آمادگی برای مصاحبه های برنامه نویسی به سبک شرکت های بزرگ دنیا

جلد اول - ویرایش یک

فروردین ۹۸

دکتر حسین سیادتی

دکتر سیما جعفری خواه

فهرست مطالب

۱. اصول مصاحبه های برنامه نویسی

در ایران روش سازمان یافته ای برای مصاحبه های برنامه نویسی وجود ندارد. شرکت ها تاکید اضافی روی بلد بودن فریم ورک ها و تکنولوژی هایی دارند که امروز بوجود می آیند و خیلی سریع با یکی بهتر جایگزین می شوند. در این دنیای وانفسا، برنامه نویسان هم نمیدانند که چه چیزی را برای مصاحبه یاد بگیرند و چگونه برای مصاحبه های برنامه نویسی آماده شوند. روش انجام مصاحبه ها موجب شده است که شرکت های توسعه نرم افزاری نتوانند برنامه نویسان را درست محک بزنند و نهایتا معمولا آنهایی انتخاب می شوند که در رزومه خود کار با ابزارهای جدید را ولو بسیار محدود گنجانده باشند. در عمل این پدیده منجر به استخدام برنامه نویسانی خواهد شد که معمولا مهارت های کلیدی یک برنامه نویس خوب را ندارند و وبال گردن شرکت خواهند شد. در شرکت های بزرگ دنیا و همچنین در دنیای استارت آپ هایی که نرم افزار تولید می کنند، **مهارت حل مسائل برنامه نویسی** یک مهارت کلیدی است و معیار اصلی برای استخدام برنامه نویس است.

روش جاری استخدام احتمال از دست دادن یک برنامه نویسی خوب و انتخاب اشتباه را افزایش می دهد. بخصوص، شخصیت ویژه برنامه نویسان این پدیده را تشدید می کند. به عنوان مثال، در حالیکه در بسیاری مشاغل درونگرا بودن یک فاکتور شخصیتی منفی تلقی می شود، این تیپ شخصیتی تعداد قابل توجهی از برنامه نویسان خوب می باشد. رویه نامناسب برای مصاحبه ممکن است موجب شود که مصاحبه کننده فردی کم مهارت اما پر حرف در زمان مصاحبه را به یک درونگرا با مهارت بسیار بیشتر ترجیح دهد. معیار اصلی ارزیابی یک برنامه نویس **قدرت حل مسئله** می باشد.

۱.۱. اهداف یک مصاحبه برنامه نویسی

اهداف اصلی یک مصاحبه کننده در یک مصاحبه برنامه نویسی آن است که ما (متقاضی استخدام برای شغل برنامه نویسی) را از دیدگاه توانایی درک و حل مسئله، پیاده سازی، و ارتباط با تیم برنامه نویسان بسنجد. بطور خاص ما باید نشان دهیم که:

- **قدرت حل مسئله:** می توانیم راه حل الگوریتمی برای یک مسئله ارائه کنیم و مزایا و معایب آن را به صورت مستدل بگوییم.
- **توانایی ارتباط تیمی:** می توانیم راه های حل مسئله را برای افراد دیگر تیم توضیح دهیم و توضیحات افراد تیم را بشنویم و در حل مسئله از آن استفاده کنیم.

- **برنامه نویسی:** می توانیم برنامه مربوط به راه حلی را که ارایه کردیم را پیاده سازی کنیم. باید نشان دهیم که می توانیم برنامه ای خوانا، موجز، و با قابلیت گسترش و استفاده مجدد بنویسیم و همچنین دقت زیاد و سرعت قابل قبولی در برنامه نویسی داریم.
- **حس مثبت:** مثبت و با انرژی هستیم و تیم از کار کردن با ما لذت خواهد برد و پیشرفت می کند.

۲.۱. چگونگی یک مصاحبه برنامه نویسی

بسیاری از برنامه نویسان جویای کار تصویر دقیقی از یک مصاحبه برنامه نویسی در شرکت های خوب دنیا ندارند. برخی به اشتباه تصور می کنند که در این مصاحبه ها تست هوش برگزار می شود. این واقعیت ندارد. هدف این بخش آن است که تصویر دقیقی از یک مصاحبه برنامه نویسی به سبک شرکت های بزرگ دنیا بدهد.

- **تعداد مصاحبه:** برخی شرکت ها چندین دور مصاحبه برای استخدام برنامه نویس دارند. به عنوان مثال شرکت گوگل یک مصاحبه آنلاین و ۵ مصاحبه حضوری برای ارزیابی یک برنامه نویس دارد. شرکت های مختلف در تعداد مصاحبه ها اندکی با هم متفاوت هستند.
- **مکان و فضا:** مصاحبه به صورت آنلاین یا در محل شرکت برگزار می شود. مصاحبه آنلاین تنها در اولین مرحله انجام می شود و مصاحبه شونده برنامه را در یک ویرایشگر آنلاین (مثلا گوگل داک) می نویسد. مصاحبه های حضوری در محل شرکت و در یک اتاق شامل یک تخته سیاه یا وایت برد انجام می شود.
- **زمان:** معمولا مصاحبه های یک ساعت طول می کشد.مصاحبه کننده راس ساعت مصاحبه کننده را ملاقات می کنند و به وی خوش آمد می گوید.
- **معرفی ابتدای جلسه:** مصاحبه کننده که خود یک برنامه نویس است رتبه شغلی و کلیات پروژه هایی کار روی آن کاری می کند را در حد ۲ دقیقه تشریح می کند. سپس از مصاحبه شونده می خواهد که کمی درباره خودش بگوید. مصاحبه شونده در حد ۲ دقیقه کلیات مدارج علمی و تجربی **مرتبط با موقعیت شغلی** که برای پذیرش در آن مصاحبه می شود را می گوید. ممکن است که مصاحبه کننده سوالاتی در مورد توضیحات وی داشته باشد که به صورت مختصر ولی شفاف توضیح می دهد.
- **پرسش الگوریتمی و پاسخ**: سپس مصاحبه کننده یک سوال برنامه نویسی مطرح می کند و از برنامه نویس می خواهد آن را حل کند. این سوال عمدا مبهم است. برنامه نویس با ارایه یک یا چند مثال از ورودی و خروجی برنامه سعی می کند که سوال را بهتر درک کند و داده های ورودی و خروجی را بشناسد. معمولا سوالات برنامه نویسی نیاز به فکر کردن از جنبه های مختلف دارند. توقع مصاحبه کننده آن است برنامه نویس در مورد **راه حل های** مسئله با دقت فکر کند، مزایای و معایب آن ها را ارزیابی کند و بهترین راه را انتخاب نماید. شروع به پیاده سازی بدون فکر خصلت

زیان بار بسیاری از نوبرنامه نویسان هست و شرکت ها حتما از چنین افرادی دوری می کنند. مصاحبه شونده برای اینکه نشان دهد که از این قماش نیست، قبل از هرگونه برنامه نویسی، مسئله و راه حل های آن را کامل در نظر میگیرد و جزییات آنچه که فکر می کند را توضیح می دهد. به اصطلاح برنامه نویس **بلند بلند فکر می کند** و با استفاده از وایت برد راه حل های خود را تشریح می کند. این کار را معمولا با تشریح ساده ترین راه حل (معمولا جستجوی کامل یا همان brute force) شروع می کند. سپس مرحله به مرحله با راه حل های بهتر سرعت یا حافظه استفاده شده توسط الگوریتم را کاهش می دهد. یک برنامه نویس موفق در عرض 20 دقیقه از یک راه حل ساده به راه حل بهینه می رسد. در صورتی که مصاحبه کننده راه حل را تایید کرد، مصاحبه شونده شروع به نوشتن کد برنامه روی وایت برد می کند. او در حین نوشتن برنامه آن را خط به خط توضیح می دهد. با پایان نوشتن برنامه، برنامه نویس برنامه خود را مرور و سپس با چند داده ورودی تست می کند. در صورتی که برنامه نویس در زمان مناسب برنامه را بنویسد (حدود 20 دقیقه) ممکن است مصاحبه کننده سوالات تکمیلی داشته باشد (مثلا چگونه برنامه را تغییر دهیم که با داده های بسیار زیاد کار کند) و بپرسد که چه تغییراتی در حل مسئله باید داده شود.

- **فرصت پرسش برای برنامه نویس:** در 5 دقیقه انتهایی برنامه مصاحبه کننده از مصاحبه شونده می خواهد که چنانچه سوالی دارد بپرسد. یک مصاحبه شونده زیرک حتما سوال می پرسد. مثلا اگر در مصاحبه پذیرفته شوم روی چه پروژه ای کار خواهم کرد؟ یا اینکه تجربه شما از کار کردن در این شرکت چیست؟ و نهایتا مراحل بعدی استخدام چیست؟
- **تشکر و تشکر:** در انتهای جلسه طرفین از یکدیگر بابت فرصت مصاحبه و وقتی که طرف مقابل گذاشته تشکر می کند. معمولا مصاحبه شونده بعد از بازگشت در همان روز یا روز بعد ایمیلی به مصاحبه کننده می زند و در ضمن تشکر مجدد ابراز می کند که برای پاسخ به هر سوال احتمالی دیگر آمادگی دارد و اینکه منتظر پاسخ آن هست.

۳.۱. توصیه هایی به مصاحبه گر

یک برنامه نویس خوب مهمترین دارایی یک شرکت نرم افزاری است و شرکت ها برای استخدام استعدادهای برتر با یکدیگر در رقابتند. همانطور که شما یک متقاضی استخدام را ارزیابی می کنید، او نیز شما را ارزیابی می کند. بنابراین بسیار مهم است که نماینده خوبی برای شرکت باشید. حتی اگر مصاحبه شونده توانایی علمی مناسب ندارد با احترام رفتار کنید. این امر نهایتا منجر به اوازه خوش کمپانی شما از نظر فضای کاری می شود.

دوستانه و دستیافتنی (approchable) باشید و سوالاتی که یک متقاضی استخدام دارد را با حوصله پاسخ دهید.

شفاف، صادق، و عادل باشید. در مورد حقوق، امکانات رفاهی برای کارکنان، تعداد کارکنان و پروژه ها، حجم و ساعات کار و غیره صادق باشید. توجه کنید که دیر یا زود عضو جدید شرکت شما به حقایق پی خواهد برد. در طولانی مدت، بی صداقتی مهلک ترین ضربه ها را به شما وارد خواهد کرد.

در انتخاب سوال مصاحبه دقت کنید. به عنوان مثال سوال ‹‹الگوریتم دایجکسترا را پیاده کن›› خوب نیست. به این دلیل که (۱) معمولا برای الگوریتم های معروف پیاده سازی های استاندارد وجود دارد و (۲) اگر هم مصاحبه شونده بتواند درست جواب دهد نشان دهنده حافظه خوب هست، نه توانایی حل مسئله. بنابراین باید از بانک سوالات درست یک سوال انتخاب شود. با جستجو در سایتهایی مانند leetcode.com می توانید سوالات خوبی را برای مصاحبه انتخاب کنید.

مصاحبه شونده را غافلگیر نکنید. مهم این هست که مصاحبه شونده بداند که شما چگونه مصاحبه می کنید. بنابراین، هنگام دعوت برای مصاحبه، نحوه انجام مصاحبه را برای وی توضیح دهد. توضیحات می تواند شامل طول مدت مصاحبه، نوع سوالی که قرار است از وی بپرسید، نحوه پاسخگویی (مثلا پای تخته یا با کامپیوتر)، طول مصاحبه و زمان بندی (مثلا حدود ۴۰ دقیقه برای حل مسئله و نوشتن کد وقت) و معرفی منبعی که بتوانند سوالات مشابه را ببینند یا تمرین کنند. شفافیت شما با مصاحبه شونده فرهنگ کاری شما را نشون می دهد و یاد می دهد که آنها نیز با شما شفاف باشند.

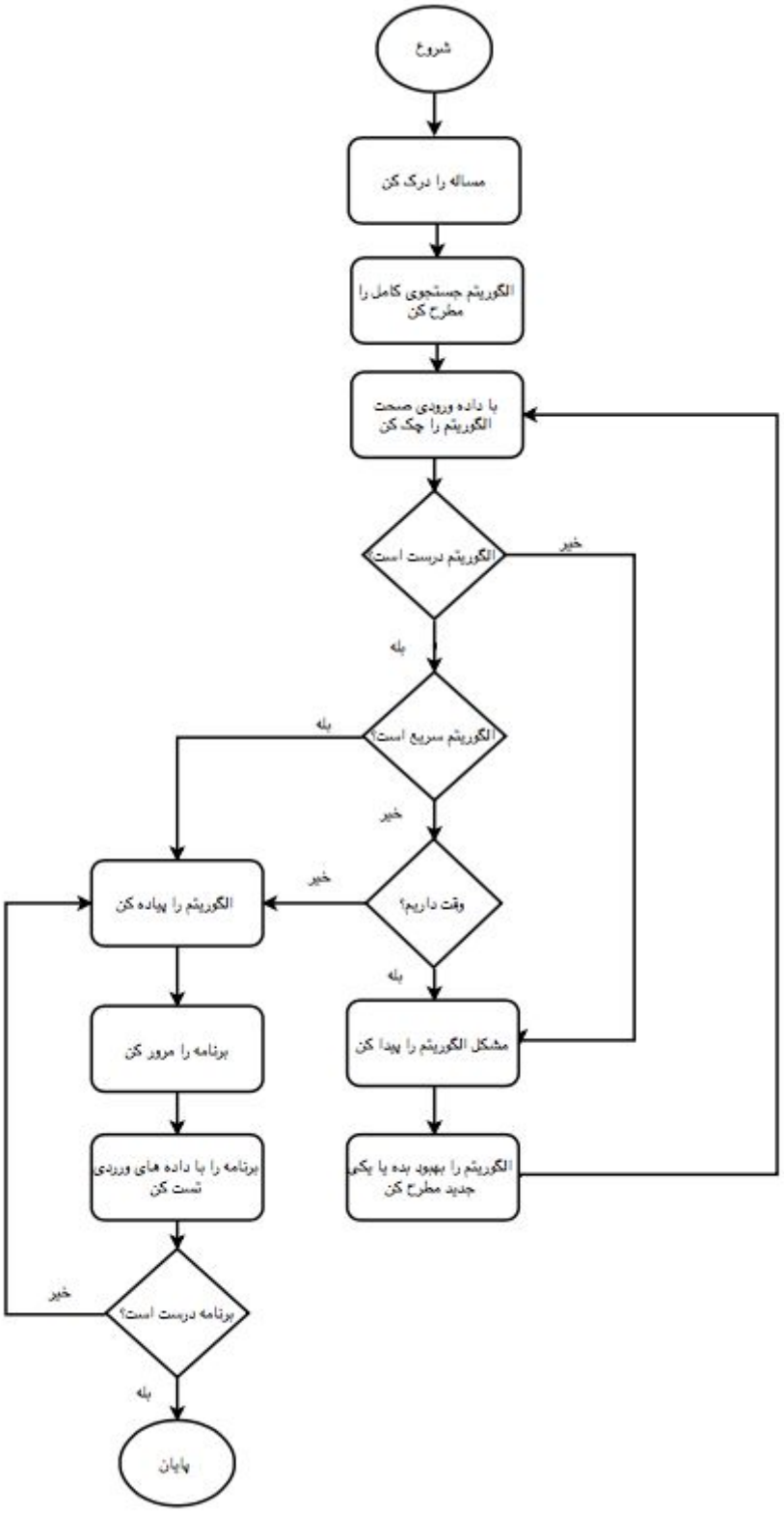
۲. استراتژی درست پاسخگویی به سوالات

۱.۲ استراتژی بهبود مرحله به مرحله

هدف و مراحل پاسخ به مسائل الگوریتمی در حین مصاحبه های برنامه نویسی در مقایسه با مسابقات برنامه نویسی (مثلا ACM) متفاوت است. در مسابقات برنامه نویسی تنها جواب آخر مهم است و نیاز نیست برای کسی راه حل خود را شرح دهیم. در مقایسه در مصاحبه های برنامه نویسی باید برای مصاحبه کننده همزمان که فکر می کنیم تعریف کنیم که چطور فکر می کنیم یا به اصطلاح **فکرمان را داد بزنیم** (Think out Loud). این کار به مصاحبه کننده نشان خواهد داد که ما چطور فکر می کنیم (thought process). مصاحبه کننده می خواهد بداند که آیا ما ذهن روشنی برای حل مسائل الگوریتمی داریم؟

طبیعی است که ذهن بسیار سریع تر از زبان فعالیت می کند و می خواهد ایده های مختلف را روی مسئله امتحان کند تا راه بهینه را بیابد اما زبان از فکر عقب می افتد. اگر استراتژی درستی نداشته باشیم این ممکن است به مکث های نامناسب یا پرت و پلا گویی منجر شود که هر دو نشانه مهارت کم در ارتباط و حل مسئله است. بنابراین باید یاد بگیریم که چگونه ذهن خود را در کانال درستی به صورت **قطاری از ایده ها** هدایت کنیم، چطور یک جریان گفتگوی نرم و مرحله به مرحله بسازیم، و کجای فکرمان را داد بزنیم.

بهترین راه برای ایجاد قطاری از ایده ها استفاده از **استراتژی بهبود مرحله به مرحله** است. در این روش پاسخگویی به صفر تا صد مسئله، یعنی از فهم سوال، سپس بدیهی ترین (اما احتمالا کندترین) راه حل تا یافتن بهترین راه را گام به گام به فکر و زبان می آوریم. در این پروسه احتمالا لازم باشد سوالی از مصاحبه کننده بپرسیم تا منظور وی از سوال را دقیق بفهمیم. همچنین احتمالا لازم شود که توضیحاتی بدهیم که از دید ما بدیهی است. باید این کار را انجام دهیم تا مصاحبه کننده بداند که ما چطور فکر می کنیم. نمودار زیر گامهای دقیق استراتژی مرحله به مرحله که باید در یک مصاحبه دنبال کنیم را در قالب یک فلوچارت نشان می دهد.



مراحل استراتژی بهبود مرحله به مرحله برای پاسخ به سوالات مصاحبه های الگوریتمی

توضیحات کلیت گامها به این صورت می باشد:

- **فهم مسئله:** ابتدا مثالی از ورودی و خروجی بزنیم که نشان دهیم مسئله را درست فهمیده ایم. همچنین این کمک می کند که نوع ورودی و فرمت آن را با مصاحبه کننده چک کنیم. این مرحله بسیار مهم است زیرا باید مطمئن شویم که مسئله اشتباهی را حل نمی کنیم.
- **راه حل جستجوی کامل:** حل مسئله را از کند ترین روش که جستجوی کامل (brute force) است آغاز می کنیم. راه حل را برای مصاحبه کننده تشریح می کنیم. سپس سرعت و میزان حافظه آن را (با استفاده از تخمین های Big O که در مبحث الگوریتم ها رایج است) ارزیابی می کنیم و شرح می دهیم. هدف از این کار این است که (۱) مسئله را بهتر بشناسیم و بتوانیم تمام حالتهای آن را در نظر بگیریم و (۲) در صورتی که نتوانستیم راه حل بهینه تر را در زمان مناسب (معمولا ۲۰ تا ۳۰ دقیقه) بیابیم به مصاحبه کننده نشان داده ایم که توانایی یافتن حداقل یک راه حل (اگرچه غیر بهینه) برای مسئله را داریم. حتی اگر جواب بهینه را نیافتیم باید از مصاحبه کننده بخواهیم که به ما اجازه دهد تا راه حل غیر بهینه را پیاده سازی کنیم تا هنر پیاده سازی یک الگوریتم را نشان دهیم.
- **آزمایش دستی با داده ورودی:** با دادن ورودی های مناسب الگوریتم خود را امتحان می کنیم.
- **تصمیم در مورد پیاده سازی:** اگر راه حل به اندازه کافی خوب است یا وقت پیدا کردن الگوریتم مناسب گذشته است به مرحله *'پیاده سازی'* بروید.
- **یافتن ایراد راه حل:** در صورتی که الگوریتم درست است اما سرعت آن کم یا حافظه استفاده شده زیاد است باید اشکال آن را به زبان بیاوریم. مثلا اگر الگوریتمی که ارایه کرده ایم سرعت آن به صورت نمایی کند می شود باید بگوییم که این الگوریتم با یک میلیون داده ورودی بسیار کند و غیر عملی است. ممکن است الگوریتم حافظه زیادی استفاده کند. بیان ایراد راه حل به ما ضربه نمی زند بلکه کمک می کند که باهوش بنظر برسیم. علاوه بر این، باید فکر کنید که کجای الگوریتم ما علت کندی آن است. مثلا ممکن است که الگوریتم پیشنهادی ما یک کار اضافی را تکرار کند. یافتن چنین ایرادی معمولا مقدمه ای برای ارائه راه حل بهتر است.
- **ارائه راه حل بهتر:** با تامل در ایراد روش قبلی، امتحان کردن ساختمان های داده شناخته شده، یا الگوریتم های مرتبط سعی می کنیم که راه حل بهتری بیابیم. معیار برای ارزیابی الگوریتم ها سرعت و میزان حافظه مورد استفاده برنامه است. معمولا جواب نهایی یک راه حل چند جمله ای (polynomial) یا چیزی بهتر از آن است.
- برو به مرحله *'آزمایش دستی با داده ورودی'*
- **پیاده سازی:** راه حل خود را پیاده سازی می کنیم. در حین نوشتن خطوط برنامه باید فکرمان را داد بزنیم. شاید لازم باشد به یکی از خطوط قبلی برنامه برگردیم تا چیزی را اصلاح کنیم. این طبیعی است و ایرادی ندارد. نوشتن برنامه

یک فرایند مرحله به مرحله است و کسی ما را بخاطر بهبود آن و بالا پایین رفتن در کد، اضافه کردن متغیر، تغییر نام متغیر، و غیره سرزنش نخواهد کرد.

- **مرور نهایی برنامه:** قبل از آنکه اعلام کنیم برنامه را کامل کرده ایم یکبار دیگر منطق برنام را چک میکنیم تا مطمئن شویم که آن را درست پیاده سازی کرده ایم. با این کار نشان می دهیم که دقت کافی برای اجتناب از اشکالات برنامه نویسی اجتناب ناپذیر را داریم. همیشه چیزی هست که باید بهتر شود.
- **تست نهایی:** بعد از نوشتن برنامه با داده های ورودی تستی برنامه خود را امتحان میکنیم تا مطمئن شویم که خوب کار می کند. مراحل تست را برای مصاحبه کننده تشریح می کنیم.

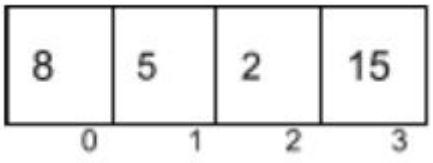
۲.۲. مثالی از اجرای استراتژی بهبود مرحله به مرحله

در این بخش راه اجرای کامل مراحل استراتژی بهبود مرحله به مرحله که در بخش قبلی مطرح شد را با یک مثال به صورت عملی شرح می دهیم.

مثال: فرض کنید که سوال زیر در یک مصاحبه برنامه نویسی از ما پرسیده می شود:

دنباله ای از اعداد صحیح یکتا و یک عدد ویژه (target) داده شده اند. برنامه ای بنویسید که به عنوان خروجی، اندیس دو عنصر از این دنباله که حاصل جمعشان برابر با مقدار ویژه است را برگرداند. فرض کنید که برای هر ورودی دقیقا یک راه حل وجود دارد.

پاسخ: ابتدا سعی میکنیم با آوردن مثالی ساده پرسش را بهتر درک کنیم (***مرحله فهم مسئله***). فرض میکنیم دنباله زیر به عنوان ورودی داده شده است ...



نمونه ای از لیست (آرایه) ورودی. مقادیر درون مربع ها عناصر لیست می باشند. عدد زیر مربع ها اندیس عنصر در لیست را نشان می دهد.

... و مقدار ویژه که دنبالش هستیم 10 می باشد. برنامه ای که می نویسیم باید اندیس زوج مقادیری که مجموعشان 10 می شود را برگرداند. در نگاه اول ممکن است بنظر برسد که دو پاسخ برای این ورودی وجود دارد بخاطر اینکه 5+5 و 8+2 هر دو برابر 10 می شوند. این موضوع را با مصاحبه کننده در میان می گذاریم. فرض ما این است که مصاحبه کننده تاکید می کند که اندیسهای زوجی که می یابیم باید متفاوت باشند. بنابراین زوج پاسخ 8 و 2 و خروجی مورد نظر اندیسهای آندو یعنی 0 و 2 است.

حال سعی میکنیم الگوریتم جستجوی کامل برای این مسئله را بیابیم و آن را تشریح کنیم (**راه حل جستجوی کامل**). الگوریتم جستجوی کامل برای حل این پرسش یعنی عناصر دنباله را دو به دو با هم جمع کنیم و با مقدار ویژه مقایسه کنیم.

با استفاده از داده های ورودی شرح می دهیم که زوج های مقادیر در مثال فوق چه هستند و چگونه بدست می آیند (**آزمایش دستی با داده ورودی**). به عنوان مثال می گوییم که عنصر اول در کنار هر یک از عناصر بعدی اش سه زوج (5, 8)، (2, 8), (15, 8) را تشکیل می دهند. به همین شکل هر عنصر در موقعیت iام با تمام عناصر در موقعیت i+1 تا انتهای لیست یک زوج

را تشکیل می دهند. در حین تشکیل زوج های ممکن، هرگاه مقدار ویژه به عنوان حاصل مشاهده شد اندیس دو مقدار یافته شده را به عنوان حاصل برمی گردانیم. در بدترین حالت روش جستجوی کامل باید همه زوج های ممکن، یعنی n * (n-1)/2 زوج مقادیر را بوجود آورد. بنابر این پیچیدگی محاسباتی این الگوریتم $O(n^2)$ می باشد.

در این لحظه احتمالا مصاحبه کننده ما را برای یافتن اولین راه حل عملی تحسین می کند. اما واضح است که این روش کند است و روی یک لیست مثلا یک میلیونی خیلی کند است. مصاحبه کننده ما را برای برنامه نویسی سیستم هایی می خواهد که داده های زیادی دارند و به این دلیل باید الگوریتم بهینه با سرعت مناسب بکار ببرند. بنابراین از ما میپرسد که چگونه می شود این را بهتر کرد؟ در این لحظه می توانیم بررسی کنیم که بکار بردن ساختمان های داده دیگر مثلا یک دیکشنری می تواند حل مسئله را تندتر کند؟ یا اینکه اگر داده ها را مرتب کنیم سریعتر به پاسخ می رسیم؟

فرض ما این است که با خونسردی کامل گام به گام دیدگاههای خود را برای راه حل هایی که سریعتر هستند بیان می کنیم. ممکن است که هنوز سرعت کافی برای پیدا کردن سریع راه حل بهینه را نداشته باشیم. در بدترین حالت ممکن است بیش از نیم ساعت از مصاحبه گذشته باشد و ما راه حل نهایی را نیافته باشیم. بنابراین برای اینکه از خرابی کامل مصاحبه جلوگیری کنیم محترمانه از مصاحبه کننده می خواهیم که اجازه دهد که راه حل جستجوی کامل که پیشنهادی مان را پیاده کنیم (**تصمیم به پیاده سازی الگوریتم درست ولی غیر بهینه به دلیل کمبود وقت**). بخاطر داشته باشیم که پیاده کردن درست یک راه حل کند بهتر از ختم مصاحبه بدون پیاده سازی کامل یک الگوریتم است. مصاحبه کننده به ما اجازه می دهد که راه حل را پیاده کنیم.

بنابراین شروع به پیاده سازی می کنیم (**پیاده سازی**). ابتدا یک نام مناسب برای تابع و پارامترهای مناسب آن را در نظر می گیریم (خط ۱). سپس پارامتر ها و معنای آن را برای مصاحبه کننده توضیح می دهیم و نظر مصاحبه کننده را جویا می شویم. اولین پارامتر این تابع nums می باشد که یک لیست حاوی دنباله ای از اعداد است. با بکارگیری دو متغیر به عنوان اندیس، می توانیم هر دو عنصر ممکن از لیست را با هم جمع و با مقدار ویژه مقایسه کنیم . ساده ترین روش برای پیاده سازی استفاده از دو حلقه تو در توست (خطهای ۳ تا ۶). حلقه بیرونی (خط ۳)، عناصر لیست را به ترتیب از ابتدا تا انتها می پیماید. برای هر عنصر در حلقه بیرونی، حلقه درونی (خطوط ۴ تا ۶) تمام عناصر بعد از آن تا انتهای لیست را می پیماید. هر تکرار در حلقه درونی یک زوج ممکن از عناصر لیست را ایجاد میکند. با محاسبه مجموع این زوج مشخص می شود که آیا مجموع آندو برابر مقدار ویژه می شود. تکه کد زیر این روش را پیاده می کند.

```
1. def twoSum(nums, target):
2.     nums_len = len(nums)
```

```
3.     for i in range(nums_len):
4.         for j in range(i+1, nums_len):
5.             if nums[i] + nums[j]==target:
6.                 return [i , j]
```

سپس اجرای برنامه فوق را روی لیست ورودی مثال بالا به صورت دستی اجرا می کنیم. برای این کار می توانیم یک جدول روی تخته وایت برد رسم کنیم که هر سطر آن یک متغیر را نشان می دهد و با اجرای الگوریتم به صورت دستی مقادیر را بروز می کنیم. وقتی یک متغیر مقدار جدیدی می گیرد می توانیم روی مقدار قبلی خط بکشیم و مقدار جدید را جلوی آن بنویسیم. به هر حال الگوریتم را تا انتها به صورت دستی اجرا می کنیم تا مطمئن شویم که درست کار می کند.

ارائه راه حل بهتر

ممکن است که مهارت کافی در یافتن راه حل های بهینه داشته باشیم. با توجه به اینکه الگوریتم جستجوی کامل فوق روی یک لیست طولانی کند است، به فکر کردن برای یافتن راه حل بهینه ادامه می دهیم (**ارائه راه حل بهتر**).

