهید شماره تلفن آن شخص را در دفترچه تلفن پیدا کنید.

پاسخ:  $O(\log n)$ .

۱.۴ شما یک شماره تلفن دارید و می‌خواهید نام شخص را در دفترچه تلفن پیدا کنید. (نکته: باید در تمام صفحات دفترچه بگردید!)

پاسخ:  $O(n)$ .

۱.۵ شما می‌خواهید شماره‌ی هر فرد نوشته‌شده در دفترچه تلفن را بخوانید.  
پاسخ:  $O(n)$ .

۱.۶ شما می‌خواهید شماره‌ی افرادی را بخوانید که نام آن‌ها با حرف A شروع شده است.  
پاسخ:  $O(n)$ . ممکن است فکر کنید، «من فقط برای ۱/۲۶ حروف این کار را انجام می‌دهم بنابراین زمان اجرا باید  $O(n/2)$  باشد. یک قانون ساده که باید به خاطر بسپارید این است که از اعدادی که جمع، تفریق، ضرب یا تقسیم می‌شوند چشم‌پوشی کنید. هیچکدام از این زمان‌های اجرای Big O صحیح نیستند:  
 $O(n + 26)$ ،  $O(n - 26)$ ،  $O(n * 26)$ ،  $O(n / 26)$   
چرا؟ اگر کنجکاو هستید، به «بازبینی نماد O بزرگ» در فصل ۴ مراجعه کنید و درباره‌ی ثابت‌ها در نماد Big O مطالعه کنید (یک ثابت فقط یک عدد است؛ ۲۶ عدد ثابت این پرسش بود).

## فصل ۲

۲.۱ فرض کنید در حال ساخت یک برنامه برای پیگیری امور مالی خود هستید.

۱. خورد و خوراک روزمره

۲. سینما

۳. حق عضویت SFBC

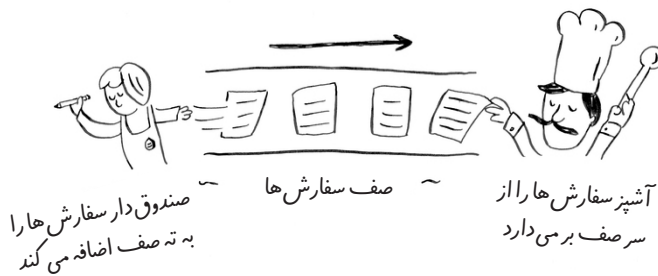
هرروز خرج‌های خود را یادداشت می‌کنید. در پایان ماه، هزینه‌ها را بررسی کرده و میزان هزینه‌های خود را جمع می‌زنید. بنابراین، شما تعداد زیادی درج‌کردن و مقدار کمی خواندن دارید. برای این برنامه باید از میان آرایه و لیست کدام را استفاده کنید؟

پاسخ: هرروز هزینه‌ها را به لیست اضافه می‌کنید و ماهی یک بار همه‌ی هزینه‌ها را مطالعه می‌کنید. آرایه‌ها دارای سرعت خواندن بالا و درج‌کردن هستند. لیست‌های پیوندی دارای



سرعت خواندن کند و درج سریع هستند. از آن جایی که بیشتر از اینکه قرار باشد بخوانید درج می‌کنید، استفاده از یک لیست پیوندی منطقی است. همچنین، تنها در صورتی که بخواهید به عناصر تصادفی در لیست دسترسی داشته باشید، خواندن فهرست‌های پیوندی کند است. چون شما در حال خواندن تمام عنصر لیست هستید، لیست‌های پیوندی هم در خواندن به خوبی عمل می‌کنند. بنابراین یک لیست پیوندی راه حل خوبی برای این مشکل است.

