

Shape ۰-۱

پیشگفتار

سپاس گزاری ها

درباره این کتاب

مقدمه ای بر الگوریتم

مقدمه

آنچه درباره عملکرد خواهید آموخت

آنچه درباره حل مسائل خواهید آموخت

جستجوی دو دویی (Binary search)

یک راه بهتر برای جستجو

زمان مورد نیاز برای اجرا شدن (زمان اجرایی)

نماد O بزرگ (Big O notation)

زمان اجرایی الگوریتم ها به میزان نرخ ورودی های متفاوت افزایش پیدا میکند

نمودار زمان اجرایی بیگ او های مختلف (به تصویر کشیدن زمان اجرایی بیگ او های مختلف)

پیدا کردن بدترین حالت ممکن برای زمان اجرایی توسط Big O (بیگ او بدترین حالت زمان اجرایی را نشان میدهد)

چند نمودار از زمان های اجرایی بیگ او های رایج

جناب کارشناس فروش دوره گرد

خلاصه

مرتب سازی گزینشی (selection sort)

مموری چگونه کار میکند

آرایه ها (Arrays) و لیست های پیوندی (Linked lists)

لیست های پیوندی

آرایه ها

اصطلاحات فنی

درج داده در وسط لیست

حذف کردن

مرتب سازی گرینشی

خلاصه

بازگشت

بازگشت

بخش پایه و بخش بازگشته

استک (Stack یا پشته)

کال استک (call stack)

کال استک با استفاده از مفهوم بازگشته

خلاصه

مرتب سازی سریع (Quicksort)

تقسیم و تبخیر

مرتب سازی سریع

نگاهی دوباره به Big O notation

(merge sort vs Quicksort) مرتب سازی ادغامی vs مرتب سازی سریع

حالت میانگین vs بدترین حالت

خلاصه

(Hash tables) جداول هش

توابع هش

موارد استفاده

استفاده از جداول هش در جست و جو

جلوگیری از ورودی های تکراری

استفاده از جداول هش برای کش کردن

خلاصه

(collision) تلاقي

عملکرد

ضریب بار (load factor)

یک تابع هش خوب

خلاصه

جست و جوی اول عمق (Breadth first search)

مقدمه ای بر گراف

گراف چیست؟

جست و جوی اول عمق

پیدا کردن کوتاه ترین مسیر

صف ها

بکارگیری گراف

بکارگیری الگوریتم

زمان اجرا

خلاصه

الگوریتم دایجسترا (Dijkstra's Algorithm)

کار با الگوریتم دایجسترا

اصطلاحات فنی

مبادله کالا برای بدست آوردن پیانو

یال هایی با وزن منفی

پیاده سازی

خلاصه

الگوریتم های حریص (Greedy Algorithms)

مسئله ای زمان بندی کلاس ها

مسئله ای کوله پشتی

مسئله ای پوشش مجموعه

الگوریتم تقریبی

مسئله ای NP کامل (NP-Complete)

فروشنده دوره گرد، قدم به قدم

چطور میتوانید بگویید آیا مسئله NP کامل است؟

خلاصه

برنامه نویسی پویا (Dynamic Programming)

مسئله‌ی کوله پشتی

راه حل ساده

برنامه نویسی پویا

سوالات رایج مسئله‌ی کوله پشتی

چه اتفاقی می‌افتد اگر یک آیتم اضافه شود؟

چه اتفاقی می‌افتد اگر ترتیب سطرها عوض شوند

آیا میتوانیم جدول را به جای ستونی به صورت سطحی پر کنیم؟

چه اتفاقی می‌افتد اگر یک آیتم کوچک‌تر اضافه شود؟

آیا میتوانیم کسری از یک آیتم را بذدیم؟

بهینه‌سازی برنامه سفر خود

مدیریت کردن آیتم‌هایی که بهم وابسته‌اند

آیا ممکن است برای حل مسئله به بیشتر از دو کوله پشتی کوچک نیاز داشته باشیم؟

آیا ممکن است که بهترین راه حل، کل کوله پشتی را پر نکند؟

بزرگ‌ترین زیر رشته مشترک (Longest common substring)

درست کردن جدول

پر کردن جدول

راه حل

بزرگ‌ترین زیر دنباله مشترک (Longest common subsequence)

راه حل بزرگ‌ترین زیر دنباله مشترک

خلاصه

K همسایه نزدیک (KNN)(K nearest neighbor)

دسته‌بندی پر تقال و گریپ فروت

ساخت یک سیستم پیشنهاد دهنده

استخراج ویژگی‌ها

رگرسیون

انتخاب ویژگی‌های خوب و مناسب

مقدمه ای بر یادگیری ماشین

OCR

ساخت فیلتر اسپم

بیش بینی بازار سهام

خلاصه

بعد از این کجا برمیم؟

درخت ها(trees)

ایندهکس معکوس یا شاخص معکوس(inverted indexes)

تبدیل فوریه (Fourier transform)

الگوریتم های موازی

کاهش نگاشت (Map Reduce)

چرا الگوریتم های توزیع شده مفید هستند؟

تابع نگاشت (Map Function)

تابع کاهشی (Reduce Function)

فیلتر بلوم و هایبر لاغلگ (Bloom Filter and HyperLogLog)

فیلتر های بلوم

هایبر لاغلگ

الگوریتم های SHA

مقایسه فایل ها

بررسی رمزهای عبور

Wrong .-۱

Wrong .-۲

برنامه نویسی خطی

سخن آخر

جواب تمرین ها

نمایه

پیشگفتار

من اولین بار به عنوان تفسیر وارد دینای برنامه نویسی شدم. کتاب **visual basic 6 for dummies** پایه های برنامه نویسی رو به من یاد داد و من هم برای یادگیری بیشتر به خواندن ادامه دادم. اما موضوع الگوریتم ها و درک آنها برای من غیرقابل فهم بود. به یاد دارم که در حال لذت بردن از فهرست مطالب اولین کتاب الگوریتم خودم بودم و پیش خودم فکر میکردم که "من بالاخره این موضوعات رو درک میکنم!". اما موضوعاتش سنگین بود و من چند هفته بعد تسليم شدم تا زمانی که اولین استاد خوب الگوریتم را پیدا کردم و فهمیدم که ایده این الگوریتم ها چقدر ساده و جذاب بودند و من آنها را ندیده بودم.

چند سال پیش، من پست مصور بلاگ خودم را نوشتم. و یک یادگیرنده بصری هستم (کسی که حافظه دیداری خوبی دارد) و به شدت از روش مصور سازی خوشنم می‌آید. پس از آن، من چند پست در حیطه برنامه نویسی تابعی (فانکشنال)، گیت، یادگیری ماشین و همزمانی نوشتم، راستی من وقتی شروع کردم یک نویسنده متوسط بودم. توضیح مفاهیم تکنیکال سخت است. مثال زدن و توضیح دادن مفاهیم سخت وقتگیر هستند. پس راحتترین راه توضیح ندادن و لاپوشانی کردن آن دسته از مسائل است. من فکر میکردم خیلی خوب دارم در این زمینه عمل میکنم تا اینکه یکی از همکارانم بعد از اینکه یکی از پست های من معروف شد پیش من آمد و گفت "من پست تو رو خوندم اما هنوز متوجه این موضوع نشدم". من هنوز خیلی چیزها باید در باره نوشتمن یاد میگرفتم.

یک جایی در اواسط نوشتمن پست بلاگ منینگ (ناشر) با من تماس گرفت و از من پرسید که آیا میخواهم که یک کتاب مصور بنویسم. خب معلوم شد که ویرایشگرهای منینگ خیلی چیزها درباره توضیح مفاهیم تکنیکال میدانند، و آنها به من یاد دادند. من این کتاب را برای از بین بردن یک مشکل خاص نوشتمن: من میخواستم کتابی بنویسم که موضوعات سخت تکنیکی را به خوبی توضیح داده باشد و میخواستم که کتابی باشد که خواندنش راحت باشد. نوشه های من راه طولانی را از اولین پست بلاگ شروع کرده و امیدوارم که مطالعه این کتاب برای شما آسان و آموزنده باشد.

این کتاب طوری طراحی شده که دنبال کردنش راحت باشد. از موضوعات پیچیده و بزرگ دوری کردم. هر وقت مبحث جدیدی را شروع نکنم یا همان لحظه توضیح میدهم یا به شما میگویم که کی قرار است کامل درباره اش حرف بزنیم. برای اینکه بتوانید سوال هایی که حین مطالعه کتاب و مفاهیم اصلی برایتان پیش می آید را راحت بررسی و حل کنید و برای اینکه مطمئن بشوم که همراه من هستید مفاهیم اصلی را با تمرین و توضیحات مختلف تقویت کردم.

من به جای استفاده از مفاهیم سنگین و یا نماد های آبکی با مثال پیش میروم. هدف من این است که شما راحت بتوانید این مفاهیم را برای خودتان تصویر سازی کنید. همچنین من فکر میکنم بهترین روش یادگیری این است که از چیز هایی استفاده کنیم که قبل آن ها را تجربه کرده ایم و برای ما ملموس هستند و مثال ها میتوانند این کار را برای ما راحت کنند. به عنوان مثال وقتی شما تلاش میکنید قرق بین آرایه ها و لینک های پیوندی را بخاطر بیاورید (چیزی که کامل در فصل دو توضیح داده شده)، میتوانید سریع به یاد مثال "نحوه انتخاب صندلی برای نشستن روی صندلی های سینما" بیفتید. همچنین با اینکه ممکن است واضح باشد اما من یادگیرنده‌ی بصری هستم (حافظه دیداری قوی دارم). به همین دلیل این کتاب پر از عکس های مختلف است.

مفاهیم این کتاب با دقت زیادی انتخاب شده. هیچ نیازی به کتابی نیست که بیاید و تمامی الگوریتم های مربوط به مرتب سازی (sorting) را پوشش بدهد (به همین دلیل [khanacademy](#) و [Wikipedia](#) را داریم!). تمام الگوریتم هایی که این کتاب شامل میشود، کاربردی هستند.

من آنها را در شغل خودم که مهندسی نرم افزار باشد کاربردی میدانم. در کنار آنها این الگوریتم ها پایه‌ی خوبی برای درک مفاهیم پیچیده تر را برای شما فراهم میکنند.

امیدوارم خوشتان بیاید (به خوشی بخوانید) (بخونین و لذت ببرین)

نقشه راه

سه فصل اول این کتاب بنا های پایه را میگذارند:

فصل اول - تو این فصل شما اولین الگوریتم کاربردی‌تون رو یادمیگیرید: جست و جوی دودویی (Binary Search)

(از این به بعد بهش میگیم بازتری سرچ). و همچنین یادگیرید که چطور با استفاده از نماد ۰ بزرگ (که از این به بعد بهش میگیم بیگ او نویشن یا بیگ او) سرعت یک الگوریتم را تحلیل کنید. داخل این کتاب برای تشخیص و تحلیل اینکه آیا الگوریتم مورد نظر سریع است یا کند از بیگ او استفاده میکنیم.

فصل دوم - شما در این فصل دو تا از ساختار داده های بنیادی را یادمیگیرید.

آرایه ها و لینک های پیوسته. این ساختار داده ها توی کل این کتاب استفاده میشوند و از آنها برای ساخت ساختار داده های پیشرفته ای مانند جداول هش (فصل ۵) استفاده میشوند.

فصل سه - شما در این فصل درمورد مفهوم بازگشت یادمیگیرید. یک تکنیک دم دستی است که توسط خیلی از الگوریتم ها استفاده میشود (مانند مرتب سازی سریع (Quicksort) در فصل چهار).

به تجربه من، بیگ او نویشن و بازگشت جز چالشی ترین مباحث برای تازه کاران است. یه همین دلیل هم من سرعتم را در این قسمت ها کم کرده ام و وقت بیشتری برای توضیح آنها صرف کردم.

بقیه کتاب، الگوریتم هایی را که استفاده‌ی گسترده تری را دارد ارائه میکند:

- تکنیک های حل مسئله که در فصل های ۴، ۸ و ۹ پوشش داده شده.

اگر به مسئله ای برخورد کردید و مطمئن نیستید که چطور آنها را به بهترین شکل و به طور موثر حل کنید میتوانید از روش تقسیم و تسخیر(فصل ۴) یا برنامه نویسی پویا (فصل ۹) استفاده کنید. یا ممکن است بفهمید که هیچ راه حل موثری وجود ندارد که مسئله به بهترین خالت حل بشود که در آنجا میتوانید به جای همه آنها جواب تقریبی را با استفاده از الگوریتم های حریص (فصل ۸) بدست بیاورید.

جداول هش-که در فصل ۵ پوشش داده شده. جدول هش یک ساختار داده‌ی بسیار پرکاربرد و مفید است. این ساختار داده شامل مجموعه‌ی از کلید‌ها و مقادیر جفت شده است مانند اسم یک شخص و آدرس ایمیلش ، یا نام کاربری و رمز عبور مربوط به آن. اغراق کردن راجب اینکه جداول هش چقدر میتوانند مفید باشند سخت است. وقتی میخواه سوالی رو حل بکنم، اول با دو نقشه‌ای که اینها باشن بهش به سوال حمله :

۱. آیا میتونم از جدول هش استفاده کنم؟

۲. آیا میتونم به صورت گراف ترسیم کنم؟

الگوریتم های گراف-که در فصل های ۶ و ۷ پوشش داده شده. گراف‌ها یک راه برای مدل و ترسیم کردن یک شبکه است، مثل: شبکه‌های اجتماعی، یا شبکه‌ای از جاده‌ها یا نورون‌ها یا هر مجموعه‌ی دیگری از اتصالات و کانکشن‌ها. جست و جوی عرضی (فصل ۶) و الگوریتم داجسترا (فصل ۷) راهیای برای پیدا کردن کوتاه‌ترین مسافت بین دو نقطه در شبکه هستند. میتوانید از این رویکرد برای محاسبه‌ی درجه جدایی بین دو فرد یا پیدا کردن کوتاه‌ترین مسیر به مقصدی معین استفاده کنید.

نزدینک ترین همسایه k (K nearest neighbor) - این مبحث در فصل ۱۰ پوشش داده شده. این الگوریتم، یک الگوریتم ساده حوزه پادگیری ماسیح است. شما میتوانید این الگوریتم برای ساخت سیستم پیشنهاد دهنده ، موتور OCR ، سیستم پیش‌بینی ارزش سهام (یا هرچیزی که شامل پیش‌بینی کردن یک مقدار باشد) یا دسته‌بندی کردن یک شی استفاده کنید.

قدم بعدی-در فصل ۱۱ ، نگاهی گذرا می‌اندازیم به ۱۰ تا از الگوریتم‌هایی که بعد از خوندن این کتاب برید سراغشون تا بعد از برخورد با اونها مشکلی نداشته باشید.-۵. wrong

چطور از این کتاب استفاده کنیم

ترتیب و محتوای این کتاب خیلی با دقت طراحی و چیده شده. اگر دیدید به مبحث خاصی علاقه دارید با خیال راحت برین سراغ اون مبحث. در غیر این صورت فصل هارو به ترتیب پیش برید چرا که بر پایه همدیگر ساخته شده اند.

بسدت توصیه میکنم کدهای هر مثال رو خودتون اجرا کنید. من نمیتوانم به اندازه کافی روی این قسمت تاکید کنم. فقط لازمه نمومه کدهای من رو کلمه به کلمه تایپ کنید (یا از یکی از این دو لینک دانلود کنید:

<https://www.manning.com/books/grokking-algorithms.1>

(https://github.com/egonschiele/grokking_algorithms.2)

و اونها رو اجرا کنید. اگر این کارو بکنید چیز‌های خیلی بیشتری یادمیگیرید.

همچنین پیشنهاد مکنم تمرین‌ها را هم انجام بدهید. تمرین‌ها خیلی کوتاه هستند معمولا ۱ تا ۲ دقیقه ، بعضی اوقات ۵ تا ۱۰ دقیقه نهایتا وقت شما را بگیرد. اونها باعث میشوند اشتباها تنان را پیدا کنید و بررسی کنید تا قبل از اینکه دیر بشود از مسیر اصلی خارج نشوابد.

این کتاب مناسب چه کسانیست؟

این کتاب کسانی را هدف گرفته که با پایه های کد زدن آشنا هستند و میخواهند الگوریتم ها را درک کنند. شاید ممکن شما از قبل مشکلی در کد زدن خود داشته باشید و در تلاش برای پیدا کردن راه حل الگوریتمیک برای آن میگردید. یا شاید میخواهیں متوجه بشین که اصلاً چرا الگوریتم ها بدرد میخورند. این زیر هم یه لیست کوتاه و ناکامل از کسانی که ممکن است این کتاب برای خودشان مفید بدانند:

کسانی که برای سرگرمی کد میزنند یا کدنویسی سرگرمی آنهاست

دانشجو های بوت کمپ

فارغ التحصیلان علوم کامپیوتر که میخوان خودشان را بروز کنند

فارغ التحصیلان فیزیک / ریاضی / دیگر رشته ها که به برنامه نویسی علاقه دارند

نحوه نوشتن کد در کتاب و نحوه دانلود آنها

در همه نمونه کد ها در این کتاب از پایتون ۲.۷ استفاده شده. همه ی کد ها در این کتاب به شکل زیر نوشته میشوند:

```
Print("Hello world!")
```

و در صورت لزوم توضیحات در کنار کد ها درج میشود و مفاهیم مهم هایلایت میشوند

از لینک های زیر هم میتوانید نموده کد هارا دانلود کنید.

<https://www.manning.com/books/grokking-algorithms>.۱

https://github.com/egonschiele/grokking_algorithms.۲

من معتقدم تنها زمانی میتوانید بهترین بهره را از این کتاب ببیرید که از یادگیری لذت ببرید

پس کلی خوش بگذرونین و اجرا کردن کد ها فراموش نشه!

درباره نویسنده

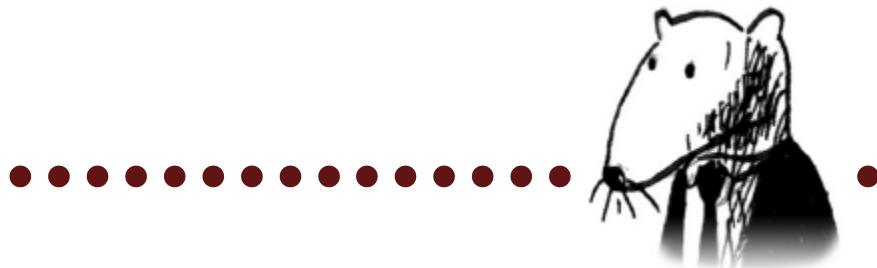
درباره نویسنده کتاب

آدیتیا بھارگاو (Aditya Bhargava) مهندس نرم افزار شرکت اتزا (Etsy) است. یک بازار آنلاین برای کالا های دست ساز. او دارای مدرک کارشناسی ارشد در رشته علوم کامپیوتر از دانشگاه شیکاگو است. همچنین او بلاگ مصور محبوبی که در حوزه تکنولوژی در adit.io هست را مدیرت میکند.

مقدمه ای بر

الگوریتم ها

۱



در این فصل:

- با پایه کار برای خواندن بقیه کتاب آشنا میشوید
- شما اولین الگوریتم جست و جوی خودتون رو مینویسید
- شما یادمیگیرین چطور در بازه زمان اجرا شدن یک الگوریتم صحبت کنید (**Big o notation**)
- با یک تکنیک رایج برای طراحی الگوریتم آشنا میشین (بازگشت)



مقدمه

الگوریتم مجموعه ای از دستورالعمل ها برای به انجام رساندن یک کار است. هر تکه کد میتواند یک الگوریتم باشد اما این کتاب بخش هایی را که بیشتر از بقیه جالب تر است را پوشش میدهد. من الگوریتم هایی که در این کتاب انتخاب کردم و گنجانده ام را یا بدلیل سریع بودن آنها (که باعث میشن وقت کمتری ازما گرفته شود) انتخاب کردم یا بدلیل اینکه متوانیم مسئله های جالبی را با کمک آنها حل کنیم یا هردو. اینجا هم برخی از نکات قابل توجه را داریم:

(مترجم: از اینکه دارم رسمی مینویسم خسته شدم و بنظرم خسته کننده میاد چون خود کتاب خیلی دوستانه داره بحث رو پیش میبره پس منم دوستانش میکنم!)

- خب فصل یک درباره باینر سرج حرف میزنه (همون جست و جوی دودویی) و نشون میده که چطور یک الگوریتم میتونه سرعت کد تون رو بالا ببره. بزارین یه مثال بزنم. با این الگوریتم میتوانیم تعداد گام ها برای بررسی یک داده رو که شامل ۴ میلیارد گام میشه رو تنها به ۳۲ گام برسونیم!
- دستگاه **GPS** از الگوریتم های گراف (که در فصل های ۶ و ۷ و ۸ یادشون میگیرین) استفاده میکنه تا کوتاه ترین مسیر رو به سمت مقصد برای شما محاسبه و پیدا کنه
- شما میتوانین از برنامه نویسی پویا (که در فصل ۹ راجبsh بحث میشه) برای ساخت الگوریتم هوش مصنوعی استفاده کنین تا بتونه چکرزا (یه نوع بازی تخته ای) بازی کنه

من الگوریتم رو در هر کدوم از حالت ها برآتون توضیح میدم و برای هر کدومشون هم یه مثال برآتون میزنم. بعدش درباره زمان اجرایی اون الگوریتم با استفتدن از بیگ او نویشن صحبت میکنم که ببینم این الگوریتم چقدر سریع یا کند عمل میکنه. درنهایت هم بررسی میکنم که ببینیم چه مسئله های دیگری رو میتوانیم با همون الگوریتم حل کنیم.

چیزهایی که درباره عملکرد یاد خواهید گرفت

خبر خوب اینه که نحوه چگونگی بکار گیری هر کدوم از این الگوریتم ها به احتمال زیاد در زبان برنامه نویسی مورد علاقه شما در دسترسه. پس شما نیاز ندارید هر الگوریتم رو خودتون پیاده سازی کنید. ولی خب اگر همینطوری شروع به نوشتمن کد کنین و کد هارو نفهمید عملا کار بیهوده ای انجام دادین. در این کتاب شما یادمیگیرن که چطور الگوریتم های مختلف رو باهم مقایسه و سبک سنگین کنید:

آیا باید از مرتب سازی ادغامی (**merge sort**) (ازاین به بعد میگیم مرج سورت)) استفاده کنم یا از مرتب سازی سریع (**quicksort**) (ازاین به بعد میگیم کوئیک سورت))؟

از آرایه استفاده کنم بهتره یا از لیست ها؟

فقط با استفاده از یک ساختار داده متفاوت میتوانیم یک تغییر خیلی بزرگ ایجاد کنیم.

چیزهایی که درباره حل مسئله یاد خواهید گرفت

شما تکنیک های حل مسئله ای رو یادمیگیرین که ممکنه تا حالا فکرتون بهش نرسیده باشه. به عنوان مثال:

- اگر به ساخت بازی های ویدیویی علاقه دارین، میتوانیم یک هوش مصنوعی بنویسیم که کاربر رو با استفاده از الگوریتم های مربوط به گراف دنبال کند
- شما یادمیگیرین که یک سیستم پیشنهاد دهنده با استفاده از "k همسایه نزدیک" درست کنید
- بعضی از مسئله از لحاظ مقدار زمانی که برای حل طول میکشه قابل حل نیستند.(مثلًا مقدار زمانی تقریبی که طول میکشه تا حل بشه ۳۰۰ ساله که خب میگیم غیرقابل حلن). بخشی از این کتاب که درمورد مسئله "NP کامل" ه بهتون نشون میده که چطور این مسئله هارو شناسایی کنیم و با الگوریتم هایی پیش برویم که به یک جواب تقریبی بررسید.

به طور کلی، در پایان این کتاب شما برخی از پرکاربردترین الگوریتم ها که درسطح گسترده ای استفاده میشوند رو میشناسید. بعد از این شما میتوانیم با دانش جدیدی که کسب کردیم چیز های بیشتری درباره الگوریتم های بخصوصی در حوزه هوش مصنوعی، دیتا بیس، و ... یادبگیریم. یا میتوانیم به مصاف چالش های بزرگتری در حیطه کاری خودتون ببریم.

چیزایی که لازمه قبل از مطالعه بدونیم:

شما برای شروع این کتاب به پایه های جبر نیاز دارین. به خصوص این قسمت:

$$f(x) = x^2$$

این تابع رو درنظر بگیرین:

اگر جوابتون ۱۰ بود همه چیز اوکیه میتوانیم به خوندن کتاب

طبق تابع جواب **f(5)** چی میشه؟
ادامه بدین.

علاوه براین، دنبال کردن این فصل (و درکل، این کتاب)، اگر با یک زبان برنامه نویسی آشنایی باشید خیلی ساده میشے. تمام مثال های این کتاب با زبان پایتون پیاده سازی شده. اگر با زبان برنامه نویسی خاصی آشنایی ندارید و میخواین شرو به یادگیری یک زبان بکنید، پایتون رو انتخاب کنید این زبان برای تازه کارها عالیه.اما اگر زبان برنامه نویسی دیگه مثل **Ruby** رو بلدین.هیچ مشکلی پیش نمیاد.

باینری سرچ

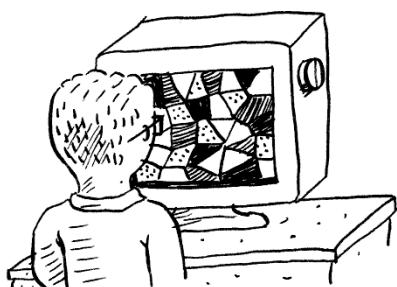
فرض کنید در دفترچه تلفن دنبال فردی میگردید (چه سبک قدیمی داشت این جمله!) اسم اون فرد با **k** شروع میشے. خب شما میتونین از اول دفترچه تلفن رو باز کنین و صفحه به صفحه ورق بزنین و بگردین تا به بخش **K** ها برسید. ولی خب احتمال زیاد این کارو نمیکنین و مستقیم میرین و صفحه ها وسط کتاب رو باز میکنین چون میدونین بخش **k** ها به وسط نزدیک ترن.



یا فرض کنین شما دارین دنبال یک کلمه در دیکشنری میگردین و اون کله با **0** شروع میشے. شما دوباره کتاب رو از وسط باز میکنین چون میدونین که **0** ها به وسط نزدیک ترن.

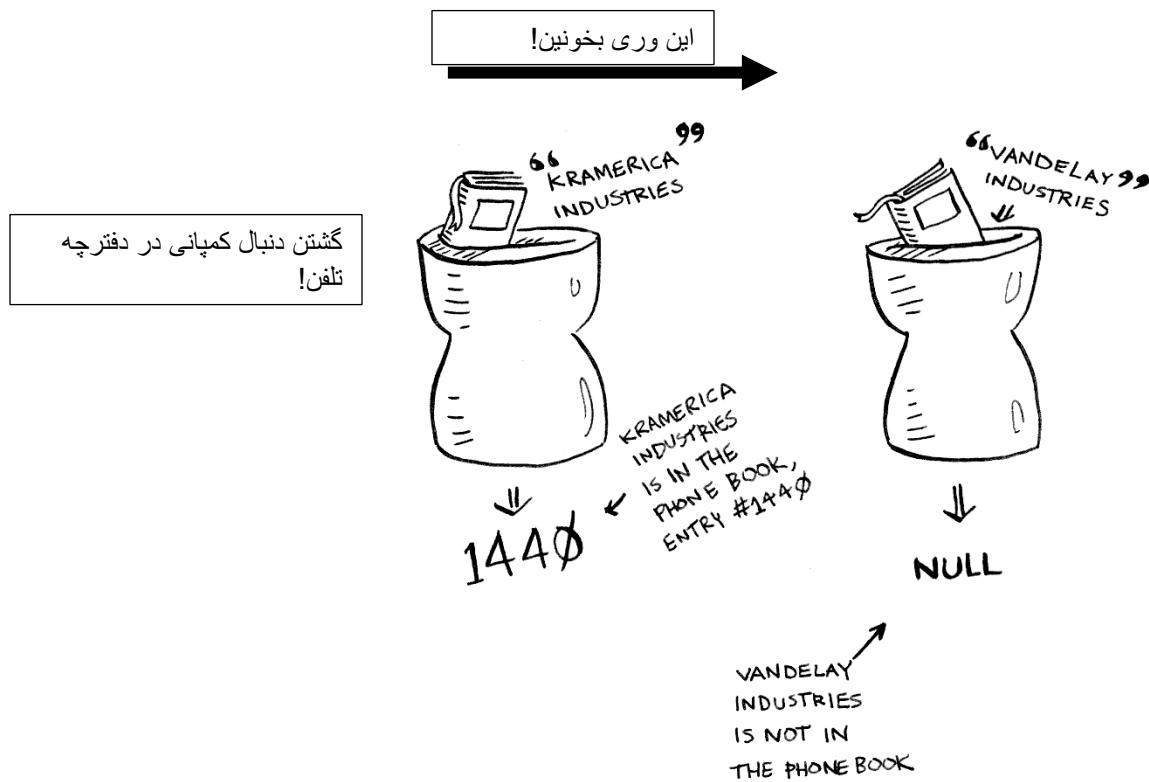
حالا تصور کنین که میخواین وارد فیس بوک بشین. وقتی شما این کارو نمیکنین، فیس بوک باید چک کنه که آیا شما اکانتی روی سایتشون داری یا نه. خب پس نیاز داره که بره و نام کاربری که شما وارد کردین رو در دیتابیس خودش چک کنه که ببینه آیا همچین نام کاربری هست یا نه. فرض کنین نام کاربری شما **Karlmageddon**. فیس بوک میتونه از اول و بالا شروع کنه گشتن به دنبال اسم شما ولی خب منطقی تر اینه که از یه جایی نزدیک وسط شروع کنه به گشتن.

این مشکل سرچی (**search**) هست که ما داریم. و همه این حالت ها از یک الگوریتم برای حل این مشکل استفاده میکنن که اون: **binary search**



باینری سرچ یه الگوریتمه. ورودی های این الگوریتم یک لیستی از عناصر مرتب شدss (بعدا بهتون توضیح میدم که چرا نیازه که مرتب شده باشه!). اگر عنصری که شما به دنبالشین در لیستش باشه، باینری سرچ موقعیت جایی که اون عنصر قرار داره رو برآتون برمیگردونه. در غیر اینصورت اگر اون عنصر موجود نباشه مقدار **null** رو برمیگردونه.

به عنوان مثال:

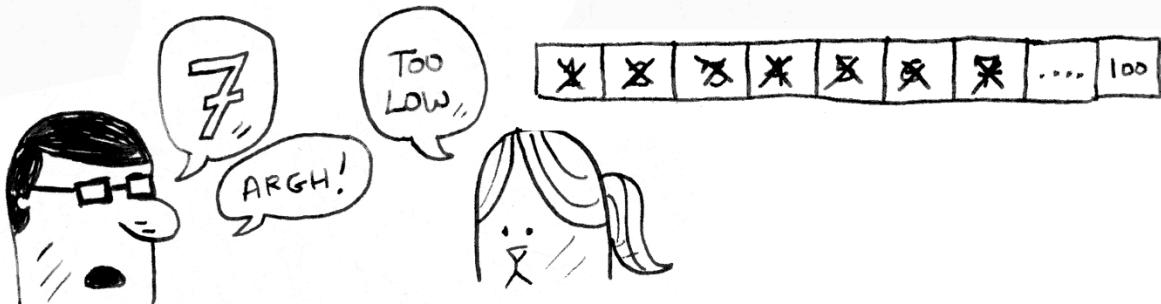


بریم یه مثال بزنیم که ببینیم باینری سرچ ها چطور کارمیکنن:

من دارم به یک عدد بین ۱ تا ۱۰۰ فک میکنم

1	2	3	...	100
---	---	---	-----	-----

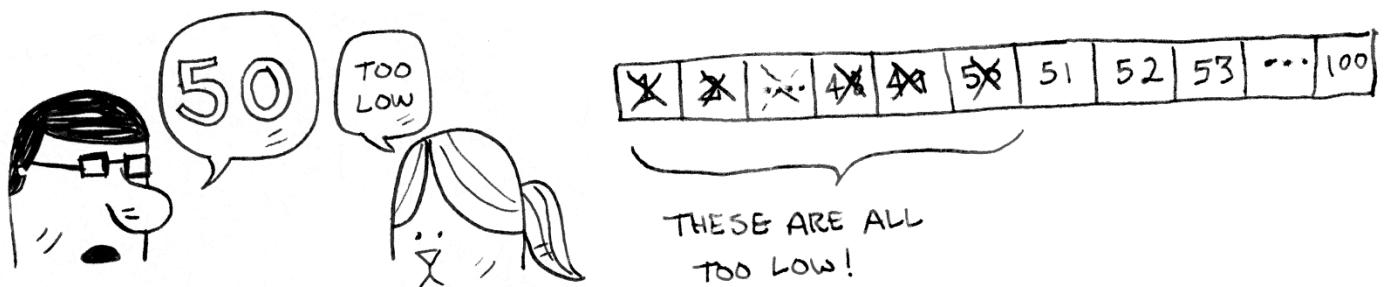
شما باید با کمترین تعداد تلاش ممکن حدس بزنین که عددی که من انتخاب کردم چیه. هر بار که عددی رو حدس میزنین من بهتون میگم حدس شما "خیلی بالاς"، "خیلی پایینه" یا "درسته". حالا فرض کنین شما اینطوری شروع به حدس زدن میکنین: ۱ و ۲ و ۳ و ۴ و..... که باعث میشه اینطوری پیش بره:



این همون جست و جوی سادس.(شایدم جستوجو احمقانه اسم بهتری باشه براش). با هر بار حدس زدن شما تنها یک عدد رو از احتمالات حذف میکنین. اگر عددی که من انتخاب کردم ۹۹ بود شما باید ۹۹ بار حدس میزدین تا به عدد من برسین

یک راه بهتر برای جست و جو

این یک تکنیک بهتره. با ۵۰ شروع کنید



خیلی پایینه، اما شما نصف عدد ها (احتمالات رو حذف کردید)! الان شما میدونید که اعداد ۱ تا ۵۰ خیلی پایین هستن. عدد بعدی که حدس میزنیم : ۷۵

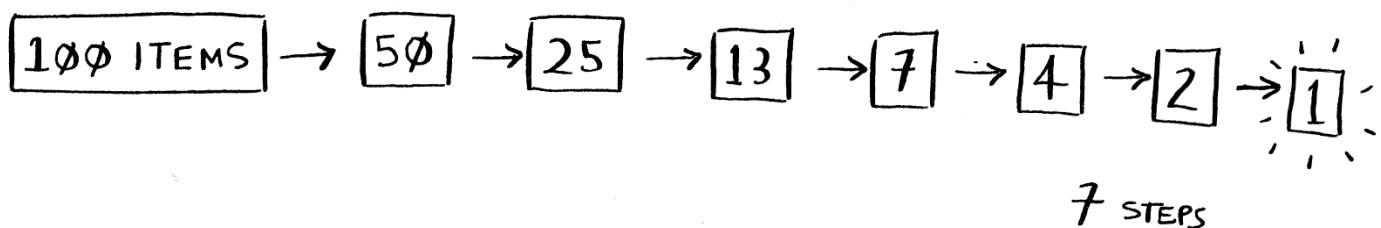


خیلی بالا، اما دوباره شما نصف اعداد باقی مونده از حدس قبل رو حذف کردید! با باینری سرج هر دفعه شما عدد وسط رو حدس میزنین و نصف احتمالات رو حذف میکنین. در حدس بعدی با ۶۳ پیش میریم (عدد بین ۵۰ و ۷۵)



به این میگن باینری سرج. تبریک میگم شما اولین الگوریتمتون رو یادگرفتید.

اینجا هم میتوونین بینین هر دفعه چند عدد رو حذف کردین.



هر عددی که من بهش (توى اين بازه) فک کنم، شما میتوونین نهايتا با ۷ حدس بهش برسين به اين دليل که شما تعداد زيادي از اعداد رو با هر حدس از بين ميبرين.

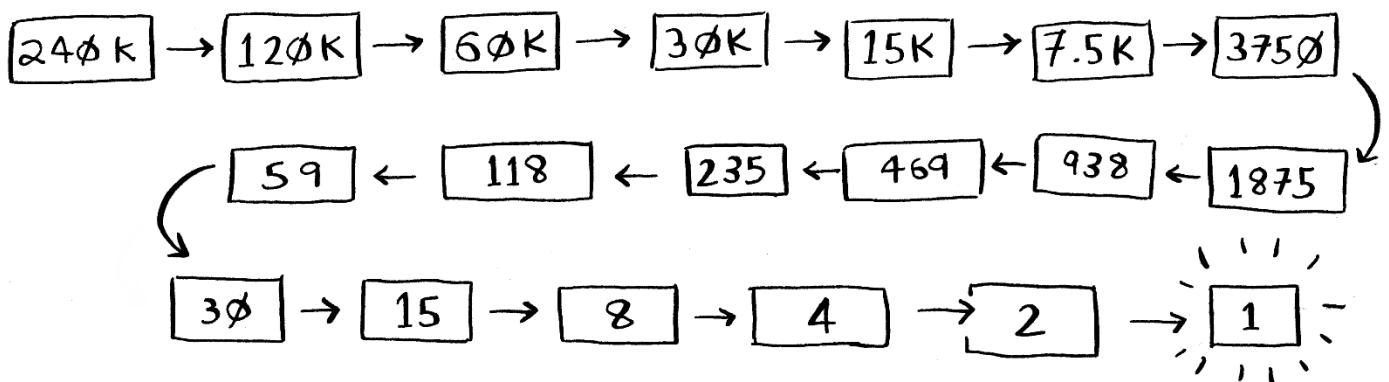
فرض کنید که شما به دنبال کلمه ای توی دیکشنری میگردین. دیکشنری خودش ۲۴۰۰۰ کلمه داره. در بدترین حالت چند گام طول میکشه که برداریم تا به کلمه مورد نظر برسیم؟

SIMPLE SEARCH: _____ STEPS

BINARY SEARCH: _____ STEPS

برای جستجوی ساده میتونه ۲۴۰۰۰ گام طول بکشه اگر که کلمه ای که شما دنبالشین دقیقا آخرین کلمه دیکشنری باشه.

با هر گام در باینری سرچ شما هر بار نصف باقی مانده احتمالات رو حذف میکنین تا اینکه یک کلمه برآتون باقی بموونه.



18 STEPS

پس با این حساب باینری سرچ فقط ۱۸ گام نیاز داره تا به کلمه موردنظر برسه. یه اخلاف فاحش. به طور کلی برای یک لیست از n مقدار، باینری سرچ تنها $\log_2 n$ گام طول میکشد تا بدترین حالت را اجرا کند در حالی که جست و جوی معمولی n گام طول میکشد.

لوگاریتم

شاید شما یادتون نیاد که لگاریتم ها چی هستن، اما احتمالا میدونید که نمایی ها چی هستن. لگاریتم زیر:

$$\log_{10} 100$$

مثل این میمونه که بپرسیم، "چنتا ۱۰ باید درهم ضرب بشن تا به ۱۰۰ برسیم؟" که جوابش میشه ۲.

$$10 \times 10 = 2$$

پس:

$$\log_{10} 100 = 2$$

لوگاریتم ها یک جوابی برعکس نمایی هان.

$$\begin{array}{c} 10^2 = 100 \leftrightarrow \log_{10} 100 = 2 \\ \hline 10^3 = 1000 \leftrightarrow \log_{10} 1000 = 3 \\ \hline 2^3 = 8 \leftrightarrow \log_2 8 = 3 \\ \hline 2^4 = 16 \leftrightarrow \log_2 16 = 4 \\ \hline 2^5 = 32 \leftrightarrow \log_2 32 = 5 \end{array}$$

در این کتاب وقتی راجب زمان اجرایی در بیگ او تویشن (که یکم دیگه بهش میرسیم)، این **log** یعنی:

$$\log_2$$

وقتی برای جست و جوی یک عنصر از جستجوی ساده استفاده میکنیم، در بدترین حالت شما باید تک تک عناصر را بررسی کنید. پس برای یک لیست که شامل ۸ عدد است شما باید حداقل ۸ تا عدد را چک کنید. در حالی برای باینری سرچ شما در بدترین حالت باید $\log n$ را چک کنید. برای یک لیست شامل ۸ عنصر، $3 = \log 8$ که

$= 2^3$. پس برای لیست ۸ تایی شما تنها باید ۳ عدد را چک کنید. برای لیستی که شامل 1024 عنصر است:

$\log 1024 = 10$ به این دلیل که $1024 = 2^{10}$. پس برای یک لیست 1024 عددی شما باید تنها ۱۰ عدد را بررسی کنید.

یادتون باشے!

من در این کتاب درباره لوگاریتم زیاد بحث میکنم. پس شما بهتره که مفهوم لگاریتم رو در ک کنید. اگر هنوز مشکلی دارید. **khanacademy** یک ویدیوی خوب داره که میتوانه همه چیز رو برآتو واضح کنه.

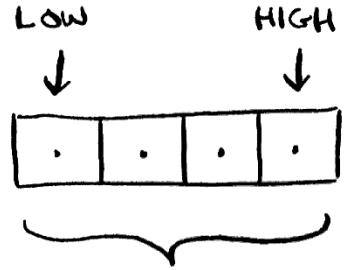
یادتون باشے!

باینری سرج تنها زمانی کار میکنه که لیست شما به ترتیب و مرتب شده باشه. برای مثال، اسم ها در دفترچه تلفن به ترتیب الفا چیده شدند، برای همینه که میتوانیں از باینری سرج استفاده کنین. چه اتفاقی می افتاد اگر اسم ها به ترتیب نبودند؟

بریم ببینیم چطور میتوانیم باینری سرج رو توی پایتون پیاده کنیم. توی نمومه کد از آرایه ها برای اینکار استفاده شده. اگر نمیدونید آرایه ها چطور کار میکنند نگران نباشید اونها در فصل بعدی پوشش داده شده اند. شما فقط نیازه اینو بدونین که میتوانین دنباله ای از عناصر رو در سطر مکان سطر ها متوالی که بهش میگن آرایه ذخیره کنین. این مکان ها شماره گذاری شده اند و از صفر شروع میشوند. اولین مکان در موقعیت ۰، دومین در موقعیت ۱، سومین در موقعیت ۲ و همینطوری تا آخر.

تابع `binary_search` یک آرایه ی مرتب شده و یک آیتم(چیزی که قراره تو اون آرایه دنبالش بگردیم) را میگیرد. اگر اون آیتم در آرایه وجود داشته باشد، تابع موقعیت اون رو در آرایه برمیگردونه. شما حواستون خواهد بود که کدوم بخش از آرایه باید بررسی بشه. در شروع کار ما کل آرایه رو در نظر میگیریم.

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



THESE ARE ALL THE
NUMBERS WE ARE
SEARCHING THROUGH

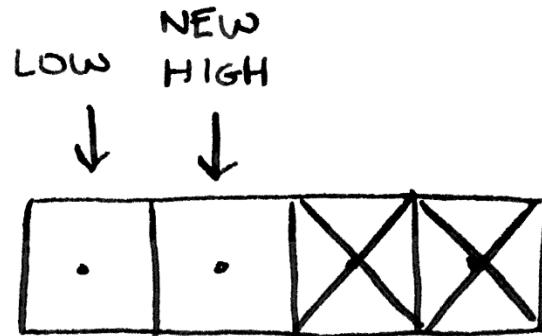
هربار شما عنصر وسط رو چک میکنین:

```
mid = (low + high) / 2
guess = list[mid]
```

اگر عدد زوج نباشه تو سط پایتون رو
به پایین گرد میشه.

اگر حدستون خیلی پایین باشه، برو طبق اون مقدار `LOW` رو تغییر میدین:

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



و اگر حدس خیلی بالا بود، مقدار `high` را آپدیت میکنیم. این کد کاملشه:

```

def binary_search(list, item):
    low = 0           low and high keep track of which
    high = len(list)-1          part of the list you'll search in.

    while low <= high:         While you haven't narrowed it down
        mid = (low + high) // 2 to one element ...
        guess = list[mid]       ... check the middle element.

        if guess == item:      Found the item.
            return mid
        if guess > item:       The guess was too high.
            high = mid - 1
        else:                  The guess was too low.
            low = mid + 1
    return None               The item doesn't exist.

my_list = [1, 3, 5, 7, 9]   Let's test it!

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

```

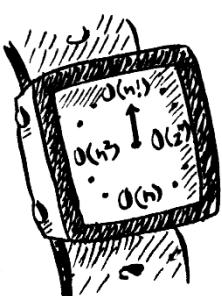
Remember, lists start at 0.
The second slot has index 1.
"None" means nil in Python. It indicates that the item wasn't found.

تمرین

- ۱/۱ فرض کنید شما یه لیست مرتب شده از ۱۲۸ اسم دارید، و شما با استفاده از باینری سرچ در حال جستجو جو کردن در داخل اون هستین. تعداد حداقل گام هایی که طول میکشد چقدر میشه؟
- ۱/۲ فرض کنید مقدار همون لیست رو دو برابر کنیم حالا حداقل چند گام بررسی اون طول میکشه؟

زمان اجرا

هر موقع راجب یک الگوریتم صحبت میکنم در باره زمان اجرایی اون هم بحث خواهم کرد. به طور کلی شما میخواهید موثرترین الگوریتم را چه از نظر زمانی یا چه از نظر مقدار فضای موردنیاز (مموری) انتخاب کنید.



بیان برگردیم به باینری سرچ. با استفاده از باینری سرچ چه مقدار زمان ذخیره میکنید؟ خب، اولین رویکرد اینه که تمام اعداد رو یکی یکی چک کنیم. اگر لیستمون شامل ۱۰۰ عدد باشد، تا ۱۰۰ حدس طول میکشد. اگر یک لیست شامل ۴ میلیارد عدد باشد تا ۴

میلیارد حدس طول میکشد. پس حداکثر تعداد حدس ها با اندازه لیست برابر است. به این میگن "زمان خطی".
 باینری سرچ متفاوت است. اگر طول لیست به اندازه ۱۰۰ آیتم باشد نهایتا ۷ حدس طول میکشد. اگر ۴ میلیارد آیتم
 باشد حداکثر ۳۲ حدس طول میکشد. قدر تمدن، نه؟ باینری سرچ در "زمان لوگاریتمی" اجرا میشود.
 این هم جدولی که یافته های امروز ما را خلاصه میکند.

SIMPLE SEARCH	BINARY SEARCH
100 ITEMS ↓ 100 GUESSES	100 ITEMS ↓ 7 GUESSES
4,000,000,000 ITEMS ↓ 4,000,000,000 GUESSES	4,000,000,000 ITEMS ↓ 32 GUESSES
$O(n)$	$O(\log n)$

LINEAR TIME LOGARITHMIC TIME

بیگ او نو تیشن

بیگ او نو تیشن یک نماد خاصه که بهتون میگه یه الگوریتم چقدر سریعه. کی اهمیت میده؟ خب، چیزی که مشخصه شما بعضی اوقات از الگوریتم های بقیه افراد استفاده خواهد کرد و وقتی که اینکارو انجام بدین خوبه که بدونین اون الگوریتم ها چقد سریع یا کند هستند. توی این بخش من بهتون میگم بیگ او نو تیشن چیه و لیستی از رایج ترین زمان های اجرایی برای الگوریتم ها که با استفاده از همین بیگ او نو تیشن بدست اومدن بهتون ارائه میدهم.

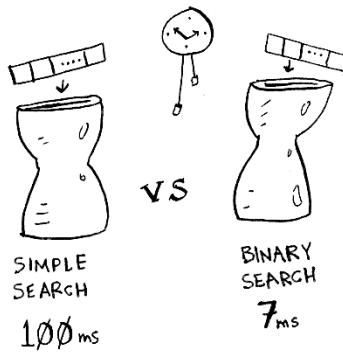
زمان اجرایی الگوریتم ها به میزان نرخ ورودی های متفاوت افزایش پیدا میکند

باب در حال نوشتن یک الگوریتم جستوجوی برای ناسا است. الگوریتم اون وقتی وارد کار میشه که موشک میخواهد روی سطح ماه فرود بیاد و این الگوریتم به موشک کمک میکنه که محاسبه کنه کجا فرو بیاد.



این یک مثال از اینکه چطور زمان اجرایی دو الگوریتم میتونه با نرخ های مختلف افزایش پیدا کنه. باب داره تلاش میکنه که از بین جستوجوی ساده و باینری سرج(جستوجوی دودویی) یکیو انتخاب کنه. الگوریتم انتخابی هم باید سریع و هم درست و دقیق باشه. از طرفی باینری سرج سریعه و باب هم فقط ۱۰ ثانیه زمان داره که بفهمه کجا فرود باید صورت بگیره، درغیر این صورت موشک از مسیر خودش خارج میشه (و این بدنه!). از طرفی هم پیاده کردن جستوجوی ساده خیلی آسون تره و احتمالش خیلی کمتره که باگی توش پیدا بشه. و باب واقعا دلش نمیخواهد که توی کدی که قرار یک موشک رو فرود بیاره باگی وجود داشه باشه. برای اینکه کار خیلی دقیق بیش بره باب تصمیم میگیره که زمان اجرایی دو الگوریتم رو با یک لیست ۱۰۰ تایی از عناصر اندازه گیری کنه.

بیاین فرض کنیم که هر عنصر ۱ میلی ثانیه طول میکشه تا بررسی بشه. با جستوجوی ساده، باب مجبوره که هر ۱۰۰ عنصر رو بررسی کنه پس ۱۰۰ میلی ثانیه طول میکشه تا این الگوریتم اجرا بشه. از طرف دیگه اون فقط باید ۷ تا عناصر با باینری سرج بررسی کنه ($\log_2 100$ تقریبا میشه ۷)، پس جستجو با این روش ۷ میلی ثانیه طول میکشه تا اجرا. ولی اگر به صورت واقع بینانه بخوایم بررسی کنیم، لیستی که قرار بررسی بشه یک چیزی در حدود ۱ میلیارد عنصر داره. اگر اینطوریه پس استفاده از جستوجوی ساده چقدر وقت میگیره؟ باینری سرج چطور؟ قبل از خوندن بقیه این بخش مطمئن بشین که برای این سوالات یک جوابی دارین.



زمان اجرایی برای سرچ ساده (جستجوی ساده) در برابر باینری سرچ با ۱۰۰ عنصر

باب این دفعه باینری سرچ رو با ۱ میلیارد عنصر اجرا میکند و ۳۰ میلی ثانیه طول میکشه. $\log_2 1000000000 = 30$. اون پیش خودش فک میکنه:

"۳۲ میلی ثانیه!". در ادامه اینطوری نیجیه گیری میکنه که: "باینری سرچ فقط ۱۵ برابر سریعتر از جستجوی سادس، چون با ۱۰۰ تا عنصر، ۱۰۰ میلی ثانیه طول کشید و باینری سرچ ۷ میلی ثانیه طول کشید. پس سرچ ساده برای ۱ میلیارد هم $30 \times 15 = 450$ میلی ثانیه طول میکشه، درسته؟ خیلی کمتر از آستانه ۱۰ ثانیه ای که من نیاز دارم." باب تصمیم میگیره که جستجوی ساده پیش بره. اما آیا این درسته؟

معلومه که نه! اینطوری که معلومه باب داره اشتباه میزنه. بدم داره اشتباه میزنم. زمان اجرایی جستجوی ساده برای ۱ میلیارد آیتم، ۱ میلیارد ثانیه س، که ینی ۱۱ روز! مشکل اینجاست که زمان اجرایی باینری سرچ و سرچ ساده یا یک نرخ مشخص افزایش پیدا نمیکنن.

SIMPLE SEARCH	BINARY SEARCH
100 ELEMENTS	100ms
10,000 ELEMENTS	10 seconds
1,000,000 ELEMENTS	11 days

اینم از این، هرچقدر آیتم ها برای بررسی بیشتر میشن، باینری سرچ فقط یک کوچولو زمان بیشتری برای اجرا شدن میبره اما سرچ ساده زمان خیلی بیشتری میبره تا اجرا بشه. پس میتونیم بفهمیم که هر چی اعداد بزرگتر میشن باینری سرچ به طور ناگهانی سریع تر از سرچ ساده عمل میکنه. باب فکر میگرد که باینری سرچ ۱۵ برابر سریع تر از سرچ سادس اما این درست نیست. اگر لیستمون شامل ۱ میلیارد آیتم باشد باینری سرچ یک چیزی حدود ۳۳ میلیون برابر سریع تر از سرچ سادس. به همین دلیله که دونستن اینکه یک الگوریتم دریک شرایط خاص چقد زمان اجرا شدنش طول میکشه کافی نیست و باید بدونین که زمان اجرایی چطور با افزایش اندازه لیست افزایش پیدا میکند. این دقیقا جایی که بیگ او نویشن وارد عمل میشه.

بیگ او نوتبشن بهتون میگه که یه الگوریتم چقدر سریعه. به عنوان مثال، شما لیستی با سایز n دارین. سرچ ساده نیاز داره که هر کدوم از عناصر رو چک کنه. پس n بار این عملیات طول میکشه. زمان اجراییش توی بیگ او نوتبشن اینطوری میشه: $O(n)$. پس تعداد ثانیه کجاست؟ هیچ جا. بیگ او سرعت یک الگوریتم رو بر مبنای ثانیه نمیگه. بیگ او نوتبشن بهتون این اجازه رو میده که تعداد عملیات هارو باهم مقایسه کنید. بیگ او بهتون میگه یک الگوریتم با چه سرعتی افزایش پیدا میکنه.



(مترجم: این ویدیو توی درک بیگ او نوتبشن خیلی میتونه کمکتون کنه:

<https://youtu.be/vSYihb4rcw>

بریم سراغ یه مثال دیگه. باینری سرچ به تعداد $\log n$ عملیات نیاز دارد تا لیستی با اندازه n را بررسی کند.

حالا زمان اجراییش بر اساس بیگ او نوتبشن چی میشه؟

$O(\log n)$

به طور کلی بیگ او نوتبشن به صورت زیر نوشته میشود.

$O(n)$
 ↗ "BIG"
 ↘ NUMBER OF
 OPERATIONS

بیگ او به شما تعداد عملیات هایی که یک الگوریتم برای اجرا شدن نیاز دارد را میگوید. دلیل اینکه بهش میگن بیگ او نویشن (که ترجمه تحت الفظیش میشه نماد او بزرگ) اینکه که شما پشت تعداد عملیات های یک الگوریتم یک ۰ بزرگ میدارین. (به نظر مسخره میاد ولی واقعاً اینطوریه)

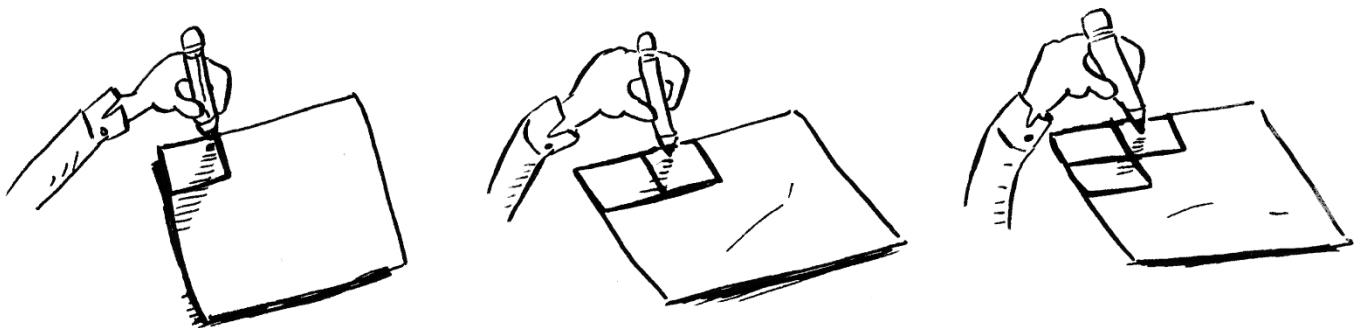
حالا بیاین یک نگاهی به چنتا مثال بندازیم تا ببینید میتوانید زمان اجرایی برای اون الگوریتم هارو پیدا کنید؟

به تصویر کشیدن زمان اجرایی بیگ او های مختلف

در اینجا یک مثال عملی داریم که شما هم میتوانید در خونه با یک تکه کاغذ و یک مداد این مثال رو دنبال کنید.
فرض کنید شما باید یک جدول ۱۶ خونه ای بکشید.

الگوریتم اول

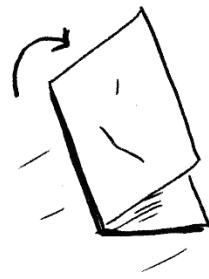
یک راه اینه که یکی یکی ۱۶ تا خونه بکشید. اگر یادتون باشه بیگ او نویشن تعداد عملیات هارو میشماره. توی این مثال، کشیدن یک خونه برابر یک عملیاته. خب شما باید ۱۶ خونه بکشید. چند عملیات در هربار کشیدن جعبه طول میکشه؟



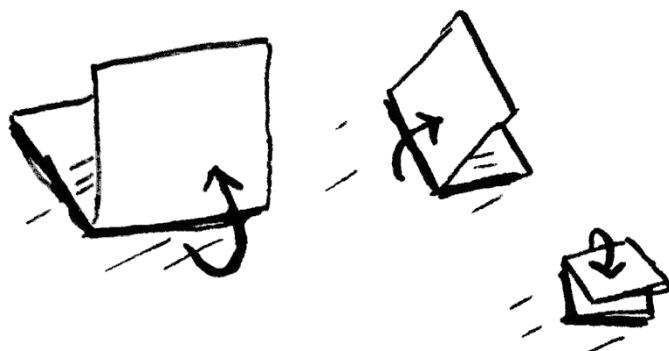
۱۶ بار طول میکشه تا ما ۱۶ خونه بکشیم. خب زمان اجرایی این الگوریتم چقدره؟

الگوریتم دوم

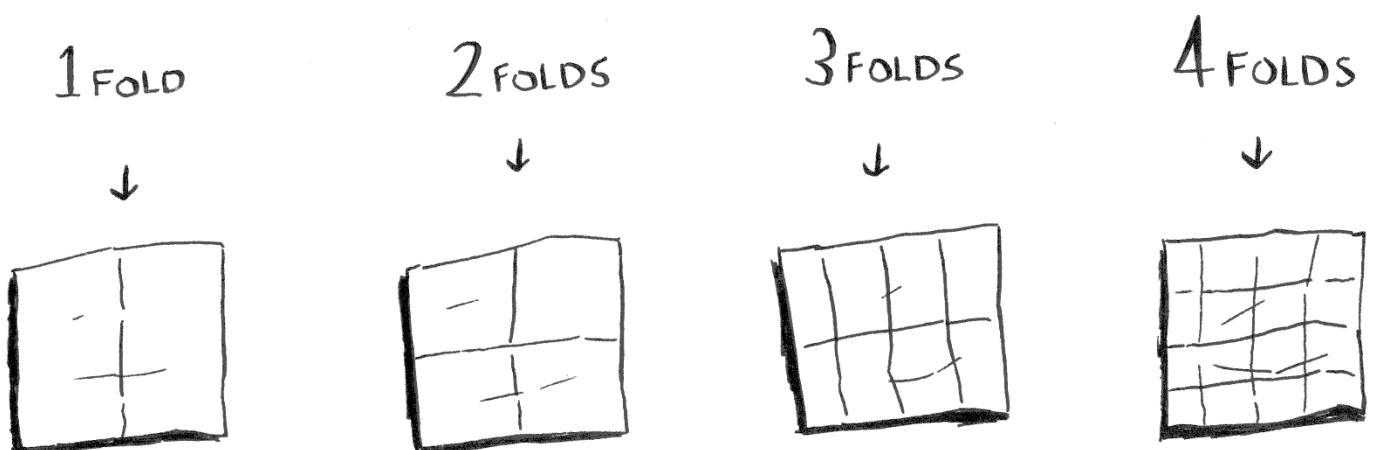
بیاین بجای اون از اینیکی الگوریتم استفاده کنیم. کاغذ رو تا بزنید.



توى اين مثال هر بار تا زدن كاغذ يك عمليات محسوب ميشه. شما يا همون يك عمليات دو تا خونه ايجاد كردید.
كاغذ رو دوباره تاکنيد. دوباره و دوباره.



بعد از اينكه چهار بار كاغذ رو تاکرديد. كامل از تا شدن درش بيارين و بازش کنيں و بعدش شما يه جدول زيبا خواهيد داشت! هر بار تا زدن تعداد خونه هارو دو برابر ميکنه. درنتيجه شما ۱۶ خونه با ۴ تا عمليات درست كردید.



شما ميتونين با هر بار تا زدن دو برابر مقدار قبل خونه بکشيد (درست كنيد). پس شما ميتونين ۱۶ خونه رو تنها در ۴ گام درست كنيد. زمان اجرایي اينيکي الگوريتم چقدر؟

بیاین قبل از ادامه دادن زمان های اجرایی هر الگوریتم رو مشخص کنیم.

جواب: الگوریتم اول به اندازه $O(n)$ طول میکشه، و الگوریتم دوم به اندازه $O(\log n)$ طول میکشه.

بیگ او بدترین حالت زمان اجرایی را نشان میدهد

فرض کنید شما برای پیدا کردن یک فرد در دفترچه تلفن از سرچ ساده استفاده میکنید. شما میدونید که سرچ ساده به اندازه $O(n)$ برای اجرا شدن طول میکشد که این یعنی در بدترین حالت شما مجبورید تماماً تک به تک هر چیزی که در دفترچه تلفن ثبت شده رو بررسی کنیں. توی این حالت فرض کنین دارین دنبال اسم "ادیت" (Adit) میگردین. همینطور که از اسمش معلومه باید توی بخش اول (مدخل اول) کتاب قسمت Aها باشه (فرض کنید اسم "ادیت" دقیقاً اولین اسم دفترچه تلفنه) خب حالا شما مجبود نبودید کل دفترچه تلفن رو دنبال اسم ادیت بگردین و شما با اولین تلاش پیدا ش کردین. حالا این الگوریتم به اندازه $O(n)$ طول کشید یا به اندازه $O(1)$ ، چون شما با اولین تلاش اسم اون بنده خدا رو پیدا کردین.

سرچ ساده هنوزم به اندازه $O(n)$ زمان میبره. اما در این حالت شما بلا فاصله چیزی که دنبالش بودین رو پیدا کردین. این بهترین حالت این سناریو. اما بیگ او نویشن درباره بدترین حالت سناریو. پس میتوانیم این شکلی بگیم که در بدترین حالت شما باید یکبار تمامی مدخل های دفترچه تلفن رو بررسی کنید. و اون میگن $O(n)$. برای اطمینان خاطر، شما میدونین که سرچ ساده هیچ وقت از زمان $O(n)$ کند تر پیش نمیره.

یادتون باشه!

در کنار "بدترین حالت" "زمان اجرایی، مهمه که یک نگاهی هم به حالت میانگین زمان اجرایی بندازیم. در مورد "بدترین حالت" در مقابل "حالت میانگین" در فصل ۴ بحث میشه.

چند نمودار از زمان های اجرایی بیگ او های رایج

اینجا هم پنج تا از زمان های اجرا شدن بیگ او ها هستن که خیلی با هاشون رو به رو میشین.

مرتب شده از سریع به کند:

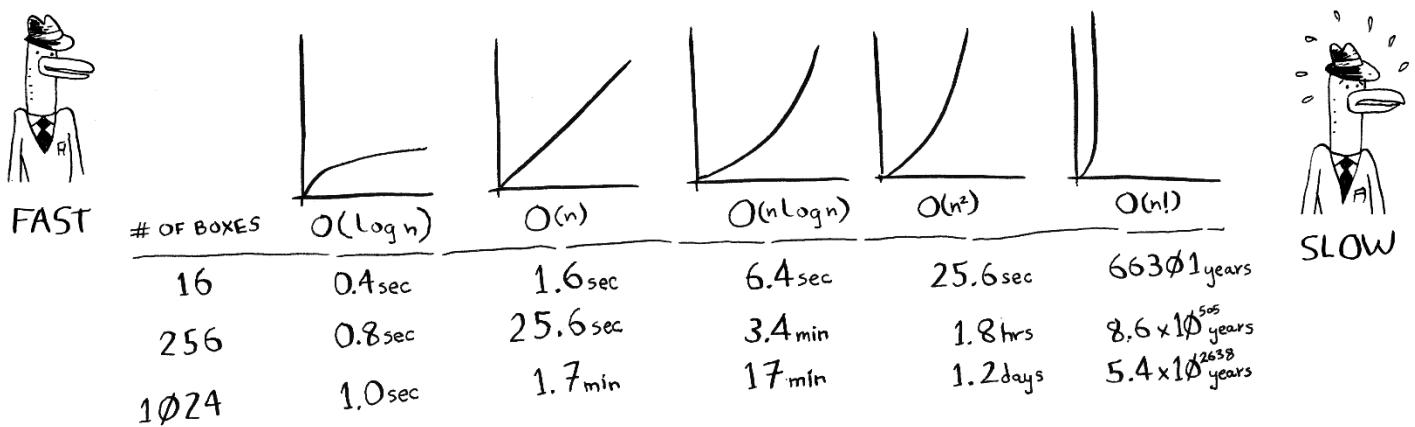
- $O(\log n)$ ، که همچنین به عنوان زمان لوگاریتمی هم شناخته میشے.مثال : باینری سرچ.
- $O(n)$ ،که همچنین به عنوان زمان خطی هم شناخته میشے.مثال:سرچ ساده
- $O(n \times \log n)$ ، مثال : الگوریتم مرتب سازی سریع مثل کوئیک سورت(مرتب سازی سریع) که در فصل ۴ باهاش کارداریم
- $O(n^2)$ ، مثال : الگوریتم مرتب سازی کند مثل مرتب سازی انتخابی(Selection Sort)
- $O(n!)$ ، یه الگوریتم خیلی کند مثل فروشنده دوره گرد(بعد از این میریم سراغش).

فرض کنید که شما دوباره میخواین یک جدول ۱۶ خونه ای بکشین. و میتوانیم از بین این ۵ الگوریتم مختلف یکیشود برای انجام این کار انتخاب کنید.اگر الگوریتم اول رو انتخاب کنید از شما به اندازه $O(\log n)$ زمان میگردد تا خونه های جدول رو بکشین. فرض کنیم شما میتوانیم ۱۰ تا عملیات در یک ثانیه انجام دهید. با $O(\log n)$ شما فقط ۴ عملیات برای کشیدن ۱۶ خانه نیاز دارین($\log 16 = 4$) پس ۰.۴ ثانیه از شما زمان میبرد تا خانه های جدول رو بکشین. ولی اگر قرار باشد که 10^{24} خانه بکشید چی؟ $10 = \log 1024$ پس ۱۰ عملیات یا میتوانیم بگیم ۱ ثانیه طول میکشه تا 10^{24} تا خانه بکشید.این اعداد با استفاده از الگوریتم اول بدست میاد.

الگوریتم دوم کنترله: به اندازه $O(n)$ طول میکشه تا اجرا بشه . ۱۶ بار عملیات باید انجام بشه تا ۱۶ تا خانه کشیده بشه و 10^{24} عملیات برای ۱۰۲۴ خانه. اگر بخوایم به ثانیه بگیم چقدر میشه؟

این جاهم میتوانید ببینید که برای بقیه الگوریتم چقدر طول میکشه که بخوان خانه های جدول رو بکشن.

از سریعترین به کندرین:



زمان های اجرایی دیگری هم هستن، اما این ۵ تا از رایج ترین ها هستن.

البته این یک ساده سازیه. در واقعیت شما نمیتوانیم به این سادگی زمان اجرایی که توسط بیگ بدست اومده رو به تعداد عملیات ها تبدیل کنید. ولی تا اینجا به اندازه کافی برای الان خوبه. ما، بعد از اینکه شما چند الگوریتم بیشتر رو یادگرفتید، در فصل ۴ دوباره به بیگ او نویشن برمیگردیم. اما درحال حاضر موارد اصلی وکلید به شرح زیرن:

- سرعت الگوریتم ها بر مبایث اندازه گیری نمیشوند. بلکه بر حسب افزایش تعداد عملیات ها سنجیده میشوند
- به جای اون ، ما در این مورد بحث میکنیم زمان اجرایی الگوریتم مورد نظر، با افزایش اندازه ورودی ها ،با چه سرعتی افزایش پیدا میکند.
- زمان اجرایی الگوریتم ها با بیگ او نویشن بیان میشوند.
- $O(\log n)$ سریع تر از $O(n)$ عمل میکند. اما سرعت این الگوریتم با افزایش اندازه لیست(لیستی که میخواهیم جستجو کنیم داخلش) افزایش پیدا میکند.

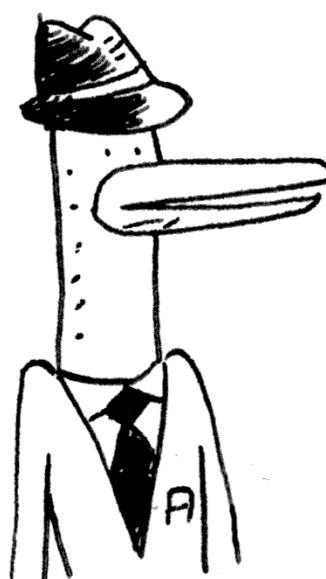
تمرین ها

زمان اجرایی هر کدام از این سناریو ها رو با استفاده از بیگ او نویشن بدید.

- ١/٣ یک اسم دارین، و میخواین شماره تلفن این فرد رو از توی دفترچه تلفن پیدا کنین.
- ١/٤ یک شماره تلفن دارین و میخواین اسم صاحب شماره رو از توی دفترچه تلفن پیدا کنین.(راهنمایی: شما باید کل کتاب رو جستجو کنین)
- ١/٥ شما میخواین شماره افراد داخل شماره تلفن رو بخونین.
- ١/٦ شما فقط میخواین شماره افراد توی بخش A هارو بخونین.(این یکم گمراه کنندس! این سوال شامل مفاهیمیه که در فصل ۴ بیشتر پوشش داده شده-برین جواب رو بخونین شاید سوپرایز بشین!)

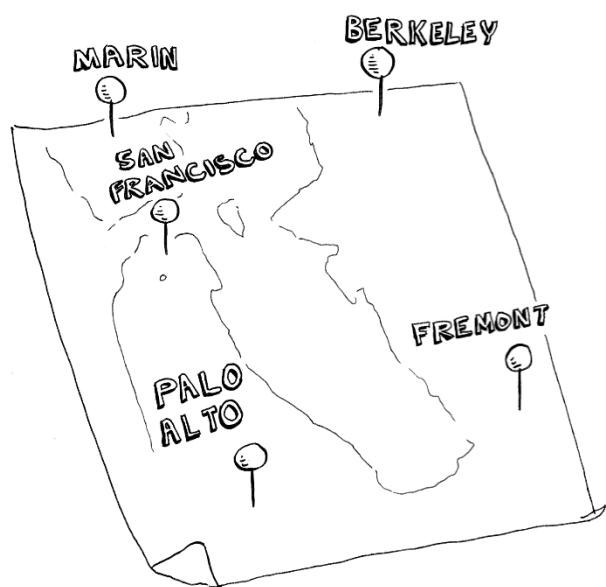
فروشنده دوره گرد

ممکنه شما بخش آخر رو خونده باشن و فک کنین که "امکان نداره که من به الگوریتمی بخورم که به اندازه $O(n!)$ ثابت کنم که دارین اشتباه میکنین! اینجا یه واقعا زمان اجراییش بده! این یک مسئله‌ی رشد اون وحشتناکه و تعدادی از افراد نمیتونه (بیشتر از این) بهبود پیدا کنه. بهش طول بکشه" خب بزارین بهتون مثل از یک الگوریتمی داریم که معروف در علوم کامپیوتره، چراکه باهوش فکر میکنند که این مسئله میگن مسئله فروشنده دوره گرد.

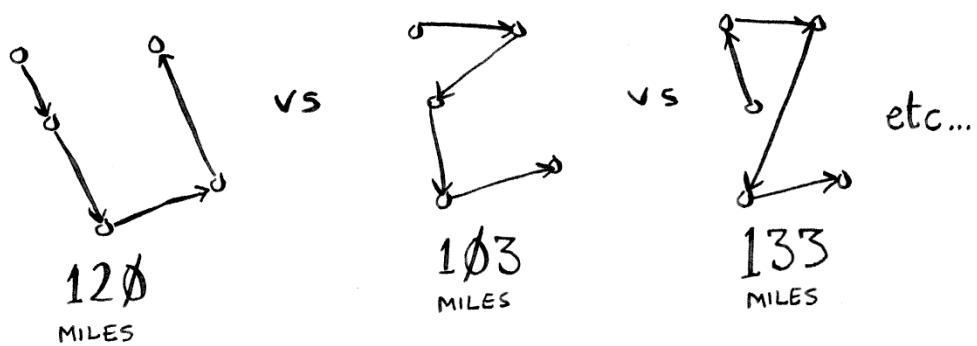


شما یک فروشنده دوره گرد دارین.

این فروشنده قرار به ۵ تا شهر سفر کنه.



این جناب فروشنده که من اوپوس(Opus) صداش میزنم، میخواود درحالی که کمترین مسیر رو طی میکنه به تمام این ۵ شهر سر بزنه. این یک راه حلش: به هر ترتیب ممکنی که او میتواند بین این شهرها سفر کند نگاه کنید.



اون همه مسافت های این راه ها رو جمع میکنه و راهی که مسافت کمتری رو داره انتخاب میکنه. ۱۲۰ جایگشت با ۵ شهر وجود دارد. پس میتوانیم بگیم ۱۲۰ بار این عملیات باید انجام بشه تا ما بتونیم این مسئله رو برای ۵ شهر حل کنیم. برای ۶ شهر این مقدار به ۷۲۰ میرسه (۷۲۰ جایگشت وجود داره). برای ۷ شهر این مقدار به ۵۰۴۰ عملیات میرسه.