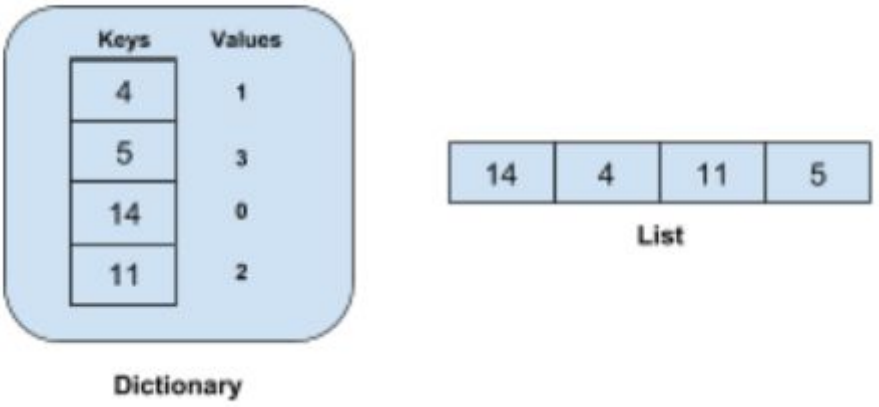
برای یافتن راه حل بهتر روش های متفاوتی وجود دارند. از جمله یافتن ایراد روش راه حل بهینه و یافتن قسمتی از الگوریتم که کارهای تکراری انجام می دهد. با اصلاح الگوریتم بگونه ای که تکرارهای اضافی انجام ندهد به الگوریتم بهتر می رسیم. روش دیگر یافتن تعابیر دیگر مسئله است که کمک می کند راه حل بهتری پیدا کنیم. در بدترین حالت می توانیم فکر کنیم که چگونه ساختمان های داده مختلف ممکن است کمک کنند که بتوانیم یک الگوریتم را سریعتر کنیم.

برای یافتن راه حل دیگر برای این مسئله به تعابیر دیگر این پرسش می اندیشیم. یک تعبیر پرسش فوق آن است که باید مقدار x ای را در لیست بیابیم که متمم آن (یعنی target-x) نیز در لیست وجود دارد. این تعبیر پیشنهاد می کند که برای حل این پرسش، به ازای هر عنصر با مقدار x در لیست، در بین مابقی عناصر داده شده مقدار target-x را جستجو کنیم. یک راه بدیهی آن است که کل لیست را هر بار از ابتدا تا انتها تک تک بررسی کنیم. اما این روش بسیار کند است. بجای آن می توانیم از ساختمان داده دیکشنری استفاده کنیم که بدون مقایسه با تک تک عناصر لیست سریعا می تواند بگوید که آیا یک مقدار در بین عناصر داده شده وجود دارد؟

بنابراین برای حل مسئله می توانیم ابتدا تنها یکبار لیست را پیمایش کنیم و مقادیر عناصر آن را به عنوان کلید و اندیس موقعیت آن را به عنوان مقدار به **دیکشنری(dictionary یا hashtable)** اضافه کنیم. سپس می توانیم از ابتدای لیست دوباره

شروع کنیم و هر برای هر عنصر با مقدار x در لیست بررسی کنیم که آیا target-x در دیکشنری وجود دارد؟ اگر متمم x در دیکشنری وجود داشت آنگاه اندیس x و target-x را به عنوان پاسخ برمی گردانیم. با توجه با اینکه عمل بررسی وجود یک کلید در یک دیکشنری بی درنگ (یعنی$O(1)$) است، الگوریتمی ارائه شده دارای پیچیدگی محاسباتی $O(n)$ خواهد بود. همچنین برای مصاحبه کننده شرح می دهیم که بخاطر بکارگیری دیکشنری، این الگوریتم n خانه حافظه بیشتر در مقایسه با الگوریتم جستجوی کامل استفاده می کند.

برای اینکه از درستی عملکرد برنامه مطمئن شویم اجرای آن روی مثال فوق را دنبال می کنیم. در مرحله اول با پیمایش لیست دیکشنری را تشکیل می دهیم (**آزمایش دستی با داده ورودی**).



وضعیت دیکشنری بعد از یک دور خواندن عناصر آرایه

سپس از ابتدای لیست دوباره شروع می کنیم. با دیدن مقدار 14 به عنوان مثال, وجود متمم آن یعنی 5- را دیکشنری وارسی می کنیم. این مقدار در دیکشنری وجود ندارد. بنابر این پاسخی شامل 14 نداریم. سپس سراغ عنصر بعدی یعنی 4 می رویم. سپس متمم آن یعنی 5 را در دیکشنری وارسی می کنیم. با توجه به اینکه 5 در دیکشنری هست، پاسخ را یافته ایم. مقدار مربوط به عنصر 5 در دیکشنری اندیس موقعیت آن را در لیست اولیه مشخص می کند. بنابر این اندیس مربوط به عنصر 4 (یعنی 1) و مقدار مربوط به کلید 5 در دیکشنری (یعنی 3) پاسخ می باشند. بنابر این الگوریتم مقدار [3, 1] را برمیگرداند.

حال شروع به پیاده سازی الگوریتم میکنیم (**پیاده سازی**). این الگوریتم را می توانیم بگونه ای پیاده سازی کنیم که دو مرحله الگوریتم یعنی ساختن دیکشنری و پیمایش مجدد لیست را باهم ترکیب نماید. یعنی در زمان پیمایش لیست، ابتدا دیکشنری را وارسی می کنیم. در صورتی که پاسخ یافت نشد عنصر جدید را به همراه اندیسش (به عنوان زوج کلید و مقدار) به دیکشنری اضافه می کنیم و به پیمایش در لیست به همین طریق ادامه می دهیم. ابتدا نام مناسب به تابع و پارامترها می دهیم (خط 1).

سپس یک دیکشنری خالی تعریف می کنیم (خط 2). آنگاه در یک حلقه تکرار در لیست پیش می رویم (خط 3 تا 7). هر وارسی می کنیم که آیا متمم عنصر iام در دیکشنری وجود دارد؟ در صورت وجود پاسخ را یافته ایم و اندیسهای مربوط به جواب مقدار عنصر متمم در دیکشنری و i خواهند بود (خط 4 و 5). در غیر این صورت عنصرi ام را به دیکشنری اضافه می کنیم (خط 7) و به کار ادامه می دهیم.

```
1. def twoSum(nums, target):
2.     h = {}
3.     for i in range(len(nums)):
4.         if target-nums[i] in h:
5.             return [h[target-nums[i]], i]
6.         else:
7.             h[nums[i]] = i
```

در انتها برنامه را دوباره مرور می کنیم و با داده های ورودی تست می کنیم. همانگونه که گفته شد، الگوریتم فوق دارای پیچیدگی محاسباتی $O(n)$ می باشد.

این مثال به خوبی نشان می دهد که چگونه تفکر مرحله به مرحله در حل یک مسئله راه گشاست و کارایی آن را برای توضیح نحوه فکر کردن شما را به خوبی نشان می دهد.

۳. لیست (List)

۱.۳ لیست در پایتون (List)

لیست یک ساختمان داده است که دنباله ای از مقادیر را در خود نگه می دارد. هر عنصر لیست با استفاده از اندیس آن قابل دسترسی است. به عنوان مثال به عنصر $(i+1)$-ام لیست a با $a[i]$ دست می یابیم.در زبان برنامه نویسی پایتون لیست و آرایه معادل هم در نظر گرفته می شوند. بجز تفاوت در نحوه پیاده سازی، تفاوت اصلی لیست با آرایه آن است که طول لیست می تواند تغییر کند. به عنوان مثال می توان عنصری را به انتهای یک لیست اضافه نمود یا عنصری را از آن حذف کرد.

لیست در حل بسیاری از مسائل کاربرد دارد. مثلا اگر بخواهیم افزایش یا کاهش ارزش سهام یک شرکت را در ۱۰ روز آینده بررسی کنیم، بهترین راه استفاده از لیست است: در این صورت مقدار ارزش سهم در هر روز یک عنصر از لیست را تشکیل می دهد و توسط اندیس آن روز به صورت مستقیم و بدون پیمایش همه داده ها فوری قابل دسترسی است.

نحوه کار با لیست در پایتون

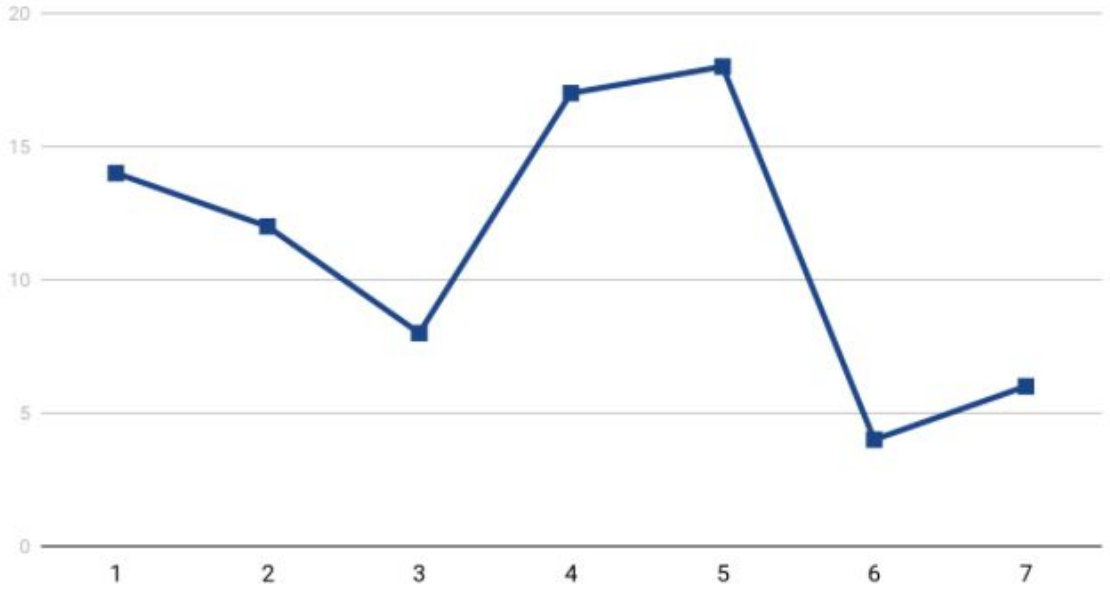
تعریف لیست خالی	`a = []`
تعریف لیست خالی -- روش دوم	`a = list()`
تعریف لیست به همراه تعدادی مقادیر	`a = [value1, value2, value3]`
تعریف لیست بطول n و مقدار دهی اولیه عناصر با صفر	`a = [0 for i in range(n)]`
بازیابی یک مقدار توسط اندیس	`value = a[index]`
بازیابی آخرین مقدار لیست	`value = a[-1]`
بازیابی یک زیر لیست از اندیس i تا j	`b = a[i: j]`
اضافه کردن یک عنصر به انتهای لیست	`a.append(value)`
بدست آوردن طول لیست	`len(a)`
تکرار یک حلقه روی همه ی عناصر لیست (با استفاده از اندیس)	`for i in range(len(a)):` `  ...`
تکرار یک حلقه روی همه ی عناصر لیست	`for value in a:` `  ...`
تکرار یک حلقه روی همه عناصر لیست (همزمان با دسترسی به اندیس لیست)	`for i, value in enumerate(a):` `  ...`
چسباندن دو لیست به هم	`a + b`

۲.۴ . خرید و فروش یک سهم با بیشترین سود ممکن

پرسش: لیستی از ارزش هر سهم یک موسسه در یک بازه زمانی داده شده است. با فرض اینکه تنها می توانید یک سهم این موسسه را خرید و فروش کنید، برنامه ای بنویسید که بیشترین سود ممکن از این کار را برگرداند.

پاسخ: ابتدا نیاز داریم که پرسش را بهتر درک کنیم (**فهم مسئله**). با آوردن یک مثال باید مطمئن شویم که منظور مصاحبه کننده را درست متوجه شده ایم. این بسیار مهم است که مسئله درست را حل کنیم.

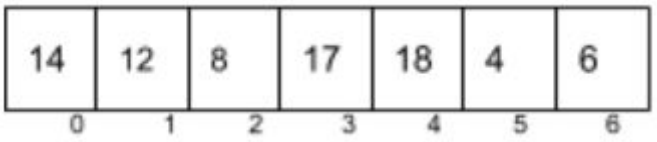
فرض کنیم که قیمت سهام یک شرکت فرضی در تعدادی روز متوالی به شکل زیر است.



ارزش سهام شرکت بین روزهای اول تا هفتم

در این مثال اگر سهم را در روز اول بخریم و همان روز بفروشیم هیچ سودی نکرده ایم. اگر آن را در روز اول بخریم و در روز دوم بفروشیم در واقع دو واحد پولی (ریال - دلار بسته به واحد محور y) ضرر کرده ایم. اگر سهم را روز دوم بخریم و روز چهارم بفروشیم ۵ واحد سود کرده ایم. بیشترین سود از خرید در روز سوم و فروش در روز پنجم بدست می آید. در این صورت مقدار سود 8-18 یا به عبارتی 10 می باشد.

داده های نمودار فوق را همچنین می توانیم در یک لیست مانند لیست زیر قرار دهیم.



نمایش نمودار بالا در قالب یک لیست

در یک مصاحبه، اولین راه حلی که باید مطرح کنیم روش جستجوی کامل است (**راه حل جستجوی کامل**). جستجوی کامل برای یافتن پاسخ این مسئله یعنی همه حالات خرید-فروش را، با در نظر گرفتن هر یک از روزها برای خرید و هریک از روزهای بعد آن برای فروش، بدست آوریم. برای این کار می توانیم از دو حلقه تو در تو استفاده کنیم که شمارنده حلقه بیرونی اندیس روز خرید (متغیر i) و شمارنده حلقه درونی اندیس روز فروش (متغیر j) در لیست را مشخص می کنند. در هر تکرار حلقه بیرونی سراغ یک عنصر می رویم (روز خرید) و برای هر چنین عنصری، با استفاده از حلقه درونی از عنصر بعد آن (i+1) تا آخر لیست (n-1) را تک تک به عنوان روز فروش در نظر میگیریم. برای هر حالت خرید-فروش بوجود آمده، میزان سود را محاسبه می کنیم. خرید-فروشی که بیشترین سود را بدهد پاسخ مسئله است. در یک مصاحبه واقعی این روش را به صورت دستی امتحان میکنیم تا مطمئن شویم که الگوریتم درست عمل می کند.

برای محاسبه سرعت این الگوریتم (یا بالعکس، پیچیدگی محاسباتی آن) ، باید در نظر بگیریم که هر روز با هر یک از روزهای بعدش یک حالت خرید-فروش را می سازد. اگر تعداد روزهایی که داده های سهام را در آن داریم n باشد آنگاه n*(n-1)/2 حالت خرید-فروش را باید امتحان کنیم. بنابر این پیچیدگی محاسباتی این الگوریتم $O(n^2)$ می باشد.

الگوریتم جستجوی کامل یک قدم درست برای حل این مسئله است. اما مصاحبه کننده از ما میخواهد که راه حل سریع تری بیابیم.

برای پیدا کردن راه حل بهتر باید اندیشیم که راه حل کنونی در واقع چگونه کار می کند و سعی کنیم دلیل کند بودن آن را بیابیم و اصلاح کنیم. با تفکر در مورد نحوه کار الگوریتم جستجوی کامل ارایه شده متوجه می شویم که این الگوریتم در واقع هر بار عنصر ماکزیمم (بزرگترین عنصر) بعد از هر عنصر (بین عناصر i+1 تا n-1) را می یابد. به این ترتیب حداکثر سود ممکن از خرید در هر یک از روزها را بدست می آورد.

با مشاهده روش یافتن عنصر ماکزیمم متوجه می شویم که حلقه دورنی در واقع تکرارهای زیادی را برای یافتن عنصر ماکزیمم بعد از یک عنصر انجام می دهد. این تکرارها چیزی است که کل الگوریتم را n بار (n طول لیست را نشان می دهد) کند می

کند. این در حالی است که پس از یافتن ماکزیمم بین عناصر بعد از i (یعنی بین i+1 تا n-1) و برای یافتن ماکزیمم بین عناصر بعد از عنصر i+1 در دور بعد، باید بین عناصر بعدی آن یعنی i+2 تا n-1 به دنبال عنصر ماکزیمم بگردیم. در واقع قسمت عمده ای از لیست که در آن عنصر ماکزیمم قبلی را یافتیم تغییر نمی کند بلکه تنها یک عنصر از ابتدای آن کم می شود (عنصر i+1 را کم می کنیم).

در عین حال متوجه می شویم که راه حل سرراستی برای بدست آوردن مقدار ماکزیمم جدید (بدون محاسبه مجدد) پس از حذف یک عنصر از ابتدای یک لیست نداریم (**یافتن ایراد راه حل**). در عوض می دانیم که عکس آن یعنی افزودن یک عنصر به لیست و بدست آوردن مقدار ماکزیمم یا مینیمم پس از آن سر راست و سریع است: در واقع برای یافتن ماکزیمم جدید تنها کافیست مقدار ماکزیمم بین ماکزیمم قبلی و عنصر جدید را محاسبه کنیم.

بنابراین برای حذف کارهای تکراری در حلقه درونی باید الگوریتم را بگونه ای تغییر دهیم که بجای حذف عناصر و بروزرسانی ماکزیمم، شامل افزودن یک عنصر و محاسبه ماکزیمم (یا مینیمم) باشد. برای این کار می توانیم الگوریتم را به این صورت عوض کنیم: از ابتدای لیست هر بار یک روز D را به عنوان روز **فروش** انتخاب می کنیم و بین روزهای **گذشته** آن مینیمم قیمت سهام را بدست می آوریم (**ارائه راه حل بهتر**). این حالت خرید-فروش پرسودترین معامله ممکن است اگر که فروش در روز D انجام شود.

به عنوان مثال با نگاهی به نمودار قیمت ها می فهمیم که اگر بخواهیم روز چهارم سهم را بفروشیم بهترین زمان برای خرید روز سوم است که قیمت هر سهم (8) کمترین قیمت سهم در روزهای گذشته است. محاسبه برای روز بعد راحت است زیرا اگر بخواهیم در روز پنجم سهم را بفروشیم لزومی ندارد که همه روزهای قبل را برای پیدا کردن مقدار کمینه بپیماییم. بلکه باید مینیمم بین 8 (مینیمم کنونی) و قیمت سهام در روز چهارم (17) را بیابیم. بنابراین نیازی به پیمایش دوباره کل عناصر لیست از ابتدا تا روز پنجم را نیست.

پس کافیست با در نظر گرفتن هر روز به عنوان روز فروش، مینیمم قیمت سهام در روزهای گذشته را بدانیم تا بتوانیم بیشینه سود ممکن را محاسبه کنیم. اگر این میزان را برای هر یک از روز های محاسبه نماییم آنگاه مقدار بیشینه بین این مقادیر جواب مورد نظر خواهد بود (**الگوریتم جدید**). در مصاحبه یک دور کامل این الگوریتم را روی داده ورودی تست می کنیم تا مطمئن شویم که الگوریتم درست است.

برای تحلیل سرانگشتی پیچیدگی محاسباتی این الگوریتم می توانیم به این نکته توجه کنیم که الگوریتم ما یک حلقه دارد که به تعداد روزهایی که قیمت سهام را داریم تکرار می شود. برای هر تکرار، باید مینیمم قیمت در روزهای قبلی را بیابیم. برای این کار کافی است که مینیمم کنونی را با قیمت سهام در روز گذشته مینیمم بگیریم. هزینه انجام این کار ثابت است (با نماد $O(1)$ نشان داده می شود) و بستگی به تعداد عناصر لیست ندارد. بنابراین اگر طول این لیست n باشد، پیچیدگی محاسباتی این الگوریتم $O(n)$ می باشد.

سپس شروع به نوشتن برنامه می کنیم. تابعی با نام مناسب و یک پارامتر از نوع لیست تعریف می کنیم (خط 2). یک متغیر به نام pmin برای نگهداری کمترین قیمت سهام با مقدار اولیه بینهایت (بعبارت دقیقتر بزرگترین مقدار صحیح در پایتون) در نظر میگیریم (خط 3). یک متغیر دیگر به نام profit برای نگهداری بیشینه سود با مقدار اولیه صفر در نظر میگیریم (خط 4). سپس با استفاده از یک حلقه تکرار در هر تکرار میزان مینیمم کنونی را بروز می کنیم. برای این کار مینیمم بین مقدار جدید و مینیمم کنونی را محاسبه میکنیم (خط 7). آنگاه سود حاصل خرید یک سهم در روز i ام را با فرض خرید با قیمت کمینه در روزهای گذشته محاسبه می نماییم. این مقدار بیشینه سود در صورتی که سهم را در روز iام بفروشیم بدست می آید. بیشینه این مقدار را با بیشینه سود ممکن در روزهای گذشته محاسبه می کنیم و به عنوان میزان بیشینه سود در نظر میگیریم (خط 8). پس از اتمام حلقه تکرار میزان بیشینه سود ممکن را برمی گردانیم (خط 10). تکه کد زیر پیاده سازی الگوریتم بهینه را نشان می دهد.

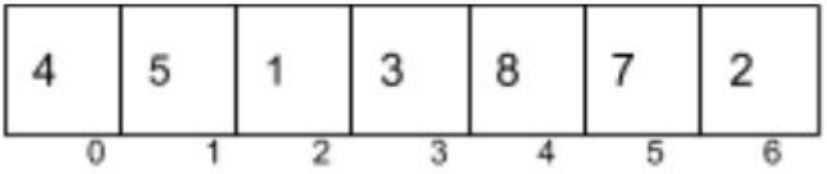
```
1. import sys
2. def maxProfit(prices):
3.     pmin = sys.maxint
4.     profit = 0
5.
6.     for i in range(len(prices)):
7.         pmin = min(prices[i], pmin)
8.         profit = max(profit, prices[i]-pmin)
9.
10.    return profit
```

در یک مصاحبه واقعی برنامه را یکبار دیگر مرور می کنیم و با داده های ورودی تست می کنیم که مطمئن شویم که درست کار می کند.

۳.۴ عدد گم شده را بیابید

پرسش: تعداد n-1 عدد غیر تکراری با مقادیر 1 تا n داده شده اند. برنامه ای بنویسید که عددی از بازه 1 تا n که در بین این n-1 عدد نیست (عدد گمشده) را بیابد.

پاسخ: کار را با یک مثال شروع میکنیم تا مطمئن شویم که مسئله را درست فهمیده ایم (**فهم مسئله**). مثلا اگر 7 عدد غیر تکراری در بازه 1 تا 8 به صورت زیر داشته باشیم، عدد گمشده بین آنها 6 می باشد.

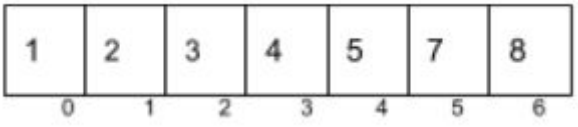


لیستی از 7 مقادیر در بازه 1 تا 8

برای یافتن پاسخ مسئله ابتدا روش جستجوی کامل را بررسی می کنیم (**راه حل جستجوی کامل**). برای این کار با استفاده از یک حلقه تکرار هر بار به دنبال یکی از اعداد 1 تا n درون لیست اعداد می گردیم. مقداری که در لیست پیدا نشود جواب مورد نظر می باشد. در بدترین حالت عدد گم شده n است که در این صورت در آخرین تکرار حلقه یافت می شود. از آنجا که در هر تکرار باید مقداره همه عناصر لیست را مقایسه نماییم، تعداد کل مقایسه ها n*(n-1) می باشد. بنابراین پیچیدگی محاسباتی این الگوریتم $O(n^2)$ می باشد.

این روش کند است و روی یک لیست طولانی بسیار زمان می برد. اشکال این روش آن است که هر بار کل لیست را می پیماید و از مشاهدات در دورهای قبلی استفاده ای نمیکند. در عین حال به خاطر نامرتب بودن داده ها نمی توان روش جستجوی کامل را مستقیما بهبود داد (**یافتن ایراد راه حل**).

بنابراین در ادامه برای راه حل بهتر بررسی می کنیم که آیا اگر لیست را مرتب کنیم الگوریتم تندتر می شود؟ با مرتب کردن لیست فوق به یک لیست جدید مانند لیست زیر میرسیم.



با دقت در لیست فوق می فهمیم که تا قبل از مقدار گم شده، هر عدد i در موقعیت i-1 در لیست قرار گرفته باشد (اندیس لیست از صفر شروع می شود). بنابراین برای یک راه حل بهتر می توانیم لیست اعداد را مرتب کنیم. سپس از یک حلقه تکرار که از ابتدای لیست (اندیس 0) تا انتهای آن (اندیس n-2) استفاده می کنیم. اگر عدد i+1 در اندیس i نبود آنگاه مقدار گمشده را (یعنی i+1) یافته ایم (**ارائه راه حل بهتر**). تکه کد زیر این الگوریتم را پیاده می کند.

```
1. def findMissing(nums, n):
2.    nums = sorted(nums)
3.    for i in range(1, n):
4.         if nums[i-1] != i:
5.              return i
6.    return n
```

برای محاسبه پیچیدگی محاسباتی این الگوریتم، یک اشتباه رایج این است که فراموش کنیم که محاسبات انجام شده در تابع sort (خط 2) را در نظر بگیریم و فکر کنیم چون این الگوریتم تنها دارای یک حلقه است که n-1 بار تکرار می شود، پیچیدگی محاسباتی کل برنامه O(n) است. اما در واقع بخاطر اینکه خود تابع sort دارای پیچیدگی محاسباتی (O(n*log(n))می باشد، کل برنامه نیز با سرعت O(n*log(n)) اجرا می شود.

راه حل با استفاده از مرتب سازی به اندازه ای که فکرش را میکردیم سریع نیست. در ادامه تفکر برای یافتن راه حل خطی (یعنی O(n)) را ادامه می دهیم.

برای پیدا کردن روش بهینه کافی است به این نکته توجه کنیم که یک فرمول برای محاسبه مجموع همه مقادیر از 1 تا n وجود دارد. این فرمول S=n * (n+1)/2 می باشد. از آنجا که عدد گم شده در این حاصل جمع وجود ندارد، می توان آن را بر اساس تفاوت بین S و مجموع عناصر لیست پیدا کرد (**ارائه راه حل بهتر**). تکه کد زیر پیاده سازی این روش را نشان می دهد.

```
1. def findMissing(nums, n):
2.    return n * (n+1) / 2 - sum(nums)
```

تابع sum (خط 2) مجموع عناصر لیست را محاسبه می کند. پیچیدگی محاسباتی این الگوریتم برابر با پیچیدگی تابع sum است که خطی O(n) می باشد.

۵. رشته (String)

۱.۵ رشته در پایتون (String)

ساختمان داده رشته روی لیست سوار است. همه کارهایی که با لیست می شود انجام داد روی رشته هم انجام می شود. تنها تفاوت آن است که مقدار یک کاراکتر در رشته را نمی توان عوض کرد (immutable). علاوه بر این، کارهای ویژه ای روی این نوع خاص از لیست متداول است که در نتیجه در درون آن پیاده سازی شده اند تا کار را برای برنامه نویسان راحت کنند. به عنوان مثال upper یک متد است که متغیرهای از نوع رشته دارند و تمام حروف رشته را به حرف بزرگ انگلیسی تبدیل می کند.

نحوه کار با رشته در پایتون

تعریف رشته	a = ""
تعریف رشته با مقدار اولیه	a = "Hello World!"
جایگزینی یک زیر رشته با رشته دیگر	a.replace("o", "u")
شکستن یک رشته در کاراکتر خاص (مثلا کاما)	"word1, word2".split(",")
چسباندن دو رشته به هم	"string 1" + "string 2"
طول رشته	len(a)
حذف فاصله های اضافی درون، ابتدایی، و انتهایی رشته	a.strip()
تبدیل رشته به لیست (هر کاراکتر یک عنصر لیست می شود)	b = list(a)
دسترسی به کاراکتر iام رشته	a[i]
گرفتن زیر رشته از اندیس start تا end	a[start: end]
مقایسه دو رشته	str1 == sta2 str1>=str2 str1<=str2 ...

۲.۵ نسخه های یک نرم افزار را مقایسه کنید

پرسش: برنامه ای بنویسید که دو رشته که نشان دهنده نسخه های یک نرم افزارند را در یافت نماید. اگر اولین رشته نسخه جدید تری باشد مقدار 1، چنانچه دومی جدیدتر باشد مقدار 1-، و در صورت مساوی بودن 0 را برگرداند.

پاسخ: کار را با چند مثال شروع میکنیم (**فهم مسئله**). اگر یک نسخه نرم افزاری "1.02" و نسخه دیگر آن "1.02.1" باشد آنگاه بخش آخر رشته دوم (یعنی ".1") نشان می دهد که ورژن "1.02.1" مربوط به ویرایش اول از نسخه "1.02" است و جدیدتر است. بنابر اگر "1.02" ورودی اول این برنامه و "1.02.1" ورودی دوم باشد این تابع باید مقدار 1- را برگرداند. اگر رشته اولی "1.2" باشد باز هم نتیجه 1- است زیرا صفر در بخش دوم (یعنی 02) در مقدار عددی ورژن تاثیری ندارد. اگر رشته دومی "1.02.0" باشد در واقع این دو ورژن یکی هستند.

با توجه به مثال های بالا واضح است که استفاده از عملگرهای مقایسه ای (یعنی ==, < , >) پاسخ صحیح را برای این سوال نمی دهد، زیرا مقایسه رشته ای حالتهای شامل 0 که در مثالها گفتیم را به درستی مدیریت نمی کنند (**یافتن ایراد راه حل**).