**۲.۲** فرض کنید در حال ساخت یک برنامه برای رستوران‌ها هستید تا سفارش مشتری‌ها در آن ثبت بشود. برنامه‌ی شما باید فهرستی از سفارشات را ذخیره کند. گارسون‌ها مدام به این لیست سفارش‌هایی را اضافه می‌کنند، و آشپزها سفارشات را از لیست می‌بینند و آن‌ها را آماده می‌کنند. این یک صف سفارش است: گارسون‌ها سفارشات را به ته صف اضافه می‌کنند، و آشپزها اولین سفارش را از سر صف بر می‌دارند و آن را حاضر می‌کنند.



برای پیاده‌سازی این صف از میان آرایه و لیست پیوندی کدام را استفاده می‌کنید؟ (راهنمایی: لیست‌های پیوندی برای درج/حذف و آرایه‌ها برای دسترسی تصادفی مناسب هستند. کدام یک را در اینجا به کار می‌برید؟)

پاسخ: لیست پیوندی. درج‌های زیادی انجام می‌شود. (گارسون‌هایی که سفارش‌ها را اضافه می‌کنند)، و لیست‌های پیوندی برای آن‌ها مناسب‌تر هستند. شما نیازی به جست‌وجو یا دسترسی تصادفی ندارید (موردی که آرایه‌ها در آن برتری دارند)، چون سرآشپزها همیشه اولین سفارش را از صف خارج می‌کنند.

**۲.۳** بیابید یک آزمایش نظری انجام بدهیم. فرض کنید فیس بوک لیستی از نام‌های کاربری را نگه می‌دارد. هنگامی که شخصی سعی می‌کند به فیس بوک وارد بشود، نام کاربری او جست‌وجو می‌شود. اگر نام آن‌ها در لیست نام‌های کاربری باشد، می‌تواند وارد بشوند. به دفعات کاربران وارد فیس بوک می‌شوند، بنابراین جست‌وجوهای زیادی از طریق این لیست از نام‌های کاربری انجام می‌شود. فرض کنید فیس بوک از جست‌وجوی دودویی برای جست‌وجوی لیست استفاده می‌کند. جست‌وجوی دودویی نیاز به دسترسی تصادفی دارد. شما باید بتوانید فوراً به وسط لیست نام‌های کاربری برسید. با دانستن این موضوع، آیا لیست را به شکل یک آرایه یا یک لیست پیوندی پیاده‌سازی می‌کنید؟

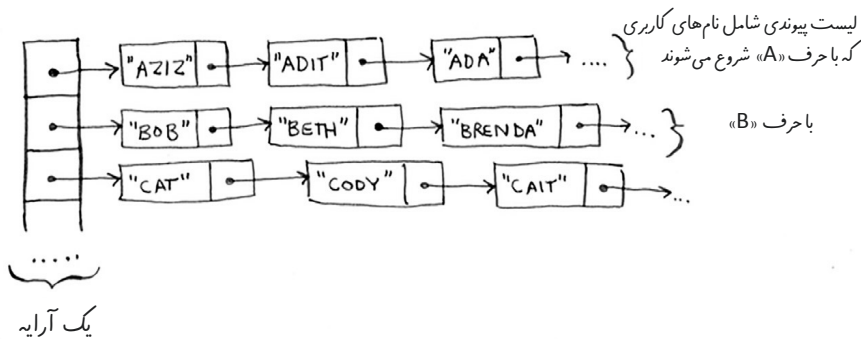
پاسخ: یک آرایه‌ی مرتب شده. آرایه‌ها به شما دسترسی تصادفی می‌دهند – شما می‌توانید یک عنصر را فوراً از وسط آرایه دریافت کنید. چنین کاری با لیست‌های پیوندی امکان‌پذیر نیست. برای رسیدن به عنصر میانی در یک لیست پیوندی، باید از عنصر اول شروع کنید و همه‌ی پیوندها را تا عنصر میانی دنبال کنید.