CITIES	OPERATIONS
6	720
7	5040
8	40320
...	...
15	1,307,674,368,000
...	...
30	2,652,52,859,812,191,058,63,630,84,80,000,000

تعداد عملیات ها بشدت رو به افزاش است

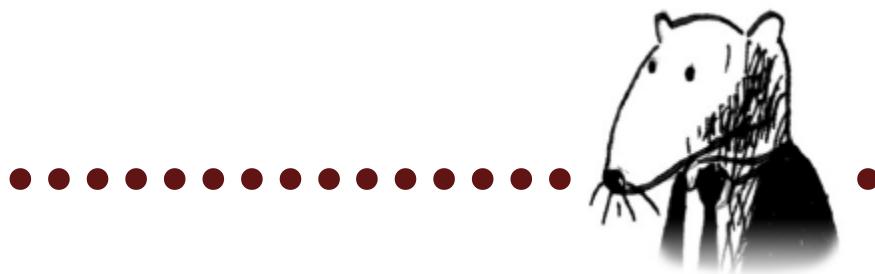
به طور کلی ، برای $n!$ آیتم، به اندازه $n!$ فاکتوریل) عملیات انجام میشود تا نتیجه محاسبه شود. پس این یک زمان $O(n!)$ یا تایم فاکتوریلی است. عملیات های خیلی زیادی برash انجام میشه مگر اینکه عددهش کوچک باشه. زمانی که شما با بیشتر از ۱۰۰ شهر کار میکنین ، غیر ممکنه که بتونین به جواب رو به موقع محاسبه کنین(به عبارتی جواب ، وقت گل نی بدست میاد).

این یک الگوریتم افتضاحه! اوپوس باید از یک الگوریتم دیگه استفاده کنه، درسته؟اما نمیتونه. این یکی از مسئله های حل نشده در علوم کامپیوتره. هیچ الگوریتم شناخته شده سریعی برای این کار وجود نداره، و افراد باهوش هم فکر میکنند داشتن الگوریتم باهوشی که این مسئله رو حل کنه غیرممکنه. بهترین کاری که میتوانیم انجام بدیم اینه که با یک راه حل تقریبی بربیم جلو، برای اطلاعات بیشتر بربین سراغ فصل ۱۰.

یادداشت آخر: اگر شما یک خواننده‌ی حرفه‌ای هستید، درخت جستجوی دودویی (Binary search trees) را چک کنید. یه توضیح خلاصه‌ای ازش در فصل آخر هست.

- باينر سرچ خيلى سريعتر از سرچ سادس
- $O(n)$ از $O(\log n)$ سريعتر، اما سرعتش بيشتر ميشه وقتی که ليستی از آيتم هايی که شما قراره بینشون سرچ کنی افزايش پيدا ميکنه.
- سرعت الگوريتم بر مبنای ثانيه اندازه گيري نميشود
- زمان الگوريتم ها بر پایه ی رشد اونها بررسی ميشه
- زمان الگوريتم ها براساس بيگ او نوتيشن نوشته ميشن

مرتب سازی انتخابی



در این فصل:

- شما آرایه ها و لیست های پیوندی رو که دو تا از پایه ای ترین ساختار داده هاست، یادمیگرین. او نا دقیقا همه جا استفاده میشن. شما قبل از آرایه ها در فصل ۱ استفاده کردیدن و تقریبا در تمام فصل های در این کتاب استفاده خواهید کرد. آرایه ها یک موضوع خیلی مهم هستند، پس خوب بهشون توجه کنیدن. ولی گاهی اوقات بپنده که به جای آرایه ها از لیست های پیوندی استفاده کنیدن. این فصل، مزايا و معایب جفت اون هارو بهتون توضیح میده تا بتونیدن تصمیم بگیرن که کدوم یک از اونها برای الگوریتمی که میخواین پیاده کنید مناسب تره.
- شما اولین الگوریتم مرتب سازی خودتون رو یادمیگرین. بسیاری از الگوریتم ها تنها زمانی کار میکنن که داده های شما مرتب و سورت شده باشند. باینری سرج رو یادتون هست؟ باینری سرج رو تنها میتوانید روی یک لیست از عناصر مرتب و سورت شده اجرا کنید. بیشتر زبان های برنامه نویسی الگوریتم های مرتب سازی رو درون خودشون دارن، پس خیلی کم پیش میاد که شما بخواین ورژن خودتون رو از ابتدا درست کنیدن. اما مرتب سازی انتخابی پله ای برای یادگیری و پیاده سازی کوئیک سورت است،

که او نو توى فصل بعدى پوشش خواهم داد. کوبيك سورت يك الگوريتم مهمه و
فهميدنش خيلي آسون ميشه اگر شما قبلش يكى از الگوريتم هاي مرتب سازی رو
بدونين

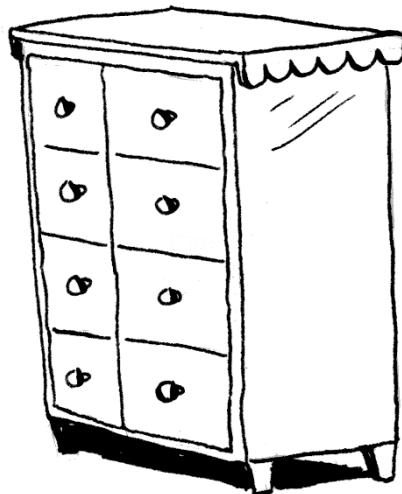


چیزایی نیازه بدونین

برای فهمیدن بحث تحلیل عملکرد بیت‌ها در این فصل، نیازه که شما با بیگ او نوتبشون و لوگاریتم آشنا باشین. اگر آشنایی ندارین بهتون پیشنهای میدم به عقب برگردین و فصل ۱ رو مطالعه کنید. از مبحث بیگ او نوتبشون در بین مطالب باقی این کتاب استفاده میشه

مموری چطور کار میکنه

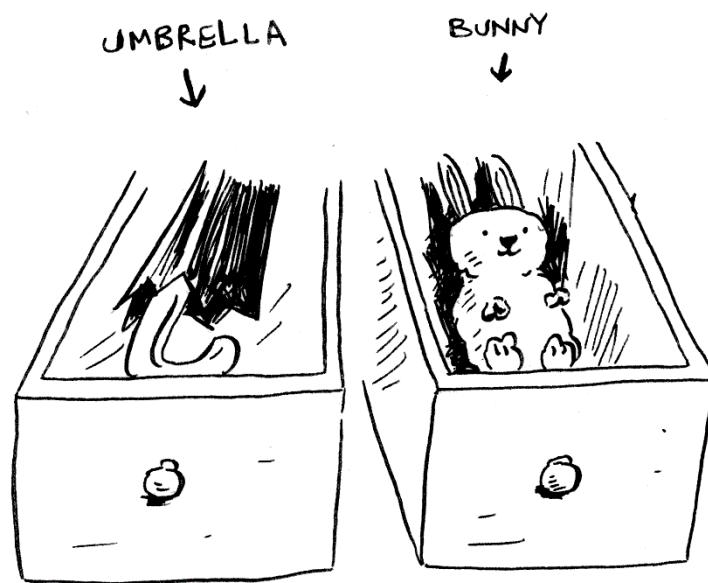
فرض کنین میخواین به یک نمایش بین و نیازه که وسایلتونو برسی کنین که ببین چی دارین چی ندارین. خب یک قفسه کشودار (دراور) در دسترسه.



هر کشو میتوانه یک شی رو در خودش نگه داره. خب اگر شما بخواین ۲ چیز رو نگه دارین باید ۲ تا کشو درخواست کنین.

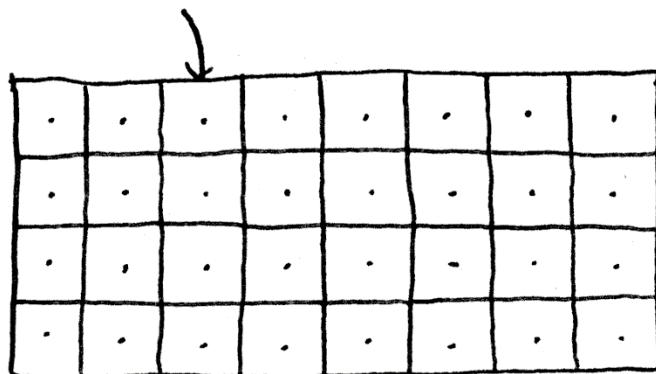


خب شما ۲ چیز رو در کشوها میدارین.



و شما برای نمایش آماده این! اساسا مموری کامپیوتر شما اینطوری کار میکنه. کامپیوتر شما مثل یک مجموعه‌ی عظیمی از کشو میمونه که هر کشو یک آدرس مختص خودش دارد.

ADDRESS: fe0ffe0b



آدرس fe · ffeeb یکی اسلات های مموری است.

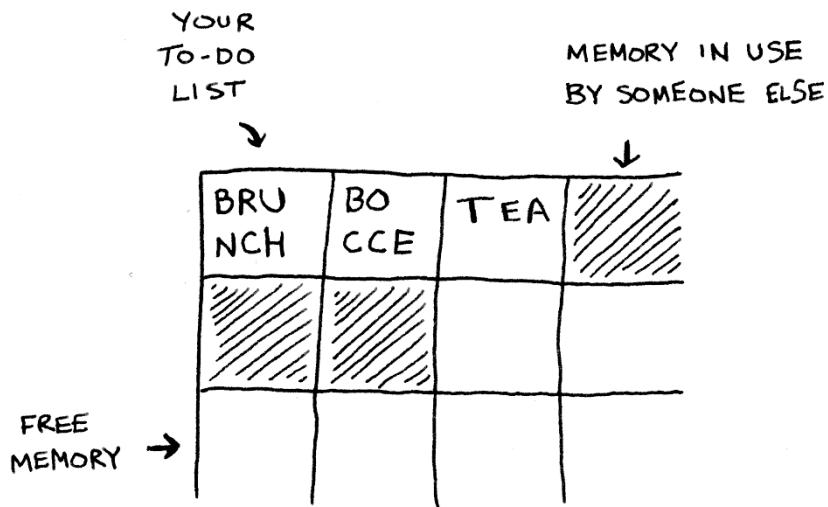
هربار که شما میخواین آیتمی رو در مموری ذخیر کنین. شما درخواستی رو برای مقداری فضا به کامپیوتر ارسال میکنین. و کامپیوتر هم بهتون آدرسی که میتوینین اون آیتم رو تو ش ذخیره کنین میده. اگر میخواین چندین آیتم ذخیره کنین، ۲ راه هست که میشه این کارو باهاش انجام داد: آرایه ها و لیست ها (Arrays and Lists). در ادامه درباره آرایه ها و لیست ها و همچنین راجبه مزایا و معایت اونها باهاتون صحبت میکنم. خوبه که بدونین فقط یک راه درست برای تمامی حالت های ذخیره آیتم ها وجود نداره. پس این مهمه که تفاوت اونها رو بدونین.

آرایه ها و لیست های پیوندی

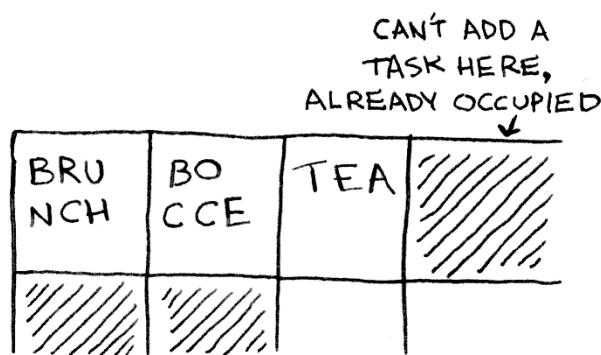


بعضی وقت ها نیاز دارین که لیستی از عناصر رو در مموری ذخیره کنید. فرض کنید میخواین برنامه ای بنویسید که باهاش کارای روزانتون رو مدیریت کنین. شما میخواین کاراها روزانتون رو به صورت لیست در مموری ذخیره کنین.

حالا بهتره که از آرایه ها استفاده کنیم یا لیست های پیوندی؟ بیاین اول کارهای روزانه در آرایه ذخیره کنیم، چون درکش راحتتره. استفاده از آرایه این معنی رو میده که همه تسك ها (tasks) پشت سر هم (دقیق کنار هم دیگه) در مموری ذخیره میشن.



حالا فرض کنین میخواین (طبق شکل) تسك چهارمی هم بهش اضافه کنین. اما کشو بعدی توسط وسایل یک نفر دیگه گرفته شده.



مثل این میمونه که شما با دوستاتون رفتین سینما و دنبال جا برای نشستن هستین اما یک دوست دیگتون بهتون ملحق میشه و هیچ جایی برای اون (کنار شما) نیست. شما باید به نقطه‌ی دیگه ای بین که همتون باهم اونجا جا میشن. به همین حالت، شما هم باید از کامپیوترون برای یک تکه ای دیگه ای از مموری که هر چهارتا میتوون تو ش جا بشن، درخواست کنین. بعدش نیاز دارین که تمام تسك هاتونو به اونجا منتقل کنین.

اگر یک دوست دیگه هم بیاد، شما بازم جا کم میارین و دوباره باید جایه جا بشین. عجب دردرسی! به طوری مشابهی، اضافه کردن یک آیتم به آرایه میتوانی همینقد دردرس ساز باشه. اگر قرار باشه که از فضای کافی نداشته باشین و هر سری به یک نقطه جا به جا بشین، اضافه کردن آیتم جدید خیلی کند پیش میره. یک راه ساده برای درست کردن این مشکل اینه که "جاهارونگه دارین": حتی اگر در لیست تسك هاتون ۳ تا آیتم دارین، شما میتوانین از کامپیوتر (محض احتیاط) ۱۰ اسلات (۱۰ تا جا) درخواست بکنین. و بعدش شما میتوین ۱۰ تا آیتم به

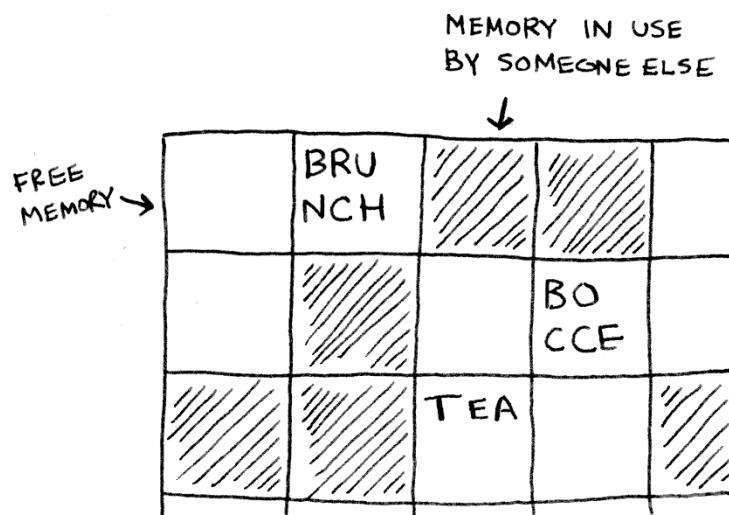
لیستتون بدون نیاز به جا به جایی، اضافه کنین. این یک راه حل خوبه، اما بهتره که از جنبه های منفیش هم آگاه باشین:

- ممکنه به اون اسلات های اضافی که درخواست کردین نیاز نداشته باشین، درنتیجه اون بخش از مموری هدر میره. شما ازش استفاده ای نمیکنین ولی کس دیگه هم ازش استفاده نمیکنه.
- ممکنه که اصلا ۱۰ تا آیتم دیگه (به علاوه اون ۱۰ آیتمی که داشتین) بهش اضافه کنین و در اون حالت مجبورید که جایه جا بشین.

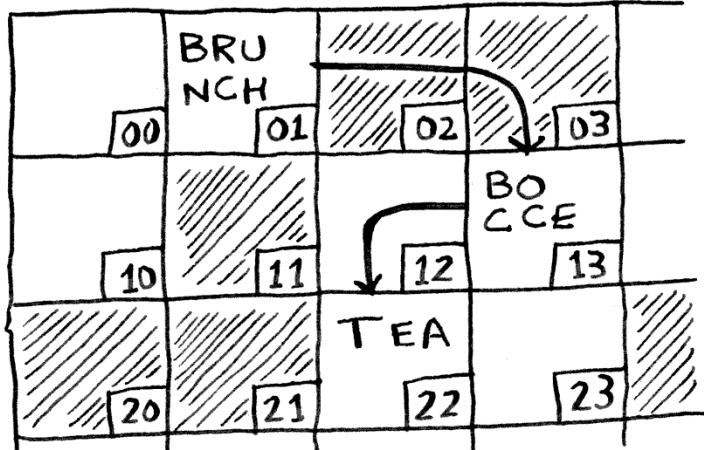
خب پس این راه حل خوبیه اما راه حل عالی و کاملی نیست. لیست های پیوندی مشکل اضافه کردن آیتم جدید رو حل کردن.

لیست های پیوندی

با لیست ها پیوندی، آیتم های شما میتوانن هرجایی توی مموری باشن.



هر آیتم آدرس آیتم بعدی رو در خودش ذخیره میکنه. دسته ای از آدرس های مموری که به صورت رندوم هستند به هم دیگه لینک شدن (وصل شدن).



مثل گشتن دنبال گنج میمونه(ممکنه نویسنده به کتاب *treasure hunt* اثر بایرن پرس و شان کلی هم اشاره کرده باشه!). شما میری سراغ آدرس اول و اون میگه "میتوñی آیتم بعدی رو توی آدرس ۱۲۳ پیدا کñی". پس شما میری سراغ آدرس ۱۲۳ و اون میگه "آیتم بعدی توی خونه ۸۴۷ پیدا میشە" و الی آخر. اضافه کردن آیتمی به لیست پیوندی آسونه: میتوñی اون آیتم رو هرجایی از مموری بچسبونی و آدرسشو در آیتم قبلی ذخیره کñی. با لیست پیوندی (لینکد لیست میگیم بهش) مجبور نیستی آیتم هاتو جابه جا کñی. همچنین از مشکل دیگه ای هم جلوگیری میکنیں. بیاین اینطور بگیم که شما با پنج تا از دوستاتون به تماشای فیلم معروفی میرین. دوست شیشمتون میاد و دنبال جای نشستن میگردد. ولی این فیلم(یا تاتر یا هرچی) مملو از جمعیته و هیچ ۶ تا جای خالی کنار همی وجود نداره. خب، بعضی موقع ها این اتفاق برای آرایه ها میفته. فرض کنین شما سعی دارین ۱۰۰۰۰ تا اسلات(یا جا) برای یک آرایه پیدا کنین. مموری شما ۱۰۰۰۰ اسلات داره اما کنار هم نیستن. شما هم نمیتوñین فضای مورد نیاز رو برای آرایتون فراهم کنین. لینکد لیست (لیست پیوندی) اینجوریه که میاد میگه "بیان پخش بشیم و فیلم رو تماشا کنیم" اگر فضایی در مموری وجود داشه باشه پس فضا برای لینکد لیست (لیست پیوندی) هم هست.

اگر لینکد لیست ها در بحث درج آیتم یا همون اضافه کردن آیتم جدید خیلی بهترن، پس آرایه ها برای چه کاری خوبن؟

آرایه ها



%10 EVIL CAT NEXT

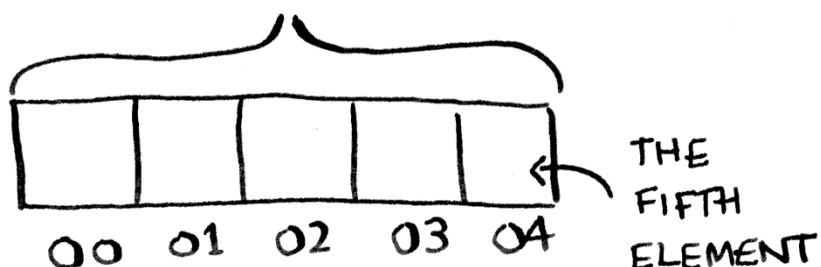
وب سایت هایی با محتواهای "تاب تن(یا ۱۰ تا بتر...)" از یک تاکتیک کثیف برای افزایش بازدید صفحه استفاده میکنن. به جای نشون دادن تمام لیست توی یک صفحه، هر آیتم رو در یک صفحه میزارن تا شما مجبور بشین روی دکمه "بعدی" کلیک کنین تا شما رو به آیتم بعدی توی اون لیست هدایت کنن. به عنوان مثال، "۱۰ تا از بهترین

شورهای تلویزیون" نمیاد تمام اون لیست رو در یک صفحه نمایش بده، بجاش میاد از #10 (*NewMan*) شروع میکنه، و شما باید روی "بعدی" توی تمام صفحات کلیک کنین تا به #10 (*Gustavo Fring*) برسین، این تکنیک به

وب سایت ۱۰ تا صفحه میده که توی هر کدومش به شما تبلیغ نشون بده. ولی این خیلی حوصله سر بره که ۹ بار کلیک کنین تا به شماره ۱ برسین. این بهتر میشه اگر شما میتونستین تمام لیست رو توی یک صفحه ببینین و میتوونستین روی اسم هر فرد برای گرفتن اطاعت بیشتر کلیک کنین. لینکد لیست (لیست پیوندی) همچین مشکل مشابهی داره. فرض کنین شما میخواین آیتم آخر در لینکد لیست رو مطالعه کنین. شما همون لحظه نمیتوانین بخونیش چراکه، شما نمیدونین اون توی کدوم آدرس نشسته. شما باید برین سراغ آیتم #۱ تا آدرس آیتم #۲ رو بدست بیارین. و از اونجا باید برین سراغ آیتم #۲ تا آدرس آیتم #۳ بدست بیارین و همینطوری تا جایی که برسین به آخرین آیتم. اگر بخواین تمام لیست رو در آن واحد بخونین لینکد لیست ها (لیست های پینودی) برای اینکار عالین اما اگر میخواین مدام به آیتم ها که به صورت مستقیم بهم وصل نیستن دسترسی پیدا کنین، لینکد لیست ها برای این کار اصلا خوب نیستند.

آرایه ها متفاوتن تو این موضوع. شما آدرس تمامی آیتم هارو که توی آرایتون هستن از قبل میدونین. به عنوان مثال فرض کنین آرایه شما شامل ۵ آیتم است و شما میدونین که از ۰۰ شروع میشن. خب پس آدرس آیتم #۵ چیه؟

ARRAY OF FIVE ITEMS



یه حساب سرانگشتی به شما میگه که : آدرسش ۰۴ . اگر میخواین عناصر یا آیتم های رندوم رو بخونین آرایه ها عالین، چراکه میتوونین هر عنصر رو در آرایه خودتون بلافصله جستوجو کنین. در لینکد لیست عناصر کنار هم نیستند پس نمیتوونین بلافصله موقعیت عنصر پنجم رو در مموری محاسبه کنین. شما باید برین سراغ عنصر اول تا آدرس عنصر دوم رو بگیرین بعدش باید برین سراغ عنصر دوم تا آدرس عنصر سوم رو بگیرین و همینطوری تا آخر تا به عنصر پنجم برسین.

اصطلاحات فنی

عناصر در آرایه به صورت شماره گذاری شده هستند. این شماره گذاری از ۰ شروع میشه نه ۱. به عنوان مثال در این آرایه، ۲۰ در موقعیت ۱ قرار دارد.

10	20	30	40
----	----	----	----

۰ ۱ ۲ ۳

و ۱۰ در موقعیت ۰ قرار دارد. این موضوع معمولاً کمی برنامه نویسان تازه کار را گیج میکند. شروع از ۰ برای هر نوعی کدی که براساس آرایه پیاده سازی شده، نوشتنش رو راحت میکنه. برای همین هم برنامه نویسان به این موضوع چسبیده ان. نقریباً هر زبان برنامه نویسی که استفاده میکنین عناصر رو در آرایه ها از ۰ شماره گذاری میکنه. به زودی بهش عادت میکنین.

به موقعیت هر عنصر "ایندکس" میگن. خب ما هم به جای اینکه بگیم، ۲۰ در موقعیت ۱ قرار داره، از اصطلاح فنیه درستش که این باشه استفاده میکنیم: ۲۰ در ایندکس ۱ قرار داره. من از کلیمه "ایندکس" برای اشاره به موقعیت عنصر در این کتاب استفاده میکنم.

اینجا هم زمان های اجرایی برای عملیات های رایج در آرایه ها و لیست ها (لينکدليست) میتوانین مشاهده کنین:

	ARRAYS	LISTS
READING	$O(1)$	$O(n)$
INSERTION	$O(n)$	$O(1)$

$$O(n) = \text{LINEAR TIME}$$

$$O(1) = \text{CONSTANT TIME}$$

$O(n)$ = زمان خطی

$O(1)$ = زمان ثابت

سوال: چرا به برای درج کردن یک عنصر در آرایه $O(n)$ طول میکشه؟ فرض کنین شما میخواین یک عنصر در اول یک آرایه درج کنین. چطور این کارو میکنین؟ چقدر طول میکشه که این کارو بکنین؟ جواب این سوال هارو میتوونید در قسمت بعدی پیدا کنین.

تمرین

۲.۱ فرض کنید در حال ساخت برنامه ای برای پیگیری وضعیت مالی خودتون هستین.

- 1. GROCERIES
- 2. MOVIE
- 3. SFBC
- MEMBERSHIP

هرروز، چیزایی که برashون پولی خرج کردین رو مینویسین. در آخر ماه هزینه هاتون رو بررسی میکنین و جمع میزنین. پس شما کلی چیز برای اضافه کردن و درج کردن دارین و چیزای ها کمی برای خوندن. شما باید از لیست استفاده کنین یا آرایه؟

درج داده در وسط لیست

فرض کنین شما میخواین فهرست کارهای روزانتون بیشتر شبیه یک تقویم باشه. قبل از این، آیتم ها و چیزای هایی که میخواستین رو به آخر این لیست اضافه میکردین. اما حالا میخواین اونارو به ترتیب الویت (که کدومشون باید زود تر انجام بشن) اضافه کنین.

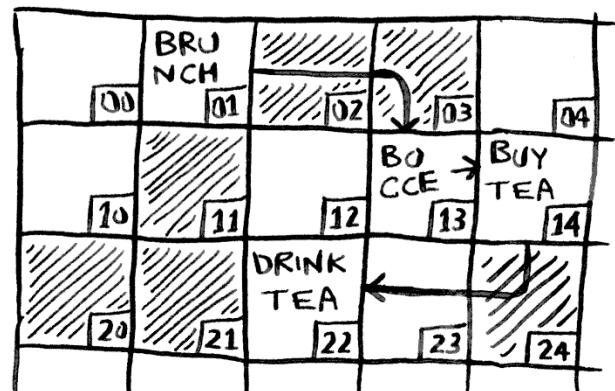
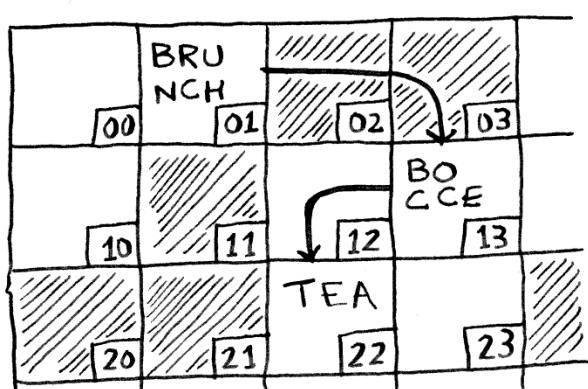


بدون ترتیب

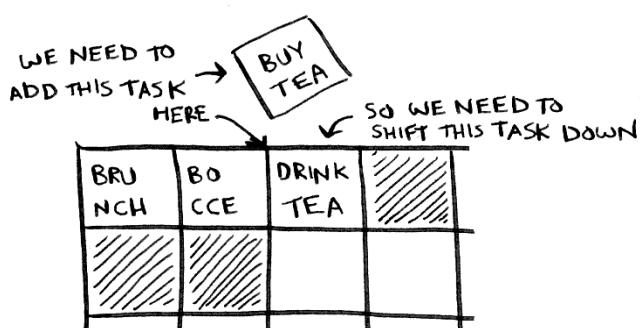


با ترتیب

از چی استفاده کنیم بهتره، اگر بخواین عنصری رو به وسط این لیست اضافه کنیم: آرایه یا لیست؟ در لیست ها این کار به همون اندازه سادست که ، عوض کردن آدرسی که عنصر قبلی بهش اشاره میکنه سادس.



اما در آرایه ها، شما باید بقیه عناصر رو بر دارین و کناربزندین



و اگر فضایی نباشد، شاید مجبور بشید همه چیزو توی یک موقعیت دیگه کپی کنیم! اگر شما میخواین چیزی رو به وسط عناصر اضافه کین لیست ها بهترن و بهتر عمل میکنن.

حذف کردن

حالا اگر بخوایم عنصری حذف کنیم چی؟ دوباره لیست ها بهترین چرا که شما بازم تنها چیزی که نیاز دارین اینکه که چیزی که عنصر قبلی داره بهش اشاره میکنه رو عوض کنیم (ینی آدرس که عنصر قبل از این عنصری که میخوایم حذف کنیم رو عوض کنیم). با آرایه ها، وقتی شما عنصری رو حذف میکنین، نیازه که همه چیز جایه جا بشه.

برخلاف درج کردن و اضافه کردن (آقا اینسربت کردن)، حذف کردن همیشه کار میکنه. درج کردن وقتی فضای دیگه ای رو مموری نمونده، میتونه صورت نگیره. اما شما همیشه میتوونین یک المان (عنصر) رو حذف کنین.

اینجا هم میتوونین یکسری از زمان های اجرایی برای عملیات های رایج در آرایه ها و لینکد لیست ها مشاهده کنین.

	ARRAYS	LISTS
READING	$O(1)$	$O(n)$
INSERTION	$O(n)$	$O(1)$
DELETION	$O(n)$	$O(1)$

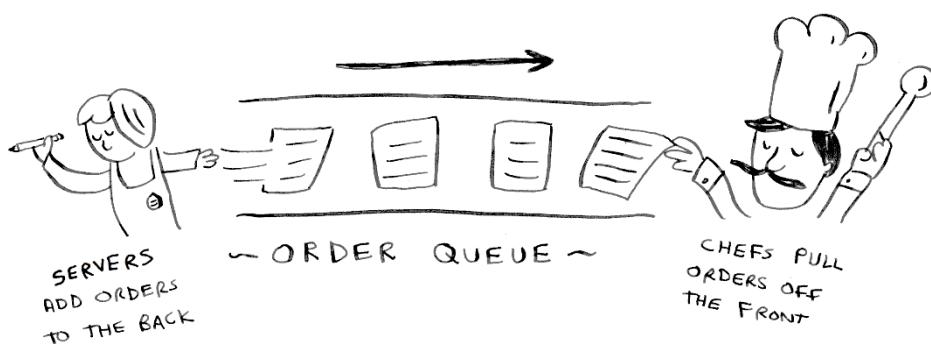
شایان ذکره که تنها زمانی زمان اجرایی درج کردن و حذف کردن $O(1)$ است که بلافاصله بتوان به آن المان (عنصر) دسترسی پیدا کرد. این که حواسمنون باشه و بدونیم که آیتم اول و آیتم آخر در لینکد لیست کجا هستن (آدرسشنون چیه) یک روش رایج، که اینطوری تنها به اندازه $O(1)$ طول میکشه تا اونها رو حذف کنیم.

کدومشون بیشتر استفاده میشن؟ آرایه ها یا لیست ها؟ خب واضحه که به جا و حالتی که میخوایم استفاده کنیم بستگی داره. اما آرایه ها به خاطر اینکه اجازه دسترسی رندوم به ما میدن، استفاده از اونها، بیشتر دیده میشه. دو نوع دسترسی وجود داره: دسترسی رندوم (*random access*) و دسترسی متوالی (*sequential access*). دسترسی متوالی ینی خوندن تک به تک المان ها (عناصر)، که از اولین المان شروع میشه. لینکد لیست ها تنها به صورت "دسترسی متوالی" کار میکنن. اگر میخواین المان i اهم رو در یک لینکد لیست بخونین، اول باید شروع به خوندن i تا اول بکنین و لینک هارو در المان ها دنبال کنید تا به المان i برسین. دسترسی رندوم ینی میتوونین مستقیم پیرین روی المان i اهم. بارها از من مشنويد که میگم آرایه ها در خوندن سریعتر عمل میکنن. این بخاطر اینه که اونا

دسترسی رندم رو برای ما فراهم میکنن. بسیاری از حالت ها و بسیاری از جاها به دسترسی رندم نیاز میشه. پس درنتیجه آرایه ها خیلی مورد استفاده قرار میگرن. آرایه ها و لیست ها (لینکد لیست) در بقیه ساختار داده ها هم مورد استفاده قرار میگرند (که در ادامه این کتاب بهشون میرسیم)

تمرین ها

۲.۱ فرض کنین شما دارین یه برنامه برای یک رستوران میسازین که سفارش مشتری ها رو دریافت کنه. برنامه شما نیازه داره که سفارش ها رو در لیستی از سفارشات رو ذخیره کنه. سرور ها به اضافه کردن سفارش ها به لیست ادامه میدن. و آشپز هم سفارش ها رو از لیست میگیره و شروع به درست کردنشون میکنه. این یک صفت از سفارش است: سرور ها سفارش رو به آخر صفت اضافه میکنن و آشپز هم سفارش هارو از اول صفت بر میداره و شروع به پخت اونها میکنه.



آیا شما از آرایه برای پیاده سازی این برنامه استفاده میکنین یا از لینکد لیست؟

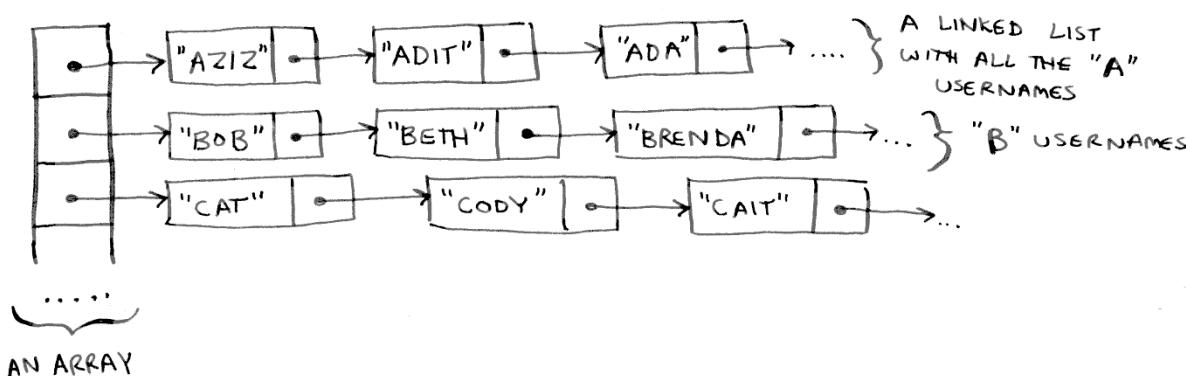
(راهنمایی: لینکد لیست ها برای درج / حذف کردن خیلی خوبن، و آرایه ها برای دسترسی رندم به خوبی عمل میکنن. شما از کدومش استفاده میکنین؟)

۲.۲ بیاین یک آزمایش فکری انجام بدیم. فرض کنید فیس بوک لیستی از نام های کاربری رو نگه میداره. وقتی یک نفر سعی میکنه که وارد حسابش بشه، یک جستجو برای پیدا کردن نام کاربری آنها انجام میشه. اگر اسم اونها در لیست نام های کاربرشون باشه، او نا میتونن وارد حساب کاربریشون بشن. افراد اغلب وارد فیس بوک میشن (لاگ این میکنن)، پس سرج های زیادی به لیست نام های کاربری زده میشه. فرض کنین فیس بوک از باینری سرج برای سرج زدن داخل لیستش استفاده میکنه. باینری سرج از دسترسی رندم استفاده میکنه شما نیاز دارین که بلا فاصله به وسط لیست دسترسی داشته باشید. حالا که اینو میدونین، شما برای پیاده سازی این لیست از آرایه استفاده میکنین یا از لینکد لیست؟

۲.۳ افراد اغلب هم برای ثبت نام در فیس بوک اقدام میکنن، خب فرض کنین که شما تصمیم گرفتین که از آرایه ها برای ذخیره لیست کاربران استفاده کنین، جوانب منفی استفاده از آرایه برای درج کردن ها (*insert*) چی هستن؟ به

ویژه، فرض کنید که شما از باینری سرج برای جستجو برای وارد کردن (لگین کردن) کاربرها استفاده میکنیں؟ چه اتفاقی میفته وقتی یک یوزر جدید به آرایه اضافه میشه؟

۲.۵ در واقع، فیس بوک نه از آرایه ها و نه از لینکد لیست ها برای ذخیره اطلاعات کاربران استفاده میکنه. بیاين یک ساختار داده‌ی ترکیبی رو درنظر بگیریم: آرایه‌ی لینکد لیستی. شما آرایه‌ای با ۲۶ اسلات دارین. هر اسلات به یک لینکد لیست اشاره میکنه (متصله). برای مثال، اولین اسلات در آرایه به لیستی (لینکد لیست) شامل نام‌های کاربری A میشن که با A شروع میشن. دومین اسلات در آرایه به لیستی (لینکد لیست) شامل نام‌های کاربری میشن که با B شروع میشن. و الی آخر.



فرض کنین **Adit B** در فیس بوک ثبت نام میکنه، و شما میخواین اون ها رو به لیست اضافه کنین. شما میرین به اسلات اول، و وارد لینکد لیست میشین و **Adit** رو به آخرش اضافه میکنین. حالا فرض کنین شما میخواین اسم **Zakhir H** رو جوستجو کنین. شما میرین به اسلات ۲۶ که به لینکد لیست اسمامی Z اشاره میکنه. و شما توی اون لیست به دنبال **Zakhir** میگردین تا پیدا ش کنین.

این ساختار داده ترکیبی رو با آرایه ها و لینکد لیست ها مقایسه کنین. آیا سریعتر یا کند تر از هر کدام از اونا توی جستجو کردن و اضافه کردنه آیتمه؟ نیاز نیست بزنین تو بحث بیگ او. فقط آیا این ساختار داده جدید سریع تره یا کند تر؟

مرتب سازی انتخابی (Selection sort)

بیاين همه چیو بچینیم کنار هم و الگوریتم و دومتونو یادبگیرین:

مرتب سازی انتخابی. برای اینکه بتونین این بخش رو دنبال کنین، نیازه که آرایه ها و لیست ها (لینکد لیست) هارو و همچنین بیگ او نویشن رو درک کرده باشد.

فرض کنین شما تعدادی موزیک روی کامپیووترتون دارین، برای هر هنرمند، تعداد باری که آهنگش پخش شده رو دارین.

~ ♫ ~	PLAY COUNT
RADIOHEAD	156
KISHORE KUMAR	141
THE BLACK KEYS	35
NEUTRAL MILK HOTEL	94
BECK	88
THE STROKES	61
WILCO	111

شما میخواین این پلی لیست رو براساس تعداد باری که پخش شده از بیشترین به کمترین مرتب کنین. که بعدش میتوانید هنرمندان مورد علاقتون رو رده بندی کنین. چطور این کارو میکنین؟

یک راهش اینه که برین داخل لیست و اوئی که از همه بیشتر پخش شده رو پیدا کنین و به لیست جدید اضافش کنین.

~ ♫ ~	PLAY COUNT
RADIOHEAD	156
KISHORE KUMAR	141
THE BLACK KEYS	35
NEUTRAL MILK HOTEL	94
BECK	88
THE STROKES	61
WILCO	111



♫ SORTED ♫	PLAY COUNT
RADIOHEAD	156

1. RADIOHEAD
IS THE MOST PLAYED
ARTIST...

2. ADD IT TO
A NEW LIST

یک بار دیگه این کارو بکنین تا بیشترین اهنگ پخش شده از یک هنرمند دیگ رو پیدا کنین.

♪	PLAY COUNT
KISHORE KUMAR	141
THE BLACK KEYS	35
NEUTRAL MILK HOTEL	94
BECK	88
THE STROKES	61
WILCO	111

♪ SORTED ♪	PLAY COUNT
RADIOHEAD	156
KISHORE KUMAR	141

1. KISHORE KUMAR
IS THE NEXT
MOST-PLAYED
ARTIST

2. SO IT IS
THE NEXT ARTIST
ADDED TO THE
NEW LIST

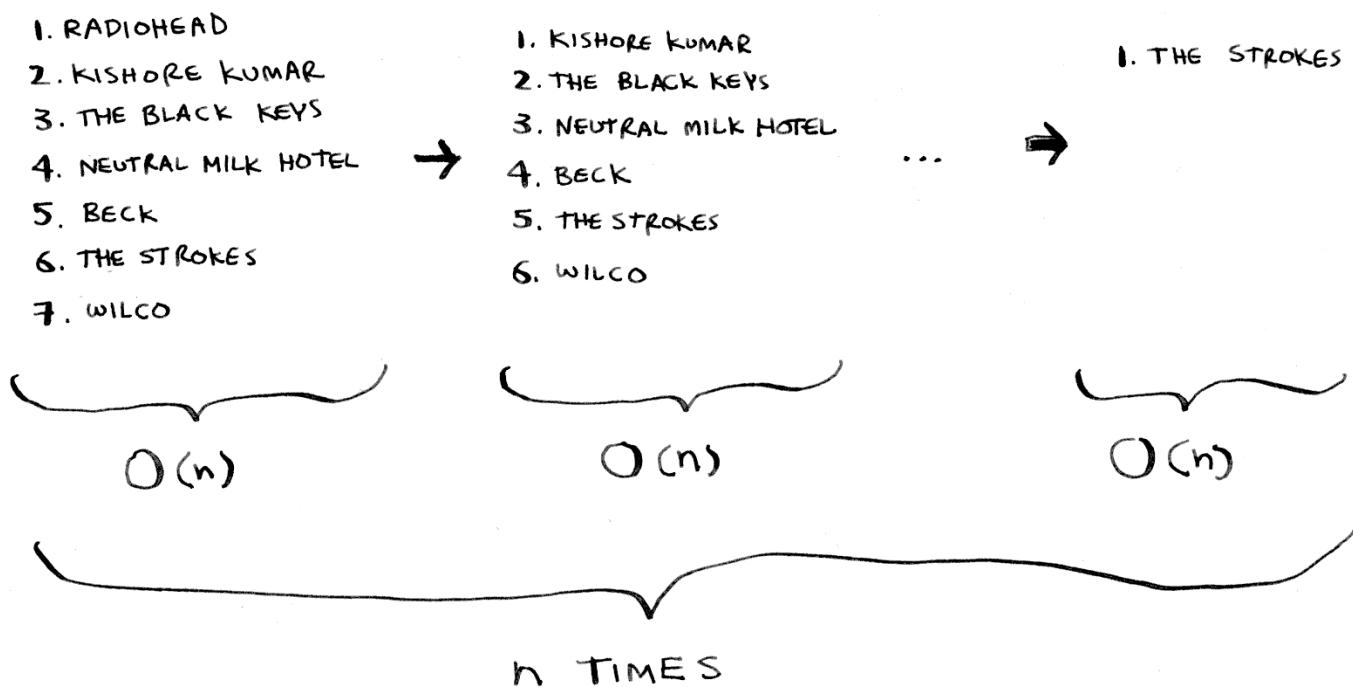
همین کارو ادامه بدین تا جایی که در نهایت با یک لیست مرتب شده مواجه خواهد شد.

♪	PLAY COUNT
RADIOHEAD	156
KISHORE KUMAR	141
WILCO	111
NEUTRAL MILK HOTEL	94
BECK	88
THE STROKES	61
THE BLACK KEYS	35

بیاین کلاه علوم کامپیوترویمونو بزاریم سرمون و ببینیم این مرتب سازی چقدر طول میکشه. اون $O(n)$ رو یادتونه که معنیش این بود که شما هر المانی رو یکبار لمس میکنی. به عنوان مثال، استفاده از سرج ساده روی لیستی از هنرمند ها یعنی هر هنرمند رو یکبار بررسی میکنیم.

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

برای پیدا کردن هنرمندی که بیشتر تعداد پخش رو داشته، شما باید تمام آیتم هارو در لیست چک کنید. همونطور که دید، این زمانی معادل $O(n)$ میگردد. پس شما عملیاتی دارین که به اندازه $O(n)$ طول میکشه، و اونو باید n بار تکرار کنید.



این مرتب سازی به اندازه $O(n \times n)$ یا $O(n^2)$ طول میکشد.

الگوریتم های مرتب سازی بسیار مفید هستند. الان شما میتوین این داده هارو مرتب کنید:

- اسم ها در شماره تلفن
- تاریخ های مسافرت
- ایمیل ها (از جدیدترین به قدیمی ترین)

بررسی المان کمتر در هر بار بررسی کردن

شاید پیش خودتون فکر کنین که بار که این مرتب سازی انجام میشه تعداد المان های برای بررسی همینطوری کاهش پیدا میکنه. درنهایت اینقدر این روند ادامه پیدا میکنه که درنهایت فقط یک المان برای بررسی میمونه. پس چطوری هنوز زمان اجرایی میتونه $O(n^2)$ باشه؟ سوال خوبیه، و جوابی که باید بدیم بهش اینه: ثابت ها در بیگ او توپیش (ترجم: اگر اون ویدیویی که برای درگ بیگ او بهتون معرفی کردم رو دیده باشین این بخشو راحت درک میکنین) در فصل ۴ بیشتر در این باره عمیق میشیم.

شما درست میگین که ما نباید هر سری یک لیست n المانی رو بررسی کنیم. شما اول n المان رو چک میکنی بعدش $n-1, n-2, \dots, 1$ به طور میانگین شما لیستی با $\frac{1}{2}n$ المان رو بررسی میکنید. زمان اجرایی میشه $O(n \times \frac{1}{2}n)$. اما ثابت هایی مثل $\frac{1}{2}$ در بیگ او نوپیش چشم پوشی میشه (بازم میگم، برای بحث کامل میتوونین به فصل ۴ نگاه کنین)، پس شما فقط مینویسین $O(n^2)$ یا $O(n \times n)$.

مرتب سازی انتخابی (سلکشن سورت) یک الگوریتم منظم و تمیزه، اما خیلی سریع نیست. مرتب سازی سریع (کوییک سورت) یک الگوریتم مرتب سازی سریعه که فقط به اندازه $O(n \log n)$ زمان میگیره. در فصل بعدی راجب شم میخونیم.

نمونه کد (برای) لیست کردن

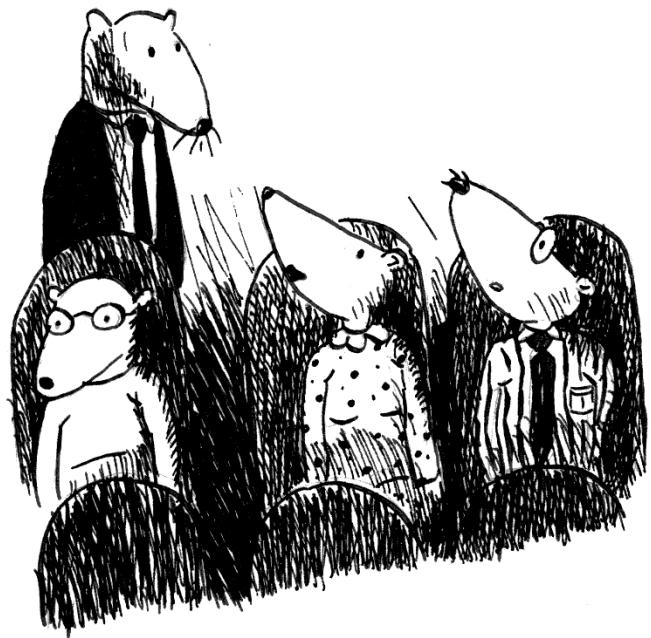
ما نمونه کد مرتب کردن لیست موزیک ها رو نشون شما ندادیم، اما در ادامه کدی چند خط کدی هست که کار مشابهی انجام میده: مرتب کردن آرایه ای از کوچکترین بیاین یک تابع بنویسیم تا کوچک ترین عضو آرایه رو پیدا کنیم:

```
def findSmallest(arr):
    smallest = arr[0] <----- Stores the smallest value
    smallest_index = 0 <----- Stores the index of the smallest value
    for i in range(1, len(arr)):
        if arr[i] < smallest:
            smallest = arr[i]
            smallest_index = i
    return smallest_index
```

حالا شما میتوانید از این تابع برای نوشتن الگوریتم مرتب سازی (سلکشن سورت) استفاده کنید:

```
def selectionSort(arr): <----- Sorts an array
    newArr = []
    for i in range(len(arr)):
        smallest = findSmallest(arr) <----- Finds the smallest element in the
            newArr.append(arr.pop(smallest))   array, and adds it to the new array
    return newArr

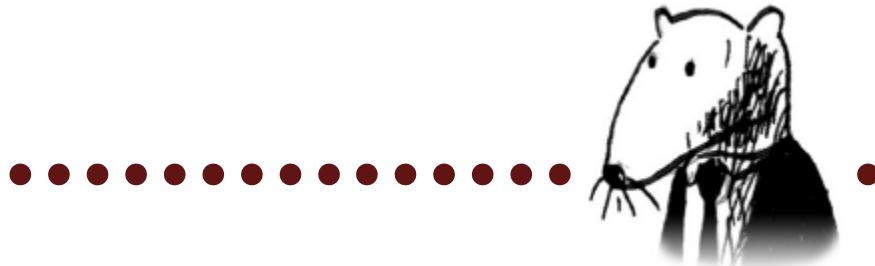
print selectionSort([5, 3, 6, 2, 10])
```



خلاصه این فصل:

- مموری کامپیووتر شما مثل یک قفسه کشودار(دراور) عظیمه(مجموعه ای عظیمی از کشوهاست)
- وقتی میخواین چندین المان رو ذخیره کنین از آرایه ها یا لیست ها (لینکد لیست ها) استفاده کنید
- با استفاده از آرایه تمامی المان های شما دقیقا کنار هم ذخیره میشوند.
- با استفاده از لیست، المان ها همه جا پراکنده میشن، و هر المان آدرس المان بعدی رو در خودش ذخیره میکنه.
- آرایه ها امکان سریع خوندن رو به ما میدن(امکان دسترسی سریع رو به ما میدن)
- لینکد لیست ها به ما امکان درج و حذف سریع رو به ما میدن
- تمام المان ها در آرایه باید از یک جنس باشن (همه عدد صحیح(*integer*) باشن یا همه اعشاری (*doubles*) باشن، و غیره)

بازگشت (Recursion)



در این فصل:

- شما درباره بازگشت یادمیگرین. بازگشت یک تکنیک کدنویسیه که در خیلی از الگوریتم ها استفاده میشه. این فصل پایه ای برای درک کردن فصل های بعدی در این کتابه.
- شما یادمیگرین که چطور یک مسئله رو بشکنینش تا به حالت ساده و حالت بازگشته اون برسین. استراتژی تقسیم و تسخیر (فصل ۴) از این مفهوم برای حل مسائل سخت استفاده میکنه.



برای این فصل خیلی هیجان دارم چون مبحث بازگشت رو پوشش میده، یک راه قشنگ برای حل مسائل سخت. بازگشت، یکی از موضوعات مورد علاقه منه. اما یک بحث جنجال برانگیزه. مردم یا عاشقش یا ازش متنفرن، یا ازش متنفرن تا زمانی که یادمیگرن در چند سال بعدی عاشقش بشن. من خودم شخصاً توی گروه سوم قراردارم: برای اینکه این موضوعات رو برآتون آسون تر بکنم، چننا پیشنهاد برآتون دارم:

- این فصل نمونه کدهای زیادی داره، کد ها رو برای خودتون اجرا کنین تا ببینین که چطور کار میکنن.
- در این فصل من درباره توابع بازگشتی باهاتون صحبت خواهم کرد. حداقل یکبار، با خودکار و کاغذ تابع بازگشتی رو دنبال کنین: مثلاً یک چیزی مثل: "بریم ببینیم، من ۵ رو به `factorial` پاس میدم، که یعنی ۵ بار باید اینکارو بکنم و هر سری یک مقدار ازش کم کنم در مقدار قبلی ضرب کنم و دوباره به `factorial` پاس بدم، حالا که ۵ رو پاس دادم سری بعد ۴ ضرب در ۵ رو به `factorial` پاس میدم، که مقدارش میشه..." (یک چیزی تو این مایه ها، اما اونطور که خودتون راحتین) و غیره. بررسی کردن همچین توابعی بهتون یادمیده که توابع بازگشتی چطور کار میکنن.

همچنین این فصل شامل تعداد زیادی شبهه کده. شبهه کد یه توضیح سطح بالا از مسئله ایه که شما سعی دارین حلش کنین، اما توضیحی در قالب کد. مثل کد نوشته شده اما طوری که به زمان انسان نزدیک باشه.

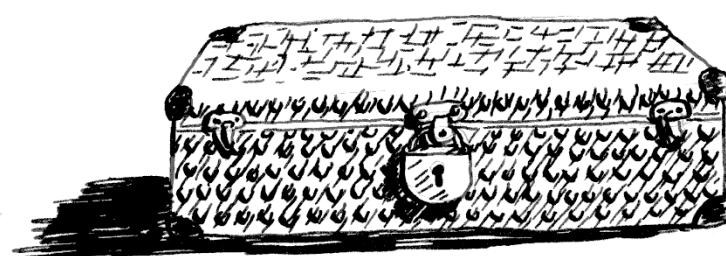
(مترجم: مثلاً این تازه برنامه نویسا هستن که رو عکس پروفایلشون میزنن:

```
While (awake()) {
    Code();
}
```

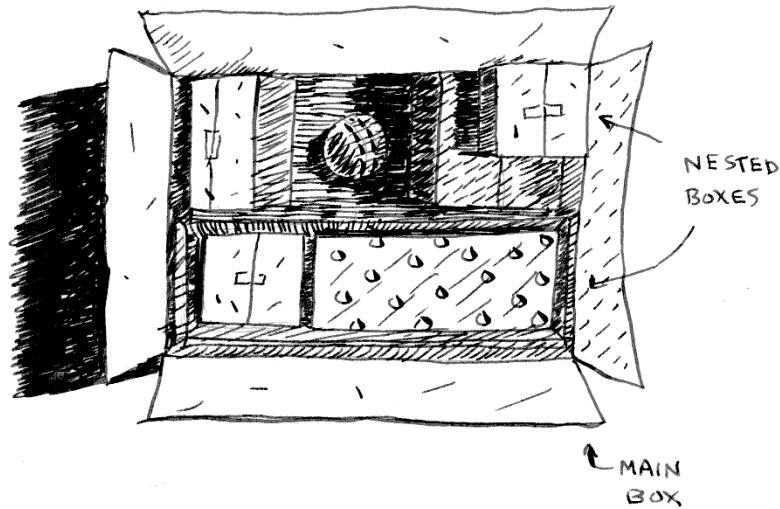
یه چیزی شبیه این:)

بازگشت

فرض کنین شما دارین داخل زیرشیرونی مادربزرگتونو میگردید که به یک چمدان مرموز قفل شده برمیخورین.

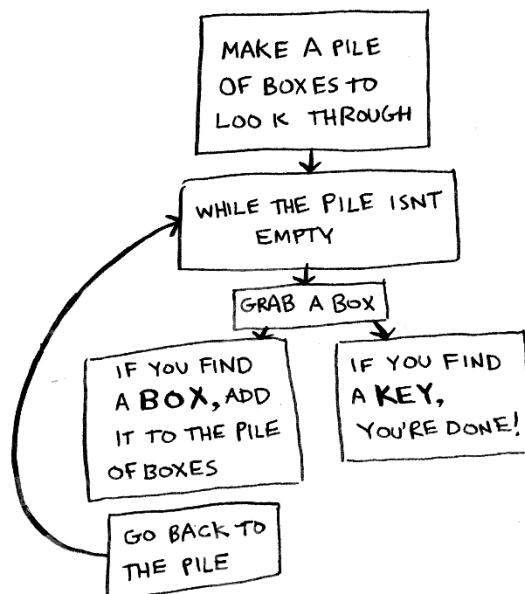


مادربزرگ بهتون میگه که کلید اون چمدان احتمالاً توی یک جبعه‌ی دیگس.



این جبعه داخلش جبعه های دیگه ای هم هستن. که داخل اون جبعه ها هم جبعه های دیگه ای هستن(عجب بساطیه). کلید یه جایی همین جاهاس. الگوریتم شما برای پیدا کردن و جستجوی کلید چیه؟ قبل از اینکه به خوندن ادامه بدید به یک الگوریتم فک کنید

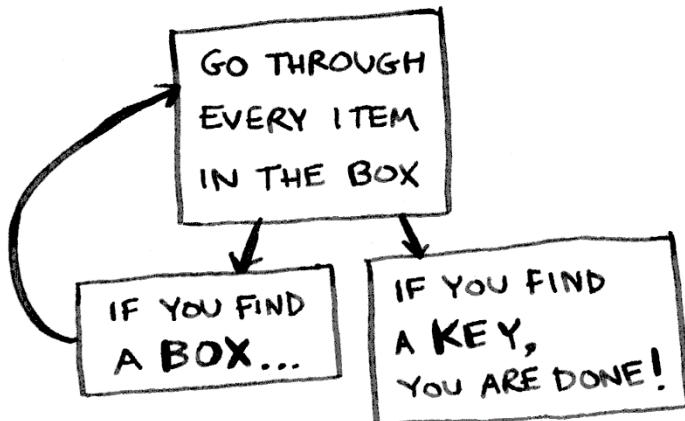
این یکی از اون روش هاست:



۱. تعداد زیادی از جبعه هارو برای بررسی کنار هم بزارین
۲. یک جبعه رو اخاب کنید و برش دارینو داخلشو نگاه کنین.
۳. اگر جبعه ی جدیدی پیدا کردین به گروه همون جبعه هایی که بررسی نشدن اضافش کنین تا بعدا بررسیش کنین
۴. اگر کلید رو پیدا کردین همه ، کار تمومه!

۵. اگر نه دوباره همین روند رو تکرار کنیں.

اینم یک روش جایگزین:



۱. داخل جعبه رو نگا کنیں.

۲. اگر جعبه ای پیدا کردین بین سراغ قدم ۱

۳. اگر کلید رو پیدا کردین، کار جمه!

کدوم روش آسون تر بنظر میاد؟ روش اول از حلقه `while` استفاده میکنه: تا زمانی کی (`while`) جعبه ها برای بررسی تموم نشدن، ادامه بده و یک جعبه بردار و داخلش رو بررسی کن:

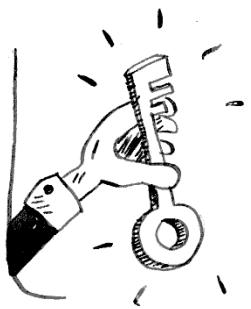
```
def look_for_key(main_box):
    pile = main_box.make_a_pile_to_look_through()
    while pile is not empty:
        box = pile.grab_a_box()
        for item in box:
            if item.is_a_box():
                pile.append(item)
            elif item.is_a_key():
                print "found the key!"
```

روش دوم از مفهوم بازگشت استفاده میکنه. بازگشت(*Recursion*) جایی اتفاق مییافته که یک تابع خودش رو صدا میزنه. اینم شبه کد راه دوم:

```
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!"
```

هر دو روش یک کار رو انجام میدن. اما روش دوم برای من واضح تره. بازگشت زمانی استفاده میشه که راه حل رو واضح تر میکنه. بازگشت، هیچ مزیت و برتری عملکردی (پرفورمنس) براش وجود نداره. درواقع، بعضی موقع ها، حلقه ها عملکرد(پرفورمنس) بهتری دارن. من این نقل قول، از لی کالدول(Leigh Caldwell) رو خیلی دوست دارم که میگه: "شاید حلقه ها عتمکرد بهتری برای برنامه شما به ارمغان بیارن، اما احتمال زیاد بازگشت(recursion) عملکرد بهتری برای برنامه نویس های شما به ارمغان میآورد. خودتون انتخاب کنید که کدام یک از آنها برای موقعیت شما مهم تر است." (<http://stackoverflow.com/a/139117/77694>)

بسیاری از الگوریتم های مهم از بازگشت (recursion) درون خودشون استفاده میکنند. پس این مهمه که مفهوم بازگشت رو درک کنید.



بخش پاپه و بخش بازگشتی

از اونجایی که تابع بازگشتی خودش رو فراخوانی میکنه، خیلی ساده پیش میاد که تابع رو اشتباه بنویسیم که به یک حلقه بینهایت ختم بشه(**infinite loop**). به عنوان مثال، فرض کنید که میخواین یک تابع بنویسید که شمارش معکوس رو به شکل زیر چاپ کند:

2.2.2.1

شما میتوانید این را به صورت بازگشتی به شکل زیر بنویسید:

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

این کد رو بنویسید و اجراش کنین. بعد از این کار متوجه یک مشکل میشین: این تابع تا ابد اجرا میشه و تمومی نداره.



$\geq 3 \cdots 3 \cdots 1 \cdots \cdot \cdots -1 \cdots -3 \cdots$

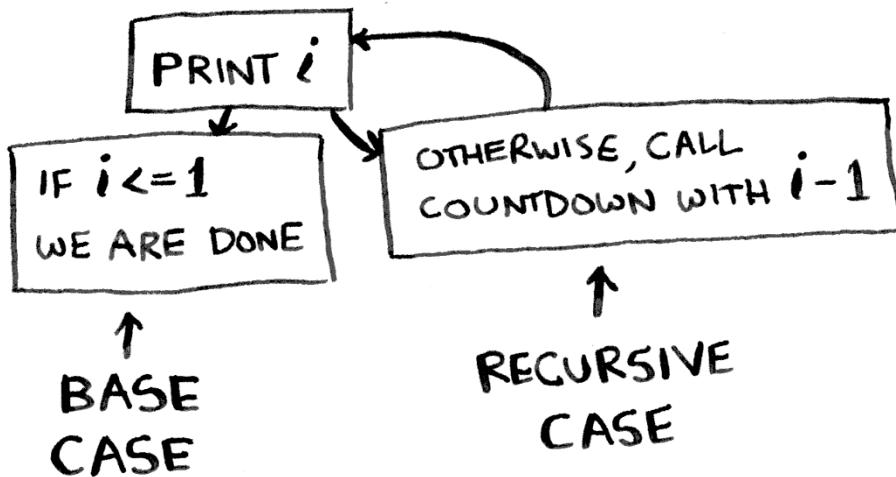
Ctrl+C رو بزنین تا اسکریپتی که نوشتهین رو متوقف کنین.)

وقتی که یک تابع بازگشتی مینویسین، باید مشخص کنین که کجا بازگشت باشد. به همین دلیله که هر تابع بازگشتی ۲ قسمت دارد: بخش پایه و بخش بازگشتی. بخش بازگشتی، وقتیه که تابع، خودش رو صدا میزنه. بخش پایه جاییه که تابع دیگه خودش رو صدا نمیزنه... پس در نیجه دیگه وارد یک حلقه‌ی بینهایت نمیشه.

بیاين حالت پایه رو به شمارش معکوس‌مون اضافه کنیم.

```
def countdown(i):
    print i
    if i <= 0: ← ..... Base case
        return
    else: ← ..... Recursive case
        countdown(i-1)
```

حالا این تابع همونطوری که انتظار داشتیم کار میکنه. این شکلی پیش میره که:

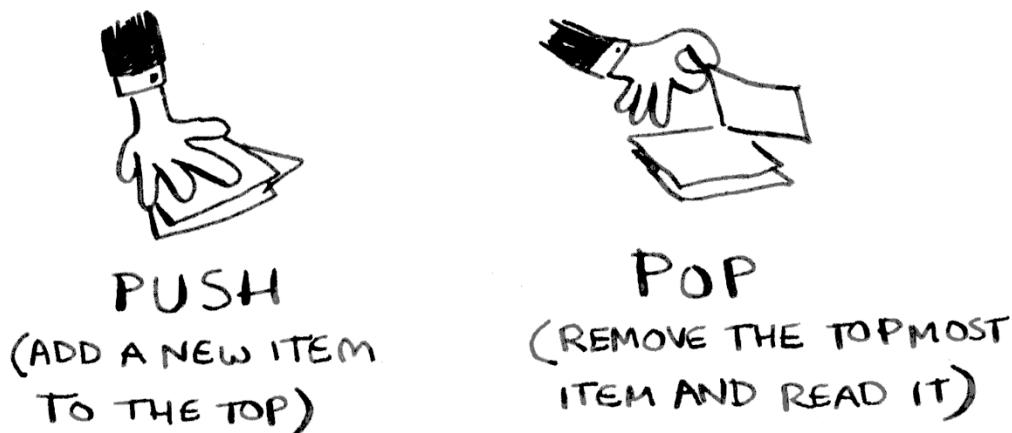


استک (پشته)

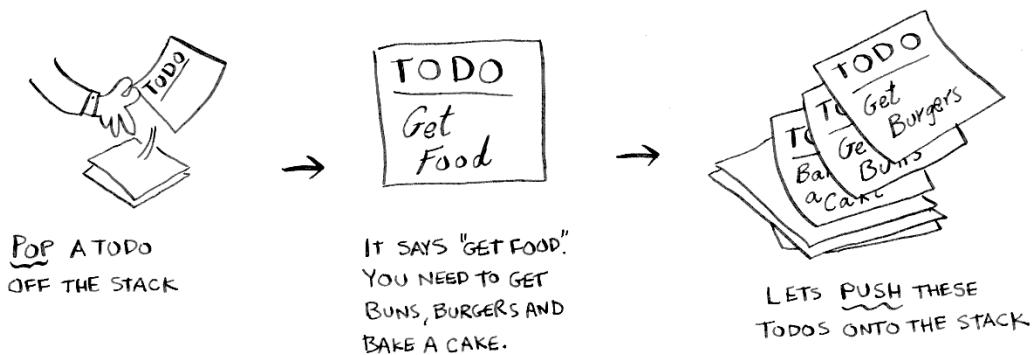
این بخش بحث کال استک (call stack) رو پوشش میده. این یک مفهوم مهم در برنامه نویسی هستش. به طور کلی مبحث کال استک یک مبحث مهم در کل دنیای برنامه نویسیه. و این در کال استک وقتی داریم از بازگشت استفاده میکنیم خیلی مهمه. فرض کنین میخواین که یک باربیکیو راه بندازین. لیستی از هر کدام از چیزایی که برای باربیکیو نیاز دارین رو در یک صفحه از یک استیکی نوت مینویسین.

یادتونه وقتی درمورد آرایه و لیست‌ها حرف زدیم و شما یک لیست از کارای روزانه داشتین؟ و میتوانستین آیتم‌هایی رو هرجایی از لیست کاراها‌تون که میخواین اضافه کنین یا آیتم‌هایی رو به صورت رندم (هر جایی که میخواین دستریشون داشته باشین) حذف کنین. استک (پشته)‌های استیکی نوت خیلی آسون تر

هستند. وقتی آیتم را اضافه میکنیم، به بالای لیست اضافه میشے. وقتی میخواین آیتمی را بخونین، فقط میتوانیم بالاترین آیتم را بخونین (که بعد از خوندن از لیست خارج میشے). خب لیست کاراهای شما هم فقط ۲ عمل را انجام میده. هل دادن (push) که همون اضافه کردنه، ترکوندن (pop) که ینی همون خوندن و حذف کردن.

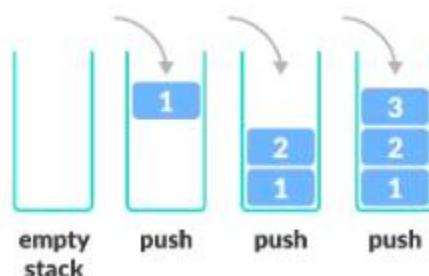


بیاین لیست کارهای روزانه را در عمل ببینیم.



به این ساختار داده، استک (پشته) گفته میشے. استک یک ساختار داده سادس. شما در تمام طول این مدت داشتن از استک استفاده میکردین بدون اینکه متوجه بشین.

(مترجم: برای فهمیدن ساختار داده ی استک شکل زیر را ببینین)



آخرین داده ای که وارد میشے، اولین داده ای هست که ما میتوانیم بهش دسترسی داشته باشیم. مثلا عدد ۱ طبق شکل بالا، اولین داده ایه که ما به استک دادیم و اگر بخوایم بهش دسترسی داشه باشیم اول باید ۲ رو بخونیم و برشداریم (حذفش کنیم)، بعدش ۲ رو باید بخونیم و برشداریم (حذفش کنیم) بعدش ۱ رو برداریم. پس ما فقط به بالاترین داده دسترسی داریم).

کال استک

کامپیووتر شما از استک در درون خودش برای انجام کارها استفاده میکنه که بهش کال استک میگن. بیاین توی عمل ببینیمش. اینجا یه تابع ساده داریم:

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

این تابع به شما خوش آمد میگه و بعد دو تابع دیگه رو صدا میزنه. اینم از اون دو تا تابع

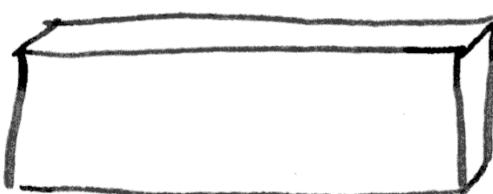
```
def greet2(name):
    print "how are you, " + name + "?"
def bye():
    print "ok bye!"
```

مترجم پایتون کارا دقیق کنن که تابع (bye) اشتباها توسط ویراستار کتاب اصلی ایندنت شده. و اینکه نویسنده در اینجا print را تابع در نظر نگرفته و بهش در ادامه اشاره میکنه.

بیاین ببینیم دقیقا چه اتفاقی میفته وقتی شما یک تابع رو فراخوانی میکنین.

یادداشت

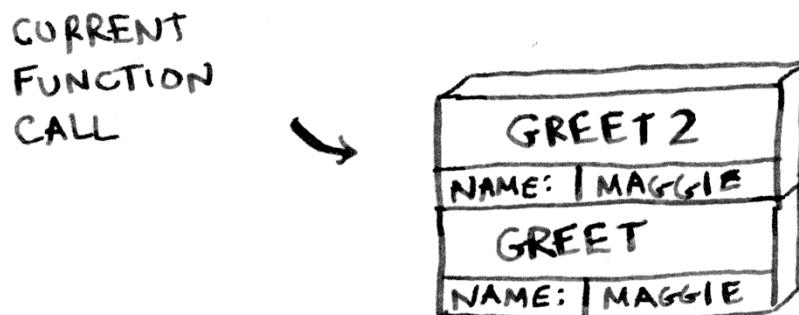
در پایتون یک تابع است. اما ما برای راحتی کار تظاهر میکنیم که نیست. ما میگیم زدیم شما هم بگین زده: فرض کنین شما ("Maggie") را صدا میزنین. اول از همه کامپیووتر شما یک جعبه (یک فضا (مترجم: نویسنده در اینجا برای درک بهتر از کلمه "جعبه" برای مقدار فضا استفاده میکنه)) از مموری رو به اون تابعی که فراخوانی شده اختصاص میده.



حالا بیان از اون بخش از مموری استفاده کنیم. مقدار "Maggie" برای متغیر name ست شده. خب پس نیاز هستش که در مموری ذخیره بشه.



هر بار که شما تابعی رو فراخوانی میکنین، کامپیووتر شما مقادیر وارد شده رو برای تمامی متغیرهای مربوطه در مموری نگه میداره، مثل همین چیزی که دیدیم. در قدم بعد شما "hello, Maggie!" رو چاپ میکنین و بعد از اون (("Maggie")) greet^۲ رو فراخوانی میکنین. دوباره کامپیووتر شما جعبه ای (فضایی) در مموری برای این تابعی که فراخوانی شده، اختصاص میده.



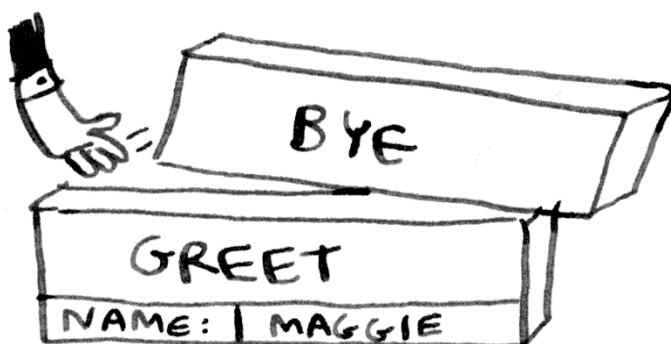
کامپیووتر شما از ساختار داده استک برای این جعبه ها استفاده میکنه. جعبه دوم روی جعبه اول اضافه میشه. حالا شما "how are you, Maggie?" رو چاپ میکنین و از اون تابع فراخوانی شده بر میگردین (اصطلاحا return میکنین). وقتی این اتفاق بیفته (وقتی از اون تابع برگردین)، اون جعبه در بالای استک برداشته و حذف میشه.



و حالا بالاترین جعبه در استک برای تابع `greet` هستش، که یعنی شما به تابع `greet` برگشتین. وقتی شما تابع `greet2` را فراخوانی کردید، تابع `greet` به اندازه ای تکمیل شده بود(نه کامل!). این ایده‌ی اصلی پشت این بخش‌های کتابه: وقتی شما تابعی رو از طریق تابع دیگری فراخوانی می‌کنید. تابع فراخوانی شده، در حالت نیمه کاملی مکث می‌کنه. تمام مقادیر برای متغیر‌ها برای تابع هم همچنان در مموری ذخیره شدن. حالا که کارتون با تابع "getting" تموم شده، از هموجایی که متوقف شدین به تابع `greet` برگردیدن. حالا اول `greet2` رو چاپ می‌کنین. سپس شما تابع `bye` را فراخوانی می‌کنین.



جعبه‌ای برای تابع `bye` در بالای استک اضافه می‌شه. و بعدش شما "ok `bye!`" رو چاپ می‌کنین از این تابعی که فراخوانی شده برگردیدن (`return` می‌کنین).



و به تابع `greet` برگردیدن. خب دیگه کاری نمونده که انجام نداده باشم پس از تابع `greet` هم برگردیدم. این استکی که برای ذخیره کردن متغیر‌ها برای چند تابع استفاده شد رو بپشت می‌گن کال استک.

تمرین

۳/۱ فرض کنین من یک کال استک مثل این نشونتون میدم:



چه اطلاعاتی بر اساس این کال استک میتوانیم بهم بدین؟

حالا بیاین کال استک رو در عمل در تابع بازگشتی ببینیم.

کال استک همراه بازگشت (recursion)

توابع بازگشتی هم از کال استک استفاده میکنند. بیاین اینو در تابع فاکتوریل بررسی کنیم. (۵) در ریاضی به شکل $5! = 5 \times 4 \times 3 \times 2 \times 1$ نوشته میشود و این شکلی تعریف میشه که: (۳) این هم یک تابع بازگشتی برای حساب کردن فاکتوریل یک عدد:

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

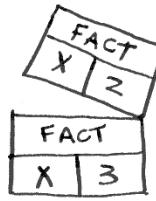
حالا شما میتوانید (۳) را صدابزنید. بیاین خط به خط بررسیش کنیم و ببینیم این استک چطور کار میکنه. یادتون باشه، بالاترین جعبه در استک بهتون میگه که الان توی کدوم مرحله از فراخوانی تابع fact هستین.