بنابراین یک راه حل درست باید مقادیر عددی بخش های متناظر از دو رشته را با هم مقایسه نماید نه خود رشته ها را (**ارائه راه حل بهتر**). به عنوان مثال، رشته "1.2" را می توانیم به صورت لیستی از مقادیر صحیح [2, 1] در نظر بگیریم و رشته "1.02.1" را به صورت یک لیست از مقادیر صحیح [1, 2, 1] در نظر داشته باشیم. با مقایسه لیست ها از چپ به راست اولین نقطه ای که دو لیست با هم متفاوت باشند مشخص می کند که کدام یک بزرگتر است. در صورتی که دو لیست تفاوتی نداشته باشند به این معنا است که دو نسخه نرم افزار با هم برابرند.

شروع به پیاده سازی الگوریتم می کنیم و همزمان با نوشتن برنامه باید چیزهایی که در ذهنمان می گذرد را برای مصاحبه کننده توضیح دهیم (**فکرمان را داد بزنیم**). مثلا <<یک تابع با پارامترهای مناسب تعریف می کنیم. این تابع نیاز به دو پارامتر برای گرفتن ورژنهای ورودی دارد (خط 1). سپس از تابع split استفاده می کنیم که بخشهای هر ورژن را بدهد (خط 2 و 3). البته اینجا می توانستیم قبل این کارها چک کنیم که آیا مقدار پارامتر version1 تهی (None) هست.>>

می توانیم از مصاحبه کننده بپرسیم که آیا لازم هست که این شرط را بررسی کنیم. این سوال ما نشان می دهد که ما حالتهای ورودی در یک برنامه واقعی را می شناسیم و مطلعیم که ممکن است ورودی شکل مورد نظر ما را نداشته باشد. بررسی این

حالت خیلی ساده است و پیشنهاد می کنیم که آن را اضافه کنیم. در اینجا برای اینکه برنامه کوتاه و قابل فهم باشد این شرط را نیاورده ایم.

<<پس از آنکه بخشهای ورژن را در آوردیم باید آنها را به عدد صحیح تبدیل کنیم. برای این کار از تابع map استفاده می کنیم که در واقع تک تک مقادیر لیست (بخش های نسخه) را به عدد صحیح تبدیل می کند (خط 2). این کار را برای رشته دوم هم انجام می دهیم (خط 3).>>

حالا باید دو لیست را با هم مقایسه کنیم. برای این کار میتوانیم یک حلقه تکرار بنویسیم که سراغ تک تک عناصر لیست می رود تا جایی که (۱) یکی از مقادیر متناظر مقایسه شده از دیگری بزرگتر باشد که در این صورت نتیجه را می دانیم، یا (۲) یا به انتهای لیست کوچکتر برسیم. حالت دوم به این معناست که همه مقادیر دو لیست تا اینجای کار با هم مساوی می باشند. شاید بنظر برسی که پس از خروج از حلقه باید اندازه لیست ها را با هم مقایسه کنیم تا لیست بزرگتر را به عنوان نسخه بزرگتر اعلام کنیم اما در مورد این مسئله خاص باید این را در نظر بگیریم که لیست طولانی تر الزاما بزرگتر نیست. به این دلیل که ممکن است باقی مانده عناصر لیست بزرگتر همگی صفر باشند. مثلا لیست های مرتبط با رشته های "1.2" و "1.2.0" عبارتند از [2, 1] و [0, 2, 1]. با وجود اینکه تا عنصر اندیس 1 این دو لیست با هم برابرند و لیست دوم طولانی تر است، به این دلیل که مقادیر از اندیس 1 به بعد در لیست دوم صفر است، تفاوتی را در نسخه نرم افزار ایجاد نمی کند. بنابراین دو لیست با هم برابرند.

برای اینکه این مشکل را حل کنیم می توانیم چک کنیم که آیا همه عناصر باقیمانده صفر می باشند. این کار برنامه ما را اندکی کثیف و ناخوانا می کند.

برای اینکه از بررسی شرط های اضافی خلاص شویم، از یک تکنیک خوب استفاده می کنیم: در این تکنیک دو لیست را ابتدا هم اندازه می کنیم و سپس آنها را مقایسه می کنیم. بنابراین در خط 5 ابتدا تفاوت طول در لیست را محاسبه می کنیم. سپس با استفاده از امکانات زبان پایتون به راحتی تعداد صفر مناسب به انتها رشته ها اضافه می کنیم (خطوط 6 و 7). در انتها دو لیست را مستقیما با عملگرهای مقایسه ای مقایسه می کنیم (خطوط 9 و 11) و نیازی به نوشتن حلقه تکرار برای مقایسه نخواهیم داشت. نهایتا در خط نتیجه را برمی گردانیم.

```
1. def compareVersion(version1, version2):
```

```
2.     v1 = map(int, version1.split("."))
3.     v2 = map(int, version2.split("."))
4.
5.     d  = len(v1)-len(v2)
6.     v1 += [0] * (-d)
7.     v2 += [0] * (d)
8.
9.     if v1 == v2:
10.      return 0
11.    return 1 if v1 > v2 else -1
```

تحلیل سرعت برنامه: یک اشتباه رایج آن است که تصور کنیم این برنامه دارای پیچیدگی محاسباتی O(1) است به این خاطر که هیچ حلقه یا فراخوانی بازگشتی ای وجود ندارد. اما باید هزینه توابع موجود و از پیش تعریف شده در زبان برنامه نویسی را نیز جزو هزینه برنامه خود محاسبه کنیم. برنامه بالا از map، split و مقایسه لیست ها استفاده می کند که دارای پیچیدگی محاسباتی O(n) است که در آن n مجموع طول دو رشته است. بنابر این پیچیدگی محاسباتی این الگوریتم O(n) می باشد.

۲.۵ ترتیب کلمات را برعکس کنید

پرسش: برنامه ای بنویسید که ترتیب کلمات درون یک رشته را برعکس می کند.

پاسخ: کار را با چند مثال شروع می کنیم. اگر رشته ورودی "Hello world" باشد آنگاه برنامه باید مقدار "world Hello" را برگرداند.

با کمی فکر کردن متوجه می شویم که اگر یک لیست از کلمات متوالی درون رشته بسازیم و لیست را برعکس کنیم آنگاه ترتیب مطلوب کلمات در لیست بدست می آید. با وصل کردن کلمات درون لیست می توانیم رشته مورد نظر را بسازیم. قطعه کد زیر این کار را انجام می دهد.

```
1. def reverseWords(s):
2.   " ".join(reversed(s.split()))
```

اما مصاحبه کننده از ما میخواهد که رشته ورودی "Hello World" را روی تست کنیم. با دنبال کردن برنامه با داده ورودی، خروجی"World Hello" تولید می شود که پاسخ مطلوب نیست زیرا چندین کاراکتر فاصله بین دو کلمه از بین رفته اند. این اتفاق به دلیل نحوه کار تابع split رخ می دهد که رشته را از نقطه فاصله می شکند ولی فاصله ها را در حاصل دخیل نمی کند.

اگر بخواهیم این الگوریتم را تصحیح کنیم باید از روشی برای جدا کردن کلمات استفاده نماییم که برخلاف split، فاصله ها را نیز در لیست خروجی در اختیار بگذارد. خوشبختانه راه حلی برای این کار وجود دارد و آن استفاده از تابع split درون کتابخانه عبارات با قاعده (regular expression) برای جدا کردن رشته در نقاط فاصله است. این روش به ازای هر فاصله های یک رشته خالی در لیست حاصل ایجاد می کند. ممکن است در زمان پاسخگویی نحوه استفاده از عبارات با قاعده را ندانیم اما مطرح کردن این گزینه ،حتی اگر نتوانیم آن را پیاده کنیم، برای مصاحبه کننده نشان دهنده آن است که شما ذهنی باز دارید و گزینه های مختلف برای حل مسئله را در نظر میگیرید. این یکی از مهمترین خصوصیات یک برنامه نویس خوب است.

اگر هم که حضور ذهن برای نوشتن عبارات با قاعده داشته باشیم می توانیم جواب درست زیر را پیاده کنیم.

```
1. import re
```

```
2. def reverseWords(s):
3.  " ".join(reversed(re.split('\s', s)))
```

اما اگر ایده استفاده از عبارت با قاعده به ذهنمان نرسید یا مصاحبه کننده از ما بخواهد که از این امکان استفاده نکنیم، باید یک ایده دیگر بزنیم و مسئله را به روش دیگری حل کنیم.

با دقت در ورودی و خروجی مثال متوجه می شویم که خروجی در واقع معادل با رشته ایست که نسبت به رشته اول برعکس شده و هر کلمه نیز پس از آن برعکس شده است. مثلا با عکس کردن رشته "Hello World" به رشته "dlroW olleH" می رسیم. سپس با برعکس کردن هریک از کلمات در جای خود به رشته نهایی "World Hello" دست خواهیم یافت. بنابر این الگوریتم درست کار می کند.

با اندکی تامل بیشتر متوجه می شویم که برعکس کردن رشته عملی است که در هر دو مرحله الگوریتم انجام می شود. بنابر این در حل برنامه تابعی بنویسیم که بتواند یک رشته را برعکس کند. برای اینکه بتوانیم عمل برعکس کردن را روی کلمه ها هم انجام دهیم باید تابع را به صورتی بنویسیم که بتواند هر زیر رشته دلخواه بین اندیس های داده شده از یک رشته را در محل برعکس کند.

با این ذهنیت شروع به نوشتن برنامه می کنیم. همانطور که برنامه را می نویسیم خطوط برنامه را برای مصاحبه کننده توضیح می دهیم. ابتدا نام مناسبی برای تابع در نظر میگیریم. این تابع یک رشته را به عنوان ورودی می گیرد. در ادامه ابتدا رشته را به لیستی از کاراکتر ها تبدیل می کنیم (خط 2). این کار را به این دلیل انجام می دهیم که در زبان پایتون رشته یک ساختمان داده تغییر ناپذیر (immutable) می باشد. در ادامه یک تابع کمکی می نویسیم که اندیس چپ و راست یک زیر لیست را می گیرد و همه عناصر لیست در آن بازه را برعکس میکند (خطوط 4 تا 7). از این تابع هم برای برعکس کردن کل رشته (خط 9) و هم برعکس کردن کلمات (خطوط 10 تا 15) استفاده می کنیم. در انتها کاراکترهای لیست را به هم متصل می کنیم تا رشته را بسازیم.

```
1. def reverseWords(s):
```

```
2.    c = list(s)
3.
4.    def inverse_helper(left, right):
5.        while (left<right):
6.            c[left], c[right] = c[right], c[left]
7.            left, right = left+1, right-1
8.
9.    inverse_helper(0, len(s)-1)
10.   p = 0
11.   for i in range(len(s)):
12.       if c[i] == ' ':
13.          inverse_helper(p, i-1)
14.          p = i+1
15.   inverse_helper(p, len(s)-1)
16.   return "".join(c)
```

تحلیل سرعت برنامه: این برنامه عملا یکبار رشته را می پیماید تا آن را برعکس می کند و سپس کلمات درون رشته را با یکبار پیمایش آنها برعکس می کند. بنابر این پیچیدگی محاسباتی این الگوریتم $O(n)$ می باشد.

۳.۵ آیا پرانتزها منطبق هستند؟

پرسش: یک رشته از پرانتزهای باز و بسته داده شده است. برنامه ای بنویسید که مشخص کند آیا پرانتزهای باز و بسته با هم منطبق می باشند.

پاسخ: حل مسئله را با چند مثال شروع می کنیم تا مطمئن شویم آن را بطور کامل درک کرده ایم.در رشته "(())" حتی تعداد پرانتزهای باز و بسته یکی نیست، چه برسد به انطباق دوبدوی پرانتزهای باز و بسته. در رشته "(()(" تعداد پرانتزهای باز و بسته یکی است اما همچنان پرانتزها برهم منطبق نمی باشند. به عنوان مثال پرانتز اولیه بسته مطابق با هیچ پرانتز باز سمت چپ آن نیست. به عنوان مثال های مثبت، در رشته های "((()))()" و "(()((((()))" پرانتز های باز و بسته با هم منطبق می باشند.

در مورد این مسئله الگوریتم جستجوی کامل مفهوم روشنی ندارد. بنابر این بجای آن تلاش می کنیم که یک الگوریتم ساده ارایه نماییم که مستقیما مبتنی بر خاصیت ذاتی یک رشته پرانتز-منطبق می باشد. خاصیت ذاتی یک پرانتز بندی منطبق آن است که اگر یک زوج منطبق (یعنی '()') آن را حذف کنیم آنگاه رشته باقیمانده نیز پرانتزبندی منطبق دارد. همچنین اگر یک زوج پرانتز باز و بسته (یعنی '()') در یک رشته نامنطبق را حذف کنیم، آنگاه پرانتزهای رشته باقیمانده نامنطبق خواهند بود. مثلا با یک بار حذف یک زوج منطبق پرانتز رشته "(())(()" به رشته "(()(" می رسیم که همچنان یک رشته نامنطبق است. بنابراین یک الگوریتم مبتنی بر این خاصیت ابتدا تعداد پرانتزهای باز و بسته را می شماریم. اگر تعداد آنها متفاوت باشد نتیجه می گیریم که پرانتزها نامنطبق اند. در صورتی که تعداد پرانتزهای باز و بسته برابر باشند، در گام بعد هر بار یک زیر رشته "()" را از داخل رشته حذف می کنیم و این کار را تا زمانی که (۱) رشته تمام شود (برای رشته منطبق) یا، (۲) دیگر زیر رشته "()" پیدا نشود (در رشته نامنطبق) تکرار می کنیم. این الگوریتم در بدترین حالت دارای پیچیدگی محاسباتی $O(n \wedge 2)$ می باشد زیرا برای یافتن هر زیر رشته "()" باید بطور متوسط از ابتدا تا وسط رشته را بپیماییم. در این لحظه می توانیم الگوریتم را روی مثالهای داده شده تست کنیم تا مطمئن شویم که درست کار می کند.

با وجود این که الگوریتم درست است اما کند است. علت کندی این الگوریتم آن است که کارهای تکراری انجام می دهد. به خصوص اینکه برای پیدا کردن زوج پرانتز باز و بسته جدید هربار از ابتدای رشته دوباره شروع به جستجو میکند. برای رفع این مشکل الگوریتم را تغییر میدهیم بگونه ای که هر بار مجبور به جستجوی مجدد از ابتدای رشته نباشد. برای این کار از ابتدای رشته شروع به حرکت می کنیم تا اولین پرانتز بسته (یعنی ")") را در اندیس Y در رشته پرانتزها بیابیم. سپس به عقب حرکت

می کند تا اولین پرانتز باز (یعنی ")") قبل آن را در نقطه X بیابیم. به این دلیل که پرانتزهای در موقعیت X و Y زوج منطبق هستند دیگر کاری با آنها نداریم. در نتیجه یا آنها را از رشته حذف می کنیم یا جای آنها را با کاراکترهای الکی (غیر از پرانتز مثلا *) پر می کنیم تا دفعات بعد با آنها کار نداشته باشیم.

سپس از اندیس Y+1 کار را برای جستجوی پرانتز بسته (یعنی ")") بعدی ادامه می دهد. و این کار را تا منطبق کردن همه پرانتزهای بسته و باز انجام می دهد. به این ترتیب برای یافتن پرانتزهای بسته تنها باید یکبار به صورت کامل رشته را می پیماییم. برای یافتن پرانتزهای باز در بدترین حالت باید تمار کاراکترهای سمت چپ اندیس Y را تا ابتدای رشته بررسی کنیم. بنابر این چنین پیچیدگی محاسباتی این الگوریتم همچنان $O(n^2)$ خواهد بود.

قسمت کند الگوریتم فوق همان یافتن موقعیت پرانتز باز (یعنی ")") استفاده نشده ای است که سمت چپ یک پرانتز بسته در موقعیت Y است. اما با توجه به صورت مسئله، آیا واقعا نیاز به یافتن اندیس موقعیت این پرانتز باز داریم یا اینکه کافیست مطمئن شویم که یک پرانتز باز استفاده نشده در سمت چپ Y وجود دارد؟ پاسخ آن است که در این مسئله نیاز به برگرداندن لیست زوج پرانتز های منطبق نداریم و بنابر این کافی است که مطمئن باشیم که یک پرانتز باز استفاده نشده در سمت چپ Y هست اما لازم نداریم اندیس آن را بدانیم. این نکته راه را برای یافتن الگوریتم بهبود یافته باز می کند.

بنابراین تنها کافی است بدانیم که آیا حداقل یک پرانتز کافی منطبق نشده در سمت چپ یک پرانتز بسته در موقعیت Y وجود دارد؟ به عنوان مثال در رشته "(()))(()" با رسیدن به اولین پرانتز بسته در اندیس 1، یک پرانتز باز جهت تطبیق با آن در سمت چپش وجود دارد. در مرحله بعد سراغ پرانتز بسته بعدی (یعنی اندیس 2 در رشته) میرویم. در این صورت تعداد صفر پرانتز باز استفاده نشده در سمت چپ آن وجود دارد. بنابر این زوجی منطبقی برای پرانتز بسته در اندیس 2 وجود ندارد و در نتیجه پرانتزها رشته منطبق نیستند.

بنابراین یک شمارنده کافی است تا تعداد پرانتز های بسته استفاده نشده در سمت چپ نقطه Y را بدانیم. هرگاه یک پرانتز بسته مشاهده می کنیم یک پرانتز باز را استفاده می کنیم و در نتیجه یک واحد از مقدار شمارنده می کاهیم. با مشاهده یک پرانتز باز یک واحد به این شمارنده اضافه می کنیم. اگر در حین حرکت روی رشته با مشاهده پرانتز بسته در اندیس Y، مقدار این شمارنده صفر باشد، آنگاه نتیجه میگیریم که پرانتزهای رشته منطبق نیستند. همچنین اگر حرکت روی رشته تمام شود و در نهایت مقدار شمارنده بیش از صفر باشد نشان دهنده آن است که تعداد پرانتزهای باز بیش از پرانتز بسته می باشد و در نتیجه بازهم پرانتزهای رشته منطبق نیستند. در غیر این دو حالت، پرانتزها منطبقند.

با توجه با اینکه در این الگوریتم تنها یکبار هر کاراکتر رشته بطول n را مشاهده می کنیم، پیچیدگی محاسباتی آن O(n) می باشد.

در یک مصاحبه واقعی با اجرای الگوریتم روی رشته های مثال بررسی می کنیم که آیا الگوریتم می تواند به درستی پاسخ بدهد. سپس شروع به پیاده سازی الگوریتم می کنیم.

همزمان با نوشتن برنامه خطوط برنامه را توضیح می دهیم. ابتدا نام مناسبی برای تابع و ورودی آن انتخاب می کنیم (خط 1). سپس یک شمارنده برای تعداد پرانتزهای باز و یک شمارنده برای حلقه و موقعیت پیشروی در رشته در نظر میگیریم (خط 2). سپس روی کاراکترهای رشته با استفاده از یک حلقه تکرار حرکت می کنیم. تا زمانی که به انتهای رشته نرسیده ایم و تعداد پرانتزهای منطبق نشده بزرگتر مساوی (یعنی بزرگتر یا مساوی) صفر است به حرکت در حلقه ادامه می دهیم (خط 3). در ادامه متناسب با کاراکتر i ام رشته شمارنده count را یک واحد افزایش یا کاهش می دهیم. در واقع اگر این کاراکتر پرانتز باز باشد یک واحد به تعداد شمارنده پرانتزهای باز اضافه می کنیم. در صورتی که پرانتز بسته ای ببینیم در واقع باید یکی از پرانتزهای باز را استفاده کنیم و در نتیجه یکی از شمارنده پرانتزهای باز کم می کنیم (خط 4). این خط را با استفاده از ساختار شرطی یک خطی نوشته ایم اما می توانستیم آن را با ساختار شرطی معمولی هم بنویسیم. در مورد این مسئله ساختار شرطی یک خطی برنامه را خواناتر کرده است.

سپس با افزایش شمارنده حلقه آماده پردازش عنصر بعد رشته می شویم (خط 5). حلقه وقتی تمام می شود که یا به انتهای حلقه رسیده باشیم و یا اینکه شمارنده تعداد پرانتزهای باز کمتر از صفر شده باشد. این حالت در صورتی رخ می دهد که تعداد پرانتزهای باز در نقطه ای از اجرای حلقه صفر باشد ولی کاراکتر بعدی پرانتز بسته باشد. در این صورت شمارنده count برابر 1- خواهد شد. به هر حال اگر بعد از اتمام اجرای حلقه مقدار متغیر count صفر باشد آنگاه تطابق بین پرانتزهای وجود داشته است. اگر مقدار count بزرگتر از صفر باشد یعنی تعداد پرانتزهای باز رشته بیش از پرانتزهای بسته بوده است. اگر مقدار count کمتر از صفر باشد به این معنا است که در نقطه ای از رشته تعداد پرانتزهای باز سمت چپ پرانتز بسته ناکافی بوده است. به هر حال اگر مقدار صفر نباشد آنگاه مقدار False را برمی گردانیم. در صورتی که مقدار count صفر باشد True را برمی گردانیم. این یعنی not count پاسخ درست را می دهد (خط 6).

```
1. def parenthesesMatch(str):
2.    count, i = 0, 0
3.    while i < len(str) and 0<=count:
4.          count += 1 if str[i] == '(' else -1
5.          i += 1
6.    return not count
```

در یک مصاحبه برنامه نویسی، برنامه را دوباره مرور می کنیم تا از درستی آن مطمئن شویم. همچنین چندین داده ورودی را روی آن تست می کنیم.

۶. پشته (Stack)

۱.۶ پشته در پایتون (Stack)

پشته یک ساختمان داده است که روی لیست یا لیست پیوندی پیاده سازی می شود. نحوه دسترسی به عناصر پشته خاص است و به همین دلیل یک ساختمان داده جداگانه در نظر گرفته می شود. الگوریتم ها با بالای پشته سر و کار دارند: یا عنصر بالا پشته را بر می دارند و یا عنصری روی پشته می گذارند.

نحوه کار با پشته در پایتون

تعریف پشته	s = [ ]
گذاشتن روی پشته	s.append(value)
برداشتن از پشته	value = s.pop( )
دیدن مقدار بالای پشته	s[-1]
بررسی خالی نبودن پشته	if s: …

مثال: دنباله زیر تعامل با یک پشته را نشان می دهد.

```
s = []
s.append(“1st message”)
a.append(“2nd message”)
s.pop()
>>> “2nd message”
s.append(“3rd message”)
p.pop()
>>> “3rd message”
p.pop()
>> “1st message”
```

۲.۶ آیا کمانکها منطبق می باشند؟

پرسش: یک رشته از کمانکهای (شامل پرانتز، آکولاد، و کروشه) باز و بسته داده شده است(یعنی کاراکترهای ()[]{}). برنامه ای بنویسید که مشخص کند آیا کمانکهای باز و بسته منطبق می باشند.

پاسخ: کار را با یک مثال شروع می کنیم. در رشته "()[]{}" کمانکهای از هر نوع با هم منطبق می باشند. در رشته "{}[" کروشه تطابق ندارد. همچنین در رشته "[{]}" تطابق وجود ندارد زیرا قبل از بسته شدن کروشه باز ،یعنی کاراکتر"]"، در اندیس دوم رشته آکولاد بسته قرار دارد. خاصیتی از رشته های با کمانک منطبق می بینیم آن است که در آنها حداقل یک جفت کمانک متوالی (یعنی یک زوج () یا {} یا []) یافت می شود و اگر چنین کمانکی را از آن حذف کنیم، رشته باقیمانده دارای کمانک منطبق خواهد بود.

بنابراین راه حل ساده برای این مسئله آن است که هربار یک زوج کمانک را از رشته حذف کنیم. اگر با تکرار این کار به رشته خالی برسیم آنگاه رشته اولیه دارای کمانکهای منطبق بوده است. اگر رشته خالی نباشد و زوج کمانکی در آن یافت نشود آنگاه رشته اولیه دارای کمانکهای منطبق نبوده است. این الگوریتم به ازای هر جفت کمانک متوالی یک بار رشته را جستجو می کند. بنابر این پیچیدگی محاسباتی آن در بدترین حالت $O(n \wedge 2)$ می باشد.

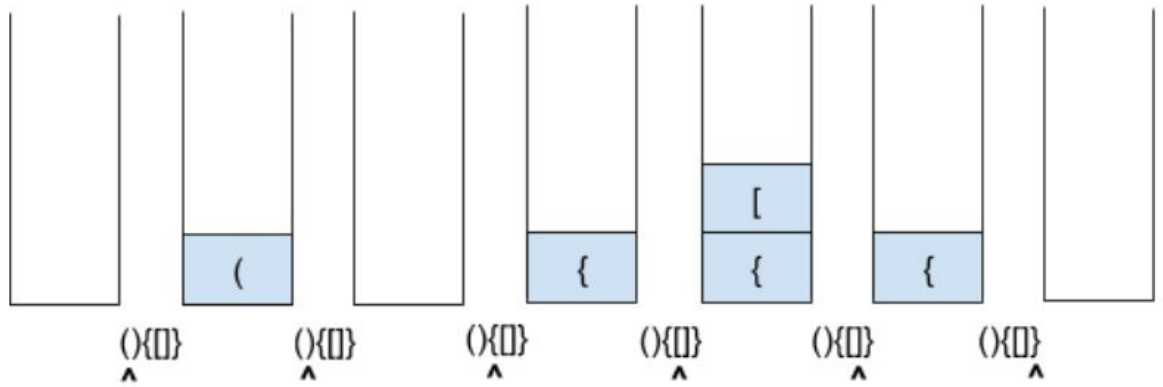
اشکال این روش جستجوی تکراری رشته برای یافتن جفت کمانک های متوالی است. بنابر این به راه حلی فکر می کنیم که از ابتدای رشته تنهای یکبار به سمت انتهای آن حرکت کند و کمانهای بسته شده را بیابد. این کار باید بگونه ای کمانک های باز منطبق نشده را بخاطر بسپارد و در صورت مشاهده کمانک بسته بررسی کند که تطابق وجود دارد یا خیر. برای بررسی کردن تطابق یک کمانک بسته همیشه باید تنها آخرین کمانک باز استفاده نشده را بیابیم. همچنین با مشاهده یک کمانک باز آن را به انتهای لیست کمانکهای باز اضافه می کنیم. اینگونه اضافه کردن و حذف کردن در لیست کمانک های باز شده رفتار <<First in Last Out>> ساختمان داده پشته را تداعی می کند. بنابراین از مفهوم ساختمان داده پشته برای حل این مسئله استفاده می کنیم.

در الگوریتم جدید از سمت چپ رشته شروع می کنیم. با دیدن هر کمانک باز آن را روی پشته می گذاریم. با دیدن هر کمانک بسته عنصر بالای پشته را بررسی می کنیم. اگر روی پشته کمانکی متناسب با کمانک بسته دیده شده باشد (یعنی ")" برای

"(" و "}" برای "{" و "]" برای "[") به این معناست که یک جفت منطبق دیده ایم. بنابر این کمانک باز بالای پشته را مصرف کرده و آن را حذف می کنیم. در صورتی که کمانک بالای پشته متناسب نباشد یعنی تطابق وجود ندارد.

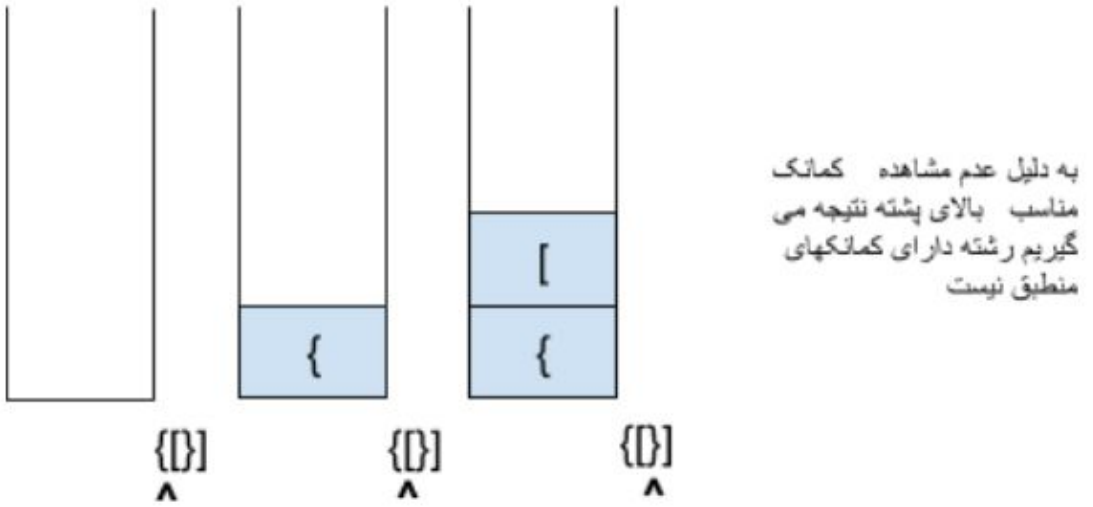
با توجه به اینکه این الگوریتم تنها یکبار رشته راپیمایش می کنیم و هر کمانک باز یکبار رو پشته قرار می گیرد، پیچیدگی محاسباتی این الگوریتم $O(n)$ می باشد.

برای اینکه از درستی الگوریتم مطمئن شویم آن را حداقل روی دو ورودی یکی با رشته منطبق مثلا "{[]}()" و دیگری رشته نامنطبق مثلا "[{]}" به صورت دستی امتحان می کنیم. برای رشته منطبق مشاهد کاراکترها و وضعیت پشته اینچنین خواهند بود:



مثالی از اجرای دسته الگوریتم روی رشته ای با کامنکهای منطبق

با این دلیل که با اتمام رشته پشته خالی است نتیجه می گیریم که همه کمانکهای رشته منطبق می باشند.برای رشته نامنطبق "[{]}" وضعیت پشته چنین خواهد بود.



مثالی از اجرای دسته الگوریتم روی رشته ای با کمانکهای نامنطبق

با توجه به این دومثال بنظر می رسد که الگوریتم می تواند جواب درست را بدهد. بنابر این شروع به نوشتن برنامه می کنیم.