**۲.۴** همچنین کاربرها بیشتر اوقات در فیس بوک ثبت نام می‌کنند. فرض کنید تصمیم گرفتید از یک آرایه برای ذخیره‌ی لیست کاربران استفاده کنید. استفاده از آرایه برای درج چه جنبه‌های منفی به دنبال دارد؟ به طور خاص، فرض کنید از جست‌وجوی دودویی برای لاگین استفاده می‌کنید. وقتی کاربران جدیدی به آرایه اضافه می‌کنید چه اتفاقی می‌افتد؟

پاسخ: درج در آرایه‌ها کند است. همچنین، اگر از جست‌وجوی دودویی برای جست‌وجوی نام‌های کاربری استفاده می‌کنید، آرایه باید مرتب شود. فرض کنید شخصی به نام Adit B در فیس بوک ثبت نام کرده است. نام‌ها در انتهای آرایه درج خواهد شد. بنابراین باید هر بار که یک نام در آن درج می‌شود آرایه را مرتب کنید!

۲.۵ در واقعیت، فیس بوک از آرایه و لیست پیوندی برای ذخیره‌ی اطلاعات کاربر استفاده نمی‌کند.

بیابید یک ساختمان داده‌ی ترکیبی را در نظر بگیریم: آرایه‌ای از لیست‌های پیوندی. شما یک آرایه با ۲۶ اسلات دارید. هر اسلات به یک لیست پیوندی اشاره می‌کند. به عنوان مثال، اولین روزه در آرایه به یک لیست پیوندی اشاره می‌کند که شامل همه‌ی نام‌های کاربری است که با حرف «A» شروع می‌شوند. اسلات دوم به یک لیست پیوندی اشاره می‌کند که شامل همه‌ی نام‌های کاربری است که با حرف «B» شروع می‌شوند و غیره.



فرض کنید B Adit در فیس بوک ثبت نام کرده است، و شما می‌خواهید او را به لیست اضافه کنید. شما به اسلات ۱ در آرایه می‌روید، به لیست پیوندی برای اسلات ۱ رفته و Adit را به پایان لیست اضافه می‌کنید. حال، فرض کنید می‌خواهید Zakhir را جست‌وجو کنید. به اسلات ۲۶ می‌روید، که به لیست پیوندی از همه‌ی نام‌های Z اشاره می‌کند. سپس در آن لیست جست‌وجو می‌کنید تا Zakhir را پیدا کنید.

این ساختمان داده‌ی ترکیبی را با آرایه‌ها و لیست‌های پیوندی مقایسه کنید. به نسبت آرایه و فهرست پیوندی برای جست‌وجو و درج کندتر است یا سریع‌تر؟ نیازی نیست زمان اجرای O بزرگ را محاسبه کنید، فقط اینکه ساختمان داده‌ی جدید سریع‌تر است یا کندتر؟

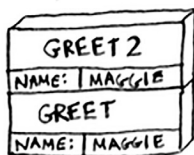
پاسخ: جست‌وجو: کندتر از آرایه‌ها، سریع‌تر از لیست‌های پیوندی. درج: سریع‌تر از آرایه‌ها، مدت زمان مشابه لیست‌های پیوندی. بنابراین برای جست‌وجو از یک آرایه کندتر است، اما سریع‌تر یا یکسان با لیست‌های پیوندی در تمامی موارد است.

ما در مورد ساختمان داده‌ی هیبریدی دیگری به نام جدول هش در این کتاب صحبت می‌کنیم. این موضوع به شما دید می‌دهد تا بتوانید ساختمان داده‌های پیچیده‌تر را به کمک ساختارهای ساده‌تر بسازید.

بالاخره فیس‌بوک از چه روشی استفاده می‌کند؟ احتمالاً از ده‌ها پایگاه داده‌ی مختلف، با ساختمان داده‌های متفاوت استفاده می‌کند: جدول‌های هش، درخت‌های -B، و غیره. آرایه‌ها و لیست‌های پیوندی، بلوک‌های سازنده‌ی این ساختمان داده‌های پیچیده‌تر هستند.

### فصل ۳

۳.۱ فرض کنید من یک پشته‌فراخوانی مانند این را به شما نشان می‌دهم.