CODE	CALL STACK												
fact(3)	<table border="1"> <tr><td>FACT</td><td></td></tr> <tr><td>X</td><td>3</td></tr> </table>	FACT		X	3								
FACT													
X	3												
if x == 1:	<table border="1"> <tr><td>FACT</td><td></td></tr> <tr><td>X</td><td>3</td></tr> </table>	FACT		X	3								
FACT													
X	3												
else:	<table border="1"> <tr><td>FACT</td><td></td></tr> <tr><td>X</td><td>3</td></tr> </table>	FACT		X	3								
FACT													
X	3												
A RECURSIVE CALL!													
return x * fact(x-1)	<table border="1"> <tr><td>FACT</td><td></td></tr> <tr><td>X</td><td>2</td></tr> <tr><td>FACT</td><td></td></tr> <tr><td>X</td><td>3</td></tr> </table>	FACT		X	2	FACT		X	3				
FACT													
X	2												
FACT													
X	3												
NOW WE ARE IN THE SECOND CALL TO fact. X IS 2	<table border="1"> <tr><td>FACT</td><td></td></tr> <tr><td>X</td><td>2</td></tr> <tr><td>FACT</td><td></td></tr> <tr><td>X</td><td>3</td></tr> </table>	FACT		X	2	FACT		X	3				
FACT													
X	2												
FACT													
X	3												
if x == 1:	<table border="1"> <tr><td>FACT</td><td></td></tr> <tr><td>X</td><td>2</td></tr> <tr><td>FACT</td><td></td></tr> <tr><td>X</td><td>3</td></tr> </table>	FACT		X	2	FACT		X	3				
FACT													
X	2												
FACT													
X	3												
else:	<table border="1"> <tr><td>FACT</td><td></td></tr> <tr><td>X</td><td>2</td></tr> <tr><td>FACT</td><td></td></tr> <tr><td>X</td><td>3</td></tr> </table>	FACT		X	2	FACT		X	3				
FACT													
X	2												
FACT													
X	3												
return x * fact(x-1)	<table border="1"> <tr><td>FACT</td><td></td></tr> <tr><td>X</td><td>1</td></tr> <tr><td>FACT</td><td></td></tr> <tr><td>X</td><td>2</td></tr> <tr><td>FACT</td><td></td></tr> <tr><td>X</td><td>3</td></tr> </table>	FACT		X	1	FACT		X	2	FACT		X	3
FACT													
X	1												
FACT													
X	2												
FACT													
X	3												
if x == 1:	<table border="1"> <tr><td>FACT</td><td></td></tr> <tr><td>X</td><td>1</td></tr> <tr><td>FACT</td><td></td></tr> <tr><td>X</td><td>2</td></tr> <tr><td>FACT</td><td></td></tr> <tr><td>X</td><td>3</td></tr> </table>	FACT		X	1	FACT		X	2	FACT		X	3
FACT													
X	1												
FACT													
X	2												
FACT													
X	3												
WOW, WE MADE THREE CALLS TO fact, BUT WE HAD NOT FINISHED A SINGLE CALL UNTIL NOW!													
return 1	<table border="1"> <tr><td>FACT</td><td></td></tr> <tr><td>X</td><td>1</td></tr> <tr><td>FACT</td><td></td></tr> <tr><td>X</td><td>2</td></tr> <tr><td>FACT</td><td></td></tr> <tr><td>X</td><td>3</td></tr> </table>	FACT		X	1	FACT		X	2	FACT		X	3
FACT													
X	1												
FACT													
X	2												
FACT													
X	3												
	<p>FIRST CALL TO fact. X IS 3.</p> <p>THE TOPMOST FUNCTION CALL IS THE CALL WE ARE CURRENTLY IN</p> <p>NOTE: BOTH FUNCTION CALLS HAVE A VARIABLE NAMED X AND THE VALUE OF X IS DIFFERENT IN BOTH</p> <p>YOU CAN'T ACCESS THIS CALL'S X FROM THIS CALL AND VICE VERSA</p> <p>THIS IS THE FIRST BOX TO GET POPPED OFF THE STACK, WHICH MEANS ITS THE FIRST CALL WE RETURN FROM</p> <p>RETURNS 1</p>												

THIS IS THE
FUNCTION CALL
WE JUST RETURNED
FROM

return $x * \text{fact}(x-1)$

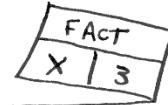
x is 2



return $x * \text{fact}(x-1)$

x is 3

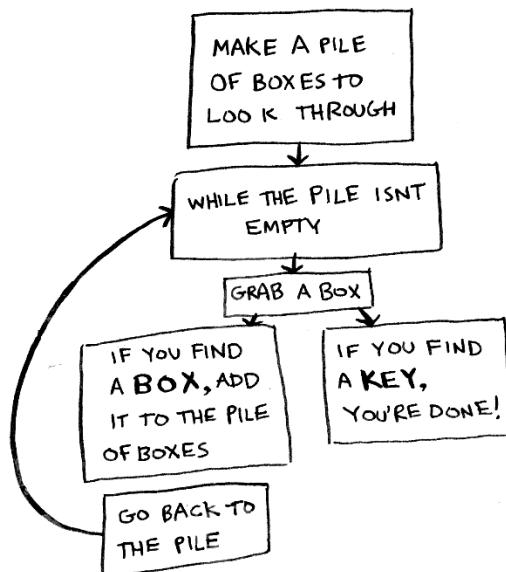
THIS CALL
RETURNED 2



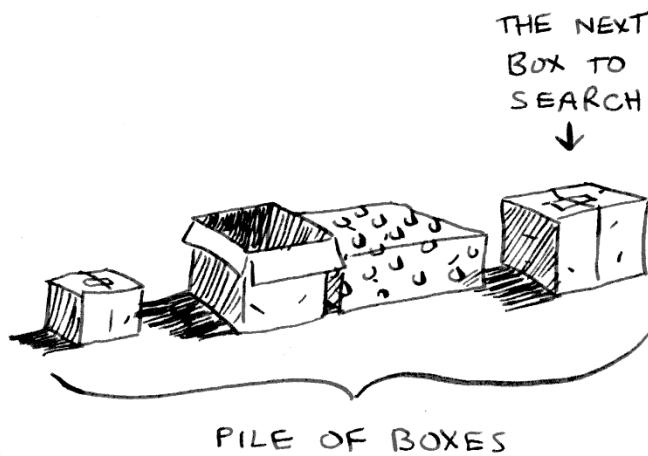
← RETURNS 6

در نظر داشته باشیم که تابع `fact` در هر فراخوانی، یک کپی از x برای خودش دارد. شما نمیتوانید به x های کپی شده‌ی توابع دیگه درسترسی داشته باشیم.

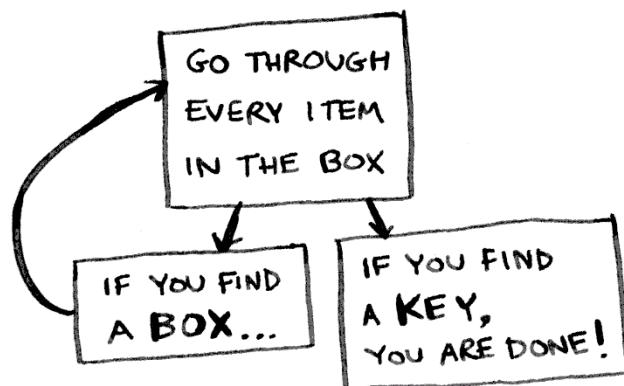
استک، نقش مهمی رو در بحث بازگشت، ایفا میکنه. در مثال ابتدایی این فصل، ۲ راه برای پیدا کردن کلید وجود داشت. این راه اول:



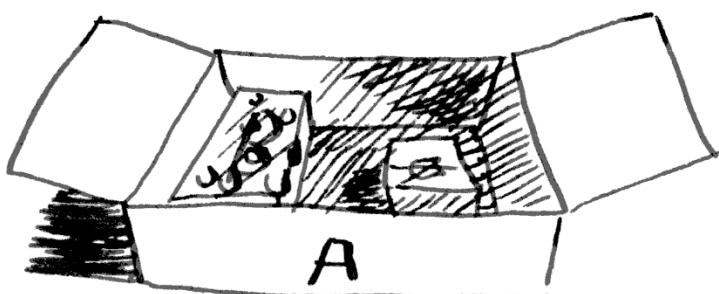
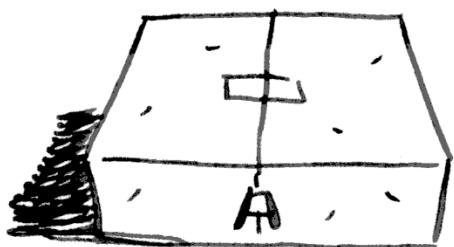
در این راه شما تعداد زیادی از جعبه‌ها که باید بررسی بشن رو کنار هم میزارین (یک توده از جعبه‌هه درست میکنین) و داخلشون رو بررسی میکنین، در نتیجه شما میدونین که کدوم جعبه‌ها هستن که هنوز بررسی نشدن و نیازه که چک بشن.



اما در حالت بازگشتی مثل قبل هیچ توده ای از جعبه ها رو جم نمکنین.

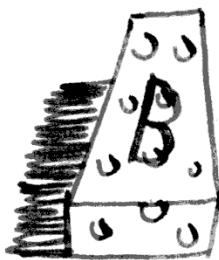


اگر توده ای از جعبه های بررسی نشده رو جم نمکنین، پس الگوریتم شما چطور میفهمه که کدوم جعبه ها رو باید بررسی کنه؟ اینجا یک مثال داریم:



YOU LOOK THROUGH
BOX A

INSIDE YOU FIND
BOXES B AND C



YOU CHECK
BOX B

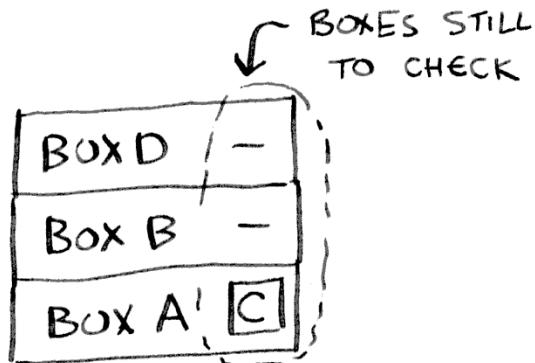
IT CONTAINS
BOX D



YOU CHECK
BOX D

IT IS
EMPTY

در این مرحله ، کال استک مثل شکل زیره:



توده جعبه ها، روی استک ذخیره میشن! این یک تابع فراخوانی شده (توسط خودش) است، و هر کدوم از آونها هم، شامل لیست نیمه کاملی از جعبه هایی هستن که قراره بررسی بشن. استفاده از استک راحته، چون دیگه نیاز نیست خودتون حواستون به تعداد زیادی از جعبه ها باشه که آیا بررسی شدن یا نه، استک خودش این کارو برآتون میکنه.

استفاده از استک راحته، اما هزئیه داره: ذخیره کردن اون همه اطلاعات میتونه حافظه‌ی زیادی رو اشغال کنه. هر کدوم از اون توابع فراخوانی شده(هر کدوم از توابعی که توسط خودش فراخوانی شده) نیاز به مقداری حافظه دارن. وقتی استک شما طولانی باشه، معنیش این میشه که کامپیوتر شما داره مقدار زیادی اطلاعات برای فراخوانی شدن اون توابع ذخیره میکنه. در اون مرحله، شما ۲ گزینه دارین:

- شما میتونین کدون رو بازنویسی کنین و بجاش از حلقه‌ها استفاده کنین.
- شما میتونین از مفهومی تحت عنوان "بازگشتی از انتهای" یا **tail recursion** استفاده کنید. این یک بحث پیشرفته در مبحث بازگشت به حساب میاد که خارج از بحث این کتابه. همچنین ، توسط بعضی از زبان‌ها پشتیبانی میشه نه همشون.

تمرین

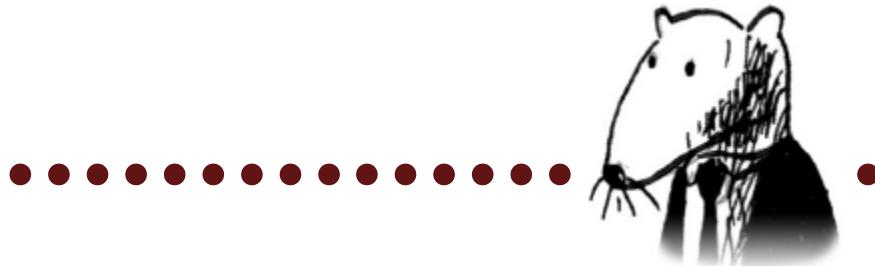
۳.۲ فرض کنین شما بر حسب اتفاق یک تابع بازگشتی نوشتن که تا ابد اجرا میشه. همونطور که دیدید کامپیوتر شما برای هر بار فراخوانی تابع توسط خودش یک فضا در مموری بهش روی استک اختصاص میده. چه اتفاقی برای استک میفته وقتی تابع بازگشتی در یک حلقه بینهایت افتاده؟

خلاصه این فصل

- بازگشت، وقتی اتفاق می‌افتد که تابع خودش را فراخوانی کند.
- هر تابع بازگشتی دو بخش دارد: بخش پایه و بخش بازگشتی.
- استک تنها دو عملیات دارد: اضافه کردن و برداشتن(**push and pop**)
- همه تابع‌هایی که فراخوانی می‌شنود به استک اضافه می‌شنود.
- کال استک می‌تواند خیلی بزرگ شود، که باعث اشغال حافظه زیادی می‌شود.



مرتب سازی سریع (Quicksort)



در این فصل:

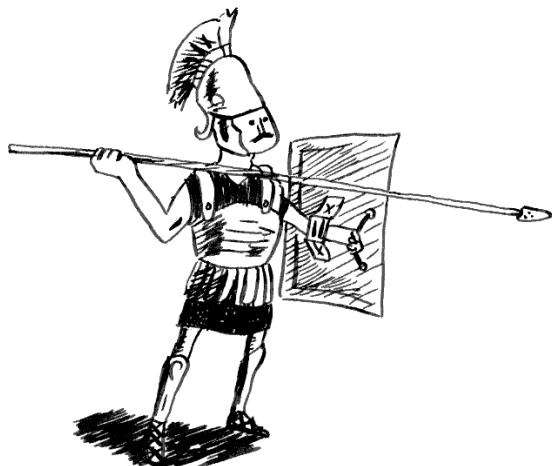
- شما مبحث تقسیم و تبخیر یادمیگرین. بعضی موقع شما با یک مسئله ای مواجه میشین که با هیچ کدام از الگوریتم هایی که یادگرفتین حل نمیشه. وقتی یک الگوریتمدان خوب به همچین مشکلی بر میخوره، زود تسلیم نمیشه. اونا یک جعبه ابزار پر از تکنیک دارن که برای حل اون مسئله، ازش استفاده میکنن تا به یک راه حل برسن. تقسیم و تبخیر (**devide and conquer**) اولین تکنیک کلی برای حل مسئلს، که شما یادمیگرین.
- شما همچنین درباره کوییک سورت یاد میگیرن، که الگوریتم قشنگ که اغلب در کارمون مورد استفاده قرار میگیره. کوییک سورت از بحث تقسیم و تبخیر برای کار خودش استفاده میکنه.



شما همه چیز رو درباره بازگشت در فصل قبل یادگرفتید. تمرکز در این فصل روی این موضوعه که شما از مهارت جدید که یادگرفتین برای حل مسائل استفاده کنین. ما در این بخش به سراغ تقسیم و تسخیر میریم و انا رو بررسی میکنیم (**devide and conquer(D&C)**).

در این فصل به طور جدی وارد الگوریتم ها میشیم. گذشته از همه اینها، یک الگوریتم اگر فقط بتوانه یک نوع مسئله را حل کنه واقعا کاربردی نیست. به جای همین، **D&C** (تقسیم و تسخیر) یک روش جدید برای فکر کردن در مورد حل مسئله بهتون ارائه میده. تقسیم و تسخیر، یک ابزار دیگه در جعبه ابزار شماست. وقتی شما با یک مسئله جدید رو به رو میشین، نباید نامید بشن و از پا دریابین. بجاش، میتوانین از خودتون بپرسین، "اگر از تقسیم و تسخیر استفاده کنم، میتونم این مسئله رو حلش کنم؟"

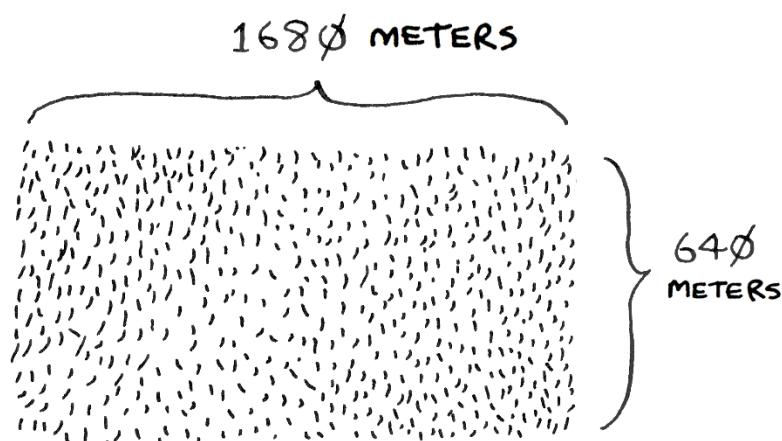
در آخر این فصل، شما اولین الگوریتم **D&C** اصلیتون رو یاد میگرید: کوییک سورت. کوییک سورت یک الگوریتم مرتب سازیه و از اون الگوریتم مرتب سازی انتخابی (که در فصل دو یادش گرفتین) خیلی سریعتره. این الگوریتم یک مثال از یک کد قشنگ و زیباست.



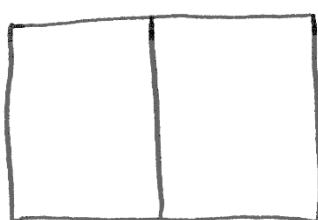
تقسیم و تسخیر

ممکنه بحث تقسیم و تسخیر رو کمی طول بکشه تا کامل درکش کنین. پس برای اینکار، ما سه تا مثال میزنیم. اول یک مثال تصویری هست رو نشونتون میدم. بعد از اون من یک نمونه کد بهتون نشون میدم که به اندازه کد اصلی قشنگ نیست اما ممکنه درکش راحت تر باشه. در آخر هم میریم سراغ کوییک سورت، یک الگوریتم مرتب سازی که از تقسیم و تسخیر استفاده میکنه.

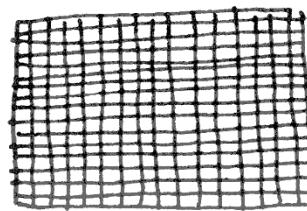
فرض کنید که شما یک کشاورز هستین و یک تکه زمین دارین.



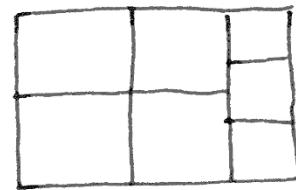
شما میخواین این قطعه زمین رو به طور مساوی (هم اندازه) به تیکه های مربع شکل تقسیم کنید. همچنین شما میخواین که این تیکه های تقسیم شده تا حد امکان در بزرگترین حالت خودشون باشن. پس با این تعریف هیچ کدوم از شکل هایی زیر بدرد نمیخورند و کارساز نیستند.



BOXES ARE
NOT SQUARE



BOXES ARE TOO
SMALL



ALL BOXES MUST
BE SAME SIZE

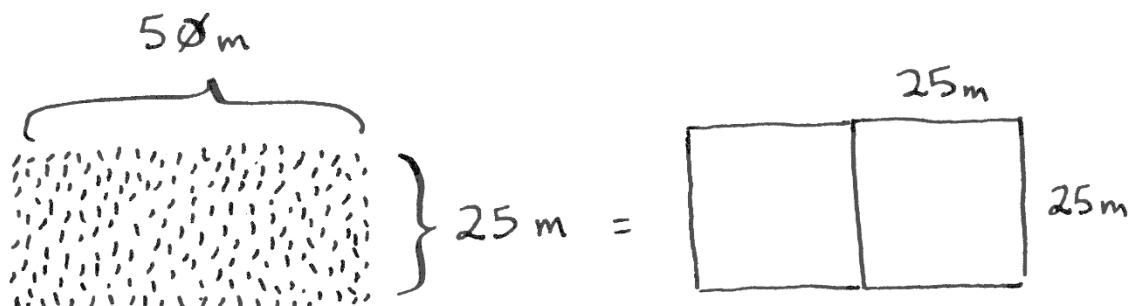
خب شما چطور میخواین بزرگترین سایز مربعی که میتوانیں برای تقسیم کردن زمین ازش استفاده کنین رو پیدا کنین؟ از استراتژی تقسیم و تسخیر (D&C) استفاده کنین، الگوریتم های تقسیم و تسخیر، جز الگوریتم های بازگشته هستند. برای حل این مسئله با استفاده از تقسیم و تسخیر ۲ قدم باید برداریم.

۱. پیدا کردن حالت پایه (**base case**) که این باید ساده ترین حالت ممکن باشه.

۲. مسئله تون رو اونقدر تقسیم کنین یا کاهش بدین که به حالت پایه برسین.

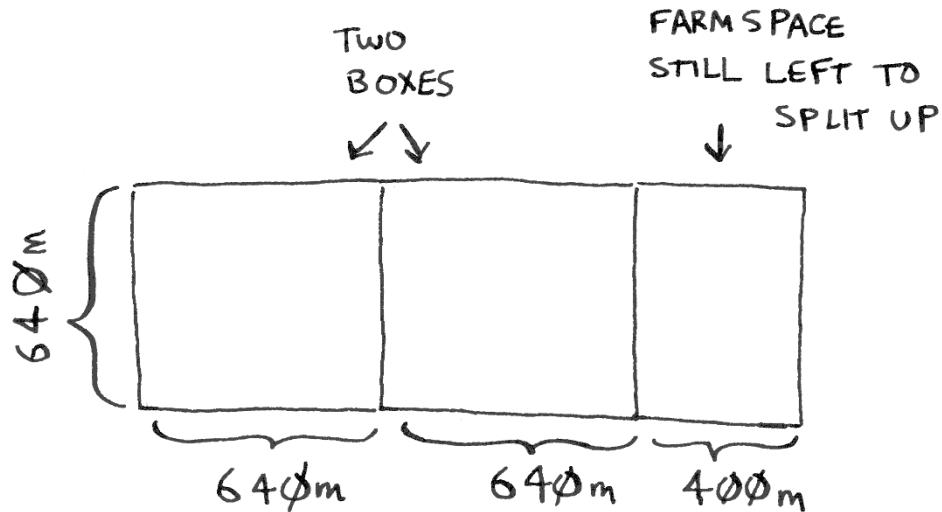
بیاین از D&C استفاده کنیم تا به راه حل این سوال برسیم. بزرگترین سایز مربعی که شما میتوانیں ازش استفاده کنین چیه؟

اول، حالت پایه رو پیدا کنین. راحترين حالت اين میتوانه باشه که يك ضلع ، ضریبی از يك ضلع دیگه باشه.

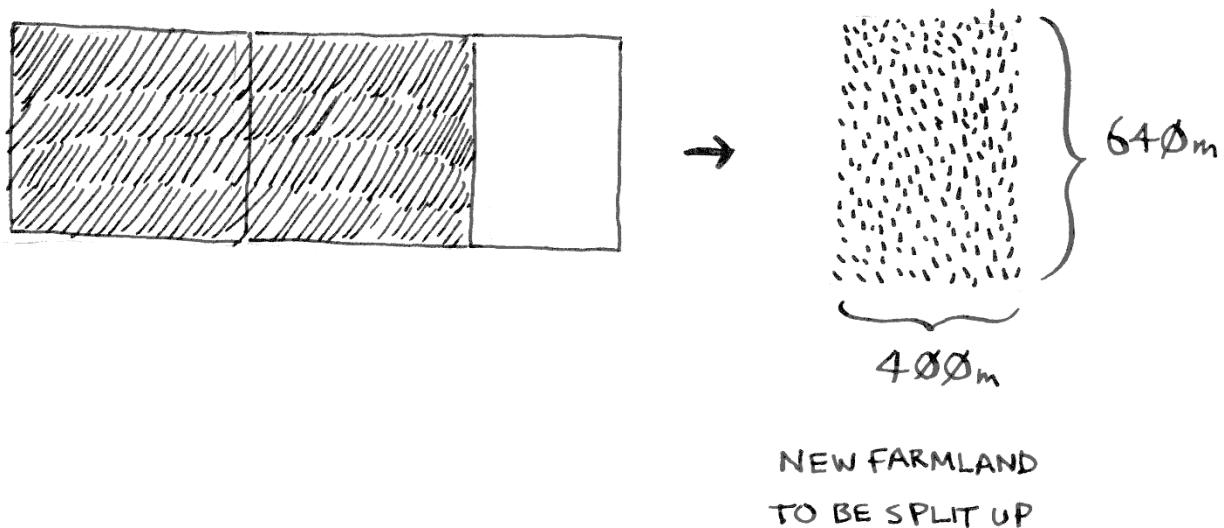


فرض کنین يك ضلع ۲۵ متر و ضلع دیگر ۵۰ متر است. پس درنتیجه بزرگترین مربع های مساوی که میتوانیں پیدا کنین، اندازشون برابر : $25m \times 25m$. پس شما به ۲ تا از این سایز مربع ها نیاز دارین تا بتوانین زمین رو تقسیم کنین.

حالا شم نیاز دارین که حالت بازگشتی رو پیدا کنیں(بخشی که باید حالت بازگشتی پیاده بشه رو پیدا کنیں). اینجا جایی که **D&C** وارد عمل میشه. براساس **D&C**، با هر بار فراخوانی بازگشتی محور، شما باید مسئله رو هر بار به تکه های کوچک تر کاهش بدین. حالا چطور این کارو بکنیم؟ بیاین با علامت گذاری و پیاده کردن بزرگترین مربعی که میتوانیم برای تقسیم زمین استفاده کنیم، شروع کنیم.



شما میتوانین دو بخش 640×640 در این زمین جا بدید. ولی همچنان مقداری زمین برای تقسیم کردن مونده. الان اون لحظه میاد که میگیم "آها!". الان یک بخش برای تقسیم مونده. حالا چرا همین الگوریتم رو برای این بخش باقی مانده اجرا نکنیم؟



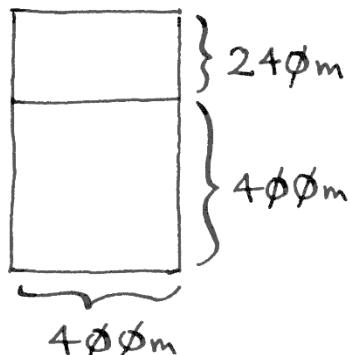
خب، شما با یک مزرعه 1640×1640 در 640×640 در 400×400 به بخش کوچک تر تقسیم بشه، شروع کردید. اما الان نیازه که اون بخش کوچک باقی مونده که اندازش 640×400 در 400×400 به بخش کوچک تر تقسیم بشه. اگر شما بزرگترین مربع رو در این بخش باقیمانده پیدا کردیدن که روی این بخش جواب بده، همین مربع پیدا شده در این بخش باقی مانده، میتوانه روی بخش های دیگه

زمین هم به عنوان بزرگترین مربع برای مزرعه شناخته بشه. شما الان مسئله رو از مزرعه 640×1680 متری به مزرعه 400×400 متری کاهش دادین.

الگوریتم اقلیدوس

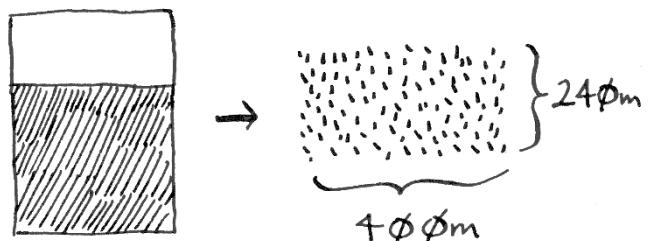
"اگر شما بزرگ ترین بخش رو طبق تعریف صورت مسئله (که میگه مربعی باشه) در این اندازه از تیکه زمین، پیدا کردید، اون بخش، بزرگترین بخش (مربعی) برای تمام مزرعه شناخته میشود." اگر برآتون واضح نیست که چرا این گزاره درسته، نگران نباشین چون واقعاً واضح نیست که چرا درسته. متاسفانه اثبات اینکه این گزاره چرا درسته یکم طولانیه که بخوایم تو این کتاب بیارمیش. پس فقط باید اینو از من قبول کنیم که این درسته. اگر میخواین بدلونین اثباتش چیه، درباره الگوریتم اقلیدوس جستجو کنین. خان آکادمی توضیحات خوبی توی این لینک داره:

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

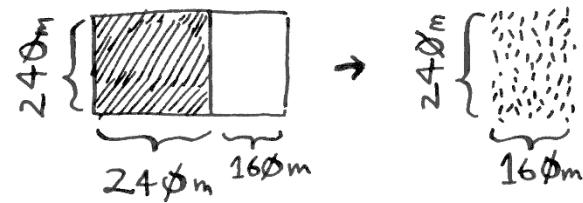


بیان دوباره همون الگوریتم رو بکار بگیریم. حالا با یک مزرعه‌ی 640×400 متر شروع میکنیم. بزرگترین بخش مربعی ای که میتوانیم درست کنیم 400×400 هستش.

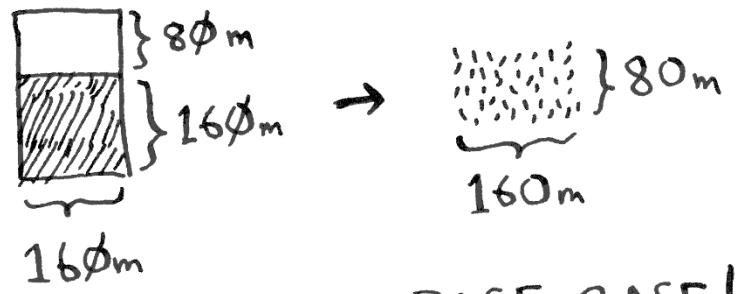
که از اون زمین هم یک بخش باقی میمانه که اندازه اش، 400×240 متر هستش.



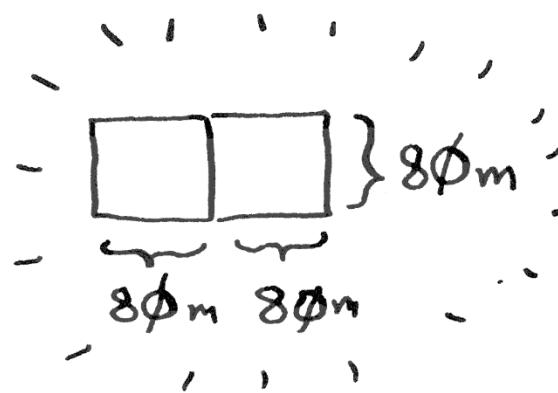
باز هم شما میتوانیم یک بخش از اون بخش باقی موند، بیرون بکشیم که که به یک بخش کوچک تر بررسیم.



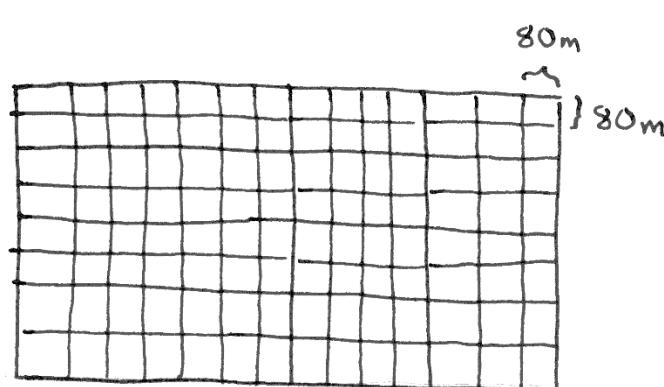
و دو باه میتوانیم یک بخش مربعی در بخش باقی مونده درست کنیم، تا به بخش کوچک تر برسین.



اوہ! شما به بخش و حالت پایه رسیدین: ۸۰×۸۰ ضریبی از ۱۶۰ هستش. اگر شما به بخش های مربعی که گفته شد تقسیم شون بکنیم، هیچ چیزی دیگه ای باقی نمیمانه!



پس برای مزرعه اصلیمون ، بزرگترین تیکه زمینی که میتوانیم برای تقسیم کردن به مربع های هم اندازه استفاده کنیم، اندازشون 80×80 هستش.



به طور خلاصه، این طرز کار **D&C** هستش:

۱. در نظر گرفتن حالت ساده به عنوان حالت و بخش پایه ای کار.
۲. پیدا کردن اینکه چطور مسئله رو کوچک تر و کوچک تر کنیم تا به حالت پایه بررسیم.

یک الگوریتم ساده نیست که برای مسئله ها بخواین ازش استفاده کنیم، بلکه یک طرز فکر برای حل مسئلش. بیاین مثال های بیشتری بزنیم.

2	4	6
---	---	---

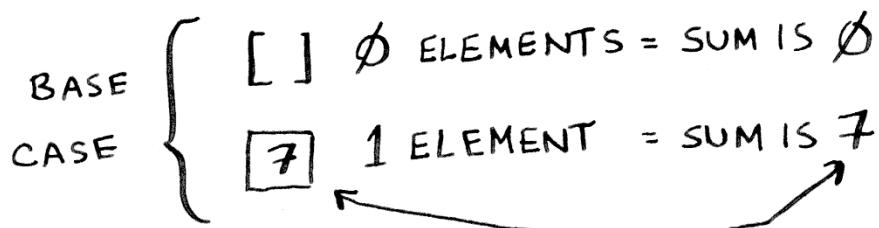
شما باید تمام اون اعداد رو باهم جمع کنید و نتیجه‌ی جمع کلشون رو برگردانیم.

این مسئله خیلی راحت با حلقه‌ها حل میشه:

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

اما با تابع بازگشتی این کارو چطور انجام میدین؟

قدم اول: پیدا کردن حالت پایه. ساده ترین آرایه‌ای که میتوانستین دریافت کنیم چی بود؟ درمورد ساده ترین حالت فکر کنیم و بعد به خوندن ادامه بدین. اگر شما آرایه‌ای شامل ۰ یا ۱ عنصر دریافت کنید. جمع کردنش خیلی آسون میشه.



خب پس همین چیزی که پیدا کردیم میشه بخش و حالت پایه ای کارمون.

قدم دوم: با هر بازگشت(منظور همون مفهوم بازگشتیه)، نیاز دارین که به یک آرایه‌ی خالی نزدیک تر بشین(تا به حالت پایه بررسین). چطور سایز مسئلتون رو کاهش میدین؟ این یک راهشه:

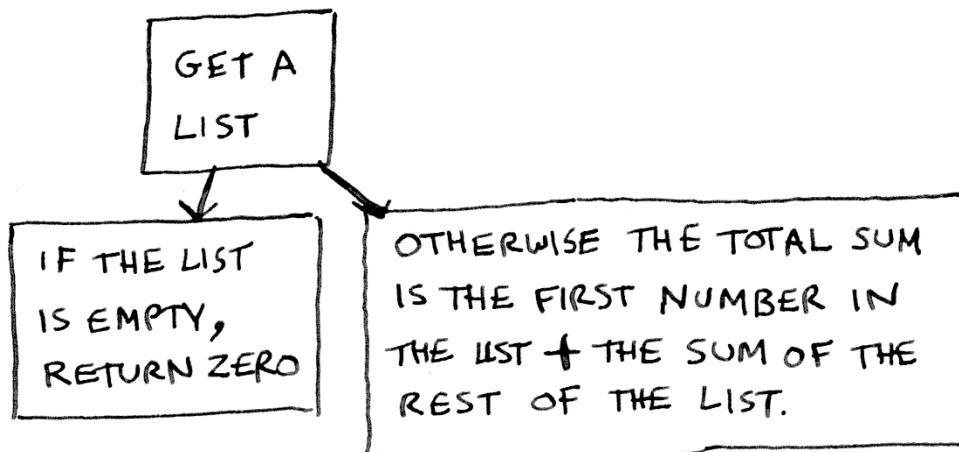
$$\text{sum}(\boxed{2 \mid 4 \mid 6}) = 12$$

و دقیقا مثل این راه میمونه:

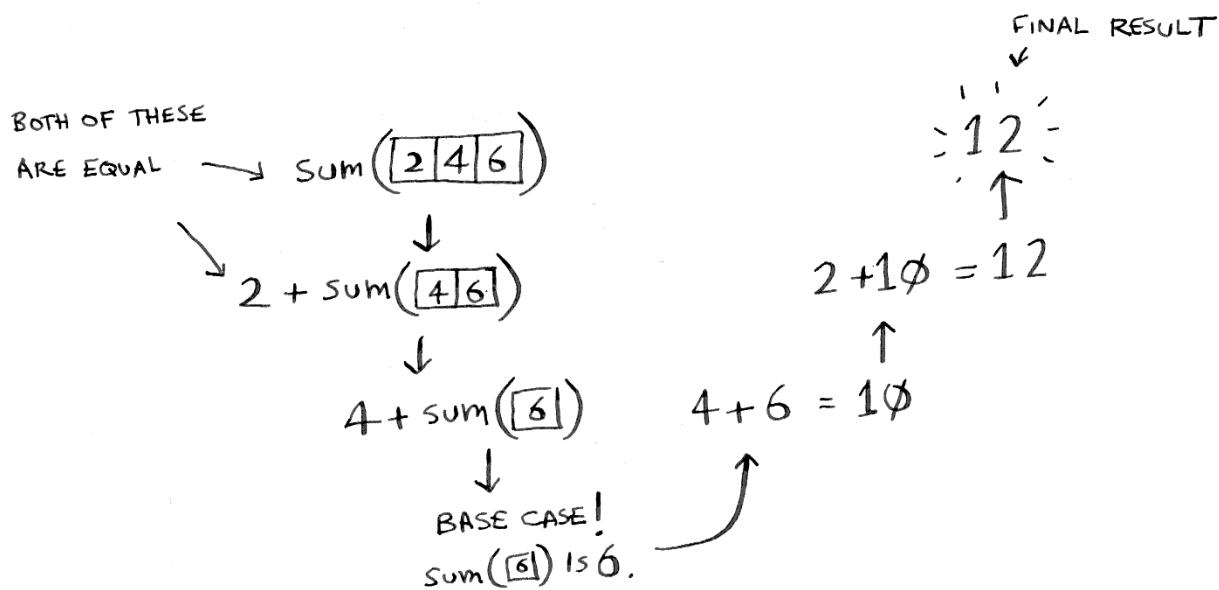
$$2 + \text{sum}([4|6]) = 2 + 1\phi = 12$$

در هر دو حالت، نتیجه ۱۲ است. اما در راه دوم، شما یک آرایه‌ی کوچک‌تر به تابع `sum` پاس میدین. که این یعنی شما سایز مسئله را کاهش دادین و کوچکش کردین.

تابع `sum` شما میتونه به این شکل کار کنه:



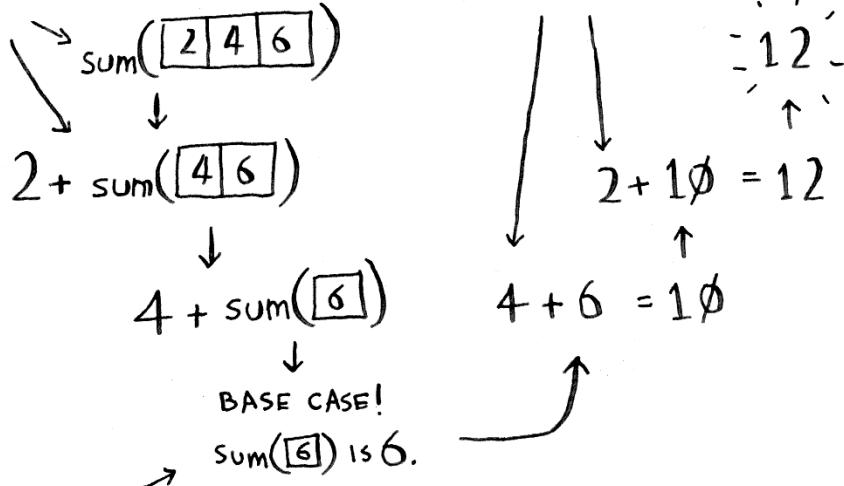
که در عمل به این شکل کار میکنه:



یادتون باشه، بازگشت (recursion) حواسش به وضعیت هست (میدونه چیارو حساب کرده چیارو نه، به لطف استک ها)

NONE OF THESE
FUNCTION CALLS
COMPLETE UNTIL
YOU HIT THE
BASE CASE!

REMEMBER, RECURSION
SAVES THE STATE FOR
THESE PARTIALLY COMPLETE
FUNCTION CALLS



THIS IS THE FIRST
FUNCTION CALL THAT
ACTUALLY COMPLETES

نکته

وقتی که در حال نوشتمن یک تابع بازگشته که شامل آرایه میشه، هستین، اغلب حالت پایه وقتی هست که آرایه یا خالی سمت یا دارای یک عنصر است. اگر در اینطور موارد گیر کردین، اول این موضوع رو امتحان کنید.

نیم نگاهی به برنامه نویسی تابعی (functional programming)

شما ممکنه فک کنین "چرا باید به صورت بازگشته انجامش بدم وقتی میتونم راحت از حلقه ها استفاده کنم؟". خب این بخش کوچک برای یک نیم نگاه به برنامه نویسی تابعیه. زبان های برنامه نویسی تابعی مانند هaskell (Haskell) حلقه ندارن (مفهومی تحت عنوان حلقه تو شون وجود نداره). پس شما باید از بازگشت برای نوشتمن توابعی مثل این (مثل قبلی) استفاده کنین. اگر در ک خوبی از بازگشت پیدا کرده باشید، راحت تر زبان های تابعی رو یادمیگرین. به عنوان مثال، این، طوری هستش که شما تابع `sum` رو در زبان هaskell مینویسین:

`Sum [] = .` ←----- Base case

`Sum (x:xs) = x + (sum xs)` ←----- Recursive case

توجه کنین که، به نظر میرسه دو تعریف برای اینتابع وجود دارد. اولین تعریف زمانی اجرا میشه که با بخش یا حالت پایه برخورد کنین. تعریف دوم وقتی که در حالت بازگشت هستیم اجرا میشه. شما همچنین میتوانین اینتابع رو در هسکل با استفاده از حالت شرطی پیاده کنین:

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

اما تعریف قبلی خوانا تر بود. به این دلیل که هسکل به شدت از ریکرزن (بازگشت) استفاده میکنه. این زبان هر نوع تکوتولوژی زیبا (و چیزهای خفنی) رو شامل میشه تا بکارگیری ریکرزن (بازگشت) رو ساده کنه. اگر از ریکرزن (بازگشت) خوشنون میاد یا علاقه به یادگیری یک زبان جدید دارین، حتما یک سری به هسکل بزنین.

تمرین ها

۴.۱ کد تابع `sum` که دربارش همین چند لحظه پیش دربارش حرف زدیم رو بنویسین.

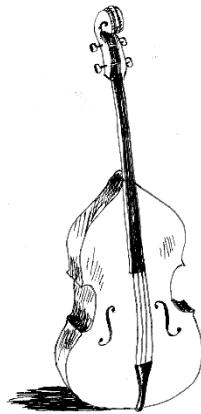
۴.۲ تابع بازگشتی بنویسید که تعداد آیتم های درون یک لیست رو بشماره.

۴.۳ بالاترین عدد رو در لیست پیدا کنین.

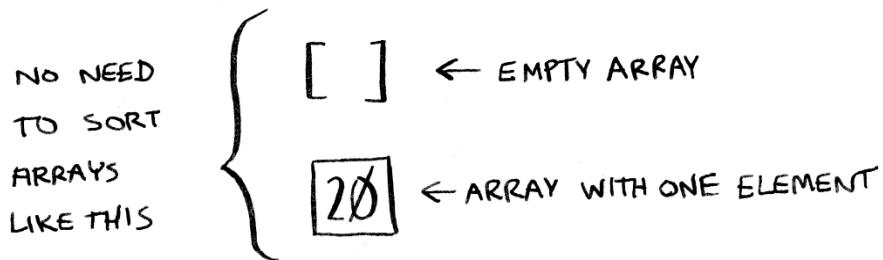
۴.۴ باینری سرج رو از فصل ۱ یادتون هست؟ اون هم یک الگوریتم تقسیم و تسخیره. آیا میتوانین بخش بازگشتی و بخش پایه ای اون رو مشخص کنین؟



کوییک سورت (Quicksort)



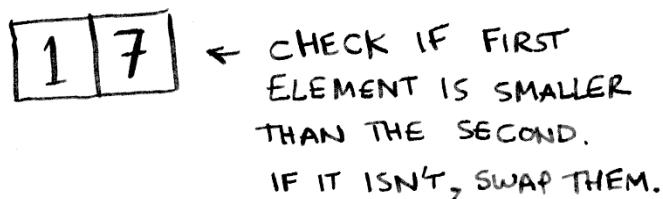
کوییک سورت یک الگوریتم برای مرتب سازی است. کوییک سورت خیلی سریع تر از الگوریتم مرتب سازی انتخابیه. و به طور مداوم در زندگی واقعیمون استفاده میشود. به عنوان مثال، کتابخانه استاندارد زبان C یک تابع داره که بهش `qsort` گفته میشود، که همون پیاده سازی شده دیگر کوییک سورت هستش. همچنین کوییک سورت از مفهوم تقسیم و تسخیر هم استفاده میکند. بیاین از کوییک سورت برای مرتب کردن یک آرایه استفاده کنیم. ساده تری آرایه ای که یک الگوریتم مرتب سازی میتوانه از پسش بربیاد چیه (نکته ای که در بخش قبلی بهتون گفتم یادتون هست؟). خب درواقع بعضی از آرایه ها اصلا نیازی به مرتب شدن ندارند.



آرایه خالی و آرایه ای شامل یک عنصر، حالت پایه خواهند بود. شما میتوانید اون آرایه هارو همنظر که هستند برگردونید. چون چیزی برای مرتب کردن وجود نداره.

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

بیاین به آرایه های بزرگتری نگاه بندازیم. مرتب سازی آرایه ای شامل دو عنصر هم خیلی سادس.



[33 | 15 | 10]

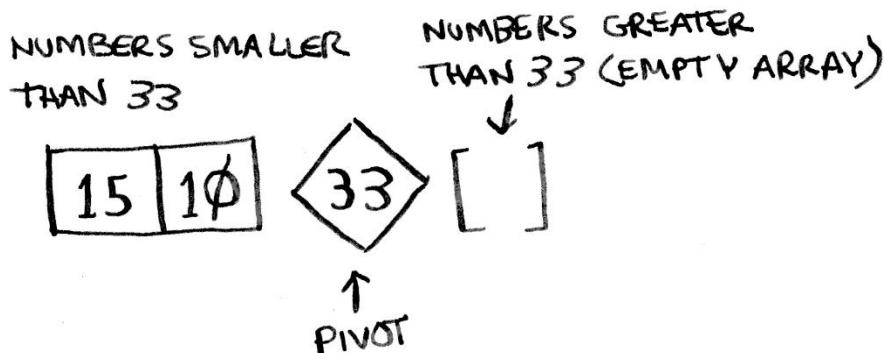
خب درمورد آرایه ای با سه عنصر چطور؟

یادتون باشه شما دارین از تقسیم و تسخیر استفاده میکنین. پس شما باید این آرایه رو به اونقدر خوردهش کنین تا به حالت پایه برسین. کوییک سورت اینطوری کار میکنه که، اول، یک عنصر از آرایه مدنظر انتخاب میکنه. به این عنصر انتخاب شده پیوت (pivot) گفته میشه.



درباره اینکه چطور یک پیوت خوب و مناسب انتخاب کنیم بعداً حرف میزنیم. اما الان بیاین فرض بکنیم که آیتم اول در آرایه، پیوت ماست.

حالا عناصری رو پیدا کنین که هم از پیوت کوچک ترند و هم ازش بزرگتر.

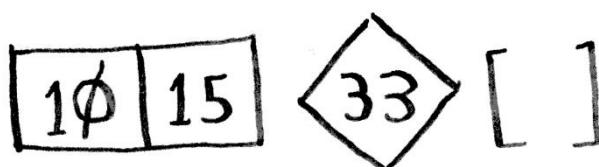


به این کار پارتیشن بندی کردن میگویند (بخش بندی کردن).

حالا شما:

- یک آرایه فرعی از تمام اعداد کوچک تر از پیوت دارین
- یک پیوت دارین
- یک آرایه فرعی از تمام اعداد بزرگ تر از پیوت دارین

اون دو آرایه‌ی فرعی هنوز مرتب نشدن. اوナ فقط پارتیشن بندی شدن. اما اگر (احياناً طبق روشی که بخش بندیشون کردیم) مرتب شده بودند، مرتب کردن کل آرایه (آرایه اصلی) کار ساده‌ای میبود.



اگر آرایه‌های فرعی، مرتب شده بودند، شما میتونسین همه چیز یه این شکل باهم ترکیب کنین و بهم بچسبونین: **left array + pivot + right array**. و شما به آرایه مرتب شدتون میرسید. در این حالت، آرایه‌هایی ما داریم به این شکل میشوند:

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

که یک آرایه‌ی مرتب شدش.

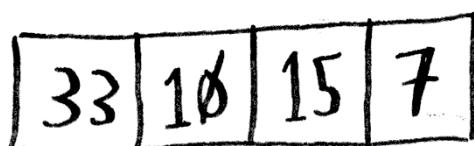
خب چطور آرایه‌های فرعی رو مرتب میکنین؟ خب، طبق حالت پایه‌ای کوییک سورت، که بررسی کردیم، از قبیل میدونیم که چطور آرایه‌ای شامل ۲ عنصر (آرایه سمت چپ (توی این مثال) و آرایه خالی (آرایه سمت راست) رو سورت کنیم. پس درنتیجه اگر کوییک سورت رو روی اون دو آرایه‌ی فرعی صدا بزنیں و در آخر جواب هارو بهم بچسبونیں، به یک آرایه مرتب شده میرسید.

```
quicksort([15, 10]) + [33] + quicksort([])  
> [10, 15, 33] <----- A sorted array
```

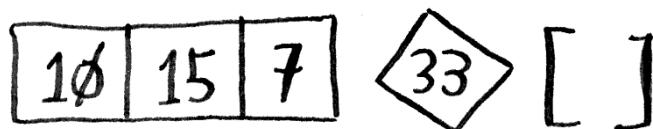
این روش با هر پیوتی که ما انتخاب کنیم کار میکنه. مثلاً فرض کنید ما ۱۵ رو به عنوان پیوت انتخاب کنیم. اون وقت هر دو آرایه‌ی فرعی هر کدام یک عنصر دارند و شما میدونیم که چطور باید اونها رو مرتب کنین. خب حالا شما میدونین چطور یک آرایه شامل ۳ عنصر رو مرتب کنین. این هم قدماًی که باید برای اینکار برداریم (به طور خلاصه):

۱. یک پیوت انتخاب کنید.
۲. آرایه رو به دو آرایه‌ی فرعی بخش بندی کنید: عنصرهای کمتر از پیوت و عنصرهای بیشتر از پیوت.
۳. کوییک سورت رو به صورت بازگشتی رو دو آرایه‌ی فرعی صدا بزنید.

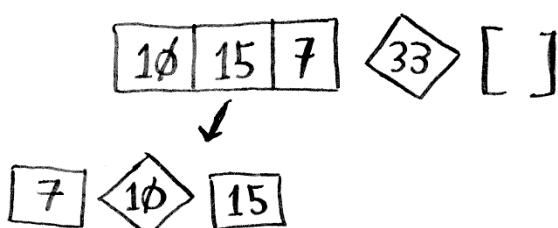
در مورد آرایه‌ای شامل چهار عنصر، این داستان چطور پیش میره؟



فرض کنین شما دوباره ۳۳ رو به عنوان پیوت، انتخاب میکنین.



آرایه (فرعی) سمت چپ، شامل سه عنصر مشه. خب شما از قبیل میدونین که چطور آرایه‌ای شامل سه عنصر رو سورت (مرتب) کنین: کوییک سورت رو روی اون به صورت بازگشتی فراخوانی میکنید.



پس میتوانیم (با همین روش) یک آرایه چهار عنصری رو هم مرتب کنیم. و میتوانیم نتیجه بگیریم اگر شما میتوانید آرایه ای شامل چهار عنصر رو مرتب کنید، پس میتوانید آرایه ای شامل پنج عنصر رو هم مرتب کنید. چرا اینطور نتیجه گیری کردم؟ فرض کنید شما آرایه ای شامل پنج عنصر دارید.

3	5	2	1	4
---	---	---	---	---

این هم تمامی انواع بخش بندی هایی هستش که شما میتوانید روی این آرایه با توجه به پیوتویی که انتخاب میکنید داشته باشید:

[]	◇ 1	3 2 1 4
-----	-----	---------------

1	◇ 2	3 5 4
---	-----	-----------

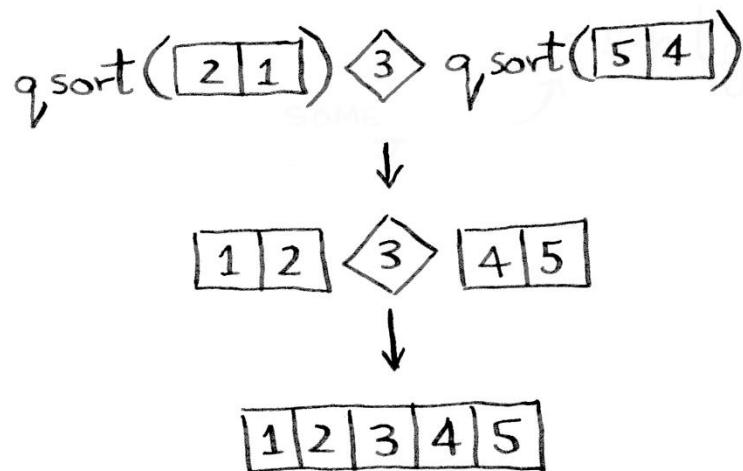
2 1	◇ 3	5 4
-------	-----	-------

3 2 1	◇ 4	5
-----------	-----	---

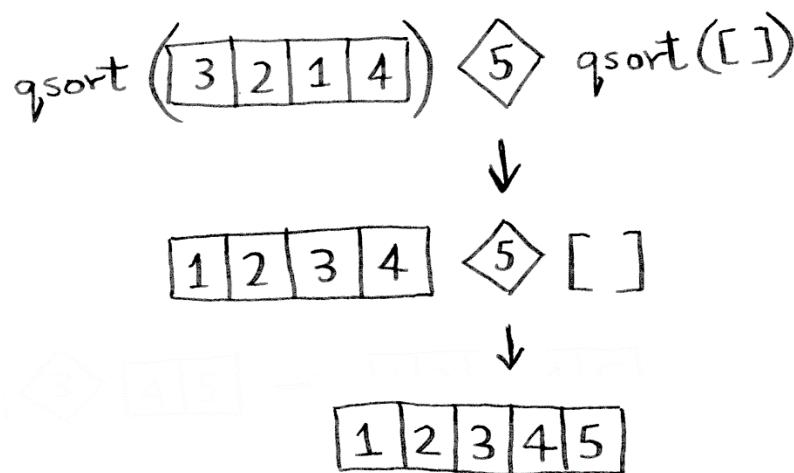
3 2 1 4	◇ 5	[]
---------------	-----	-----

توجه کنید که تمامی این آرایه های فرعی یک چیزی بین ۰ تا ۴ عنصر رو شامل میشن. و شما از قبل میدونید که چطور آرایه هایی شامل ۰ تا ۴ عنصر رو با استفاده از کوییک سورت مرتب کنید. پس اهمیتی نداره که شما چه پیوتویی رو انتخاب میکنید. شما میتوانید راحت کوییک سورت رو روی دو آرایه ای فرعی به صورت بازگشتی صدا بزنید.

به عنوان مثال. فرض کنیم شما ۳ رو به پیوست انتخاب میکنین. شما کوییک سورت رو روی آرایه های فرعی صدا میزنید.



آرایه های فرعی بعد از فراخوانی کوییک سورت، مرتب میشوند، و شما تمام اونها رو بهم میچسبونید و به یک آرایه ی مرتب شده ی کامل میرسین. این روش، حتی اگر شما ۵ رو به عنوان پیوست، انتخاب کنیم هم کار میکنه.



این روش با هر عنصری که شما به عنوان پیوست انتخاب کنین، کار میکنه. پس میتوانیم بازم نتیجه بگیریم که شما میتوانین آرایه پنج عنصری رو هم ، مرتب کنین. با استفاده از همین منطق، شما میتوانین آرایه ای شامل شش عنصر رو مرتب کنین و ... الی آخر.

اثبات استقرایی

خب شما همین الان به بخش "یه نگاه کوتاه به اثبات استقرایی" رسیدین. اثبات های استقرایی، راهی هستن که ثابت کنیم که الگوریتم ما کار میکنه. هر اثبات استقرایی شامل دو قدم میشه: حالت پایه و حالت استقرایی. بنظرتون آشنا نیومد؟ به عنوان مثال ، فرض کنین من میخواهم ثابت کنم که میتونم تا بالای یک نردبان بالا برم. در حالت استقرایی میگیم که، اگر پاهای من روی یک پله باشه، میتونم پامو روی پله بعدی بزارم. پس اگر فرض کنیم که من روی پله دوم هستم، میتونم به پله سوم برم. به این میگن حالت استقرایی. در حالت پایه اینطور میریم جلو که، من میگم، پاهای من روی پله اول قرارداره، بنابراین، میتونم کل نردبان رو بالا برم. هر بار یک پله.

شما از همین استدلال برای الگوریتم کوییک سورت استفاده میکنین. در حالت پایه، من بهتون نشون دادم که این الگوریتم چطور در حالت پایه کار میکنه: آرایه \cdot عنصری و 1 عنصری. در حالت استقرایی قضیه، من بهتون نشون دادم که اگر کوییک سورت داره برای آرایه یک عنصری جواب میده، روی آرایه دو عنصری هم جواب میده. و اگر برای آرایه 2 عنصری جواب میده، پس برای آرایه 3 عنصری هم جواب میده و الی آخر. و در آخر میتونم نتیجه گیری کنم که کوییک سورت برای همه آرایه ها و هر اندازه ای از آرایه کار میکنه و روی اونها جواب میده. زیاد وارد بحث اثبات استقرایی نمیشم، اما اونها یک مبحث جالب هستند و دست به دست مفهوم تقسیم و تسخیر پیش میرن.



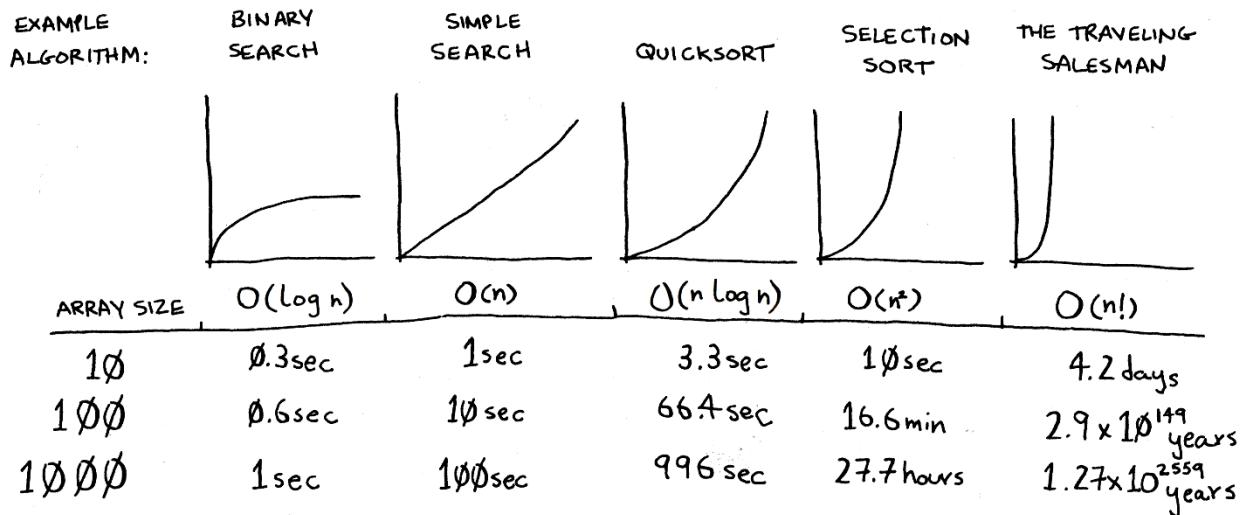
اینم از کد کوییک سورت:

```
def quicksort(array):
    if len(array) < 2:
        return array           Base case: arrays with 0 or 1 element are already "sorted."
    else:
        pivot = array[0]       Recursive case
        less = [i for i in array[1:] if i <= pivot]      Sub-array of all the elements
                                                               less than the pivot
        greater = [i for i in array[1:] if i > pivot]     Sub-array of all the elements
                                                               greater than the pivot
        return quicksort(less) + [pivot] + quicksort(greater)

print quicksort([10, 5, 2, 3])
```

نگاهی دوباره به بیگ او نو تیشن

کوییک سورت یک الگوریتم خاص و منحصر به فرد ه چرا که سرعتش به پیوتوی (pivot) ما انتخاب میکنیم بستگی دارد. قبل از اینکه درباره کوییک سورت صحبت کنیم، بیاین دوباره یک نگاهی به رایج ترین زمان های اجرایی بیگ او بندازیم.



این آمار براساس کامپیوتر های کندي است، که در ثانیه تنها قادر به انجام ۱۰ عملیات هستند.

این نمودار ها خیلی دقیق نیستند، فقط اونها رو آوردم که درک و لمس کنین که زمان های اجرایی شون چقدر باهم متفاوت هستند. در واقعیت، کامپیوتر شما خیلی بیشتر از ۱۰ عملیات در ثانیه میتونه انجام بدی.

در هر کدوم از این زمان های اجرایی هم، یک مثال از الگوریتم مربوطش ضمیمه شده. مثلا الگوریتم مرتب سازی انتخابی رو بررسی کنین، که در فصل ۲ یادش گرفتیم. زمان اجراییش $O(n^2)$ هستش، که زمان اجرایی نسبتا کنده. یک الگوریتم مرتب سازی دیگه ای به نام مرتب سازی ادغامی (merge sort) (مرج سورت) وجود داره که زمان اجراییش $O(n \log n)$ هستش که خیلی سریعتره. کوییک سورت یک ذره گمراه کنندس (زمان اجراییش). چرا که در بدترین حالت، زمان اجراییش میتونه به اندازه $O(n^2)$ طول بکشه.

این زمان به اندازه، زمان اجرایی الگوریتم مرتب سازی انتخابی (سلکشن سورت)، کنده! اما خب بدترین حالت رو در نظر گرفتیم. در حالت میانگین، کوییک سورت، به اندازه $O(n \log n)$ طول میکشه. خب شما ممکنه براتون سوال بشه که:

- منظور از بدترین حالت و بهترین حالت دراینجا چیه؟

- اگر کوییک سورت تنها در حالت میانگین به اندازه $O(n \log n)$ طول میکشه، اما مرج سورت (`merge sort`) همیشه و در هر حالتی به اندازه $O(n \log n)$ طول میکشه، چرا از مرج سورت استفاده نکنیم؟ اینطوری سریع تر نیست؟

مرج سورت VS کوییک سورت

فرض کنیم شما اینتابع ساده را برای چاپ کردن تمام آیتم های داخل یک لیست، در اختیار دارین:

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

این تابع به سراغ تمام آیتم های داخل لیست میره و تمام اونها را چاپ میکنه. دلیلش هم اینکه یک بار روی کل لیست حلقه میزنن. این تابع در زمان $O(n)$ اجرا میشه. حالا فرض کنیم ، شما این تابع را طوری تغییر میدین که قبل از چاپ هر آیتم ۱ ثانیه صبر کنه.

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

پس چی شد. قبل از چاپ هر آیتم ۱ ثانیه توقف میکنه. حالا فرض کنیم با استفاده از هر دو تابع ، میخواین لیستی شامل پنج آیتم را چاپ کنیم.

2	4	6	8	10
↓				

`print_items`: 2 4 6 8 10

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

هر کدوم از اون دو تابع، یکبار روی کل لیست حلقه میزنن. پس زمان اجرایی جفتشون میشه $O(n)$. فکر میکنیں کدومشون در عمل از اونیکی سریع تر هستند؟ من فکر میکنم `print_items` خیلی سریعتر باشه، چراکه قبل چاپ هر آیتم توقف ۱ ثانیه ای نداره. با وجود اینکه سرعت جفتشون در تعریف بیگ او نویشن، یکی هست. اما `print_items` خیلی در عمل سریعتره. وقتی شما بیگ او نویشن رو به این شکل مینویسین: $O(n)$. در واقع این معنی رو میده:

C * n
 ↑
 SOME ↑
 FIXED AMOUNT
 OF TIME

C یعنی مقدار مشخصی از زمان ، که الگوریتم شما برای اجرا نیاز دارد. به این C "ثابت (constant)" میگن. به عنوان مثال، ممکنه برای print_items ۱۰ milliseconds * n: باشد

و برای print_items ۱ second * n: اینقدر باشد:

شما معمولا از اون ثابت، صرف نظر میکنین. چرا که اگر دو الگوریتم ، دو زمان بیگ او متفات داشته باشن، اون ثابت دیگه اهمیتی نداره. برای مثال، باینری سرچ و سرچ ساده رو درنظر بگیرین. فرض کنین جفت این الگوریتم ها این ثابت هارو دارن:

$10 \text{ ms} * n$ <hr/> SIMPLE SEARCH	$1 \text{ sec} * \log n$ <hr/> BINARY SEARCH
--	---

شاید بگین که ، "واو! ثابت سرچ ساده ۱۰ میلی ثانیس، اما ثابت باینری سرچ ۱ ثانیس. پس سرچ ساده خیلی سریعتره!" حالا فرض کنین که با همین شرایط، شما میخوان یک لیست شامل ۴ میلیارد عنصر رو سرچ کنین. این مقدار زمان هایی که طول میکشه:

SIMPLE SEARCH	$ $	$10 \text{ ms} * 4 \text{ BILLION} = 463 \text{ days}$
BINARY SEARCH	$ $	$1 \text{ sec} * 32 = 32 \text{ seconds}$

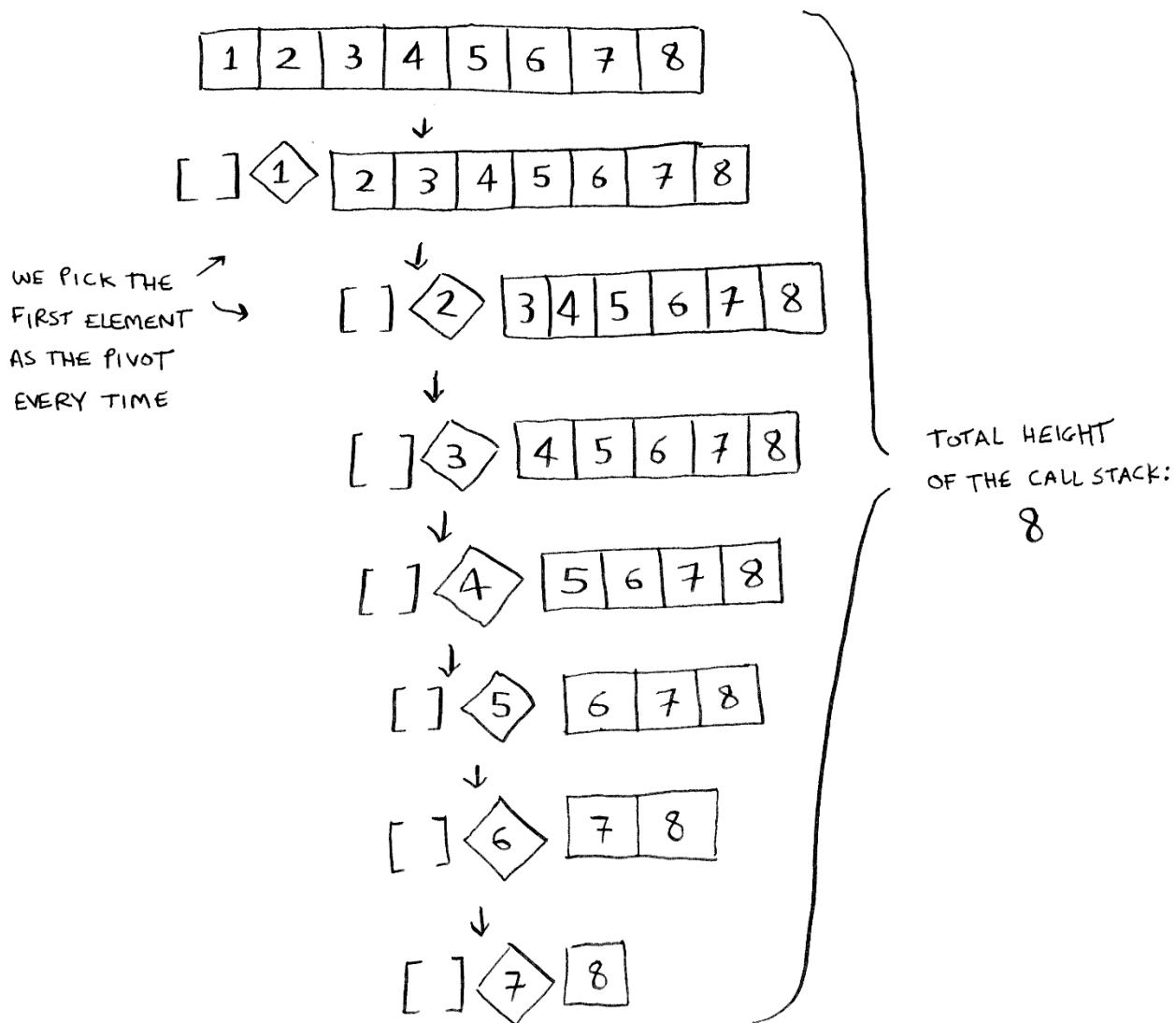
همونطور که میبینین، باینری سرچ هنوز خیلی سریعتره. اون ثابت اصلا تغییری ایجاد نکرد.

اما بعضی موقع ، این ثابت، میتونه یک تغییر ایجاد کنه. کوییک سورت و مرچ سورت یکی از این مثال هاست. کوییک سورت ثابت کمتری، نسبت به مرچ سورت دارد. پس اگر بیگ او جفتشون $O(n \log n)$ باشد، در اون حالت کوییک سورت سریعتره. و در کنار اون، کوییک سورت در عمل سریعتره، چرا که اکثر اوقات با حالت میانگین مواجه میشه تا بدترین حالت.

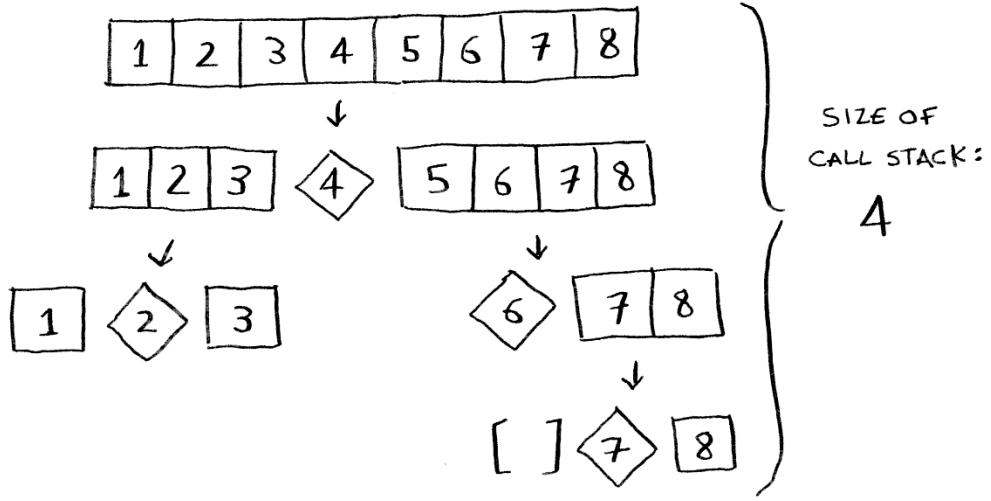
خب ممکنه برآتون سوال پیش بیاد که : حالت میانگین و بدترین حالت چیا هستن؟

حالت میانگین vs بدترین حالت

نحوه عملکرد کوییک سورت ، به شدت به نحوه انتخاب پیوب بستگی داره. بیاین فرض کنیم شما همیشه عنصر اول را انتخاب به عنوان پیوت انتخاب میکنید. و شما کوییک سورت رو روی آرایه ای که قبلا مرتب شده، فراخوانی میکنین. کوییک سورت نمیاد چک کنه که آیا آرایه ای که به عنوان ورودی میگیره مرتب شده س، یا نه. پس همچنان تلاش میکنه تا (به روش خودش) مرتبش کنه.