ابتدا برای تابع نام مناسبی انتخاب می کنیم. سپس یک پشته، که در پایتون با لیست پیاده می شود، تعریف می کنیم (خط 2). سپس یک دیکشنری تعریف می کنیم که زوج کاراکترهای کمانکها را مشخص میکند (خط 3). سپس در یک حلقه تکرار (خطوط 4 تا 10) هربار یک کاراکتر را می خوانیم. در صورتی که کاراکتر یک کمانک باز باشد آن را روی پشته می گذاریم. در غیر اینصورت اگر که پشته خالی نباشد و کاراکتر بالای پشته کمانک باز مرتبط به کمانک بسته کنونی باشد (خط 7) آن را از روی پشته حذف می کنیم (خط 8). در غیر این صورت کمانکهای رشته منطبق نمی باشند (خط 10). با اتمام حلقه در صورتی که پشته خالی باشد نتیجه می گیریم که دارای کمانکهای منطبق می باشد (خط 11).

```
1. def bracesMatch(s):
2.     stack = []
3.     pairs = {'(':')', '{': '}', '[': ']'}
4.     for w in s:
5.         if w in pairs:
6.             stack.append(w)
7.         elif len(stack)!=0 and pairs[stack[-1]] == w:
8.             stack.pop()
9.         else:
10.             return False
11.    return len(stack)==0
```

علاوه بر الگوریتم سریع، برنامه فوق از روشی استفاده می کند که کد را تمیز و مرتب و همچنین قابل توسعه می کند. این تکنیک مربوط به استفاده از دیکشنری در خط 3 برای نگهداری زوج های متناظر کمانکها است.

استفاده از این روش دو مزیت دارد:

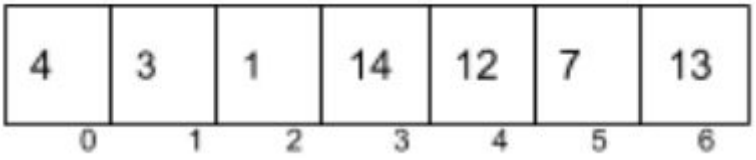
(۱) نیاز به استفاده از تعداد زیادی خط برای نوشتن if else برای تطبیق کمانک ها نمی باشد. بلکه همه آنها در خط 7 انجام می شوند.

(۲) برنامه قابلیت توسعه دارد. به این معنا که اگر در آینده یک کمانک جدید مثلا <> نیز به عنوان کمانک قابل قبول در رشته ارائه شود تنها لازم است که این زوج به خط 3 اضافه شوند و هیچ جای دیگر برنامه نیاز به تغییر ندارد.

۳.۶ اولین مقدار بزرگتر در لیست

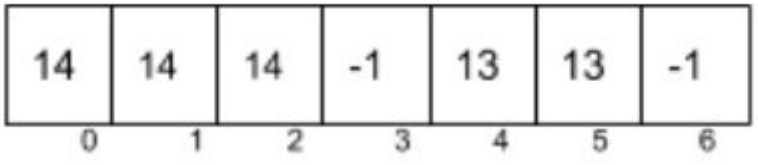
پرسش: یک لیست از مقادیر داریم. برنامه ای بنویسید که برای هر مقدار در این لیست اولین مقدار بعد از آن که بزرگتر از آن می باشد را مشخص کند و حاصل را به صورت یک لیست برگرداند. اگر چنین مقدار برای عنصری وجود نداشته باشد مقدار 1 - پاسخ خواهد بود.

پاسخ: کار حل مسئله را با یک مثال شروع می کنیم تا بتوانیم آن را بهتر درک کنیم. فرض کنیم که لیست زیر داده شده است.



لیست ورودی

اولین مقدار بعد از 4 درون لیست که بزرگتر از آن می باشد 14 است. اولین عنصر بعد از 3 که بزرگتر از آن می باشد 14 می باشد هیچ عنصری بعد از 14 بزرگتر از آن نمی باشد. بنابر این باید مقدار 1- در آن موقعیت قرارگیرد. لیست زیر پاسخ نهایی می باشد.



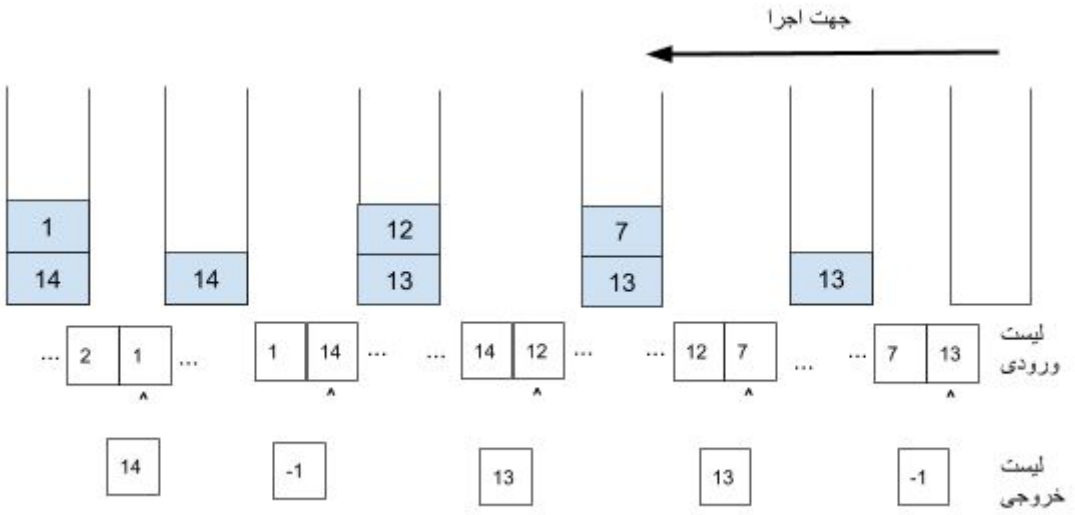
پاسخ مطلوب

به عنوان گام نخست حل مسئله از روش جستجوی کامل شروع می کنیم. در این الگوریتم برای یافتن عنصر بعدی بزرگتر از یک مقدار در لیست، همه ی عناصر بعد از آن را بررسی می کنیم تا (۱) به یک عنصر با مقدار بزرگتر از آن برسیم، یا (۲) به انتهای لیست برسیم. در بدترین حالت باید تا انتهای لیست را بپیماییم. با توجه با این که این کار را باید برای هر n عنصر لیست انجام دهیم، پیچیدگی محاسباتی این الگوریتم $O(n^2)$ می باشد. اشکال این الگوریتم آن است که از نتیجه پیمایش های قبلی برای یافتن پاسخ برای عنصر بعدی استفاده نمی کند. به عنوان مثال با وجود اینکه برای یافتن پاسخ برای عنصر

اندیس صفر (عنصر با مقدار 4) عناصر بعدی تا عنصر اندیس 3 (عنصر با مقدار 14) رد می شویم، اما باز هم برای یافتن جواب برای عنصر اندیس 1 باید همه عناصر بعد تا عنصر اندیس 3 (عنصر با مقدار 14) را بپیماییم.

اما فرض کنیم اگر کار برعکس بود. یعنی اگر جواب مطلوب برای عنصر اندیس 1 را می دانستیم، آیا کمکی به حل مسئله برای عنصر اندیس صفر می کرد؟ جواب مثبت است زیرا در حین یافتن عنصر بزرگتر از عنصر 1 (با مقدار 3) می فهمیم که عنصر با مقدار 1 (عنصر اندیس 2) از آن کوچکتر است و بنابراین یک گزینه احتمالی برای پاسخ عنصر در اندیس صفر نمی باشد. این مشاهده ذهن را برای یافتن راه حل بهینه باز می کند.

بنابراین بررسی می کنیم که آیا شروع از انتهای لیست امکان استفاده از نتایج قبلی را فراهم می کند؟ به عنوان مثال فرض کنیم که از انتهای لیست تا عنصر با مقدار 14 را پیموده و پاسخ را برای آنها یافته ایم. با توجه به اینکه همه مقادیر بعد از 14 (یعنی 13، 7، و 12) از آن کوچکترند مطمئن خواهیم بود که برای یافتن اولین عنصر بزرگتر از عناصر سمت چپ 14 نیازی به استفاده از آنها نمی باشد. زیرا یا 14 از آنها بزرگتر است (که در این صورت 14 جواب می باشد) یا 14 کوچکتر از آنهاست که در این صورت باقی عناصر نیز کوچکتر از آن می باشند. بنابراین با پیمایش از راست به چپ لیست تنها کافیست عناصر بالارونده بعدی را در یک لیست جدا نگه داریم. برای ساختن چنین لیستی می توانیم از ساختمان داده پشته استفاده کنیم.
بنابر الگوریتم جدید چنین خواهد بود: از سمت راست لیست و با یک پشته خالی شروع می کنیم و به سمت چپ لیست حرکت می کنیم. با دیدن هر عنصر جدید آن را با عنصر بالای پشته مقایسه می کنیم. اگر عنصر بالای پشته بزرگتر از آن است جواب را یافته ایم. در غیر این صورت آنقدر از بالای پشته برمی داریم که به عنصری بزرگتر از عنصر جدید برسیم یا اینکه پشته خالی شود. اگر پشته خالی شد مقدار 1- پاسخ است. در غیر این صورت مقدار بالای پشته پاسخ می باشد. با توجه به اینکه مقدار جدید کوچکتر از بالای پشته است آن را به بالای پشته اضافه می کنیم و محتوای پشته همچنان بالارونده باقی خواهد ماند.
الگوریتم پیشنهادی را روی مثال فوق اجرا میکنیم تا مطمئن شویم که درست کار میکند. در مثال زیر به علت کمبود فضا تنها تا عنصر با مقدار 1 را پیموده ایم. یادآور می شویم که روند اجرا از راست به چپ نمایش داده شده است.



قسمتی از اجرای الگوریتم روی داده ورودی

بنابر این الگوریتم روی مثال داده شده درست کار می کند. سپس کد آن را می زنیم.

در حین نوشتن برنامه کاری که هر خط می کند را توضیح می دهیم. ابتدا نامی مناسب و پارامتر ورودی تابع را تعریف می کنیم (خط 1). سپس یک پشته خالی تعریف می کنیم (خط 2). بعد یک لیست به طول لیست ورودی تعریف می کنیم که نتیجه را برای هر عنصر نگه می دارد. به هر عنصر مقدار اولیه 1- می دهیم (خط 3). سپس در یک حلقه تکرار از انتهای به سمت ابتدای لیست حرکت می کنیم (خط 5). چنانچه پشته خالی باشد آنگاه مقدار نتیجه 1- (یعنی پیشفرض عناصر لیست نتیجه) می باشد. بنابر این نیاز به تغییری در لیست نتیجه نیست اما این عنصر را باید روی پشته می گذاریم (خطوط 7 و 8).

چنانچه پشته خالی نباشد تا زمانی که عنصری بزرگتر از عنصر i ام لیست در بالای پشته نبینیم از بالای پشته برمی داریم (خط 9). عنصر بالای پشته، در صورت وجود، مقدار اولین بزرگترین عنصر بعد از عنصر iام را مشخص میکند (خط 12). سپس عنصر iام را روی پشته قرار می دهیم تا در محاسبه نتیجه برای عنصر بعدی (سمت چپ آن) استفاده کنیم (خط 13). در انتها لیست نتیجه را برمی گردانیم (خط 14).

```
1. def nextGreater(nums):
2.    stack = []
3.    res = [-1] * len(nums)
4.
5.    for i in range(len(nums)-1, -1, -1):
6.         if not stack:
7.              stack.append(nums[i])
8.         else:
9.              while s and stack[-1]<=nums[i]:
10.                  stack.pop()
11.
12.             res[i] = stack[-1] if stack else -1
13.             stack.append(nums[i])
14.   return res
```

۷. صف (Queue)

۱.۷ صف (Queue)

صف یک ساختمان داده است که روی لیست یا لیست پیوندی پیاده سازی می شود. نحوه دسترسی به عناصر صف خاص است و به همین دلیل یک ساختمان داده جداگانه در نظر گرفته می شود. الگوریتم ها با سر و ته صف کار دارند: یا عنصر به ته صف اضافه می شود یا از ابتدای صف حذف می شود.

نحوه کار با صف در پایتون

در پایتون می توان از یک لیست به عنوان صف استفاده نمود. در این صورت برای اضافه کردن به ته صف از append و حذف از ابتدای صف از $pop(0)$ استفاده کنیم. ایراد این کار آن است که عملیات حذف عنصر دارای پیچیدگی محاسباتی $O(n)$ می باشد که خیلی کند است. به همین دلیل باید از ماژول deque استفاده نماییم.

افزودن ماژول به برنامه	**from** collections **import** deque
ساختن یک صف	q = deque()
افزودن به انتهای صف	q.append(a)
حذف از ابتدای صف	q.popleft()
خالی کردن صف	q.clear()

۲.۷ کوتاهترین دنباله تغییرات

پرسش: یک عدد صحیح شروع (start)، یک عدد صحیح پایان (end)، و یک لیست از اعداد داده شده اند. می خواهیم عناصر مناسبی از لیست را، با هر تعداد تکرار که لازم است، به عدد شروع اضافه کنیم تا به عدد پایان برسیم. اگر در حین محاسبات مجموع بزرگتر از 1000 شد باید باقیمانده آن را بر 1000 استفاده کنیم. برنامه ای بنویسید که کمترین تعداد عمل جمع برای رسیدن از ابتدا به انتها را برگرداند.

پاسخ: حل مسئله را با یک مثال شروع می کنیم تا مطمئن شویم آن را خوب درک کرده ایم. اگر مقدار شروع 2, مقدار انتها 9 و لیست داده شده [3,4] باشند آنگاه می توان به روشهای مختلف از 2 به 9 رسید. مثلا ابتدا 3 را با 2 جمع بزنیم تا به حاصل میانی3 + 2 = 5 برسیم. سپس 4 را به آن می افزاییم تا به مقدار پایانی 4+5 = 9 برسیم. بنابر این حداقل 2 بار باید از عملگر جمع استفاده نماییم تا از ابتدا به انتها برسیم.

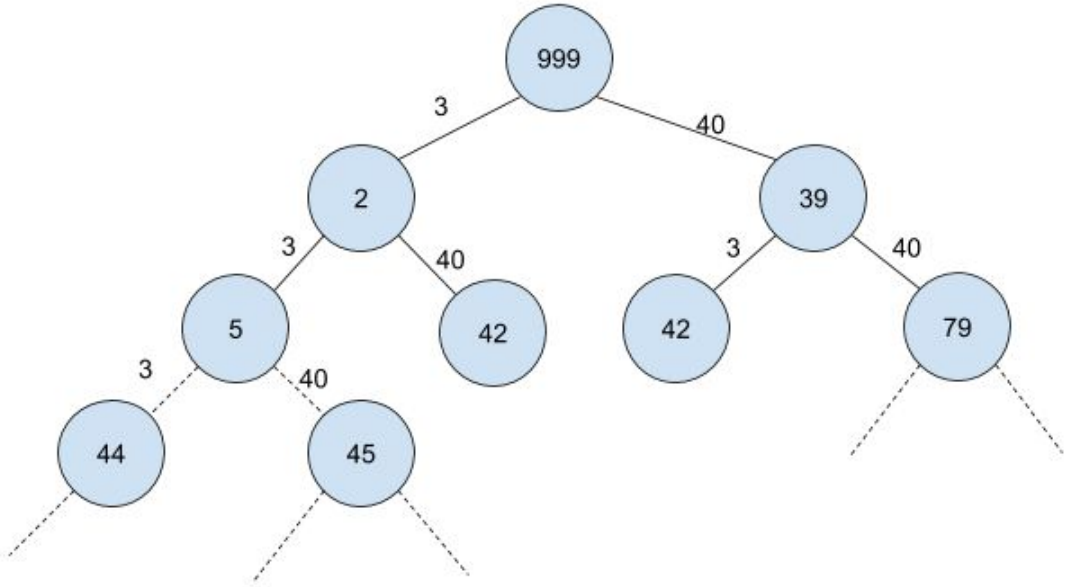
به عنوان مثالی دیگر، اگر مقدار شروع 999 و مقدار پایان 42 باشد و لیست داده شده [40, 3] باشد آنگاه می توانیم 40 را به مقدار شروع اضافه کنیم. با توجه به اینکه حاصل میانی 1039 می باشد باید باقیمانده آن را بر 1000 حساب می کنیم. به این ترتیب به مقدار جدید 1000%(40+999) = 39 میرسیم. با یک بار افزودن 3 به مقدار 42 میرسیم. اگر راههای دیگر را نیز امتحان نماییم می فهمیم که برای مقادیر شروع، پایان، و لیست داده شده تعداد 2 کمترین تعداد عمل جمع مورد نیاز می باشد. به عنوان مثالی دیگر اگر مقدار ابتدا 1، مقدار انتها 7، و مقدار لیست [10, 5] باشند آنگاه نمی توان با استفاده از لیست و عملگر جمع از مقدار ابتدا به انتها رسید. در این صورت برنامه باید مقدار 1- را برگرداند.

به عنوان اولین الگوریتم سراغ جستجوی کامل می رویم. در چنین الگوریتمی تمام حالت هایی که از مقدار شروع به مقدار پایان میرسند را می سازیم. سپس حالتی که کمترین تعداد عمل جمع را بکار می برد به عنوان نتیجه برگردانیم.

برای واضح شدن نحوه کارکرد الگوریتم جستجوی کامل یک مثال می زنیم. در مثال بالا یک حالت شامل شروع از 999, افزودن 3, سپس 3، سپس 3, سپس 3, ... است که در این بین مجموع میانی از 1000 رد می شود و به مجموع میانی 2 می رسد و سپس در نقطه ای به مجموع میانی 41 میرسیم. در مرحله بعد با افزودن مقدار 3 به 44 میرسیم. با افزودن تعداد زیادی 3 تا مقدار میانی 1001 می رسیم و با افزودن تعداد دیگری 3 به مقدار 43 میرسیم. نهایتا با یکبار دیگر رد شدن از 1000 به

مقدار 42 میرسیم. این حالت شامل تعداد وحشتناکی عمل جمع بود. در ایجاد حالت بعدی به جای یکی از این 3 ها از مقدار 40 استفاده می کنیم. در واقع با رسیدن به یک مجموع میانی هر دو حالت استفاده از 3 و استفاده از 4 را امتحان می کنیم.

دنبال کردن حالتها یک ساختار درخت و پیمایش در آن را تداعی می کند. شکل قسمتی از درخت حالتها و مقادیر میانی آنها را نشان می دهد.



درخت خیالی از نحوه ایجاد حالتهای مختلف برای رسیدن به مقدار پایانی

البته در الگوریتم جستجوی کامل که گفتیم در هر زمان تنها یک حالت را باید به صورت عمقی پیش برد تا به مقدار پایانی برسیم و سپس می توان حالت بعدی را امتحان کرد. البته مشخص است که در هر گام در حرکت عمقی درون یک حالت نیز چندین انتخاب داریم (به تعداد مقادیر درون لیست) که بعد از اتمام حرکت عمقی یکی از حالتها باید جداگانه پیگیری شوند.

این الگوریتم جستجوی کامل دارای یک نقص اساسی است که باعث می شود در برخی از حالتها هرگز متوقف نشود. علت آن است که عملا بینهایت حالت برای رسیدن از مقدار شروع به پایان وجود دارد زیرا حاصل جمع های میانی می تواند تا 1000 برود و دوباره به صفر برگردد.

برای اینکه از این حالت جلوگیری کنیم باید الگوریتم جستجوی کامل پیشنهادی را تغییر دهیم. به این صورت که حالت های ایجاد شده را محدود به آنهایی می کنیم که حاصل جمع های میانی آنها تکراری نباشد. در این صورت، با وجود اینکه همه حالتها را ایجاد نمی کنیم اما مطمئن هستیم که پاسخ با تعداد عمل جمع مینیمم در بین پاسخ های تولید شده است. دلیل آن است که هیچ یک از حالتی هایی که حاصل میانی تکراری دارند عملا تعداد عمل جمع مینیمم ندارند.

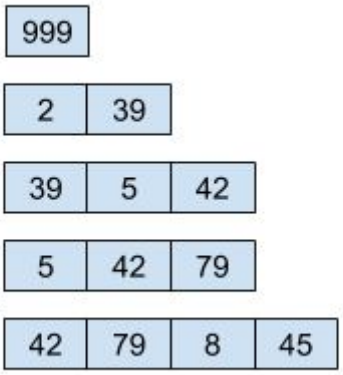
حتی با این بهینه سازی، روش جستجوی کامل دارای پیچیدگی محاسباتی نمایی است.

اشکال اصلی این روش که منجر به پیچیدگی محاسباتی بالا می شود آن است که دنباله هایی که طولانی تر از پاسخ بهینه هستند را نیز به صورت کامل می سازد، درحالیکه این پرسش تنها حالت با مینیمم عملگر جمع را می خواهد. به عبارت دیگر، روش جستجوی کامل تا عمق هر دنباله تا رسیدن به عدد انتها یا ناکام ماندن (یعنی رسیدن به یک مقدار تکراری) را ادامه می دهد.

نقطه مقابل روش عمقی پیمایش در درخت خیالی بالا، روش سطح به سطح برای ساخت حالت ها است. در این روش حالتها را به صورت موازی می سازیم و پیش می بریم. البته اینجا منظورمان استفاده از امکانات چند نخی یا چند پردازنده ای نیست. در عوض، منظورمان استفاده از روش است که بتواند همه حالتها را هر بار تنها یک گام پیش ببرد. این کار به راحتی توسط یک روش شناخته شده به نام جستجوی اول سطح (breadth-first search یا به اختصار bfs -- بخوانید بی-اف-اس) انجام می شود.

برای این کار از عنصر شروع استفاده می کنیم و همه حاصل جمع های ممکن با تک تک عناصر لیست را می سازیم. در اولین قدم تعداد حالت ها به تعداد عناصر لیست می باشد. در مراحل بعد هر حالت خود به چندین حالت دیگر منشعب می شود. این کار با افزودن هریک از عناصر لیست به مقدار کنونی هر حالت انجام می شود. این کار را به همین صورت برای حالت ها تا زمانی ادامه می دهیم که به مقدار پایانی برسیم، یا اینکه امکان پیشروی در هیچ حالتی (به دلیل مشاهده مجموع میانی تکراری) وجود نداشته باشد. اولین حالتی که به مقدار پایانی برسد مطمئنا دارای کمترین مقدار عمل جمع می باشد. زمانی در هیچ حالت امکان پیشرفت نداریم در واقع الگوریتم باید مقدار 1- را برگرداند.

برای نگه داشتن حالتها و ادامه دادن آنها به صورت سطح به سطح از ساختمان داده صف استفاده می کنیم. برای این کار هر عنصر صف آخرین مقدار یک حالت را نگه می دارد. هر بار یک عنصر از ابتدای صف برمی داریم (در واقع مجموع میانی یکی از حالتها) و به ازای هر عنصر لیست یک حاصل جمع میانی (حاصل جمع عنصری که از صف برداشتیم و یک عنصر لیست) می سازیم. هر حاصل جمع میانی جدید را به انتهای صف اضافه می کنیم. البته این کار را در صورتی انجام می دهیم که مقدار حاصل جمع تاکنون دیده نشده باشد. این کار را تا دیدن مقدار پایانی ادامه می دهیم. شکل زیر مقادیر داخل صف را برای مثال دوم (یعنی عدد ابتدا 999، انتها 42، و لیست اعداد [40, 3]) از مثال های فوق نشان می دهد.



مقادیر قرار گرفته در صف در پیمایش سطحی. هر بار یک عنصر (مقدار میانی) از صف برمی داریم و حاصل جمع آن با هر یک از عناصر لیست را به انتهای صف اضافه میکنیم.

با توجه با این که مثال داده شده به مقدار پایان می رسد الگوریتم می تواند نتیجه درست را تولید کند. اما این الگوریتم یک مشکل دارد و آن اینکه نهایتا وقتی به مقدار پایان می رسیم، نمی دانیم که چند عمل جمع برای رسیدن به آن انجام دادیم. بنابراین باید بگونه ای بدانیم در سطح چندم از درخت هستیم. این کار را می توان به سه روش انجام داد:

۱- علاوه بر مقدار حاصل جمع میانی شماره سطح آن را نیز همراه آن نگه داریم. به عنوان مثال به همراه 999 شماره سطح یعنی 0 را نیز نگه داریم. با استفاده از این مقدار می توانیم شماره سطح مقدار میانی بعدی منتج از آن را محاسبه کنیم.

۲- از دو صف استفاده کنیم: صف کنونی و صف جدید. هر یک از این صف ها فقط عناصر مربوط به یک سطح را نگهداری می کنند و از یک شمارنده استفاده می کنیم که نشان دهد عناصر صف مربوط به چه سطحی از درخت هستند. عناصر صف جدید از بیرون کشیدن عناصر صف کنونی و جمع آنها با عناصر لیست حاصل می شوند. این مجموع های میانی مقادیر صف جدید را تشکیل می دهند. با خالی شدن صف کنونی، صف جدید را جایگزین آن می کنیم و کار را ادامه می دهیم. این روند را تا دیدن مقدار پایانی یا اتمام صفها ادامه می دهیم.

۳- مقادیر ایجاد شده در جمع های میانی بین 0 تا 999 می باشند و این مقادیر در صف قرار می گیرند. در این روش از یک لیست (که از اینجا به بعد به آن *لیست سطح ها* می گوییم) استفاده می کنیم. طول این لیست 1000 است و مقدار آن مشخص می کند که آن مقدار در چه سطحی تولید شده است. مقدار اولیه همه عناصر 1- می باشد به این معنا که هنوز در پیمایش سطحی مشاهده نشده است. تنها در اندیس مقدار شروع (999 در مثال فوق) مقدار 0 را قرار می دهیم که نشان می دهد در

سطح صفر ایجاد شده است. هرگاه یک مقدار X از صف بیرون میکشیم (۱) اگر مقدار X برابر عنصر پایان باشد آنگاه مقدار عنصر X ام در لیست سطح ها تعداد عملیات جمع مورد نیاز را مشخص می کند، و (۲) در غیر این صورت حاصل جمع های میانی جدید را محاسبه می کنیم و در لیست سطح ها مقدار مشخصه کننده سطح آن ها از ریشه را برابر سطح عنصر Xام بعلاوه یک قرار می دهیم.

روش سوم بدون توجه به اندازه لیست ورودی 1000 عنصر خواهد داشت و برای لیست های طولانی ورودی حافظه کمتری استفاده خواهد نمود. بنابراین روش برای دنبال کردن سطح مربوط به مقادیر ایجاد شده استفاده می کنیم.

با این اوصاف شروع به پیاده سازی الگوریتم می کنیم. برای اینکه از ساختمان داده صف آماده استفاده کنیم ماژول deque را به برنامه اضافه می کنیم (خط 1). سپس نام و پارامترهای مناسب برای تابع استفاده می کنیم (خط 2). در خط 3 یک اسم برای مقدار پیمانه تقسیم تعریف می کنیم. این کار برای بهبود خوانایی و قابلیت استفاده مجدد برنامه انجام می شود. چنانچه عدد پایانی (یعنی مقدار end) از پیمانه بیشتر یا از صفر کمتر باشد نمی توانیم از عدد شروع به پایان برسیم. بنابر این مقدار 1 - را برمی گردانیم (خطوط 5 و 6).

برای دنبال کردن سطحی که مقدار میانی (یا پایانی) در آن قرار دارد از یک لیست سطح ها به نام step به طول 1000 با مقدار اولیه 1- برای هر عنصر تعریف می کنیم (خط 8).

سپس یک صف تعریف می کنیم و عنصر شروع را در آن قرار می دهیم (خط 10). از آنجا که هیچ کاری برای رسیدن به عدد شروع لازم نیست، مقدار مربوط به آن در لیست سطح ها را برابر صفر تنظیم می کنیم (خط 11). سپس در یک حلقه تکرار تا زمانی که به عدد انتها برسیم یا صف خالی شود (یعنی دیگر عدد جدیدی در بازه 0 تا 999 در دنباله ها ظاهر نشود) کارهایی را انجام می دهیم. این کارها شامل برداشتن یک عنصر از ابتدای صف، مقایسه آن با عدد پایان (خط 15) و افزودن تک تک مقادیر لیست به آن برای ایجاد مقادیر میانی جدید، و نهایتا افزودن آن به صف می باشد (خطوط 13 تا 21). تکه برنامه زیر این الگوریتم را پیاده سازی می کند.

```
1. from collections import deque
2. def minSteps(start, end, nums):
3.    mod_value = 1000
4.
5.    if mod_value<end or end<0:
6.          return -1
7.
8.    step = [-1] * mod_value
9.
10.   q = deque([start])
11.   step[start] = 0
12.
13.   while q:
14.         value = q.popleft()
15.         if value == end:
16.               return step[value]
17.         for n in nums:
18.               new_value = (value + n) % mod_value
19.               if step[new_value]==-1:
20.                     q.append(new_value)
21.                     step[new_value] = step[value] + 1
22.   return -1
```

برای تحلیل پیچیدگی محاسباتی الگوریتم فوق باید به این نکته توجه کنیم:

در هر مرحله از این الگوریتم تنها یک عدد از مقادیر ممکن در بازه 0 تا mod_value-1 محاسبه می شود و در هر مرحله به طول لیست (یعنی nums) مقدار جدید محاسبه می شود. اگر طول لیست را با n نشان دهیم، آنگاه پیچیدگی محاسباتی این الگوریتم برابر با O(n) خواهد بود.

۸. لیست پیوندی (Linked List)

۱.۸ لیست پیوندی

لیست پیوندی یک ساختمان داده است که در حل مسائلی که نیاز به حافظه با اندازه متغیر دارند مفید است. مثلا پشته و صف، که تعداد نامشخصی عنصر دارند، را می توان به صورت موثری توسط لیست پیوندی پیاده سازی کنیم. یک لیست پیوندی همانطور که از اسمش مشخص است از پیوند دادن متغیرها به هم توسط اشاره گرها ایجاد می شوند. در کنار مقدار برای هر عنصر یک متغیر اضافی وجود دارد که تنها علت وجودش اتصال آن عنصر به عنصر بعدی است. یک راه ساده پیاده سازی چنین ساختاری تعریف یک کلاس به اسم گره یا Node است. متغیر next اشاره گری است به عنصر بعدی در لیست پیوندی.