تنها بر اساس این پشته‌فراخوانی، چه اطلاعاتی می‌توان داد؟  
پاسخ:

- ابتدا تابع greet با name = maggie فراخوانی می‌شود.
- سپس تابع greet تابع greet2 را با name = maggie فراخوانی می‌کند.
- در این مرحله، تابع greet در حالت ناتمام و تعلیق شده است.
- فراخوانی تابع فعلی، تابع greet2 است.
- پس از اتمام فراخوانی این تابع، فعالیت تابع greet از سر گرفته می‌شود.

**۳.۲** فرض کنید به طور تصادفی یک تابع بازگشتی نوشتید که تا ابد اجرا می‌شود. همان‌طور که دیدید، کامپیوتر شما برای هر فراخوانی تابع، حافظه را در پشته تخصیص می‌دهد. وقتی تابع بازگشتی شما تا ابد اجرا بشود چه اتفاقی برای پشته می‌افتد؟

پاسخ: پشته برای همیشه رشد می‌کند. هر برنامه فضای محدودی در پشته فراخوانی دارد. وقتی فضای برنامه‌ی شما تمام شود (که در نهایت اتفاق می‌افتد)، با یک خطای سرریز پشته<sup>۱</sup> خارج می‌شود.

## فصل ۴

**۴.۱** کد را برای تابع `sum` قبلی بنویسید.

پاسخ:

```
def sum(list):  
    if list == []:  
        return 0  
    return list[0] + sum(list[1:])
```

**۴.۲** یک تابع بازگشتی برای شمارش تعداد آیتم‌های یک لیست بنویسید.

پاسخ:

```
def count(list):  
    if list == []:  
        return 0  
    return 1 + count(list[1:])
```

1. Stack-overflow error

**۴.۳** حداکثر عدد را در یک لیست بیابید.

پاسخ:

```
def max(list):
    if len(list) == 2:
        return list[0] if list[0] > list[1] else list[1]
    sub_max = max(list[1:])
    return list[0] if list[0] > sub_max else sub_max
```

**۴.۴** جست‌وجوی دودویی را باز فصل ۱ به خاطر دارید؟ این الگوریتم، این یک الگوریتم تقسیم و حل نیز هست. آیا می‌توانید برای جست‌وجوی دودویی، صورت پایه و صورت بازگشتی را پیدا کنید؟

پاسخ: صورت پایه برای جست‌وجوی دودویی، آرایه‌ای با یک آیتیم است. اگر آیتیمی که به دنبال آن هستید با آیتیم موجود در آرایه مطابقت دارد، آن را پیدا کرده‌اید! در غیر این صورت، در آرایه نیست.

در حالت بازگشتی برای جست‌وجوی دودویی، آرایه را به نصف تقسیم می‌کنید، نیمی را کنار می‌گذارید و نیمی دیگر را با جست‌وجوی دودویی فرا می‌خوانید.

هر یک از این عملیات بر اساس نماد  $O$  بزرگ چقدر طول می‌کشد؟  
**۴.۵** چاپ مقدار هر عنصر در یک آرایه.

پاسخ:  $O(n)$

**۴.۶** دو برابر کردن مقدار هر عنصر در یک آرایه.

پاسخ:  $O(n)$

**۴.۷** دو برابر کردن مقدار فقط اولین عنصر در یک آرایه.

پاسخ:  $O(1)$

۴.۸ ایجاد جدول ضرب با تمام عناصر موجود در آرایه. بنابراین اگر آرایه‌ی شما [۱۰، ۸، ۷، ۳، ۲] باشد، ابتدا هر عنصر را در ۲ ضرب کنید، سپس هر عنصر را در ۳، سپس در ۷ و غیره ضرب کنید.  
پاسخ:  $O(n^2)$

## فصل ۵

کدام یک از این توابع هش معتبر هستند؟

۵.۱

$f(x) = \backslash$  ←..... تابع درازای هر ورودی «۱» را برمی‌گرداند

پاسخ: بله

۵.۲

$f(x) = \text{rand}()$  ←..... در نوبت یک عدد تصادفی را برمی‌گرداند

خیر

۵.۳

$f(x) = \text{next\_empty\_slot}()$  ←..... ایندکس اسلات خالی بعدی در جدول هش را برمی‌گرداند

پاسخ: خیر

۵.۴

$f(x) = \text{len}(x)$  ←..... از طول رشته برای ایندکس استفاده می‌کند

پاسخ: بله

فرض کنید این چهار تابع هش را دارید که با رشته‌ها کار می‌کنند:  
آ. «۱» را برای همه‌ی ورودی‌ها برگردانید.