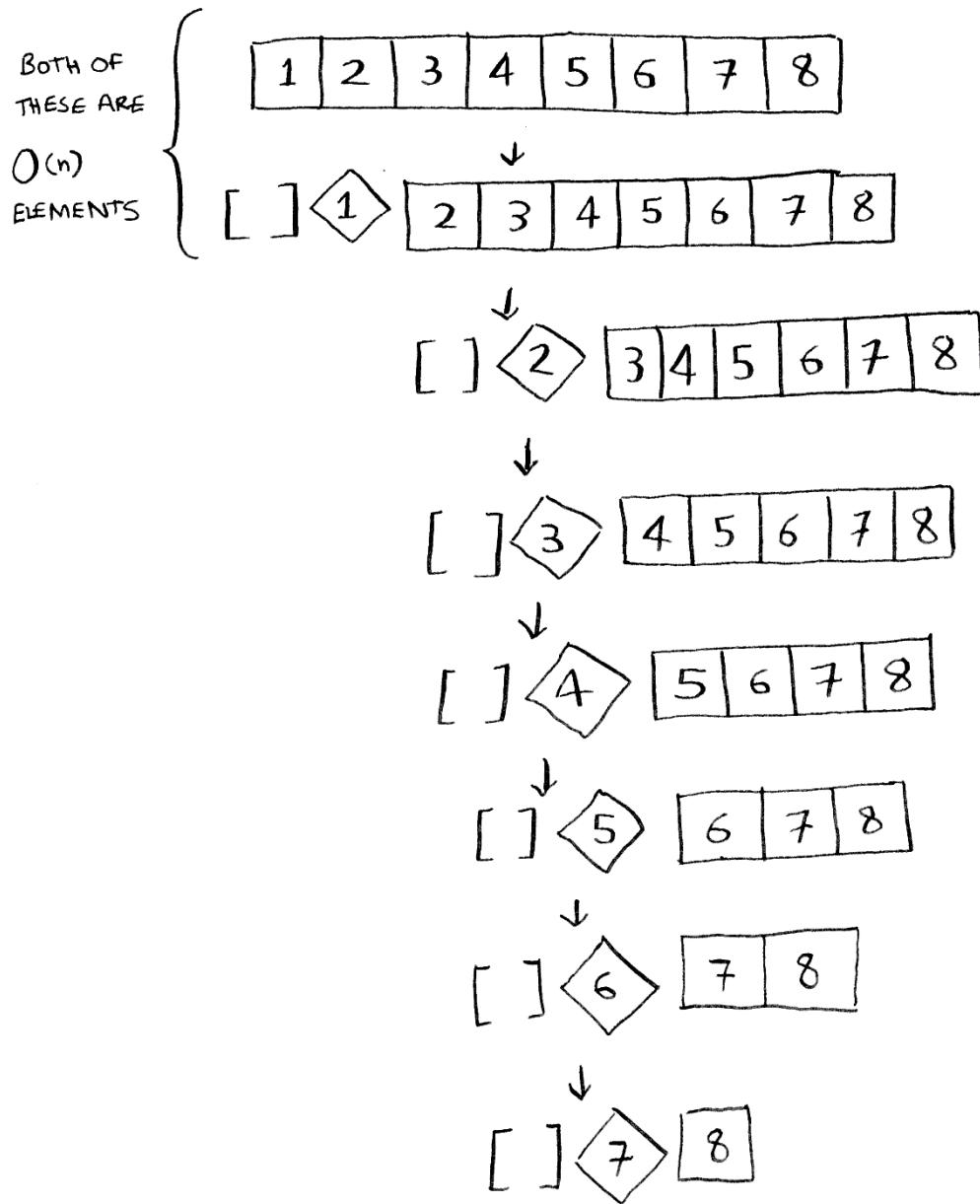
متوجه شدین که شما با انتخاب عنصر اول در این آرایه مرتب شده و انتخاب کردن پیوت نامناسب، درواقع آرایه رو به دو بخش تقسیم نکردین، چرا که یکی از آرایه های فرعی به کل تا آخر این الگوریتم خالیه. درنتیجه کال استک ما، خیلی طولانی میشه. حالا بیاین فرض کنیم، که شما همیشه عنصر وسط رو به عنوان پیوت در نظر میگیرین. حالا به کال استک اون نگاه کنین.



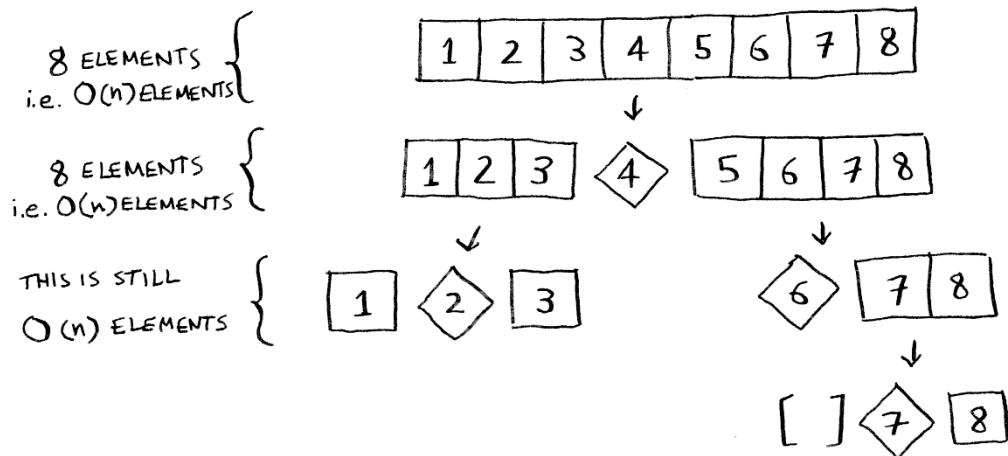
خیلی نسبت به قبلی کوتاه تر! به این دلیل که، شما هر بار آرایه رو به دو قسمت تقسیم میکنین، که دیگه نیازی نیست به اندازه قبل اونقدر، این الگوریتم رو به طور بازگشتی صدا بزنیم. اینجا، در این حالت، شما زود تر به حالت پایه میرسین و کال استک خیلی کوتاه تر میشه.

اولین مثالی که دیدین یک سناریو از بدترین حالت هستش، و مثال دوم بهترین سناریو هستش. در بدترین حالت اندازه استک برابر $O(n)$. اما در بهترین حالت، اندازه استک برابر $O(\log n)$.

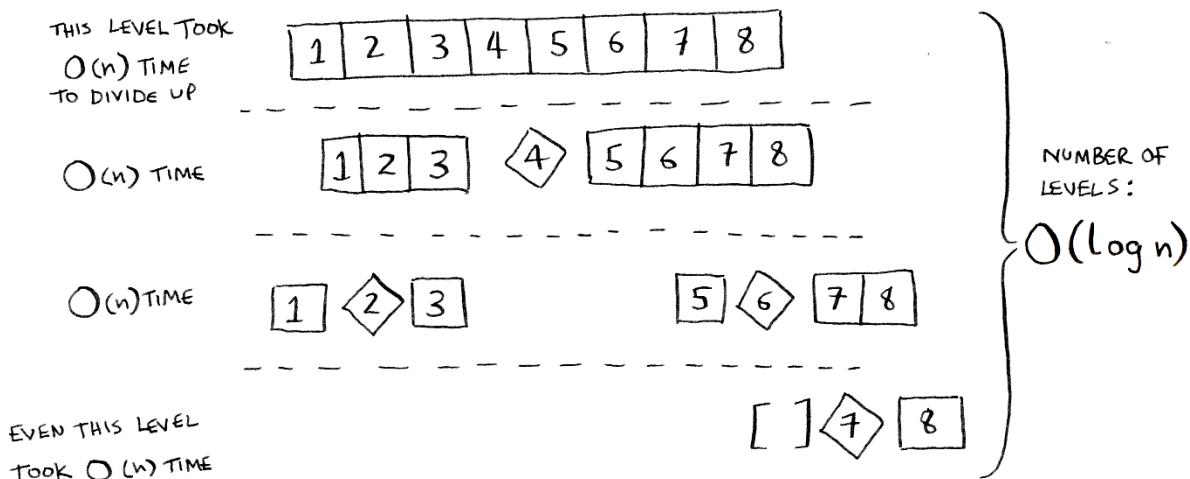
حالا به اولین مرحله در استک توجه کنین. شما عنصر اول رو به عنوان پیوتو انتخاب میکنین، و بقیه عناصر، به دو آرایه فرعی تقسیم میشن. شما هر هشت عنصر رو در آرایه بررسی میکنین. پس اولین حرکت ما به اندازه $O(n)$ طول میکشه. شما تمام هشت عنصر رو در این مرحله از استک بررسی کردین. اما درواقع شما هر بار $O(n)$ عنصر رو در هر مرحله از کال استک بررسی میکنین. (این بده)



حتی اگر هم به طور متفاوتی این آرایه رو بخش بندی کنیں، همچنان هر بار مقدار $O(n)$ عنصر رو بررسی مکنین.



خب پس هر مرحله به اندازه $O(n)$ طول میکشه تا تکمیل بشه.



در این مثال، $O(\log n)$ مرحله وجود داره (اگر به طور تکنیکی بخوایم بگیم اینطوری میشه که: "طول کال استک برابر $O(\log n)$ هستش"). و هر مرحله به اندازه $O(n)$ زمان میبره. زمان کل الگوریتم میشه $O(n) \times O(\log n) = O(n \log n)$.

در بدتری حالت، $O(n)$ مرحله وجود داره، پس زمان الگوریتم میشه: $O(n) \times O(n^2) = O(n^3)$

خب حدس بزنین چی شده؟ من اینجام تا بہتون بگم، بہترین حالت همون حالت میانگین هستش. اگر همیشه یک عنصر رندوم از آرایه رو به عنوان پیوتوت انتخاب کنیں، کوییک سورت در زمان $O(n \log n)$ که زمان میانگین هست، کارش کامل انجام میشه. کوییک سورت یکی از سریعترین الگوریتم های مرتب سازیه. و یک مثال خیلی خوب برای بحث D&C (تقسیم و تسخیر) هستش.

تمرین ها

براساس بیگ او نوتیشن، هر کدام از عملیات های زیر چقدر طول میکشند؟

۴.۵ چاپ کردن مقادیر همه عناصرهای یک آرایه.

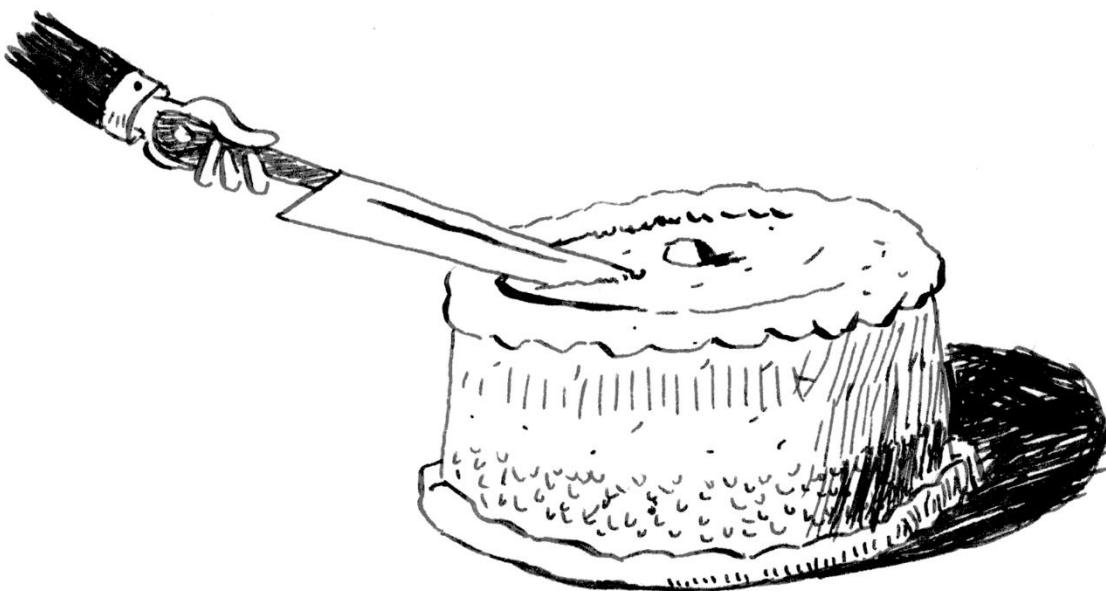
۴.۶ دوباره کردن مقادیر هر کدام از عناصرها در یک آرایه.

۴.۷ دو برابر کردن مقدار اولین عنصر در آرایه.

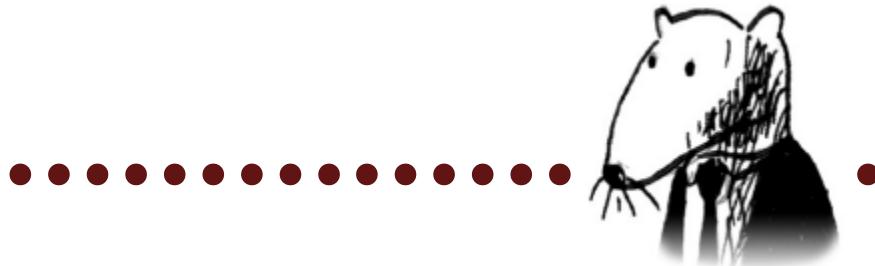
۴.۸ درست کردن جدول ضرب با مقادیر تمام عناصر در یک آرایه. به این شکل که، اگر شما یک آرایه $[10, 8, 7, 3, 2]$ داشته باشید، اول باید تمام عناصر را در ۲ ضرب کنید، سپس تمام عناصر را در ۳ ضرب کنید، و بعد از آن تمامی مقادیر را در ۷ ضرب کنید و تا آخر به همین شکل.

خلاصه این فصل

- D&C (تقسیم و تسخیر) به این روش کار میکنه که، مسئله رو به قطعه های کوچک و کوچک تر خورد میکنه. اگر دارین از D&C روی یک لیست استفاده میکنین، احتمالاً حالت پایه شما یک آرایه خالی یا یک آرایه شامل یک عنصر باشد.
- اگر دارین از کوییک سورت استفاده میکنین. یک عنصر رندوم (تصادفی) رو به عنوان پیوست انتخاب کنید. زمان اجرایی متوسط کوییک سورت برابر $O(n \log n)$.
- بعضی موقع ها وجود ثابت در بیگ او نوتیشن میتونه اهمیت داشته باشه. به همین دلیله که کوییک سورت از مرج سورت سریع تره.
- ثابت ها تقریباً اصلاً در سرج ساده مقابله میکنند. سرج اهمیتی ندارند. چرا که $O(\log n)$ به شدت از $O(n)$ وقتی لیست موردنظر برای بررسی بزرگ و بزرگ تر میشه، سریعتره.



جداول هش (Hash tables)



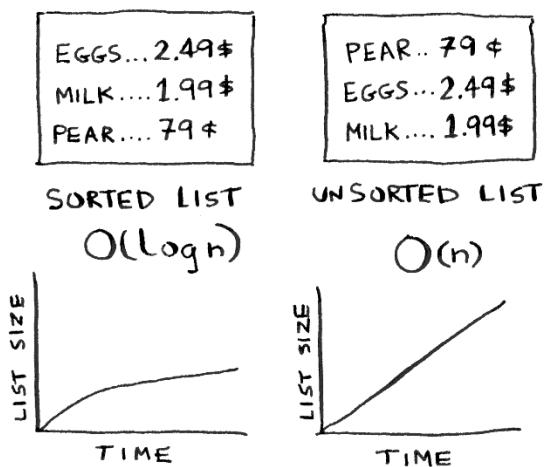
در این فصل:

- در مورد هش تیبل(جداول هش) یادمیگرین. یکی از پرکاربردترین ساختار داده های اساسی. جداول هش، استفاده های زیادی دارن. این فصل ، موارد استفاده رایج جداول هش رو پوشش میده.
- با اجزای داخلی جداول هش آشنا میشین: کاربردشون، مبحث تلاقی، و توابع هش. این به شما در درک تحلیل عملکرد جداول هش کمک میکنه.





فرض کنین شما در یک مغاره خواربار فروشی کار میکنید. وقتی مشتری، محصولی رو میخره، شما باید قیمت اون رو از توی یک کتاب پیدا کنید. اگر اون کتاب براساس الفا مرتب نشده باشه، پیدا کردن محصولی (به عنوان مثال سیب)، میتونه خیلی زمان بر باشه. درواقع شما دارین از سرچ ساده که در فصل ۱ باهاش آشنا شدیم استفاده میکنین، که در اون مجبورید تمام خط ها را در کتاب بررسی کنید. یادتون چقدر طول میکشید؟ به اندازه $O(n)$. اگر کتاب براساس الفبا مرتب شده باشه، شما میتونین از باینری سرچ برای پیدا کردن قیمت سیب، استفاده کنین. که تنها به اندازه $O(\log n)$ طول میکشه.



برای یادآوری، فرق زیادی بین زمان $O(n)$ و $O(\log n)$ وجود داره! خب بیاین فرض کنیم شما میتونین ۱۰ خط از کتاب رو در ۱ ثانیه بخونین. اینم جدولی که نشون میده سرچ ساده و باینری سرچ چقدر طول میکشن.

# OF ITEMS IN THE BOOK	$O(n)$	$O(\log n)$
100	10 sec	1 sec $\leftarrow \log_2 100 = 7 \text{ LINES}$
1000	1.66 min	1 sec $\leftarrow \log_2 1000 = 10 \text{ LINES}$
10000	16.6 min	2 sec $\leftarrow \log_2 10000 = 14 \text{ LINES}$ = 2 sec

شما از قبل میدونین که باینری سرچ به شکل وحشینای سربعتره. اما به عنوان صندوقدار، به دنبال قیمت چیزی داخل کتاب گشتن خیلی زجر آوره، حتی اگر هم محصولات در کتاب، به ترتیب و مرتب شده باشن. میتونین حس کنین وقتی دارین دنبال آیتمی داخل کتاب میگردین، مشتری داره جوش میاره. چیزی شما نیاز دارین، یک رفیقه خوبه که تمام اسم محصولات با قیمتاشون رو حفظه. و بعدش دیگه نیاز نیست دنبال چیزی بگردین، شما از اون رفیقتون میپرسین و اون بلافصله جواب رو به شما میده.



رفیق شما که اسمش مگی باشه، میتونه قیمت هر آیتمی رو در زمان $O(1)$ به شما بده، اصلاً اهمیت نداره که کتاب قیمت‌ها چقدر بزرگ باشه. مگی حتی از باینری سرچ هم سریعتره.

# OF ITEMS IN THE BOOK	SIMPLE SEARCH	BINARY SEARCH	MAGGIE
100	$O(n)$	$O(\log n)$	$O(1)$
1000	10 sec	1 sec	INSTANT
10000	1.6 min	1 sec	INSTANT
100000	16.6 min	2 sec	INSTANT

چه شخص قدر تمدنی! چطور میشه به مگی دست پیدا کرد؟

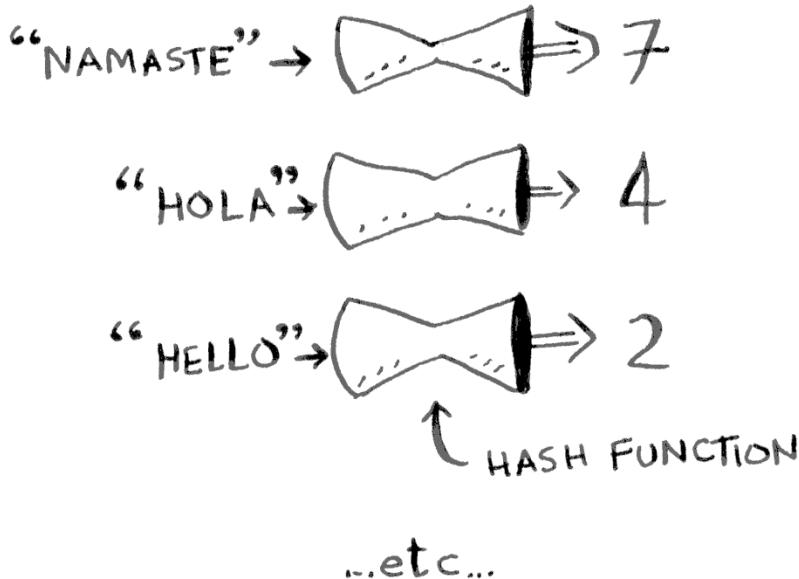
بیاین کلاه ساختار دادمون رو بازاریم سرمون. تا الان شما، دو تا ساختار داده رو میشناسین. آرایه‌ها و لیست‌ها درمورد استک‌ها حرفی نمیزنم چون واقعاً نمیشه تو شون سرچ زد). شما میتونین از این کتاب به عنوان آرایه استفاده کنین (کتاب لیست قیمت‌های فروشگاه).

(EGGS, 2.49)	(MILK, 1.49)	(PEAR, 0.79)
--------------	--------------	--------------

هر آیتم در آرایه، در واقع دو آیتم هستش: یکی اسم اون محصول، دیگری قیمت‌ش. اگر این آرایه رو بر اساس اسم مرتبش کنین، میتونین از باینری سرچ برای پیدا کردن قیمت یک محصول استفاده کنین. درنتیجه، شما میتونین آیتم‌ها رو در زمان $O(\log n)$ پیدا کنید. اما شما میخواین آیتم‌هارو در زمان $O(1)$ پیدا کنین. بخاطر همین هم هست که میخواین یک مگی برای خودتون درست کنین. اینجا جایی که توابع هش وارد عمل میشن.

تابع هش

تابع هش، تابعیه که یک رشته ورودی از شما میگیره و یک عدد برمیگردونه.(در اینجا ، منظور از رشته یک نوع داده هستش(دنباله ای از بایت ها)).



در اصطلاح فنی ، میگیم که تابع هش، رشته ها را به اعداد نگاشت میکند(ربط میدهد). ممکنه فکر کنید الگوی قابل تشخیصی برای اینکه در ازای رشته ورودی چه عددی خروجی میدهد، وجود ندارد. اما یکسری پیش نیاز ها برای تابع هش وجود داره:

- باید ثابت باشه و تغییری نداشته باشه. به عنوان مثال ، فرض کنین شما به عنوان ورودی بپرس کلمه "سیب" رو میدین و برآتون ۴ رو برمیگردونه. هر بار که شما کلمه سیب رو میدین باید ۴ برگردونه. بدون این، جدول هش شما کار نخواهد کرد.
- این تابع باید کلمات متفاوت رو به اعداد متفاوت نگاشت کنه. به عنوان مثال، یک تابع هش اگر همیشه ۱ رو به ازای هر کلمه ای برگردونه، اصلا خوب نیست. در بهترین حالت، هر کلمه ای متفاوتی به عدد متفاوتی نگاشت میشه(ربط داده میشه).

پس تابع هش، رشته رو به عدد نگاشت میکنه(ربط میده). خب حالا این ، برای چه کاری خوبه؟ خب ، شما میتوانین از این ، برای ساختن مگی خودتون استفاده کنین!

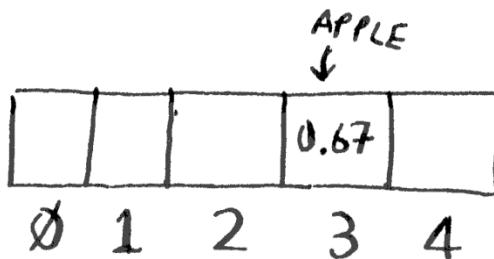
با یک آرایه ای خالی شروع میکنیم:

∅	1	2	3	4

شما تمام قیمت هاتون رو در این آرایه ذخیره میکنین. بیاين قیمت سیب رو به این آرایه اضافه کنیم. کلمه "سیب" رو به خورد تابع هش پاسش میدیم).

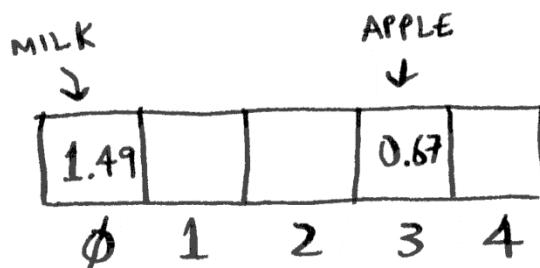


تابع هش، مقدار ۳ رو در خروجی به ما نشون میده. خب بیاين قیمت یک سیب رو در ایندکس شماره ۳ در این آرایه ذخیره کنیم.



بیاين "شیر" رو اضافه کنیم.
"MILK" → → \emptyset
شیر رو به عنوان ورودی به تابع هشمون میدیم.

تابع هش هم در جواب به ما میگه ". ". پس بیاين قیمت شیر رو در ایندکس ۰ آرایه، ذخیره کنیم.



همینطوری ادامه بدین، درنهایت کل آرایمون پر میشه از قیمت محصولات.

1.49	0.79	2.49	0.67	1.49
------	------	------	------	------

حالا شما میاين میپرسین که، "ببخشید، قیمت آووکادو چقدر؟" شما نیازی ندارین توی آرایه به دنبالش بگردین. تنها کاری باید بکنین ، اینکه که "آووکادو" رو به عنوان ورودی به تابع هش بدین.



تابع هم به شما میگه قیمت این محصول در ایندکس شماره ۴ ذخیره شده. و مطمئنا همون جاست.

AVOCADO = 1.49

1.49	0.79	2.49	0.67	1.49	...
1	1	1	1	1	...

تابع هش، بهتون میگه قیمت یک کالا دقیقا در کجا ذخیره شده. پس در نتیجه، اصلا نیاز ندارین که جوستوجویی انجام بدین. این موضوع به این دلیل کار میکنه که:

- تابع هش ، همواره یک اسم رو به یک ایندکس نگاشت میکنه(ربط میده). هر بار که شما "آووکادو" رو وارد کنین، یک عدد بدست میارین. برای همین میتوانیں برای اولین بار برای پیدا کردن ایندکس مناسب برای اون محصول ازش استفاده کنین، و بعد از اون میتوانیں ازش برای پیدا کردن اون اینکدسى که قیمت مورد نظر تو ش ذخیر شده استفاده کنین.
- تابع هش، رشته های متفاوت رو به ایندکس ها متفاوت نگاشت میکنه(ربط میده). "آووکادو" به اندکس ۴ نگاشت میشه. "شیر" به ایندکس ۰ . هر چیزی به اسلات متفاوتی در آرایه نگاشت میشه ، که شما میتوانیں توی اسلات اون آرایه قیمت مورد نظر رو ذخیره کنین.
- تابع هش ، اندازه بزرگی آرایه شمارو میدونه. به همین دلیل ایندکس های معتبری رو بر میگردونه . اگر آرایه شما ۵ آیتم داره. تابع هش ۱۰۰ رو برنمیگردونه. به این دلیل که ۱۰۰ ایندکس معتبری برای این آرایه نیست.

شما همین الان مگی رو ساختین! از کنار هم گذاشتن تابع هش و آرایه ، شما به یک ساختار داده میرسین که بهش جدول هش(hash table) گفته میشه. جدول هش اولین ساختار داده ای هستش که یک منطق اضافی پشتیش هست و شما اونو یادمیگرین. آرایه ها و لیست ها مستقیم به مموری (حافظه کامپیووتر) نگاشت میشن. اما جداول هش، باهوش تر هستند. آنها از تابع هش، به طور هوشمندانه، برای پیدا کردن مکانی برای ذخیره یک عنصر استفاده میکنند.

جداول هش به احتمال زیاد پر کاربرد ترین، ساختار داده ای پیچیده ای هست که شما یادش میگیرین. همچنین جدول هش به عنوان هش مپ، مپ، دیکشنری و آرایه ها انجمنی(associative arrays) هم شناخته میشه. و جداول هش خیلی سریع هستند. بحثمون راجب آرایه ها و لینکد لیست ها در فصل دو یادتون هست؟ شما میتوانین بلافاصله یک آیتم رو از یک آرایه دریافت کنین(random access). و جداول هش از آرایه برای ذخیره داده استفاده میکنند. پس اونها به یک اندازه سریع هستند.

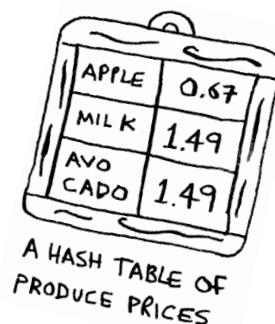
احتمالاً شما مجبور نمیشید که جداول هش رو خودتون پیاده کنین. هر زبان برنامه نویسی خوبی، ساختار جداول هش رو قبل پیاده کرده. پایتون هم جدول هش داره که بهش دیکشنری میگن. شما میتوانین یک جدول هش جدید با استفاده از تابع `dict` درست کنین.

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



یک جدول هش جدیده. بیاین چنتا قیمت به `book` اضافه کنیم:

```
>>> book["apple"] = 0.67 <----- An apple costs 67 cents.  
>>> book["milk"] = 1.49 <----- Milk costs $1.49.  
>>> book["avocado"] = 1.49  
>>> print book  
{'avocado': 1.49, 'apple': 0.67, 'milk': 1.49}
```



خیلی سادس! خب بیاین بپریم ببینیم قیمت آووکادو چقدره:

```
>>> print book["avocado"]  
1.49 <----- The price of an avocado
```

جدول هش شامل کلید ها و مقادیر میشه (*keys and values*)

در هش `book`، اسم محصولات میشه کلید ها، قیمت محصولات میشه مقادیر اون. یک جدول هش، کلید هارو به مقادیر نگاشت میکنه.

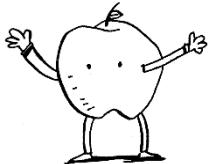
در بخش بعد شما مثال هایی رو میبینید، که جدول هش در اونها خیلی کاربردیه.

تمرین ها

برای توابع هش ضروریه که، همواره یک خروجی در ازای یک ورودی مشخص، برگردون. اگر اینکارو نکن، شما بعد از وارد کردن ورودی مد نظر، قادر به پیدا کردن آیتمتون نیستید.

کدوم یک از توابع هش زیر ثابت هستند؟

5.1	<code>f(x) = 1</code>	Returns "1" for all input	برای همه ورودی ها ۱ برمیگردونه
5.2	<code>f(x) = rand()</code>	Returns a random number every time	هر بار یک عدد رندوم برمیگردونه
5.3	<code>f(x) = next_empty_slot()</code>	Returns the index of the next empty slot in the hash table	ایندکس اسلات خالی بعدی رو در جدول هش برمیگردونه
5.4	<code>f(x) = len(x)</code>	Uses the length of the string as the index	از طول رشته به عنوان ایندکس استفاده میکنه



موارد استفاده

جداول هش، همه جا استفاده میشن. این بخش ، قسمت کوچکی از موارد استفاده اونها رو بهتون میگه.

استفاده از جداول هش برای جستجو

تلفن همراه شما دارا یک دفترچه تلفن داخلی دم دستیه.

هر اسم یک شماره تماس مرتبط با خودش رو داره.



BADE MAMA → ۵۸۱ ۶۶۰ ۹۸۲۰
ALEX MANNING → ۴۸۴ ۲۳۴ ۴۶۸۰
JANE MARIN → ۴۱۵ ۵۶۷ ۳۵۷۹

حالا فرض کنین ، که شما میخواین یک دفترچه تلفن مثل همین درست کنین. شما اسم افراد رو به شماره تماسشون نگاشت میکنین. دفترچه تلفن شما باید چنین عملکردی داشته باشه:

- قابلیت اضافه کردن اسم یک فرد و شماره تماس مربوط به همون فرد
- با وارد کردن اسم فرد باید شماره بتونیم شماره تماس اون فرد رو دریافت کنیم.

این یک مورد استفاده عالی برای جداول هش هستش! جداول هش عالی هستن، وقتی که شما بخواین:

- یک موردی رو بسازین که چیزی رو به چیز دیگری نگاشت کنه
- چیزی رو جستجو کنین.

ساختن دفترچه تلفن خیلی سادس. اول یک جدول هش جدید بسازید.

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

راستی، پایتون یک شورت کات برای ساخت جدول هش جدید داره. شما میتوینین از دو تا آکولاد استفاده کنید.

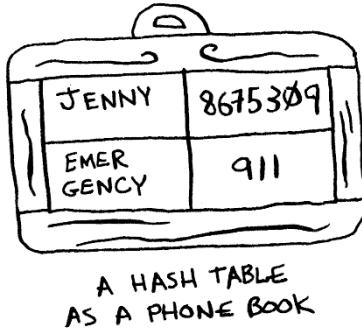
```
>>> phone_book = {} <----- Same as phone_book = dict()
```

بیاین شماره تماس چند فرد رو به این دفترچه تلقن اضافه کنیم:

```
>>> phone_book["jenny"] = 8675309  
>>> phone_book["emergency"] = 911
```

تمام کاری که باید انجام بدیم، همینه! حالا فرض کنین ما میخوان شماره تماس جنی(jenny) رو پیدا کنیم. فقط کافیه که کلید مورد نظر و به هش پاس بدیم:

```
>>> print phone_book["jenny"]  
8675309 <----- Jenny's phone number
```



تصور کنین، اگر ما میخواستیم این کارو با استفاده از آرایه بکنیم. چطور باید اینکارو میکردیم؟ جدول هش کار مارو برای مدل کردن یک رابطه بین یک چیز و چیز دیگه راحت کرده.

جدول هش برای جستجو در مقیاس های بسیار بزرگ ترهم استفاده میشه. به عنوان مثال، شما به یک وب سایت مثل <http://adit.io> میشنین. کامپیوتر شما باید adit.io رو به یک آی پی آدرس، ترجمه کنه.

ADIT.IO → 173.255.248.55

برای هر وب سایتی که شما واردش میشنین، آدرس اون باید به یک آی پی آدرس ، ترجمه بشه.

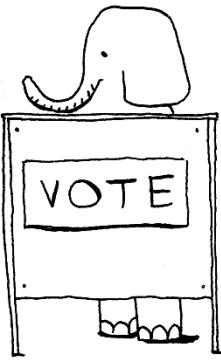
GOOGLE.COM → 74.125.239.133

FACEBOOK.COM → 173.252.128.6

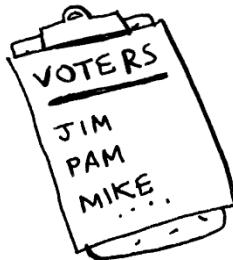
SCRIBD.COM → 23.235.47.175

واو! ربط دادن آدرس یک وبسایت به آی پی آدرس اون؟(نگاشت) به نظر یک مورد عالی برای استفاده از جداول هش هستش. به این فرایند دی ان اس رزولوشن میگن. جداول هش یکی از این راهها برای پیاده کردن این عملکرد هستند.

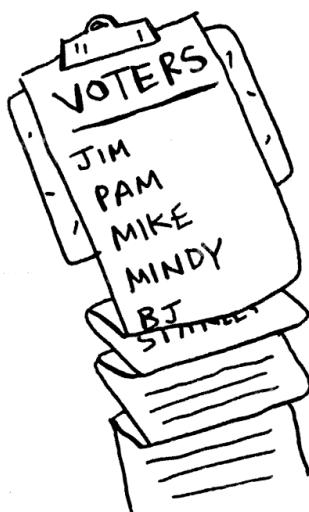
جلوگیری از ورودی های تکراری



فرض کنین شما در حال مدیریت یک اتاق رای گیری هستید. طبیعتاً، هر شخص فقط یکبار میتوانه رای بده. چطور میتوانیم مطمئن بشین که او نا قبل رای ندادن؟ وقتی یک نفر برای رای دادن میاد، شما ازش اسم کاملش رو میپرسین و بعد از اون لیستی از افرادی که قبل رای دادن رو برای اسم اون فرد بررسی میکنین.



اگر اسم اون فرد در لیست بود، یعنی این فرد قبل رای داده، (شوتش کنین بیرون). در غیر اینصورت اسم او نهارو وارد لیست میکنین و اجازه میدین که رای بدن. حالا فرض کنین تعداد زیادی برای رای دادن او مدن، و اون لیستی که اسم افرادی که رای دادن داخلش هست، خیلی طولانیه.



هر بار که فرد جدیدی برای رای دادن میاد، شما باید این لیست بلند بالا رو بررسی کنید تا ببینید آیا اونا قبل رای دادن یا نه. اما یک راه بهتر وجود داره: از هش استفاده کنین.

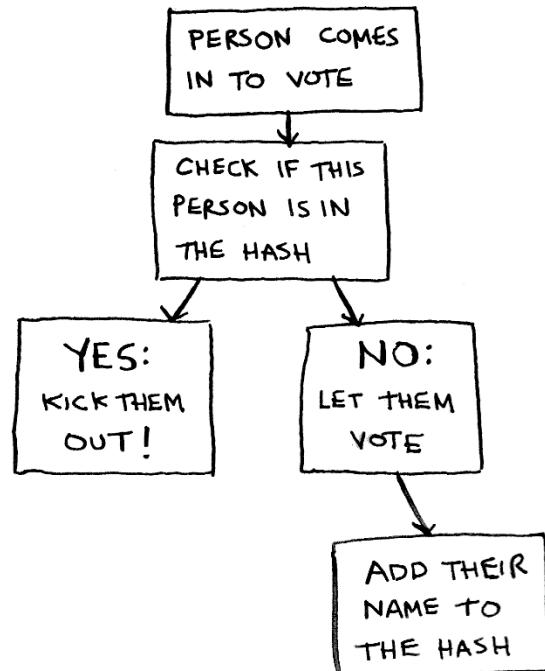
اول، یک هش درست کنین که حواستون به کسانی که رای دادن باشه:

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

وقتی فردی جدید برای رای گیری وارد میشه، بررسی کنین ببینین که آیا از قبل توی هش هستن یا نه؟:

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

تابع get مقدار "tom" را اگر در هش وجود داشته باشد برمیگردونه. در غیر اینصورت، مقدار None را برمیگردونه. شما میتوانید از این روش استفاده کنید که بررسی کنید که آیا کسی قبلاً رای داده یا نه!



این هم کدش:

```

voted = {}

def check_voter(name):
    if voted.get(name):
        print "kick them ou
    else:
        voted[name] = True
        print "let them vote!"
  
```

بیاین چندتا آیتم رو باهم بررسی کنیم:

```

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

وقتی تام برای بار اول وارد هش میشه، خروجی میشه "let them vote!" (بزار رای بدن) بعد از اون مایک وارد میشه و دوباره "let them vote!" چاپ میشه. اما بعدش مایک سعی میکنه که برای بار دوباره وارد بشه که اینبار این کد این خروجی رو میده "kick them out!" (شوتشون کنید بیرون).

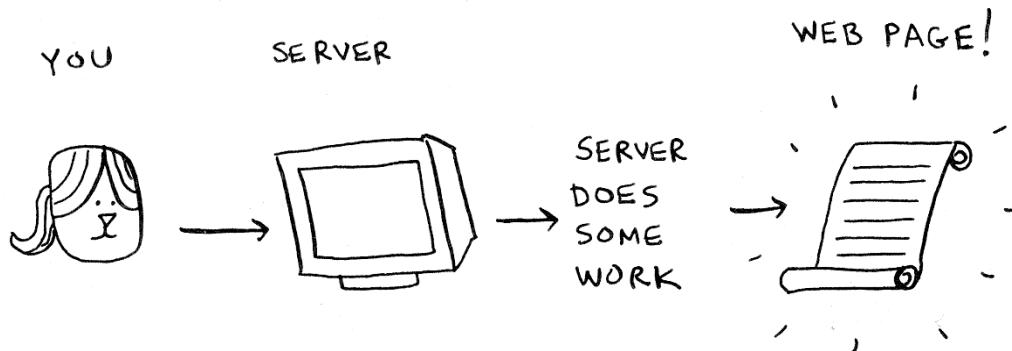
یادتون باش، اگر شما میخواستین اسمایی کسانی که رای دادند رو در یک لیست ذخیره کنید، در نهایت این تابع به شدت کند میشد. به این دلیل که باید سرچ ساده رو روی کل لیست پیاده میکرد. اما بجاش، شما دارین اسمایی رو در

یک جدول هش ذخیره میکنین، و جدول هش بلا فاصله بهتون میگه آیا این فرد در این جدول هش هست یا خیر.
بررسی کردن برای وجود آیتم های تکراری با جدول هش خیلی سریع انجام میشه.

استفاده از جدول هش برای کش

آخرین مثال برای موارد استفاده هش: کش کردن. اگر روی یک وب سایت کار میکنین، شاید قبل اسم کش کردن که یک کار خیلی خوب برای انجام دادن روی سایته، به گوشتون خورده. اینجا یک ایده داریم. فرض کنید شما به فیس بوک سر میزنید:

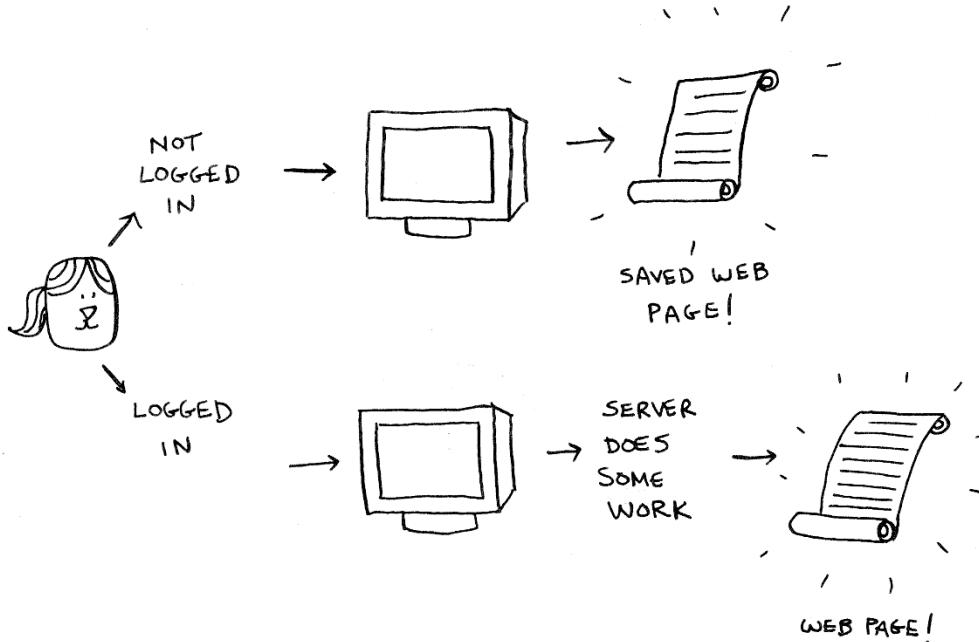
۱. شما یک درخواست به سرور های فیس بوک میفرستید.
۲. سرور یکم فکر میکنه و یک صفحه پیدا میکنه که برای شما بفرسته.
۳. شما صفحه رو دریافت میکنین.



به عنوان مثال، شاید سرور های فیس بوک در حال جمع آوری فعالیت های دوستان شما هستن تا اونها رو به شما نشون بدن. چند ثاینه طول میکشه که تا تمام اون اطلاعات جمع آوری بشن و به شما نشون داده بشه. اون چن ثانیه ممکنه برای کاربری خیلی طولانی باشه. ممکنه فک کنین "چرا فیس بوک اینقدر کند؟"، از طرفی سرور های فیس بوک باید به میلیون ها نفر خدمت رسانی کنند. و اون چن ثانیه هم به تایم اونا اضافه میشه. سرور های فیس بوک واقعا سخت کار میکنن تا بتونن به اون صفحات وب خدمان برسونن. آیا راهی هست که فیس بوک سریعتر بشه و در عین حال سرور ها کار کمتر انجام بدن؟

فرض کنین یک خواهر زاده کوچولو دارین که مدام از شما درباره سیاره ها سوال میپرسه. "مریخ چقدر از زمین فاصله دارد؟" "ماه چقدر از دوره؟" "با مشتری چقدر فاصله داریم؟" هر بار شما باید گوگل کنین و به خواهر زادتون جواب بدین. این موضوع چند دقیقه وقت میبره. حالا فرض کنیم همیشه سوال میپرسید که "ماه چقدر از ما فاصله دارد؟" خیلی زود جواب رو حفظ میکردید که ماه در فاصله ۲۳۸۹۰۰ مایلی ما قرار دارد. دیگه مجبور نیستید هر بار گوگل کنین. شما فقط بیاد میارین و جواب میدین. کش کردن هم اینطوری کار میکنه. سایت ها اطلاعات رو بخاطر میارن به جا اینکه دوباره محاسبش کنن. اگر شما وارد فیس بوک بشید (لاگ این بکنید)، (متوجه میشین که) تمام

محتوها برای شما طراحی شده اند. هر بار که وارد facebook.com میشین، سرور های اون باید درباره این فکر کنن که شما به چه محتوایی علاقمند هستید. اما اگر وارد فیس بوک نشدید، شما با صفحه لاغ این(صفحه ورود) مواجه میشین. همه (همه کسانی که وارد نشده اند) همون صفحه لاغ این را میبینن. فیس بوک هم داره یک چیز رو بارها و بارها از سرور هاش درخواست میکنه. "وقتی که من از سیستم خارج شدم بهم هوم پیج (صفحه خانه) رو نشون بده" درنتیجه به جای اینکه سرور رو مجبور کنه که بفهمه هوم پیج (صفحه خانه) چه شکلیه، سریع اونو بخاطرمیاره و اونو برای شما میفرسته.



به این عمل، کش کردن گفته میشه.

دو مزیت مهم داره:

- شما خیلی سریع تر به صفحه وب دسترسی پیدا میکنید. دقیقا مثل بخاطر آوردن فاصله‌ی زمین تا ماہ.دفعه بعد که خواهرزادتون ازتون سوال کنه، شما دیگه مجبود نیستید اونو گوگل کنید. شما میتوانید بلا فاصله جواب بدین.
- فیس بوک کار کمتری انجام میده.

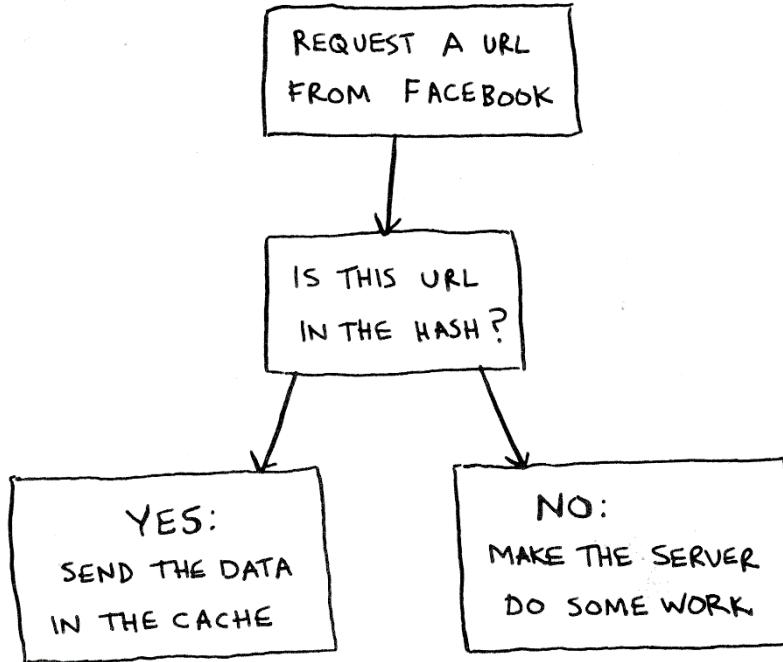
کش کردن یک کار رایج برای سریعتر کردن چیز های مختلفه. تمام وب سایت های بزرگ از کش کردن استفاده میکنن. و اون دیتا در هش ذخیره میشه!

فیس بوک فقط هوم پیج رو کش نمیکنه. اون همچنین صفحات درباره ما، ارتباط با ما، شرایط و ضوابط و خیلی صفحات دیگرو کش میکنه. پس یک ارتباط (یک نگاشت) بین آدرس صفحه و اطلاعات صفحه نیاز داره.

`facebook.com/about` → DATA FOR THE ABOUT PAGE

`facebook.com` → DATA FOR THE HOME PAGE

وقتی شما از صفحه‌ای در فیس بوک بازدید می‌کنید، فیس بوک اول چک می‌کنه که آیا اون صفحه در هش وجود داره یا نه.



اینم کد این بخش:

```
cache = {}

def get_page(url):
    if cache.get(url):
        return cache[url] Returns cached data
    else:
        data = get_data_from_server(url)
        cache[url] = data Saves this data in your cache first
    return data
```

در اینجا، شما تنها وقتی سرور رو وادار به کار می‌کنین که *URL* کش نشده باشه. به هر حال، قبل از اینکه داده رو برگردانیم اونو در کش ذخیره می‌کنیم. دفعه بعدی اگر کسی این آدرس رو درخواست کنه، میتوانیم داده هارو از کش برای اون بفرستیم بجای اینکه سرور رو وادار به کار بکنیم.

خلاصه این بخش

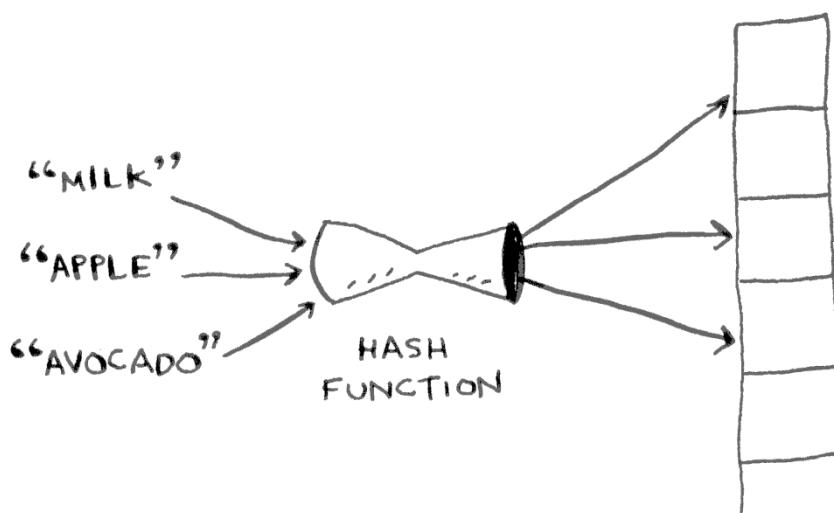
به عنوان خلاصه این بخش، استفاده از هش در این موارد بسیار خوب است:

- مدل کردن رابطه ها از چیزی به چیز دیگر. (ارتباط دادن دو چیز مرتبط به هم)
- فیلتر کردن برای خارج کردن موارد تکراری
- کش کردن/بخاطر سپردن داده به جای وادار کردن سرور برای کار.

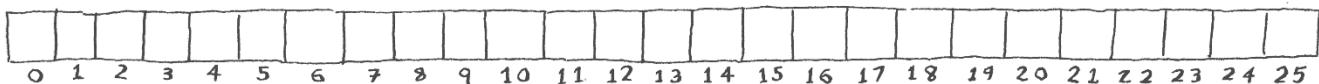
تلاقی (Collision) *(Collision)*

همونطور که قبلاً گفتم، اکثر زبان های برنامه نویسی، جدول هش رو دارند. نیازی ندارین که بدونین چطور برای خودتون باید پیاده سازیش کنین. پس من درباره مفاهیم داخلی هش با شما زیاد صحبت نمیکنم. اما هنوز نحوه عملکرد برای شما مهمه! برای درک نحوه عملکرد جداول هش، اول باید بدونین که تلاقی ها چی هستن. دو بخش بعدی بحث تلاقی و عملکرد رو توضیح میده.

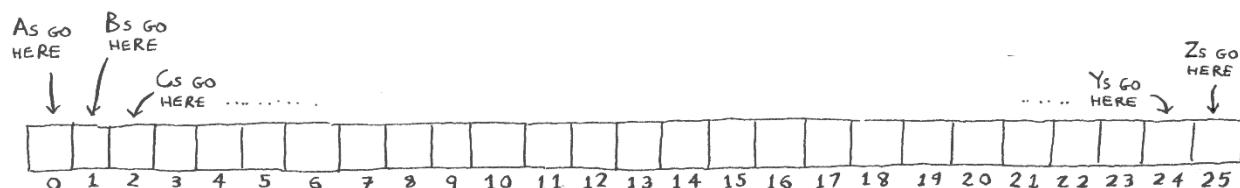
اول از همه، باید بگم که من به شما یک دروغ مصلحتی گفتم. بهتون گفتم که تابع هش همیشه کلید های متفاوت رو به اسلات ها متفاوت در آرایه ارتباط میده (نگاشت میکنه).



در واقع، این کار که یک تابعی بنویسیم که همچین کاری کنه، تقریباً غیر ممکنه. بیاین یک مثال ساده بزنیم. فرض کنیں آرایه شما شامل ۲۶ اسلات میشود.

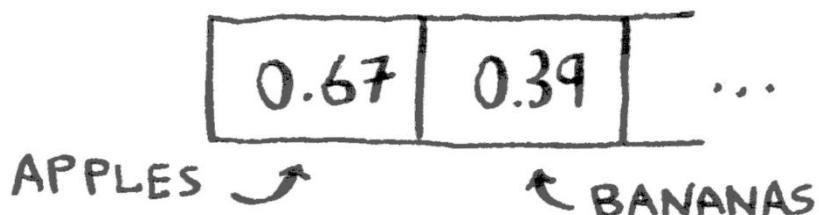


و تابع هش شما، بسیار سادس. یک نقطه رو به ورودی مورد نظر، طبق حروف الفبا، نسبت میدهد. (الفبای انگلیسی)

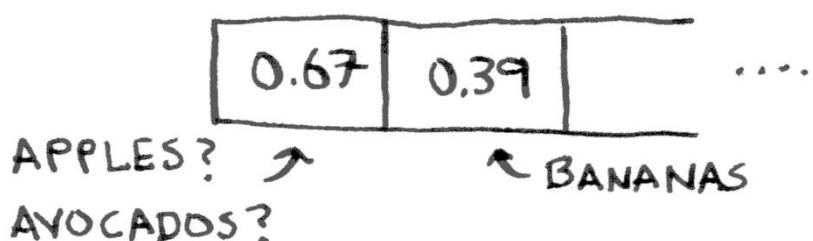


شاید از قبل بتونین مشکل این مسئله رو ببینین. شما میخواین قیمت مثلا سیب ها رو (که در انگلیسی کلمه سیب با حرف a شروع میشه یعنی اولین حرف حروف الفبا انگلیسی) میخواین در هش ذخیره کنین. شما اسلات اول رو به این مورد اختصاص میدین.

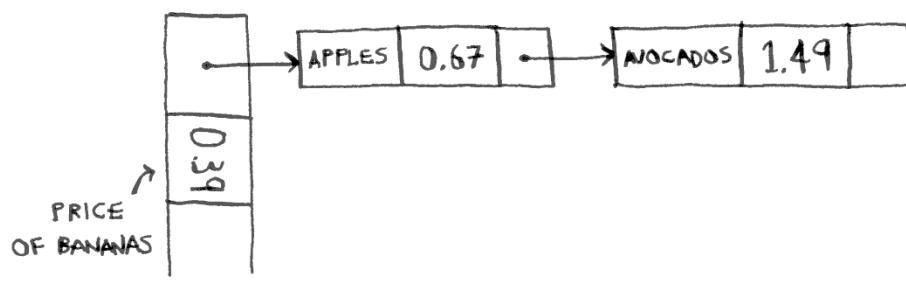
و بعد از اون میخواین که قیمت موز ها رو (که در انگلیسی کلمه موز با b شروع میشه (دومین حرف حروف الفبا انگلیسی)) در هش قرار بدین. پس اسلات دوم رو هم به اینکار اختصاص میدین.



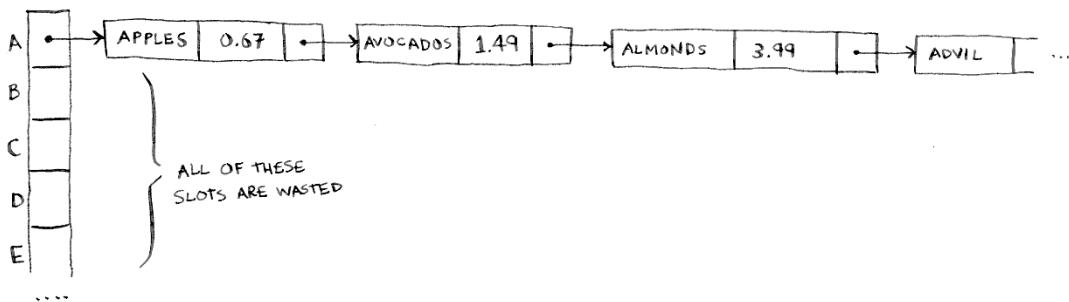
همه چیز داره خوب پیش میره! اما شما میخواین قیمت آووکادو رو هم در هش اضافه کنین (که آووکادو هم در انگلیسی با a شروع میشه). و شما دوباره میخواین اسلات اول رو به این کار اختصاص میدین.



اووه نه! قبل از سیب به اون اسلات، اختصاص داده شده. به این اتفاق تلاقی گفته میشه. دو کلید به یک اسلات اختصاص داده شدن. اگر آووکادو رو به اون اسلات اختصاص بدین، با این کار قیمت سیب هارو بازنویسی کردین(overwrite). و اگر برای دفعه بعد کسی قیمت سیب رو بپرسه به جاش، قیمت آووکادو بهش داده میشه. تلاقی ها چیز بدی هستن. و شما باید با آنها سروکله بزنین. راهای بسیار مختلفی برای نحوه برخورد با تلاقی ها وجود داره. راحت ترین اونها اینه: اگر چند کلید به یک اسلات مشترک مرتبط میشنوند (نگاشت میشنود)، یک لینک کد لیست در اون اسلات ایجاد کنید.



در این مثال ، هم سیب و هم آووکادو به یک اسلات مرتبط میشوند (نگاشت میشنود). پس درنتیجه شما در اون اسلات یک لینکد لیست ایجاد میکنید. اگر میخواین قیمت موز ها رو بدونین، هنوز این روش سریعه. اما اگر میخواین قیمت سیب هارو بدونین، این روش یکم کنده. شما باید برین داخل این لینکد لیست رو پیدا کردن سیب بگردین. اگر لینکد لیست کوچیک باشه، مشکلی نیست، شما فقط نیازه که بین ۳ یا چهار تا عنصر جستجو کنین. اما فرض کنین که شما در یک خواربار فروشی کار میکنین که فقط محصولاتی رو میفروشین که با A شروع میشن.



یه لحظه صبر کنین. کل جدول هش به جز یک خونه خالیه. و اون اسلات شامل یک لینکد لیست بزرگ میشه. هر کدوم از عناصر در این جدول هش داخل لینکد لیست هستن. این به همون اندازه افتضاوه که داده هامون رو داخل مستقیم داخل یک لینکد لیست بربیزیم. این کار جدول هش شما رو کند میکنه.

دو تا درس اینجا هست:

- تابع هش شما بسیار مهمه. این تابع هش شناس که تمام کلید هارو داره به یک اسلات وصل میکنه. حالت ایده آlesh اینه که، تابع هش شما کلید هارو به طور مساوری در تمام جدول هش، متصل کنه (نگاشت کنه).
- اگر اون لینکد لیست طولانی بشه. خیلی جدول هشتون رو کند میکنه. ولی اگر از یک تابع هش خوب استفاده کنین این دیگه اون لینکد لیست طولانی نمیشه.

تابع هش واقعا مهم هستن. یک تابع هش خوب، فقط تعداد کمی ، تلاقی به شما تحویل میده. خب چطور یک تابع هش خوب استفاده میکنین؟ در ادامه بخش بعد توضیح میدم بهتون.

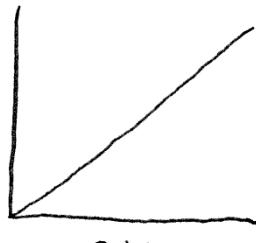
AVERAGE CASE WORST CASE

SEARCH	$O(1)$	$O(n)$
INSERT	$O(1)$	$O(n)$
DELETE	$O(1)$	$O(n)$

عملکرد

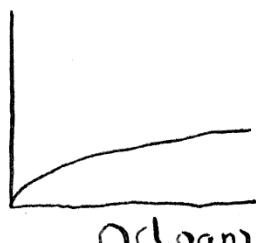
شما این فصل رو با مثال خواربار فروشی شروع کردید. شما میخواستید چیزی درست کنید که قیمت کالا ها رو بلا فاصله به شما تحویل بده. خب، جداول هش واقعا سریع هستند.

در حالت میانگین، جدول هش برای هر چیزی به اندازه $O(1)$ زمان ثابت گفته میشه. به $O(n)$ طول میکشه. شما قبل از زمان ثابت رو ندیده بودید. زمان ثابت، معنیش "بلافاصله" نمیشه. معنیش این میشه که زمانی که طول میکشه تا کاری انجام بشه، صرف نظر از بزرگی جدول هش، یک مقدار باقی میمونه. به عنوان مثال شما میدونین که، سرچ ساده زمان خطی رو طی میکنه.



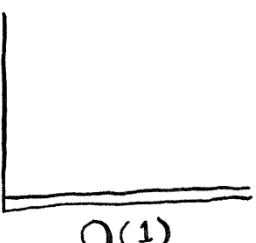
$O(n)$
LINEAR TIME
(SIMPLE SEARCH)

باينري سرچ، سريع تره و زمان لگاريتمي رو طي ميکنه:



$O(\log n)$
LOG TIME
(BINARY SEARCH)

به دنبال چيزی در جدول هش گشتن هم زمان ثابت رو طی ميکنه.



$O(1)$
CONSTANT TIME
(HASH TABLES)

ميبينين چقدر خطش صافه؟ اين يعني اصلا اهميتی نداره که تابع هش شما شامل يك عنصر باشه يا 1 ميليارد عنصر. بدست آوردن هر چيزی با استفاده از جدول هش، يك مقدار زمان (مشخص) مibره. درواقع شما زمان ثابت رو قبل از دیده بودید. برای گرفتن يك آيتم از يك آرایه هم يك زمان ثابت طی میشه. اصلا اهميتی نداره که آرایتون چقدر

بزرگ باشد، فقط یک مقدار مشخص زمان میبره تا به عنصر مورد نظر دست پیدا کنیم. در حالت میانگین جداول هش خیلی سریع هستند.

در بدترین حالت جدول هش برای هر چیزی به اندازه $O(n)$ (زمان خطی) زمان میبره، که خیلی کنده. بیاین جدول هش رو با آرایه ها و لیست ها مقایسه کنیم.

	HASH TABLES (AVERAGE)	HASH TABLES (WORST)	LINKED ARRAYS	LINKED LISTS
SEARCH	$O(1)$	$O(n)$	$O(1)$	$O(n)$
INSERT	$O(1)$	$O(n)$	$O(n)$	$O(1)$
DELETE	$O(1)$	$O(n)$	$O(n)$	$O(1)$

به حالت میانگین درمود جدول هش نگاه کنیم. جداول هش به اندازه آرایه هادر جستجو کردن سریع هستند (دربیافت یک مقدار از یک ایندکس). و به اندازه لینکد لیست در اضافه کردن و حذف کردن هم سریع هستند. بهترین ها رو از دو طرف برای خودش برداشتند. اما در بدترین حالت، جدول هش در تمام اون موارد کنده. خب پس این مهمه که هیچ وقت عملکردن توں به بدترین حالت در جدول هش نزدیک نشه. و برای اینکار شما باید از تلاقی ها دوری کنیم. برای دوری کردن از تلاقی نیازه که شما:

- یک ضریب بار کمی داشته باشین.
- یک تابع هش خوب داشته باشین.

یادداشت

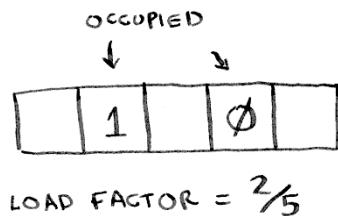
قبل از اینکه شروع به خوندن این بخش بکنیم. لازمه که بدونین نیازی به خوندنش ندارین. من میخوام درباره نحوه پیاده سازی جدول هش صحبت کنم، اما شما اصلا نیازتون نمیشه که خودتون این کارو بکنین. هر زبان برنامه نویسی که دارین باهاش کار میکنین از قبل جداول هش داخلش پیاده شده. شما میتوانین از جدول هش داخلی اون زبان برنامه نویسی استفاده کنین و فرضتون رو بر این بزارین که عملکرد خوبی خواهند داشت. بخش بعدی نگاهی از دل این داستان به شما میدهد.

ضریب بار

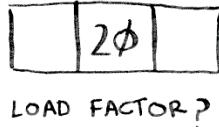
$$\frac{\text{تعداد آیتم ها در جدول هش}}{\text{تعداد تمام اسلات ها}} = \frac{\text{NUMBER OF ITEMS IN HASH TABLE}}{\text{TOTAL NUMBER OF SLOTS}}$$

محاسبه ضریب بار جدول هش خیلی سادس.

جدول هش برای ذخیره سازیش از آرایه ها استفاده میکنه، پس شما تمام اسلات های اشغال شده رو در آرایه مشمارید. برای مثال این، ضریب بار این آرایه برابر $\frac{2}{5}$ یا ۰.۴.

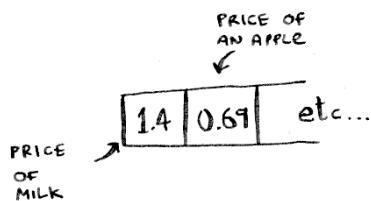


ضریب بار این جدول هش چیه؟

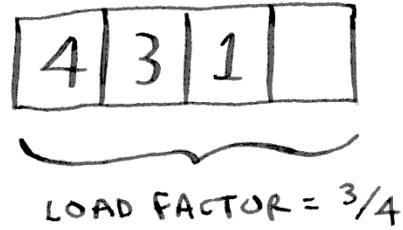


اگر گفتین $\frac{1}{3}$ ، جواب شما درسته. ضریب بار میاد اندازه میگیره که چننا از اسلات ها در جدول هش باقی موندند.

فرض کنین شما نیاز دارین که قیمت ۱۰۰ محصول رو در یک جدول هش ذخیره کنین. و جدول هش شما هم ۱۰۰ اسلات داره. در بهترین حالت، تمام محصولات اسلات مختص خودشونو میگرن.



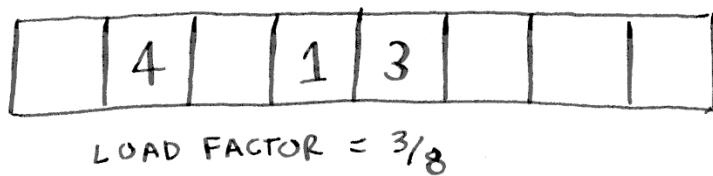
ضریب بار این جدول هش برابر ۱ هستش. چه میشد اگر جدول هش شما شامل ۵۰ اسلات بود؟ خب با این اوصاف، ضریب بار اینیکی برابر ۲ میشه. هیچ راهی وجود نداره که هر آیتم اسلات مخصوص خودشو داشته باشه، چرا که اسلات های کافی برای اینکار وجود نداره. داشتن ضریب بار بالا ۱ یعنی آیتم های شما بیشتر از تعداد اسلات های شما در آرایه س. وقتی ضریب بار رو به افزایش میره، نیازه که شما اسلات های بیشتری به جدول هشتون اضافه کنین. به این کار تغییر اندازه (resizing) گفته میشه. به عنوان مثال، فرض کنین، یک همچین جدول هشی دارین که تقریباً داره پر میشه:



نیازه که شما جدول هش رو ریسایز کنین(*resize*). اول شما آرایه ای درست میکنین که از قبلی بزرگتره. قاعده کلی اینکه یک اون آرایه ای که درست میکنین اندازش دو برابر آرایه قبلی باشه.



حالا دوباره نیازه که آیتم هاتون رو دوباره به این جدول هش جدید، با استفاده از تابع *hash*، اضافه کنین.

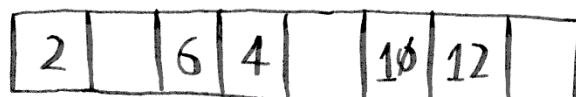


ضریب بار این جدول هش برابر $\frac{3}{8}$. خیلی بهتر شد! با ضریب بار کمتر، تلاقی کمتری خواهید داشت و جدول شما عملکرد بهتر خواهد داشت. یه قاعده کلی دیگه هست که میگه وقتی ضریب بارتون بیشتر از ۰.۷ شد ریسایز کنید.

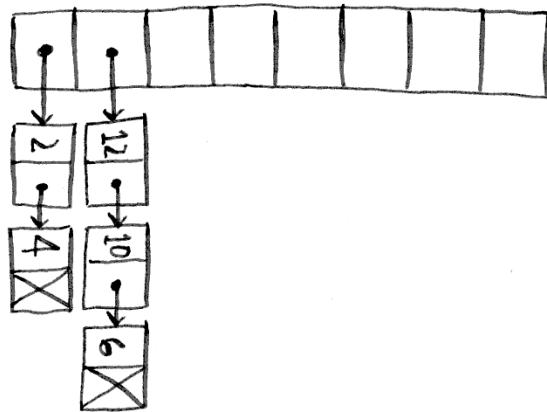
شما ممکنه فک کنین "این همه ریسایز، زمان زیادی رو میگیره!" و درسته! دارین درست میگین. ریسایز کردن خیلی هزینه بره و شما هم اینو نخواین که مدام جدولتونو ریسایز کنین. اما به طور میانگین، حتی با وجود ریسایز کردن، جدول هش به اندازه $O(1)$ زمان میبره.

یک تابع هش خوب

یک تابع هش خوب، مقادیر رو به طور مساوی در آرایه توزیع میکنه.



یک تابع هش بد هم، مقادیر رو به صورت گروهی درمیاره و تلاقی های زیاد ایجاد میکنه.



تمرین ها

برای یک تابع هش، مهمه که توانایی توزیع خوبی داشته باشه. آنها باید تا حد امکان آیتم هارو به صورت گسترده تری به هم نگاشت کنن (به صورت گستردگی تر پخششون کنن). بدتری حالت در تابع هش اونیکه تمام آیتم هارو در یک اسلات در جدول هش قرار میده.

فرض کنین شما توابع زیر رو به عنوان تابع هش در اختیار دارین که هر کدام از این توابع یک رشته به عنوان ورودی دریافت میکنند:

A. این تابع در ازای هر ورودی a رو برمیگردونه.

B. این تابع طول رشته رو به عنوان ایندکس برای ذخیره سازی برمیگردونه. (مثلا طول رشته "apple" هستش)

C. از کاراکتر اول هر رشته به عنوان ایندکس استفاده میکنه، مثلا، همه رشته هایی که با a شروع میشن رو در یکجا باهم ذخیره میکنه.

D. هر حرف رو به یک عدد اول نسبت میده (نگاشت میکنه): $a = 2$ ، $b = 3$ ، $c = 5$ ، $d = 7$ ، $e = 11$ ، و الی آخر. این تابع هش، برای یک رشته، جمع تمام اعداد نسبت داده شده به هر حرف رو با توجه به اندازه هش برمیگردونه. به عنوان مثال. اگر اندازه جدول هش ما ۱۰ باشه، و رشته ورودی برابر با "bag" باشه، ایندکس این رشته برابر میشه با $2 + 17 + 3 = 22 \% \cdot 10 = 2$.

برای هر کدام از مثال های زیر، کدام یک از توابع بالا میتونن توزیع خوبی داشته باشن؟ فرض کنین جدول هش ما ۱۰ اسلات داره.

۵/۵ یک دفترچه تلفن که کلید های اون اسماهаш و مقادیر اون شماره تلفن همون افراد هستش. اسامی به این شرح: استر^۱(Esther)، بن(Ben)، باب(Bob)، و دن(Dan).

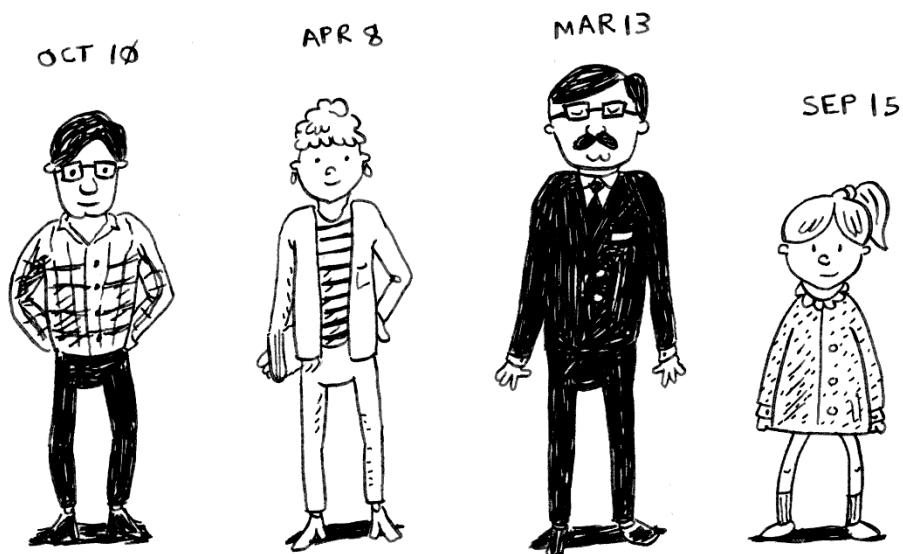
۵/۶ یک نگاشت از سایز باتری به قدرت باتری. سایز ها این ها هستن: A، AA، AAA و AAAA

خلاصه این فصل

شما تقریبا هیچ وقت مجبور نمیشید که جدول هش رو خودتون پیاده کنین. زبان برنامه نویسی که باهاش کار میکنین این رو باید برای شما پیاده کرده باشه. شما میتوانین از جدول هش پایتون استفاده کنین و فرض رو بر این بزارین که عملکرد حالت میانگین که همون "زمان ثابت" باشه رو برآتون فراهم میکنه.

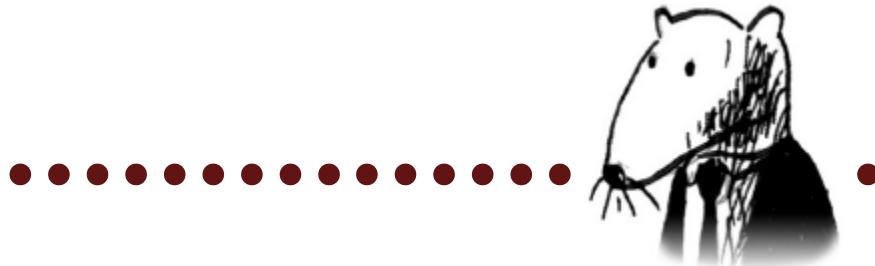
جداول هش یکی از قدر تمدن ترین ساختار داده هستن به این دلیل که خیلی سریعن و به شما این اجازه رو میدن که داده هاتون رو با راهای مختلفی مدل کنین(شکل بدین). ممکنه به زودی به این موضوع پی ببرین که دارین در تمام مدت از اونها استفاده مکنین.

- شما میتوانین یک جدول هش با ترکیب کردن یک تابع هش با آرایه درست کنین.
- تلاقی ها چیز بدی هستن. شما به یک تابع هش نیاز دارین که تلاقی هارو به حداقل برسونه
- جداول هش قابلیت جستوجوی سریع، درج سریع و حذف سریع رو دارن.
- جداول هش یک راه خوب برای مدل کردن رابطه از یک چیز به چیز دیگر هستند.
- وقتی که ضریب بارتون بیش تر از ۰.۷ باشه نشون میده که زمان ریسایز کردن جدول هش هستش.
- جداول هش در کش کردن داده ها مورد استفاده قرار میگیرن (یه عنوان مثال، در یک وب سرور)
- جداول هش یک راه عالی برای جلوگیری از وجود موارد تکراری هستند.



جستوجوی اول عمق

(Breadth-first search)



در این فصل:

- یادمیگرید چطور یک مدل شبکه رو با استفاده از یک ساختار داده انتزاعی جدید رو مدل کنین: گراف ها.
- در مورد جستوجوی اول عمق یادمیگرین، یک الگوریتم که میتوانین برای جواب به این سوال که "کوتاه ترین راه برا رفتن به مکان X کدامه؟" اجرایش کنین.
- شما درباره گراف های جهت دار و غیر جهت دار یاد خواهید گرفت.
- شما در مورد مرتب سازی توپولوژیکی یاد خواهید گرفت. که نوعی از الگوریتم های مرتب سازیست که وابستگی های بین گره ها را مشخص میکند.



این فصل به معرفی نمودار ها میپردازد. اول ، درباره اینکه گراف ها چی هستند صحبت میکنم (اونا شامل محور های X و Y نمیشن). و بعد از اون من اولین الگوریتم گراف شمارو بهتون نشون میدم. به این الگوریتم ، جستوجوی اول عمق گفته میشه(BFS).

جستوجوی اول عمق، این اجازه رو بهتون میده که بتونین کوتاه ترین مسیر رو بین دو چیز پیدا کنین. اما کوتاه ترین مسیر میتونه معانی زیادی داشته باشه. با استفاده از جستوجوی اول عمق میتونین:

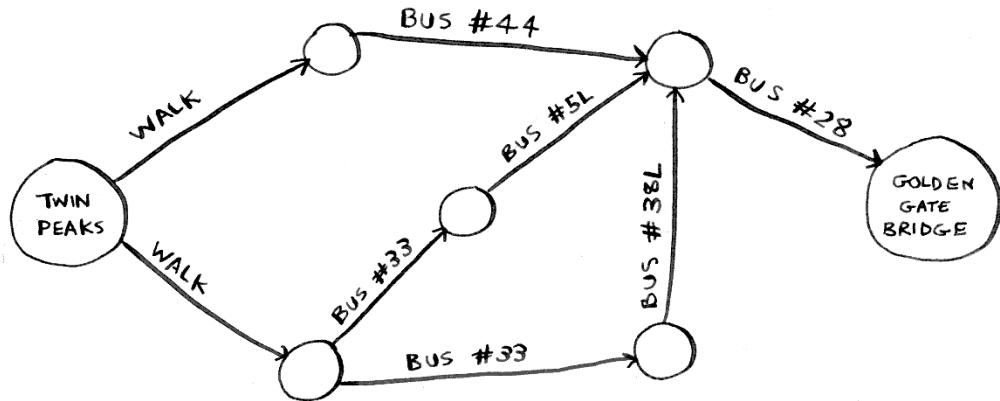
- هوش مصنوعی بازی چکرز رو باهاش بنویسین که کمترین حرکات برای پیروزی رو حساب کنه.
- یک غلط گیر املاکی بنویسین(مثل همین هایی که روی گوشیمون داریم و وقتی مثلثا مینویسیم "سلام" به "سلام" تغییرش میده).
- برای پیدا کردن نزدیک ترین دکتر به شما ازش استفاده کنین.

الگوریتم های مربوط به گراف از پرکاربردترین الگوریتم هاییست که من میشناسم. مطمئن بشین که این بخش و چند فصل بعدی رو با دقت میخونین. این ها الگوریتم هایی هستند که به طور مداوم از اون ها استفاده میکنین.

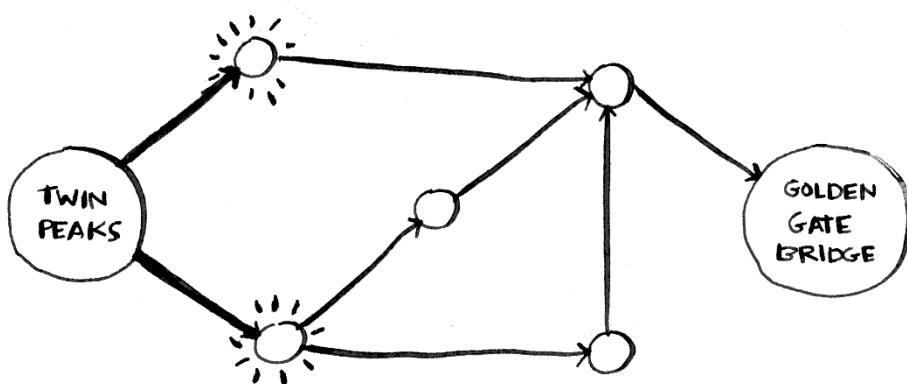
مقدمه ای بر گراف ها



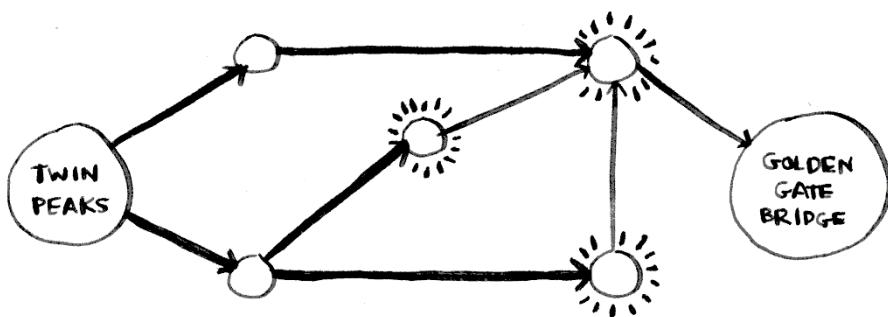
فرض کنین شما در سانفرانسیسکو (San Francisco) هستین، و میخواین از تویین پیکس (Twin peaks) به پل گلدن گیت (Golden Gate Bridge) بین. شما میخواین از اتوبوس برای این کار استفاده، با حداقل حمل و نقل با اتوبوس. این گزینه هاییه که شما در اختیار دارین:



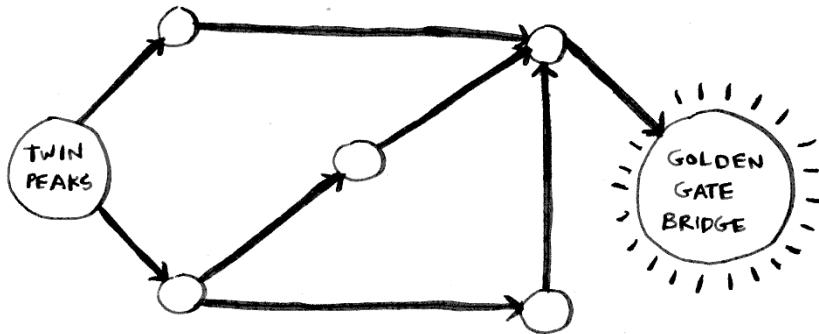
الگوریتم شما برای پیدا کردن مسیری با کمترین تعداد گام، برای این مسئله چیست؟
خب، آیا میتوانیم با یک گام به اونجا برسیم؟ اینجا تمام مسیرهایی را نشون دادم که با یک گام میشه بهشون رسید.



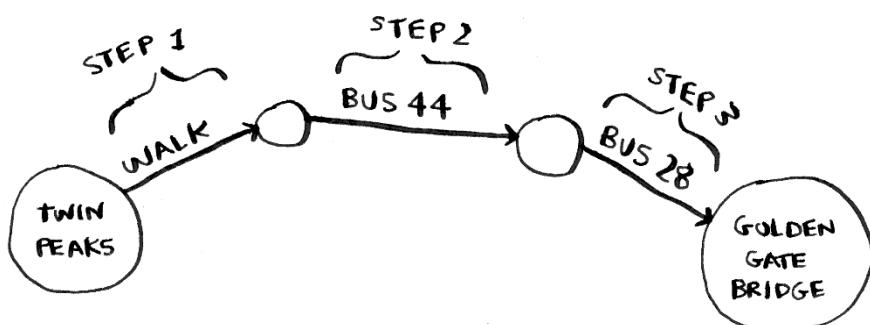
هنوز به پل نرسیدیم پس نمیتوانیم در یک گام به اونجا برسیم. آیا میشه با دو گام به اونجا رسید؟



دوباره، به پل نرسیدیم. درنتیجه با دو گام هم نمیتوانیم به پل برسیم، درمورد سه گام چطور؟



اها! بالاخره پل پیدا شد. پس ، ۳ گام نیازه که از توضیح به پل با استفاده از این مسیر برسیم.



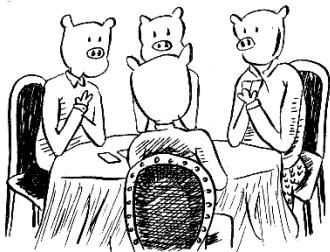
راهای دیگه ام هم هستن که شما رو به پل میرسون. همه طولانی ترн(چهار گام). الگوریتم به ما میگه که کوتاه ترین مسیر به پل ۳ گام داره. به این دست مسائل ، مسائل کوتاهترین-مسیر، گفته میشه. شما همیشه سعی میکنین کوتاهترین چیز رو در این مسائل پیدا کنین. حالا این میتونه کوتاه ترین مسیر به خونه دوستتون باشه. میتونه کمترین حرکت برای کیش مات در شطرنج باشه. به الگوریتمی که برای حل مسائل کوتاهترین-مسیر استفاده میشه، جستوجوی اول عمق گفته میشه.

دو قدم برای اینکه بفهمیں چطور کوتاه ترین مسیر از توضیح پیکس به پل گلدن گیت رو پیدا کنین ، وجود داره:

۱. مسئله رو به شکل گراف دربیارین(مدل کنین).
۲. حالا مسئله رو با استفاده از جستوجوی اول عمق حل کنید.

در ادامه ، به این موضوع که گراف ها چی هستند میپردازم. و بعد وارد جزئیات جستوجوی اول عمق میشم.

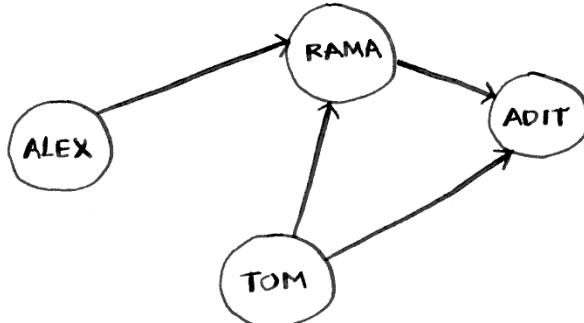
گراف چیست؟



یک گراف، مجموعه ای از ارتباطات رو مدل میکنه(به حالت شکل درمیاره). به عنوان مثال فرض کنین، شما و دوستانتون مشغول بازی پوکر هستین. و شما میخواین این موضوع رو که چه کسی به چه کسی پول بده کار است رو مدل کنین.

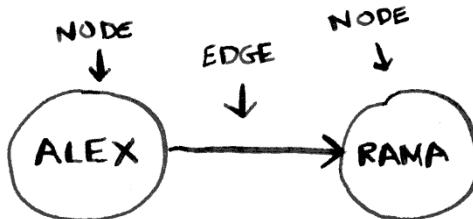


گراف کاملش میتوانه این شکلی باشه.



گرافی که نشون میده، چه کسی به چه کسی پول بده کار است

الکس به راما بده کار است، تام به ادیت بدهکار است و الی آخر. هر گراف از چند گره و یال تشکیل شده است.



تمام چیزی که وجود داره همیناس. گراف ها از گره ها و یال ها تشکیل شده اند. یک گره میتوانه مستقیماً به چند گره دیگر متصل بشه. که اون گره ها همسایه گفته میشه. در این گراف ، راما ، همسایه الکسه. اما ادیت همسایه راما و تام هستش.

گراف ها یک راه برای اینن که ببینیم چطور یک چیز به چیز ها دیگر متصل میشه. حالا ببین جستوجوی اول عمق در در عمل ببینیم.

جستوجوی اول عمق

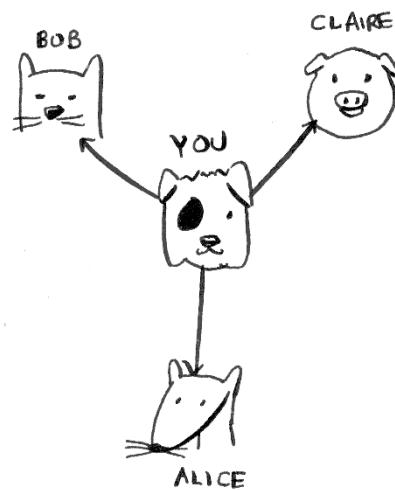
ما در فصل اول یک الگوریتم مربوط به جستجو رو بررسی کردیم که باینری سرچ بود. جستوجوی اول عمق یک نوع متفاوت از الگوریتم های جستجو: نوعی که روی گراف ها اجرا میشه. این الگوریتم به ما در جواب دادن دو نوع سوال کمک میکنه.

- سوال نوع اول : آیا راهی از گره A به گره B وجود دارد؟
- سوال نوع دوم: کوتاه ترین مسیر از گره A به گره B کدامه؟

شما یکبار وقتی میخواستید کوتاه ترین مسیر از توابع پیکس به پل گلدن گیت را محاسبه کنید، جستجوی اول عمق را دیدیدن. که سوالی از نوع سوال دوم بود: "کوتاه ترین مسیر کدام؟" حالا بیاین با جزئیات بیشتری از نالگوریتم را بررسی کنیم. شما یک سوال از نوع سوال یک میپرسین: "آیا راهی وجود دارد؟".

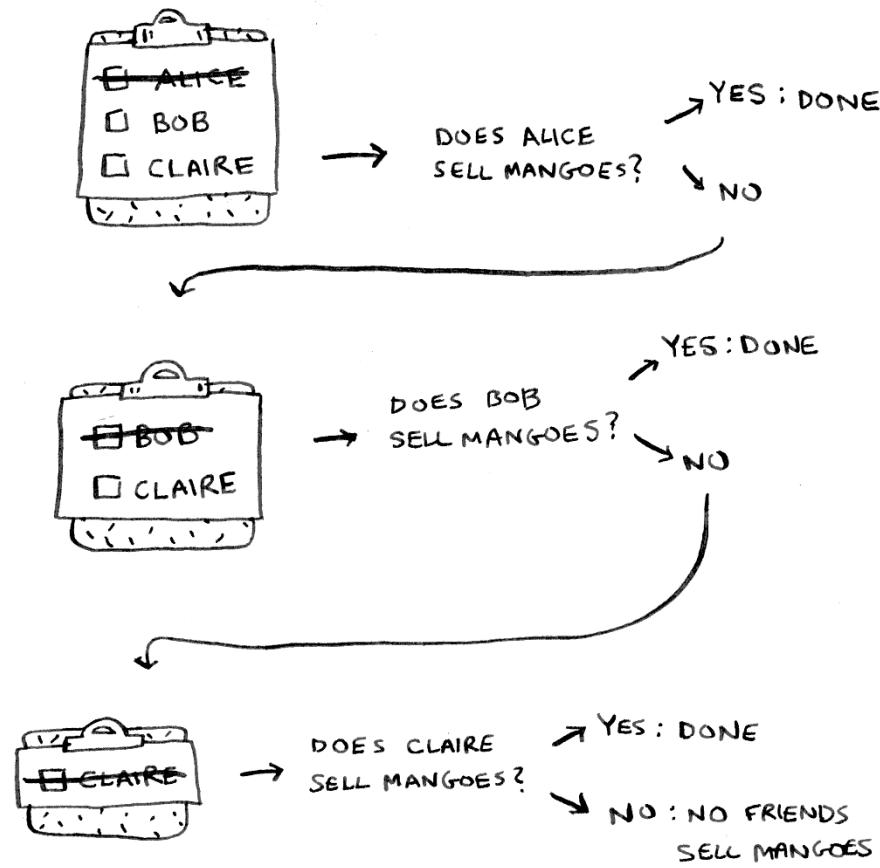


فرض کنید شما دارنده افتخاری یک مزرعه انبه هستیدن. شما دنبال یک فروشنده انبه میگردیدن که بتوانه انبه های شمارو بفروشه. آیا در فیس بود با یک فروشنده انبه در ارتباط هستیدن(فروشنده انبه در فیس بود دور برو برتون هست؟؟) خب ، شما میتوانید بین دوستانتون جستجو کنیدن.

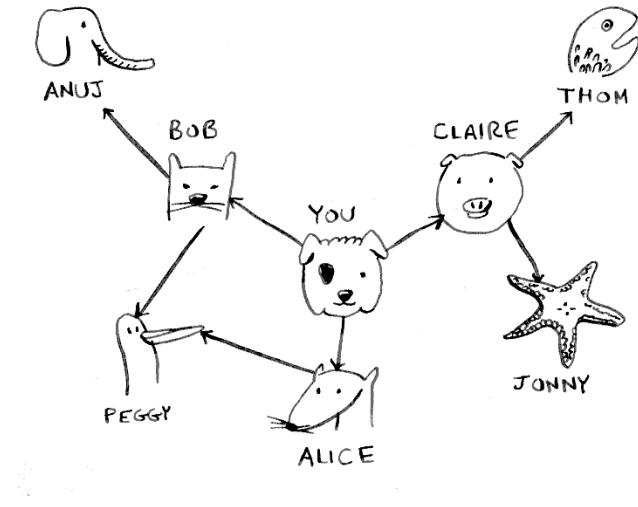


این جستجو خیلی سر راست و سادس. اول یک لیست از دوستانتون برای جستجو کردن تهیه کنیدن.

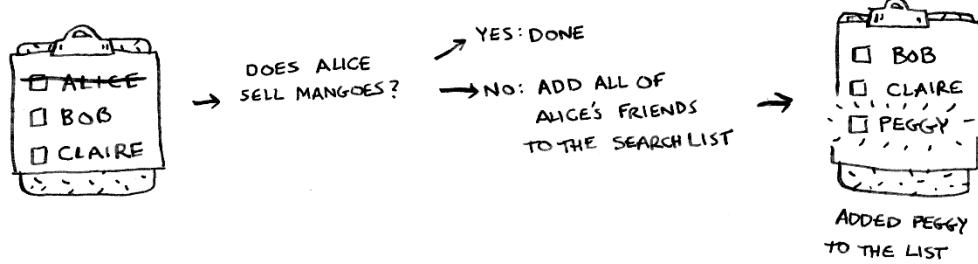
حالا به لیستتون مراجعه کنیدن و بررسی کنیدن ببینیدن که کدام یک از این افراد انبه میفروشن.



فرض کنین هیچ کدوم از دوستان شما، انبه فروش نیستند. حالا باید در میان دوستان دوستانتون جستجو کنین که ببین آیا انبه فروش پیدا میکنین یا نه.



هر بار که شما دارین کسی رو در لیست بررسی میکنین، لیست دوستانشون هم در لیست، برای بررسی کردن قرار بدین.



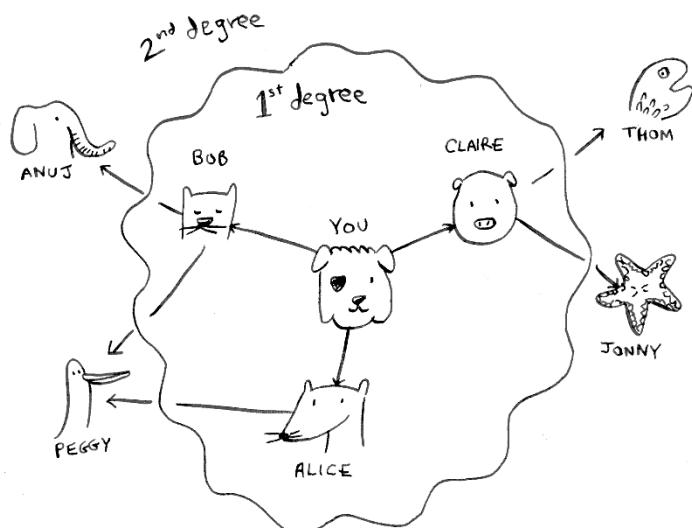
با این راه نه تنها دوستان خودتون رو بررسی میکنین بلکه دوستان دوستانتون رو هم بررسی میکنین. یادتون باشه، هدف اینکه که یک انبه فروش در شبکه دوستان خودتون پیدا کنین. پس اگر مثلاً آلیس، انبه فروش نیست، دوستان آلیس رو هم به لیست اضافه میکنین. که یعنی شما نهایتاً دوستان آلیس رو بررسی میکنین و بعدش دوستان دوستان اون رو والی آخر. با استفاده از این الگوریتم شما کل شبکه دوستانتون رو تا زمانی که با یک انبه فروش موجه نشدید بررسی میکنید. این الگوریتم همون جستوجوی اول عمقه.

پیدا کردن کوتاه ترین مسیر

برای خلاصه: این ها دو سوالی هستند که جستوجوی اول عمق میتوانه براتون جواب بدنه:

- سوال نوع اول: آیا راهی از گره A به گره B وجود داره؟ (آیا انبه فروشی در شبکه دوستان شما وجود داره؟)
- سوال نوع دوم: کوتاه ترین مسیر از گره A به گره B کدامه؟ (نزدیک ترین انبه فروش چه کسیه؟)

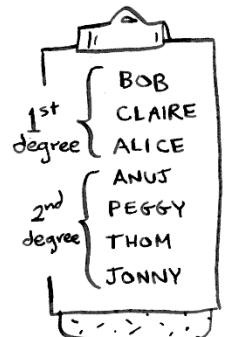
شما دید که چطور باید به سول اول پاسخ بدین. حالا بیاین سعی کنیم سوال ۲ رو پاسخ بدیم. آیا میتوانیم نزدیک ترین انبه فروش رو پیدا کنیم؟ به عنوان مثال، دوستان شما ارتباطات درجه یک هستند، و دوستان اونا ارتباطات درجه دو.



شما قطعاً ارتباط درجه اول رو به ارتباط درجه دوم و ارتباط درجه دوم رو به ارتباط درجه سوم ترجیح میدین. پس با این اوصاف شما نباید ارتباطات درجه دوم رو قبل از اینکه مطمئن بشین در ارتباطات درجه اولتون انبه فروشی نیست بررسی کنین. خب، جستوجوی اول عمق این کار رو از قبل برای شما انجام میده. جستوجوی عمق اول از نقطه شروع، بررسی کردن رو آغاز میکنه. پس با این کار اول ارتباط درجه اولتون رو بررسی میکنین بعدش ارتباطات درجه دوم. یک تست کوچک: کی اول بررسی میشه کلر یا آنجو(Clair or Anju) جواب: کلر جز ارتباطات درجه اوله و آنجو در ارتباطات درجه دوم. پس کلر قبل از آنجو بررسی میشه.

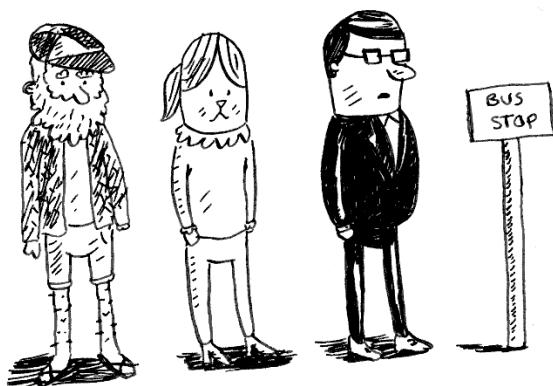
یک راه دیگه برای انجام این کار اینکه، ارتباطات درجه اول رو قبل از ارتباطات درجه دوم در لیست بررسی قرار بدم.

شما همینطوری که روبه پایین میرین، لیست رو بررسی میکنین که بینین آیا انبه فروشی پیدا میکنین. ارتباطات درجه اول قبل از ارتباطات درجه دوم بررسی میشن. پس با این کار شما نزدیک ترین انبه فروش به خودتون رو پیدا میکنین. جستوجوی عمق اول نه تنها راهی از نقطه A به نقطه B پیدا میکنه بلکه کوتاه ترین مسیر بین آنها رو هم پیدا میکنه.

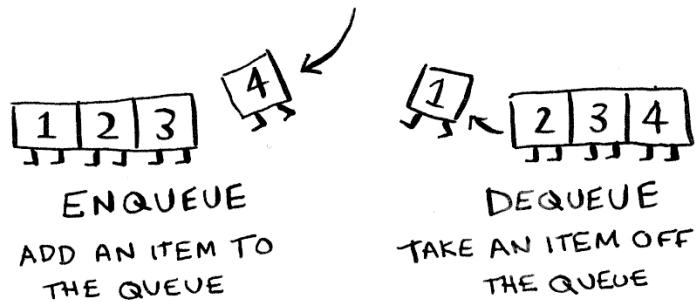


توجه کنین که این راه تنها زمانی کار میکنه که شما افراد رو به همون ترتیبی که به لیست اضافه شدن بررسی کنین. مثلاً اگر کلر قبل آنجو به لیست اضافه شده، پس کلر باید قبل از آنجو بررسی بشه. چه اتفاقی میفته اگر آنجو قبل از کلر بررسی بشه و جفتشون هم انبه فروش باشن؟ خب، همونطور که میدونین آنجو در ارتباطات درجه دومه و کلر در ارتباطات درجه اول، که با این کار شما انبه فروشی پیدا میکنین که نزدیک ترین انبه فروش به شما نیست. پس درنتیجه شما باید افراد رو به همون ترتیبی که اضافه شده، بررسی کنین. یک ساختار داده برای اینکار هست که بهش صف گفته میشه.

صف ها

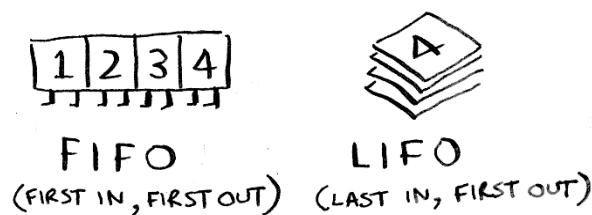


صف ها دقیقاً اونطوری که توی زندگی واقعی ما عمل میکنن، کار میکنن. فرض کنین شما و دوستانتون در ایستگاه اتوبوس صف میکشین. اگر شما جلوتر از دوستتون در صف باشین، شما زودتر سوار اتوبوس میشین. صف ها هم (queue) به همین شکل کار میکنن. صف ها شبیه استک ها هستن. شما نمیتوانین به یک عنصر به طور رندم در صف ها دسترسی داشته باشین. به جاش دو عمل در صف ها انجام میشه، وارد کردن در صف(enqueue) و خارج کردن از صف(dequeue).



اگر دو آیتم رو به لیست اضافه کنیں (enqueue)، اولین آیتمی که به لیست اضافه کرده بودین قبل از اضافه شدن آیتم دوم، از لیست خارج میشه (dequeue). شما میتوانید از اینکار برای لیستی که میخواهیں بررسی کنید استفاده کنید. افرادی که اول از همه به لیست اضافه میشنوند، اول از همه هم بررسی شده و از لیست خارج میشنوند.

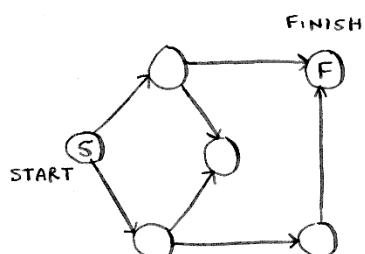
به صفر، ساختار داده فیفو (FIFO: first in first out) گفته میشه، که یعنی چیزی که اول وارد میشه، اول هم خارج میشه. برخلاف این موضوع، به استک، ساختار داده لیفو (LIFO: last in first out) گفته میشه، که یعنی چیزی که آخر وارد میشه، اول خارج میشه.



حالا که میدونیدن صف ها چطور کار میکنند، بیاین جستوجوی عمق اول رو پیاده سازی کنید.

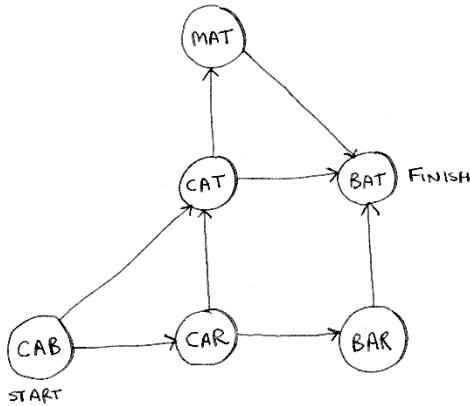
تمرین ها

جستوجوی عمق اول رو روی هر کدام از این گراف ها اجرا کنید تا راه حل رو پیدا کنید.



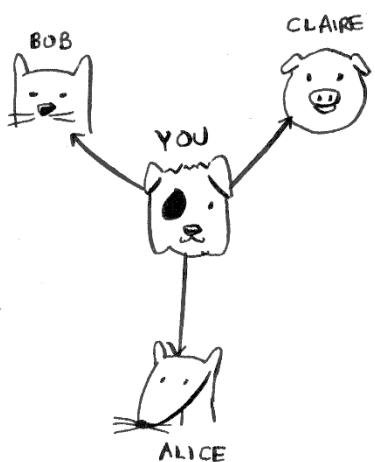
۱. طول کوتاه ترین مسیر رو از نقطه شروع تا پایان پیدا کنید.

۲. طول کوتاه ترین مسیر از "bat" به "cab" رو پیدا کنید.



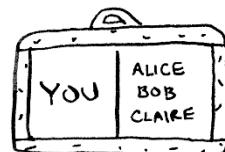
بکارگیری گراف

اول از همه، نیازه که گراف رو در کد پیاده سازی کنیم. یک گراف شامل چندین گره میشه.



و هر گره به گره همسایه خود متصله. چطور ارتباطی رو مثل "شما ← باب" پیاده میکنیں؟ خوشبختانه شما ساختار داده ای رو بلد هستین که اجازه پیاده سازی این کار رو به شما میده: جدول هش.

یادتون باشه، یک جدول هش این اجازه رو به شما میده که یک کلید رو به یک مقدار مرتبط کنین. در این مثال شما میخواین یک گرفه رو به همسایه هاش مرتبط کنین.



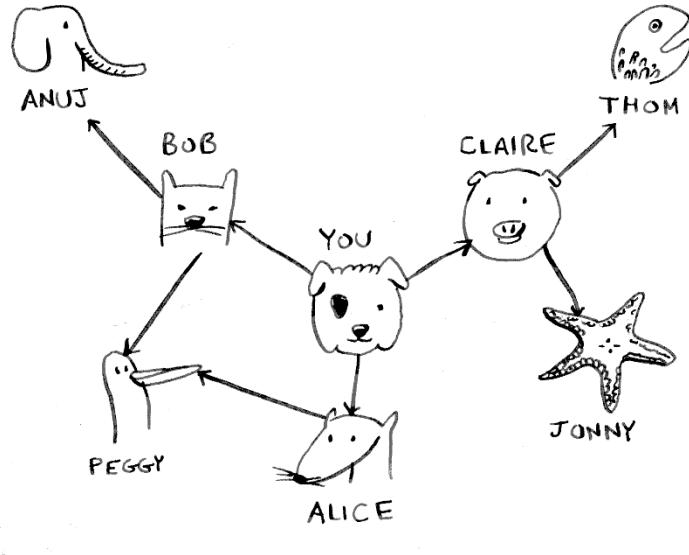
این هم روش پیاده سازیش در پایتون.

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

توجه کنین که "you" به یک آرایه، مرتبط شده(نگاشت شده).

در نتیجه [“you”] به شما یک آرایه از تمام همسایه های "you" میدهد.

یک گراف چیزی جز چند گره و یال نیست ، پس اینا تنها چیز هایی هستند که شما در پایتون به آنها نیاز دارین. درمورد گراف های بزرگ تر چطور مثل شکل صفحه بعد چطور؟



اینم کد پایتون این گراف:

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

یک تست کوچیک: آیا اهمیت داره که کلید ها و مقادیر رو به چه ترتیبی به جدول هش اضافه کنیم. آیا مهمه که اینطوری بنویسیم:

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

به جای اینطوری:

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

به فصل قبل فک کنین. جوابش اینه که: نه اهمیتی نداره. جداول هش هیچ ترتیبی ندارن پس اهمیتی نداره که به چه ترتیبی اطلاعات رو وارد میکنین.

آنجو، پگی، تام، و جان هیچ همسایه ای ندارن. اونها فلش هایی دارن که داره بھشون اشاره میکنه اما هیچ فلشی از اونا به بقیه وجود نداره. به این مدل گراف، گراف جهت دار میگن. ارتباط فقط یک طرفه. پس آنجو همسایه بابه. اما باب همسایه آنجو نیست. در گراف غیر جهت دار هیچ فلشی وجود نداره و هر دو گره همسایه هم هستند به عنوان مثال هر دوی این گراف ها با هم برابر هستند:



DIRECTED
GRAPH



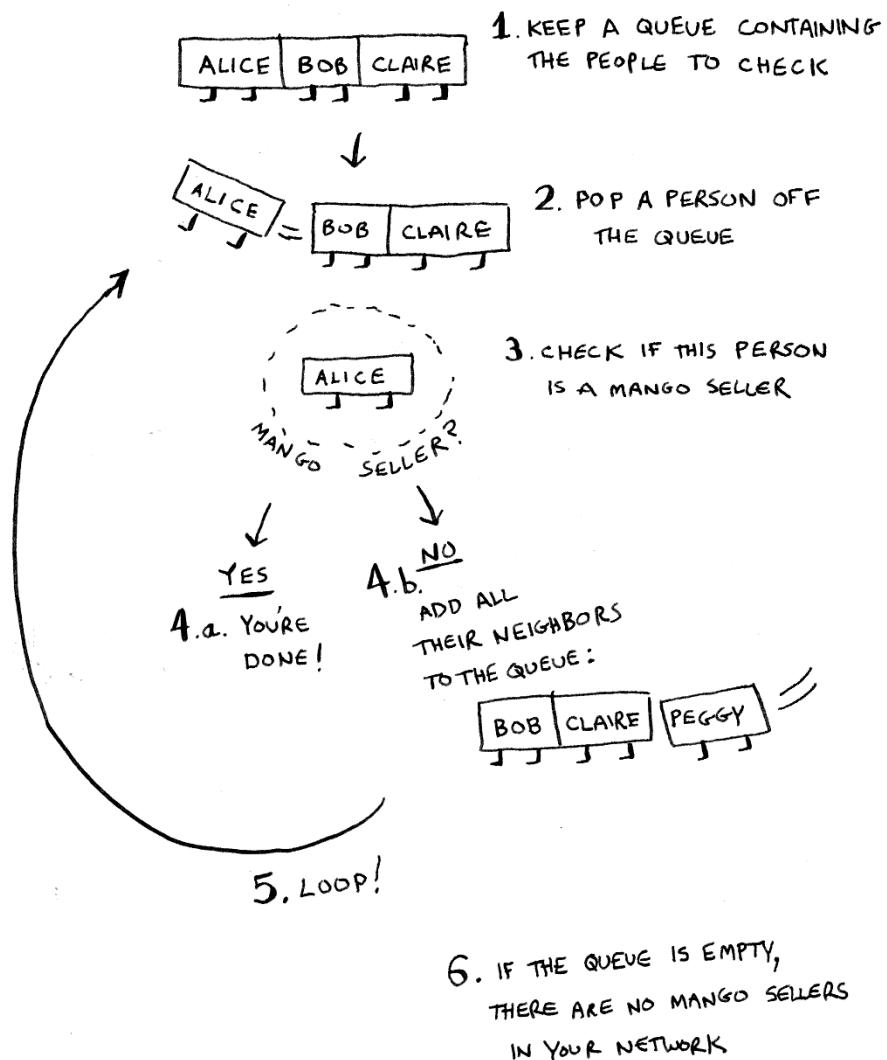
UNDIRECTED
GRAPH

بکار گیری الگوریتم

برای خلاصه، اینکه چطور پیاده سازی این الگوریتم کار میکنه به این شکله:

یادداشت

وقتی که صف هارو بروزرسانی میکنیم من از اصطلاح انکیو (enqueue) و دیکیو (dequeue) استفاده میکنم. همچنین ممکنه با اصطلاح پاپ (push) و پوش (pop) هم مواجه بشین. پوش تغییر با همیشه با انکیو هم معنیه و پاپ هم تقریبا همیشه با دیکیو هم معنیه.



یک صف رو برای شروع درست کنید. در پایتون ، شما از تابع دابل انده کیو (dequeue) برای اینکار استفاده میکنید:

```

from collections import deque
search_queue = deque() <----- Creates a new queue
search_queue += graph["you"] <----- Adds all of your neighbors to the search queue

```



یادتون باشه، `graph["you"]` یک لیست از تمام همسایه های شما در اختیار تون میزاره، تمام اونها به صف برای بررسی اضافه میشن. مثل `["alice", "bob", "claire"]`. بقیه کد رو ببینیم:

```

while search_queue: <----- While the queue isn't empty ...
    person = search_queue.popleft() <----- ... grabs the first person off the queue
    if person_is_seller(person): <----- Checks whether the person is a mango seller
        print person + " is a mango seller!" <----- Yes, they're a mango seller.
        return True
    else:
        search_queue += graph[person] <----- No, they aren't. Add all of this
return False <----- If you reached here, no one in           person's friends to the search queue.
                  the queue was a mango seller.

```

و در آخر، شما به تابع `person_is_seller` نیاز دارین تا بهتون بگه که کی انبه فروش هستش. این یکی از چند منطقی هستش که میتوانیم برای این تابع پیاده کنیم:

```

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

```

این تابع بررسی میکنه که آیا اسم فرد با حرف `m` تموم میشه یا نه. اگر که آره، اون فرد انبه فروش هست. این یک جوابایی یک راه احتمانه برای انجام اینکاره، اما خب برای این مثال کار ما رو راه میندازه. خب حالا ببین جستجوی عمق اول رو در عمل ببینیم.

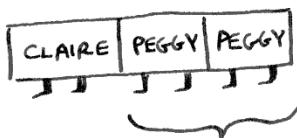


...etc...

و الی آخر. این الگوریتم اونقدر پیش میره تا بالآخره:

- یا یک آنبه فروش پیدا میکنه، یا
- صف خالی میشه، که دراون حالت هیچ آنbeh فروشی وجود نداره.

آلیس و باب، یک دوست مشترک دارن: پگی. پس پگی دوبار به صف اضافه میشه: یک بار وقتی که دارین دوستای آلیس رو به صف اضافه میکنین و دیگه هم وقتی که دارین دوستای باب رو اضافه میکنین. و درنهای شما در صفتون با دو تا پگی رو به رو میشین.



UH OH, PEGGY IS IN
THE SEARCH QUEUE

TWICE!

ولی شما فقط نیاز دارین که یکبار پگی رو چک کنین که ببینین آیا انبه فروش هست یا نه. اگر دوبار چکش کنین ، درواقع دارین کار غیرضروری و بیهوده انجام میدین. پس وقتی که یکبار کسی رو بررسی کردین، شما باید اونها رو به عنوان "بررسی شده" علامت بزنین تا دوباره بررسی نشن.

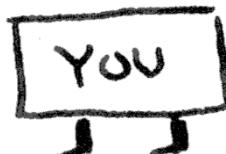
اگر این کارو نکنین، میتوانه موجبه این بشه که شما در یک حلقه بینهايت بیفتین. فرض کنین گراف انبه فروش به این شکله:



برای شروع، صفى که قرار درست کنیم تا بررسیشون کنیم شامل تمام همسایه های شما میشن.



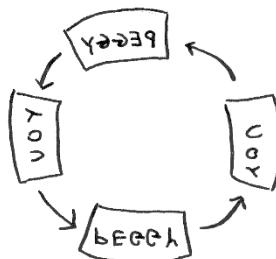
حالا شما پگی رو بررسی میکنین، خب میفهمین که اون انبه فروش نیست ، پس همسایه ای اون رو به صف اضافه میکنین.



پس بعد از اون شما خودتون رو بررسی میکنین. شما انبه فروش نیستین، پس همسایه شما به صف اضافه میشه.



و این روند همینطور ادامه داره. این یک حلقه بینهايت میشه، به این دلیل که باید برای پیدا کردن انبه فروش بررسی کنیم اول شامل پگی میشه بعدهش شما و مدام بین جفتتون جابجا میشه.



قبل از بررسی کردن کسی، خیلی مهمه که چک کنیم ببینیم آیا قبلا بررسی شده یا خیر. برای اینکار شما لیستی از کسانی که قبلا چک کردین رو در اختیار دارین.

خب اینم کد نهایی با درنظر گرفتن همین موضوع:

```
def search(name):
    search_queue = deque()
    search_queue += graph[name]
    searched = [] This array is how you keep track of  
which people you've searched before.
    while search_queue:
        person = search_queue.popleft()
        if not person in searched: Only search this person if you  
haven't already searched them.
            if person_is_seller(person):
                print person + " is a mango seller!"
                return True
            else:
                search_queue += graph[person] Marks this person as searched
                searched.append(person)
    return False

search("you")
```

سعی کنین این کدو خودتون اجرا کنین. شاید بهتر باشه تلاش کنین یک منطق با معنی تر برای تابع `person_is_seller` بنویسین و ببینین آیا اون چیزی رو که شما میخواین چاپ میکنه یانه.

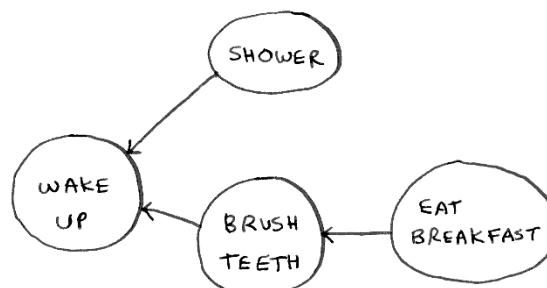
زمان اجرایی

اگر شما کل شبکه دوستانتون رو برای انبه فروش بررسی میکنین، معینش این میشه که شما تمام یال ها در گراف (یادتون باشه، یال، فلش یا ارتباطی از یک فرد به فرد دیگر است) رو دنبال میکنین. خب زمان اجرایی حداقل این مقداره: (تعداد یال ها) O .

همچنین شما یک صفحه افراد برای بررسی کردن در اختیار دارین. اضافه کردن یک نفر به صفحه اندازه $O(1)$ زمان میبرد. انجام این کار برای هر فرد به اندازه (تعداد هر فرد) O جمعا زمان میبره. جستجوی اول عمق به اندازه $(\text{تعداد یال ها} + \text{تعداد افراد})O$ زمان میبره. و به طور به معمول به این شکل نوشته میشه $O(V + E)$.

v: برابر `vertices` در انگلیسی به معنیه رئوس و E : برابر `edges` به معنیه یال در گراف میباشد).

تمرین



این گرافی که میبین رو تین صباحی منه.

این گراف بهتون میگه، من نمیتونم تا زمانی که دندونامو مسوак نزدم، صبحانه بخورم. پس "خوردن صبحانه" به "مسواک زدن" بستگی دارد. از طرفی، دوش گرفتن به مسوак زدن بستگی نداره چون من میتونم قبل از مسواك زدن، دوش بگیرم. از روی این گراف، شما میتوانین لیستی از ترتیب کارایی که من برای روتین صبحا باید انجام بدم رو درست کنین:

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

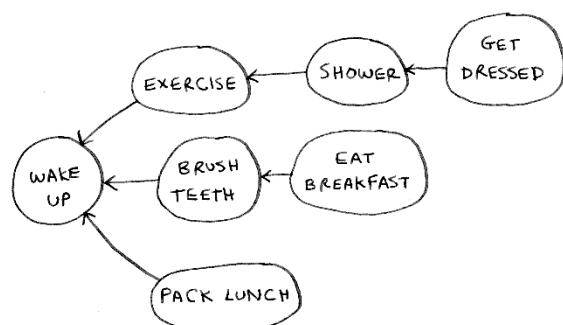
حواستون باشه که "دوش گرفتن" میتوانه در لیست جابه جا بشه (چون هیچ وابستگی نداره)، درنتیجه این لیست هم درسته:

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

۶.۳ برای سه لیست زیر، مشخص کنین که کدامشون درسته و کدامشون درست نیست.

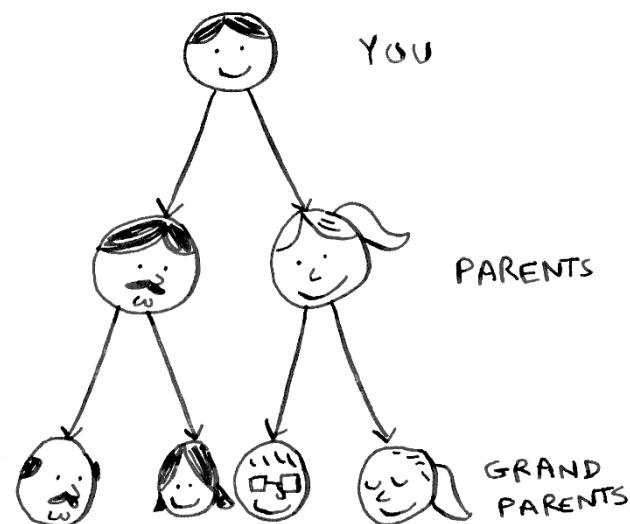
A.	B.	C.
1. WAKE UP	1. WAKE UP	1. SHOWER
2. SHOWER	2. BRUSH TEETH	2. WAKE UP
3. EAT BREAKFAST	3. EAT BREAKFAST	3. BRUSH TEETH
4. BRUSH TEETH	4. SHOWER	4. EAT BREAKFAST

۶.۴ اینجا یک گراف بزرگتر داریم. یک لیست درست و مورد قبول برای این گراف درست کنین.

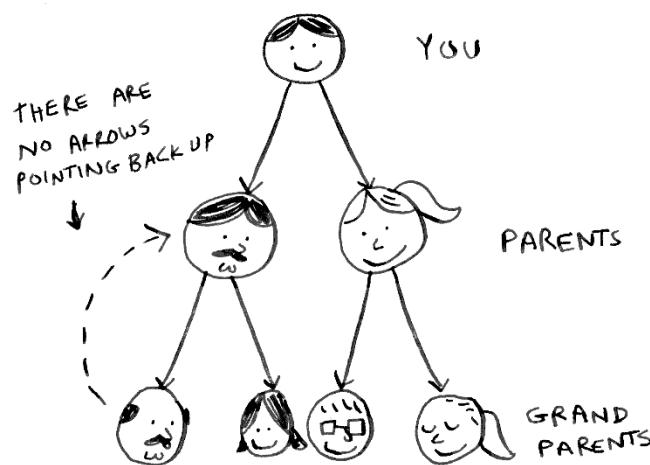


شما میتوانید یه جو رایی بگین این لیست ، یک لیست مرتب شدش. اگر تسك A به تسك B بستگی داشته باشد پس تسك A خودش رو در آخر لیست (جایی بعد از تسك B) نشون میده. به این عمل میگن مرتب سازی توپولوژیکی (topological sort) فرض کنین شما دارین برنامه یک عروسی رو میچینین و یک گراف بزرگ ، پر از تسك ها و کارایی که باید انجام بدین دارین و مطمئن نیستید که باید از کجا شروع کنین. شما میتوانید به صورت توپولوژیکی گراف رو مرتب کنین و بعدش شما به یک لیست دارای ترتیب از کارهایی که باید انجام بدین میرسین.

فرض کنین شما یک شجره نامه دارین



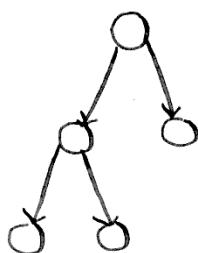
این شجره نامه، یک گراف محسوب میشه ، به این دلیل که شما هم گره دارین(افراد) و هم یال. هر یار به پدر و مادر یک گره اشاره میکنه. اما همه یال ها به سمت پایین هستن، معنی نداره اگر یالی به بالا اشاره کنه! که این اگر اتفاق بیفته یک چیز کاملا بی معنی میشه ، مثلا پدر شما نمیتونه پدر پدر بزرگتون باشه!



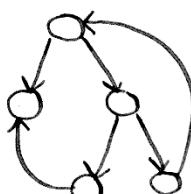
به این طور گراف ها درخت می‌گن. یک درخت یک نوع خاصی از گراف ها هستند، که هیچ گره‌ای به عقب خودش اشاره نمی‌کنه.

۶.۵ کدام یک از گراف‌های زیر هم درخت محسوب می‌شود؟

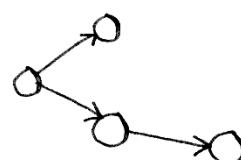
A.



B.

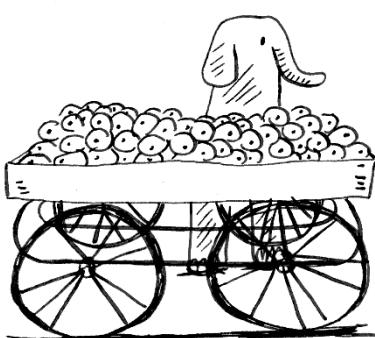


C.



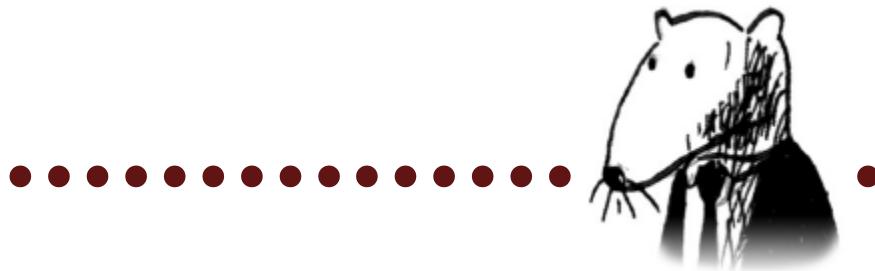
خلاصه

- جستجوی اول عمق بهتون می‌گه که آیا راهی از A به B وجود داره یا نه.
- اگر راهی وجود داشت، جستجوی اول عمق کوتاه ترین مسیر رو پیدا می‌کنه.
- اگر مسئله‌ای مثل "پیدا کردن کوتاه ترین X" دارین، سعی کنین مسئله رو به صورت یک گراف مدل کنین، و از جستجوی اول برای حلش استفاده کنین.
- گراف جهت دار دارای فلش هستش، و جهت فلش نحوه ارتباط بین دو نقطه رو مشخص می‌کنه.(rama - <adicity یعنی "rama به adicity بده کاره").
- گراف‌های غیر جهت دار، فلش ندارن. و نحوه ارتباط بین دو نقطه دو طرفس. (راس - ریچل یعنی "راس با ریچل قرار گذاشت و ریچل با راس قرار گذاشت"(Richest and Rascist))
- صفحه‌ها FIFO-First in first out چیزی که اول وارد می‌شود اول خارج می‌شوند.
- استک‌ها LIFO - last in first out چیزی که آخر وارد می‌شود، اول خارج می‌شوند.
- شما باید افراد رو با همون ترتیبی که وارد لیست شدن بررسی کنین، درنتیجه لیستی که میخواین بررسی کنین بحالت صفحه‌هایی دریابین. در غیر اینصورت، شما نمیتوانید کوتاه ترین مسیر رو پیدا کنین.
- وقتی یک نفو چک کردین بعدش مطمئن بشین که دوباره چکش نمکنین. در غیر اینصورت ممکنه در یک حلقه بینهایت گیر بیفتین.



الگوریتم دایجسترا

(Dijkstra's algorithm)

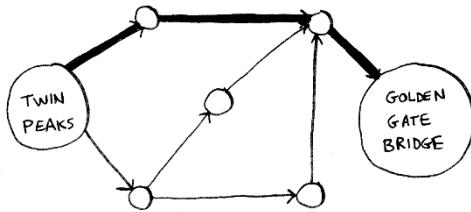


در این فصل:

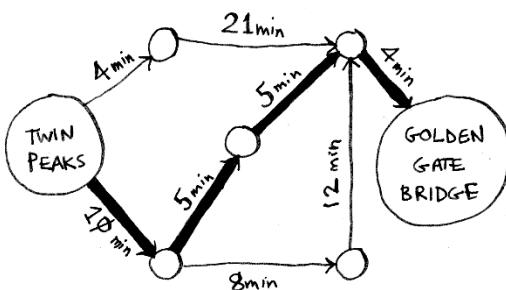
- بحثمون رو درباره گراف ها ادامه میدیم، و شما درباره گراف های وزن دار یادخواهید گرفت: یک برای اختصاص دادن وزنی زیاد یا کم به تعدادی از یال ها.
- در مورد الگوریتم دایجسترا یادخواهید گرفت، که بهتون در این اجازه رو میده که بتونین این سوال رو در باره گراف های وزن دار جواب بدین "کوتاه ترین مسیر به سمت X کدام است؟".
- شما درباره دور در گراف ها یادخواهید گرفت، که الگوریتم دایجسترا در اون کار نمیکنه.



در فصل قبل، شما یادگرفتین که چطور از نقطه A به نقطه B بین.



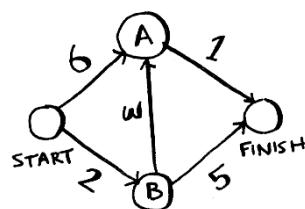
اما این راه لزوما سریعترین راه نیست. این کوتاه ترین راهه. ، چرا که شامل کمترین بخش هستش(سه بخش). اما فرض کنین ما میزان زمانی که هر بخش طول میکشه تا طی بشه رو بهشون اضافه کنیم. حالا شما میتوانین ببینین که راه سریع تری هم هست.



شما از جستجوی اول عمق در فصل قبل استفاده کردین. جستجوی اول عمق راهی رو که شامل کمترین تعداد بخش باشے رو برای شما پیدا میکنه(اولین گرافی که در اینجا نشون داده شد). اما اگر به جای این شما دنبال سریعترین راه باشین چی(گراف دوم)? شما میتوانین برای پیدا کردن سریع ترین راه از الگوریتمی به نام الگوریتم دایجسترا استفاده کنین.

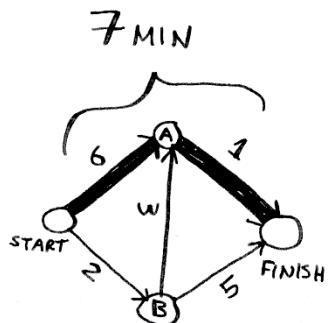
کار کردن با الگوریتم دایجسترا

بیاین ببینیم این الگوریتم روی این گراف چطور کار میکنه.



هر بخش زمان مورد نیاز برای طی کردنشو داره. شما از الگوریتم دایجسترا برای طی کردن مسیر از نقطه شروع تا نقطه پایان در کوتاه ترین زمان ممکن، استفاده میکنین.

اگر شما جستجوی اول رو روی این گراف اجرا میکردین، به این مسیر کوتاه در این گراف میرسیدید.

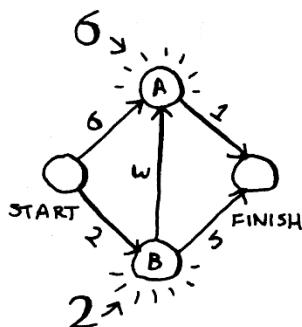


اما طی کردن اون را ۷ دقیقه طول میکشه

بیاین ببینیم میتوانیم راهی رو پیاده کنیم که کمتر از این مقدار طول بکشه. چهار قدم برای اجرا الگوریتم دایجسترا وجود دارد:

۱. پیدا کردن "ارزان ترین" گره. این گره، گره ایه که شما میتوانین در کمترین زمان ممکن بهش برسین.
۲. بروزرسانی کردن هزینه های همسایه های این گره. برآتون توضیح میدم که منظورم از این جمله کوتاه چیه.
۳. این کارو تا زمانی که برای تمام گره ها نکردیم، تکرار میکنیم.
۴. محاسبه کردن مسیر نهایی.

قدم اول: پیدا کردن ارزانترین گره. شما در نقطه شروع ایستاده اید. و بین انتخاب گره A و B مردد هستید. چقدر طول میکشه تا به هر کدام از گره ها برسین؟

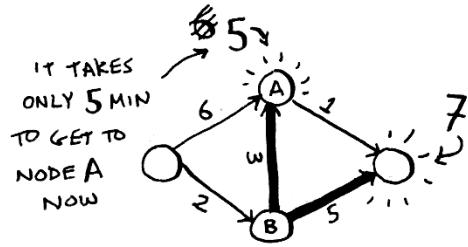


۶ دقیقه طول میکشه تا به گره A برسین و ۲ دقیقه تا به B برسین. اینکه چقدر طول میکشه تا به بقیه گره ها رو برسین هنوز مشخص نیست و اینو نمیدونین.

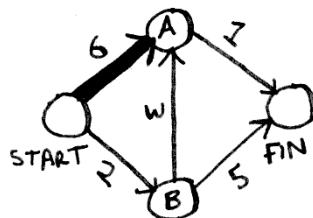
NODE	TIME TO NODE	TIME TO NODE
A	6	6
B	2	2
FINISH	∞	∞

قدم دوم: حساب کنین چقدر طول میکشه تا از یالی که به B ختم میشه به همسایه های دیگه B برسیم.

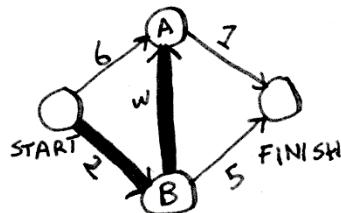
NODE	TIME
A	5
B	2
FINISH	7



ا، شما یک راه سریعتر برای رسیدن به گره A پیدا کردین! قبل از این ۶ دقیقه طول میکشید تا به گره A برسیم.



اما اگر از گره B به گره A ببریم فقط ۵ دقیقه طول میکشه.



وقتی راهی کوتاه تر برای همسایه های B پیدا کردین، هزینه رسیدن به هر گره رو بروزرسانی کنین. در این حالت شما اینارو پیدا کردین:

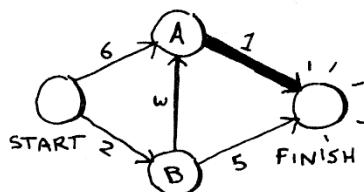
- یک راه کوتاه تر به A (از ۶ دقیقه به ۵ دقیقه کاهش پیدا کرد)
- یک راه کوتاه تر به نقطه پایان (از بینهایت دقیقه به ۷ دقیقه کاهش پیدا کرد)

قدم سوم: همین قدم هارو تکرار کنین.

تکرار قدم اول: یک گره رو پیدا کنین که کمترین زمان رو برای رسیدن بهش باید صرف کنیم. شما کارتون با B تموم شده، پس گره A کمترین زمان رو بعداز گره B از ما میگیره.

NODE	TIME
A	5
B	2
FINISH	7

تکرار قدم دوم: هزینه های رسیدن به همسایه های گره A را که از طریق همین گره بهشون میرسیم رو بروزرسانی کنیم.



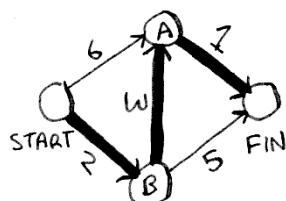
او، با این کار متوجه میشیم که ۶ دقیقه طول میکشه که ما به نقطه پایان برسیم!

شما همین الان الگوریتم دایجسترا رو برای همه گره ها اجرا کردین (نیاز نیست که این الگوریتم برای نقطه پایان اجرا بشه. در این نقطه شما میدونین که:

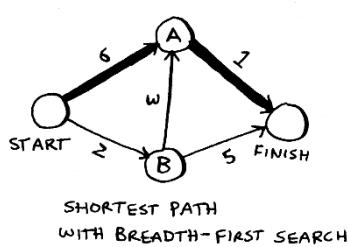
- ۲ دقیقه طول میکشه که به گره B برسیم.
- ۵ دقیقه طول میکشه که به گره A برسیم.
- ۶ دقیقه طول میکشه تا به گره پایان برسیم.

NODE	TIME
A	5
B	2
FINISH	6

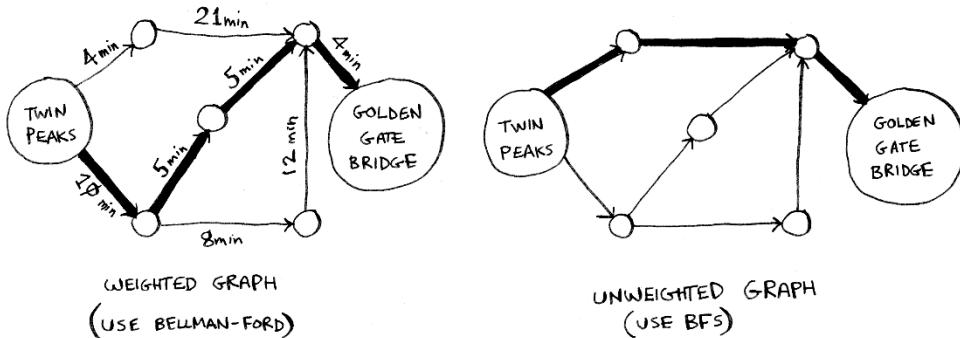
نحوه محاسبه کردن مسیر نهایی رو برای بخش بعد نگه میدارم، اما فعلاً بهشتون مسیر نهایی رو نشون میدم.



به دلیل ۳ بخشی بودن این مسیر، جستجوی اول عمق این مسیر کوتاه رو پیدا نمیکرد. و یک راه دو بخشی برای رسیدن به نقطه پایان از نقطه شروع وجود دارد.



در فصل قبل ، شما از جستجوی اول عمق برای پیدا کردن کوتاه ترین مسیر بین دو نقطه استفاده میکردیدن. در اونجا معنی "کوتاهترین مسیر" مسیری بود که شامل کمترین بخش باشد. اما در الگوریتم دایجسترا، شما یک عدد یا یک وزن به هر بخش اختصاص میدین و الگوریتم دایجسترا مسیری که جمع وزن بخشها یش کوچک تر از همه باشد را پیدا میکنه.



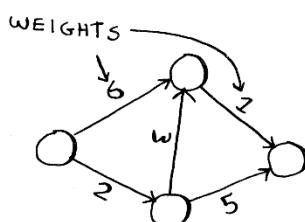
برای جمع بندی، الگوریتم دایجسترا شامل ۴ قدم میشه:

۱. پیدا کردن ارازن ترین گره. این گره ، گره ایه که شما در کمترین زمان میتوینین بهش برسین.
۲. بررسی کنین که آیا راه ارزانتری برای رسیدن به همسایه های این گره هست. اگر آره، هزینه هارو بروزرسانی کنین.
۳. این کارو برای تمام گره در گراف تکرار کنین.
۴. مسیر نهایی رو محاسبه کنین.(که در بخش بعدی درموردش حرف میزنیم)

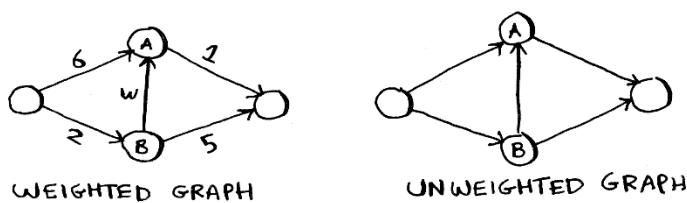
اصطلاحات فنی

میخوام مثال های بیشتری از الگوریتم دایجسترا در عمل، برآتون بزنم. اما بزارین اول برآتون یک سری اصطلاحات رو روشن کنم و توضیحشون بدم.

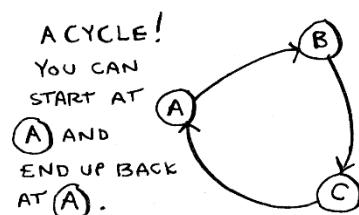
وقتی دارین با الگوریتم دایجسترا کار میکنین، به هر یال در گراف یک عدد اختصاص داده شده که بهشون میگن وزن.



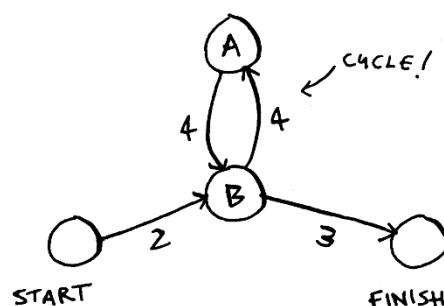
یک گراف که شامل وزن میشه رو بهش میگن گراف وزن دار. یک گراف بدون وزن رو گراف غیروزن دار صدا میزنن.



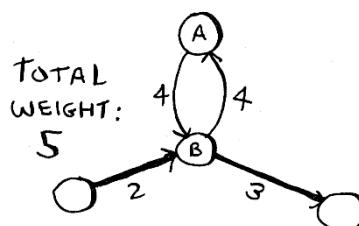
برای محاسبه کردن کوتاه ترین مسیر در گراف های غیروزن دار از جستجوی اول عمق استفاده کنیم. و برای محاسبه ی کوتاه ترین مسیر در گراف های وزن دار از الگوریتم دایجسترا استفاده کنیم. گراف ها هم میتوانند گردش داشته باشند. یک گردش یا چرخش این شکلی میشه:



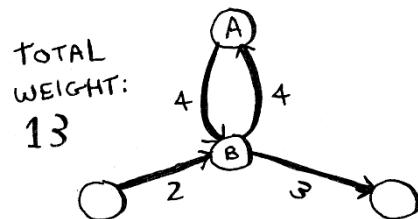
که این یعنی شما میتوانید از یک گره شروع کنید و مسیرها را طی کنید و دوباره به همان گره ای که ازش شروع کردیدن برسیدن. فرض کنید شما میخواهید کوتاه ترین مسیر را در این گراف که شامل چرخش میشه، رو پیدا کنید.



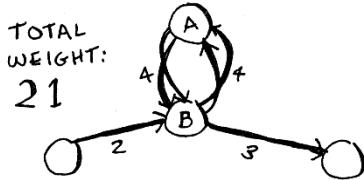
اصلاً منطقی هست که این حلقه چرخشی را دنبال کنید تا به کوتاه ترین مسیر برسیم؟ در واقع شما میتوانید از مسیری استفاده کنید که از اون حلقه دوری میکنید.



یا میتوانید مسیری که شامل حلقه میشه را دنبال کنید.

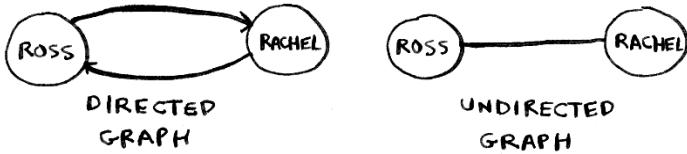


که اگر حلقه را دنبال کنید در هر صورت به گره A میرسیدن، اما این حلقه فقط باعث اضافه شدن وزن میشه. همچنین شما میتوانید اگر خواستیدن، دوبار حلقه رو طی کنیدن.

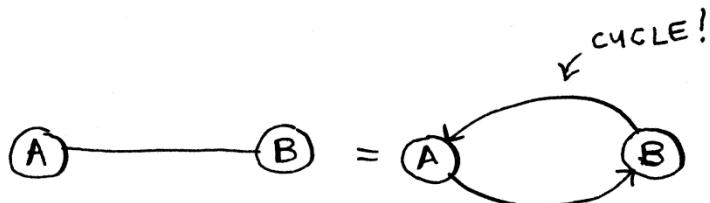


اما با هر بار طی کردن این حلقه شما فقط مقدار ۸ رو به وزن کلی مسیر اضافه ممکنیم. پس دنبال کردن حلقه هیچ وقت باعث پیدا کردن کوتاه ترین مسیر نمیشے.

در آخر ، بحثمون در باره گراف های جهت دار و غیر جهت دار که در فصل ۶ داشتیم رو یادتون هست؟



یک گراف غیر جهت دار یعنی هر دو گره متصل بهم، بهم دیگه اشاره میکنند. که این همون حلقه، گردش یا چرخش هستش.



در گراف های غیر جهت حساب میشه. الگوریتم دایجسترا تنها روی گراف های بی دور جهت دار (directed acyclic graphs) کار میکنه که به اختصار به این گراف ها DAG ها گفته میشه.

مبادله کالا برای بدست آوردن پیانو

دیگه اصطلاحات فنی بسه، بیاین یک مثال دیگ رو بررسی کنیم! این "rama" است.
rama میخواهد کتاب موسیقی شو برای گرفتن یک پیانو مبادله کنه.

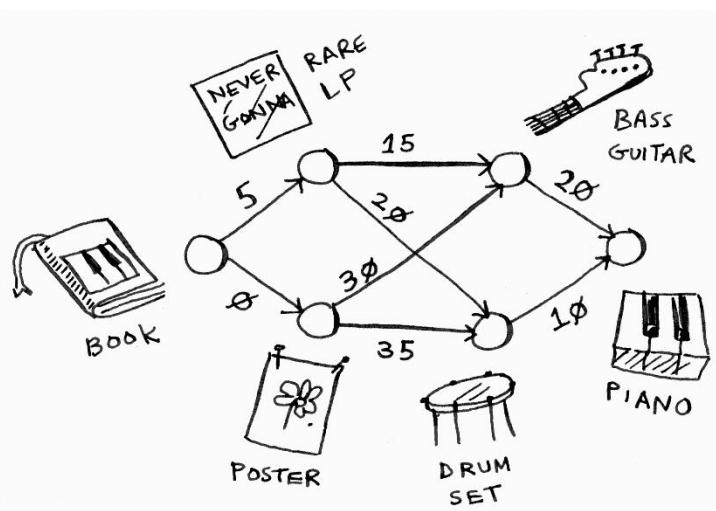


الکس میگه "من این پوستر رو در ازای کتابت بہت میدم". "این یک پوستر از گروه موسیقی محبوب منه، گروه دیسترویر (destroyer). یا بہت این ال پی کمیاب از ریک استلی (LP of Rick Astley) رو در ازای کتابت و ۵ دلار اضافه بہت میدم. "امی میگه "اوو، من شنیدم ال پی آهنگی عالی داره، من گیتارمو یا مجموعه دراممو در ازای کتابت یا ال پی بہت میدم".



بتهوون فریاد میزنه "من میخواستم گیتارو بگیرم!". "هی، من پیانومو برای هرچیزی از امی میگیری باهات مبادله میکنم."

عالی شد! با یکم هزینه کردن ، راما میتوانه از یک کتاب پیانو به یک پیانوی واقعی برسه. حالا اون باید بفهمه که چطور با کمترین هزینه این مبادله رو به سرانجام برسونه. بیاین چیزایی که به راما پیشنهاد شد رو به شکل گراف دربیاریم.



در این گراف ، گره ها همان آیتم هایی هستند که راما میتوانه اونها رو با کتاب مبادله کنه. وزن ها روی يالها مقدار پولی رو نشون میدن که راما باید برای بدست آوردن اون آیتم هزینه کنه. پس اون میتوانه پوستر رو با گیتار با همراه ۳۰ دلار معاوضه کنه. یا میتوانه ال پی رو با گیتار در ازای ۱۵ دلار معاوضه کنه. خب راما چطور میتوانه مسیری رو که با حداقل هزینه به پیانو میرسه رو پیدا کنه؟ الگوریتم دایجسترا اینجاست که مارو نجات بده! یادتون باشه الگوریتم دایجسترا ۴ گام داره. در این مثال شما تمام این ۴ گام رو انجام میدین و در آخر هم مسیر نهایی رو محاسبه میکنین.

NODE	COST
LP	5
POSTER	0
GUITAR	∞
DRUMS	∞
PIANO	∞

} WE HAVEN'T
REACHED
THESE NODES
FROM THE
START YET

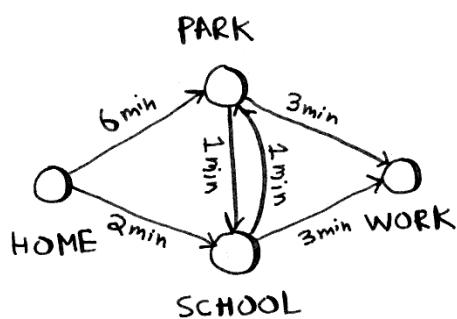
قبل از اینکه شروع کنیم، نیازه که اول یکسری چیز هارو آماده کنیم. یک جدول برای هزینه های هر گره درست کنیم. هزینه هر گره ، برابر مقدار پولیه که ما برای رسیدن به اون گره باید بپردازیم.

شما مدام این جدول رو تا زمانی که الگوریتم داره اجرا میشه ، بروزرسانی میکنین. همچنین شما برای محاسبه نهایی به یک ستون والد در این جدول نیاز دارین (مثلا اگر ما از طریق پوستر به درام برسیم، پوستر میشه والد درام).

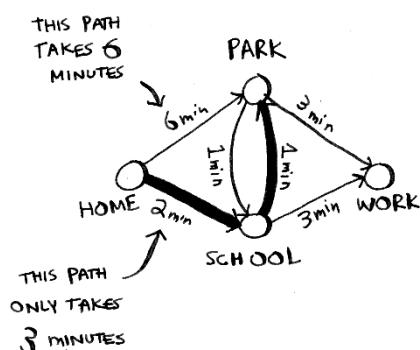
NODE	PARENT
LP	BOOK
POSTER	BOOK
GUITAR	—
DRUMS	—
PIANO	—

به زودی بهتون نشون میدم که این ستون چطور کار میکنه.

قدم اول: ارزون ترین گره رو پیدا کنین. در این مورد، پوستر ارزون ترین مبادله ایه که ما میتوانیم با صفر دلار داشته باشیم. آیا راه ارزون تری هست تا به پوستر برسیم؟ ایم یک نکته‌ی مهمه، پس خوب راجب‌ش فک کنین. آیا میتوانیم یک سری از مبادلات رو پیدا کنین که پوستر رو با قیمت کمتر از صفر دلار به راما برسونه؟ هر وقت آماده بودید به خوندن ادامه بدین. جواب: خیر. چرا که پوستر ارزون ترین گره ایه که راما میتوانه بهش برسه، هیچ راه ارزون تر از این برای رسیدن به پوستر وجود نداره. اینجا یک راه متفاوت برای بررسی کردن این موضوع بهتون ارائه میدم. فرض کنین شما میخواین از خونه به محل کارتون بین.



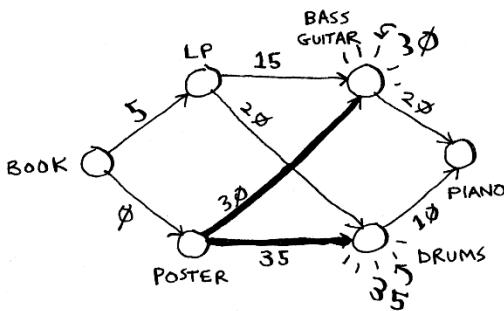
اگر شما مسیری رو که به سمت مدرسه سمت انتخاب کنین ۲ دقیقه طول میکشه تا بهش برسین. اگر مسیری رو انتخاب کنین که به سمت پارک میره ۶ دقیقه طول میکشه. آیا راهی وجود داره که ما بخوایم مسیرمون به مدرسه ختم بشه و از مسیر ۶ دقیقه ای پارک ببریم و کمتر از ۲ دقیقه وقتمنو بگیره؟ این غیرممکنه، چرا رسیدن به خود پارک بیشتر از دو دقیقه طول میکشه. حالا از طرف دیگه، آیا شما میتوانین راهی پیدا کنین که مارو سریع تر به پارک برسونه؟ بله.



این ایده کلیدی پشت الگوریتم دایجستراست: ارزون ترین گره رو در گراف پیدا کنین. هیچ مسیر ارزون تری برای رسیدن به اون گره وجود نداره!

برگردیم سر مثال موسیقیمون. پوستر ارزون ترین مبادله ایه که ما میتوانیم داشته باشیم.

گام دوم: پیدا کنین که چقدر برای رسیدن به همسایه‌های اون گره باید هزینه کنیم.



PARENT	NODE	COST
BOOK	LP	5
BOOK	POSTER	0
POSTER	GUITAR	30
POSTER	DRUMS	35
—	PIANO	∞

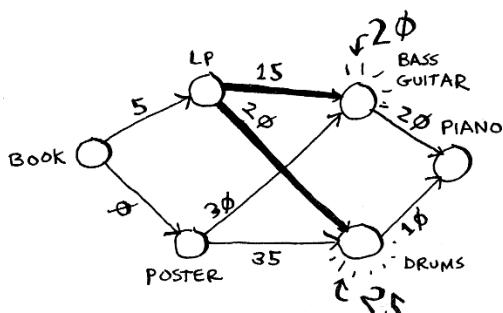
شما قیمت گیتار باس و مجموعه درام رو دارین. قیمت اونها، اگر از راه پوستر میخواین بهشون برسین از قبل مشخص شدن، پس در نتیجه پوستر به عنوان والد اونها قرار میگیره (در ستون والد). که این یعنی شما برای رسیدن به گیتار باس، یا پوستر رو دنبال کردین، و همین موضوع برای درامز هم صدق میکنه.

WE GO FROM "POSTER" TO GET TO THESE NODES {

PARENT	NODE	COST
BOOK	LP	5
BOOK	POSTER	0
POSTER	GUITAR	30
POSTER	DRUMS	35
—	PIANO	∞

تکرار قدم اول: ال پی، ارزون ترین گره بعدیه.

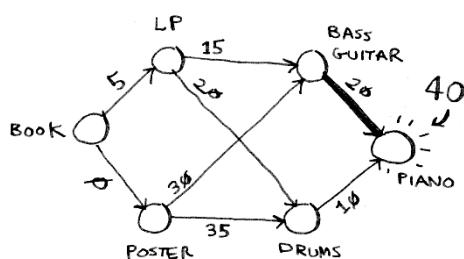
تکرار قدم دوم: تمام هزینه ها رسیدن به همسایه هاشون بروزرسانی کنین.



PARENT	NODE	COST
BOOK	LP	5
BOOK	POSTER	0
LP	GUITAR	20
LP	DRUMS	25
—	PIANO	∞

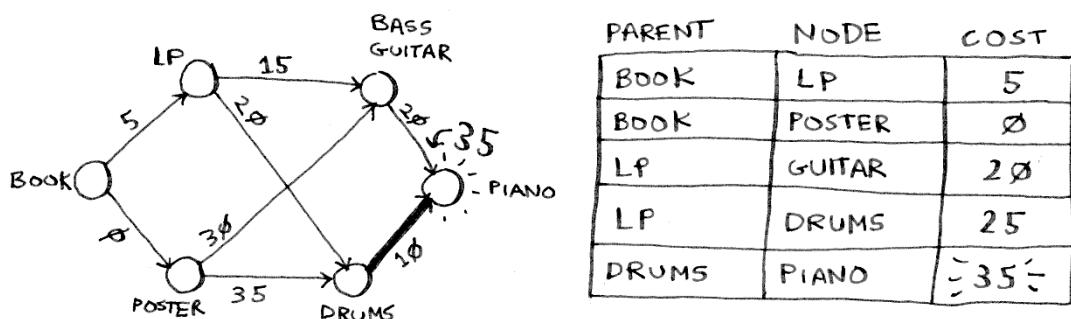
نگاه کنین، شما هزینه های گیتار و درام رو بروزرسانی کردین! که این یعنی رسیدن به گیتار و درامز، اگر از گره ال پی مسیر رو دنبال کنیم، ارزونتر درمیاد. پس درنتیجه، ال پی رو به عنوان والد گیتار باس و درامز قرار میدیم.

گیتار باس ارزون ترین آیتم بعدیه. هزینه رسیدن به همسایه هاشو بروزرسانی کنین.



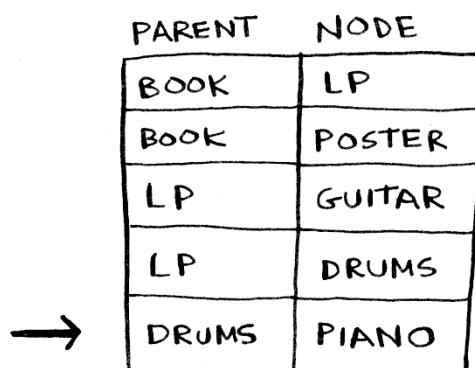
PARENT	NODE	COST
BOOK	LP	5
BOOK	POSTER	0
LP	GUITAR	20
LP	DRUMS	25
GUITAR	PIANO	40

اوکی، در نهایت شما با مبادله گیتار با پیانو هزینه ای که باید برای پیانو رو پرداخت کنین رو بدهست میارین. پس در نتیجه شما گیتار رو والد پیانو قرار میدین. در نهایت یک گره میمونه که بررسی نکردیم، مجموعه درام.



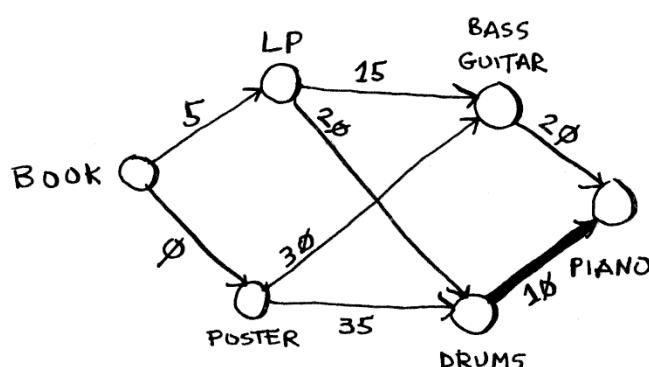
rama حتی میتونه پیانو رو با مبادله کردن مجموعه درام ارزونتر بدهست بیاره. پس با انجام ارزونترین مجموعه مبادلات راما ۳۵ دلار باید هزینه کنه.

حالا همونطور که قول داده بودم ، شما باید بتونین مسیر رو تشخیص بدین. تا الان، شما میدونین که کوتاه ترین راه ۳۵ دلار هزینه داره، اما چطور این مسیر رو پیدا میکنین؟ برای شروع ، به والد پیانو نگاه کنین.

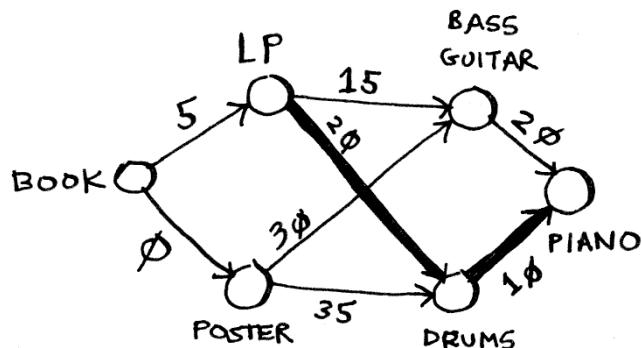


پیانو ، درامز رو به عنوان والد خودش داره. این یعنی راما ، درامز رو برای بدهست آوردن پیانو مبادله کرده. پس شما این یال رو دنبال میکنین.

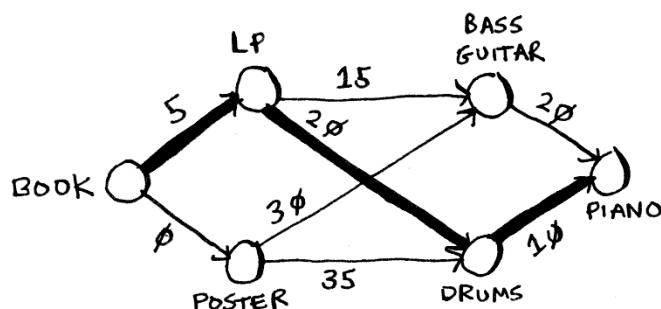
بیاین رو گراف ببینیم که چطور باید این کار رو بکنیم. پیانو ، درامز رو به عنوان والد خودش داره.



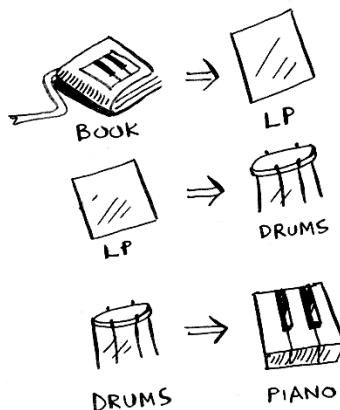
و درامز ، ال پی رو به عنوان والد خودش دارد.



پس راما ال پی رو برای بدست آوردن درامز مبادله میکنه. و البته راما کتابش رو برای بدست آوردن ال پی مبادله میکنه. حالا شما مسیر کامل رو با دنبال کردن والد ها به صورت عقب گرد ، دراختیار دارین.



این جا سری مبادلاتی رو داریم که راما باید انجام بده.

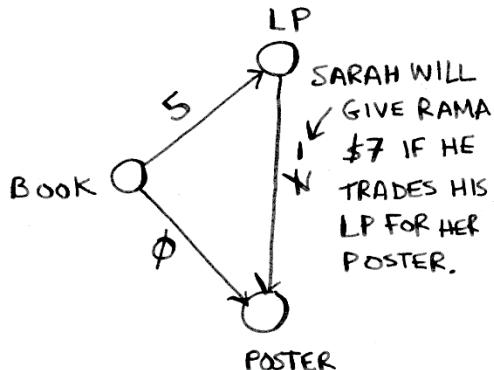


تا الان، به معنای واقعی کلمه خیلی از اصطلاح "کوتاه ترین مسیر" استفاده کردم: حساب کردن کوتاه ترین مسیر بین دو موقعیت مکانی یا بین دو فرد. امیدوارم این مثال بهتون نشون داده باشه که "کوتاه ترین مسیر" الزاماً درباره فاصله فیزیکی نیست. میتوانه درباره که حداقل رسوندن چیزی باشه. در این مثال راما میخواست هزینه های پرداختی رو به حداقل برسونه. با تشکر از دایجسترا.

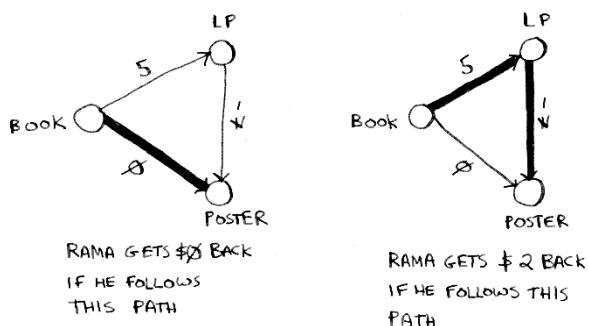
یال های دارای وزن منفی

در مثال مبادله راما، الکس به راما رو آیتم برای مبادله کردن رو پیشنهاد داد.

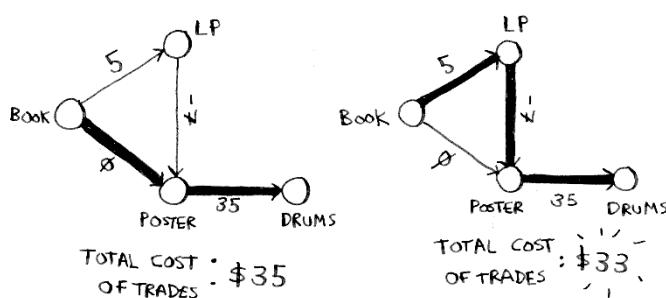
فرض کنین سارا فرض کنین سارا پیشنهاد مبادله ال پی رو در ازای پوستر به راما بده و در کنارش سارا ۷ دلار اضافه به راما بده. در واقع این کار برای راما هزینه نداره و علاوه بر اون ۷ دلار هم گیرش میاد. چطوری این رو روی گراف نشون میدین؟



یالی که از ال پی به سمت پوستر کشیده شده دارای وزن منفیه! راما از این مبادله ۷ دلار گیرش میاد. حالا راما دو راه داره برای اینکه به پوستر برسه.



پس منطقی که مبادله دوم رو انجام بدیم. اونطوری راما ۲ دلار بدست میاره! اگر یادتون باشه راما میتونست پوستر رو در ازای بدست آوردن درامز مبادله کنه. دو راه وجود داره که اون بتونه این کارو انجام بده.



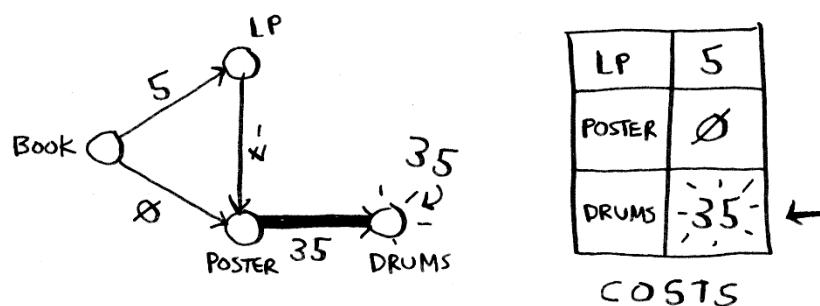
راه دوم ۲ دلار کمتر برای راما هزینه داره، پس اون باید راه دوم رو انتخاب کنه درسته؟ خب، حدس بزنین چیشد؟ اگر شما الگوریتم دایجسترا رو برای این گراف اجرا کنین، راما مسیر اشتباہ رو انتخاب میکنه. اون مسیر طولانی تر

رو انتخاب میکنه. پس ، شما نمیتوانیں ، وقتی گرافی با وزن منفی دارین ، الگوریتم دایجسترا رو روش اجرا کنین. یال های دارای وزن منفی الگوریتم رو خراب میکنن. بیاين بیینیم چه اتفاقی میفته وقتی شما الگوریتم دایجسترا رو روی این الگوریتم اجرا میکنین. اول جدول هزینه هارو درست کنین.

LP	5
POSTER	Ø
DRUMS	∞

COSTS

در قدم بعد، اون گره ای رو که کمترین هزینه برای رسیدن بهش رو داره رو پیدا کنین، و هزینه های رسیدن به همسایه اون گره رو در جدول هزینه ها بروزرسانی کنین. در مورد این گرافی که ما داریم ، پوستر گره کم هزینس. پس با توجه به الگوریتم دایجسترا، هیچ راه ارزومن تر از صفر دلاری برای رسیدن به پوستر وجود نداره (که شما میدونین این اشتباس). به هر حال، بیاين هزینه های رسیدن به همسایه هاشو بروزرسانی کنیم.



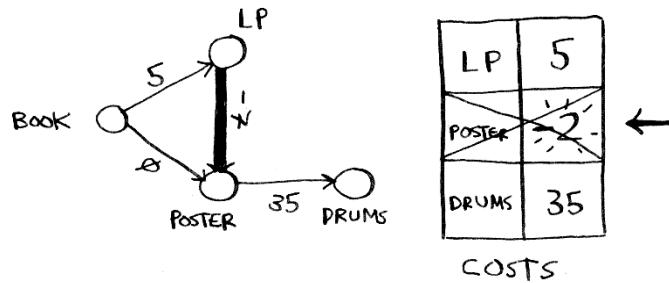
خیلی خوب، الان بدست آوردن درامز ۳۵ دلار هزینه برمیداره.

بیاين برعیم سراغ ارزون ترین گره بعدی که قبل ابررسیش نکردیم.

→ ←

LP	5
POSTER	Ø
DRUMS	35

هزینه هارو برای رسیدن به همسایه های اون گره، بروزرسانی کنین.



شما قبلاً گره مربوط به پوستر رو بررسی کردین، اما شما دارین هزینه رسیدن بهش رو بروزرسانی میکنین. این یک پرچم قرمز خیلی بزرگه. وقتی شما یک گرهی رو بررسی کردین معنیش این میشه که هیچ راه سریعتری و ارزونتری برای رسیدن بهش وجود نداره. اما با این اوصاف شما یک راه سریع تری برای رسیدن به پوستر پیدا کردین. خود درامز توی این الگوریتم هیچ همسایه ای نداره، پس در نتیجه این پایان الگوریتمه. این شما و این هزینه های نهایی:

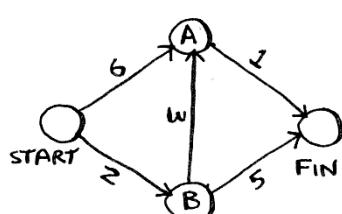
LP	5
POSTER	-2
DRUMS	35

FINAL COSTS

هزینه رسیدن به درامز ۳۵ دلار خرج برミداره. شما میدونین که یک راه وجود داره که فقط ۳۳ دلار هزینش میشه، اما الگوریتم دایجسترا او نو پیدا نکرد. الگوریتم دایجسترا فرض رو بر این گذاشت که شما گره پوستر رو قبل بررسی کردین و هیچ راه سریعتر و ارزونتری برای رسیدن بهش وجود نداره. این فرض تنها زمانی کار میکنه که شما هیچ یالی با وزن منفی در گراف نداشته باشین. پس در نتیجه نمیتوانیں برای گراف های دارای وزن منفی از الگوریتم دایجسترا استفاده کنین. اگر که شما میخواین کوتاه ترین مسیر رو در یک گراف که دارای یالهایی با وزن منفی هست رو پیدا کنین، یک الگوریتم برای اینکار هست. بهش میگن الگوریتم بلمن فورد (Bellman-Ford algorithm). بلمن فورد خارج بحث این کتابه، اما شما میتوانیں چیزای خیلی خوبی ازش توی اینترنت پیدا کنین.

پیاده سازی

بیاین ببینیم چطوری میتوانیم الگوریتم دایجسترا رو با کد پیاده کنیم. این گرافیه که من برای مثال ازش استفاده کردم.



برای نوشتن کد این مثال، شما به سه جدول هش نیاز دارین.

The three tables are:

START	A	6
	B	2
A	FIN	1
B	A	3
FIN	FIN	5
FIN	—	—

A	6
B	2
FIN	∞

A	START
B	START
FIN	—

شما جداول هزینه و والد رو همینطور که الگوریتم داره پیش میره و اجرا میشه، بروزرسانیش میکنین. اول، نیاز دارین که گراف رو پیاده کنین. برای اینکار شما از یک جدول هش، مثل جدول هشی که در فصل ۶ درست کردین، درست میکنین.

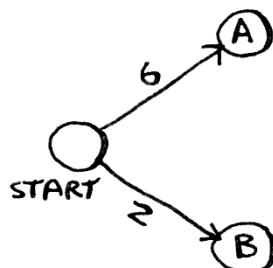
```
graph = {}
```

و در فصل قبل شما همسایه های یک گره رو در گراف در یک جدول هش مثل این ذخیره کردین:

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

اما اینبار نیاز دارین که هم همسایه ها و هم هیزینه‌ی رسیدن به همسایه هارو درش ذخیره کنین. به عنوان مثال،

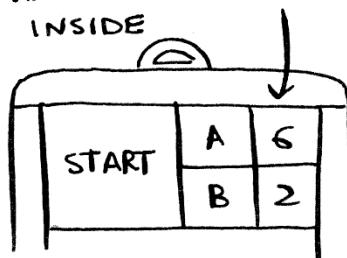
نقطه شروع، دو همسایه داره، A و B.



خب چطور میخواین وزن گراف هارو در کد نشون بدین؟ چرا از یک جدول هش دیگه استفاده نکنیم؟

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

THIS HASH TABLE
HAS MORE HASH TABLES
INSIDE



پس [“start”] هم خودش یک جدول هشه. شما میتوانید تمام همسایه های نقطه شروع را اینطوری دریافت کنید (بگیرینشون، گت کنید):

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

یک یال از نقطه شروع به A کشیده شده و یک یال هم به B. حالا اگر بخوایم بفهمیم وزن این یالها چقدر باشد چیکار کنیم؟