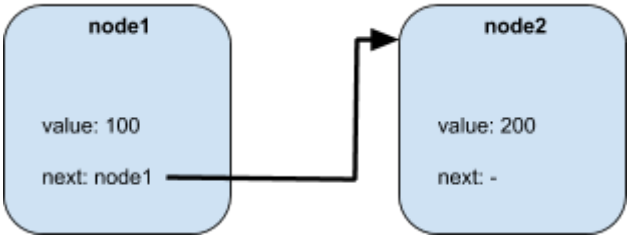
```
class Node:
     def __init__(self, value):
          self.value = value
          self.next   = None
```

برای ساختن دو گره که به هم وصل می شوند به این صورت عمل می کنیم:

```
node1 = Node(100)
node2 = Node(200)

node1.next = node2
```

درون حافظه این گره ها به صورت زیر به هم میپیوندند.



این لیست را به صورت 200<-100 نیز نشان می دهیم. این لیست ساده ترین نوع لیست است و لیست پیوندی یک طرفه نام دارد.

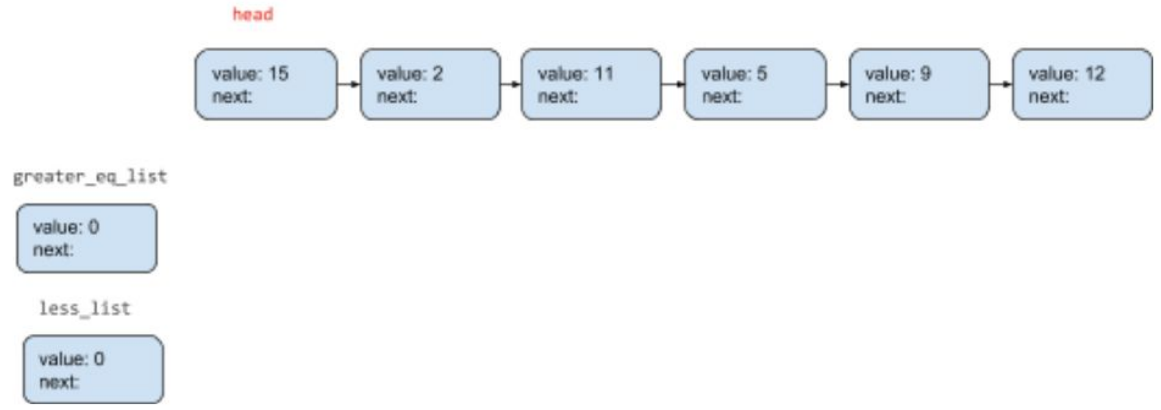
۲.۸ لیست را تفکیک کن

پرسش: برنامه ای بنویسید که یک لیست پیوندی و یک عدد دریافت نماید و آن را بگونه ای تغییر دهد که همه مقادیر کوچکتر از x قبل از مقادیر بزرگتر یا مساوی x قرار گیرند. ترتیب نسبی عناصر کوچکتر نسبت به هم و عناصر بزرگتر یا مساوی نسبت به هم نباید تغییر کند.

پاسخ: طبق حل مسئله را با یک مثال شروع می کنیم تا مطمئن شویم آن را درست درک کرده ایم. چنانچه یک لیست پیوندی شامل مقادیر 12->9->5->11->2->15 داشته باشیم و x=9 باشد، آنگاه در لیست پیوندی نهایی حاصل مقادیر کمتر از 9 باید قبل آن و بقیه باید بعد آن قرار گیرند. بنابراین لیست نهایی باید به صورت 12->9->11->15->5->2 در آید.

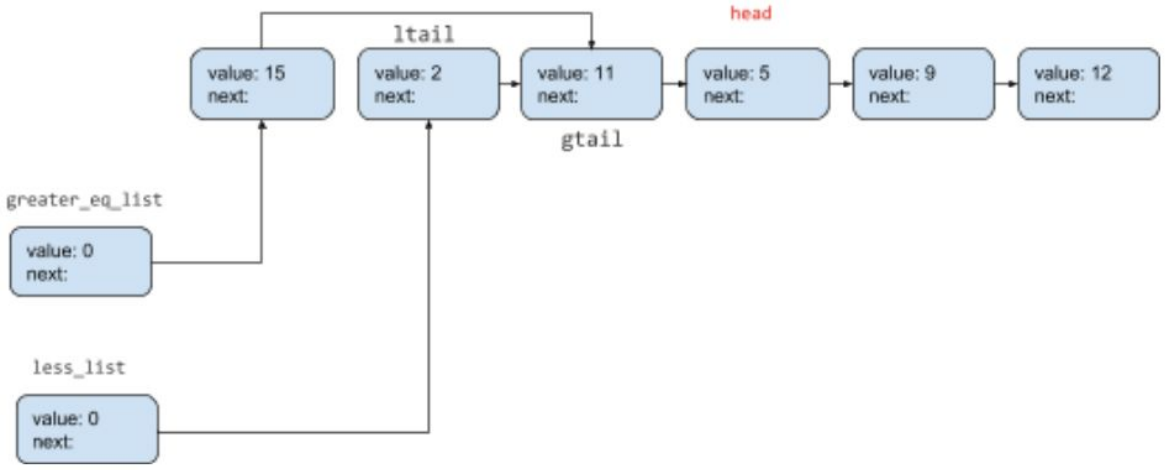
ساده ترین راه برای حل این مسئله ایجاد دو لیست پیوندی جداگانه است که یکی شامل عناصر بزرگتر مساوی x و دیگری شامل عناصر کوچکتر از x می باشند. بعد از جدا کردن مقادیر عناصر کوچکتر از بزرگتر، این دو لیست را به هم وصل کنیم تا یک لیست جدید با خاصیت گفته شده بوجود آید. برای ایجاد دو لیست جدا باید لیست پیوندی اصلی را یکبار پیمایش کنیم و با دیدن هر عنصر تصمیم بگیریم که به انتهای کدام یک از دو لیست جدید وصلش کنیم.

برای اینکه از درستی این الگوریتم اطمینان حاصل کنیم باید آن را روی لیست پیوندی مثال امتحان کنیم. با این کار همچنین دقیق تر می فهمیم که اشاره گرهای next را در هنگام پیاده سازی چگونه تنظیم نماییم. همزمان که شکل زیر را رسم می کنیم در یک به یک مراحل اجرای الگوریتم را شرح می دهیم و دنبال می کنیم. شکل زیر وضعیت ابتدایی لیست پیوندی اصلی و دو لیست پیوندی جدید یعنی greater_eq_list (عناصر بزرگتر یا مساوی x) و less_list (عناصر کوچکتر از x) را نشان می دهد.



وضعیت اولیه لیست پیوندی اصلی و لیستهای پیوندی جدید.

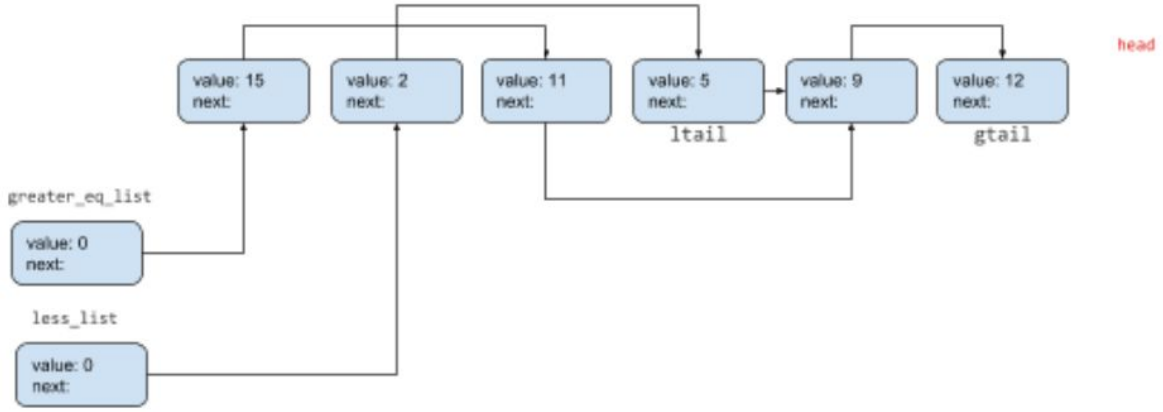
شکل زیر وضعیت لیست ها بعد از پیمایش سه عنصر اول لیست اصلی را نشان می دهد.



وضعیت لیست های پیوندی پس از پیمایش سه عنصر اول لیست اصلی. عنصر شامل مقدار 15 به انتهای لیست بزرگتر مساوی ها و 2 به انتهای لیست کوچکتر ها اضافه می شود.

با توجه به اینکه عنصر اول بزرگتر از 9 می باشد به لیست greater_eq_list اضافه می شود. اگر یکی از عناصر بعدی نیز بزرگتر از 9 باشد باید آن را به انتهای این لیست اضافه کنیم. بنابراین نیاز داریم که اشاره گری داشته باشیم که آخرین عنصر این لیست را بداند. اسم این اشاره گر را gtail (کلمه tail به معنای دم یا انتهاست و g یادآور کلمه greater) می نامیم. در ادامه پیمایش لیست اصلی به دومین عنصر می رسیم. به این خاطر که آن کوچکتر از 9 می باشد، به لیست less_list اضافه اش می کنیم. یک اشاره گر به نام ltail برای این لیست پیوندی تعریف می کنیم که اشاره گری به انتهای آن را نگه دارد. عنصر بعدی مقدار 11 دارد که از 9 بیشتر است. بنابراین باید آن را به انتهای لیست greater_eq_list اضافه کنیم. این کار را با تغییر اشاره گر next عنصر آخر لیست یعنی gtail انجام می دهیم. سپس مقدار gtail را بروز می کنیم تا به عنصر آخر جدید

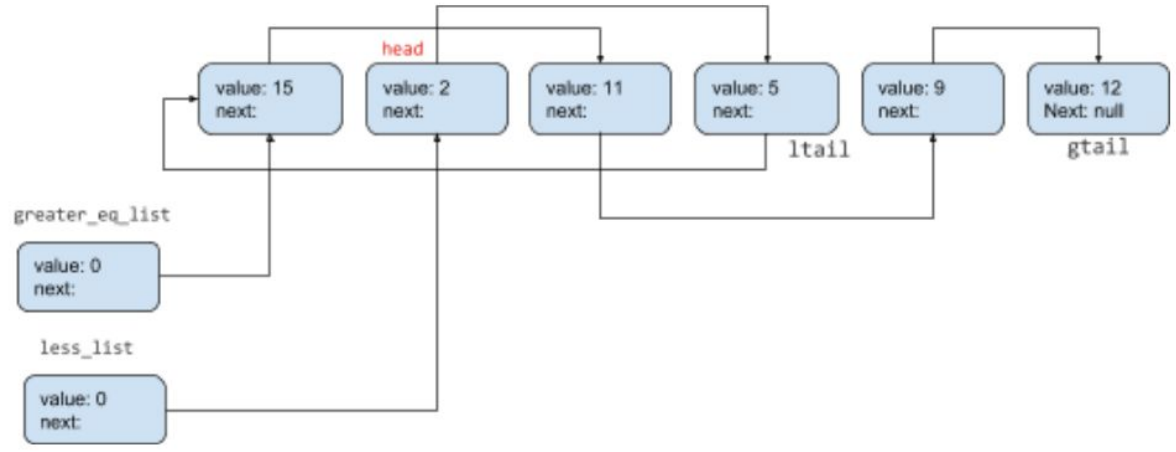
اشاره کنید. این کار را تا رسیدن به انتهای لیست انجام می دهیم. با رسیدن به انتهای لیست وضعیت اشاره گر ها اینگونه خواهد بود.



وضعیت لیست های پیوندی پس از پیمایش همه عناصر لیست پیوندی اصلی

خوب تا اینجای کار عناصر بزرگتر یا مساوی را در یک لیست جدا گذاشته ایم و عناصر کوچکتر را در یک لیست پیوندی دیگر.

حالا باید این دو لیست را با هم ترکیب کنیم بگونه ای که همه عناصر کوچکتر قبل از همه عناصر بزرگتر از x قرار بگیرند. برای این کار باید (۱) عنصر انتهای لیست کوچکترها به عنصر ابتدای لیست بزرگتر مساوی ها اشاره کند، (۲) مقدار next در عنصر انتهایی در لیست بزرگتر مساوی ها به null تنظیم شود، و (۳) متغیر head که به ابتدای لیست نهایی اشاره گر می کند باید به اولین عنصر لیست کوچکترها اشاره کند. شکل زیر وضعیت نهایی اشاره گرها را نشان می دهد.



با این اوصاف مطمئن می شویم که الگوریتم درست عمل می کند و می فهمیم که در پیاده سازی به تعدادی اشاره گر و متغیر اضافی نیاز داریم. از آنجا که لیست دارای ترتیب مشخصی نمی باشد بنابراین پیمایش آن اجتناب ناپذیر است. به این خاطر که

نیاز به پیمایش کامل لیست داریم، اگر طول لیست n باشد، آنگاه پیچیدگی محاسباتی (مرتبه اجرایی) این الگوریتم O(n) می باشد و الگوریتمی سریعتر از آن یافت نخواهد شد.

می توانیم در مورد روش پیاده سازی دیگری نیز به صورت خلاصه صحبت کنیم. به این صورت که یکبار لیست را بپیماییم تا عنصر انتهایی لیست را بیابیم. سپس دوباره از ابتدای لیست پیوندی حرکت کنیم. اگر عنصری بزرگتر یا مساوی مقدار x بود آن را به انتهای لیست منتقل کنیم و در غیر این صورت کاری نکنیم. این الگوریتم دارای پیچیدگی محاسباتی مانند روش ایجاد دو لیست پیوندی است. با این تفاوت که لیست را دوبار باید بپیماییم.

بنظر می رسد که الگوریتم اول سر راست تر است و پیاده سازی آن خوانا تر خواهد بود. بنابراین شروع به پیاده سازی روش اول می کنیم. ابتدا نام مناسب و پارامترهای مناسب شامل عنصر اول لیست پیوندی و مقداری x را به آن می دهیم (خط 1). سپس یک لیست خالی برای ساختن لیست از عناصر بزرگتر مساوی x تعریف میکنیم (خط 2). یک اشاره گر نیز برای اشاره با عنصر انتهایی لیست میگیریم (خط 3). لیست مشابهای برای عناصر کوچکتر از x در نظر میگیریم (خطوط 5 و 6). سپس در یک حلقه تکرار هر بار یک عنصر در لیست پیش می رویم و عنصر کنونی را متناسب با مقدار آن به انتهای لیست بزرگتر مساوی ها یا لیست کوچکتر ها اضافه می کنیم (خطوط 8 تا 16). در انتها مقدار اشاره گر عنصر آخر لیست پیوندی بزرگتر مساوی ها (gtail) را null می کنیم و اشاره گر عنصر آخر لیست پیوندی کوچکترها را به ابتدای لیست بزرگتر مساوی ها تنظیم می کنیم (خط 18 و 19). در انتها عنصر ابتدایی رشته کوچکترها را به عنوان ابتدای لیست جدید برمیگردانیم (خط 20).

```
1. def partition(head, x):
2.     greater_eq_list = ListNode(0)
3.     gtail = greater_eq_list
4.
5.     less_list = ListNode(0)
6.     ltail = less_list
7.
8.     while head:
9.         if head.val<x:
11.            ltail.next = head
12.            ltail = ltail.next
13.        else:
14.            gtail.next = head
15.            gtail = gtail.next
16.        head = head.next
17.
18.    gtail.next = None
19.    ltail.next = greater_eq_list.next
20.    return less_list.next
```

۹. درخت دودویی (Binary Tree)

۱.۹ درخت دودویی

درخت دودویی یک ساختمان داده برای نگهداری داده هایی است که ارتباط عناصر آن شکل یک درخت وارون را تداعی می کند و هر گره درخت حداکثر دو فرزند دارد. این ساختار داده ها در کاربردهای مختلف ظهور می کند. مثلا فرمول $x + y$ را می توان به صورت یک درخت دودویی با ریشه + و x به عنوان فرزند چپ و y به عنوان فرزند راست نشان داد. به عنوان مثالی دیگر می توان حالتهای مختلف حاصل از چندین بار پرتاب یک سکه را در قالب یک درخت دودویی کامل نشان داد که هر نود مقدار صفر یا یک دارد. الگوریتم های ایجاد شده برای پردازش درخت دودویی قدرت پردازش ای به این داده ها می دهد که بدون آن امکان پذیر نیست. واقعیت این است که درخت های دودویی خاص، مثلا درخت جستجوی دودویی یا درخت دودویی هیپ، کاربردهای گسترده تری دارند. با این حال در بسیاری از موارد درخت دودویی سوژه خوبی برای پرسشهای برنامه نویسی است.

نحوه کار با درخت دودویی در پایتون

در زبان پایتون کتابخانه استانداردی برای تعریف و اجرای الگوریتم های روی درخت دودویی وجود ندارد. پیاده سازی درخت دودویی بسیار ساده است. مهمترین جزء آن تعریف یک گره در درخت دودویی است.

تعریف گره درخت دودویی:

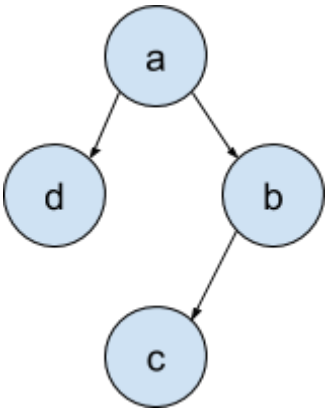
تعریف یک گره درخت دودویی توسط یک کلاس انجام می شود. این کلاس دارای یک متغیر برای نشان دادن مقدار یک گره، یک متغیر برای اشاره به گره چپ، و دیگری برای اشاره به گره راست است.

```
1. class Node:
2.     def __init__(self, data):
3.         self.data   = data
4.         self.left   = None
5.         self.right  = None
```

برای ساختن یک درخت دودویی باید نودهای آن را تک تک بسازیم و آنها را به درستی به هم وصل کنیم.

مثال: می خواهیم درختی که در تصویر زیر نمایش داده شده را در حافظه یک برنامه بسازیم و با آن کار کنیم. این کار را می توان به روش زیر و با استفاده از ساختمان داده فوق انجام دهیم.

```
root = Node(“a”)
root.left = Node(“d”)
root.right = Node (“b”)
root.right.left = Node(“c”)
```

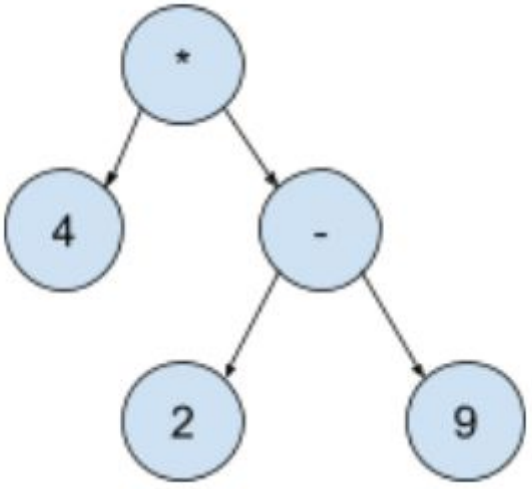


درخت X

۲.۹ عبارت ریاضی را محاسبه کنید.

پرسش: یک درخت دودویی داده می شود که یک عبارت ریاضی را به این صورت نشان می دهد: هر گره یک عملگر یا یک مقدار است. عملگر روی حاصل زیر درخت چپ و راست عمل می کند. برنامه ای بنویسید که حاصل عبارت ریاضی مرتبط با درخت دودویی را محاسبه نماید.

پاسخ: حل مسئله را با یک مثال شروع می کنیم تا مطمئن شویم که آن را درست درک کرده ایم. فرض کنیم که درخت عبارت ریاضی شکل زیر را داریم. این درخت معادل عبارت ریاضی (9-2) * 4 می باشد.



ریشه این درخت عملگر * (ضرب) می باشد. این به آن معناست که حاصل نهایی این عبارت شامل حاصل ضرب زیر درخت چپ (یعنی 4) در حاصل زیر درخت سمت راست می باشد. بنابراین قبل از آنکه بتوانیم حاصل ضرب را حساب کنیم به دانستن حاصل زیر درخت سمت راست نیاز داریم. ریشه زیر درخت سمت راست عملگر - (تفریق) می باشد به این معنا که حاصل این زیر درخت برابر با تفریق زیر درخت های سمت چپ و راست آن (یعنی 2 و 9) خواهد بود. بنابراین حاصل این زیر درخت 2 - 9 = 7- می باشد. با توجه به اینکه اکنون حاصل زیر درخت سمت راست ریشه را داریم می توانیم عملگر ضرب را اعمال نموده و با حاصل نهایی 4*(7-) = 28- می رسیم.

این روش بدست آوردن حاصل مربوط به عبارت ریاضی درخت پیشنهاد می کند که ما از یک الگوریتم بازگشتی برای پاسخ به مسئله استفاده نماییم: برای بدست آوردن مقدار هر زیر درخت در صورتی که ریشه ی زیر درخت یک عملگر باشد باید ابتدا حاصل زیر درخت سمت چپ و سپس زیر درخت سمت راست آن را به صورت بازگشتی محاسبه کنیم. آنگاه عملگر ریشه را روی حاصل زیر درخت چپ و راست اجرا کنیم. در صورتی که مقدار ریشه زیر درخت یک عدد باشد، حاصل آن خود مقدار عددی است.

برای اطمینان از درستی این الگوریتم، آن را روی درخت مثال فوق اجرا می کنیم.

۱- محاسبه مقدار برای ریشه درخت

مشاهده ریشه (عملگر *)

۱.۱ - محاسبه مقدار زیر درخت سمت چپ

مشاهده مقدار 4 به عنوان ریشه. چون یک عملگر نیست خود مقدار 4 حاصل است.

۲.۱ - محاسبه مقدار زیر درخت سمت راست

مشاهده ریشه زیر درخت (عملگر -)

۱.۲.۱- محاسبه زیر درخت سمت چپ

مشاهده مقدار 2 به عنوان ریشه و برگرداندن آن به عنوان حاصل

۲.۲.۱- محاسبه زیر درخت سمت راست

مشاهده مقدار 9 به عنوان ریشه و برگرداندن آن به عنوان حاصل

اعمال عملگر - روی حاصل زیر درخت چپ (2) و زیر درخت راست (9) و برگرداندن 7-

اعمال عملگر * روی حاصل زیر درخت چپ (4) و زیر درخت راست (7-) و برگرداندن 28-

بنابراین اجرای الگوریتم روی درخت مثال پاسخ درست را تولید می کند.

سپس شروع به نوشتن برنامه می کنیم. چند خط اول برنامه (خط 1 تا 5) تعریف ساختار گره های درخت است. گره یک کلاس ساده است که دارای چند فیلد data برای نگهداری مقدار گره، left اشاره گری به زیر درخت چپ، و right اشاره گری به زیر درخت راست می باشد.

سپس قسمت الگوریتمی برنامه شروع می شود. ابتدا نام و پارامتر مناسب برای تابع تعریف می کنیم (خط 7). در ابتدای تابع چک می کنیم که آیا ریشه مقداری تهی است. در این صورت مقدار 0 را بر می گرداند (خط 8 و 9). این کار بخصوص در مواجهه با عملگر تکی منها مفید می باشد.

سپس یک دیکشنری از عملگرها تعریف می کنیم. مقدار مربوط به هر عنصر دیکشنری یک تابع است که دو پارامتر می گیرد و عملگر را روی آنها اعمال و حاصل را برمی گرداند (خط 11 تا 14). از این تکنیک به عنوان جایگزینی برای ساختار شرطی متوالی مورد نیاز و تکرار کد مشابه استفاده می کنیم.

دو خط آخر برنامه (در واقع یک خط که به دلیل طولانی بودن شکسته شده است) بسیار فشرده است و فراخوانی های بازگشتی در آن انجام می دهیم. این کد بینهایت زیباست.

در هر بازگشت ریشه یک زیر درخت وارسی می کنیم. اگر مقدار ریشه عملگر (یکی از مقادیر +، -، *، /) نباشد خود مقدار برمی گردانیم. در غیر این صورت حاصل زیر درخت چپ و سپس راست با فراخوانی بازگشتی محاسبه می کنیم و آنگاه عملگر روی آنها برای محاسبه حاصل اعمال می شود (خط 15 و 16).

```
1. def calc(root):
2.     if root == None:
3.           return 0
4.
5.     opt = {"+": lambda x, y: x + y,
6.            "-": lambda x, y: x - y,
7.            "*": lambda x, y: x * y,
8.            "/": lambda x, y: x / y}
9.     return opt[root.data](calc(root.left), calc(root.right))\
10.           if root.data in opt else root.data
```

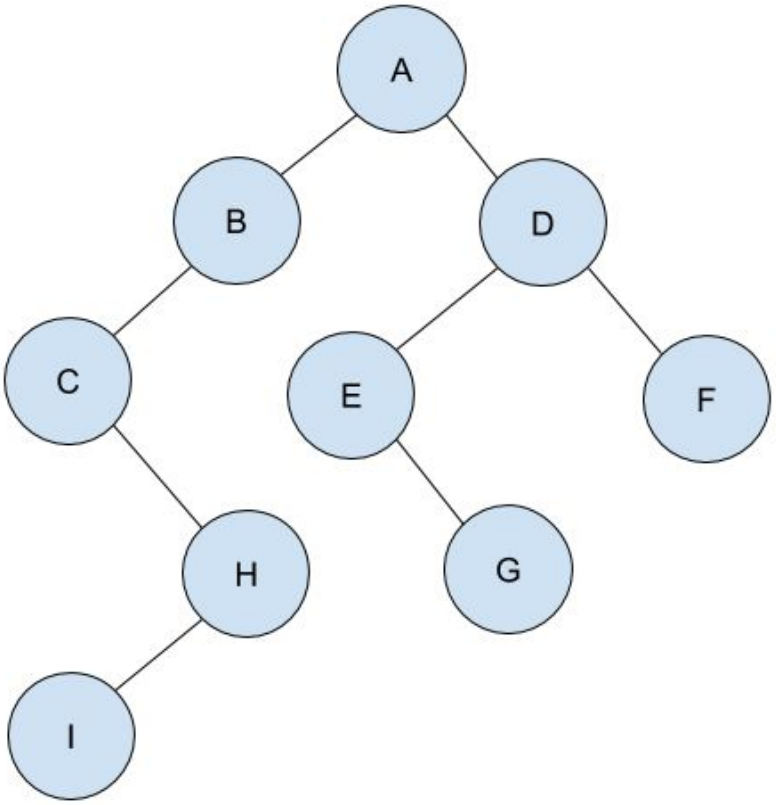
در خط 11 در برنامه فوق از یک دیکشنری استفاده شده است. مقدار مربوط به هر کلید در دیکشنری یک تابع lambda است که در واقع امکان تعریف توابع یک خطی را می دهد. توابعی که اینجا تعریف کردیم دو پارامتر می گیرند و حاصلی را محاسبه می کنند.

بدلیل استفاده از این ساختار دیکشنری، دیگر نیازی به استفاده از شرط های پی در پی برای فراخوانی تابع وجود ندارد. در ضمن برنامه بسیار قابل توسعه است و اگر عملگر جدیدی به عملگرهای محاسباتی درخت اضافه شود تنها کافی است یک خط به دیکشنری اضافه شود.

۳.۹ مینیمم ارتفاع برگها را بیابید

پرسش: برنامه ای بنویسید که مینیمم ارتفاع برگ یک درخت را محاسبه نماید.

پاسخ: حل مسئله را با یک مثال شروع می کنیم تا مطمئن شویم که آن را درست فهمیده ایم. درخت زیر را در نظر بگیریم.



یک درخت دودویی که مینیمم ارتفاع برگهای آن 2 می باشد. برگ F کمترین ارتفاع را دارد.

ارائه راه حل را با روش جستجوی کامل شروع می کنیم. روش جستجوی کامل شامل محاسبه ارتفاع برای همه برگها و محاسبه مقدار مینیمم بین آنهاست.

راه راحت پیاده سازی این الگوریتم استفاده از فراخوانی بازگشتی برای پیمایش عمقی درخت است. اما علاوه بر پیمایش درخت، باید ارتفاع هر گره را هم حساب کنیم. چالش معمول آن است که چطور پس از رسیدن به یک برگ (leaf) بدانیم که ارتفاع آن چقدر بوده است؟ و اینکه نهایتا در چه نقطه ای از الگوریتم باید مینیمم بین ارتفاع همه برگها را حساب کنیم؟ آیا باید ارتفاع ها را در یک لیست بریزیم و نهایتا مینیمم آن را بگیریم یا یک متغیر عمومی داشته باشیم که همیشه مینیمم در آن قرار دارد و کمترین ارتفاع برگ در حین محاسبات در آن قرار گیرد؟

واقعیت این است که محاسبه مقدار نهایی در برگها تنها مدل فکر کردن در برنامه های بازگشتی نیست. البته در برخی از مسائل از جمله برنامه بازگشتی مربوط به ب.م.م (gcd) از این روش استفاده می شود. اما در بسیاری مسائل دیگر محاسبات نهایی پس از برگرداندن مقادیر محاسبه شده از زیر درخت ها و در گره پدر انجام می شود.

برای رسیدن به پیاده سازی تمیزتر باید محاسبات در راه حل کنونی در گره های پدر انجام شوند. باید به این فکر کنیم که چگونه می توان کمترین ارتفاع برگها نسبت به گره پدر را بر اساس مینیمم ارتفاع برگ ها در زیر درخت های سمت چپ و راست آن بدست آورد؟ منظور از مینیمم ارتفاع زیر درخت ها بدون توجه به ارتفاع آنها در درخت پدر است. یعنی اگر زیر درخت تنها یک گره باشد، مینیمم ارتفاع برگها در آن 0 است.