ب. از طول رشته به عنوان ایندکس استفاده کنید.

پ. از اولین کاراکتر رشته به عنوان ایندکس استفاده کنید. بنابراین، تمام رشته‌هایی که با  $a$  شروع می‌شوند با هم هش می‌شوند و به غیره.

ت. هر حرف را با یک عدد اول نگاشت کنید:  $a = 2, b = 3, c = 5, d = 7, e = 11$  و غیره. برای یک رشته، تابع هش مجموع تمام کاراکترهای باقی‌مانده‌ی اندازه‌ی هش است. به عنوان مثال، اگر اندازه‌ی هش شما ۱۰ است و رشته‌ی آن «bag» است، ایندکس  $3 + 2 + 17 = 22 = 10\% = 2$  است. برای هر یک از این مثال‌ها، کدام تابع‌های هش توزیع خوبی ارائه می‌کنند؟ اندازه‌ی جدول هش را ۱۰ اسلات فرض کنید.

**۵.۵** دفترچه تلفنی که در آن کلیدها اسامی و مقادیر شماره‌ی تلفن هستند. اسامی عبارتند از: Dan, Bob, Ben, Esther.

پاسخ: توابع هش C و D توزیع خوبی ارائه می‌کنند.

**۵.۶** نگاشتی از اندازه‌ی باتری به توان باتری. اندازه‌ها A, AA, AAA و AAAA هستند.

پاسخ: توابع هش B و D توزیع خوبی ارائه می‌کنند.

**۵.۷** نگاشتی از عنوان کتاب به نویسنده‌ها. عناوین *Maus*, *Fun Home* و *Watchmen* هستند.

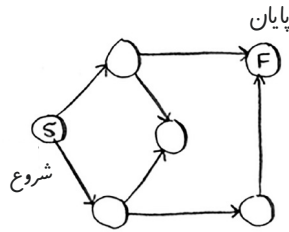
پاسخ: توابع هش B, C و D توزیع خوبی ارائه می‌کنند.



## فصل ۶

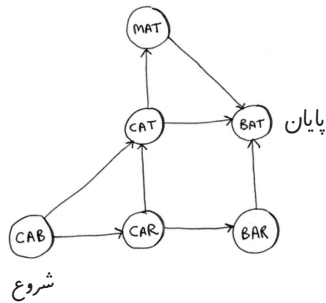
برای یافتن راه حل، الگوریتم جست‌وجوی سطح اول را روی هر یک از این گراف‌ها اجرا کنید.

۶.۱ طول کوتاه‌ترین مسیر را از ابتدا تا انتها پیدا کنید.



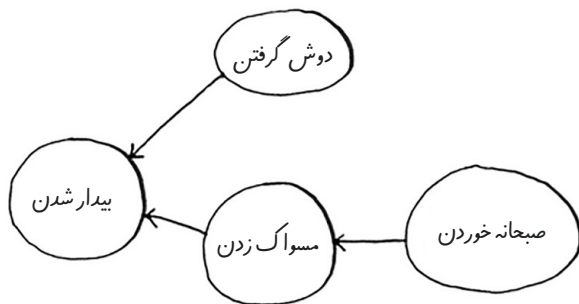
پاسخ: طول کوتاه‌ترین مسیر ۲ است.

۶.۲ طول کوتاه‌ترین مسیر را از «cab» تا «bat» بیابید.



پاسخ: طول کوتاه‌ترین مسیر ۲ است.