```
>>> print graph[“start”][“a”]
2
>>> print graph[“start”][“b”]
6
```

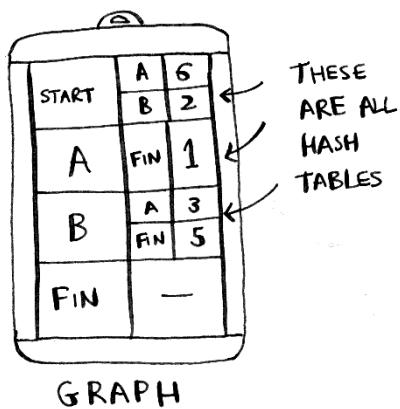
بیاین بقیه گره ها و همسایه هاشونو به graph اضافه کنیم:

```
graph[“a”] = {}
graph[“a”][“fin”] = 1

graph[“b”] = {}
graph[“b”][“a”] = 3
graph[“b”][“fin”] = 5
```

```
graph[“fin”] = {} <----- The finish node doesn’t have any neighbors.
```

جدول هش کامل شده graph اینشکلی میشه:



در قدم بعد، شما یک جدول هش دیگه برای ذخیره کردن هزینه در اون، نیاز دارین.

هزینه ای که برای یک گره در این جدول ذخیره میشه، نشون میده که چقدر رسیدن به اون گره از نقطه شروع هزینه بر میداره. شما میدونین که ۲ دقیقه طول میکشه تا از نقطه شروع به گره B برسیم. و اینم میدونین که ۶ دقیقه طول میکشه تا از نقطه شروع به A برسیم (همچنین ممکنه شما راهی رو پیدا کنین که زمان کمتری ببره). شما فعلاً نمدونین که چقدر طول میکشه تا به نقطه پایان برسیم. اگر شما هنوز هزینه رسیدن به یک گره رو

نمیدنیم، مقدار هزینش رو برابر بی نهایت قرار میدین. آیا میشه بینهایت رو در پایتون نشون داد؟ اینطور که معلومه

میشه:

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

اینم کدیه که برای ساختن جدول هزینه بکار میگیریم:

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

شما همچنین یک جدول هش دیگه برای والد ها نیاز دارین:

A	START
B	START
FIN	-

PARENTS

اینم کدیه که برای ساختن جدول والد به کار میگیریم:

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

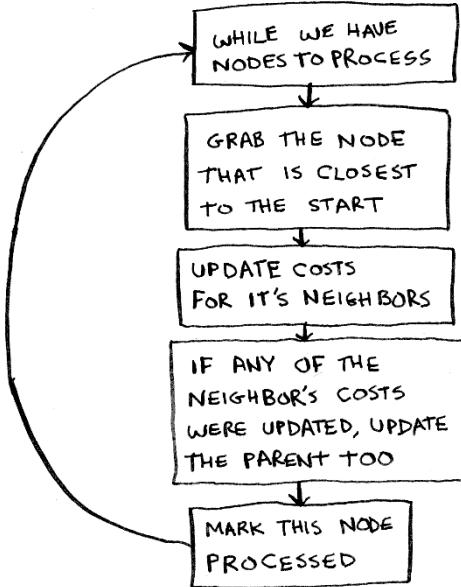
و در نهایت شما به یک آرایه نیاز دارین تا حواسش به گره هایی که بررسی کردین باشه و اوナ رو تو خودش نگه داره،

چرا که نیاز ندارین که یک گره رو بیشتر از یکبار بررسی کنیں:

```
processed = []
```

تمام تنظیمات اولیه و چیزی هایی که برای اجرای این الگوریتم نیاز داشیتم همینا بودن. حالا یک نگاه به این

الگوریتم بندازین.



اول کد رو بهتون نشون میدم و بعد میریم داخلش و خط به خط بررسیش میکنیم. اینم کد:

```

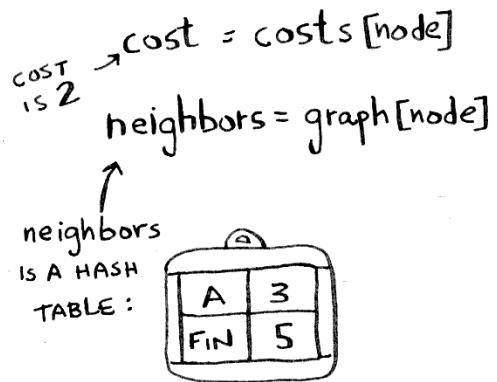
node = find_lowest_cost_node(costs) ← Find the lowest-cost node
                                         that you haven't processed yet.
while node is not None: ← If you've processed all the nodes, this while loop is done.
    cost = costs[node]
    neighbors = graph[node]
    for n in neighbors.keys(): ← Go through all the neighbors of this node.
        new_cost = cost + neighbors[n]      If it's cheaper to get to this neighbor
        if costs[n] > new_cost:← by going through this node ...
            costs[n] = new_cost ← ... update the cost for this node.
            parents[n] = node ← This node becomes the new parent for this neighbor.
    processed.append(node) ← Mark the node as processed.
    node = find_lowest_cost_node(costs) ← Find the next node to process, and loop.
  
```

این الگوریتم دایجسترا در پایتونه! من کد تابع `find_lowest_cost_node` رو بعدا بهتون نشون میدم اما اول بیاين بررسی کنیم ببینیم این کد در عمل چی کار میکنه.

این تابع میره و کم هزینه ترین گره رو پیدا میکنه.



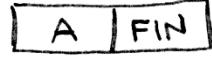
حالا بعدش که پیداشه کرد میره هزینه رسیدن به اون گره و همسایه های اون گره رو برامون میاره.



GRAPH

START	A	6
	B	2
A	FIN	1
B	A	3
	FIN	5
FIN		-

روی همسایه های اون گره حلقه بزن.

$\text{for } n \text{ in neighbors, keys():}$
 n IS "A"
 A LIST OF NODES: 

KEYS VALUES

A	3
FIN	5

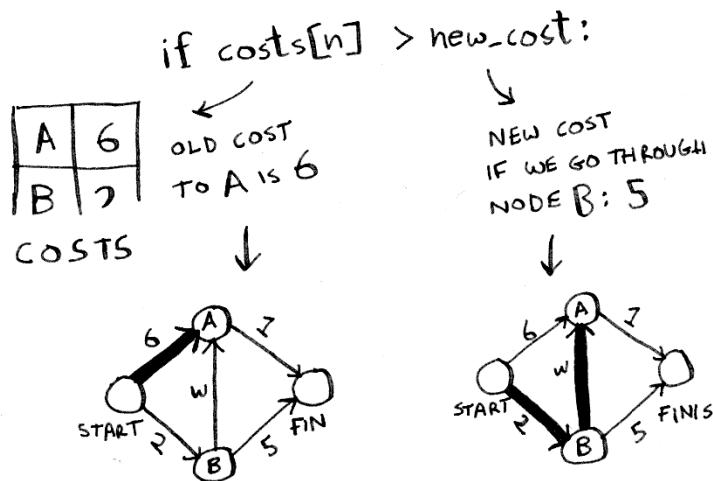
هر گره هزینه ای داره ، و اون هزینه ، نشون میده که چقدر طول میکشه تا از نقطه شروع به اون گره برسیم. اینجا شما دارین محاسبه میکنین که اگر از مسیر گره A < Start < گره B < گره A چقدر طول میکشید تا به گره A برسین.

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

↑ COST OF "B", i.e. 2 ↓ DISTANCE FROM B TO A: 3

}
}
new-cost = 2 + 3
= 5

بیاین مقایسه کنیم.



با این کار شما یک راه کوتاه تر به گره A پیدا کردیدن. هزینه هارو بروزرسانی کنین.

A	5
B	2
FIN	∞

COSTS

$\text{costs}[n] = \text{new-cost}$

↑ ↑
"A" 5

مسیر جدید از گره B عبور میکنه، پس گره B رو به عنوان والد جدید قرار بدین.

A	B
B	START
FIN	-

PARENTS

$\text{parents}[n] = \text{node}$

↑ ↑
"A" "B"

اوکی، دوباره برگشتیم به سر حلقه for و بررسی نقطه پایان.

for n in neighbors.keys():

↑ {
n is }
"FIN"

A	FIN
---	-----

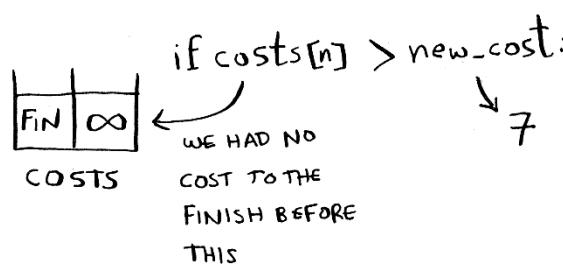
اگر از گره B عبور کنیم، چقدر طول میکشه تا به نقطه پایان برسیم؟

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

↓ ↓
2 DISTANCE FROM
 B TO THE FINISH:
 5

$\left. \right\} = 7$

۷ دقیقه طول میکشه. هزینه قبلی که در جدول بود بینهایت بود، و ۷ دقیقه کمتر از بینهایته پس برای آپدیت کردنش.



حالا یک مقدار جدید برای هزینه و والد نقطه پایانی قرار بدین.

$\text{costs}[n] = \text{new-cost}$

"FIN" ↑ ↑ "7"

A	5
B	2
FIN	7

COSTS ←

$\text{parents}[n] = \text{node}$

"FIN" ↑ ↑ "B"

A	B
B	START
FIN	7

PARENTS ←

اونکی، الان شما هزینه های رسیدن به تمام همسایه های گره B رو بروزرسانی کردین. پس این گره رو به عنوان بررسی شده، علامت بزنین.

`processed.append(node)`

"B" ↑

PROCESSED

NODES:

B

گره بعدی رو جهت بررسی کردن پیدا میکنیم.

CHEAPEST UNPROCESSED NODE →							
node = find_lowest_cost_node(costs)	→						
"A" ↑	ALREADY PROCESSED →						
	<table border="1"> <tr> <td>A</td><td>5</td></tr> <tr> <td>B</td><td>2</td></tr> <tr> <td>FIN</td><td>7</td></tr> </table>	A	5	B	2	FIN	7
A	5						
B	2						
FIN	7						

COSTS ←

هزینه رسیدن به گره A و همسایه هاش رو دریافت میکنیم.

$\text{cost} = \text{costs}[node]$

5 ↑

$\text{neighbors} = \text{graph}[node]$

↑
FIN 1

گره A تنها یک همسایه دارد که اونم نقطه پایانه.

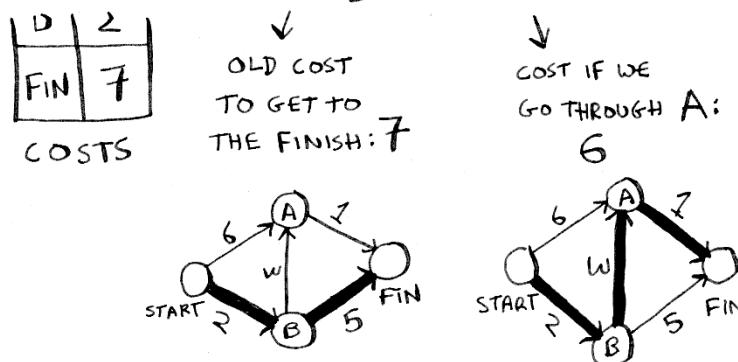
for n in neighbors.keys():
 ↑
 "FIN"
 }
 FIN

در حال حاضر، فعلاً ۷ دقیقه طول میکشد تا به نقطه پایان برسیم. حالا چقدر طول میکشد تا از طریق گره A به نقطه

پایان برسیم؟

$$\left. \begin{array}{l} \text{new_cost} = \text{cost} + \text{neighbors}[n] \\ \quad \downarrow \\ \text{COST TO} \\ \text{GET TO A} \\ \text{FROM THE} \\ \text{START: 5} \end{array} \quad \begin{array}{l} \downarrow \\ \text{DISTANCE FROM} \\ \text{A TO THE FINISH:} \\ 1 \end{array} \right\} = 6 \quad 5 + 1$$

if costs[n] > new_cost:



رفتن به نقطه پایان از طریق گره A سریعتر اتفاق میفته. بیاین هزینه ها و والد هارو آپدیت کنیم.

$$\text{costs}[n] = \text{new_cost}$$

↑
 "FIN"
 ←

A	5
B	2
FIN	6
COSTS	

$$\text{parents}[n] = \text{node}$$

↑
 "FIN"
 ↑
 "A"
 ←

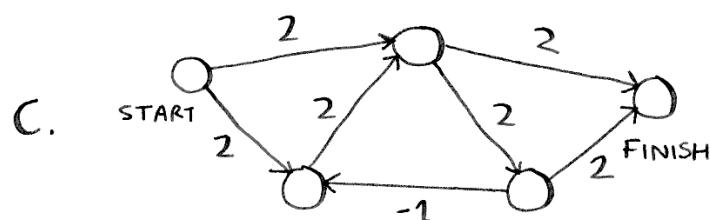
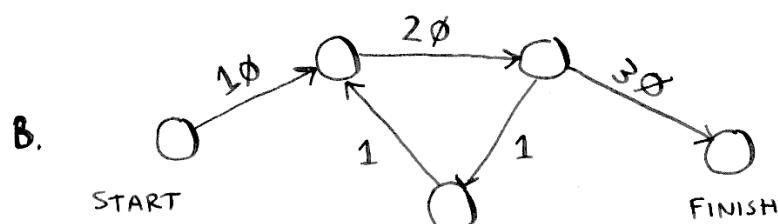
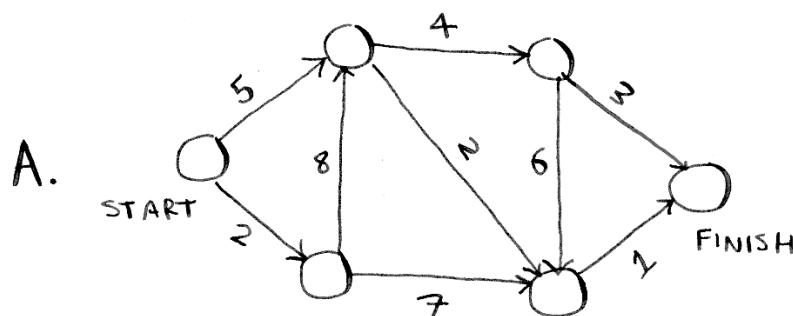
A	B
B	START
FIN	A
PARENTS	

وقتی که شما تمام گره ها را بررسی کردین ، الگوریتم به اتمام میرسه. امیدوارم اینکه او میدیم خط به خط کد رو بررسی کردیم، تونسته باشه یکم بهتر توی فهم این الگوریتم به شما کمک کنه. پیدا کردن ارزون ترین گره یا به عبارتی گره ای که هزینه رسیدن بهش کمتر باشه با کمک تابع `find_lowest_cost_node` تقریبا آسونه. این کدشه:

```
def find_lowest_cost_node(costs):
    lowest_cost = float("inf")
    lowest_cost_node = None
    for node in costs: <----- Go through each node.
        cost = costs[node]
        if cost < lowest_cost and node not in processed: <----- If it's the lowest cost
            so far and hasn't been
                lowest_cost = cost <----- ... set it as the new lowest-cost node.
                lowest_cost_node = node
    return lowest_cost_node
```

تمرین

۷.۱ در هر کدام از این گراف ها بررسی کنیں که وزن کوتاه ترین مسیر از نقطه شروع تا پایان چقدر؟

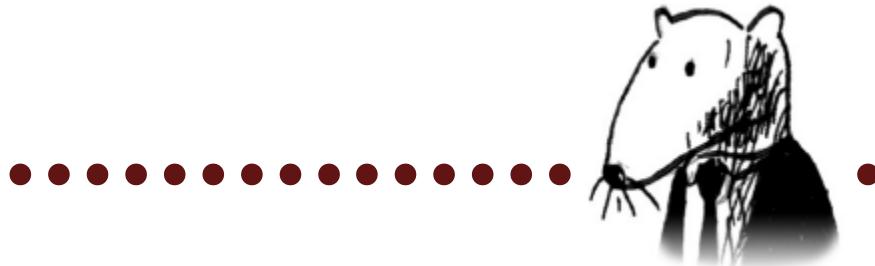


خلاصه

- جستوجوی اول عمق برای زمانی استفاده میشے که ما میخوایم کوتاه ترین مسیر رو در یک گراف غیر وزن دار محاسبه کنیم.
- الگوریتم دایجسترا زمانی استفاده میشے که ما میخوایم کوتاه ترین مسیر رو در یک گراف وزن دار محاسبه کنیم.
- الگوریتم دایجسترا زمانی استفاده میشے که تمام وزن های گراف مثبت باشند.(مقدار منفی نداشته باشیم).
- اگر گرافی با وزن منفی دارین ، از الگوریتم بلمن فورد استفاده کنید.



الگوریتم های حریص (Greedy algorithms)

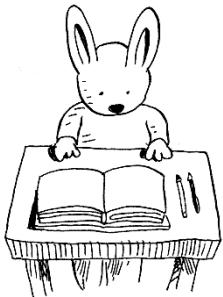


در این فصل:

- شما یادمیگرین چطور با مسائل غیر ممکن مقابله کنین: مسئله هایی که هیچ راه حل الگوریتمی سریعی برashون وجود نداره(مسئله ان-پی کامل)(**NP-complete**)
- شما یاد میگرین که چطور این مسائل رو شناسایی کنین تا وقتتون رو برای پیدا کردن یک الگوریتم سریع برای آنها هدر ندید.
- شما درباره الگوریتم های تقریبی یادمیگرین، که میتوانین ازش برای پیدا کردن سریع یک راه حل و جواب تقریبی برای یک مسئله ان-پی کامل ازش استفاده کنین.
- شما درباره استراتژی حریص یادمیگرین. یک استراتژی حل مسئله ساده.



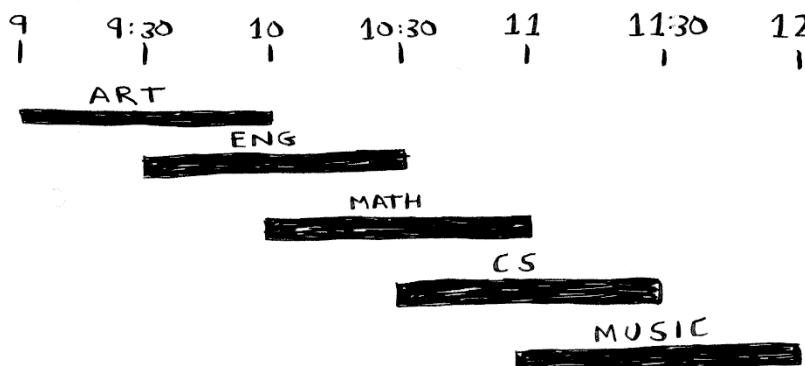
مسئله زمان بندی کلاس ها



فرض کنین شما یک کلاس درس دارین و میخواین تا جایی که امکانش هست کلاسای مختلف رو شرکت کنین. شما لیستی از کلاس ها در اختیار دارین.

CLASS	START	END
ART	9 AM	10 AM
ENG	9:30 AM	10:30 AM
MATH	10 AM	11 AM
CS	10:30 AM	11:30 AM
MUSIC	11 AM	12 PM

شما نمیتونین تمام اون کلاسا رو باهم بردارین چراکه تایم کلاس ها باهم تداخل دارن.



شما میخواین تا جایی که امکان داره و میتوانین کلاس بردارین، این کارو انجام بدین. چطوری مجموعه ای از کلاس ها رو بر میدارین به طوری که بتونین بزرگترین مجموعه رو در اختیار داشته باشین؟

بنظر مشکل سختی میاد درسته؟ درواقع الگوریتمش خیلی راحته. شاید سوپرایزتون کنه. الگوریتم اینطوری کار میکنه:

۱. کلاسی بردارین که زودتر تموم میشه. این اولین کلاسی میشه که شما بر میدارینش.
۲. حالا شما باید یک کلاسی رو انتخاب کنین که بعد از این کلاس شروع میشه. دوباره کلاسی رو بردارین که زود تر تموم میشه. این دومین کلاسی میشه که شما بر میدارینش.

همینطوری اینکار رو ادامه بدین تا در نهایت به جواب برسین. بیاین امتحانش کنیم. درس هنر زود تر از همه تموم میشه، ساعت ۱۰ صبح. پس در نتیجه این کلاس، یکی از کلاسایی که شما انتخابش میکنین.

ART	9 AM	10 AM	✓
ENG	9:30 AM	10:30 AM	
MATH	10 AM	11 AM	
CS	10:30 AM	11:30 AM	
MUSIC	11 AM	12 PM	

حالا شما به کلاس بعدی نیاز دارین، کلاسی که ساعت ۱۰ شروع میشه و زود تر از همه تموم میشه.

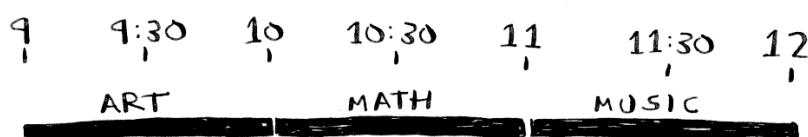
ART	9 AM	10 AM	✓
ENG	9:30 AM	10:30 AM	X
MATH	10 AM	11 AM	✓
CS	10:30 AM	11:30 AM	
MUSIC	11 AM	12 PM	

انگلیسی رو از لیست انتخابامون خارج میکنیم چون با هنر تداخل داره.

در آخر هم، علوم کامپیوتر با ریاضی تداخل داره اما موسیقی مشکلی نداره.

ART	9 AM	10 AM	✓
ENG	9:30 AM	10:30 AM	X
MATH	10 AM	11 AM	✓
CS	10:30 AM	11:30 AM	X
MUSIC	11 AM	12 PM	✓

پس درنتیجه این ها کلاس هایی هستند که شما برشون میدارین.



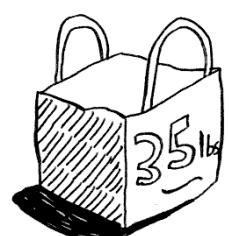
خیلیا بهم میگن این الگوریتم خیلی ساده به نظر میاد. این خیلی واضحه پس حتماً اشتباهه. اما این زیبایی الگوریتم های حریصه: اونا سادن. یک الگوریتم حریص خیلی سادس: در هر گام ، یک حرک بھینه انتخاب کن. در این موردی که بررسی کردیم ، هر بار شما باید یک کلاس انتخاب کنین، پس شما کلاسی رو انتخاب میکنین که زود تر از بقیه تموم میشه. از نظر فنی بخوایم بگیم: در هر گام شما یک راه حل بھینه محلی (*locally optimal*) را انتخاب میکنین و در نهایت شما میمونین و راه حل های بھینه کلی (*globally optimal solution*) را انتخاب میکنین یا نه، این الگوریتم ساده یک راه حل بھینه برای مسئله زمان بندی کلاس ها پیدا میکنه. باور کنین یا نه، به طوری واضحی معلومه که الگوریتم های حریص ، همیشه جواب نمیدن. اما نوشتنشون خیلی سادس. بیاين به یک مثال دیگه نگاه کنیم.

مسئله کوله پشتی



فرض کنین یک دزد حریص هستین. شما با یک کوله پشتی در یک فروشگاه هستید. و خیلی آیتم ها هست که شما میتونین بذدین اما شما فقط میتوانین آیتم هایی رو بردارین که در کوله پشتیتون جا بشه. کوله پشتی میتونه تا ۳۵ پوند رو تحمل کنه.(هر پوند تقریباً برابر نیم کیلو گرمه).

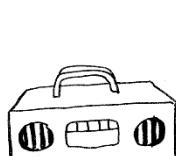
شما میخواین جوری آیتم هارو بردارین که بیشترین سود رو برآتون داشته باشه. از چه الگوریتمی استفاده میکنین؟



دوباره، استراتژی حریص تقریباً سادس:

۱. اون آیتمی که از همه گرونتره و توی کول پشتیت جا میشه رو بردار.
۲. آیتم گرون بعدی رو که داخل کوله پشتیت جا میشه رو بردار و الی آخر.

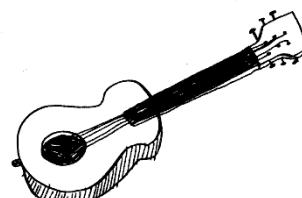
اما این بار، این کار جواب نمیده! برای مثال فرض کنین ۳ آیتم وجود داره که شما میتوانین بذدین.



STEREO
\$ 3000
30lbs



LAPTOP
\$ 2000
20lbs



GUITAR
\$ 1500
15lbs

کوله پشتی شما میونه ۳۵ پوند از آیتم هارو تو خودش جا بده. سیستم استریو از همه گرون تره، پس میتونین اونو بردارین. حالا دیگه فضایی برای آیتم دیگه ای ندارین.

شما از کالاهایی که برداشتین ۳۰۰۰ دلار سود کردین. اما صبر کنین، اگر شما لپ تاپ و گیتار رو به جاش بردارین، میتوانین ۳۵۰۰ دلار سود کنین.

به طور واضحی استراتژی حریص در این مورد به شما راه حل بهینه رو نمیده. اما تقریباً به راه حل بهینه نزدیکتون میکنه. در فصل بعدی، برآتون توضیح میدم که چطور راه حل درست رو محاسبه کنین. اما اگر یک دزد در مرکز خرید باشید اصلاً به عالی بودن و راه حل بهینه اهمیت نمیدید و به همون راه حل تقریبی بسنده میکنین.

این هم چیزیه که میتوانیم از این مثال دوم برداشت کنیم: بعضی وقتاً عالی بودن دشمن خوب بودنه. بعضی وقت ها هم شما به یک الگوریتم نیاز دارین که بتونه مسئله به صورت تقریب خوبی حل کنه. و اون جاست که الگوریتم حریص میدرخشه، به این دلیل که نوشتنشون راحته و غالباً به جواب نزدیکن.

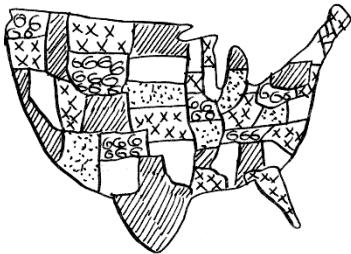
تمرین ها

۸.۱ شما در یک شرکت لوازم خانگی کار میکنین و باید این لوازم رو به سراسر کشور ارسال کنید. شما باید کامیون های بارکشیتون رو با جعبه های مختلف پر کنید. تمام جعبه سایزشون باهم فرق میکنه و شما میخواین حداکثر استفاده رو از فضا کامیون ها ببرین. چطوری جعبه هارو برای استفاده حداکثری از فضای کامیون ها انتخاب میکنین؟ با استراتژی حریص این مسئله رو حل کنین. آیا این استراتژی به راه حل بهینه رو بهتون ارائه میده؟

۸.۲ شما دارین به اروپا سفر میکنین. و ۷ روز فرصت دارین تا از هرجایی و چیزی که میخواین بازدید کنین. یک مقدار به هر جایی که میخواین ازش بازدید کنین نسبت میدین (این مقدار نشون میده که شما بر حسب چقدر میخواین اونجارو بگردین) و زمانی رو که این کار شما طول میکشه رو تخمین بزنین. چطور میتوانین در زمان اسکانتون، حداکثر استفاده رو برای دیدن جاهایی که دوست دارین ببرین؟ از الگوریتم حریص استفاده کنین. آیا اون الگوریتم بهتون بهینه ترین راه رو ارائه میده؟

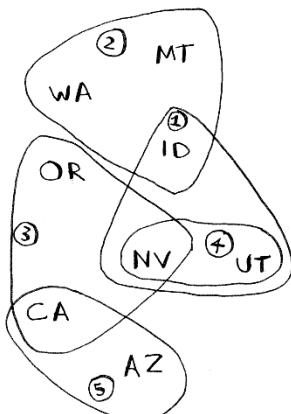
بیاین به آخرین مثال نگاهی بندازیم. این مثالیه که استفاده از الگوریتم حریص برای رسیدن به جواب، کاملاً ضروریه.

مسئله پوشش مجموعه



فرض کنین میخواین یک نمایش رادیویی پخش کنین. شما میخواین تمام شنونده ها در ۵۰ ایالت کشور به این نمایش دسترسی داشته باشند. باید انتخاب کنین که این اجرا روی کدام ایستگاه های رادیویی پخش شوند تا همه شنونده ها اون رو بشنوند. پخش برنامه روی هر ایستگاه هزینه میبره، پس شما باید تلاش کنین که تعداد ایستگاه هایی رو که میخواین این نمایش پخش بشه رو به حداقل برسونین. شما لیستی از ایتسگاه در اختیار دارین.

هر ایستگاه یک منطقه رو پوشش میده، و بعضی از این ایستگاه ها باهم هم پوشانی دارن.



چطوری میخواین کوچیک ترین مجموعه از ایستگاه ها رو برای پخش این اجرا پیدا کنین که تمام ۵۰ ایالت رو پوشش بده؟ ساده بنظر میاد مگه نه؟ اما وقتی واردش بشیم معلوم میشه که خیلی سخته. اینم نحوه انجام این کار:

RADIO STATION	AVAILABLE IN
KONE	ID,NV,UT
KTWO	WA, ID, MT
KTHREE	OR, NV, CA
KFOUR	NV, UT
KFIVE	CA, AZ

...etc...

۱. تمام زیر مجموعه ایستگاه های ممکن رو لیست کنین. به این مجموعه، مجموعه توان گفته میشه. 2^n زیرمجموعه ممکن برای هر مجموعه وجود داره.



۲. در بین این زیر مجموعه ها یکی از مجموعه ها با کمترین تعداد عضو رو انتخاب کنین که تمام ۵۰ ایالت رو پوشش میده.

مشکل اینجاست که ، حساب کردن تمام زیر مجموعه ایستگاه های ممکن ، خیلی زمان میبره. به اندازه $O(2^n)$ زمان میبره، چرا که 2^n ایستگاه وجود داره. انجام این کار اگر مجموعه ای شامل ۵ تا ۱۰ عضو داشته باشین ممکنه. اما با مثالی که اینجا زدیم ، درباره این فکر کنین که اگر آیتم های زیادی داشته باشیم چی میشه؟ اگر ایستگاه های زیادی داشته باشیم خیلی طول میکشه تا به جواب برسیم. فرض کنین شما میتوانین ۱۰ تا زیر مجموعه در یک ثانیه حساب کنین.

هیچ الگوریتمی وجود نداره که بتونه این کارو به اندازه کافی سریع انجام بدنه! چی کار میتونین بکنین؟

NUMBER OF STATIONS	TIME TAKEN
5	3.2 sec
16	102.4 sec
32	13.6 years
100	4×10^{21} years

الگوریتم ها تقریبی

الگوریتم های حریص اینجان تا نجاتمون بدن! این جا یک الگوریتم حریص داریم که خیلی مارو به جواب نزدیک میکنه:

۱. ایستگاهی رو انتخاب کنین که بیشترین ایالاتی رو پوشش میده که تا حالا پوششون ندادیم. مشکلی نداره اگر ایستگاهی رو انتخاب کردیم که یک ایالت رو که از قبل پوشش داده شده، پوشش بده.
۲. این کارو تا زمانی که تمام ایالت ها پوشش داده بشن، تکرار کنین.

به این الگوریتم ، الگوریتم تقریبی گفته میشه. وقتی که محاسبه دقیق یک راه حل خیلی طول میکشه ، اونجاست که الگوریتم تقریبی جواب میده. الگوریتم های تقریبی براساس این دو مورد سنجیده میشن:

- اینکه چقدر سریع هستند
- اینکه چقدر به راه حل بهینه نزدیک هستند

الگوریتم های حریص انتخاب خوبی هستند که چون نه تنها کار کردن باهاشون راحته بلکه همین راحتی باعث میشه که این الگوریتم ها سریع هم اجرا بشن. در این مثالی که بررسی کردیم ، الگوریتم حریص در زمان $O(n^2)$ اجرا میشن ، که n نشان دهنده تعداد ایستگاه های رادیویی هستش.

بیان بینیم این مسئله توی کد چطوریه.

کدایی که برای راه اندازی این الگوریتم نیاز داریم

برای این مثال، و برای اینکه همه چیز رو ساده ببریم جلو من از مجموعه ای ایالت ها و ایستگاه ها استفاده میکنم.

اول از همه، یک لیستی از ایالت هایی که قصد پوششون رو دارین رو درست کنین:

```
states_needed = set(["mt", "wa", "or", "id", "nv", "ut",
"ca", "az"]) ----- You pass an array in, and it gets converted to a set.
```

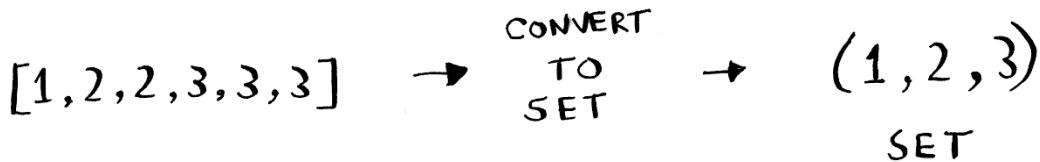
من از **set** ها برای این کار استفاده کرم. **Set** ها مثل لیست هستن اما هر آیتم فقط یکبار میتونه داخل یک **set** قرار بگیره و اگر بیشتر ۱ بار تکرار بشه فقط ۱ بار درنظر گرفته میشه. **set** ها نمیتونن آیتم های تکراری داشته باشن. برای مثال. فرض کنین شما این لیست رو دارین:

```
>>> arr = [1, 2, 2, 3, 3, 3]
```

و این رو به یک ست (معنیش هم میشه همون مجموعه) تبدیلش کردین:

```
>>> set(arr)
set([1, 2, 3])
```

اعداد ۱، ۲ و ۳ فقط یکبار در این ست (**set**) نمایش داده میشن.



همچنین شما به یک لیست از ایستگاه هایی نیاز دارین که قرار انتخابشون بکنین. من برای اینکار از یک هش استفاده میکنم:

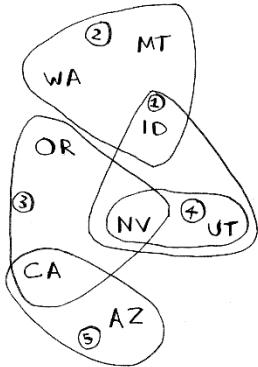
```
stations = {}
stations["kone"] = set(["id", "nv", "ut"])
stations["ktwo"] = set(["wa", "id", "mt"])
stations["kthree"] = set(["or", "nv", "ca"])
stations["kfour"] = set(["nv", "ut"])
stations["kfive"] = set(["ca", "az"])
```

کلید های این هش، اسم های ایستگاهها هستند، و مقادیر این اونها، ایالت هایی هستن که میتونن پوشش بدن. خب در این مثال ایستگاه **kone** آیدaho، نوادا و یوتا رو پوشش میده. تمام مقادیر هم به صورت ست هستند. کلا تبدیل هر چیزی به ست زندگیتون آسون میکنه (مترجم: توی این مثال نویسنده میخواسته به ضرورت استفاده از ست ها اشاره کنه). در ادامه خواهید دید که چرا.

در نهایت شما به یک چیزی نیاز دارین که اسامی تمام ایستگاه هایی که میخواین ازشون استفاده کنین رو درش نگه دارین:

```
final_stations = set()
```

محاسبه کردن جواب



حالا نیازه که شما محاسبه کنین که از چه ایستگاهایی میخواین استفاده کنین. یک نگاه به عکس رو به رو بندارین، و ببینین آیا میتوانیں پیش بینی کنین که بهتر از چه ایستگاه هایی استفاده کنیم.

میتوانه بیشتر از یک راه حل درست برای این مسئله وجود داشته باشه. شما باید تک تک ایستگاه ها رو بررسی کنین و اونی رو انتخاب کنین که بیشتر مناطق پوشش داده نشده رو پوشش بده. من اسم همچین ایستگاهی رو `best_station` میدارم:

```
best_station = None
states_covered = set()
for station, states_for_station in stations.items():
```

یک مجموعه از تمام ایالت هایی پوشش داده نشده ایه که اون ایستگاه پوشش شون میده. حلقه `for` این اجازه رو بہتون میده که تک تک ایستگاه هارو بررسی کنین و ببینین کدامیک از این ایستگاه ها بهترین ایستگاه ست. بیاین به بدنه حلقه `for` نگاهی بنداریم:

```
covered = states_needed & states_for_station
if len(covered) > len(states_covered):
    best_station = station
    states_covered = covered
```

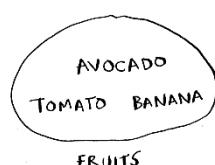
← New syntax! This is called a set intersection.

یک خط بازمه توی کد وجود داره:

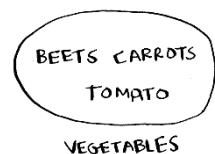
```
covered = states_needed & states_for_station
```

چی شد؟

ست ها (مجموعه ها)



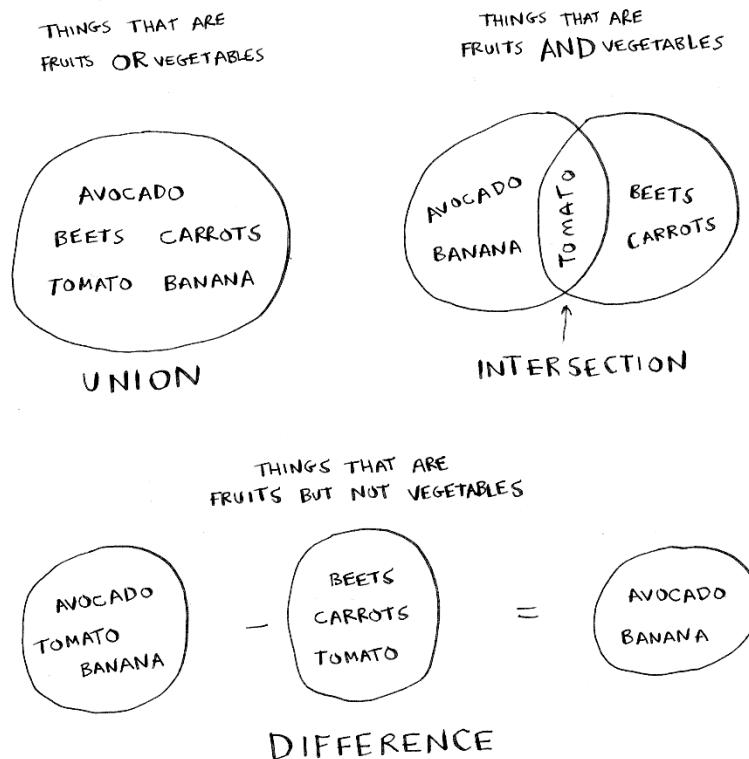
فرض کنی شما مجموعه ای از میوه ها در اختیار دارین.



همچنین مجموعه ای از سبزیجات هم در اختیار شماست.

وقتی که شما دو تا مجموعه داشته باشین میتوانیں یک سری کارای باحال باهاش انجام بدین.

اینها یک سری از کارایی که شما میتوانید با مجموعه انجام بدین.



- اجتماع دو مجموعه یعنی ترکیب اون دو مجموعه باهم دیگه. (جمع دو مجموعه باهم)
- اشتراک دو مجموعه یعنی پیدا کردن آیتم های مشترک در بین دو مجموعه
- تفاضل دو مجموعه یعنی حذف مقادیر مشترک مجموعه ها در اون مجموعه ای که قصد حذف مقادیر مشترکش رو داریم.

به عنوان مثال:

```
>>> fruits = set(["avocado", "tomato", "banana"])
>>> vegetables = set(["beets", "carrots", "tomato"])
>>> fruits | vegetables <----- This is a set union.
set(["avocado", "beets", "carrots", "tomato", "banana"])
>>> fruits & vegetables <----- This is a set intersection.
set(["tomato"])
>>> fruits - vegetables <----- This is a set difference.
set(["avocado", "banana"])
>>> vegetables - fruits <----- What do you think this will do?
```

برای خلاصه:

- ست ها (مجموعه ها) مثل لیست ها هستند با این تفاوت که نمیتوان مقادیر تکراری داشته باشند.
- میتوانید عملیات های جالبی روی ست ها بزنید مثل اجتماع، اشتراک و تفاضل.

برگردیم سراغ کد

بیان برگردیم سراغ مثال اصلی.

این تیکه کد همون اشتراک مجموعه هاست:

```
covered = states_needed & states_for_station
```

یک مجموعه ای از ایالت هایی که هم در `States_needed` وجود دارن هم در `covered`. پس درنتیجه `covered` مجموعه ای از ایالت های پوشش داده نشده که این ایالتی که داریم بررسیش میکنیم پوشششون میده. در قدم بعد چک میکنیم که آیا این ایستگاه، ایالات بیشتری را از `best_station` که در اختیار داریم پوشش میده یا نه:

```
if len(covered) > len(states_covered):
    best_station = station
    states_covered = covered
```

اگر آره، این ایستگاه در متغیر `best_station` به عنوان مقدار جدید، ریخته میشود. در نهایت، بعد از اینکه حلقه `for` به اتمام رسید. شما این `best_station` یی رو که در اختیار دارین به لیست ایستگاه های نهایی (انتخابی) اضافه میکنین:

```
final_stations.add(best_station)
```

هم چنین شما بعد از این کار باید `states_needed` رو بروزرسانی کنین. چرا که این ایستگاهی که به لیست اضافه کردیم، ایالاتی رو پوشش داده که دیگه نیاز نیست اون ایالات پوشش داده بشن:

```
states_needed -= states_covered
```

و تا زمانی که `states_needed` خالی نشده به حلقه زدن ادامه میدین. اینم کد کامل برای حلقه مورد نظر:

```
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)
```

درنهایت شما میتوانید `final_station` را چاپ کنید و باید یک همچین چیزی ببینید:

```
>>> print final_stations
set(['ktwo', 'kthree', 'kone', 'kfive'])
```

این همون چیزیه که میخواستید؟ به جای ایستگاه های ۱، ۲، ۳، ۴ و ۵ شما میتوانستیدن ایستگاه های ۲، ۳، ۴ و ۵ را انتخاب کنید. بیاین زمان اجرایی الگوریتم حریص با الگوریتمی که جواب رو به صورت دقیق محاسبه میکرد، مقایسه کنید.

NUMBER OF STATIONS	$O(n!)$ EXACT ALGORITHM	$O(n^2)$ GREEDY ALGORITHM
5	3.2 sec	2.5 sec
10	102.4 sec	10 sec
32	13.6 yrs	102.4 sec
100	4×10^{24} yrs	16.67 min

تمرین ها

مشخص کنید که کدام یک از این الگوریتم ها جز الگوریتم های حریص به حساب میان.

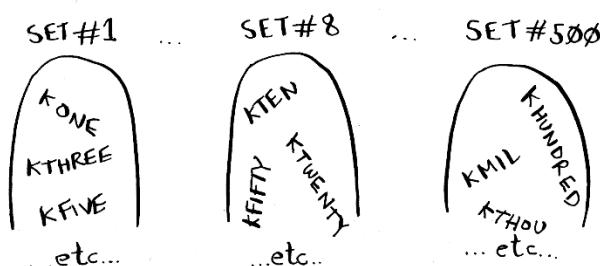
۸.۳ کوییک سورت یا مرتب سازی سریع

۸.۴ جستوجوی اول عمق

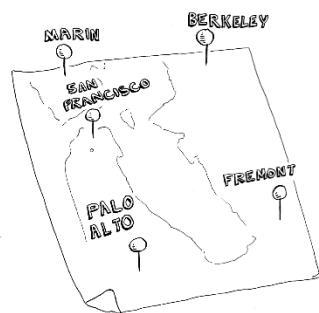
۸.۵ الگوریتم دایجسترا

مسائل ان پی کامل

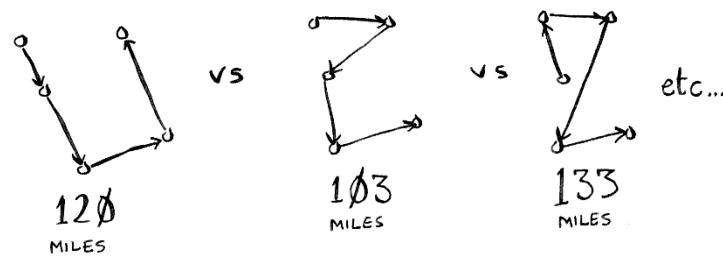
برای حل مسئله پوشش مجموعه ای، شما باید هر کدام از مجموعه های ممکن رو بررسی میکردیدن.



شاید مثال فروشنده دوره گرد در فصل ۱ یادتون افتاده باشه. در این مسئله، یک فروشنده دوره گرد باید از پنج شهر بازدید میکرد.



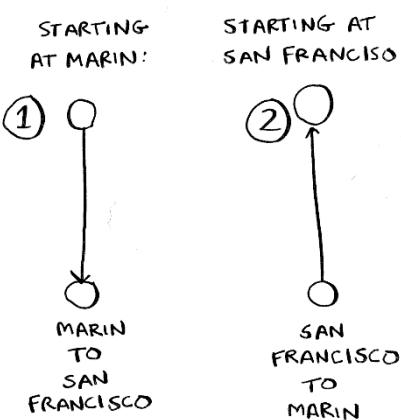
و اون تلاش میکنه تا کوتاه ترین مسیری رو که باعث میشه از تمام اون ۵ شهر بگذرد و بازدید بکنه، پیدا کنه. برای پیدا کردن کوتاه ترین راه شما اول باید تمام مسیر های ممکن رو بررسی و محاسبه کنین.



چند مسیر رو شما باید برای ۵ شهر بررسی کنین؟

فروشنده دوره گرد، قدم به قدم

بیاین با مقدار کم شروع کنیم. فرض کنین شما فقط ۲ شهر دارین. خب، دو راه وجود داره که میتوانیم انتخابشون بکنیم.



همون مسیر یا مسیری متفاوت؟

شاید شما فکر کنین که اینکه کلا یدونه راهه چرا میگیم دو راه؟ از همه اینا که بگذریم. مگه راه سانفرانسیکو به سمت مارین با راه مارین به سمت سانفرانسیسکو یکی نیست؟ جواب: نه لزوما. خیلی از شهرها مثل سانفرانسیسکو راههای یک طرفه زیادی دارند، پس شما نمتنین از همون راهی که او مدین، دوباره همونو برگردین. شما ممکنه حتی برای پیدا کردن مسیر برگشت ۱ یا ۲ مایل اضافه تر برای پیدا کردن یک مسیر به بزرگراه رو هم طی کنین. پس لزوماً این دو راه باهم یکی نیستند.

شاید شما پیش خودتون فکر کنین که، "در مسئله فروشنده دوره گرد، آیا شهر خاصی هست که ما باید از اون جا مسیرمون رو شروع کنیم؟" برای مثال، بیاین فرض کنیم که من فروشنده دوره گرد هستم. در سانفرانسیکو هم زندگی میکنم. و باید به ۴ شهر دیگه سفر کنم. پس درنتیجه سانفرانسیسکو شهری که من باید از اونجا شروع کنم. اما بعضی وقتا، شهر مبدا بی برای ما مشخص نشده. فرض کنین شما فدرال اکسپرس (شرکت حمل و نقل) هستین و میخواین بسته ای رو به "بی اریا" (Bay Area) منطقه در کالیفرنیا) ارسال کنین. این بسته از شیکاگو به یکی از ۵۰ انبار فدرال اکسپرس در بی اریا ارسال میشود. بعد از اون، بسته در کامیون های ارسال بسته ها که به نقاط مختلف ارسال میشه، قرار میگیره. این بسته به کدام انبار باید فرستاده بشه؟ در اینجا نقطه مبدا مشخص نیست. این به شما بستگی داره که مسیر بهینه و نقطه شروع رو محاسبه کنین.

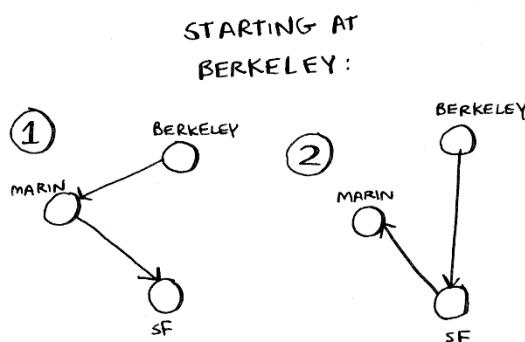
زمان اجرایی برای هردو حالت یکیه. اما اگر این مثال رو با مشخص نبودن محل مبدا جلو ببریم آسونتره، پس من با همین حالت جلو میرم.

دو شهر = دو مسیر ممکن.

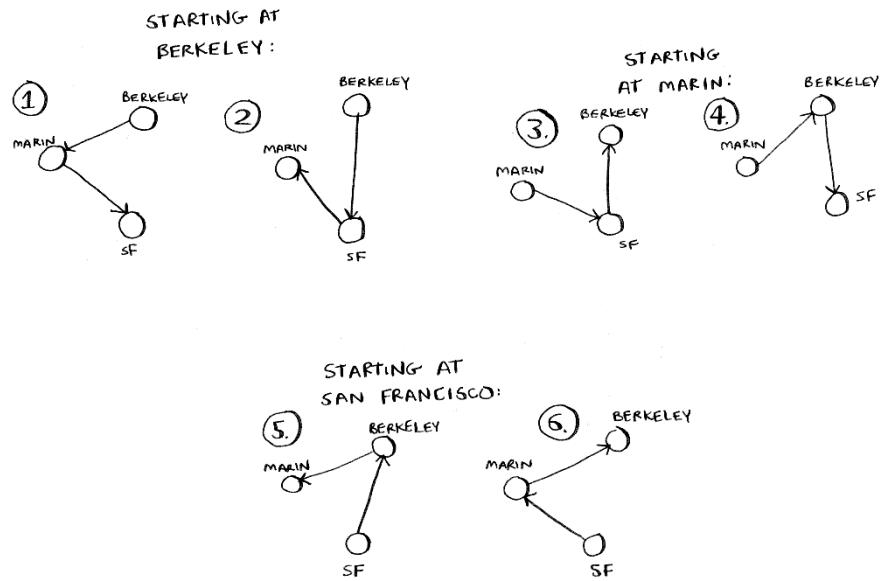
۳ شهر

حالا فرض کنین، یک شهر دیگه اضافه میکنیم. چند مسیر ممکن وجود داره؟

اگر شما از برکلی شروع کنین، ۲ شهر دیگه برای بازدید کردن دارین.



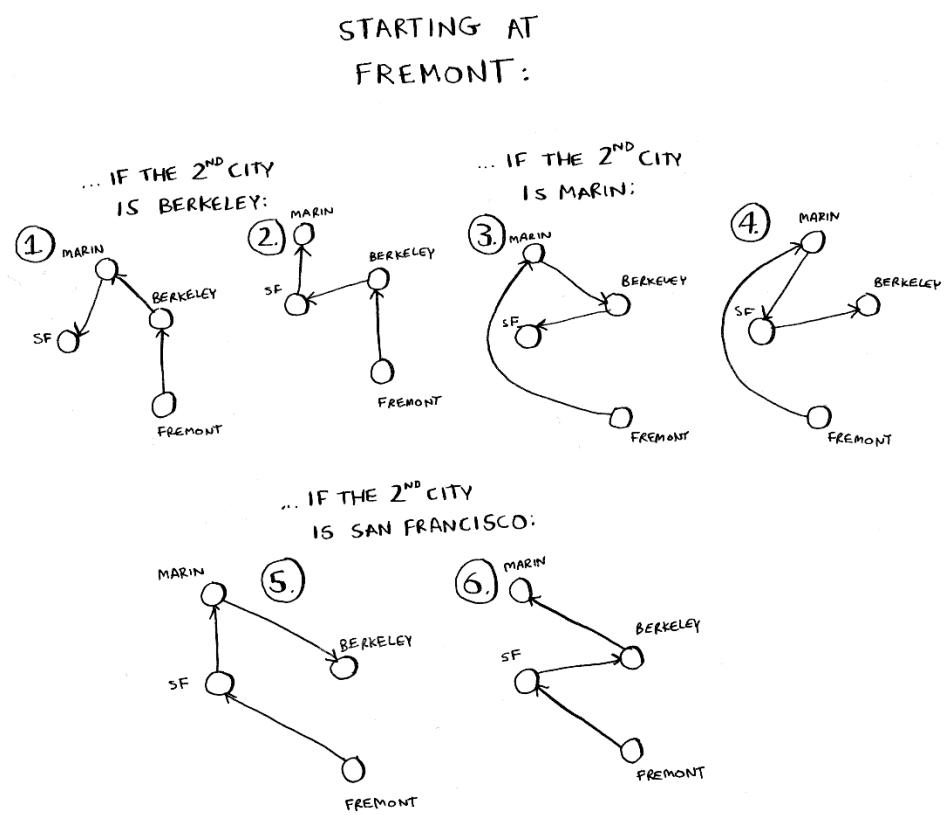
به طور کلی شش مسیر وجود. چون اگر از هر کدام از شهرها شروع کنیم ۲ شهر دیگه برای بازدید می‌مونه.



پس درنتیجه، سه شهر = شش راه ممکن.

۴ شهر

بیاين شهر فرمونت رو اضافه کنیم. حالا فرض کنیم مبدعا ما شهر فرمونت هستیم.



اگر از فرمونت شروع کنیم ۶ مسیر ممکن وجود دارد. و صب کنین! این دقیقاً مثل همون ۶ مسیری که قبل اکه ۳ تا شهر داشتیم برآش حساب کردیم. به جز الان تمام مسیرها یک شهر اضافه هم دارن، فرمونت! یک الگو اینجا هست. فرض کنین که شما ۴ شهر دارین و یک شهر رو به عنوان مبدأ انتخاب میکنین، مثلاً فرمونت. ۳ تا شهر دیگه موندن. و شما اینو از قبل میدونین که اگر ۳ شهر داشته باشیم ۶ مسیر متفاوت برای رسیدن به هر کدوم از اون شهرها هم داریم. اگر شما از فرمونت مسیر رو شروع کنین، ۶ مسیر ممکن برای شما وجود دارد. همچنین شما میتوانستین مبدأ رو یکی از شهرهای دیگه درنظر بگیرین.

STARTING
AT MARIN:

= 6 POSSIBLE ROUTES =

STARTING AT
SAN FRANCISCO:

= 6 POSSIBLE ROUTES =

STARTING AT
BERKELEY:

= 6 POSSIBLE ROUTES =

پس ۴ تا شهر داریم که میتوانیم از هر کدوم مسیرمونو شروع کنیم. که ۶ مسیر برای هر شهر که به عنوان مبدأ تعیین بشه وجود داره که این یعنی $4 \times 6 = 24$ مسیر ممکن.

الگو رو میتوانیں ببینین؟ هر بار که شما شهر رو اضافه میکنین به تعداد مسیرها ممکنی هم که شما باید محاسبه کنین اضافه میشه.

NUMBER
OF CITIES

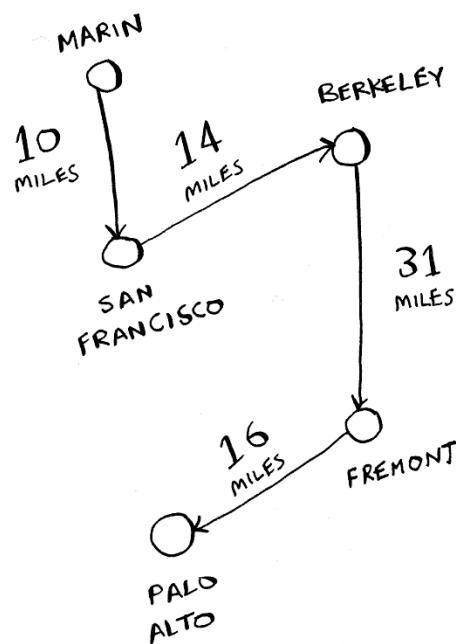
1	\rightarrow	1 ROUTE	
2	\rightarrow	2 START CITIES * 1 ROUTE FOR EACH START =	2 TOTAL ROUTES
3	\rightarrow	3 START CITIES * 2 ROUTES =	6 TOTAL ROUTES
4	\rightarrow	4 START CITIES * 6 ROUTES =	24 TOTAL ROUTES
5	\rightarrow	5 START CITIES * 24 ROUTES =	120 TOTAL ROUTES

چند مسیر ممکن برای ۶ شهر وجود داره؟ اگر حدستون ۷۲۰ بود، حدستون درسته. ۵۰۴۰ مسیر برای ۷ شهر، ۴۰۳۲۰ برای ۸ شهر.

به این الگو تابع فاکتوریل گفته میشه) یادتونه درباره این توی فصل ۳ خوندیم؟). پس $120 = 5! \cdot 10!$. فرض کنیں که شما ۱۰ شهر دارین. چند راه ممکن برای برقراری ارتباط بین این شهرها وجود داره؟ $3628800 = 10! \cdot$ شما باید تعداد بالای ۳ میلیون مسیر ممکن رو فقط برای ۱۰ شهر حساب کنیں. همینظور که میبینین تعداد مسیرهای ممکن به طور سریعی افزایش پیدا میکنه. به همین دلیله که حساب کردن راه حل درست برای فروشنده دوره گرد وقتی که تعداد شهرها زیاد باشه، غیر ممکنه. مسئله فروشنده دوره گرد و پوشش مجموعه ای هردو یک چیز مشترک در خودشون دارن: شما تمام راه حل های ممکن رو محاسبه میکنین و کوتاه ترین / سریعترین راه رو انتخاب میکنین. هردو این مسئله ها، ان-پی کامل هستند.

تقریب زدن

یک الگوریتم تقریبی خوب برای مسئله فروشنده دوره گرد چی میتونه باشه؟ یک چیز ساده که بتونه یک مسیر کوتاه رو پیدا کنه. ببینین خودتون میتونین قبل از اینکه به خوندن ادامه بدین به جوابی برسین؟ این کاری که من ترجیح میدم انجام بدم: یک شهر به طور دلخواه انتخاب کنیں. بعد از اون فروشنده دوره گره باید یک شهر برای بازدید کردن انتخاب کنه، در نتیجه اون نزدیک ترین شهر بعدی برای خودش انتخاب میکنه. فرض کنین اونا (فروشنده دوره گرد و بَکس) مارین رو به عنوان شروع انتخاب میکنن.



فاصله کلی مسیر: ۷۱ مایله. شاید کوتاه ترین مسیر نباشه اما هنوز تقریباً کوتاه حساب میشه.

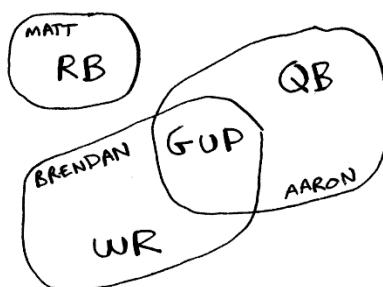
این هم یک تعریف کوتا از مسائل ان-پی کامل: حل بعضی از مسائل اینقدر سخته که این مسائل به عنوان مسائل خیلی سخت معروف شدن. مسائل فروشنده دوره گره و پوشش مجموعه ای دو تا مثال از این دست مسائل هستند. خیلی از افراد باهوش فک میکنند نوشتند یک الگوریتم که بتوانه سریع مسئله را حل کنه ممکن نیست.

چطور متوجه میشین که مسئله ان-پی کامل است؟

جونا داره برای تیم فوتبال خیالی (فوتبال آمریکایی) خودش تیم جمع میکنه. اون یک لیست از استعدادها و توانایی هایی بازکین ها باید داشته باشند در اختیار داره: یک کوارتربک خوب، یک مهاجم توپ بر خوب، بازکنی که در شرایط هوایی بارانی خوب عمل کنه، بازکنی که بتون حرف رو تحت فشار بزاره، و غیره. اون لیستی از بازکینان داره که هر کدام از این بازکینها بعضی از این توانایی های خواسته شده رو براورده میکنند.

PLAYER	ABILITIES
MATT FORTE	RB
BRENDAN MARSHALL	WR / GOOD UNDER PRESSURE
AARON RODGERS	QB / GOOD UNDER PRESSURE
...	...

جونا به تیمی نیاز داره که بتوانه تمام اون توانایی هارو براورده کنه. و اندازه تیم محدوده. جونا میگه "صب کن ببینم!" اون میفهمه که "این مسئله پوشش مجموعه ایه!".



جونا میتوانه از همون الگوریتم تقریبی برای درست کردن تیمش استفاده کنه:

۱. پیدا کردن بازکنی که بیشترین توانایی هایی رو براورده میکنه قبل از براورده نشدن.
۲. تکرار مرحله ۱ تا جایی که تمام توانایی های خواسته شده براورده بشن (یا به محدودیت بخوریم و فضا برا اضافه کردن فرد جدید تموم بشه).

مسئله ان-پی کامل همه جا پیداش میشه! این خوبه که بدونین مسئله که تلاش میکنین حلش کنین ان-پی کامله يا نه. اگر هست، پس در اونجا شما باید دست از تلاش کردن برای پیدا کردن راه حل دقیق کامل بردارین و بجاش بربین سراغ الگوریتم تقریبی. اما سخته که بگیم یک مسئله ان-پی کامل هست یا نه. غالبا یک تفاوت بسیار کوچیک بین مسائلی که خیلی راحت حل میشن و مسائل ان-پی کامل هست. به عنوان مثال، در فصل قبل، من خیلی درباره کوتاه ترین مسیر حرف زدم. شما الان میدونین که چطور از طریق کوتاه ترین مسیر از نقطه A به نقطه B بربین. اما اگر شما میخواین کوتاه ترین مسیری رو که چند نقطه رو بهم وصل میکنه پیدا کنین، که این همون مسئله فروشند دوره گره هست، این مسئله میشه ان-پی کامل. جواب کوتاه اینه: هیچ راه ساده ای نیست که بگیم مسئله که شما دارین روش کار میکنین ان-پی کامله يا نه. اما اینجا چنتا نکته بدرد بخور برآتون آوردم:

- الگوریتم شما خیلی سریع با آیتم ها یدم دستی و کم اجرا میشه اما با آیتم های زیاد سرعت میاد پایین.
- "تمام ترکیبات X ..." (اینطور مسئله ها) غالبا به مسائل ان-پی کامل اشاره میکن.
- آیا شما مجرورین تمام شرایط های محتمل مسئله X رو به این دلیل که نمیتونین به مسائل کوچک تر تقسیمش کنین، بررسی کنین؟ ممکنه این مسئله ان-پی کامل باشه.
- اگر مسئله شما شامل رشته ها و دنباله هاست (مثل رشته ای از شهرها در مسئله فروشند دوره گرد) و حل کردنش سخته، احتمالاً مسئله ان-پی کامله.
- اگر مسئله شما شامل مجموعه ها است (مثل مجموعه های ایستگاه های رادیویی) و حل کردنش سخته، احتمالاً ان-پی کامله.
- میتوانین مسئله تون رو مثل های پوشش مجموعه ای و فروشند دوره گرد بیانش کنین؟ اگر اینطوریه پس قطعاً مسئله ان-پی کامله.

تمرین ها

۸.۶ یک پستچی، باید بسته پستی ۲۰ خونه رو بهشون تحويل بده. اون باید کوتاه ترین مسیری رو پیدا کنه که از تمام اون ۲۰ خونه رد میشه. آیا این مسئله ان-پی کامله؟

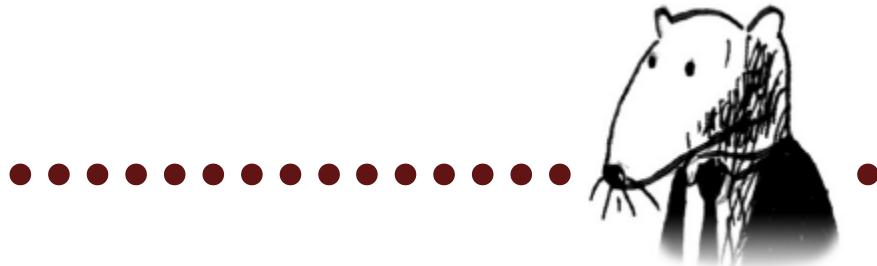
۸.۷ پیدا کردن بزرگترین گروه از افرادی که همدیگرو میشناسن، در مجموعه از مردم. آیا این مسئله ان-پی کامله؟

۸.۸ شما دارین نقشه آمریکا رو میکشین. و شما باید ایالت های مجاور هم رو با رنگ های متفاوت رنگ کنین. شما باید حداقل تعداد رنگی که برای کشیدن این نقشه نیاز دارین رو پیدا کنین، رنگ کردن باید طوری باشه که هیچ دو ایالت مجاور همی، باهم هم رنگ نباشند. آیا این مسئله ان-پی کامله؟

خلاصه

- الگوریتم های حریص به صورت محلی (تیکه های کوچیک) مسئله رو بهینه میکنن، به امید اینکه به راه حل بهینه کلی برسن.
- هیچ راه حل سریعی برای مسائل ان-پی کامل وجود نداره.
- اگر شما مسئله ای دارین که ان-پی کامل محسوب میشه، برگ برندتون اينه که از الگوریتم های تقریبی استفاده کنین.
- نوشتن الگوریتم های حریص راحته و سریع هم اجرا میشن، پس میتونن یک الگوریتم تقریبی خوبی برای ما درست کنن.

برنامه نویسی پویا (Dynamic programming)



در این فصل:

- شما درباره برنامه نویسی پویا یادخواهید گرفت، یک تکنیک برای حل مسائل سخت، به طوری مسئله رو به مسئله کوچک تر میشکنیم و اول سعی میکنیم اون مسئله کوچک رو حل کنیم.
- با استفاده از مثال ها، شما یادمیگرین که چطور یک راه حل برای یک مسئله با استفاده از تکنیک برنامه نویسی پویا پیدا کنید.

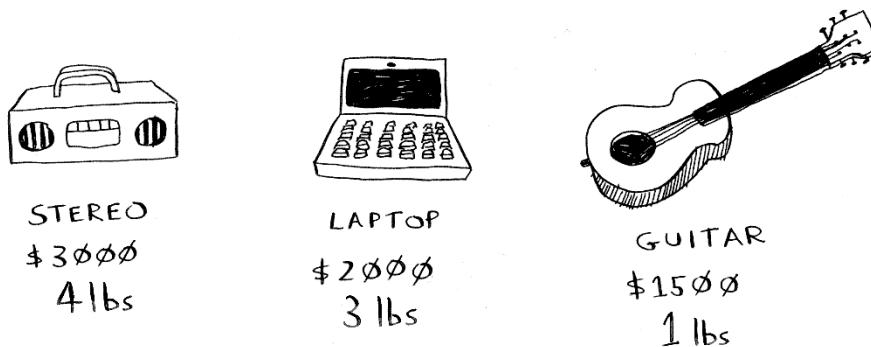


مسئله کوله پشتی



بیاین دوباره به مسئله فصل ۸ نگاهی بندازیم. شما یک دزد هستید و همراه خود کوله پشتی دارین که میتوانه ۴ پوند(نزدیک ۲ کیلوگرم) از وسیله هارو داخل خودش نگه داره.

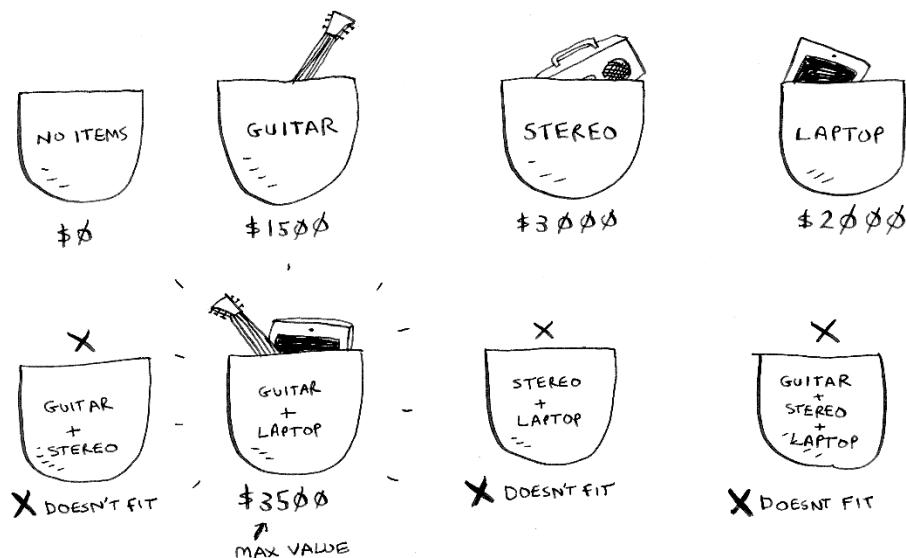
شما ۳ تا آیتم دارین که میتوانین توی کوله پشتی قرارش بدین.



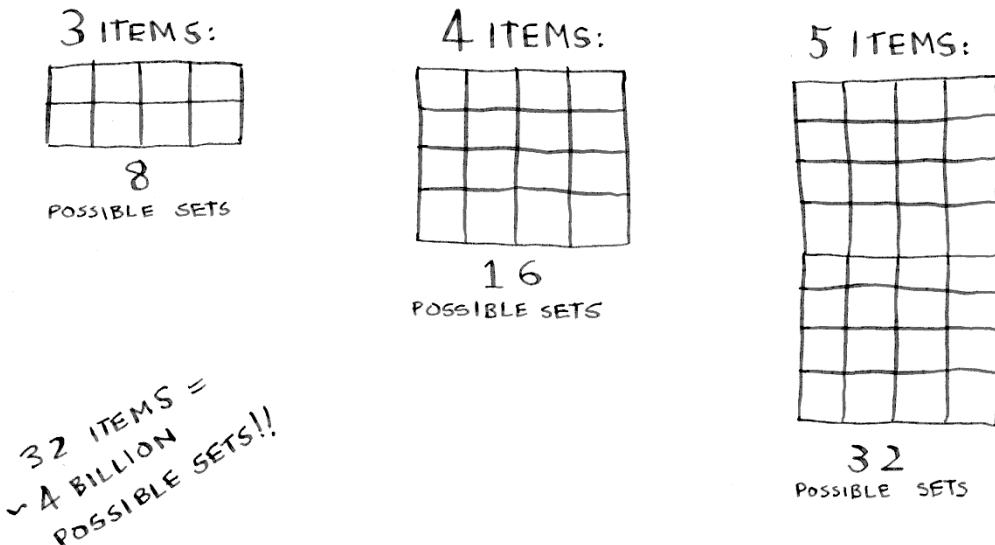
چه آیتم هایی رو باید بذدیم تا بتوانیم بیشترین سود رو داشته باشیم؟(متترجم: من واقعا بابت این مثال مسخره از شما معذرت میخوام 😞)

راه حل ساده

راحترین الگوریتم اینه: تمام مجموعه های ممکنی که میتوانین بردارین رو امتحان میکنین و بهترین مجموعه ای که به شما بیشترین سود رو بده پیدا و انتخاب میکنین.



این روش جواب میده اما خیلی کنده. فقط برای ۳ آیتم شما باید ۸ مجموعه محتمل رو بررسی کنین. برای ۶ آیتم باید ۱۶ مجموعه رو بررسی کنین. با هر بار آیتمی که اضافه میکنیم، تعداد مجموعه های محتملی که باید بررسی کنیم دو برابر میشه! این الگوریتم به اندازه $O(2^n)$ زمان میبره، که خیلی خیلی کنده.

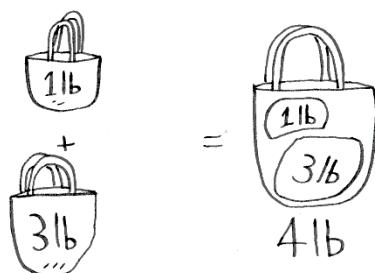


این الگوریتم برای هر تعداد کالای معقولی، غیر عملی است. در فصل ۸، شما دید که چطور این مسائل رو با یک روش تقریبی محاسبه کنین. اون راه حل به راه حل بهینه خیلی نزدیک خواهد بود، اما ممکنه راه حل بهینه نباشه. پس چطوری راه حل بهینه رو محاسبه و پیدا میکنین؟

برنامه نویسی پویا

جواب: با استفاده از برنامه نویسی پویا! بیاین ببینیم که الگوریتم برنامه نویسی پویا چطور قرار این مسئله رو حل کنه. برنامه نویسی پویا اول با حل زیرمسئله (بخش کوچیکی از مسئله اصلی) شروع میکنه و همینطور ادامه تا مسئله اصلی رو حل کنه.

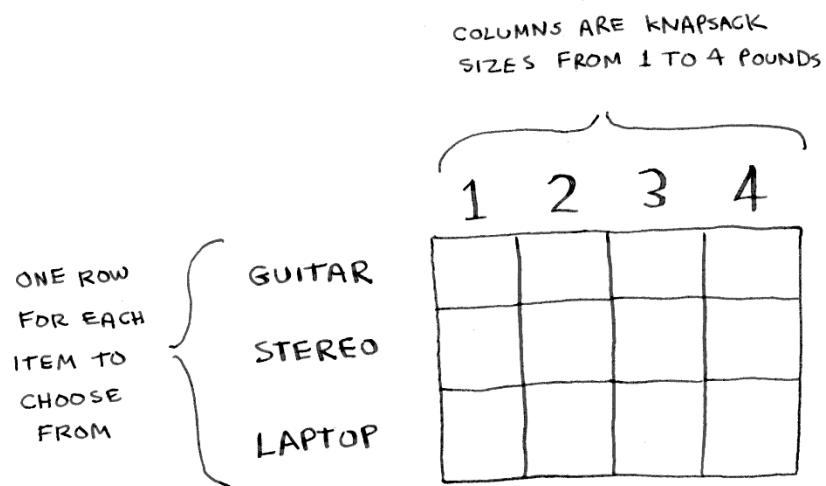
برای مسئله کوله پشتی، شما مسئله رو با کوله پشتی های کوچیک (یا زیرکوله پشتی) شروع به حل میکنین و همینطوری ادامه میدین تا بتونین مسئله اصلی رو حل بکنین.



برنامه نویسی پویا یک مفهوم پیچیدس، پس اگر الان کامل نفهمیدن که چی شده و این مفهوم چی داره میگه اصلا نگران نباشین. ما قرار خیلی از مثال هارو باهم بررسی کنیم.

من اول با نشون دادن اینکه این الگوریتم چطوری در عمل کار میکنه، شروع میکنم. بعد از اینکه این الگوریتم رو در عمل دیدین، خیلی سوالا برآتون پیش میاد! من تمام تلاشم میکنم تا بتونم به خوبی جواب سوالاتونو بدم.

هر الگوریتم برنامه نویسی پویا با یک شبکه یا یک جدول شروع میشه. این هم یک جدول یا شبکه برای مسئله کوله پشتی.



سطرهای این جدول آیتم ها هستن. و ستون هاش وزن کوله پشتی از ۱ پوند تا ۴ پوند. شما به تمام اون ستون ها نیاز دارین چرا که بهتون کمک میکنن تا بتونین ارزش کوله پشتی های کوچک رو محاسبه کنین.

برای شروع این جدول به صورت خالی رسم میشه. شما قراره که هر کدوم از سلول های یا خونه های این جدول رو پر کنین. و که این جدول کامل پر شد شما هم به جواب سوالتون میرسین! خواهشا کارایی که من میکنم رو دنبال کنین. جدول خودتون رو درست کنین و ما باهم دیگه پرش میکنیم.

سطر گیتار

من بعدا فرمول دقیق این الگوریتم رو برای محاسبه خونه های این جدول بهتون نشون میدم. اما فعلا بیاین الان وارد مثال بشیم. با سطر اول شروع میکنیم.

	1	2	3	4
GUITAR				
STEREO				
LAPTOP				

این سطر، سطر گیتاره، که ینی شما تلاش میکنین که گیار رو داخل کوله پشتی قرار بدین. در هر کدوم از خونه های این جدول باید یک تصمیم گیری داشته باشیم، یک تصمیم گیری راحت: آیا شما گیتار رو میدزدی یا نه؟ یادتون باشه شما دنبال مجموعه ای از آیتم ها هستین که از دزدیدنشون بیشترین سود رو ببرین.

اولین خونه یک کوله پشتی با فضای ۱ پونده (کلا اون خونه تو اون سطر ۱ پوند آیتم میتوانه تو خودش نگه داره). همچنین وزن گیتار هم ۱ پونده (متترجم: از فیزیکدانها و کسانی که فیزیک خوندن معذرت میخواه در واقع باید بگیم جرم اما خب لفظی که تو جامعه جا افتاده وزن هستش پس برای درک بهتر از همون وزن استفاده میکنم)، که یعنی گیتار داخل کوله پشتی جا میشه. پس درنتیجه ارزش این خونه در جدول برابر \$1500 هستش و شامل گیتار میشه.

بیاین شروع کنیم که جدول رو پر کنیم.

	1	2	3	4
GUITAR	\$1500 G			
STEREO				
LAPTOP				

بقیه خونه های جدول هم مثل همین روشی که رفتیم قراره که شامل لیستی از تمام آیتم هایی باشن که در اون نقطه و اون مرحله در کوله پشتی جا میشه.

بیاین به خونه بعدی نگاه کنیم. در اینجا شما یک کوله پشتی با فضای ۲ پوند که یعنی میتوانه ۲ پوند آیتم رو در خودش نگه داره در اختیار دارین. پس قطعاً گیتار هم در این کوله پشتی ۲ پوندی هم جا میشه!

	1	2	3	4
GUITAR	\$1500 G	\$1500 G		
STEREO				
LAPTOP				

دقیقاً این موضوع برای بقیه خونه های این سطر هم صدق میکنه و یکیه. یادتون باشه، این سطر اوله، پس شما الان فقط گیتار رو در دسترستون دارین که بتونین انتخابش کنین. شما طوری وانمود میکنین که انگار فعلاً اون دو تا آیتم دیگه برای دزدیدن در دسترس نیستن. (چون هنوز بهشون نرسیدیم).

	1	2	3	4
GUITAR	\$1500 G	\$1500 G	\$1500 G	\$1500 G
STEREO				
LAPTOP				

ممکنه تا اینجای کار یکمی گیج شده باشین. ممکنه بپرسین چرا دارین این کارو با کوله پشتی که فضاش ۱ پوند، ۲ پوند و غیرس دارین این کارو میکنین، درحالی مسئله داره درباره کوله پشتی ۴ پوندی حرف میزنه؟ یادتونه بهتون گفتم که برنامه نویسی پویا از مسئله کوچک شده شروع به حل میکنه و همینطوری پیش میره تا مسئله اصلی برسه؟ شما الان دارین زیرمسئله هارو حل میکنین تا بتونین مسئله اصلی رو حل کنین. همنطور که به خوندن ادامه بدین این موضوعات برآتون واضح تر میشن.

در این مرحله جدول شما باید این شکلی باشه.

	1	2	3	4
GUITAR	\$1500 G	\$1500 G	\$1500 G	\$1500 G
STEREO				
LAPTOP				

یادتون باشه شما میخواین ارزش آیتم های داخل کوله پشتی در بیشترین حالت خودش باشه. این سطر بهترین حدس رو با آیتم هایی که در دسترس داریم به ما نشون میده. خب پس تا الان، باتوجه به این سطر، اگر شما کوله پشتی که بتونه ۴ پوند رو تحمل کنه داشته باشین، نهایت ارزشی رو که یک آیتم میتونه داشته باشه و شما میتونین داخلش قرار بدین \$1500.

	1	2	3	4
GUITAR	\$1500 G	\$1500 G	\$1500 G	\$1500 G
STEREO				
LAPTOP				

←

OUR CURRENT
BEST GUESS
FOR WHAT THE
THIEF SHOULD STEAL:
THE GUITAR
FOR \$1500

شما میدونین که این راه حل نهایی نیست. همینطور که با الگوریتم پیش میریم، شما هم براورد خودتون رو از آیتم ها اصلاح میکنین.

سطر استریو

بیاین برمی سراغ سطر بعدی. این سطر متعلق به استریو. حالا که شما در سطر دوم قرار دارین، شما یا میتوانین استریو رو بذدین یا گیتار رو (یا اگر جا داشته باشین هردو رو). در هر سطر شما هم میتوانین آیتمی که در اون سطر وجود داره بذدین هم آیتم های که در سطر بالا اون وجود داره. پس درنتیجه شما فعلاً نمیتوانین لپ تاپ رو برای دزدین انتخاب کنین. اما شما میتوانین استریو و / یا گیتار رو بذدین. بیاین برمی سراغ خونه اول این سطر، یک کوله پشتی با فضای ۱ پوند. در حال حاضر آیتمی که بیشترین ارزش رو داره که میشه توی کوله پشتی ۱ پوندی جا داد،

. \$ ۱۵۰۰

	1	2	3	4
CURRENT MAX FOR A 1lb KNAPSACK	\$1500	\$1500	\$1500	\$1500
GUITAR	G	G	G	G
STEREO				
LAPTOP				
NEW MAX FOR A 1lb KNAPSACK				

آیا شما باید استریو رو بذدین؟

شما یک کوله پشتی با فضا ۱ پوند دارین. آیا استریو توی اون جا میشه؟ نه ، استریو خیلی سنگینه. چون شما نمیتوانین استریو رو در اون کوله پشتی جا کنین پس بیشترین ارزش برای کوله پشتی ۱ پوندی همون \$ ۱۵۰۰.

	1	2	3	4
GUITAR	\$1500	\$1500	\$1500	\$1500
STEREO	G			
LAPTOP				

برای دوتا خونه بعدی هم همینطوره، این کوله پشتی ها فضای ۲ پوندی و ۳ پوندی دارن، مقدار قبلی جفتشون \$ ۱۵۰۰ بوده و از اونجایی که استریو برای این کوله پشتی ها هم سنگینه همون مقدار قبلی رو برای این خونه ها قرار میدیم.

	1	2	3	4
GUITAR	\$1500 G	\$1500 G	\$1500 G	\$1500 G
STEREO	\$1500 G	\$1500 G	\$1500 G	
LAPTOP				

اما چی میشه اگر شما یک کوله پشتی با فضای ۴ پوند داشته باشین؟ آها : بالاخره استریو داخل کوله پشتی جا میشه! ارزش قبلی این کوله پشتی $\$1500$ بود ، اما اگر شما به جای گیتار داخل این کوله پشتی ۴ پوندی ، استریو رو قرار بدین ، قیمتیش میشه $\$3000$! پس بیان استریو رو برداریم.

	1	2	3	4
GUITAR	\$1500 G	\$1500 G	\$1500 G	\$1500 G
STEREO	\$1500 G	\$1500 G	\$1500 G	\$3000 ' 115
LAPTOP				

شما قیمت قبلی رو که تخمین زده بودین بروزرسانیش کردین! اگر شما یک کوله پشتی ۴ پوندی داشته باشین میتوانین کالایی حداقل به ارزش $\$3500$ داخش جا بدین. اگر نگاهی به جدول بندازین میبینین که دارین به طور تدریجی قیمت های تخمینی رو آپدیت میکنین (بروزرسانی میکنین).

	1	2	3	4
GUITAR	\$1500 G	\$1500 G	\$1500 G	\$1500 G
STEREO	\$1500 G	\$1500 G	\$1500 G	\$3000 ' 115
LAPTOP				

← OLD ESTIMATE
 ← NEW ESTIMATE
 ← FINAL SOLUTION

سطر لپ تاپ

بیان همین کارای قبلی با لپ تاپ انجام بدیم! وزن لپ تاپ ۳ پونده ، پس در کوله پشتی های ۱ یا ۲ پوندی جا نمیشه. تخمین ارزش برای خونه ها اول و دوم همون $\$1500$ باقی میونه.

	1	2	3	4
GUITAR	\$1500 G	\$1500 G	\$1500 G	\$1500 G
STEREO	\$1500 G	\$1500 G	\$1500 G	\$3000 S
LAPTOP	\$1500 G	\$1500 G		

بخش کوله پشتی ۳ پوندی، قیمت قبلیش \$1500 بود اما الان شما میتوانین بجای گیتار، لپ تاپ رو بردارین، و همونطور که میدونین لپ تاپ \$2000 میارزه. بخش قیمت جدید این خونه در جدول ! \$2000

	1	2	3	4
GUITAR	\$1500 G	\$1500 G	\$1500 G	\$1500 G
STEREO	\$1500 G	\$1500 G	\$1500 G	\$3000 S
LAPTOP	\$1500 G	\$1500 G	\$2000 L	

بخش کوله پشتی ۴ پوندی قرار خیلی جالب بشه. این مهمترین بخش. آخرین تخمین قیمت این کوله پشتی \$3000. شما میتوانین لپ تاپ رو در کوله پشتی قرار بدین، اما لپ تاپ فقط \$2000 میارزه.

\$3000 vs \$2000
STEREO LAPTOP

همممم، این به اندازه تخمین قیمت قبلی خوب نیست. اما یه لحظه صب کنین! وزن لپ تاپ فقط ۳ پونده، پس ۱ پوند وزن دیگه در کوله پشتی ۴ پوندی فضا داریم. شما میتوانین یک چیزی بجای اون ۱ پوند خالی قرار بدین.

\$3000 vs $\left(\begin{matrix} \$2000 & + & ? & ? & ? \\ LAPTOP & & & & \end{matrix} \right)$
STEREO $\frac{1 LB OF FREE SPACE}{}$

بالرژش ترین وسیله ای که میتوانیم داخل فضای ۱ پوندی کوله پشتی قرار بدم چیه؟ خب ، درواقع شما داشتین تا همینجا حسابش میکردین.

1	2	3	4
\$1500	\$1500	\$1500	\$1500
G	G	G	G
↓	↓	↓	
\$1500	\$1500	\$1500	\$3000
G	G	G	S
↓	↓	...	
\$1500	\$1500	\$2000	
G	G	'L'	

براساس آخرین و بهترین تخمینی که برای فضا ۱ پوندی زدیم، شما میتوانین گیتار رو به جای اون فضای ۱ پوندی قرار بدین، که اون گیتار $\$1500$ میارزه. پس در واقع قیاس اصلی چیزی که این زیر میپینین.

ممکنه برآتون سوال شده باشه که چرا داشتین بالرزترین مقادیر رو برای کوله پشتی ها کوچک تر حساب میکردین. امیدوارم الان دلیلشو متوجه شده باشین و منطقی بنظر بیاد که چرا نیاز به حساب اون داشتیم. وقتی فضای خالی برای ما میمونه میتوانیم از جواب های زیرمسئله ها برای پر کردن اون فضا استفاده کنیم. پس اینجا بهتر لپ تاپ رو با گیتار به ارزش \$۳۵۰۰ برداریم.

جدول نهایی به این شکل در میاد.

	1	2	3	4
GUITAR	\$1500 ↓ G	\$1500 ↓ G	\$1500 ↓ G	\$1500 G
STEREO	\$1500	\$1500	\$1500 G	\$3000 S
LAPTOP	\$1500 ↓ G	\$1500 ↓ G	\$2000 L	\$3500 L G
				↑ THE ANSWER!

و جواب اونجاست (آخرین خونه جدول): با ارزشترین چیز هایی که در کوله پشتی میتوانستیم جا کنیم \$ ۳۵۰۰ میارزید، که متشكل از گیتار و لپ تاپ بود.

ممکنه فکر کنین که من از یک فرمول متفاوت برای محاسبه ارزش آخرین خونه این جدول استفاده کردم. این به این دلیله که من از روی یک سری مفاهیم پیچیده غیرضروری هنگام پرکردن خونه قبلی رد شدم. هر خونه از این جدول به همون فرمول که خونه آخر رو پر کردیم، محاسبه میشه. این از فرمول سادش:

$$\text{CELL}[i][j] = \max \text{ of } \left\{ \begin{array}{l} 1. \text{ THE PREVIOUS MAX (VALUE AT CELL } [i-1][j] \text{) } \\ \text{VS} \\ 2. \text{ VALUE OF CURRENT ITEM + VALUE OF THE REMAINING SPACE } \\ \text{CELL}[i-1][j - \text{ITEM'S WEIGHT}] \end{array} \right.$$

شما میتونین از این فرمول برای همه خونه ها در این جدول استفاده کنین و شما باید در آخر به همون چیزی برسین که من رسیدم. یادتون درباره حل کردن زیرمسئله برآتون گفتم؟ شما راه حل ها زیرمسئله هارو با هم ادغام میکنین تا مسئله اصلی رو حل کنین.



سوالات رایج مسئله کوله پشتی

شاید این موضوع هنوز برآتون عجیب باشد. این بخش به یک سری از سوالات رایج درباره این بخش پاسخ میده.

چی میشه اگر یک آیتم دیگه به محصولات اضافه کنین؟



فرض کنین شما میفهمین که یک آیتم چهارمی هست که میتونین بذدینش (متترجم: ۷) و قبلا متوجهش نشدهین! شما میتونین یک آیфон رو هم در کنار آیتم های قبلی هم بذدین.

آیا شما باید با اضافه شدن آیфон، همه چیز رو از اول محاسبه کنین؟ نه. برنامه نویسی پویا به

طور تدریجی براساس برآورد های شما پیش میرود. تا اینجا که حساب کردیم اینها با

ارزشترین آیتم هایی بودن که تونستیم انتخاب کنیم.

	1	2	3	4
GUITAR	\$1500 G	\$1500 G	\$1500 G	\$1500 G
STEREO	\$1500 G	\$1500 G	\$1500 G	\$3000 S
LAPTOP	\$1500 G	\$1500 G	\$2000 L	\$3500 LG

که این یعنی با یک کوله پشتی ۴ پوندی، میتونین کالاهایی به ارزش ۳۵۰۰ دلار بذدین. شما فکر کردین که نهایت ارزشی که میتونین بردارین همینه. اما بیاین یک سطر جدید برای آیفون به جدول اضافه کنیم.

	1	2	3	4
GUITAR	\$1500 G	\$1500 G	\$1500 G	\$1500 G
STEREO	\$1500 G	\$1500 G	\$1500 G	\$3000 S
LAPTOP	\$1500 G	\$1500 G	\$2000 L	\$3500 LG
IPHONE				

↑
NEW ANSWER

اینطور که معلومه شما یک مقدار حداکثر جدید دارین. سعی کنین خودتون، قبل از اینکه به خوندن ادامه بدین، این سطر رو پر کنین.

بیاین برمیم سراغ خونه اول این سطر. آیفون در کوله پشتی ۱ پوندی جا میشه. مقدار قبلی این کوله پشتی \$1500 بود، اما آیفون ۲۰۰۰ دلار میازد. بیاین بحاش آیفون رو برداریم.

	1	2	3	4
GUITAR	\$1500 G	\$1500 G	\$1500 G	\$1500 G
STEREO	\$1500 G	\$1500 G	\$1500 G	\$3000 S
LAPTOP	\$1500 G	\$1500 G	\$2000 L	\$3500 LG
IPHONE	\$2000 I			

در خونه بعدی این سطر شما میتوانید هم آیفون و هم گیتار رو داخل کوله پشتی جا کنید.

\$1500 G	\$1500 G	\$1500 G	\$1500 G
\$1500 G	\$1500 G	\$1500 G	\$3000 S
\$1500 G	\$1500 G	\$2000 L	\$3500 LG
\$2000 I	\$3500 IG		

برای خونه ۳ ، بازم نمیتوانید چیز بهتری از آیفون و گیتار برداریدن، پس همونارو بزاریدن باشه.

برای خونه آخر موضوع خیلی جالب میشه. مقدار حداکثری که تا الان در اختیار داریم ۳۵۰۰ دلاره. شما میتوانید برای کوله پشتی ۴ پوندی آیفون رو برداریدن و به اندازه ۳ پوند برآتون فضا باقی بمونه.

$$\begin{array}{c} \$3500 \\ \text{LAPTOP + GUITAR} \end{array} \quad \text{vs} \left(\begin{array}{c} \$2000 \\ \text{IPHONE} \end{array} + \underbrace{\begin{array}{c} ? \\ ? \\ ? \end{array}}_{\text{3 LBS FREE}} \right)$$

اون ۳ پوند فضای خالی خودش ۲۰۰۰ دلار میارزه! ۲۰۰۰ دلار از طریق آیفون + ۲۰۰۰ از طریق زیرمسئله قبلی که داشتیم: که جمعاً میشه ۴۰۰۰ دلار. یک مقدار حداکثری جدید!

اینم از نتیجه نهایی جدول جدیدمون.

\$1500 G	\$1500 G	\$1500 G	\$1500 G
\$1500 G	\$1500 G	\$1500 G	\$3000 S
\$1500 G	\$1500 G	\$2000 L	\$3500 LG
\$2000 I	\$3500 IG	\$3500 IG	\$4000 IL

↑
NEW
ANSWER

سوال: آیا ممکنه که ارزش و مقدار ستون پایین بیاد؟ آیا اصلاً این ممکنه؟

1	2	3	4
\$1500	\$1500	\$1500	\$1500
Ø	Ø	Ø	\$3000

MAX VALUE
 DECREASED
 AS WE
 WENT ON

قبل از اینکه به خوندن ادامه بدین به جوابش فکر کنیں.

جواب: نه. در هر بار تکرار (هر بار که میریم یک خونه در ستون به سمت پایین میریم)، شما مقدار حداکثری رو که تخمين زدين رو در اون خونه ذخیره میکنین. وضعیت این مقداری تقریبی، نمیتونه از چیزی که قبلاً بوده بدتر بشه.

تمرین

۹.۱ فرض کنین شما میتوانین یک آیتم دیگه هم بذدين: یک ام پی تری پلیر. وزنش ۱ پوند و ۱۰۰۰ دلار میازه. آیا باید برش دارین؟

اگر ترتیب سطرها رو عوض کنیم چه اتفاقی میفته؟

آیا جواب متفاوت میشه؟ فرض کنین سطر هارو با این ترتیب پر میکردين: استریو، لپ تاپ، گیتار. جدول چه شکلی میشه؟ قبل از اینکه ادامه بدین، این جدول رو خودتون پر کنین.

جدول این شکلی میشه.

	1	2	3	4
STEREO	Ø	Ø	Ø	\$3000
LAPTOP	Ø	Ø	\$2000	\$3000
GUITAR	\$1500	\$1500	\$2000	\$3500

↓
 ↓
 ↓

جواب تغییری نمیکنه. ترتیب سطرهای اصلًا مهم نیست.

میتونیم جدول رو به جای اینکه سطري پر کنيم، ستونی پر کنيم؟

خودتون امتحانش کنین! انجام این کار برای این مسئله تغييری ايجاد نمیکنه. ولی ممکنه برای يك سري مسائل دیگه تغيير ايجاد کنه.

چه اتفاقی ميفته اگر که شما آيتم هاي کوچكتري بهش اضافه بکنин؟

فرض کنین شما میتونین يك گردنبند هم بدزدين. وزنش به اندازه ۰.۵ پونده (تقریباً ۲۳۰ گرم) و ۱۰۰۰ دلار ارزش داره. تا اینجا فرضمون بر این بود که وزن تمام کالاها اعداد صحیح هستند و اعشاری نیستند. حالا شما تصمیم به برداشتن گردنبند گرفتین. و وقتی اون رو بردارین در کوله پشتی ۴ پوندی، ۳.۵ پوند دیگه باقی میمونه. بالازترین چیزی که شما میتونین در ۳.۵ پوند فضا جا بدین چیه؟ نمیدونین! شما فقط ارزش کالاهای رو با توجه به کوله پشتی های ۱ و ۲ و ۳ و ۴ پوندی محاسبه کردین. شما باید ارزش يك کوله پشتی ۳.۵ پوندی رو بدونین (بدونین که حداقل چقدر میشه کالا بالازش توش جا داد).

بخاطر گردنبند، شما در جدول باید خونه های با فضای کوچکتر رو هم در نظر بگیرین، پس جدول باید تغيير کنه.

	0.5	1	1.5	2	2.5	3	3.5	4
GUITAR								
STEREO								
LAPTOP								
JEWELRY								

آيا میتونین کسري از يك آيتم رو بدزدين؟ (مثلاً نصف يك کالا رو بدزدين؟)

فرض کنین شما يك دزد هستین و رفتین سراغ يك خواربار فروشی. شما میتونین کيسه های عدس و برنج رو بدزدين. اگر کل اين کيسه در کوله پشتی شما جا نشد، میتونین اون کيسه رو باز کنین و هرچقدر که میتونین با خودتون حمل کنین رو از داخلش بردارین. پس نه کل کيسه برداشتین نه کلا دست خالي اوميدین ، بلکه کسري از يك آيتم رو برداشتین. شما میتونین کسري از يك آيتم رو بردارین. حالا چطور با استفاده از برنامه نويسي پویا اين کار رو انجام ميدين؟

جواب: شما نمیتونین با برنامه نويسي پویا اين کار رو انجام بدین. در برنامه نويسي پویا، يا شما کل آيتم رو برميدارین يا برنميدارین. هیچ راهی نداره که شما بتونین نصف يا کسري از يك آيتم رو دراين روش بردارين.

اما این حالت از مسئله خیلی راحت با الگوریتم های حریص حل میشه! اول از همه، تا جایی که میتوانیم از بالرژش ترین آیتم برداریم. وقتی که اون آیتم تموم شد، بین سراغ آیتم با ارزش بعدی و تا جایی که میتوانیم ازش برداریم و الی آخر.

برای مثال، شما این ۳ آیتم رو برای انتخاب در اختیار دارین.

		
QUINOA \$6/lb	DAL \$3/lb	RICE \$2/lb



هر پوند از کوینو از همه هر پوند از آیتم های دیگه گرونتره. پس تمام کوینویی که میتوانیم با خودتون حمل کنیم رو بردارین. اگر این آیتم کل کوله پشتیتونو پر کنه، دیگه نمیتوانیم چیزی بردارین و این بهترین کاری بود که میتوانستین انجام بدین. اگر هم کوینو تموم بشه و شما هنوز داخل کوله پشتیتون جا داشته باشین، بین سراغ بالرژشترین آیتم بعدی و همینطوری تا آخر.

بهینه سازی برنامه سفر خود

فرض کنین برای سپری کردن یک تعطیلات عالی به لندن سفر میکنین. دو روز برای اونجا وقت دارین و خیلی کارا هست که شما میخواین انجامش بدین. شما نمیتوانین همه کارها رو انجام بدین پس یک لیست درست میکنین.

ATTRACTION	TIME	RATING
WESTMINSTER ABBEY	1/2 DAY	7
GLOBE THEATER	1/2 DAY	6
NATIONAL GALLERY	1 DAY	9
BRITISH MUSEUM	2 DAYS	9
ST. PAUL'S CATHEDRAL	1/2 DAY	8

برای هر کدام از مکان هایی که میخواین بین بینیش، داخل لیست مینویسین که بازدید از اون مکان چقدر طول میکشه و بهش امتیازی میدین که مشخص میکنه که شما چقدر اشتیاق برای دیدن اون مکان ها دارین. آیا میتوانین براساس لیستی که تهیه کردین، پیدا کنین که چه مکان هایی رو باید بینین؟

این دوباره همون مسئله کوله پشتیه! به جای کوله پشتی شما یک محدودیت زمانی دارین و بجای استریوها و لپ تاپ ها، لیستی از مکان هایی دارین که میخواین بین و بینیشون. قبل از اینکه به خوندن ادامه بدین، جدول برنامه نویسی پویا این مسئله رو برای خودتون بکشین.

جدول این مسئله این شکلی میشه.

	$\frac{1}{2}$	1	$1\frac{1}{2}$	2
WESTMINSTER				
GLOBE THEATRE				
NATIONAL GALLERY				
BRITISH MUSEUM				
ST. PAUL'S				

درست کشیدینش؟ حالا جدول رو پر کنیم. در نهایت چه مکان هایی رو باید بین و ببینیم؟ اینم از جواب.

	$\frac{1}{2}$	1	$1\frac{1}{2}$	2
WESTMINSTER	7w	7w	7w	7w
GLOBE THEATRE	7w	13WG	13WG	13WG
NATIONAL GALLERY	7w	13WG	16WN	22WGN
BRITISH MUSEUM	7w	13WG	16WN	22WGN
ST PAUL'S	8s	15WS	21WGS	24WNS

↑
FINAL ANSWER:
WESTMINSTER ABBEY,
NATIONAL GALLERY,
ST. PAUL'S CATHEDRAL

مدیریت کردن آیتم هایی که بهم وابسته اند

فرض کنیم شما میخواین به پاریس سفر کنین ، پس لیستی از جاهایی که میخواین بازدید کنین تهیه میکنین.

EIFFEL TOWER	$1\frac{1}{2}$ DAY	8
THE LOUVRE	$1\frac{1}{2}$ DAY	9
NOTRE DAME	$1\frac{1}{2}$ DAY	7

بازدید از این مکان ها خیلی زمان میبره، به این دلیل که شما اول باید از لندن به پاریس سفر کنین. این خودش نصف روز زمان میبره. اگر بخواین به تمام اون سه آیتم سر بزنین ۴ روز و یک نصفه روز طول میکشه.

صب کنین، یه جای کار میلنگه. نیاز نیست برای بازدید هر کدام از این آیتم ها یکبار به پاریس سفر کنین. وقتی که به پاریس سفر کردین هر آیتم باید یک روز طول بکشه. پس باید این شکلی باشه که :

یک آیتم در روز + یک نصفه روز سفر به پاریس = ۳.۵ روز طول میکشه نه ۴.۵. اگر ایفل رو توی کوله پشتیتون قرار بدین رفتن به موز لور ارزون در میشه ، تنها یک روز ازتون زمان میبره نه ۱.۵ روز. خب این مسئله رو چطور در برنامه نویسی پویا نشون میدین؟

نمیتونین این کارو بکنین. برنامه نویسی پویا به این دلیل قدر تمنده که میتوانه زیرمسئله ها رو حل کنه تا از اونا برای حل مسئله اصلی استفاده کنه. برنامه نویسی پویا زمانی کار میکنه که هر زیرمسئله به صورت گسته باشد وقتی که زیر مسئله ها بهم وابسته باشند برنامه نویسی پویا جواب نمیده. اینی یعنی نمیتونیم راهی داشته باشیم که این مسئله پاریس رفتن رو با برنامه نویسی پویا محاسبه کنیم.

آیا ممکن است برای حل مسئله به بیشتر از دو کوله پشتی کوچک نیاز داشته باشیم؟

ممکنه که بهترین راه حل ، دزدین بیشتر ۲ آیتم رو شامل بشه. جوری که این الگوریتم کار میکنه اینطوری که شما نهايانا محتويات دو کوله پشتی رو (کوله پشت کوچک) رو باهم ادغام میکنیم. شما هیچ وقت بیشتر از ۲ زيرکوله پشتی(کوله پشتی کوچیک) نخواهید داشت. اما ممکنه اون زیر کوله پشتی ها خودشون زیرکوله پشتی ها خودشونو داشته باشن.



آیا ممکنه که کامل ترین راه حل ، کل فضای کوله پشتی رو پر نکنه؟

بله. مثلا فرض کنین شما، الماس رو هم میتونستین بددین.

این یک الماس خيلي بزرگه: وزنش ۳.۵ پونده. یک ميليون دلار ارزشش، خيلي بیشتر از چيزی ها ديگه. شما قطعا باید اونو بردارین. اما هنوز نیم پوند فضا باقی مونده. وهیچ چيز اون فضا رو پر نمکینه.

تمرین

فرض کنین دارین ميرين جايی برای کمپ کردن (چادر زدن) ، شما کوله پشتی دارين که ميتوشه ۶ پوند رو تحمل کنه. و ميتوينين آيتم های زير رو برای کمپ بردارين. هر کدوم از اين آيتم ها ارزشی دارن(امتيازی بهشون نسبت داده شده) که هرچقدر اين ارزش بيشتر باشه ، آيتم مهم تريه:

- آب ، ۳ پوند ، ۱۰
- کتاب ، ۱ پوند ، ۳
- غذا ، ۲ پوند ، ۹
- ژاكت ، ۲ پوند ، ۵
- دوربین عکاسي ، ۱ پوند ، ۶

بهنيه ترين مجموعه اي که ميتوينين با خودتون به اين کمب ببرين چие؟

بزرگترین زير رشته مشترک

تا الان يك مسئله از برنامه نويسي پويا رو ديدин. چه چيزايی برداشت کردیم؟

- برنامه نويسي پويا وقتی کاربرديه که شما سعی ميکنيم مسئله اي که داخلش محدوديت دارييم رو به حالت بهينه درش بياريم و حلش کنيم. در مسئله کوله پشتی، شما باید ارزش کالاهايی که ميخواستين بذدين در بالاترين حالت ممکن باشه که اين مسئله با اندازه و وزني که کوله پشتی ميتوونست تحمل کنه محدود شده بود.

- زمانی ميتوينين از برنامه نويسي پويا استفاده کنین که بشه اون مسئله رو به زير مسئله های گسته از هم تقسيم کرد، به طوری که اين زير مسئله ها بهم وابسته نباشن.

پيدا کردن راه حل که از برنامه نويسي پويا استفاده کنه ميتوونه سخت باشه. اين چيزيه که ميخوايم توی اين بخش روش تمرکز کنيم. يك سري نكته های کلی برای دنبال کردن اين موضوع:

- هر مسئله برنامه نويسي پويا ، شامل جدول ميشه.
- مقادير داخل خونه های جدول چيزايی هستن که شما تلاش ميکنيں بھينش کنین. برای مسئله کوله پشتی، مقادير و ارزش ها ، مقادير کالاهايی بودن که ميخواستيم برداريم.
- هر خونه يا سلول از جدول يك زير مسئله س ، پس به اين فكر کنین که چطور ميتوينين مسئله تون رو به يك زيرمسئله تقسيم کنین. اين بهتون کمک ميکنه که بفهمين گير مسئله کجاست.

بياين برييم سrag يك مثال ديگه. فرض کنین شما سایت dictionary.com رو اداره ميکنيں. يك نفر مياد و يك کلمه رو داخل سایت تايپ ميکنه، و شما بهش تعريف اون کلمه رو ميدين.

شما میخواین اگر یک نفر یک کلمه رو با غلط املایی وارد کنه، شما بتونین اون کلمه ای که اون فرد مدنظرش بوده حدس بزنین. الکس داره دنبال کلمه **fish** میگرده، اما به اشتباه کلمه **hish** رو وارد میکنه. اما این کلمه توی دیکشنری نیست اما شما لیستی از کلمات دارین که شبیه این کلمه هستند.

SIMILAR TO "HISH":

- FISH
- VISTA

(این یک مثال خیلی سادس، در نتیجه لیست پنهانی شامل ۲ کلمه س اما در واقعیت این لیست میتوانه هزار تا کلمه رو شامل بشه).

الکس **hish** رو تایپ کرد. حالا منظور الکس کدوم کلمه میتونسته باشه؟ **fish** یا **vista**؟

ساختن جدول

جدول که برای این مسئله میخوایم پیاده کنیم چه شکلی میشه؟ نیازه تا به این سوالات جواب بدین:

- ارزش خونه های جدول (مقادیر خونه های جدول) باید چی باشن؟
- چطور این مسئله رو به زیرمسئله تقسیم میکنیں؟
- محور جدول چطوری باید باشه (ستون و سطراش چی باشن)؟

در برنامه نویسی پویا شما میخواین یک چیزی رو به حداکثر برسونین. در این حالت، شما میخواین طولانی ترین زیر رشته ای رو که دو کلمه در اون زیررشته باهم مشترک هستند رو پیدا کنین. کلمات **hish** و **fish** در چه زیررشته ای باهم مشترک هستند؟ **vista** چطور؟ این چیزیه که شما میخواین حساب کنین.

یادتون باشه که، اغلب، مقادیر خانه های جدول چیز هایی هستن که شما میخواین به حداکثر برسونید یا حداکثر استفاده رو ازشون ببرید. در این موردی که میخوایم بررسی کنیم مقادیر احتمالاً عدد باشن: طول طولانی ترین زیر رشته ای که دو رشته در اون باهم مشترک هستن.

خب چطور این مسئله رو به مسئله کوچیک تر خورد میکنین؟ در اینجا شما میتوانین زیر رشته هارو باهم مقایسه کنین. به جای اینکه بیاین و **fish** و **hish** رو بررسی کنین میتوانین اول بیاین برعین سراغ بررسی کردن **.fis**. هر خانه در جدول شامل طول طولانی ترین زیر رشته مشترکی که این دو در اون باهم مشترک هستن. پس با چیزایی که

گفته شد این سر نخ رو به شما میده که احتمال زیاد محورها در جدول همان دو کلمه باشن. پس در نتیجه احتمال زیاد جدول به این شکل درمیاد.

	H	I	S	H
F				
I				
S				
H				

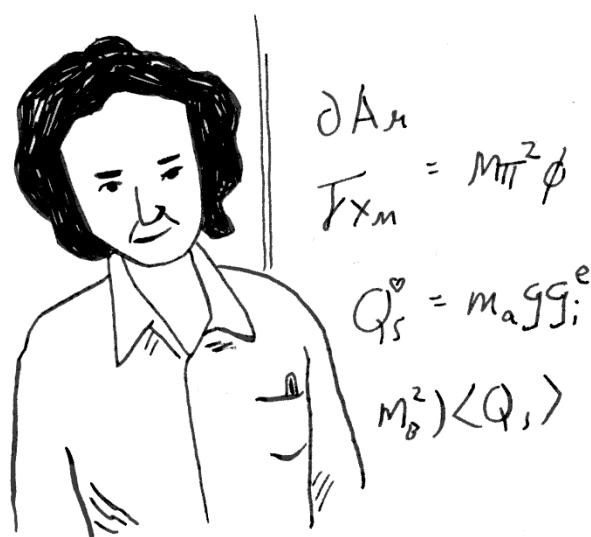
اگر این جدول برآتون کمی نامفهومه نگران نباشین. این مبحث، مبحث پیچیده ایه برای همین درس دادنشو گذاشتم آخر کتاب! یکم دیگه بهتون تمرینی میدم تا خودتون بتونین مبحث برنامه نویسی پویا رو تمرین کنین.

پر کردن جدول

حالا شما یک دید خوب از اینکه جدول باید چه شکلی باشه در اختیار دارین. حالا فرمول پر کردن جدول چطوریه؟ شما یکم میتونین تقلب کنین چون از قبل میدونین راه حل چطور باید باشه. fish و Hish یک زیرشته مشترک به طول ۳ دارن: .ish

اما همچنان این موضوع فرمولی برای استفاده کردن از این جدول به شما نمیده. گاهی اوقات دانشمندان داده درباره استفاده از الگوریتم فاینمن باهم شوخی میکنن. اسم الگوریتم فاینمن (feynman algorithm) از فیزیکدان مشهور ریچارد فاینمن گرفته شده و اینطوری کار میکنه که:

۱. مشکل رو بنویسین.
۲. خیلی سخت راجبش فک کنین.
۳. راه حل رو بنویسین.



دانشمندان داده یک گوله نمکن !

واقیعت اینه که در اینجا هیچ راه ساده‌ای برای محاسبه این فرمول وجود نداره. شما باید برای پیدا کردن فرمولی که برای مسئله کار بکنه آزمون و خطا بکنین. بعضی مواقع الگوریتم‌ها یک دستور کار دقیق نیستند. اونا یک چهاچوب هستن که شما ایدتون رو براساس اونها شکل میدین.

سعی کنین خودتون یک راه حل برای این مسئله پیدا کنین. من یک راهنمایی بهتون میکنم – بخشی از جدول به این شکله:

	H	I	S	H
F	0	0		
I				
S			2	0
H				3

مقادیر دیگه جدول چیا میتونن باشن؟ یادتون باشه که هر خونه در جدول یک زیرمسئله‌س. چرا خونه (۳ و ۳) دارای مقدار ۲؟ چرا خونه (۴ و ۳) دارای مقدار صفر؟

بعد از اینکه خودتون برای بدست آوردن فرمول تلاش کردین به خوندن ادامه بدین. حتی اگر به جواب درستی نرسیدین توضیحی که برای جواب این مسئله دادم حتما مطالعه کنین چون خیلی منطقی به نظر میرسه.

راه حل

این جدول نهاييه.

	H	I	S	H
F	0	0	0	0
I	0	1	0	0
S	0	0	2	0
H	0	0	0	3

اینم هم فرمول من برای پر کردن هر خونه از جدول:

1. IF THE LETTERS
DONT MATCH,
THE VALUE IS
ZERO.

	H	I	S	H
F	0	0	0	0
I	0	1	0	0
S	0	0	2	0
H	0	0	0	3

2. IF THEY DO MATCH,
THIS VALUE IS
VALUE OF TOP-LEFT NEIGHBOR + 1

اینم از شبکه کد این فرمول:

```
if word_a[i] == word_b[j]: ..... The letters match.
    cell[i][j] = cell[i-1][j-1] + 1
else: ..... The letters don't match.
    cell[i][j] = 0
```

این هم یک جدول برای کلمه های vista و hish

	V	I	S	T	A
H	0	0	0	0	0
I	0	1	0	0	0
S	0	0	2	0	0
H	0	0	0	0	0

↑
FINAL ANSWER ↑
NOT THE FINAL ANSWER

یک چیزی که نیاز هست یادتون بمونه اینه که : برای این مسئله ، ممکنه جواب نهایی در آخرین خونه از جدول نباشه! برای مسئله کوله پشتی ، آخرین خونه همیشه جواب نهایی بود. اما برای مسئله طولانی ترین زیر رشته مشترک، جواب نهایی ، بزرگترین عدد در جدول هستش که ممکنه در خونه آخر جدول نباشد.

بیاين برگردیم سر سوال اصلی: کدوم رشته حروف مشترک بیشتری باهم دارن؟ fish و hish یک زیر رشته ۳ حرفی مشترک دارن. hish و vista هم یک زیر رشته ۲ حرفی مشترک دارن. احتمال زیاد الکس میخواسته fish رو تایپ کنه.

بزرگترین زیر دنباله مشترک

فرض کنین الکس به طور اتفاقی fosh رو تایپ میکنه. منظورش کدوم کلمه بوده fish یا fort؟

بیاين با استفاده از فرمول طولانی ترین زیر رشته مشترک باهم دیگه مقایسشون کنیم.

	F	O	S	H		F	O	S	H
F	1	0	0	0		1	0	0	0
O	0	2	0	0		0	0	0	0
R	0	0	0	0		0	0	1	0
T	0	0	0	0		0	0	0	2

vs

طبق جدول هر دو شون یکین: دو حرف! اما fish به fosh نزدیک تره.

$$\begin{matrix} F & O & S & H \\ \downarrow & \downarrow & \downarrow & \\ F & I & S & H \end{matrix} = 3$$

$$\begin{matrix} F & O & S & H \\ \downarrow & \downarrow & & \\ F & O & R & T \end{matrix} = 2$$

درواقع شما داریم طولانی ترین زیر رشته مشترک بین این دو کلمه رو مقایسه میکنین، اما شما نیاز داریم تا طولانی ترین زیر دنباله مشترک رو در بین این دو کلمه مقایسه کنین. تعداد حروف یک دنباله مشترک درین این دو کلمه. خب حالا چطور طولانی ترین زیر دنباله مشترک رو محاسبه میکنین؟

این بخشی از جدول برای کلمه های **fosh** و **fish** هستش.

	F	O	S	H
F	1	1		
I	1			
S		1	2	2
H				

میتوینیں فرمول این جدول رو پیدا کنین؟ طولانی ترین زیر رشته مشترک خیلی شبیه طولانی ترین زیر دنباله مشترک، و فرمول او نهایا هم تقریبا شبیه هم دیگس. اول خودتون سعی کنین حلش کنین، من جواب رو در ادامه بهتون میدم.

طولانی ترین زیر دنباله مشترک - راه حل

جدول در نهایت به این شکل درمیاد:

The figure consists of two tables side-by-side, separated by a vertical line labeled "vs".

Left Table (LCS = 2):

	F	O	S	H
F	1 → 1 → 1 → 1			
O	↓ 1 → 2 → 2 → 2			
R	↓ 1 → 2 → 2 → 2			
T	↓ 1 → 2 → 2 → 2			

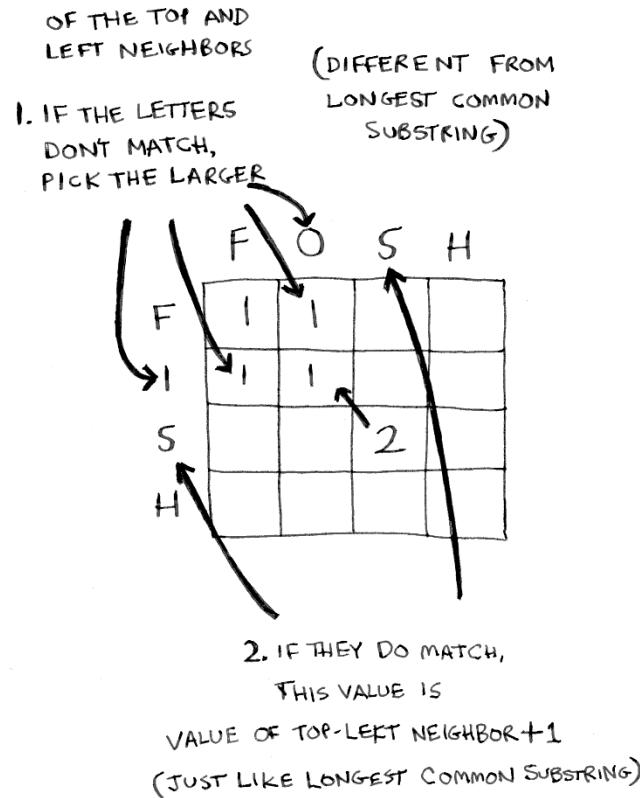
Below this table is the text: LONGEST COMMON SUBSEQUENCE = 2

Right Table (LCS = 3):

	F	O	S	H
F	1 → 1 → 1 → 1			
I	↓ 1 → 1 → 1 → 1			
S	↓ 1 → 1 → 2 → 2			
H	↓ 1 → 1 → 2 → 3			

Below this table is the text: LONGEST COMMON SUBSEQUENCE = 3

اینم فرمولی که من برای پر کردن هر خونه از جدول استفاده کردم:



و این هم شبکه کدش:

```

if word_a[i] == word_b[j]: <----- The letters match.
    cell[i][j] = cell[i-1][j-1] + 1
else: <----- The letters don't match.
    cell[i][j] = max(cell[i-1][j], cell[i][j-1])
  
```

ایول — انجامش دادی! این فصل قطعاً یکی از سختترین فصل‌های این کتابه. حالا این همه گفتیم، اصلاً برنامه نویسی پویا کاربردی هم دارد؟ باید بگم که بله!

- زیست‌شناسان از طولانی ترین زیردباله مشترک برای پیدا کردن شباهت‌ها در رشته‌های DNA استفاده می‌کنند. اونها با این روش می‌توانند بگن که دو گونه از حیوانات چقدر شبیه هم هستند یا دو ویروس چقدر شبیه هم هستند. از طولانی ترین زیردباله مشترک در پیدا کردن درمان برای ام اس استفاده می‌شود.
- تا حالا از diff استفاده کردین (مثل `git diff`) تغییرات بین دو فایل رو به شما میده، و همچنین این دستور از برنامه نویسی پویا برای اینکار استفاده می‌کند.
- ما درباره میزان شباهت بین دو رشته حرف زدیم. فاصله لونشتاین (Levenshtien distance)، میزان شباهت دو رشته رو با استفاده از برنامه نویسی پویا اندازه گیری می‌کنند. فاصله لونشتاین برای همه چی از بررسی غلط املایی تا پیدا کردن فایل‌های آپلود شده دارای حق کپی رایت استفاده می‌شود.
- آیا تاحالا از برنامه‌ای که سرکارش با کلمه‌ها باشه (واژه بندی بکنه) رو داشتین؟ مثل ورد مایکروسافت؟ خب چطوری واژه بندی رو انجام میده که طول هر خط ثابت می‌مونه؟ برنامه نویسی پویا!

تمرین

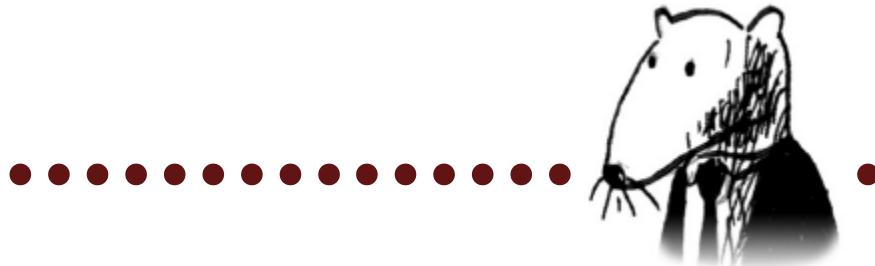
۹.۳ جدولی رو برای محاسبه‌ی طولانی ترین زیر رشته مشترک برای دو کلمه **blue** و **clues** رو بکشید و پرکنید.

خلاصه

- برنامه نویسی پویا وقتی میخواین چیزی که درش محدودیت داریم رو به صورت بهینه پیاده سازی کنیں بسیار کاربردیه.
- زمانی که میتوانیں مسئله رو به زیرمسئله های گستته از هم بشکونیں میتوانیں از برنامه نویسی پویا استفاده کنیں.
- در هر مسئله برنامه نویسی پویا جدول ها دخیل هستند.
- مقادیر داخل هر خونه از جدول غالباً چیز هایی هستند که شما میخواین بهینش کنیں.
- هر خونه از جدول یک زیرمسئله‌س، پس فکر کنیں که چطور میتوانیں یک مسئله رو به چند زیر مسئله تقسیم کنیں.
- هیچ فرمول واحدی برای محاسبه راه حل هایی که از روش برنامه نویسی پویا استفاده میکنن وجود نداره.

k-نزدیک ترین همسایه

(k-nearest neighbors)



در این فصل:

- شما درباره ساخت یک سیستم طبقه بندی که از الگوریتم k-نزدیک ترین همسایه استفاده میکنه یادخواهید گرفت.
- شما درباره استخراج اطلاعات و ویژگی های یک موضوع یادخواهید گرفت.
- شما درباره رگرسیون یادخواهید گرفت: پیش بینی کردن یک عدد، مثل ارزش سهام بازار بورس فردا، یا اینکه یک کاربر چقدر از یک فیلم لذت برده.
- شما درباره کاربرد ها و محدودیت های k-نزدیک ترین همسایه یادخواهید گرفت.

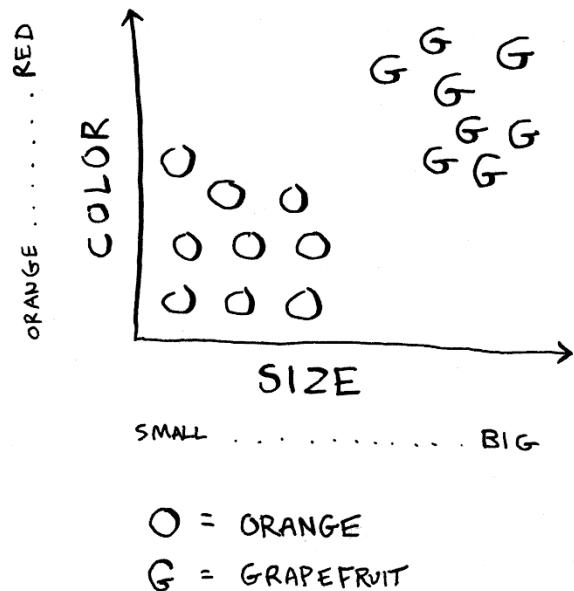


طبقه بندی کردن پرتقال ها و گریپ فروت ها

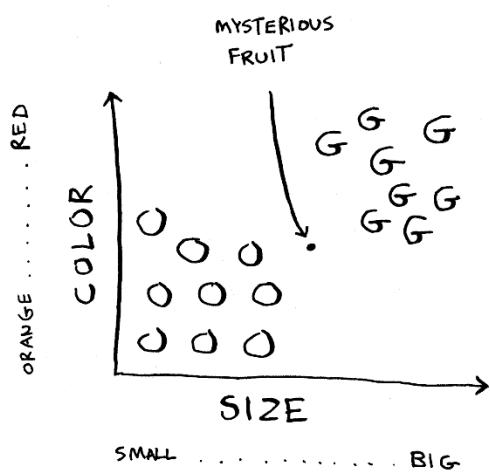
به این میوه نگاه کنیں. بنظرتون این گریپ فروته یا پرتقال؟ خب به طورکلی میدونم که گریپ فروت ها بزرگتر و قرمز تر از پرتقال ها هستن.



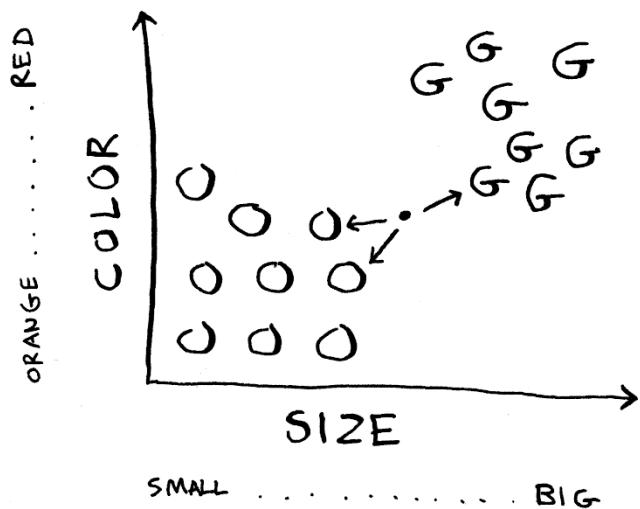
فرايندي که توی فکر من برای تقسيم کردن اين دو میوه ميگذره اينشكليه: يك نمودار توی ذهنم دارم که اينشكليه:



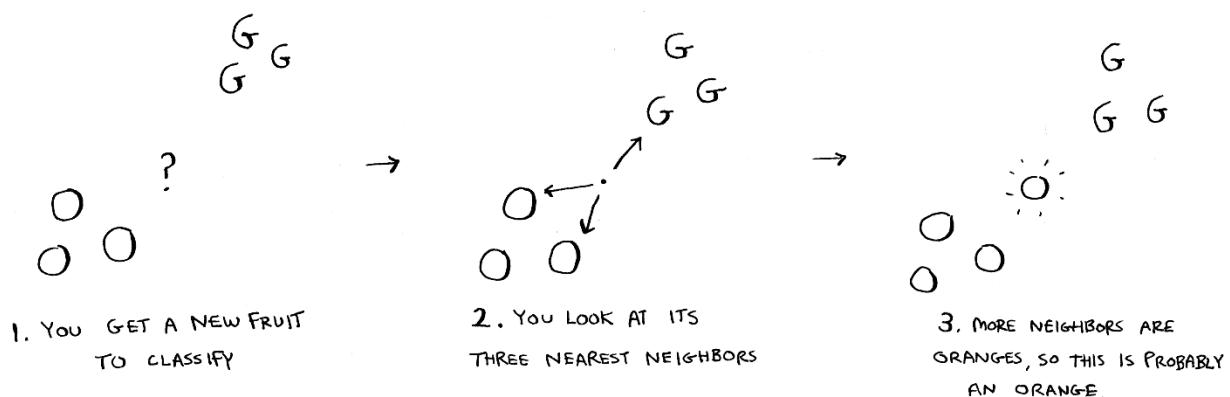
اگر بخواهيم به طور کلي راجب اين موضوع حرف بزنيم به اين صورت است که ، میوه های بزرگتر و قرمزتر گریپ فروت هستند. پس در نتیجه میگوییم اين میوه بزرگ و قرمز است پس احتمال زیاد گریپ فروت است. اما چی اتفاقی میفتند اگر به يك میوه به شکل زير بخورد کنیم ؟



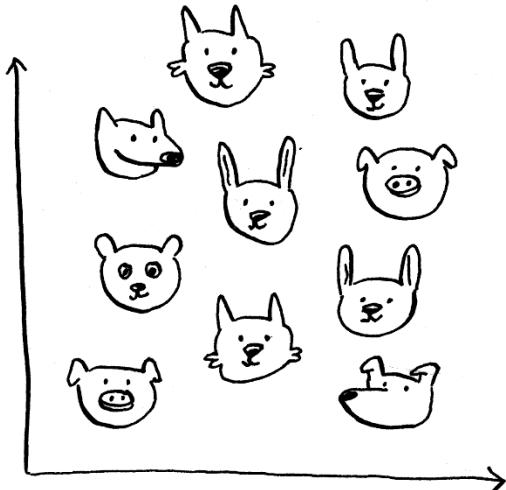
چطور اين میوه رو دسته بندی ميکنин به عبارتی اين میوه در کدام دسته بندی قرار ميگيرد؟ يك راهش اين است که به همسایه های اين میوه در اين نقطه دقت کنیم. پس بیاين به ۳ تا از همسایه های نزدیکش يك نگاهي بیندازیم.



بیشتر این همسایه‌های این نقطه پر تقال هستن تا گریپ فروت. پس به احتمال زیاد این میوه هم پر تقال است. تبریک میگم شما همین الان از الگوریتم k -نزدیک ترین همسایه برای دسته بندی داده‌ها استفاده کردین. مفهوم کلی الگوریتم ساده‌س.



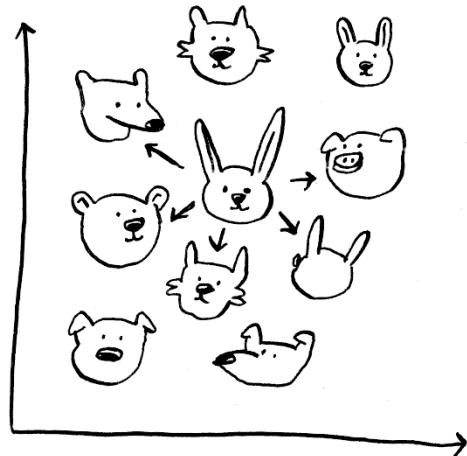
ساخت یک سیستم پیشنهاد دهنده



فرض بگیرین که شما نتفلیکس هستین و در شرکت نتفلیکس مشغولین و میخواین یک سیستم پیشنهاد دهنده (فیلم) برای کاربران تون درست کنین. این مسئله شبیه مسئله گریپ فورت هستش اما در مقیاس و سطح بالاتر.

برای این مسئله شما میتوانین تمام کاربران تون رو روی یک نمودار ترسیم و پیاده سازی کنین.(رو به رو)

این کاربران بر اساس شباهت‌های سلیقه هاشون در انتخاب فیلم در کنار هم دیگر قرار گرفته‌اند. پس درنتیجه کاربرانی که سلیقه‌های مشابهی باهم دارن نزدیک هم دیگر قرار گرفته‌اند. فرض کنین شما میخواهید یک فیلم به پریانکا (Priyanka) پیشنهاد بدھید. برای این کار ۵ کاربر نزدیک به اون رو پیدا کنین.



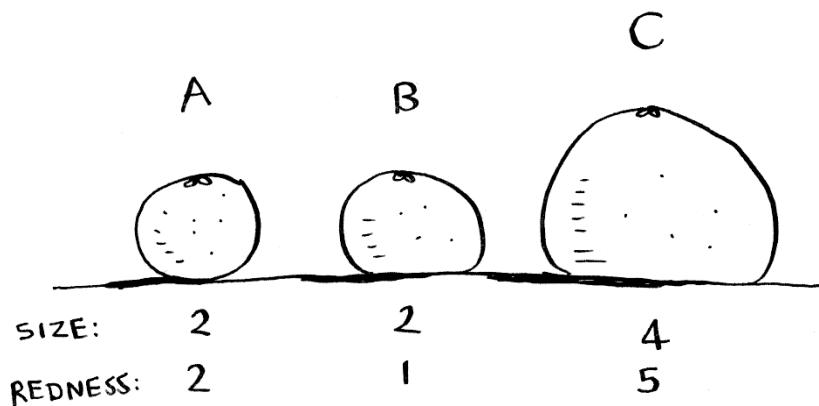
جاستین (Justin)، جی‌سی (JC)، لانس (Lance) و کریس (Chris)، همه سلیقه‌های مشابهی در انتخاب فیلم دارن. پس میتوانیم نتیجه بگیریم که اون‌ها دوست دارن احتمال زیاد پریانکا هم اون رو دوست خواهد داشت! وقتی که این نمودار در اختیارتون باشه ساختن یک سیستم پیشنهاد دهنده ساده میشه. اگر جاستین از یک فیلم خوشش اومد اونو به پریانکا هم پیشنهاد بده.



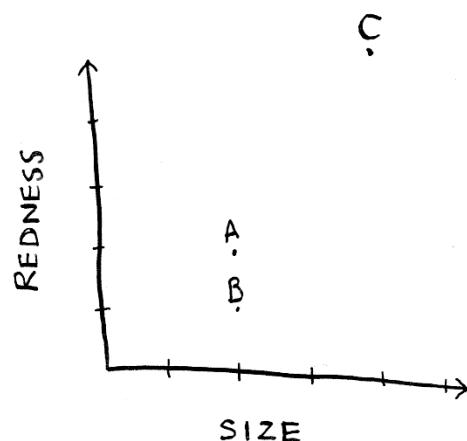
اما هنوز یک بخش بزرگی از کار میلنجه. شما کاربران رو باتوجه به شباهتشون روی نمودار آوردن اما چطوری میفهمیم که دو کاربر شبیه هم دیگه هستن که بخواین کنار هم قرارشون بدین.

استخراج ویژگی‌ها

در مثال گریپ فروت شما میوه ها رو براساس اینکه چقدر سایزشون بزرگه و چقدر قرمز هستن مقایسه و طبقه بندی کردید. پس در این مثال سایز و رنگ ویژگی هایی بودن که شما برای مقایسه ازشون استفاده کردید. حالا فرض کنیم که شما ۳ میوه دارین پس بر این اساس میتوانیم ویژگی هاشون رو استخراج کنیم.



حالا میتوانیم با توجه به این اطلاعات نمودار رو برای این سه میوه رسم کنیم.



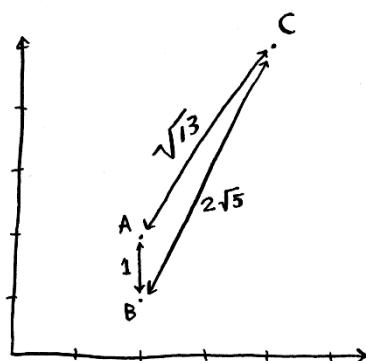
با توجه به نمودار میتوان گفت که میوه های A و B شبیه به هم هستند. بیاین فاصله بیشتر را اندازه بگیریم تا ببینیم چقدر به هم نزدیک هستند. برای پیدا کردن فاصله بین دو نقطه از فرمول فیثاغورث استفاده میکنیم.

$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

برای مثال، این فاصله بین نقطه A و B :

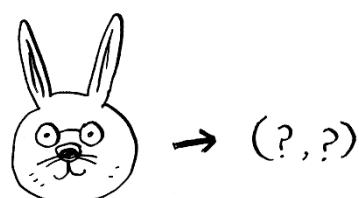
$$\begin{aligned}
 & \sqrt{(2-2)^2 + (2-1)^2} \\
 & = \sqrt{0+1} \\
 & = \sqrt{1} \\
 & = 1
 \end{aligned}$$

فاصله بین A و B برابر 1 است. شما میتوانید بقیه فواصل رو هم پیدا کنید.



این فرمول که برای پیدا کردن فواصل ازش استفاده کردیم، چیزی رو که با توجه و تنها با مشاهده از روی نمودار گفتیم رو تصدیق میکنه: میوه A و B شبیه هم هستند.

حالا به جای این، فرض کنیم که شما در حال مقایسه کاربران نتفلیکس با یکدیگر هستید. شما به یک راه نیاز دارین تا بتونین کاربران رو روی نمودار بیارین. پس درنتیجه شما نیاز دارین تا هر کاربر رو به یک مختصات روی نمودار تبدیل کنین، دقیقا کاری که برای میوه ها انجام دادیم.



وقتی که شما تو نستین کاربران رو روی نمودار بیارین ، اون موقعس که میتوینین فاصله بینشون رو اندازه گیری کنین. این روشیه که شما میتوینین کاربران رو تبدیل به مجموعه‌ای از اعداد و درنهایت مختصات تبدیل کنین.

			
PRIYANKA		JUSTIN	MORPHEUS
COMEDY	3	4	2
ACTION	4	3	5
DRAMA	4	5	1
HORROR	1	1	3
ROMANCE	4	5	1

پریانکا و جاستین جفت‌شون ژانر رمانیک رو دوست دارن و از ژانر ترسناک منتفرون. مورفس به فیلم های اکشن علاقه داره و از ژانر رمانیک منتفره (از زمان هایی هم که یک فیلم اکشن خوب توسط صحنه رمانیک خراب میشه هم بدش میاد). یادتون توی مقایسه پرتقال با گریپ فروت هر میوه با مجموعه‌ای متتشکل از ۲ عدد نشون داده میشد؟ خب اینجا هر کاربر با مجموعه‌ای متتشکل از ۵ عدد نمایش داده میشه.

$$\text{apple} \rightarrow (2, 2)$$

$$\text{rabbit} \rightarrow (3, 4, 4, 1, 4)$$

خب با این اوصاف اگر یک ریاضیدان اینجا باشه میگه که ، این دفعه بجای محاسبه فاصله در ۲ بعد، در ۵ بعد فاصله رو محاسبه میکنین. اما فرمول به همون شکلی که بود باقی میمونه.

$$\sqrt{(a_1 - a_2)^2 + (b_1 - b_2)^2 + (c_1 - c_2)^2 + (d_1 - d_2)^2 + (e_1 - e_2)^2}$$

فقط به جای اینکه این فرمول شامل مجموعه دو عددی باشد شامل یک مجموعه پنج عددیه.

فرمولی که برای بدست آوردن فاصله داریم انعطاف پذیره: میتوانستیں یک میلیون عدد داشته باشیدن و این فرمول برای بدست آوردن فاصله همچنان ثابت میموند. شاید براتون سوال شده باشد که "فاصله، چه معنی میده زمانی که مجموعه پنج عددی داریم؟" خب فاصله به شما میگه که چقدر این مجموعه از اعداد بهم شباهت دارند.

$$\begin{aligned} & \sqrt{(3-4)^2 + (4-3)^2 + (4-5)^2 + (1-1)^2 + (4-5)^2} \\ &= \sqrt{1+1+1+0+1} \\ &= \sqrt{4} \\ &= 2 \end{aligned}$$

این فاصله بین پریانکا و جاستین هستش.

سلایق پریانکا و جاستین تقریباً شبیه هم هستند. چه تفاوتی بین پریانکا و مورفس وجود دارد؟ قبل از اینکه به خوندن ادامه بدین فاصله بینشون رو محاسبه کنیں.

درست بدستش آوردین؟ پریانکا و موفس به اندازه ۲۴ باهم فاصله دارند. فاصله بیشتر میگه که سلیقه پریانکا بیشتر شبیه جاستین هستش تا مورفس.

عالی شد! الان پیشنهاد فیلم به پریانکا راحت شد: اگر جاستین از فیلیمی خوشش اوهد، اون رو به پریانکا هم پیشنهاد بده، و برعکس. شما همین الان یک سیستم پیشنهاد دهنده فیلم ساختین!

اگر کاربر نتفلیکس باشیدن میبینید که نتفلیکس مدام به شما میگه که "لطفاً به فیلم‌های بیشتری امتیاز بدین، هرچقدر شما بیشتر به فیلم‌ها امتیاز بدین، سیستم پیشنهاد دهنده فیلم‌های بهتر عمل میکنه و فیلم‌های بهتری بیشتر پیشنهاد میده." خب دیگه الان دلیل و چرایی این موضوع رو میدونیدن. هر چقدر شما بیشتر امتیاز بدین، نتفلیکس دقیق‌تر میتوانه تشخیص بده که سلیقه شما شبیه کدام کاربران دیگرش است.

تمرین ها

۱۰.۱ در مثال نتفلیکس، شما فاصله بین دو کاربر را با استفاده از فرمول فاصله بدست آوردین. اما همه کاربران به یک شکل به فیلم‌ها امتیاز نمیدهند. به عنوان مثال فرض کنید که شما دو کاربر داریدن به اسم‌های یوگی و پینکی، که سلیقه یکسانی در انتخاب فیلم دارند. اما یوگی از هر فیلمی که خوشش بیاد بشه ۵ میده، درحالی که پینکی خیلی

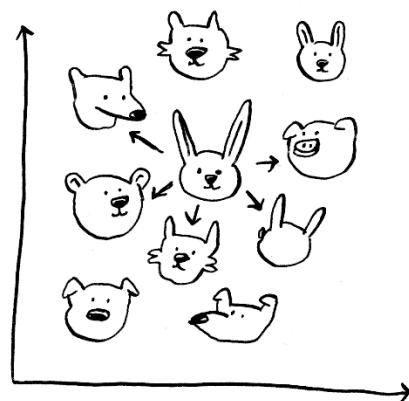
سخت‌گیره و امتیاز ۵ رو فقط به بهترین و تاپترين فیلم‌ها میده. سلیقه این دو کاربر خیلی خوب بهم میخوره اما براساس الگوریتم فاصله (که همون فرموله باشه)، او نا باهم همسایه نمیشن. در این مورد چطور استراتژی امتیاز دادن آنها رو در محاسبات خودتون لحاظ میکنین که به چیزی که میخواین برسین؟

۱۰.۲ فرض کنین که نتفلیکس گروهی از اینفلوئنسرها رو به عنوان نامزد قرار میده (نامزد چی؟ الان بهتون میگم). فرض کنین کوئنتین تارانتینو و وس اندرسون اینفلوئنسرها بی در نتفلیکس هستند، خب به طبع رای و امتیاز اونها به یک فیلم بیشتر از کاربران عادی محاسبه میشه و اعتبار بیشتری داره. خب در این حالت شما چطور سیستم پیشنهاد دهنده رو با توجه به این موضوع که امتیاز این اینفلوئنسرها تاثیر بیشتر در امتیاز دهی به فیلم دارند، تغییر میدهید؟

رگرسیون

فرض کنین که شما میخواهین کاری بیشتر از پیشنهاد دادن فیلم انجام بدین: شما میخواین حدس بزنین که پریانکا چطور امتیاز میدهد (امتیاز پریانکا به یک فیلم را پیش بینی کنید). برای این کار پنج نفری که سلیقه نزدیکی به اون رو دارن رو انتخاب کنین.

راستی من مدام درباره انتخاب پنج فرد نزدیک حرف میزنم، هیچ چیز خاص و بخصوصی راجب این عدد ۵ وجود نداره: شما میتوانین ۲ فرد نزدیک رو انتخاب کنین یا ۱۰ تا فرد نزدیک رو با حتی ۱۰۰۰۰. به همین دلیله که اسم این الگوریتم k -نزدیکترین همسایه است نه ۵-نزدیکترین همسایه!



JUSTIN :	5	فرض کنین میخواین حدس بزنین که چطور به فیلم Pitch Perfect رای داده میشه. خب
JC :	4	، جاستین، جی‌سی، جویی، لانس، و کریس چطور به این فیلم رای و امتیاز دادن؟
JOEY :	4	
LANCE :	5	
CHRIS :	3	

شما میتوانید میانگین امتیازان اونا رو بگیرین که میشه ۴.۲ ستاره. به این کار رگرسیون گفته میشه. اینها دو تا از کارهای پایه ای هست که شما با استفاده از طبقه بندی KNN و رگرسیون انجام خواهید داد:

- طبقه بندی = دسته بندی کردن داخل یک گروه
- رگرسیون = پیش بینی کردن جواب (مثل پیش بینی کردن عدد)

رگرسیون خیلی کاربردیه. فرض کنین که شما یک نانوایی کوچک در برکلی (یک شهر در کالیفرنیا) دارید و هر روز نان های تازه میپزید. شما سعی میکنین که پیش بینی کنید که برای امروز چقدر نان باید پخته شود. در اینجا شما مجموعه ای از ویژگی ها در اختیار دارید.

- آب و هوا در مقیاس ۱ تا ۵ (۱ = بد، ۵ = عالی)
- آخر هفته یا تعطیلات؟ (۱ = برای تعطیلات یا آخر هفته، صفر برای روز های عادی)
- آیا بازیی در جریان؟ (اگر آره ۱، اگر نه صفر)

و همچنین شما میدونین که چند نان در روز های گذشته برای مجموعه متفاوت فروختین.

$$\boxed{A.} (5, 1, \emptyset) = 3\phi\phi \quad \boxed{B.} (3, 1, 1) = 225 \text{ LOAVES}$$

$$\boxed{C.} (1, 1, \emptyset) = 75 \text{ LOAVES} \quad \boxed{D.} (4, \emptyset, 1) = 2\phi\phi$$

$$\boxed{E.} (4, \emptyset, \emptyset) = 15\phi \text{ LOAVES} \quad \boxed{F.} (2, \emptyset, \emptyset) = 5\phi \text{ LOAVES}$$

خب حالا امروز یکی از روزهای آخر هفته با هوای خوب هستش. با توجه به اطلاعاتی که این بالا مشاهده کردیم، امروز چند قرص نان میتوانید بفروشید؟ بیاین از KNN استفاده کنیم که در اینجا K را برابر ۴ قرار میدیم. اول، چهار همسایه نزدیک این نقطه رو پیدا کنیم.

$$(4, 1, \emptyset) = ?$$

اینها فوائلی هستند که در اختیار داریم. A و B و C و D نزدیک ترین همسایه ها هستند.

- A. 1 ←
- B. 2 ←
- C. 9
- D. 2 ←
- E. 1 ←
- F. 5

وقتی که یک میانگین از تعداد نان های فروخته شده در آن روز ها بگیرین به عدد ۲۱۸.۷۵ میرسین. این تعداد قرص نانی است که شما باید برای امروز بپزین.

تشابه کسینوسی

تا اینجای کار، شما از فرمول فاصله برای مقایسه فاصله بین دو کاربر استفاده میکردیدن. اما آیا این بهترین فرمول برای محاسبه فاصله است که میتوانیم استفاده کنیم؟ یکی از رایجترین فرمول هایی که در عمل از آن استفاده میشود تشابه کسینوسی. فرض کنیم که سلیقه دو کاربر در انتخاب فیلم شبیه هم هست اما یکی از آنها بیشتر در امتیاز دادن محافظه کار و سختگیر است. هردو فیلم **Manmohan Desai's Amar Akbar Anthony** رو دوست داشتند. پاول به فیلم ۵ ستاره داده اما روان (**Rowan**) به اون فیلم ۴ ستاره داده. اگر از همان فرمول فاصله استفاده کنیم ممکنه که این دو کاربر همسایه همدیگر نشوند با وجود اینکه سلیقه مشابهی دارند.

تشابه کسینوسی اندازه فاصله بین دو بردار رو محاسبه نمیکنه. بجاش میاد و زاویه بین دو بردار رو مقایسه میکنه. این فرمول در برخورد با چنین مواردی بهتر عمل میکنه. تشابه کسینوسی خارج از بحث این کتابه، اما اگر از KNN استفاده میکنین یک نگاه بهش بندازین!

انتخاب ویژگی های خوب و مناسب

برای اینکه بتونین پیشنهادات فیلم رو پیدا کنین و به کاربران نشان دهید، برای این کار دسته‌بندی امتیازات کاربران رو برای فیلم ها در اختیار دارید. خب اگر چه اتفاقی میفتاد اگر شما امتیاز کاربران به عکس گربه ها رو بجای فیلم ها در اختیار داشتین؟ خب نتیجش میشد اینکه کاربرانی پیدا میکردید که به اون عکس ها به طور مشابهی امتیاز دادن. این به احتمال زیاد میشد بدترین موتور پیشنهاد دهنده فیلم ویژگی هایی که میتوانیم از عکس گربه ها استخراج کنیم اطلاعات زیادی درباره سلیقه کاربران در فیلم به ما نمیدهد.

یا فرض کنین که شما از کاربرانتون میخواین که به فیلم ها امتیاز بدن که شما بتونین بھشون فیلم های دیگه ای پیشنهاد بدین اما فقط از اونا میخواین که به فیلم های داستان اسباب بازی ها ، داستان اسباب بازی های ۲ و داستان اسباب بازی های ۳ امتیاز بدن. باز هم دریافت اطلاعات از این روش چیز زیادی درمورد سلیقه کاربران در اختیار ما قرار نمیده.

زمانی که دارین با KNN کار میکنین، مهمه که ویژگی های درست رو برای مقایسه با یکدیگر انتخاب کنین. انتخاب درست ویژگی ها یعنی:

- ویژگی هایی که مستقیما با فیلم هایی که میخواهید معرفی کنین به هم مرتبط هستند.
- ویژگی هایی که شامل تعصب یا محدودیت نباشند.(به عنوان مثال اگر شما فقط از کاربرانتون بخواین که به فیلم های کمدی امتیاز بدن اطلاعاتی درباره اینکه آیا اونا از فیلم های اکشن هم خوشنون میاد یا نه بهتون نمیده)

آیا فکر میکنین که صرف امتیاز دادن راه خوبی برای پیشنهاد دادن فیلم هست؟ شاید من امتیاز بیشتری به فیلم House Hunter نسبت به The wire داده باشم اما درواقع زمان بیشتری برای نگاه کردن فیلم گذاشت. خب حالا شما چطور سیستم پیشنهاد دهنده نتفلیکس رو (بر این اساس) توسعه میدین؟

بیاین برگردیم به مثال نانوایی: میتوనین به ۲ ویژگی خوب و دو ویژگی بدی که میتوونیستیم برای نانوایی انتخاب کنیم رو نام ببرین؟ شاید نیاز باشه که نان های بیشتری بعد از تبلیغ کردن رو کاغذ در شهر ، بپزین. یا شاید اصلا نیازه که نون بیشتری در روز دوشنبه بپزین.

راستش یک جواب درست برای انتخاب کردن ویژگی خوب و اینکه چطور یک ویژگی خوب رو استخراج کنیم وجود نداره. شما باید به تمام چیزی هایی که میخواین در نظر بگیرین فک کنین.

تمرین

۱۰.۳ نتفلیکس میلیون ها کاربر داره.در مثالی که بالاتر زدیم تنها به ۵ تا از همسایه های یک نقطه برای ساخت سیستم پیشنهاد دهنده استفاده کردیم؟ آیا این مقدار خیلی کمه؟ یا آیا خیلی زیاده؟

مقدمه‌ای بر یادگیری ماشین



KNN یک الگوریتم بسیار کاربردی و دروازه ورود شما به دنیای جادویی یادگیری ماشین هستش! مبحث یادگیری ماشین تماماً مربوط به باهوش‌تر کردن کامپیوتر تون هستش. شما قبل از این یک مثال از یادگیری ماشین رو مشاهده کردید: ساخت سیتم پیشنهاد دهنده. بیاین به چننا مثال دیگه نگاه بندازیم.

OCR

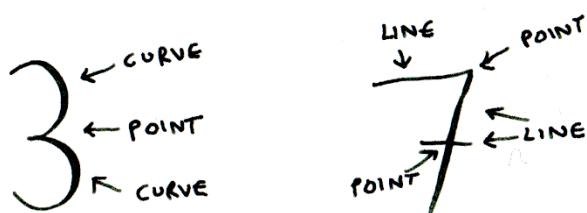
اختصار OCR (تشخیص کاراکتر نوری). یعنی شما میتوانین یک عکس از یک نوشته بگیرید و کامپیوتر شما به طور اتوماتیک اون نوشته را برایتان میخواند. گوگل از OCR برای دیجیتالی کردن کتاب‌ها استفاده میکند. OCR چطور کار میکند؟ برای مثال، این عدد را در نظر بگیرید.

۷

چطوری میتوانید به طور اتوماتیک تشخیص بدین که این چه عددی ست؟ میتوانید از KNN برای این کار استفاده کنین:

۱. باید بین سراغ عکس‌های زیادی از اعداد و ویژگی هر کدام از اعداد را استخراج کنید.
۲. هنگامی که عکس جدیدی دریافت میکنید، ویژگی‌های اون عکس رو استخراج کنید و ببینید که همسایه‌های نزدیک به اون کدام هستند.

این مسئله مانند مسئله پرتفوال و گریپ فروت هستش. به طور کلی الگوریتم OCR خطها نقاط و منحنی‌های موجود در شکل را بررسی میکند.



بعد از اون، وقتی که شما کاراکتری جدیدی دریافت میکنید، میتوانید همان ویژگی‌ها را از اون استخراج کنید.

استخراج ویژگی ها در OCR خیلی پیچیده تر از مثال میوه‌ای است که بالاتر زدیم. اما این مهمه که بدانید که حتی تکنولوژی های پیچیده هم بر پایه ایده‌های ساده مانند KNN ساخته می‌شوند. شما میتوانید از همین ایده برای تشخیص صدا و تشخیص چهره نیز استفاده کنید. هنگامی که شما عکسی در فیسبوک آپلود می‌کنید، گاهی فیسبوک انقدر باهوش هست که افراد در اون تصویر رو تشخیص دهد و آنها را تگ کند. این همان یادگیری ماشین در عمل است!

به گام اول OCR، جایی که شما به سراغ عکس‌هایی از اعداد و استخراج ویژگی های آنها می‌روید، آموزش (training) گفته می‌شود. بیشتر الگوریتم های یادگیری ماشین گام آموزش را دارند: قبل از اینکه کامپیوتر شما بتونه اون تسك رو انجام بده باید آموزش ببینه. مثال بعدی شامل فیلتر کردن پیام های اسپم هستش و شامل گام آموزش می‌شود.

ساخت یک فیلتر اسپم

فیلتر اسپم از یک الگوریتم ساده دیگر استفاده می‌کند که دسته‌بند بیز ساده (Naive Bayes classifier) نام دارد. اول، شما با استفاده از مقداری داده دسته‌بند بیز ساده تون را آموزش میدین.

SUBJECT	SPAM?
"RESET YOUR PASSWORD"	NOT SPAM
"YOU HAVE WON 1 MILLION DOLLARS"	SPAM
"SEND ME YOUR PASSWORD"	SPAM
"NIGERIAN PRINCE SENDS YOU 10 MILLION DOLLARS"	SPAM
"HAPPY BIRTHDAY"	NOT SPAM

فرض کنین شما ایمیلی با این عنوان دریافت می‌کنین که "همین الان یک میلیون دلار خود را بدست آورید!" آیا این اسپم؟ میتوانید این جمله رو به کلماتش تقسیم کنید. سپس، برای هر کلمه بررسی کنید که چقدر احتمالش هست که در یک ایمیل اسپم وجود داشته باشه. برای مثال کلمه "میلیون" تنها در ایمیل های اسپم پیدا شد. بیز ساده احتمال اینکه چقدر یک چیز میتوانه اسپم باشه رو بررسی می‌کنه. این الگوریتم کاربردی مشابه KNN دارد.

به عنوان مثال میتوانید از بیز ساده برای طبقه‌بندی کردن میوه‌ها استفاده کنید: شما میوه دارید که بزرگ و قرمزه. چقدر احتمال داره که این میوه گریپفروت باشه؟ این یک الگوریتم ساده دیگر است که نسبتاً موثر است. ما عاشق این نوع الگوریتم هاییم!



پیش بینی کردن بازار سهام

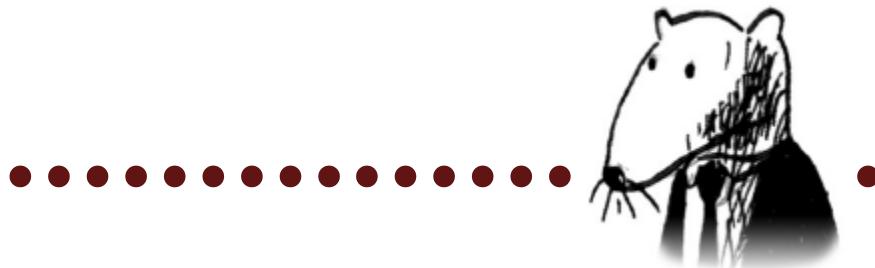
این چیزیه که انجام دادنش با یادگیری ماشین سخته: پیش بینی واقعی اینکه آیا بازار سهام روبه بالا یا پایین خواهد رفت. چطور یک ویژگی خوب برای استخراج در بازار سهام انتخاب میکنید؟ فرض کنین که شما میگین اگر بازار سهام دیروز رو به بالا رفت امروز هم روبه بالا میره. آیا این ویژگی خوبیه؟ یا فرض کنین که میگین بازار سهام همیشه در ماه می رو به پایین میره. آیا کار خواهد کرد؟ هیچ راه تضمینی وجود نداره که از اعداد و ارقام گذشته برای پیش بینی عملکرد آینده استفاده کنیم. پیش بینی آینده کار سختیه و تقریبا وقتی متغیر های زیادی دخیل هستند کار غیر ممکنیه.

خلاصه

امیدوارم این فصل ایده کلی کاراهای مختلفی که میتوانیم با الگوریتم KNN و یادگیری ماشین انجام دهیم را به شما بدهد! یادگیری ماشین یک رشته خیلی جذابه که میتوانید به انتخاب خودتون توش عمیق بشید.

- KNN برای دسته‌بندی و رگرسیون استفاده میشود و شامل بررسی k تعداد همسایه نزدیک نقطه مورد نظر میشود.
- دسته‌بندی = طبقه‌بندی در یک گروه
- رگرسیون = پیش بینی کردن یک جواب (مانند عدد)
- استخراج ویژگی ها یعنی تبدیل یک آیتم (مثل میوه یا یک کاربر) به یک لیست و مجموعه‌ای از اعداد که بتوانیم باهم مقایسه‌شون کنیم.
- انتخاب یک ویژگی خوب، یک بخش مهم از یک الگوریتم KNN موفقیت آمیزه.

بعد از این کجا بریم ادامه راه...

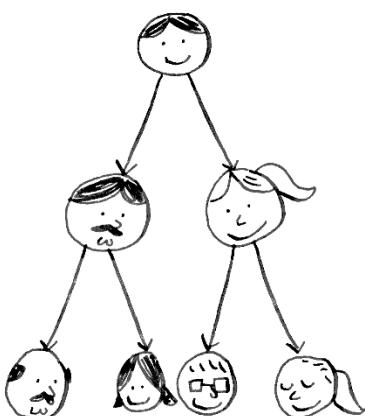


در این فصل:

- یک مرور مختصر از ۱۰ الگوریتمی که در این کتاب پوشش داده نشده و چرایی کاربرد آونها در اختیارتون قرارمیگیره.
- نکاتی درباره ادامه راه شما براساس آنچه بهش علاقه دارید دراختیارتون قرار میگرد.



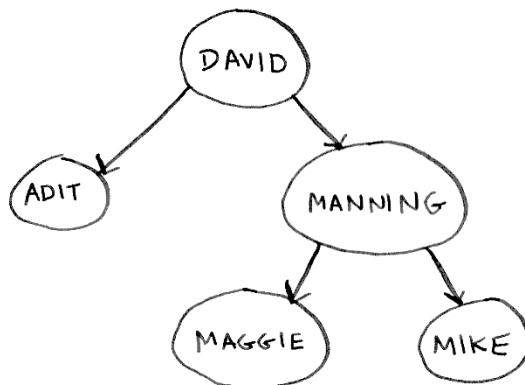
درختها



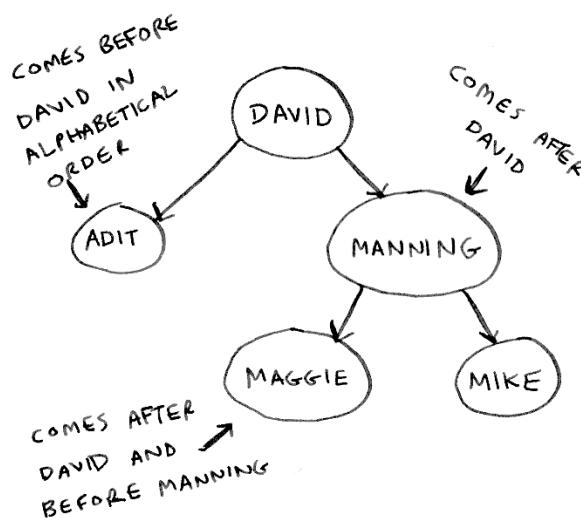
بیاین برگردیم سراغ مثال جستوجوی دودویی یا همان باینری سرج. هنگامی که کاربری میخواهد وارد فیس بوک شود، فیس بوک باید بین یک آرایه بزرگ بررسی کند که آیا این نام کاربری در آن وجود دارد یا خیر. گفتیم که سریع ترین راه برای جستجو کردن در آرایه اینه که باینری سرج را روی آن پیاده کنیم. اما مشکلی وجود داره: هر بار که یک کاربر در سایت ثبت نام میکند، شما نام کاربری آن را در آرایه ذخیره میکنید، سپس باید یکبار دیگر آرایه را مرتب

یا به عبارتی **resort** کنید چراکه باینری سرج تنها روی آرایه های مرتب شده قابل اجراست. بهتر نبود اگر میتوانستیم نام کاربری را مستقیم در همان جا در اسلات درستش در آرایه قرار دهیم، که اینطوری مجبور نباشیم **binary** هر بار آرایه را **resort** (مرتب) کنیم؟ این همان ایده پشت ساختار داده جستوجوی دودویی درختی (search tree) است.

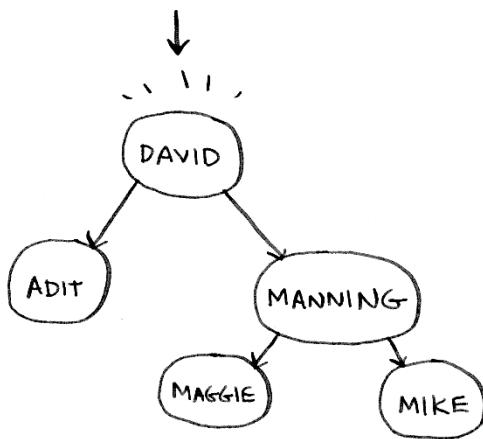
باینری سرج تری یا جستوجوی دودویی درختی مانند شکل زیر است.



بر ازای هر گره، گره های سمت چپ کوچکتر از ارزش این گره است و گره های سمت راست آن بزرگتر از ارزش این گره است. (ممکنه منظور مقدار اعداد باشه یا ترتیب حروف الفبا مثلا "الف" عقب تر از "ت" قرار داده پس الف در سمت چپ حرف "ت" میفته).

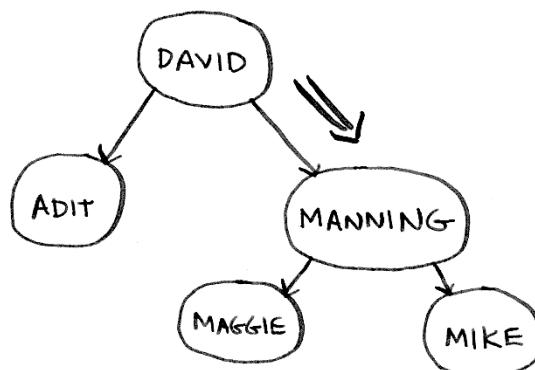


فرض کنین شما دنبال **Maggie** میگردید. شما از ریشه گره شروع میکنین(گره اصلی).

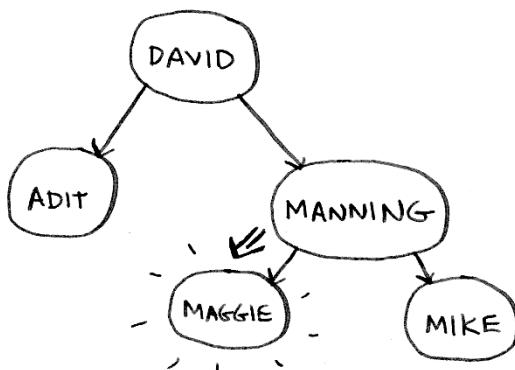


بعد از کلمه David قرار میگیره (طبق حروف الفبا).

پس باید بریم سمت راست این درخت.



قبل از Manning قرار میگیره ، پس باید بریم سمت چپ.

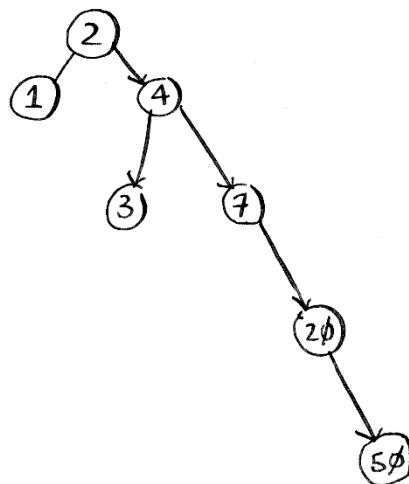


با این کار شما Maggie رو پیدا کردید! تقریبا مثل اجرا کردن باینری سرج هستش. جستجو برای یک عنصر در یک جستجوی دودویی درختی به طور میانگین به اندازه $O(\log n)$ طول میکشه و در بدترین حالت به اندازه $O(n)$ طول خواهد کشید. جستجو در بین یک آرایه مرتب شده به اندازه $O(\log n)$ در بدترین حالت طول خواهد

کشید در نتیجه ممکنه فکر کنین که یک آرایه مرتب شده خیلی بهتره (برای سرچ کردن و این داستانا). اما جستوجوی دودویی درختی در حذف آیتم و اضافه کردن آن به طور میانگین خیلی سریعتره.

ARRAY	BINARY SEARCH TREE
SEARCH $O(n)$	$O(\log n)$
INSERT $O(n)$	$O(\log n)$
DELETE $O(n)$	$O(\log n)$

البته جستوجوی دودویی درختی یک سری جنبه ها منفی هم داره: یکی اینکه، شما نمتوانین دسترسی رندوم و تصادفی داشته باشین. مثلا نمتوانین بگین که "عنصر پنجم این درخت رو به من بده.". اون زمان عملکردی که در بالا دیدم هم براساس شرایط میانگین است و به متعادل بودن درخت بستگی داره. فرض کنین که شما یک درخت غیر متعادل مانند درختی که در شکل زیر نمایش داده شده در اختیار دارید.



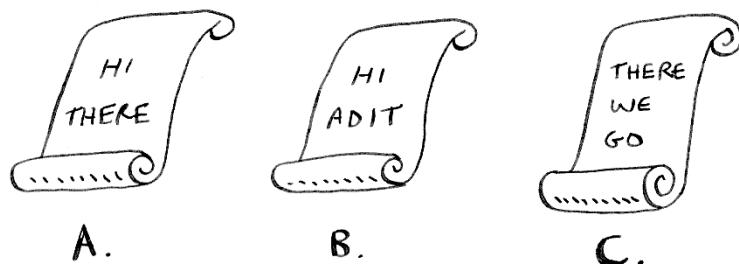
میبینین که چطور داره به سمت راست متمایل میشه؟ این درخت عملکرد خوبی نداره چراکه اصلاً متعادل و بالанс نیست. یک سری جستجو های دودویی خاص وجود دارند که خودشون خودشونو بالانس میکنن. یک مثالش درخت قرمز-سیاه (red-black tree) است.

در چه زمانی جستجوی دودویی درختی استفاده میشے؟ **B-trees**، یک نوع خاص از درخت دودویی که معمولاً از آن در ذخیره اطلاعات در پایگاه داده یا دیتابیس استفاده میشے. اگر به دیتابیس یا به ساختار داده های پیشرفته تر علاقه دارین، اینها رو هم بررسی کنین:

- **B-trees**
- درخت قرمز-سیاه (**Red-black trees**)
- هیپس (**Heaps**)
- ساختار درختی درهم (**splay trees**)

ایندکس معکوس یا شاخص معکوس (**Inverted indexes**)

در این قسمت نحوه کار موتور ها جستجو به صورت نسخه ساده شده را داریم. فرض کنین که ما این سه صفحه وب را با این محتوا های ساده در اختیار داریم.



HI	A, B
THERE	A, C
ADIT	B
WE	C
GO	C

بیاین یک جدول هش برای محتواهایی که داریم درست کنیم. کلید های این جدول هش کلمات هستند و مقادیر آنها به شما میگوید که این کلامات در کدام صفحات استفاده شده اند. حالا فرض کنین که کاربر کلمه **hi** رو سرچ میکند. بیاین ببینیم که کلمه **hi** توی چه صفحاتی پیداش میشه.



آها: این کلمه توی صفحات **A** و **B** وجود داره. بیاین این دو صفحه رو به عنوان نتیجه به کاربر نشون بدیم. یا فرض کنین که کاربر کلمه **there** رو سرچ میکنه. خب شما میدونید که این کلمه داخل صفحه های **A** و **C** وجود داره. خیلی راحته، نه؟ این یک ساختار داده کاربردیه: یک هش که کلمات رو به جایی که در اونجا ظاهر میشن وصل میکنه. به این ساختار داده ایندکس معکوس (**inverted index**) گفته میشے. و معمولاً از اون در ساخت موتور های جستجو استفاده میشود. اگر شما به سرچ کردن علاقه دارین این مبحث جای خوبی برای شروع هستش.

تبدیل فوریه (The Fourier transform)

تبدیل فوریه از آن دست الگوریتم های خیلی خاص، زیبا و عالی و با میلیون ها کاربرد. بهترین توضیح و قیاس برای مبحث تبدیل فوریه رو میتوانیم توی Better Explained (که یک وب سایت عالیه که ریاضی رو به صورت ساده توضیح میده) پیدا کنیم: شما میتوانیم یک اسموتویی به تبدیل فوریه بدین و بہتون میگه مواد تشکیل دهنده اون اسموتویی چیه). Kalid, "An Interactive Guide to the Fourier Transform," Better Explained, <http://mng.bx/84X>.

یا بیاین یک طور دیگه بهش نگاه کنیم شما یک آهنگ بهش میدین و تبدیل فوریه میتونه اونو به فرکانس های جدا برای شما تقسیمش کنه.

به نظر میاد که این ایده ساده خیلی کاربرد ها دارد. به عنوان مثال اگر شما میتوانیم آهنگ هارو به فرکانس های اون تقسیم کنید، میتوانیم اون فرکانسی که خودتون میخواین رو تقویت کنیم شما میتوانیم باس آهنگ رو تقویت و صدای زیرشو از بین ببریم. تبدیل فوریه یک ابزار عالی برای تحلیل سیگنال هاست. همچنین شما میتوانیم از اون برای فشرده سازی یک آهنگ استفاده کنیم. در گام اول با استفاده از این تبدیل فایل صوتی رو به نت هایی که از اون ساخته شده تقسیم میکنیم. این تبدیل به شما میگه هر نوت چقدر روی کل آهنگ تاثیر دارد. در نتیجه شما میتوانیم نت هایی که مهم نیستند رو پیدا و حذفشون کنیم. این روش کار فرمت MP3 هستش.

موزیک تنها نوع از سیگنال های دیجیتالی نیست. فرمت JPEG یک فرمت فشرده شده دیگر است، و به همین صورت کار میکنه. افراد از تبدیل فوریه برای پیش‌بینی زلزله‌هایی که در آینده ممکنه رخ بده یا تحلیل DNA استفاده میکنند.

میتوانیم از تبدیل فوریه برای ساخت اپ هایی مانند Shazam که آهنگی که در حال پخش هست رو حدس میزنیم استفاده کنیم. تبدیل فوریه استفاده های بسیاری دارد. احتمال اینکه برین سراغ این خیلی بالاست!

الگوریتم های موازی

سه موضوع بعدی درباره مقیاس پذیری و کار با داده و اطلاعات زیاده. در قدیم کامپیوتر ها مدام سریع و سریع تر میشدند. شما اگر میخواستین که الگوریتمتون سریع تر کار کنه، میتوانستین چند ماه صبر کنید و کامپیوتر ها خودشون سریع تر میشدند و الگوریتم شما سریع تر اجرا میشد. اما ما دیگه آخر های آن دوره زمانی هستیم. بجاش لپ تاپ ها و کامپیوتر ها با چند هسته عرضه میشنود. برای اینکه الگوریتمتون رو سریع تر اجرا کنیم نیازه که تغییرشون بدین تا به صورت موازی و همزمان روی همه هسته اجرا بشن!

یک مثال ساده میزند. کاری که شما میتوانید در بهترین شرایط با یک الگوریتم مرتب سازی اجرا کنید تقریباً به اندازه $O(n \log n)$ هستش. خیلی واضحه که شما نمیتوانید یک آرایه رو در زمان $O(n)$ مرتب کنید (مگر اینکه به از اون الگوریتم به صورت موازی استفاده کنید). یک ورژن موازی از کوییک سورت (مرتب سازی سریع) وجود دارد که یک آرایه رو در زمان $O(n)$ مرتب میکنه.

طراحی الگوریتم های موازی سخته. و همچنین مطمئن شدن از عملکرد درست اون الگوریتم و اینکه بفهمیم چقدر افزایش سرعت رو خواهیم داشت هم سخته. اما یک چیز مسلمه اونم اینکه مقدار زمانی که سود میکنیم به صورت خطی نیست. پس اگر شما در لپ تاپتون دو هسته بجاوی یک هسته دارین، کاملاً این معنی رو نمیده که الگوریتم شما به صورت جادویی دوبرابر سریع تر کار کنه. چند دلیل برای این موضوع هست:

- مقدار زمانی که مدیریت موازی کردن یک الگوریتم طول میکشه خودش به زمان محاسبه اضافه میکنه (**Overhead of managing the parallelism**). فرض کنید که شما یک آرایه برای مرتب کردن شامل ۱۰۰۰ آیتم دارید. چطور این تسك هارو بین دو هسته تقسیم میکنید؟ آیا به هر کدام ۵۰۰ آیتم برای مرتب کردن میدین و در آخر دوتا آرایه سورت شده رو باهم داخل یک آرایه بزرگ مرتب شده ادغام میکنین؟ ادغام دو آرایه زمان میبره.
- متعادل نمودن بار ترافیکی یا لود بالانسینگ (**load balancing**). فرض کنید شما ۱۰ تسك برای انجام دادن دارید خب به هر هسته ۵ تسك میدین. اما هسته A تمام تسك های آسان رو دریافت مکینه که در نتیجه مثلاً در ۱۰ ثانیه کار رو تموم میکنه درحالی که هسته B تمام تسك های سخت رو دریافت کرده که در نتیجه ۱ دقیقه طول میکشه تا اون هارو انجام بدنه. که این یعنی هسته A ۵۰ ثانیه بیکار نشسته درحالی هسته B داشته تمام کار های سخت رو انجام میداده! شما چطور این تسك هارو به طور مساوی بین این دو هسته پخش میکنید که هر دو به یک اندازه سخت کار کنند؟

اگر شما به بحث تئوریه عملکرد سیستم و مقیاس پذیری علاقه دارین، ممکنه بحث الگوریتم های موازی برای شما باشه!

کاهش نگاشت MapReduce

یک نوع خاص از الگوریتم موازی وجود داره که به طور فزاینده ای داره به محبوبیتش اضافه میشه و بین افراد بیشتر و بیشتر محبوب میشه: الگوریتم توزیع شده (**distributed algorithm**). اجرا کردن الگوریتم های موازی روی لپ تاپتون اگر که دو تا چهار هسته نیازتون میشه مشکلی نداره اما اگر شما به هزاران هسته نیاز داشته باشین چی؟ بعدش میتوانید الگوریتمتون برای اجرا در چندین ماشین بنویسین. الگوریتم **MapReduce** یک الگوریتم توزیع شده محبوبه. شما میتوانید از اون از طریق ابزار محبوب متن باز **Apache Hadoop** استفاده کنید.

چرا الگوریتم توزیع شده مفید هستند؟

فرض کنین شما یک جدول با یک میلیارد یا یک تریلیون سطر در اختیار دارین، و میخواین یک دستور SQL پیچیده را روی اون اجرا کنید. نمیتوانین از MySQL برای این کار استفاده کنید چرا که این دیتابیس بعد از میلیار سطر اول به تقدا میفته و به مشکل میخوره. برای اینکار از طریق Hadoop از MapReduce استفاده کنید!

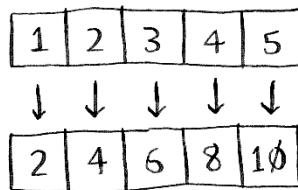
یا فرض کنید شما باید یک لیست بلند بالا از موقعیت ها شغلی رو بررسی کنید. بررسی هر موقعیت شغلی ۱۰ ثانیه طول میکشه و نیازه که شما ۱ میلیون موقعیت شغلی مثل این رو بررسی کنین. اگر این کارو با یک ماشین انجام بدین، ماه ها از شما وقت میگیره! اگر میتوانستین بین ۱۰۰ ماشین اجراس کنین، احتمالا این کار توی چند روز تمام میشه.

الگوریتم های توزیع شده برای وقت هایی که کار زیادی برای انجام دارین و میخواین سرعت زمانی که نیاز دارین رو بالا ببرین به شدت عالین. MapReduce به طور خاص بر اساس دو ایده ساده بنا شده: تابع map تابع reduce

تابع map

نحوه کار تابع map خیلی آسونه: یک آرایه میگیره و به هر آیتم توی اون آرایه اون تابعی رو که ما بهش دادیم رو اعمال میکنه. به عنوان مثال میخوایم که توی این آرایه، هر آیتم رو دو برابر کنیم (تابع ما وظیفه دوباره کردن رو داره):

```
>>> arr1 = [1, 2, 3, 4, 5]
>>> arr2 = map(lambda x: 2 * x, arr1)
[2, 4, 6, 8, 10]
```



الآن arr1 شامل [2, 4, 6, 8, 10] هستش،-- هر عنصری توی آرایه اول دو برابر شده! دوباره کردن یک عنصر تقریبا سریع انجام میشه. اما فرض کنین شما تابعی رو اعمال کنین که زمانی بیشتری رو برای این روند طی کنه. به این شبه کد نگاه کنین:

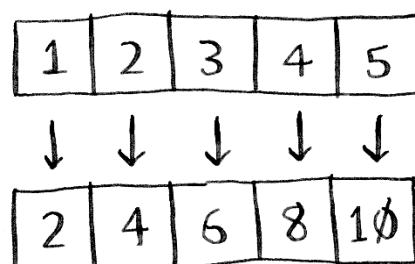
```
>>> arr1 = # A list of URLs
>>> arr2 = map(download_page, arr1)
```

در اینجا شما لیستی از url (آدرس اینترنتی)ها در اختیار دارین و میخواین صفحه های اون آدرس رو دانلود کنین و مقادیر رو در آرایه دوم بزرین. خب این میتونه چند ثانیه برای هر url طول بکشه. اگر شما ۱۰۰۰ تا url داشته باشین، ممکنه که دانلودش چندین ساعت طول بکشه.

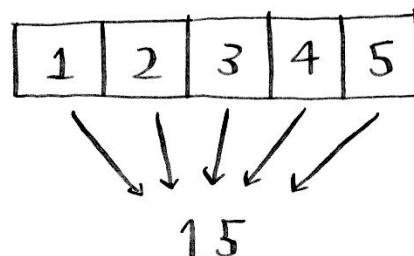
عالی نمیشد اگر که شما ۱۰۰ تا ماشین داشتین که میتوانست این کارو برآتون انجام بده و تابع `map` به طور اتوماتیک این کارارو بین این ماشین ها بخش میکرد؟ اینطوری شما میتوانستین ۱۰۰ صفحه رو همزمان دانلود کنید، و کار خیلی سریع تر انجام میشد! این همون ایده پشت `map` در `MapReduce` هستش.

تابع Reduce

گاهی اوقات تابع `reduce` افراد رو گیج میکنه. ایده کلی به این شکله که `reduce` لیستی از آیتم هارو به یک آیتم تبدیل میکنه(کاهش میده). با `map` شما از یک آرایه به یک آرایه دیگر میرین، به این شکل:



با `reduce` شما یک آرایه رو به یک آیتم تبدیل میکنید:



این هم یک مثال:

```
>>> arr1 = [1, 2, 3, 4, 5]
>>> reduce(lambda x,y: x+y, arr1)
15
```

توى این مثال شما تمام عناصر این آرایه رو باهم جمع میکنید: $1 + 2 + 3 + 4 + 5 = 15$! دیگه بیشتر از این وارد جزئیات `reduce` نمیشم چراکه آموزش های آنلاین زیادی ازش توى سطح وب هست.

از این دو مفهوم ساده برای اجرا کردن دستورات درباره داده ها در بین چند ماشین استفاده میکنه. وقتی که شما مجموعه داده بزرگی در اختیار دارین(میلیارد ها سطر)، `MapReduce` میتوانه جواب رو در عرض چند دقیقه به شما بده درصورتی که با استفاده از دیتابیس های مرسوم ممکنه چند ساعت طول بکشه.

بلوم فیلتر و هایپر لاگ (Bloom filter and HyperLogLog)

فرض کنین شما در حال اداره Reddit (یک شبکه اجتماعی) هستید. وقتی یک نفر یک لینک پست میکنه، شما میخواین ببینین که آیا این لینک قبلا پست شده یا نه. استوری هایی که قبلا پست نشده با ارزش از پست هایی که قبلا پست شدن به لحاظ میان. پس نیازه که شما بدونین آیا این لینک قبلا پست شده یا نه.

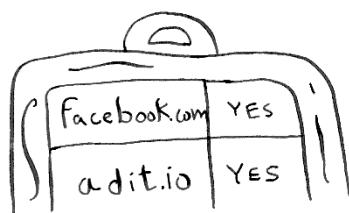
یا فرض کنین که شما گوگل هستید و دارین صفحه های وب رو بررسی میکنین (به اصطلاح کروول میکنین). شما میخواین تنها صفحاتی رو بررسی کنین که قبلا بررسی نکردین پس باید راهی پیدا کنین که ببینین این صفحه ای که دارین بررسی میکنین قبلا بررسیش کردین با نه.

یا فرض کنین که در حال اداره bit.ly که یک کوتاه کننده url است هستید. شما نمیخوايد که کاربرانتون رو به یک وب سایت مخرب هدایت کنین. برای این کار هم مجموعه ای از url هایی در اختیار دارید که مخرب شناخته میشوند. حالا شما باید راهی پیدا کنید که بفهمید آیا کاربر را به یک url در آن مجموعه هدایت میکنید یا خیر.

تمام این مثال ها یک مسئله رو دربر میگیرند. شما یک مجموعه بسیار بزرگ در اختیار دارید.



و شما یک آیتم جدید دارید و میخواین که بررسی کنین که آیا این آیتم جدیدی که تازه دریافت کردید در این مجموعه وجود دارد یا خیر. میتونین این کار خیلی سریع با هش انجام بدید. به عنوان مثال، فرض کنین که گوگل یک هش بزرگ در اختیار داره که کلید های این هش همان صفحه هایی هستند که بررسی شده اند.



شما میخواین بررسی کنین که آیا قبل از سایت [adit.io](#) رو بررسی کردید یا خیر. داخل هش دنبالش میگردید.

۲۶۵ → adit

یک کلید داخل این هستش پس شما قبل این سایت را بررسی کردید. زمان متوسط چک کردن سایت داخل هش (1) هستش. [adit.io](#) داخل هش بود پس بررسی شده و شما این موضوع رو در یک زمان ثابت پیدا که همان (1) باشد پیدا کردید. خیلی خوبه!

جز این موضوع که این هش قراره خیلی بزرگ باشه. گوگل تریلیون صفحه وب رو ایندکس میکنه. اگر این هش شامل تمام url هایی باشه که گوگل ایندکس کرده پس این هش خیلی فضا اشغال میکنه. [Reddit](#) و [bit.ly](#) هم همین مشکل کمبود فضا رو دارند. وقتی که با داده های زیادی سروکار دارید باید خلاقیت به خرج بدید!

بلوم فیلتر

بلوم فیلترها یک راه حل در اختیار ما قرار میدن. بلوم فیلترها ساختار داده احتمالی هستند. اونا جوابایی به شما میدن که ممکنه اشتباه باشه اما احتمالا درسته. بجای هش شما میتوینی از بلوم فیلتر پرسین که آیا من قبل این url رو بررسی کردم یا نه. یک جدول هش به شما جواب دقیق رو میده. یک بلوم فیلتر به شما جوابی میده که احتمالا درسته:

- ممکنه جواب مثبتی بده که اشتباهه. ممکنه گوگل بگه، "شما قبل این url رو بررسی کردید" درحالی که بررسی نشده.
- ممکن نیست جواب منفی اشتباه بده. اگر بلوم فیلتر بگه که "شما قبل این وب سایت رو بررسی نکردید!" پس قطعاً شما این وب سایت رو بررسی نکردید.

بلوم فیلترها واقعاً عالین چون فضای خیلی کمی رو اشغال میکنند. یک جدول هش باید تمام url هایی که توسط گوگل بررسی شده رو در خودش ذخیره کنه اما بلوم فیلتر مجبور نیست این کار رو بکنه. اون ها زمانی که شما به یک جواب قطعی نیاز ندارید (همونطور که در مثال ها دیدیم) واقعاً عالی عمل میکنن. این برای سایت [bit.ly](#) مشکلی نداره که بگه "من فکر میکنم که این آدرس ممکنه مخرب باشه پس خیلی مراقب باش".

هایپر لاغ

در کنار توضیحات بالا، یک الگوریتم دیگر به نام هایپر لاغ وجود دارد. فرض کنین گوگل میخواهد تعداد جستجوهایی یونیک (یکتا) بی که توسط کاربرانش انجام شده رو بشماره. یا فرض کنین آمازون میخواهد تعداد آیتم های یونیکی که کاربران، امروز به اون نگاه انداختن رو بشماره. جواب دادن به این سوالاً فضای زیادی رو اشغال میکنه! با گوگل شما باید یک لاغ (سیستم گزارش) از تمام جستجو های یونیک نگه دارید. وقتی که کاربر چیزی رو

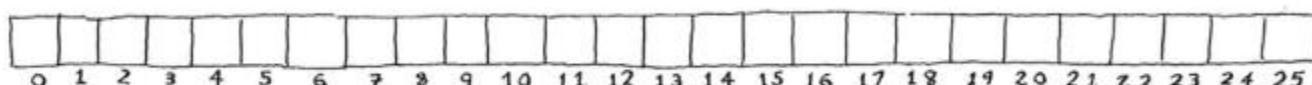
جستوجو میکنید شما باید چک کنین که آیا توی لگ وجود داره یا خیر. اگر نه شما باید اون لگ رو به داده هاتون اضافه کنین حتی برای یک روز هم این لگ ها بشدت زیاد میشن!

هایپرلگ لگ تعداد عناصر یونیک رو در یک مجموعه تخمین میزنه. دقیقا مثل بلوم فیلتر، به شما یک جواب دقیق نمیده ولی خوب جواب نزدیکی میده و تنها کسری از حافظه ای که اگر میخواستیم به جواب دقیق برسیم رو میگره.

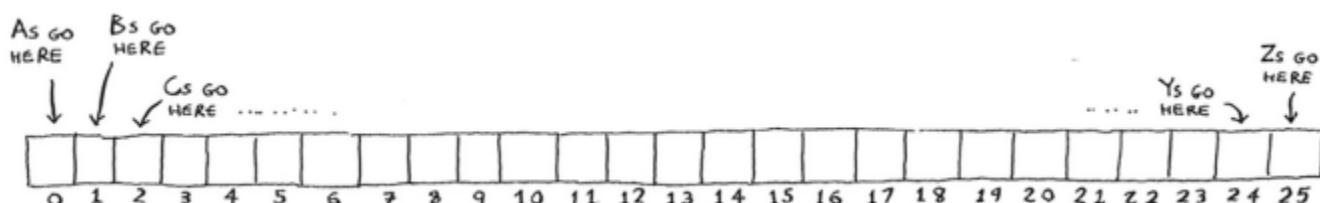
اگر داده های زیادی دارید و با پاسخ های تقریبی مشکلی ندارید، نگاهی به الگوریتم های احتمالی بندازید!

الگوریتم های SHA

هش کردن رو از فصل ۵ یادتونه؟ فقط برای یادآوری، فرض کنین که یک کلید دارین و میخواین مقدار مرتبط با این کلید رو در آرایه قرار بدین.



از یک تابع هش برای اینکه اسلاتی که میخواین مقدار رو توش قرار بدین رو بهتون بگه استفاده میکنین.



و درنهایت مقدار مورد نظر رو در اسلات جا گذاری میکنید.

این به شما اجازه جستوجو در یک زمان ثابت رو میده. وقتی که شما میخواین مقدار یک کلید رو بدونین میتوینین دوبار از تابع هش استفاده کنین و اون در زمان $O(1)$ بهتون میگه کدوم اسلات رو باید چک کنید.

در این مورد شما میخواهید که تابع هش یک توزیع خوب به شما بدهد، در نتیجه تابع هش یک رشته دریافت میکند و شماره اسلات مربوط به آن رشته را بر میگرداند.

مقایسه فایل ها

یک تابع هش دیگه، تابع الگوریتم هش امن (SHA secure hash algorithm) یا SHA هستش. با دریافت یک رشته، یک هش به شما در ازای اون رشته میده.

“hello” \Rightarrow 2cf24db...

اصطلاحات این بخش ممکنه یکم گیج کننده باشه. **SHA** یک تابع هش هستش که یک هش تولید میکنه، که این هش تولید شده یک رشته کوتاه هستش. تابع هش برای جداول هش یک رشته رو به ایندکس یک آرایه تبدیل میکرد درحالی که **SHA** از رشته به رشته تبدیل میکنه.

به ازای هر رشته، رشته متفاوتی تولید میکنه.

“hello” \Rightarrow 2cf24db...

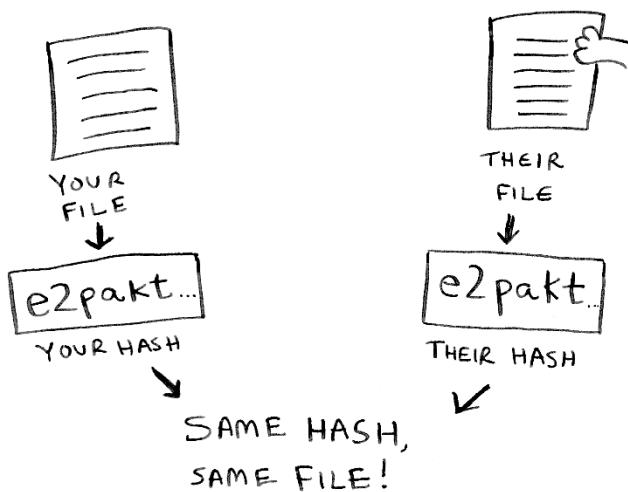
“algorithm” \Rightarrow b1eb2ec...

“password” \Rightarrow 5e88489...

نکته!

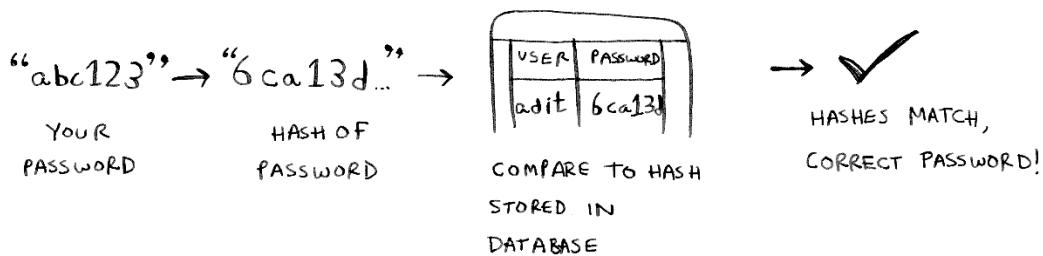
هش های **SHA** های مثال بالا کامل به دلیل طولانی بودن کامل نشان داده نشده و ادامش حذف شده.

شما میتوانید از **SHA** برای بررسی اینکه آیا دو فایلی که در اختیار داریم یکی هستند یا خیر، استفاده کنید. این روش زمانی که شما فایل با حجم بالا دارین خیلی کاربردیه. فرض کنین شما یک فایل ۴ گیگی در اختیار دارین. میخواین چک کنین بینین دوست شما هم این فایل ۴ گیگی رو داره یا خیر. نیازی نیست تلاش کنین که این فایل رو به دوستتون ایمیل کنین. بجاش میتوانید **SHA** فایل هارو محاسبه کنین و اون هارو باهم مقایسه کنین.



بررسی پسورد

SHA همچنین زمان هایی که میخواین دو رشته رو بدون اینکه معلوم باشه رشته های اصلی چی هستند باهم مقایسه کنین، خیلی کاربردیه. برای مثال، فکر کنین Gmail هک میشه و هکرها تمام پسورد هارو میدزدن. آیا پسورد شما اون بیرون لو رفته؟ نه لو نرفته. گوگل پسود با رشته اصلی رو ذخیره نمیکنه بلکه تنها SHA پسورد رو ذخیره میکنه. وقتی که شما پسوردتون رو تایپ میکنین گوگل اونو هش میکنه و با رشته هشی که در دیتابیس خودش داره مقایسه میکنه.



پس فقط هش هارو باهم مقایسه میکنه. نیازی نیست که خود پسورد رو ذخیره کنه! از SHA به طور خیلی رایجی برای هش کردن پسورد مثل همین مثالی که زدیم استفاده میشه. این یک هش یک طرفه. شما میتوانین هش یک رشته رو دریافت کنین. اما نمیتوانین هش یک رشته رو به رشته اصلی برگردونین.

$$\text{abc123} \rightarrow 6ca13d$$

$$? \leftarrow 6ca13d$$

که این یعنی اگر هش های SHA رو از گوگل بدزده، نمیتونه اونا رو به پسورد اصلی برگردونه! شما میتوانین یک پسورد رو به هش تبدیل کنین اما بر عکسش نه.

درواقع SHA یک خانواده از الگوریتم هاست: SHA-0, SHA-1, SHA-2, SHA-3. تا الان که من دارم اینو مینوسم SHA-0 و SHA-1 یکسری نقطه ضعف ها دارند. اگر شما دارین از الگوریتم SHA برای هش کردن پسورد استفاده میکنین، از SHA-2 یا SHA-3 استفاده کنین. در حا حاضر استاندارد طلایی برای هش کردن پسورد bcrypt (گرچه که هیچ چیز غیرقابل خطأ نیست).

هشینگ حساس به محل (Locality-sensitive hashing)

SHA یک ویژگی مهم دیگر هم دارد: و اون هم حساس نبودن به محل. فرض کنین که یک رشته دارین و میخواین هشش رو تولید کنید.

$d \rightarrow cd6357$

اگر فقط یکی از کاراکتر های رشته رو تغییر دهید و دوباره هش رو تولید کنید خواهید دید که هش تولید شده کاملاً متفاوت است.

این ویژگی خوبه بخاطر اینکه هر کس نمیتوانه هش هارو باهم دیگه مقایسه کنه تا ببینه چقدر به شکستن پسورد نزدیک شده.

اما گاهی اوقات شما برعکس این موضوع رو میخواین: شما تابع هشینگ حساس به محل رو میخوايد. این جا جاییه که $Simhash$ وارد داستان میشه. اگر تغییر کوچکی در رشته وارد کنید، $Simhash$ هشی تولید میکنه که فقط یکم فرق داره. این به شما اجازه میده که هش هارو باهم مقایسه کنین و ببینین که چقدر شبیه هم هستند چه تقریباً کاربردیه!

- گوگل از $Simhash$ برای شناسایی محتواهای تکراری هنگام بررسی کردن وب استفاده میکنه.
- یک استاد میتوانه از $Simhash$ استفاده کنه تا ببینه آیا دانشجو مقالشو از روی وب کپی کرده یا نه.
- $Scibd$ به کاربرانش امکان آپلود مستندات یا کتاب ها برای ب اشتراک گذاری با دیگران رو میده. اما این سایت اجازه آپلود محتواهای دارای حق چا یا همون کپی رایت رو نمیده! مثلاً این سایت میتوانه از $Simhash$ استفاده کنه تا چک کنه که آیا این چیزی که آپلود شده شبیه کتاب هری پاتره اگر آره بصورت اتوماتیک ردش کنه.

زمانی که میخواین چک کنین که چند آیتم بهم شبیه هستند یا خیر ، خیلی کاربردیه.

پروتکل تبادل کلید دیفی-هلمن (Diffie-Hellman key exchange)

الگورتم دیفی-هلمن شایستگی اینو داره که اینجا بهش اشاره کنیم چرا که یک مسئله قدیمی رو با استفاده از یک روش عالی حل میکنه. چطور یک پیام رو رمزگاری کنیم که فقط اون کسی که پیام رو دریافت میکنه بتوانه اونو بخونه؟

راحت ترین راه اینه که با روش سایفر (cipher) بریم جلو ، اینطوری که $a = 1$ ، $b = 2$ و ... بعش اگر من یک پیام برای شما بفرستم که "4,15,7" شما میتوینین اون به این ترجمه کنین "d,o,g". اما برای اینکار هم جفتمون روی استفاده از سایفر توافق کردیم. ما نمیتونیم توی ایمیل اینو باهم توافق کنیم چرا که یک نفر میتوانه نفوذ کنه و بفهمه که ما از سایفر استفاده میکنیم و پیام رو رمزگشایی کنه. این بدء، حتی اگر به صورت حضوری هم این پیام رو بهم

بدیم یک نفر ممکنه حدس بزنه که ما داریم چی میگیم چون سایفر اینقدر پیچیده نیست. پس ما باید هر روز اینو تغییر بدیم.

حتی اگر هم طوری مدیریتش کنیم که هر روز تغییرش بدیم، شکستن سایفر ساده‌ای مثل با یک حمله بروت فورس خیلی ساده‌س. فرض کنین من این پیام رو میبینم "9, 6, 13, 13, 16, 24, 16, 19, 13, 5": حدس میزنم که این از $a = 1$, $b = 2$ و ... استفاده میکنه.

9	6	13	13	16	24	16	19	13	5
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
i	f	m	m	p	x	p	s	m	e

که خب میبینم چرت و پرت شده. بیاین $a = 2$, $b = 3$ و ... رو امتحان کنیم.

9	6	13	13	16	24	16	19	13	5
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
h	e	l	l	o	w	o	r	l	d

این کار کرد! شکستن یک سایفر ساده‌ای مثل این خیلی سادس. آلمانی‌ها از یک سایفر به شدت پیچیده در جنگ جهانی دوم استفاده کردند اما همچنان شکسته شد. دیفی-هلمن هردو مسئله رو حل میکنه:

- جفت طرفین نیاز نیست که چیزی درباره سایفر بدونن. پس نیازی نداریم که باهم ملاقات کنیم و سر اینکه سر چه سایفری باید باهم توافق کنیم فکر کنیم.
- شکستن پیام‌های رمزگذاری شده توی این الگوریتم به شدت سخته.

دیفی-هلمن برای اینکار دو تا کلید داره: یک کلید عمومی و یک کلید خصوصی. کلید عمومی همونطور که از اسمش معلومه عمومیه. شما میتوینین توی یک وبسایت پستش کنین یا به یک دوست ایمیلش کنین یا هر کاری که دوس دارین باهاش بکنین. نیازی نیست که پنهانش کنین. وقتی کسی میخواهد به شما پیام بده با استفاده از کلید عمومی اونو رمزگذاری میکنه و این پیام رمزگذاری شده تنها با کلید خصوصی رمزگشایی میشه. تا زمانی که شما تنها کسی هستید که کلید خصوصی دستشه، تنها شما هستین که میتوینین پیام هارو رمزگشایی کنین.

الگوریتم دیفی-هلمن همچنان در عمل در کنار جانشینش **RSA** استفاده میشه. اگر به رمزنگاری علاقه‌مند هستید دیفی-هلمن یه جای خوب برای شروعه: این الگوریتم خیلی عالیه و دنبال کردنش زیاد سخت نیست.

برنامه نویسی خطی (Linear programming)

بهترین رو برای آخر کار کنار گذاشتم. برنامه نویسی خطی یکی از خفن ترین چیزایی که میشناسم.

برنامه نویسی خطی سعی میکنه چیزی رو که بهش میدی با در نظر گرفتن محدودیت ها به حداکثر برسونه. به عنوان مثال فرض کنین که شما یک شرکت دارین که ۲ تا محصول تولید میکنه، پیراهن و ساک. پیراهن ۱متر پارچه و ۵ تا دکمه نیاز داره. ساک هم ۲متر پارچه و ۲ دکمه نیاز داره. شما هم ۱۱ متر پارچه و ۲۰ تا دکمه در اختیار دارین. شما از هر پیراهن ۲ دلار و از هر ساک ۳ دلار درمیارین. چنتا پیراهن و ساک باید تولید کنین تا سودتونو به حداکثر برسونین؟

در اینجا شما سعی میکنین که سود رو به حداکثر برسونین و با مقدار مواد اولیه‌ای که دارین محدود شدین.

یک مثال دیگه شما یک سیاست مدار هستید و میخواین تعداد رای هایی که میگیرین رو به حداکثر برسونین. تحقیقات شما نشون میده که به طور میانگین به ازای هر رای از سانفرانسیکویها یک ساعت کار (ماکتینگ ، تحقیق، و ...) یا یک ساعت و نیم کار به ازای هر شیکاگویی. شما حداق به رای ۵۰۰ سانفرانسیکویی و ۳۰۰ شیکاگویی نیاز دارین. شما ۵۰ روز فرصت دارید. همچنین سانفرانسیکو ۲ دلار و شیکاگو ۱ دلار هزینه میبره. کل بودجه شما ۱۵۰۰ دلاره. تعداد حداکثر رایی که شما میتوانید بیگرید چقدر است(سانفرانسیکو + شیکاگو)؟

در اینجا شما سعی میکنین که رای هارو به حداکثر برسونید و با زمان و مقدار پول محدود شدین.

ممکنه که فکر کنین : تو خیلی راجب نحوه بهینه سازی در این کتاب بحث کردی. چطور اینا به برنامه نویسی خطی ربط دارن؟ تمام الگوریتم های گرافی میتونه توسط برنامه نویسی خطی انجام بشه. برنامه نویسی خطی یک چهارچوب کلی تره، و مسئله های گراف زیر مجموعه اون هستند. امیدوارم که ذهنتون منفجر شده باشه!

برنامه نویسی خطی از الگوریتم **Simplex** استفاده میکنه. این یک الگوریتم پیچیدس، بخارط همین هم هست که داخل این کتاب نیاوردمش. اگر شما به بحث بهینه سازی علاقه‌مند هستید، نگاهی به برنامه نویسی خطی بندازین!

سخن آخر

امیدوارم این تور سریع ۱۰ الگوریتمی که داشتیم بهتون نشون داده باشه که چقدر چیزای زیادی برای کشف کردن مونده. بنظر من بهترین راه برای یادگیری اینه که چیزی رو که بهش علاقه دارین پیدا کنین و مستقیم شیرجه بزنین توش این کتاب هم به شما یک پایه و پی محکم برای انجام این کارو داد.