برای ترکیب نتیجه محاسبات زیر درخت ها در گره پدر باید مینیمم بین نتیجه محاسبه برای زیر درخت چپ و راست را بگیریم و یک واحد به آن اضافه نماییم تا مینیمم ارتفاع برگ ها در درخت پدر محاسبه شود. با اجرای بازگشتی این رویه از ریشه درخت، فراخوانی به زیر درخت ها می رسد. با بازگشت از هر زیر درخت، نتایج محاسبات برای زیر برگها ترکیب می شود و نتیجه برای زیر درخت کنونی را می سازد. نهایتا ترکیب نتیجه محاسبات برای زیر درخت های چپ و راست ریشه، نتیجه نهایی را بدست می دهد.

این پیمایش از ریشه تا هریک از برگها می رود و همه گره ها را تنها یکبار مشاهده می کند. چنانچه درخت دارای n گره باشد، پیچیدگی محاسباتی این الگوریتم O(n) می باشد.

در یک مصاحبه واقعی الگوریتم را روی یک مثال اجرا می کنیم تا مطمئن شویم که درست کار می کند.

سپس شروع به پیاده سازی الگوریتم می کنیم. نوشتن ساختار گره (Node) اختیاری است ولی برای واضح شدن الگوریتم آن را می آوریم. سپس شروع به نوشتن الگوریتم بازگشتی می کنیم. نام و پارامتر مناسب برای تابع در نظر میگیریم (خط 1). اگر ریشه زیر درخت Null باشد، مقدار 1- را برمی گردانیم (خط 9 و 10). با توجه به نحوه فراخوانی بازگشتی، این شرط تنها یکبار ممکن است اجرا شود و آن حالتی است که ریشه درخت اصلی Null باشد. سپس بررسی می کنیم که آیا گره ریشه زیر درخت یک برگ است. در این صورت مینیمم ارتفاع برگ در این زیر درخت 0 می باشد و درجا آن را برمی گردانیم (خط 11 و 12).

دو خط بعدی برنامه (خط 13 و 14) فشرده است و نیاز به تمرکز زیادی برای نوشت دارد. در این خط مینیمم ارتفاع برگ برای زیر درخت راست و چپ محاسبه می کنیم. سپس مینیمم بین حاصل فراخوانی های بازگشتی را محاسبه می کنیم. با توجه به اینکه ریشه یک واحد به ارتفاع مینیمم برگ اضافه می کند، نهایتا یک واحد به مینیمم حاصل های می افزاییم و نتیجه را برمی گردانیم.

البته قبل از فراخوانی بازگشتی بررسی می کنیم که آیا زیر درخت چپ و راست خالی هستند. در صورتی که خالی هستند مقدار بینهایت (یعنی بزرگتری مقدار صحیحی ممکن) را به عنوان مینیمم ارتفاع برگ در زیر درخت در نظر میگیریم تا اثر آن در محاسبه مینیمم خنثی شود.

```
1. import sys
2. class Node:
3.    def __init__(self, value):
4.          self.value = value
5.          self.right = None
6.          self.left  = None
7.
8.  def findMinLeaf(root):
9.    if not root:
10.         return -1
11.   if not root.left and not root.right:
12.         return 0
13.   return min(findMinLeaf(root.left) if root.left\
14.                                     else sys.maxint,
15.              findMinLeaf(root.right) if root.right\
16.                                     else sys.maxint) + 1
```

اشکال استفاده روش پیمایش عمقی برای این مسئله آن است که همه برگها از جمله عمیق ترین را نیز باید پیمایش کند. درحالیکه اگر به صورت سطح به سطح درخت را پیمایش کنیم، آنگاه اولین برگی که به آن برمی خوریم کم ارتفاع ترین است و در نتیجه نیاز به پیمایش مابقی درخت نخواهیم داشت.

در صورتی که مصاحبه کننده تمایل داشته باشد شروع به پیاده سازی روش دوم می کنیم. همزمان با نوشتن کد سطرها را توضیح می دهیم. برای جلوگیری از اطاله کلام، در این نوشته تنها بخشهایی از کد را توضیح می دهیم. نکته قابل توجه آن است که به همراه گره ها در صف، سطحی که در آن مشاهده شده اند نیز در صف قرار می گیرد. در واقع یک تاپل در صف

قرار می دهیم. مثلا در سطر 6, گره ریشه با همراه سطح آن که 0 می باشد به صف اضافه می شود. وقتی که فرزندهای یک گره را به آخر صف اضافه می کنیم، سطح آنها برابر می شود با سطح گره کنونی بعلاوه یک (خط های 11 تا 14). سطح اولین گره ای که مشاهده کردیم برابر است با مینیمم ارتفاع برگهای درخت (خط 9 و 10). قطعه کد زیر پیاده سازی الگوریتم با روش جستجوی سطح به سطح را نشان می دهد.

```
1. from collections import deque
2. def findMinLeaf(root):
3.     if root == None:
4.           return -1
5.
6.     q = deque([(root, 0)])
7.
8.     while q:
9.           a, level = q.popleft()
10.          if not (a.left or a.right):
11.                return level
12.          if a.left:
13.                q.append((a.left,  level+1))
14.          if a.right:
15.                q.append((a.right, level+1))
```

گفتن اینکه شما روش های مختلف حل مسئله را دیده اید و توانایی پیاده سازی آن را دارید، مطمئنا وی را تحت تاثیر قرار خواهد داد!

۱.۱۰ دیکشنری در پایتون (Dictionary)

ساختمان داده دیکشنری به نام جدول درهم سازی (Hashtable) نیز شناخته می شود و در حل بسیاری از پرسشها کاربرد دارد. مثلا اگر بخواهیم جمعیت هر شهر ایران را در حافظه یک برنامه نگهداری کنیم و مرتب بر اساس نام شهر به جمعیت آن دست یابیم، ساختمان دیکشنری بهترین راه است. هر شهر و جمعیت آن به صورت یک زوج (کلید، مقدار) در این ساختمان داده نگهداری می شود. با دریافت نام شهر (کلید)، ساختمان داده دیکشنری فورا جمعیت (مقدار) آن را برمی گرداند. بعدا توضیح می دهیم که دیکشنری چگونه این کار را بدون پیمایش همه داده ها، فوری انجام می دهد.

ساختمان داده دیکشنری به صورت یک حافظه ذخیره و بازیابی بسیار سریع برای زوج های کلید-مقدار می باشد. برای ذخیره یک زوج در دیکشنری یک کلید و مقدار را به آن می دهیم. برای بازیابی اطلاعات باید کلید (key) مورد نظر را بدهیم و دیکشنری بسرعت مقدار مرتبط با آن را برمی گرداند.

نحوه کار با دیکشنری در پایتون

تعریف دیکشنری	h = {}
تعریف دیکشنری به همراه تعدادی زوج کلید-مقدار (مثال برای سه زوج)	h = {key1: value1, key2: value2, key3: value3}
اضافه کردن یک زوج کلید و مقدار	h[key] = value
بازیابی یک مقدار با داشتن یک کلید	value = h[key]
حذف یک عنصر با کلید key از دیکشنری	h.pop(key)

key in h	وارسی وجود زوج با کلید key در دیکشنری (حاصل True یا False)
for k in h: ...	تکرار یک حلقه روی تمام عناصر دیکشنری روش اول
for k, v in h.items(): ...	تکرار یک حلقه روی تمام عناصر دیکشنری روش دوم
h.keys()	گرفتن تمام کلیدهای یک دیکشنری
k.values()	گرفتن تمام مقادیر یک دیکشنری

۱۱. درخت هیپ (Heap)

۱۱.۱ درخت هیپ در پایتون (Heap)

ساختمان داده درخت هیپ دو نوع دارد. درخت هیپ مینیمم یک درخت دودویی است با این خاصیت ویژه که مقدار هر گره پدر از گره های فرزند کوچکتر است. همینجور درخت هیپ ماکزیمم یک درخت دودویی است که مقدار هر گره پدر از گره های فرزند بزرگتر است. این ساختمان داده کاربردهای زیادی دارد. به خصوص زمانی که نیاز به محاسبه مقدار کمینه برای یک سری داده که مرتبا بروز می شوند داریم.

نحوه کار با درخت هیپ در پایتون

ایجای یک درخت هیپ (heap)	heap = []
ایجاد یک درخت هیپ با مقادیر اولیه	heap = heapify([value1, value2, ...])
افزودن یک عنصر به هیپ	heappush(heap, value)
برداشتن کوچکترین عنصر از هیپ	value = heappop(heap)

البته برای استفاده از هرم باید ماژول heapq و کلاسهای مرتب را به برنامه اضافه نماییم.

۲.۱۱ مرتب سازی لیست k-مرتب

پرسش: یک لیست k-مرتب داده شده است به این معنا که هر عنصری از این لیست حداکثر K خانه از موقعیت کاملا مرتب آن فاصله دارد. برنامه ای بنویسید که این لیست را مرتب کند.

پاسخ: حل مسئله را با یک مثال شروع می کنیم. لیست [5, 6, 2, 4, 3, 1] یک لیست 3-مرتب است زیرا هر عنصر آن حداکثر سه خانه از موقعیت آن در لیست مرتب شده یعنی [6, 5, 4, 3, 2, 1] فاصله دارد. مثلا مقدار 2 باید در خانه اندیس 1 باشد درحالیکه در اندیس 3 قرار گرفته است. به عبارت دیگر برای مرتب کردن لیست، هیچ عنصری نیاز به جابجایی بیش از 3 خانه ندارد.

بدیهی است که برای مرتب کردن این لیست می توان از الگوریتم های مرتب سازی لیست استفاده نمود. این روشها معمولا پیچیدگی محاسباتی O(n*log(n)) دارند. اما با توجه به اینکه این لیست تقریبا مرتب است باید بتوان با هزینه کمتری آن را مرتب نمود.

در واقع به این دلیل که هر عنصری در فاصله k از موقعیت درستش می باشد، با n بار مرتب سازی زیر لیست هایی به طول 2k+1 می توانیم لیست را مرتب کنیم. پیچیدگی محاسباتی چنین الگوریتمی O(n*k*log(k)) می باشد. با اندکی دقت بیشتر متوجه می شویم که برای هر عنصر لیست تنها نیاز به مرتب کردن k+1 عنصر، شامل خودش و k عنصر بعد آن داریم. مثلا اگر برای یافتن مقداری که در موقعیت صفر لیست قرار می گیرد تنها نیاز به مرتب کردن k+1 عنصر اول لیست می باشد. با این کار طبق تعریف لیست k-مرتب عنصر اول در بین این k+1 عنصر است که با مرتب سازی آنها در محل درست قرار می گیرد.

نکته مهم آن است که لیست حاصل همچنان k-مرتب هست. بنابراین در مرحله بعد برای یافتن مقداری که در موقعیت یک قرار می گیرد مشابه این کار را تکرار می کنیم: در واقع k عنصر بعد از عنصر اول لیست را را به همراه خودش مرتب می کنیم تا مقدار درست در محل اندیس یک قرار گیرد. هزینه کل مرتب سازی با این الگوریتم هم O(n*k*log(k)) می باشد.

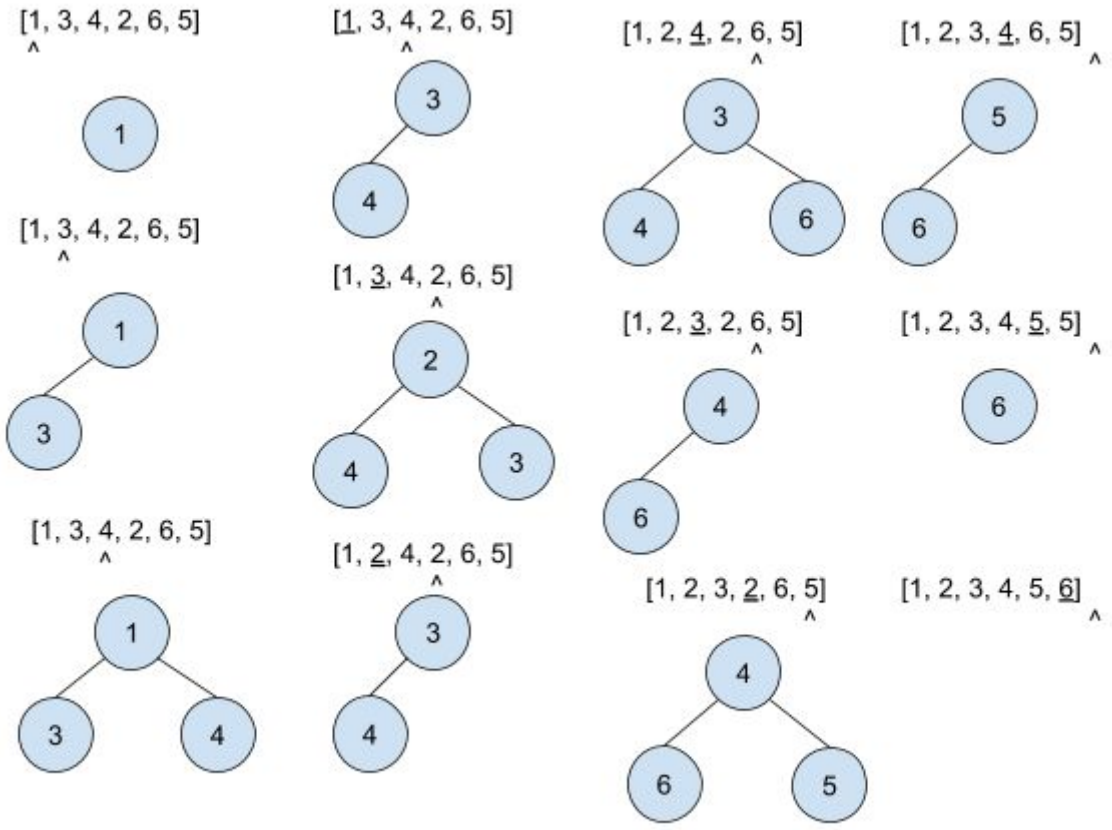
این الگوریتم نسبت به قبلی هوشمندانه تر است اما الزاما سریعتر نمی باشد. اشکال این روش آن است که هر بار کل k عنصر را از نو مرتب می کند. در حالیکه هر بار تنها عنصر اول از لیست مرتب شده حذف می شود و یک عنصر جدید به آن اضافه

می شود. بنابراین هر بار تنها نیاز به دانستن کوچکترین عنصر بین k عنصر داریم نه مرتب سازی کل آن. این همان کاری است که درخت هیپ می تواند در انجامش به ما کمک کند.

برای این کار ابتدا یک درخت هیپ با k+1 عنصر اول لیست می سازیم. سپس با برداشتن ریشه درخت هیپ عنصری که در اندیس صفر باید درج شود را بدست می آوریم و آن در عنصر صقر لیست می گذاریم. در مرحله بعدی عنصر در موقعیت k+1 را در درخت هیپ درج می کنیم. با این کار درخت بروز می شود و عنصر با مقدار مینیمم دوباره در ریشه قرار می گیرد. این مقدار باید در موقعیت اندیس یک لیست قرار گیرد. عمل برداشتن عنصر مینیمم و افزودن عنصر در k+1 عنصر آنسوتر را تا رسیدن با انتهای لیست ادامه می دهیم.

از آنجا که برای درج هر عنصر در هیپ با اندازه k حداکثر (O(log(k)) عمل مقایسه و جابجایی انجام می شود، و هر یک از عناصر لیست یکبار در هیپ درج و از آن حذف می شود، آنگاه پیچیدگی محاسباتی این الگوریتم (O(n*log(k)) می باشد. چنانچه k بسیار کوچکتر از n باشد آنگاه این الگوریتم خیلی سریعتر از مرتب سازی کل لیست عمل خواهد کرد.

برای اینکه از نحوه عملکرد و درستی الگوریتم مطمئن شویم آن را روی مثال لیست بالا اجرا می کنیم. مراحل در شکل زیر نشان داده شده است.



اجرای دستی الگوریتم روی یک لیست ورودی. این شکل حرکت در لیست برای درج عنصر در هیپ را با علامت ^ دنبال می کند و عنصر مرتب شده را با قرار دادن خط زیر آن مشخص می کند. همزمان مقادیر و ساختار هیپ را نشان می دهد.

با توجه به درست بودن اجرا روی مثال از درستی الگوریتم اطمینان حاصل می کنیم.

سپس شروع به پیاده سازی الگوریتم می کنیم. ابتدا ماژولهایی که برای ایجاد یک درخت هیپ لازم است را در برنامه وارد می کنیم (خط 1). سپس تابع با پارامترهای مناسب را تعریف میکنیم (خط 3). در ابتدای تابع یک لیست خالی تعریف می کنیم که برای نگهداری درخت هیپ بکار خواهیم برد (خط 4). سپس ابتدا با پیشروی در لیست به اندازه k خانه یک درخت هیپ k عنصری ایجاد می کنیم (خط 5 تا 6). سپس در یک حلقه تکرار هر بار کوچکترین عنصر درخت هیپ را از آن برمی داریم و در موقعیت i ام لیست می گذاریم. آنگاه عنصر بعدی که در درخت هیپ درج نشده است را در آن درج می کنیم (خطوط 10 و 11). آنگاه سراغ عنصر بعدی لیست میرویم (خط 12). زمانی که درخت هیپ خالی شود یعنی همه عناصر را در جای درستشان نشانده ایم.

```
1. from heapq import heappush, heappop
2.
3. def sort_k_unsorted_array(arr, k):
4.  knext = []
5.  for _ in range(min(k+1, len(arr))):
6.    heappush(knext, arr[_])
7.  i = 0
8.  while knext:
9.    arr[i] = heappop(knext)
10.   if i < len(arr)-k-1:
11.     heappush(knext, arr[i + k + 1])
12.   i += 1
13. return arr
```

۳.۱۱ لیست های مرتب شده را ترکیب کنید

پرسش: برنامه ای بنویسید که k لیست مرتب شده را در هم ترکیب نماید و یک لیست کل مرتب شده به طول n را برگرداند.

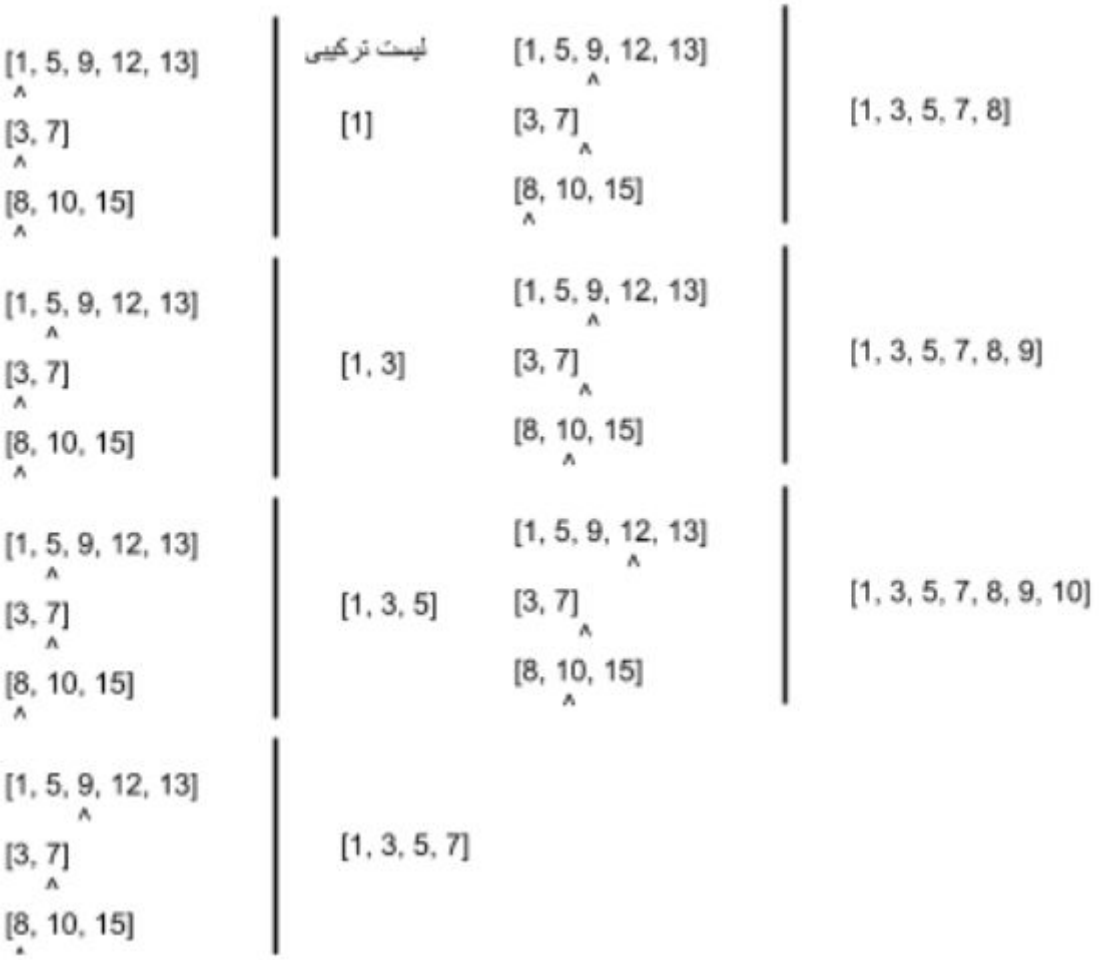
پاسخ: حل مسئله را با یک مثال شروع می کنیم. اگر سه لیست مرتب شده زیر را داشته باشیم:
[13, 12, 9, 5, 1] و [7, 3] و [15, 10, 8] آنگاه حاصل ترکیب آنها یک لیست به طول 10 و به صورت [15, 13, 12, 10, 9, 8, 7, 5, 3, 1] خواهد بود.

یک راه حل بدیهی برای این مسئله آن است که لیست ها را بدون در نظر گرفتن ترتیب به هم وصل کنیم و سپس لیست کل بوجود آمده را مرتب نماییم تا به لیست کل مرتب دست یابیم. پیچیدگی محاسباتی این الگوریتم (O(n*log(n)) می باشد. اشکال این الگوریتم آن است که از خاصیت مرتب بودن لیست ها استفاده نمی کند و بنابر این کار اضافی انجام می دهد.

برای یک راه حل بهینه تر می توانیم از خاصیت مرتب بودن لیست ها استفاده کنیم تا یک به یک عناصر را در جای درستشان بنشانیم. مثلا مشاهده می کنیم که کوچکترین عنصر لیست کل مرتب اولین عنصر یکی از لیست ها می باشد. در مثال بالا، این مقدار 1 است که از لیست اول است زیرا کوچکترین مقدار بین کوچکترین مقادیر لیست ها است. برای یافتن کوچکترین عنصر بعدی باید عنصر دوم از لیست اول به همراه عنصر اول لیست های دیگر در نظر گرفته شوند. با دنبال کردن این مثال به یک الگوریتمی می رسیم.

در یک الگوریتم بهینه هر بار مینیمم بین کوچکترین عنصر استفاده نشده لیست ها را می یابیم و در انتهای لیست ترکیبی درج می کنیم.

در یک مصاحبه باید الگوریتم مطرح شده را روی یک مثال اجرا کنیم تا مطمئن شویم که درست کار می کند. شکل زیر قسمتی از این اجرا را نشان می دهد.



اجرای یک مثال روی الگوریتم. علامت ^ اولین عنصر استفاده نشده از سمت چپ هر لیست را نشان می دهد. مراحل اجرای الگوریتم را از بالا به پایین و از چپ به راست دنبال کنید.

بنابر این کلیات الگوریتم درست کار می کند.

برای پیاده سازی الگوریتم فوق نیاز به یک لیست از مینیمم ها داریم که از k عنصر که هر یک مینیمم استفاده نشده در یکی از لیستها است تشکیل می شود. در این الگوریتم هر بار (۱) مقدار مینیمم بین این k مقدار را پیدا کنیم و (۲) آن را پس از استفاده از لیست مینیمم ها حذف کنیم، و (۳) مقدار بعدی استفاده نشده از لیستی که مقدار مینیمم از آن انتخاب شد را در لیست مقادیر مینیمم درج کنیم.

پیدا کردن مینیمم بین یک لیست به طول k دارای پیچیدگی محاسباتی O(k) می باشد. با توجه به اینکه این لیست بروز می شود و هر بار باید مقدار مینیمم جدید را محاسبه کنیم، این الگوریتم دارای پیچیدگی محاسباتی O(n*k) خواهد بود.

واقعیت این است که اگر در لیست k عنصری مینیمم ها تنها مقدار عناصر را بگذاریم نمی توانیم پس از یافتن عنصر مینیمم به راحتی بفهمیم که این عنصر مربوط به کدام یک از لیستها بوده است. بنابراین همراه با مقادیر، اندیس لیستی که مقدار متعلق به آن است و اندیس موقعیت عنصر در آن لیست باید نگهداری شود.

با توجه به اینکه نیاز به یک لیست داریم که بتواند سریع عنصر مینیمم را بیابد بنظر می رسد که درخت هیپ می تواند موثر باشد. قبل از محاسبه هزینه کل الگوریتم در صورت استفاده از یک درخت هیپ به جای یک درخت معمولی، باید موارد ذیل را در مورد یک درخت هیپ با اندازه k یادآوری کنیم:

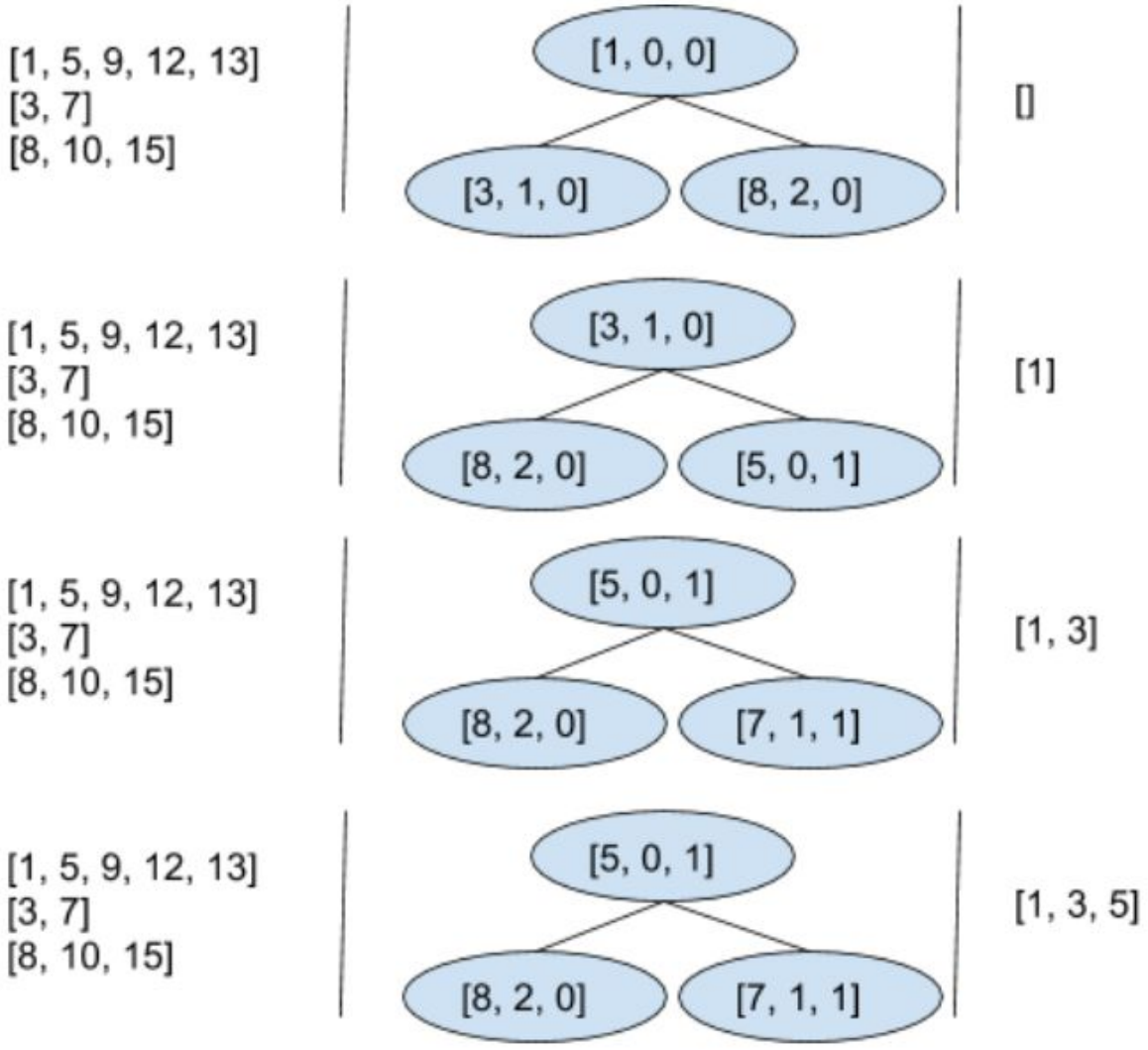
۱- هزینه یافتن عنصر مینیمم O(1) می باشد.

۲- هزینه حذف عنصر مینیمم و بازسازی درخت هیپ O(log(k)) می باشد.

۳- هزینه اضافه کردن یک عنصر و بازسازی درخت هیپ O(log(k)) می باشد.

در الگوریتم فوق این اتفاقات در ارتباط با درخت هیپ اتفاق می افتد: هربار عنصر مینیمم از لیست مینیمم ها را پیدا می کنیم. این کار در درخت هیپ دارای هزینه O(1) می باشد. سپس باید این عنصر را از لیست مینیمم ها حذف کنیم که هزینه آن O(log(k)) می باشد. نهایتا باید عنصر بعدی از لیستی که عنصری از آن را استفاده کردیم را به لیست مینیمم ها اضافه کنیم که هزینه آن log(k) می باشد. این کار باید برای همه n عنصر لیست کل انجام شود. بنابراین هزینه این الگوریتم در صورت استفاده از درخت هیپ O(n*log(k)) هست که سرعت آن بسیار بهتر از سرعت نسخه اول این الگوریتم است که از یک لیست عادی استفاده می کند.