۶.۳ گراف کوچکی از برنامه‌ی صبحگاهی من به این شکل است:



درستی یا نادرستی هر یک از این سه لیست را مشخص کنید.

A.

۱. بیدار شدن
۲. دوش گرفتن
۳. صبحانه خوردن
۴. مسواک زدن

B.

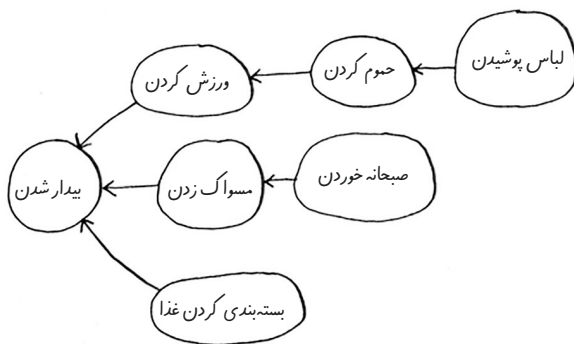
۱. بیدار شدن
۲. مسواک زدن
۳. صبحانه خوردن
۴. دوش گرفتن

C.

۱. دوش گرفتن
۲. بیدار شدن
۳. مسواک زدن
۴. صبحانه خوردن

پاسخ: A- نادرست. B- درست. C- نادرست.

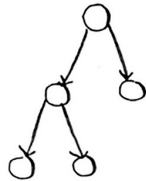
۶.۴ در اینجا یک گراف بزرگ‌تر وجود دارد. یک لیست صحیح برای این گراف تهیه کنید.



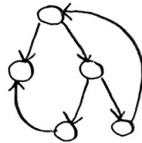
پاسخ: ۱- بیدار شدن. ۲- ورزش کردن ۳- دوش گرفتن ۴- مسواک زدن ۵- لباس پوشیدن ۶- ناهار را بسته بندی کدن ۷- صبحانه خوردن

۶.۵ کدام یک از تصاویری که در ادامه آمده علاوه بر گراف درخت هم هستند؟

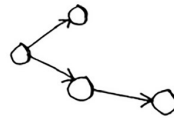
A.



B.



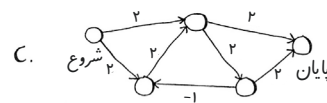
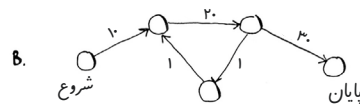
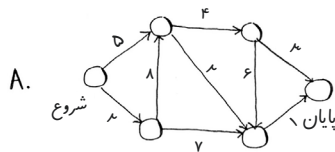
C.



پاسخ: A - درخت است، B - درخت نیست c - درخت است. آخرین مثال فقط یک درخت کناری است. درختان زیرمجموعه ای از گراف‌ها هستند. بنابراین یک درخت همیشه یک گراف است، اما یک گراف ممکن است یک درخت باشد یا نباشد.

## فصل ۷

۷.۱ در هر یک از این گراف‌ها وزن کوتاه‌ترین مسیر ابتدا تا انتها چقدر است؟



پاسخ‌ها: A—8; B—60; ج- سوال انحرافی. هیچ کوتاه‌ترین مسیری ممکن نیست (چرخه‌ی وزن منفی<sup>۱</sup>).

## فصل ۸

**۸.۱** شما برای یک شرکت مبلمان کار می‌کنید و باید تولیدات این شرکت را به سراسر کشور ارسال کنید. شما باید بسته‌ها را در کامیون بار بزنید. بسته‌ها اندازه‌های مختلفی دارند و شما سعی می‌کنید حداکثر استفاده را از فضای کامیون داشته باشید. بسته‌ها را چگونه انتخاب می‌کنید تا از فضای داخل کامیون به شکل بهینه استفاده کنید؟ از استراتژی حریصانه استفاده کنید. آیا این الگوریتم شما را به راه حل بهینه می‌رساند؟

پاسخ: استراتژی حریصانه به این صورت است که بزرگ‌ترین جعبه‌ای را که در فضای باقی‌مانده جا می‌شود، انتخاب کنید و آن قدر این عمل را تکرار کنید که دیگر امکان جادادن جعبه‌های بیشتری وجود نداشته باشد. خیر، این راه حل بهینه را در اختیار شما قرار نمی‌دهد.

**۸.۲** شما به اروپا می‌روید و هفت روز فرصت دارید تا جایی که زمان اجازه بدهد از مکان‌های گوناگون بازدید کنید. شما یک ارزش نقطه‌ای به هر مورد اختصاص می‌دهید (میزان اشتیاق به دیدن آن) و تخمین می‌زنید که بازدید هر کدام چقدر طول می‌کشد. چگونه می‌توانید مجموع ارزش را به حداکثر برسانید (بازدید از تمام مکان‌هایی که واقعاً دوست دارید از آن‌ها بازدید کنید)؟ با یک استراتژی حریصانه. آیا این راه حلی بهینه به شما می‌دهد؟

پاسخ: به انتخاب گزینه‌ای با بالاترین ارزش نقطه‌ای که هنوز در زمان باقی‌مانده می‌توانید انجام دهید، ادامه دهید. وقتی امکان آن نبود که کار بیشتری انجام دهید،

1. negative-weight cycle

متوقف شوید. خیر، این راه حل بهینه را به شما نمی‌دهد.

هر یک از الگوریتم‌های زیر را بررسی کنی که آیا یک الگوریتم حریصانه هستند یا خیر.  
**۸.۳** مرتب‌سازی سریع

پاسخ: خیر.

**۸.۴** جست‌وجوی سطح اول

پاسخ: خیر.

**۸.۵** الگوریتم دایجسترا

پاسخ: بله.

**۸.۶** یک پستچی باید به ۲۰ خانه بسته تحویل بدهد. او باید کوتاه‌ترین مسیری را که به تمام ۲۰ خانه می‌رود، پیدا کند. آیا این یک مسئله‌ی ان‌پی کامل است؟

پاسخ: بله.

**۸.۷** پیدا کردن بزرگ‌ترین دسته در یک مجموعه از افراد (یک دسته به مجموعه افرادی گفته می‌شود که همه‌ی افراد آن مجموعه یکدیگر را می‌شناسند). آیا این یک مسئله‌ی ان‌پی کامل است؟

پاسخ: بله.

**۸.۸** شما مشغول رسم‌کردن نقشه‌ی ایالات متحده هستید و باید ایالت‌های مجاور را با

رنگ‌های مختلف رنگ‌آمیزی کنید. باید حداقل تعداد رنگ مورد نیاز خود را پیدا کنید تا هیچ دو ایالت مجاور هم‌رنگ نباشند. آیا این یک مسئله‌ی ان پی کامل است؟

پاسخ: بله.

## فصل ۹

**۹.۱** فرض کنید می‌توانید یک آیتم دیگر را سرقت کنید: پخش‌کننده‌ی MP۳. وزن آن ۱ پوند است و ۱۰۰۰ دلار ارزش دارد. آیا باید آن را بدزدید؟

پاسخ: بله. سپس می‌توانید MP۳ پلیئر، آیفون و گیتار را به ارزش کل ۴۵۰۰ دلار بدزدید.

**۹.۲** فرض کنید می‌خواهید به پیک‌نیک بروید. یک کوله پشتی دارید که ۶ پوند را در خود جای می‌دهد و می‌توانید آیتم‌های زیر را بردارید. به هر کدام یک عدد اختصاص داده شده و هر چه مقدار آن بیشتر باشد، آیتم اهمیت بیشتری دارد:

- آب، ۳ پوند، ۱۰
- کتاب، ۱ پوند، ۳
- غذا، ۲ پوند، ۹
- ژاکت، ۲ پوند، ۵
- دوربین، ۱ پوند، ۶

مجموعه‌ی بهینه‌ای از آیتم‌هایی که باید در سفر کمپینگ همراه خود داشته باشید چیست؟

پاسخ: آب و غذا و دوربین را انتخاب کنید.