برای اینکه مطمئن شویم که الگوریتم با استفاده از درخت هیپ درست کار می کند آن را روی لیست های داده شده مثال بالا اجرا می کنیم. به دلایلی که در بکارگیری لیست مینیمم ها نیز گفته شد، یک گره درخت هیپ مینیمم ها تنها شامل مقدار مینیمم از یکی از لیستها نیست، بلکه اندیس لیستی که مقدار متعلق به آن است و اندیس موقعیت عنصر در آن لیست نیز در گره نگهداری می شوند. شکل زیر مقادیر درخت هیپ در حین اجرای الگوریتم را نشان می دهد:



هر سطر نشان دهنده یک گام برای یافتن یک عنصر مینیمم است. در هر گره فوق مقداری از یک لیست، اندیس لیست، و اندیس عنصر در لیست قرار می گیرد.

با توجه به مثال فوق الگوریتم روی مثال بالا به درستی عمل می کند. بنابراین آن را پیاده می کنیم.

ابتدا ماژول های مناسب را برای استفاده در ساختمان داده درخت هیپ در برنامه وارد می کنیم (خط 1). سپس نام تابع و پارامترهای مناسب را تعریف می کنیم (خط 2). تعداد لیست های داده شده را می شماریم (خط 3). سپس همه عناصر اول لیستها را در یک درخت هیپ می گذاریم. برای هر عنصر علاوه بر مقدار اندیس لیستی که از آن انتخاب شده اند و اندیس عنصر (0 که نشان دهنده عنصر اول است) را نیز قرار می دهیم (خط 4). سپس لیست را تبدیل به درخت هیپ میکنیم (خط 5). آنگاه یک لیست خالی از جواب ها می سازیم (خط 6). سپس تا زمانی که درخت هیپ تمام نشده است یک حلقه تکرار را انجام می دهیم (خط 8 تا 12). هر بار عنصر بالای درخت هیپ را به عنوان مینیمم تا این نقطه برمی داریم (خط 9).

سپس آن را به لیست جوابها اضافه میکنیم (خط 10). سپس مقدار بعدی از لیستی که عنصر آن استفاده شده را به درخت هیپ اضافه می کنیم (خط 11 و 12). با اتمام حلقه لیست پاسخ را برمی گردانیم (خط 13).

```
1. from heapq import heappush, heappop, heapify
2.
3. def merge_k_sorted_array(arrs):
4.   k = len(arrs)
5.
6.   heap = [(arrs[i][0], i, 0) for i in range(k)\
7.           if len(arrs[i])>0]
8.   heapify(heap)
9.   res = []
10.
11.  while heap:
12.    minv, arr_index, index = heappop(heap)
13.    res.append(minv)
14.    if index<len(arrs[arr_index])-1:
15.      heappush(heap, (arrs[arr_index][index+1],
16.                      arr_index, index+1))
17.
18.  return res
```

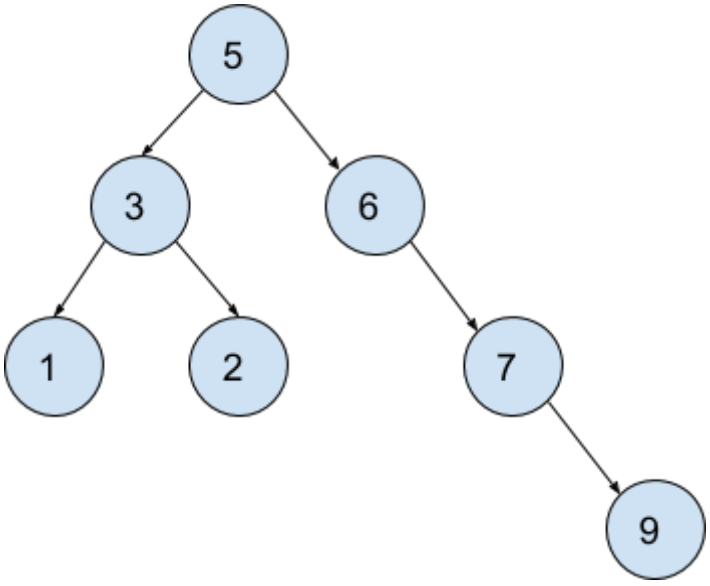
در یک مصاحبه واقعی برنامه را دوباره مرور می کنیم و با داده های تستی امتحان می کنیم.

۱۲. درخت جستجوی دودویی (Binary Search Tree)

۱.۱۲ درخت جستجوی دودویی

درخت جستجوی دودویی (به اختصار درجد -- می خوانیم درژد بر وزن درخت) نوع خاصی از درختان دودویی است. ویژگی اصلی این درخت آن است که مقادیر درخت با نظم خاصی در آن قرار می گیرند به این صورت که همه مقادیر در زیر درخت چپ هر گره از آن کوچکتر و تمام مقادیر در زیر درخت راست آن بزرگتر از آن می باشند. این خاصیت حل مسائل ویژه ای را امکان پذیر می کند. به عنوان مثال می توان یک مقدار در چنین درختی را به صورت متوسط با $O(log\ n)$ مقایسه پیدا نمود. این بهبود قابل ملاحظه ای در مقایسه با جستجوی خطی در یک لیست بی نظم می باشد.

مثال: درخت دودویی زیر یک درخت جستجوی دودویی است.



۲.۱۲. یافتن تعداد عناصر کوچکتر بعدی

پرسش: برنامه ای بنویسید که یک لیست از اعداد دریافت نماید و برای هر عنصر لیست تعداد عناصر کوچکتر بعد از آن را برگرداند.

پاسخ: حل مسئله را با یک مثال شروع می کنیم تا مطمئن شویم که آن را درست درک کرده ایم. فرض کنیم لیست [5, 3, 7, 11 , 2, 4] را داریم. دو عنصر با مقادیر 2 و 3 بعد از اولین 4 قرار دارند و کوچکتر از آن می باشند. هیچ عنصری بعد از 2 کوچکتر از آن نیست و هر سه مقدار 7، 3، و 5 از 11 کوچکتر می باشند. بنابراین برنامه باید لیست [0, 0, 2, 3, 0, 2] را به عنوان جواب برگرداند.

ارایه الگوریتم را از روش جستجوی کامل (brute force) شروع می کنیم تا هم بیشتر مسئله را درک کنیم و هم نشان دهیم که می توانیم حداقل یک راه حل ساده برای مسئله بیابیم.

در روش جستجوی کامل از سمت چپ لیست شروع می کنیم و برای هر عنصر تمام عناصر بعد آن را تا آخر لیست یک به یک بررسی می کنیم. از یک شمارنده کمک می گیریم تا تعداد عناصری در لیست که کوچکتر از آن می باشند را بشماریم. بنابراین راه حل جستجوی کامل از دو حلقه تکرار تودرتو تشکیل می شود که حلقه بیرونی سراغ تک تک عناصر لیست می رود و حلقه درونی از آن عنصر تا انتهای لیست را می پیماید تا تعداد عناصر کوچکتر از آن را بشمارد. پیچیدگی محاسباتی این الگوریتم ($O(n \wedge 2$ می باشد.

پیاده سازی این الگوریتم بسیار ساده است اما خود الگوریتم کند است. مشکل این الگوریتم کند بودن حلقه درونی است که باید مابقی لیست را تا آخر بپیماید تا تعداد عناصر کمتر از یک عنصر را بشمارد.

اما اگر عناصر بعد از یک عنصر مرتب بودند این مشکل وجود نداشت. مثلا فرض کنیم در مثال قبل عناصر بعد از 4 را ابتدا مرتب کنیم و در یک لیست جدا به صورت [11, 7, 5, 3, 2] قرار دهیم. آنگاه اندیس اولین عنصری که بزرگتر مساوی 4 هست تعداد عناصر کوچکتر از آن را نشان می دهد. در مثال بالا اولین عنصر بزرگتر از 4، عنصر 5 می باشد که اندیس آن 2 می باشد. در یک لیست مرتب شده عمل جستجوی برای یافتن اولین عنصر بزرگتر از یک مقدار خواسته شده بسیار سریعتر از جستجوی خطی است. در واقع اگر لیستی از n عنصر مرتب داشته باشیم با استفاده از جستجوی دودویی، جستجو نیاز به

حداکثر $\log(n)$ مقایسه خواهد داشت. بنابراین اگر عناصر بعد از هر عنصر را مرتب کنیم می توانیم به این روش تعداد عناصر کوچکتر از آن را بیابیم.

اما مشکل اصلی آن است که برای هر عنصر باید ابتدا تمام عناصر آن را مرتب کنیم تا بتوانیم از جستجوی سریع استفاده نماییم.

یک راه حل به ظاهر احمقانه برای حل این مشکل این است که تنها یکبار لیست را مرتب کنیم و قبل از یافتن تعداد عناصر کوچکتر از هر عنصر آن را از لیست مرتب شده حذف کنیم. با این کار لیست مقادیر بعد از هر عنصر را به صورت مرتب شده خواهیم داشت و می توانیم در آن سریع به دنبال اولین عنصر بزرگتر از آن دست یابیم. نهایتا می توانیم تعداد عناصر کوچکتر از آن را محاسبه کنیم. مشکل این الگوریتم آن است که هر بار باید یکی از عناصر را از لیست مرتب شده حذف کنیم. متاسفانه حذف یک عنصر از یک لیست هزینه بالایی دارد و پیچیدگی محاسباتی آن $O(n)$ می باشد.

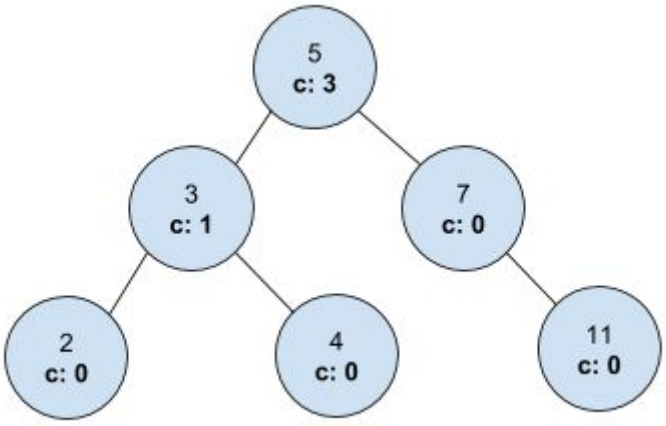
معادل ایده احمقانه بالا آن است که بجای پیمایش لیست از چپ به راست، آن را از راست به چپ بپیماییم. در این صورت بجای مرتب سازی کل لیست و حذف عناصر مصرف شده، با حرکت از راست به چپ عناصر پیمایش شده را در محل مناسب لیست مرتب درج می کنیم. این لیست مرتب در ابتدا خالی است و عناصر بگونه ای به آن اضافه می شود که مرتب بماند. با این کار عمل درج در یک لیست مرتب جایگزین حذف از یک لیست مرتب می شود که باز هم پیچیدگی محاسباتی $O(n)$ دارد.

روش احمقانه دوم تفاوتی سرعتی با روش قبل ندارد اما جرقه ای در ذهن برای استفاده از ساختمان داده ای را می دهد که امکان جستجوی سریع با سرعتی که در یک لیست مرتب شده ممکن است را می دهد و همزمان هزینه درج در آن بسیار کمتر از درج در یک لیست است. این ساختان داده دِرَجد می باشد. یک درجد مشابه یک لیست مرتب شده است با این تفاوت که درج و حذف عناصر آن با پیچیدگی محاسباتی $O(\log(n))$ انجام می شود.

اما درجد یک کمبود هم دارد و آن اینکه نمی توان با اندیس به عناصر درجد دست یافت. به همین دلیل نمی توان تعداد عناصر کوچکتر از یک عنصر در یک درجد را بدون پیمایش همه عناصر کوچکتر از آن شمرد.

همچنان تسلیم نمی شویم. گویا چیزی در پس این ایده است که اگر آن را بفهمیم مشکل حل می شود.

شاید بتوانیم تغییری در درجد ایجاد کنیم که این کار را راحت کند؟ خاصیتی که درجد تغییر یافته در این مسئله نیاز دارد آن است که شمارش تعداد عناصر کوچکتر از یک عنصر را آسان کند. برای این کار میتوانیم یک فیلد به گره های درخت اضافه کنیم که تعداد گره های زیر درخت چپ آن گره را در خود نگه دارد. اسم این فیلد جدید را شمارنده (counter) می گذاریم و آن را با C نشان می دهیم. به عنوان مثال درجد با فیلد اضافی برای لیست [5, 3, 7, 11, 2, 4] با فرض اینکه عناصر را از راست به چپ در درخت درج کنیم این است:



نمونه ای از مقادیر گره ها و فیلد اضافی آن در درخت دودویی در انتهای اجرای الگوریتم

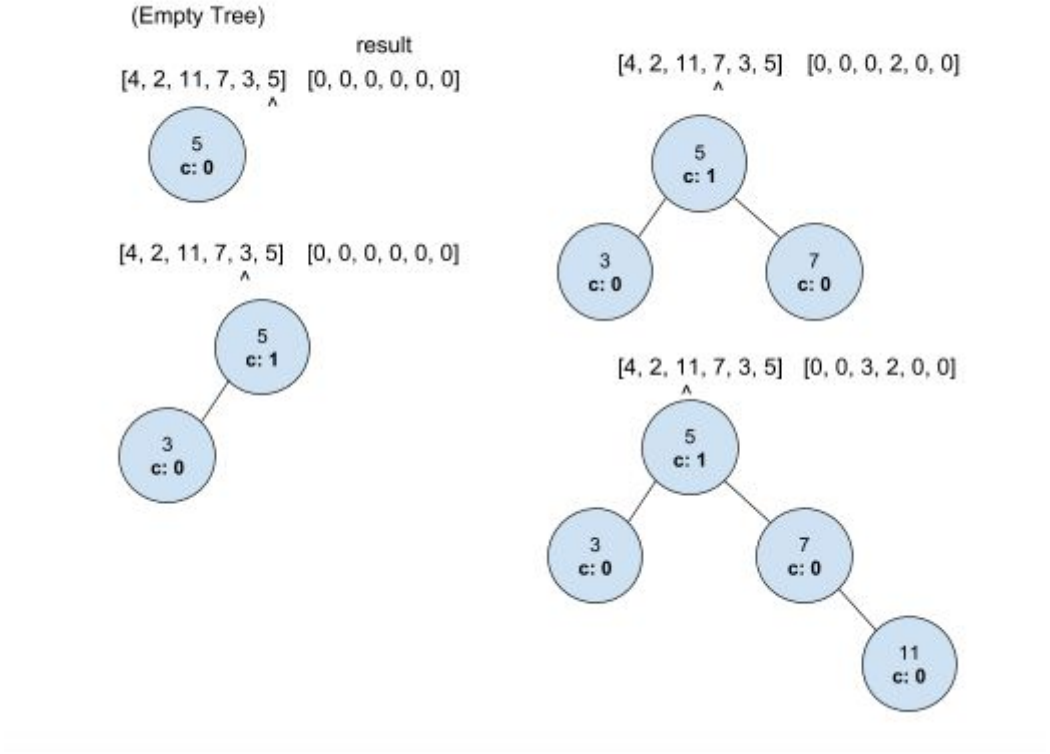
در این شکل مقادیر مربوط به شمارنده را توسط "c:" مشخص نموده ایم. برای محاسبه تعداد عناصر کوچکتر از یک مقدار داده شده باید درخت را پیمایش کنیم و مقدار مجموع شمارنده "c:" را به روش خاصی جمع بزنیم. برای این کار هرگاه در مسیر رسیدن به عنصر مورد نظر (که می خواهیم تعداد عناصر کوچکتر از آن را بیابیم) به سمت زیر درخت راست یک گره حرکت کنیم باید مقدار شمارنده آن گره بعلاوه یک را به مجموع اضافه کنیم. به عنوان مثال برای یافتن تعداد عناصر کوچکتر از 11 مقادیر (1+3) و (1+0) را با هم جمع می کنیم که برابر با 5 می باشد.

با استفاده از این تغییر کوچک می توانیم تعداد عناصر کوچکتر از یک عنصر را با تنها با صرف $O(\log(n))$ بیابیم. به این دلیل که این مقادیر را در زمان درج عناصر در درخت جستجوی دودویی بروز می کنیم، بار اضافی برای محاسبه اولیه این مقادیر نخواهیم داشت.

بنابراین کلیت الگوریتم جدید آن است که از سمت راست لیست شروع می کنیم. برای شمارش سریع تعداد عناصر کوچکتر در سمت راست هر عنصر از یک درجد استفاده می کنیم. این درجد در ابتدا خالی است. با حرکت از راست به چپ لیست ابتدا

تعداد عناصر کوچکتر از آن را با استفاده از درجد محاسبه می کنیم. سپس خود این عنصر را در درجد درج می کنیم و مقادیر مربوط به شمارنده تعداد عناصر کوچکتر از گره ها را بروز می کنیم تا برای محاسبه تعداد عناصر کوچکتر عنصر بعدی در لیست آماده باشیم.

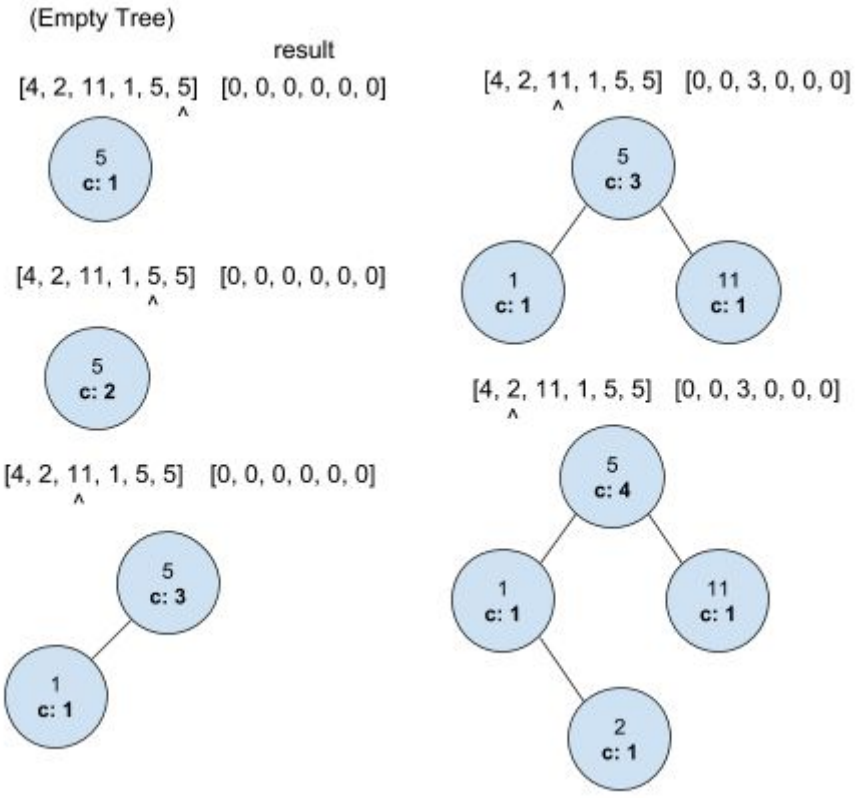
برای اینکه از درستی الگوریتم مطمئن شویم آن را روی مثال ورودی بالا مرحله به مرحله اجرا می کنیم.



روند بروز رسانی درجد برای یک مثال ورودی

مثال بالا نشان می دهد که الگوریتم در صورتی که مقادیر عناصر لیست یکتا باشند به درستی عمل می کند.

در صورتی که مقادیر عناصر تکراری باشند نیاز به تغییر رویه کوچکی در محاسبه شمارنده در گره ها هستیم. به این صورت که در زمان درج یک عنصر جدید در درجد مقدار شمارنده را با حرکت به سمت چپ یک گره یا مشاهده مقدار مساوی با آن یک واحد افزایش می دهیم. در ضمن به شمارنده یک گره جدید مقدار اولیه 1 می دهیم. این مشکل مربوط به مقادیر تکراری را حل می کند. با یک مثال دیگر که شامل عناصر تکراری نیز هست بررسی می کنیم که آیا این تغییر درست عمل می کند. فرض کنید لیست ورودی برابر با [5, 5, 1, 11, 2, 4] است. دنباله درخت های زیر وضعیت درجد را پس از مشاهده مقادیر مختلف نشان می دهد.



مثال فوق نشان می دهد که این تغییر می تواند پاسخگوی ورودی با تکرار نیز می باشد. با توجه به اینکه تست های ابتدایی نشان می دهد که الگوریتم درست است، با گرفتن تایید از مصاحبه کننده بسراغ پیاده سازی الگوریتم می رویم.

در ابتدا یک گره از درجد که تغییر یافته است را تعریف می کنیم. یک متغیر شی به نام leq_counter برای نگه داشتن شمارش عناصر مساوی با یک عنصر یا کوچکتر از آن که در سمت چپ آن هستند در نظر میگیریم (خطوط 7-1). این متغیر معادل 'c:' در شکلهای بالاست.

سپس یک تابع با پارامترهای مناسب تعریف میکنیم (خط 8). اگر لیست خالی باشد باید یک لیست خالی را به عنوان پاسخ برگردانیم (خط 9 و 10). سپس ریشه درجد را با ایجاد یک گره بر اساس آخرین عنصر لیست است (خط 13). این کار را به این دلیل انجام می دهیم که نتیجه را برای آخرین عنصر لیست میدانیم (تعداد عناصر کوچکتر بعد آن همیشه 0 است) و محاسبات را عمل از عنصر یکی به آخر شروع می کنیم. سپس لیست نتایج را با صفر مقدار دهی اولیه می کنیم (خط 14). در یک حلقه تکرار از عنصر یکی به آخر لیست شروع می کنیم و به سمت چپ لیست (عناصر با اندیس کوچکتر) حرکت می کنیم (خط 16). سپس در یک حلقه تکرار (خط 20 تا 27) با شروع از ریشه درخت جای درست این عنصر را در درجد می یابیم و همزمان با این کار مقدار شمارنده گره ها را بروز می کنیم. برای این کار یک متغیر به اسم c_node که ابتدا به ریشه درخت

اشاره می کند و یک متغیر به اسم p_node که ابتدا مقدار نال (None) دارد تعریف می کنیم (خط 17). این دو اشاره گر در هنگام حرکت به سمت محل مناسب برای درج عنصر i ام بروز می شوند. سپس یک متغیر برای شمارش تعداد عناصر کوچکتر از عنصر iام که می خواهیم آن را در درجد درج کنیم با مقدار اولیه 0 در تعریف می کنیم (خط 18). با حرکت به سمت راست گره ها مقدار شمارنده گره ها را با این شمارنده جمع می کنیم.

سپس مقدار c_node را درون متغیر p_node نگه می داریم به این خاطر که c_node می خواهید یک سطح به سمت پایین درخت حرکت کند و چنانچه سطح بعدی نال باشد محل درج عنصر i ام را از دست ندهیم (خط 21). سپس مقدار عنصر i ام را با عنصر c_node مقایسه می کنیم. در صورتی که عنصر i ام کوچکتر از c_node باشد محل درج آن در سمت چپ c_node می باشد. بنابراین باید متغیر شی مربوط به عنصر c_node را یک واحد بیفزاییم و سپس به سمت چپ آن حرکت کنیم (خط 22 تا 24). در صورتی که عنصر iام بزرگتر از c_node باشد محل درج آن در سمت راست c_node می باشد. این همچنین به این مفهوم است که همه عناصر سمت چپ c_node از عنصر i ام کوچکترند و بنابر این تعداد آنها (مقدار متغیر شی leq_counter) باید به متغیر counter که تعداد چنین عناصری را می شمارد اضافه شود (خطوط 25 تا 27).

پس از انتهای حلقه درونی محل درست درج عنصر iام در سمت راست یا چپ p_node می باشد. بنابراین با مقایسه مقدار عنصر iام با مقدار p_node آن را در سمت راست یا چپ این گره درج میکنیم (خطوط 29 تا 32). نهایتا تعداد عناصر کوچکتر از آن در لیست نتایج res قرار می گیرد (خط 34).

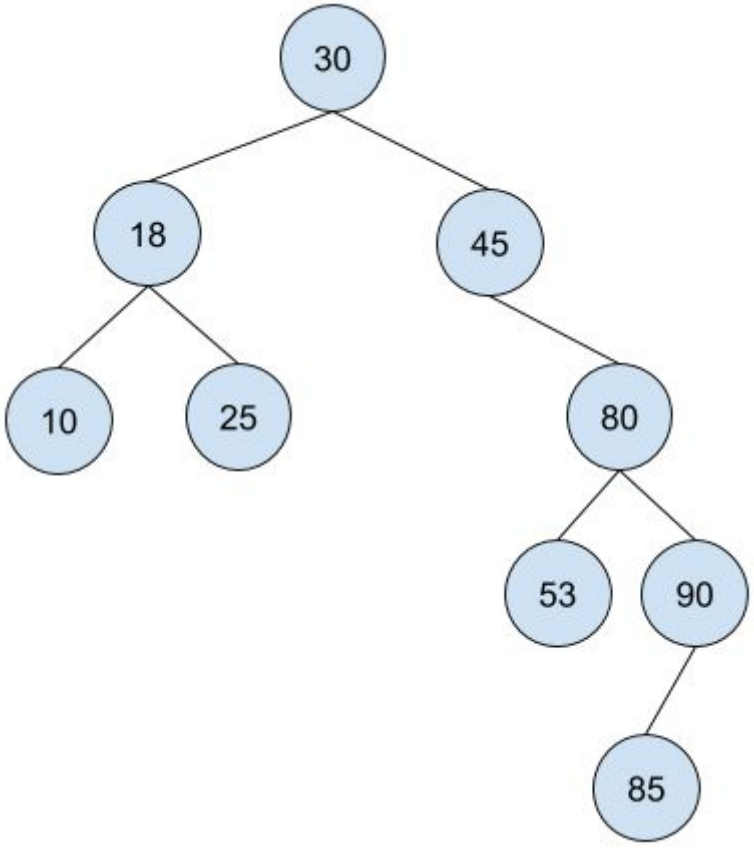
```
1. class Node(object):
2.     def __init__(self, value):
3.         self.value = value
4.         self.left  = None
5.         self.right = None
6.         self.leq_counter = 1
7.
8. def countSmaller(nums):
9.         if nums == []:
10.            return []
11.
12.        len_nums = len(nums)
13.        root = Node(nums[len_nums-1])
14.        res = [0] * len_nums
15.
16.        for i in range(len_nums-2, -1, -1):
17.            i_val, c_node, p_node = nums[i], root, None
18.            counter = 0
19.
20.            while c_node:
21.                p_node = c_node
22.                if i_val<=c_node.value:
23.                    c_node.leq_counter += 1
24.                    c_node = c_node.left
25.                elif c_node.value<i_val:
26.                    counter += c_node.leq_counter
27.                    c_node = c_node.right
28.
29.            if p_node.value<i_val:
30.                p_node.right = Node(i_val)
31.            elif i_val<p_node.value:
32.                p_node.left = Node(i_val)
33.
34.            res[i] = counter
35.
36.        return res
```

با توجه به اینکه برای هر i حداکثر log(n) جستجو در درجد نیاز است، پیچیدگی محاسباتی این الگوریتم O(n*log(n)) می باشد.

۲.۱۲ محاسبه مجموع مقادیر در یک بازه

پرسش: یک درجد و دو عدد L و R (که L<=R) داده شده اند. برنامه ای بنویسید که مجموع تمام مقادیر درون درخت بین L و R (شامل خود این مقادیر در صورت وجود) را محاسبه نماید.

پاسخ: کار را با یک مثال شروع میکنیم تا مطمئن شویم که آن را خوب درک کرده ایم. فرض کنیم که درجد زیر و مقادیر L=19 و R=60 داده شده اند.



نمونه ای از یک درجد به عنوان ورودی برنامه

آنگاه برنامه باید مقدار 153 (مجموع 25, 30, 45, 53) را برگرداند.

حل مسئله را با ارائه یک الگوریتم جستجوی کامل شروع می کنیم. در این پرسش الگوریتم جستجوی دودویی به معنای آن است که کل درخت را پیمایش کنیم و برای هر گره چک کنیم که آیا مقدار در بازه بین L و R می باشد.

اشکال این روش آن است که بسیاری از زیر درخت ها که شامل پاسخ نمی باشند را نیز در آن می پیماییم. درحالیکه مثالا وقتی به گره با مقدار 80 می رسیم می دانیم که عنصری با مقدار بین 19 و 60 در سمت راست آن وجود ندارد. این خاصیت درجد است زیرا تمام عناصر زیر درخت راست یک گره از آن بزرگتر می باشند. این مشاهده ایده یک الگوریتم بهینه را می دهد.

در یک الگوریتم بهینه سه حالت برای مقدار یک گره در نظر میگیریم:

(۱) مقدار گره کوچکتر از L می باشد. در این صورت از زیر درخت سمت چپ آن صرف نظر میکنیم چون هیچ عنصری بزرگتر از L در آن وجود ندارد. بنابر این کافی است زیر درخت سمت راست آن را برای یافتن مقادیر در بازه بین L و R جستجو کنیم.

(۲) در صورتی که مقدار ریشه بزرگتر از R باشد تنها باید زیر درخت سمت چپ آن را بجوییم و از زیر درخت سمت راست صرفنظر می کنیم.

(۳) چنانچه مقدار ریشه در بازه L و R باشد آن را به مجموع اعداد بازه L و R اضافه می کنیم و هر دو زیر درخت سمت چپ و راست را برای یافتن مابقی عناصر در بازه داده شده جستجو می کنیم.

این الگوریتم یک تعریف بازگشتی دارد و بنابراین می تواند به صورت بازگشتی پیاده شود. در یک مصاحبه واقعی، با اجرای الگوریتم روی مثال بالا مطمئن میشویم که به درستی عمل می کند. سپس شروع به پیاده سازی می کنیم.