**۹.۳** برای محاسبه‌ی بزرگ‌ترین زیردنباله‌ی مشترک، میان blue و clues شبکه را ترسیم

و آن را پر کنید.

	C	L	U	E	S
B	.	.	.	.	.
L	.	۱	.	.	.
U	.	.	۲	.	.
E	.	.	.	۳	.

پاسخ:

## فصل ۱۰

۱۰.۱ در مثال نتفلیکس، فاصله‌ی بین دو کاربر مختلف را با استفاده از فرمول فاصله محاسبه کردید. اما همه‌ی کاربران با یک معیار ثابت به فیلم‌ها امتیاز نمی‌دهند. فرض کنید دو کاربر با نام‌های یوگی و بینکی سلیقه‌ی یکسانی در فیلم دارند. یوگی به هر فیلمی که دوست داشته باشد ۵ امتیاز می‌دهد، در حالی که بینکی گزیده‌تر عمل می‌کند و امتیاز ۵ را فقط برای بهترین فیلم‌ها نگه می‌دارد. آن‌ها سلیقه‌ی مشابهی دارند، اما طبق الگوریتم فاصله، همسایه نیستند. چه راهکاری برای استراتژی متفاوت آن‌ها در رتبه‌بندی در نظر می‌گیرید؟

پاسخ: می‌توانید از چیزی به نام نرمال‌سازی<sup>۱</sup> استفاده کنید. به میانگین رتبه‌بندی هر فرد نگاه کنید و از آن برای مقیاس‌بندی رتبه‌بندی آن‌ها استفاده کنید. برای مثال، ممکن است متوجه شوید که میانگین امتیاز بینکی ۳ است، در حالی که میانگین امتیاز یوگی ۵.۳ است. بنابراین رتبه‌بندی بینکی را کمی افزایش می‌دهید، تا زمانی که میانگین امتیاز او نیز به ۵.۳ برسد. سپس می‌توانید رتبه‌بندی آن‌ها را در همان مقیاس مقایسه کنید.

1. normalization

**۱۰.۲** فرض کنید نتفلیکس گروهی از «اینفلوئنسرها» را گزینش کرده است. به عنوان مثال، کوئنتین تارانتینو و وس اندرسون اینفلوئنسرهای نتفلیکس هستند، بنابراین رتبه‌بندی این افراد بیشتر از یک کاربر عادی حساب می‌شود. چه تغییراتی برای سیستم پیشنهادت نیاز است تا نسبت به رتبه‌بندی اینفلوئنسرها جانبدارانه عمل کند؟

پاسخ: هنگام استفاده از KNN می‌توانید به رتبه‌بندی افراد تأثیرگذار اهمیت بیشتری بدهید. فرض کنید سه همسایه دارید: جو، دیو و وس اندرسون (یک اینفلوئنسر). آن‌ها به فیلم پادوی گلف به ترتیب ۳، ۴ و ۵ امتیاز داده‌اند. به جای اینکه فقط میانگین رتبه‌بندی آن‌ها را بگیرید ( $3 + 4 + 5 / 3 = 4$  ستاره)، می‌توانید به رتبه‌بندی وس اندرسون وزن بیشتری بدهید:  $3 + 4 + 5 + 5/5 = 4,4$  ستاره.

**۱۰.۳** نتفلیکس میلیون‌ها کاربر دارد. در مثال قبلی به پنج همسایه‌ی نزدیک برای ساختن سیستم پیشنهادت نگاه کردیم. این تعداد خیلی کم است؟ یا خیلی زیاد است؟

پاسخ: خیلی کم است. اگر به همسایه‌های کمتری نگاه کنید، احتمال بیشتری وجود دارد که نتایج منحرف شوند. یک قانون خوب این است که اگر  $N$  کاربر دارید، باید به  $\sqrt{N}$  همسایه نگاه کنید.