ابتدا برای تابع نام و پارامترهای مناسب انتخاب می کنیم (خط 1). چنانچه مقدار ریشه نال (null) باشد آنگاه مقدار 0 را برمی گردانیم (خطوط 2 و 3). اگر مقدار ریشه بزرگتر از R باشد زیر درخت سمت چپ آن را جستجو می کنیم (خط 4 و 5). در صورتی که مقدار ریشه کمتر از L باشد زیر درخت سمت راست آن را جستجو می کنیم (خط 6 و 7). در غیر این صورت مقدار ریشه را به حاصل محاسبه شده از هر دو زیر درخت سمت چپ اضافه و آن را برمی گردانیم (خطوط 8 و 9).

```
1. def LtoRSumDrajd(root, L, R):
2.       if not root:
3.           return 0
4.       if R<root.val:
5.           return LtoRSumDrajd(root.left, L, R)
6.       if root.val<L:
7.          return LtoRSumDrajd(root.right, L, R)
8.       return root.val + LtoRSumDrajd(root.left, L, R) +\
9.              LtoRSumDrajd(root.right, L, R)
```

این الگوریتم را می توان به صورت غیر بازگشتی، با استفاده از ساختمان داده صف نیز پیاده سازی کنیم. از ساختمان داده صف برای دنبال کردن ریشه زیر درخت هایی که ممکن است مقادیری در بازه مناسب داشته باشند استفاده می شود. تکه کد زیر این کار را انجام می دهد.

```
1. from collections import deque
2. def rangeSumDrajd(root, L, R):
3.   s = 0
4.   queue = deque([root])
5.   while queue:
6.      c = queue.popleft()
7.      if c:
8.          if R<c.val:
9.              queue.append(c.left)
10.          elif c.val<L:
11.              queue.append(c.right)
12.          else:
13.              s += c.val
14.              queue.append(c.left)
15.              queue.append(c.right)
16.  return s
```

در هر دو الگوریتم فوق، با توجه با اینکه حداکثر به اندازه ارتفاع درخت گره نامرتبط را پیمایش می کنیم و مابقی گره ها در بازه R تا L می باشند، در صورتی که L و R خود در بازه مقادیر درخت باشند، پیچیدگی محاسباتی این الگوریتم ((O(log(n) + (R-L می باشد. در بدترین حالت پیچیدگی محاسباتی O(n) می باشد.

۱۳. گراف

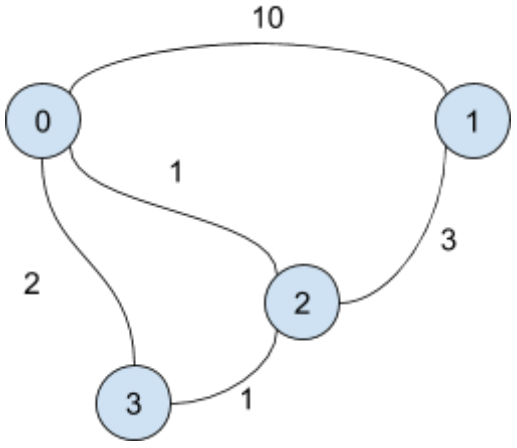
۱.۱۳ گراف

گراف یک ساختمان داده بسیار مهم است که در پاسخ به بسیاری از مسائل برنامه نویسی به کار می آید. داده ها در برخی از مسائل ذاتا ساختار گرافی دارند. به عنوان مثال نقشه شهرها و جاده ها که در برنامه نقشه گوگل استفاده می شود ساختار گراف دارد. در برخی مسایل دیگر مدل کردن داده ها به صورت گراف به ما کمک می کند که بتوانیم از الگوریتم های گراف به سرعت برای یافتن پاسخ استفاده نماییم. مثلا یک برنامه مدیریت پروژه که شامل مراحل انجام پروژه به همراه وابستگی نتایج آنهاست را می توان به صورت یک گراف در نظر گرفت. با این کار می توان از الگوریتم های گراف برای پاسخ به بسیاری از سوالات در مورد پروژه، شامل کوتاه ترین مسیر تکمیل پروژه، و زمان پایان یافتن پروژه استفاده کرد.

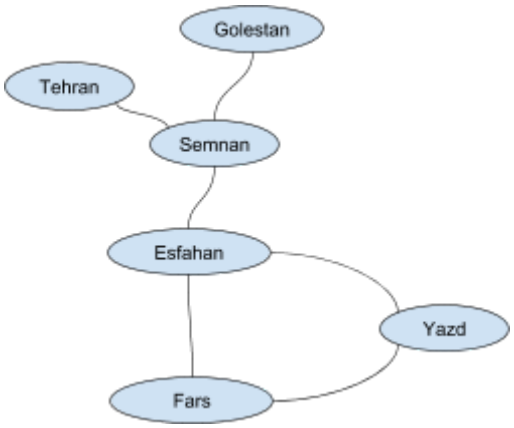
نگهداری گراف در برنامه

گراف را به دو روش عمده می توان در برنامه نویسی پیاده نمود. روش اول استفاده از یک ماتریس است. ماتریس ذیل گراف الف را نمایش می دهد.

```
g = [[0,   10,  1,   2],
     [10,  0,   3,   0],
     [1,   3,   0,   1],
     [2,   0,   1,   0]]
```



گراف مشابه زیر را می توان با یک دیکشنری از لیست ها در برنامه نگهداری نمود.




```
g = {"Golestan": ["Semnan"],
     "Semnan": ["Tehran", "Esfahan"],
     "Tehran": ["Semnan"],
     "Esfahan": ["Semnan", "Yazd", "Fars"],
     "Fars": ["Yazd", "Esfahan"],
     "Yazd": ["Esfahan", "Fars"]
    }
```

همانطور که می بینید تفاوت نوع اتصالات و نحوه رجوع به گره ها جزو دلایلی هستند که یک پیاده سازی گراف بین این دو روش را انتخاب می کنیم.

۲.۱۳ یافتن دنباله کلمات

پرسش: یک کلمه شروع، یک کلمه پایان، و یک مجموعه از کلمات که همگی هم طولند داده شده اند. هدف، تبدیل کلمه شروع به کلمه پایان با استفاده از دنباله ای از تغییرات است با این محدودیت که در هر تغییر تنها یک حرف از یک کلمه عوض می شود. کلمات میانی دنباله باید متعلق به لیست مجموعه کلمات داده شده باشند. برنامه ای بنویسید که طول کوتاهترین دنباله تبدیل برای رسیدن از کلمه شروع به پایان را برگرداند.

پاسخ: طبق معمول کار را با یک مثال آغاز می کنیم تا مطمئن شویم که مسئله را درست درک کرده ایم. به عنوان مثال اگر کلمه شروع tell و کلمه پایان book باشد و مجموعه کلمات داده شده cool, bell, tool, wool, wage, rage,cook, toll, cage باشند، آنگاه دنباله tell ->toll-> tool-> cool-> cook->book کوتاه ترین دنباله تبدیل می باشد.

قبل از اینکه در مورد راه حل این مسئله بحث کنیم بگذارید روی یک اصطلاح توافق کنیم: اگر کلمه ای با کلمه ی دیگر تنها در یک حرف متفاوت باشد آنها را 'مجاور' هم می نامیم. مثلا toll و tool دو کلمه مجاورند.

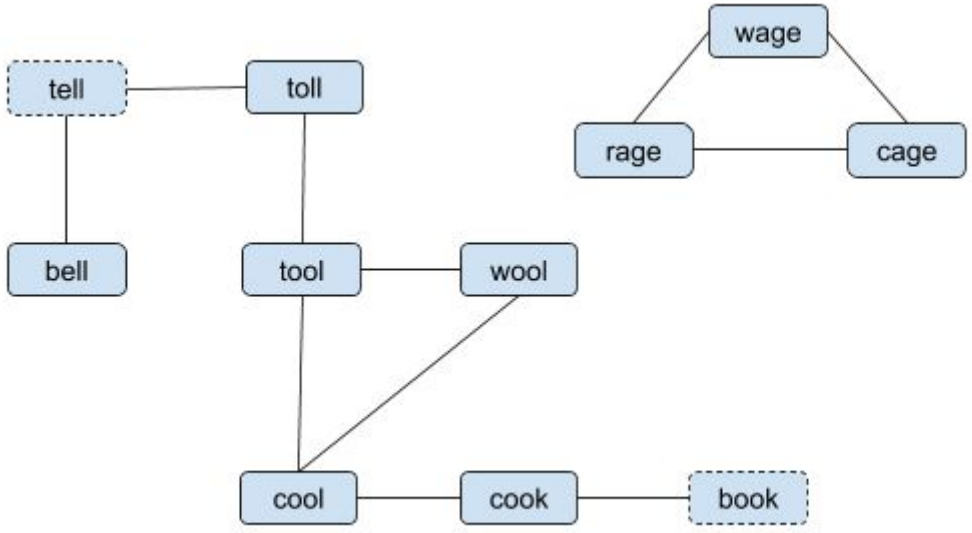
اولین الگوریتمی که به آن فکر می کنیم روش جستجوی کامل است. در این روش باید تمام دنباله های کلمات مجاور که رشته شروع را به پایان را می رسانند ایجاد نماییم و در انتها کوچکترین دنباله را به عنوان پاسخ برگردانیم.

برای ساخت هر دنباله باید از کلمه شروع آغاز کرد، یک کلمه مجاور از لیست کلمات یافت، و به انتهای دنباله اضافه نمود. در واقع در هر نقطه از یک دنباله باید w (تعداد کل کلمات) را با کلمه کنونی مقایسه کنیم تا بررسی کنیم که آیا مجاور هستند؟ اگر طول کلمات L باشد، حداکثر طول دنباله کلمات L می باشد. با توجه به اینکه هر یک از کلمات مجاور ممکن تا رسیدن به کلمه پایان بکار برده شوند، پیچیدگی این الگوریتم $O(W^L)$ می باشد که یک تابع نمایی است.

این الگوریتم دو مشکل دارد که آن را اینچنین کند می کنند: (I) آن که هر مسیر تبدیل را باید تا آخر برویم تا همه مسیرهای ممکن تبدیل را بیابیم، تا اینکه نهایتا بتوانیم کوتاهترین آنها را انتخاب کنیم، و (II) اینکه هر بار باید همه کلمات را با آخرین کلمه ای که در دنباله تبدیلات به آن رسیده مقایسه کنیم تا کلمات مجاور بعدی در دنباله را انتخاب نماییم. این کار بنظر تکراری می رسد.

در ادامه فکر می کنیم که چگونه این مشکلات را حل کنیم.

برای واضح تر شدن مشکل این الگوریتم مرتبط با دنبال کردن مسیرهای تبدیل، ساختار مجاورت کلمه ها در مثال بالا را ترسیم کنیم.



نمودار مجاورت کلمات داده شده در ورودی

شکل بالا یک ساختمان داده گراف را تداعی می کند که در آن کلمات گره ها را می سازند و خط بین آنها ارتباط 'مجاور بودن' (یعنی تفاوت تنها در یک حرف) دو کلمه را نشان می دهد.

الگوریتم جستجوی کامل ارایه شده در واقع این گراف را به صورت عمقی جستجو می کند. در زبان گراف، دلیل اول مطرح شده برای توضیح کند بودن این الگوریتم آن است که باید ابتدا تمام مسیرها را از شروع به پایان یافت و سپس کوتاهترین را تشخیص داد.

خوب، آیا ممکن است که نقطه مقابل روش عمقی جستجوی گراف این مشکل را رفع کند؟ نقطه مقابل جستجوی عمقی، جستجوی سطح به سطح است که این امکان را می دهد در همه مسیرها همزمان و هر بار یک گام پیش برویم. به این ترتیب

اولین باری که در یک مسیر به کلمه پایان می رسیم می توانیم مطمئن باشیم که کوتاهترین مسیر تبدیل را یافته ایم و نیازی به پیمایش همه مسیر ها تا پایان وجود ندارد. بنابراین استفاده از جستجوی سطح به سطح برای پیمایش گراف مشکل اول الگوریتم را حل خواهد کرد.

ایراد دوم الگوریتم مربوط به مقایسه مجدد کلمه ها با همدیگر برای یافتن مجاور های یک کلمه است. برای رفع این مشکل می توانیم ابتدا یکبار همه کلمات را با یکدیگر مقایسه نماییم و نتیجه را به صورت گراف فوق ذخیره کنیم. با استفاده از چنین پیش پردازشی، سریعا می توانیم لیست کلمات مجاور هر کلمه را بدست آوریم.

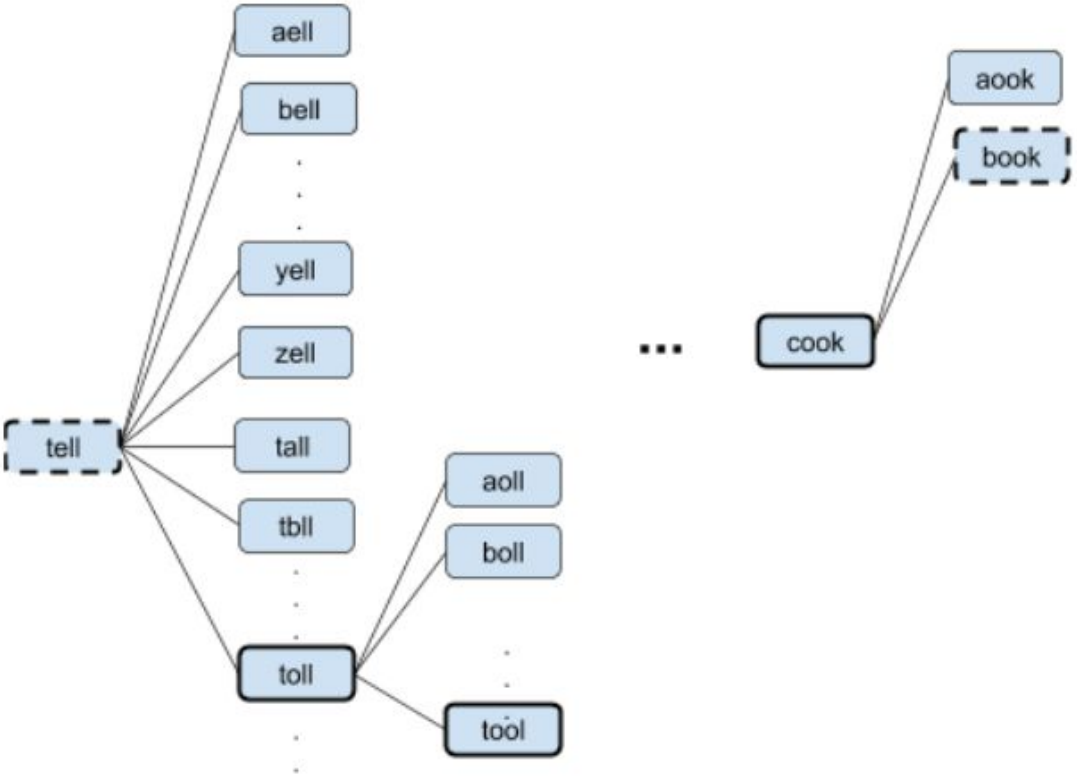
اما برای ساختن چنین گرافی هر کلمه باید با کلمه دیگر مقایسه شود. این یعنی $w \wedge 2$ مقایسه که هزینه هر مقایسه L است. بنابراین هزینه ساختن این گراف $O(L*w \wedge 2)$ می باشد.

این ساختار هزینه های آتی یافتن مجاورها را بشدت کم می کند، اما ممکن است بسیار غیر بهینه باشد. در گراف مثال فوق این ایراد عیان می شود: ممکن است که گراف یک مولفه (زیر گرافی متصل از گره ها) داشته باشد که به هیچ وجه نتوانیم از کلمه شروع به آنها برسیم. مثلا از کلمه شروع هیچ مسیری به کلمات cage, wage, rage وجود ندارد. بنابر این دلیلی برای لحاظ کردن آنها در ساختن گراف مجاورت وجود ندارد. اما قبل از پیمایش گراف نمی توانیم این را بفهمیم. این شبیه مسئله مرغ و تخم مرغ است و یک وابستگی حلقه مانند بین این دو مرحله الگوریتم وجود دارد.

برای شکستن مسئله مرغ و تخم مرغ باید از روش دیگری برای پیمایش گراف استفاده کنیم. جرقه این ایده جدید بر اساس تعریف مجاورت به ذهن می رسد. کلمه مجاور در دنباله تغییرات را می توان مستقیما با جایگزین کردن یک حرف الفبا از L موقعیت حروف در یک کلمه ایجاد نمود. مثلا کلمات bell, cell، aell، ... و غیره با جایگزین کردن حرف اول، و کلمات

tall، tbll، tcll ، ... با جایگزین کردن در محل دوم کلمه tell ایجاد می شوند. کلماتی که با این ترتیب ایجاد می شوند مجاور کلمه اول می باشند.

بنابراین بجای بررسی بین کلمات داده شده برای یافتن کلمه بعدی، آن ها را خودمان ایجاد می کنیم. با استفاده از این روش در واقع گرافی مانند گراف زیر را ایجاد می کنیم.



نمایی از گراف مفهومی تولید شده در روش ایجاد کلمات و جستجوی سطح به سطح

مشکل کوچک آن است که همه کلمات ایجاد شده الزاما در مجموعه کلمات داده شده وجود ندارند. برای حل این مشکل قبل از استفاده از یک کلمه ایجاد شده برای توسعه به سطح بعد، ابتدا باید بررسی کنیم که آیا آن در لیست کلمات داده شده می

باشد؟ خبر خوب آن است که ساختمان داده مجموعه (set) خیلی سریع می تواند پاسخ دهد که آیا یک کلمه تولید شده در مجموعه کلمات داده شده می باشد.

اگر از روش ایجاد کلمات و جستجوی سطح به سطح استفاده نماییم، برای هر کلمه L*26 کلمه مجاور تولید می کنیم. از آنجا که w کلمه داریم، هزینه کل یافتن کوتاه ترین تبدیل حداکثر w*L*26 خواهد بود که برای مقادیر بزرگ L و W نسبت به الگوریتم جستجوی کامل عمقی بسیار سریع تر است. الگوریتم جستجوی سطح به سطح که از گراف اتصالات پیش پردازش شده استفاده می کند در بدترین حالت دارای پیچیدگی محاسباتی $O(W * L + W^2)$ می باشد. قسمت W^2 این فرمول مربوط به محاسبه اولیه اتصالات گراف است.

اینکه روش پیش پردازش و سپس جستجوی سطح به سطح در مقایسه با روش ایجاد کلمات و جستجوی سطح به سطح چگونه عمل می کنند بستگی به پارامترهای واقعی W و L دارد. اگر W خیلی از L بزرگتر باشد، آنگاه الگوریتمی که از پیش پردازش استفاده می کند هزینه سنگینی خواهد داشت. از طرف دیگر، اگر که در یک سیستم لیست کلمات ثابت باشند و هر بار کلمه شروع و پایان جدیدی داده شوند، آنگاه صرف می کند که تنها یکبار گراف مجاورت ها را بسازیم و در بارهای دیگر از آنها استفاده کنیم. این هزینه سرشکن ایجاد گراف را عملا صفر خواهد بود.

می توانیم این موارد را با مصاحبه کننده در میان بگذاریم و ببینیم کدام الگوریتم برای سناریویی که او می خواهد مفید تر است. در اینجا فرض می کنیم که مصاحبه کننده از الگوریتم ایجاد کلمات و جستجوی سطح به سطح استقبال می کند. در این صورت این الگوریتم روی یک داده ورودی تست می کنیم که مطمئن شویم درست کار می کند. سپس شروع به پیاده سازی الگوریتم می کنیم.

نام مناسب برای تابع با پارامترهای مناسب در نظر میگیریم (خط 1). سپس لیست کلمات داده شده را به همراه کلمه پایان در یک ساختمان داده مجموعه (set) می ریزیم (خط 2). یک شمارنده گام یا سطحی که در مسیرها تا بحال رسیده ایم در نظر میگیریم (خط 3). سپس لایه صفر را با قرار دادن کلمه شروع در آن می سازیم (خط 4). سپس در یک حلقه تکرار که تا رسیدن به آخرین لایه پیش می رویم (خط 5). یک لیست برای قرار دادن عناصر لیست بعد در نظر میگیریم (خط 6). به ازای هر عنصر در لایه کنونی (خط 7) هر بار آن کلمه را با کلمه پایان مقایسه می کنیم. در صورتی که به کلمه پایان برسیم شماره لایه را به عنوان تعداد مراحل رسیدن به آن برمی گردانیم (خط 8 و 9). در غیر این صورت تمام کلمات ممکن مشتق از این کلمه را می سازیم و چک می کنیم که آیا در لیست کلمات داده شده هستند (خط 11 تا 14). در صورتی که کلمه جدید در لیست کلمات باشد آن را به لایه بعد اضافه می کنیم (خط 15) و از لیست کلمات حذف می کنیم (خط 16) تا از مشاهده تکراری این کلمه در مسیرهای طولانی تر جلوگیری کنیم. هنگامی که تمام کلمات مشتق از لایه کنونی را در لایه جدید گذاشتیم، شمارنده لایه را یک واحد می افزاییم (خط 17) و لایه جدید را به عنوان لایه کنونی در نظر میگیریم (خط 18). اگر تمام لایه ها پیمایش و به کلمه پایان نرسیدیم آنگاه مقدار 1- را به نشان عدم وجود پاسخ بر میگردانیم (خط 19).

```
1. def findShortestPath(beginWord, endWord, wordList):
2.   wordList = set(wordList + [endWord])
3.   layer_count = 0
4.   layer = [beginWord]
5.   while layer:
6.      new_layer = []
7.      for w in layer:
8.         if w == endWord:
9.            return layer_count
10.        else:
11.            for i in range(len(w)):
12.              for c in "abcdefghijklmnopqrstuvwxyz":
13.                 n_word = "{}{}{}".format(w[:i], c, w[i+1:])
14.                 if n_word in wordList:
15.                  new_layer.append(n_word)
16.                  wordList.remove(n_word)
17.     layer_count += 1
18.     layer = new_layer
19.  return -1
```

۳.۱۳ یافتن یک کلمه در جدول

پرسش: یک جدول از حروف و یک کلمه کلیدی داریم. الگوریتمی بنویسید که مشخص کنید آیا کلمه کلیدی در جدول وجود دارد بگونه ای که حروف آن در جدول مجاور باشند. دو حرف مجاورند اگر بالا، راست، پایین، یا سمت چپ یکدیگر باشند. نمی توانیم از یک حرف در یک نقطه از جدول دوبار استفاده کنیم.

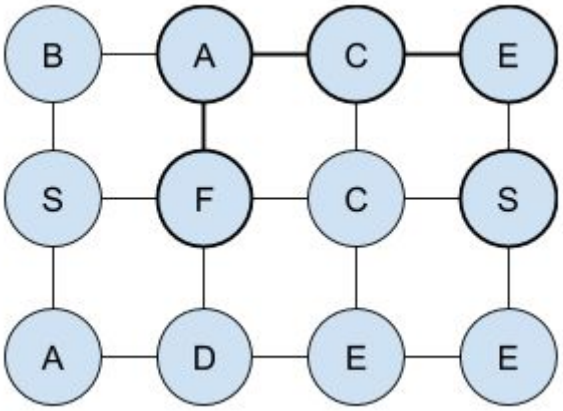
پاسخ: برای درک بهتر مسئله حل آن را با یک مثال شروع می کنیم. اگر کلمه کلیدی Faces و جدول زیر را داشته باشیم، آنگاه الگوریتم باید مقدار True را برگرداند زیرا می توان کلمه Faces را از کنار هم گذاشتن حروف مجاور جدول (همانطور که با رنگ مشخص شده است) ساخت.

B	A	C	E
S	F	C	S
A	D	E	E

مثالی از جدول کلمات و کلمه ای که با دنبال آن هستیم

ابتدا الگوریتم جستجوی کامل را مطرح می کنیم. برای این پرسش الگوریتم جستجوی کامل شامل ایجاد همه کلمات ممکن متشکل از حروف مجاور را بسازیم. اگر کلمه کلیدی میان کلمات تولید شده باشد آنگاه مقدار True را برمی گردانیم. برای ساختن تمام کلمات متشکل از حروف مجاور در جدول باید از هر نقطه جدول شروع کنیم و به 4 جهت بالا، پایین، راست، و چپ حرکت کنیم. در حرکت به هر سمت به خانه جدیدی از جدول میرسیم. باید حرکت به بالا و پایین، راست و چپ بگونه ای انجام دهیم که به خانه قبلی برنگردیم. این کار را تا زمانی انجام می دهیم که به طول کلمه ایجاد شده معادل طول کلمه کلیدی باشد. پس از ایجاد یک کلمه آن را با کلمه کلیدی مقایسه می کنیم.

این روش را می توانیم با مقایسه تدریجی حروف کلمه ایجاد شده با کلمه کلیدی تسریع کنیم. با این صورت که خانه اول را با حرف اول کلمه کلیدی مقایسه می کنیم. چنانچه برابر نبودند عمل ایجاد کلمات از آن خانه را ادامه نمی دهیم. بنظر می رسد که راه حلی جز پیمایش کل ماتریس به این روش نداریم. اما چیزی که این بین کمک می کند راه حل را بهتر درک کنیم در نظر گرفتن مشابهت ارتباط بین خانه های جدول و یک گراف است. در واقع هر خانه جدول یک گره در گراف است و به گره هایی که در سمت راست، چپ، بالا، و پایین آن هستند متصل می شوند.



بنابر این الگوریتم تعریف شده در واقع یک جستجوی عمقی در گراف از هر گره آن می باشد. برای پیاده سازی نیاز به ساختن گره ها و اتصالات آنها به روش مرسوم ساخت گراف نمی باشد. شماره سطر ستون یک خانه از جدول برای یافتن گره هایی که در گراف همسایه هستند استفاده می شوند.

بنابر این شروع به پیاده سازی برنامه می کنیم. ابتدا نام تابع و پارامترهای مناسب را تعریف می کنیم (خط 1). سپس تعداد سطر و ستون جدول را در متغیر هایی می گذاریم که بعدا بتوانیم راحت از آنها استفاده کنیم (خط 2 و 3). به دلیل مشابه طول کلمه کلیدی را در متغیری می گذاریم (خط 4). سپس یک ماتریس به عنوان سایه ماتریس اصلی با مقادیر اولیه False برای همه خانه های میسازیم (خط 5 و 6). در ادامه پیاده سازی با تغییر مقدار خانه های این جدول به True از استفاده مجدد یک حرف در ایجاد یک کلمه جلوگیری می کنیم. برای فرمول کردن حرکت به راست، چپ، بالا، و پایین از یک لیست از لیست ها استفاده می کنیم (خط 7). هر عنصر لیست دارای دو عنصر است که اولین عنصر میزان حرکت در سطر و دومی در ستون را برای رسیدن به عنصر مجاور مناسب نشان می دهد. تابع داخلی dfs در واقع از خانه (i , j)جدول شروع به یافتن ادامه کلمه کلیدی (از اندیس seq به بعد) می کند (خط 9). قبل از تشریح داخل این تابع سراغ مابقی سطرهای برنامه می کنیم. خط 25 تا 28 در دو حلقه تکرار تو در تو ماتریس را می پیماید و امتحان می کند که آیا می توان کلمه کلیدی را از یک خانه ماتریس ساخت. برای تشخیص این امر هر بار تابع داخلی را فراخوانی می کند (خط 27). حال به تشریح تابع داخلی ادامه می دهیم. این یک تابع بازگشتی است و خود را فراخوانی می کند تا جایی که با انتهای کلمه کلیدی نرسیده ایم و همچنان حروف جدول در مسیر پیموده شده با حروف کلمه کلیدی برابرند. به این دلیل هر بار حرف در موقعیت (i, j)ماتریس را با عنصر موقعیت seq در کلمه کلیدی مقایسه می کنیم. در صورتی که برابر نباشند این مسیر را ادامه نمی دهیم (خط 22 و 23). در صورتی که برابر باشند و این آخرین حرف در کلمه کلیدی باشد با این معناست که کلمه کلیدی در جدول موجود است (خط 11 و 12). اگر به انتهای کلمه کلیدی نرسیده باشیم باید این خانه (i, j) را نشان گذاری کنیم تا در ادامه مسیر کنونی از آن دوباره استفاده نکنیم (خط 13). سپس برای هر یک از جهت های حرکت در یک حلقه

تکرار (خط 14) اندیس موقعیت های بعدی را می سازیم (خط 15 و 16). آنگاه بررسی می کنیم که اندیس ها خارج از ماتریس نیفتاده باشند و همچنین خانه مورد نظر قبلا در این مسیر استفاده نشده باشد (خط 17 و 18). در صورت امکان حرکت به نقطه جدید تابع را به صورت بازگشتی فراخوانی می کنیم (خط 19). در صورتی که حاصل جستجو موفقیت آمیز باشد به جستجو خاتمه می دهیم (خط 20). در صورتی که جستجو موفقیت آمیز نبود نشان خانه (i, j) را به false تنظیم می کنیم تا مسیر بعدی بتواند برای تشکیل کلمات از آن استفاده نماید (خط 22).

```
1. def find(self, board, word):
2.     row_len  = len(board)
3.     col_len  = len(board[0])
4.     word_len = len(word)
5.     checked    = [[False for _ in range(col_len)] for _ in \
6.                                                 range(row_len)]
7.     directions = [[0,1], [0, -1], [-1, 0], [1, 0]]
8.
9.     def dfs(i, j, seq):
10.         if board[i][j] == word[seq]:
11.             if seq+1 == word_len:
12.                 return True
13.             checked[i][j] = True
14.             for d in directions:
15.                 ni = i+d[0]
16.                 nj = j+d[1]
17.                 if 0<=ni<row_len and  0<=nj<col_len\
18.                    and (not checked[ni][nj]):
19.                     if dfs(ni, nj, seq+1):
20.                         return True
21.
22.         checked[i][j] = False
23.         return False
24.
25.     for i in range(row_len):
26.         for j in range(col_len):
27.             if dfs(i, j, 0):
28.                 return True
20.     return False
```