هوالعلیم

طراحی و تحلیل الگوریتم‌ها

نیمسال دوم سال تحصیلی ۱۴۰۱ – ۱۴۰۰

تمرینات نظری ۳

مریم رضائی

۱. مجموعه $A = \{a_1, a_2, \cdots, a_n\}$ از اعداد صحیح مثبت داده شده است و ما می‌خواهیم یک زیرمجموعه ناتهی $S \subset A$ را بیابیم که تساوی $\sum_{a_i \in S} a_i = \sum_{a_i \notin S} a_i$ صادق باشد (البته چنین زیرمجموعه‌ای ممکن است وجود نداشته باشد).

الف) یک الگوریتم ساده‌اندیشانه را برای این مسأله طراحی کنید و کارایی زمانی آن را اندازه بگیرید.

ب) الگوریتمی کاراتر از الگوریتم ساده‌اندیشانه را برای این مسأله طراحی کنید و کارایی زمانی آن را اندازه بگیرید.

جواب:

الف) در الگوریتم ساده‌اندیشانه، کافی‌ست تمامی زیرمجموعه‌های ممکن را برای مجموعه شکل داده و مجموع هر کدام را حساب کنیم. قصد ما در اصل یافتن زیر مجموعه‌ای ست که مجموع آن نصف مجموع کل مجموعه باشد. پس در اصل، در ابتدا می توانیم مجموع مجموعه را یافته و نیمی از آن را حساب کنیم. از آنجا که اعضا فقط اعداد صحیح هستند، متوجه میشویم که در صورت اعشاری بودن این نیم مجموعه (یعنی فرد بودن مجموع مجموعه) چنین زیر مجموعه‌ای وجود ندارد و عدم وجود آن را اعلام می‌کنیم.

در غیر این صورت و صحیح بودن نیم مجموع (زوج بودن مجموع مجموعه) با حلقه شروع به شکل دادن تمامی زیر مجموعه‌های ممکن مجموعه کرده و هر کدام را که ساختیم، مجموعش را محاسبه کرده و با نیم مجموع مجموعه اصلی مقایسه میکنیم؛ اگر برابر بود، این زیرمجموعه مورد نظر ما بوده و در غیر این صورت به جستجوی خود ادامه میدهیم تا زیرمجموعه مورد نظر را بیابیم و یا تمامی زیرمجموعه‌ها را امتحان کنیم و چنین زیر مجموعه‌ای را پیدا نکنیم. در حالت اول زیرمجموعه یافت شده را خروجی داده و در حالت دوم باز عدم وجود را اعلام میکنیم.

کارایی این الگوریتم با توجه به اعضا محاسبه می‌شود. الگوریتم سه بخش مستقل دارد:

۱) **یافتن مجموع مجموعه‌ی اصلی:** از تک تک اعضا یک بار گذر می‌کند پس کارایی زمانی $O(n)$ است.

۲) **تقسیم مجموع و بررسی صحیح بودن:** از آنجا که عدد رشد زیاد به نسبت رشد آرایه ندارد، آن را عددی ناچیز در نظر گرفته و تقسیم و بررسی آن را در $O(1)$ محاسبه می‌کنیم.

۳) **شکل دادن تک‌تک زیرمجموعه‌ها و محاسبه مجموع هر کدام:** این مرحله دارای دو بخش درون هم است. ابتدا یک زیرمجموعه شکل داده می‌شود و سپس از اعضای آن باز گذر کرده و جمع را حساب می‌کنیم. عملیات یافتن جمع اعضا به مانند قبل زمان خطی $O(n)$ برده و به تعداد تک تک زیرمجموعه ها تکرار می‌شود، که این تعداد بنا بر قوانین ریاضی 2^n می‌باشد. پس کارایی زمانی این بخش $O(n) \times O(2^n)$ است.

پس کارایی زمانی کل الگوریتم به شکل زیر است:

$$O(n) + O(1) + O(n) \times O(2^n) = O(2^n n)$$

ب) برای یافتن الگوریتمی بهینه‌تر نیاز است از برنامه‌نویسی پویا استفاده کنیم. در این نوع برنامه‌نویسی قصد کوچک کردن مسئله ای بزرگ به حالات و مسائل کوچک تر است که بهترین جواب به هرکدام، بهترین جواب به کل مسئله را به همراه دارد. این روش برای مسائلی چون کوله پشتی به کار می رود که هر عنصر وزن خاصی داشته و میخواهیم انتخاب هایی بر اساس وزن ها انجام دهیم که کمترین و یا بیشترین وزن و یا وزنی معین داشته باشیم.

در مسئله حال حاضر، که در اصل نوعی از مسئله جمع زیرمجموعه (*subset sum*) می باشد، قصد تعیین عناصری از مجموعه است که با هم وزنی حاصل نیمی از جمع کل مجموعه داشته باشند. از آنجا که عناصر این مجموعه تنها اعداد صحیح مثبت هستند، جمع مورد نظر را می‌توانیم تنها با جمع همه‌ی عناصر به دست آوریم و اعداد منفی در هر مرحله از جمع زیر مجموعه ها باعث پیچیدگی نمیشوند. پس اگر جمع مجموعه را *sum* بنامیم، مسئله حال یافتن عناصری از مجموعه است جمعشان $^{sum}/_{2}$ می باشد. با یافتن این جمع ابتدا در صورت فرد بودن آن الگوریتم را متوقف میکنیم زیرا نمیتواند به دو تقسیم شود و جمع اعداد صحیح باش. در غیر این صورت ادامه می‌دهیم.

همانطور که ذکر شد، در مسائل برنامه‌نویسی پویا مسئله به حالات کوچکتر تقسیم می شود. بدین علت همواره آرایه ای برای ذخیره حالت های تولید شده و وضعیتشان نیاز داریم. در اینجا، نیاز است که در ماتریسی، برای هر i عنصر اول مجموعه تعیین کنیم که آیا می‌توانیم از این مجموعه زیر مجموعه ای با جمع های مختلف بسازیم یا خیر. این جمع ها نیز تمام جمع های ممکن از ۰ تا عدد مورد نظر $^{sum}/_{2}$ می باشند. برای مثال، اگر مجموعه ای ۳ عضوی داشته باشیم و مثلا زیرمجموعه ای با جمع ۴ بخواهیم، ماتریس در اصل بررسی میکند که با هیچ عنصر اول، یک عنصر اول، دو عنصر اول، و سه عنصر اول آیا میتوان مجموعه ای ساخت (که شاید تمام عناصر را هم نداشته باشد) و جمع های ۰ تا ۵ رو خروجی دهد یا خیر. این ماتریس نمونه به شکل زیر است که i ها به تعداد عناصر + ۱ و j ها به تعداد جمع ها + ۱ هستند (یعنی خودشان، با در نظر گرفتن صفر و تهی).

$$\begin{bmatrix} True & False & False & Flase & False \\ True & False & False & True & False \\ True & True & False & True & True \\ True & True & True & True & True \end{bmatrix}$$

سطر یک برای مجموعه با هیچ عضو، سطر دو برای عضو اول، سطر سه برای دو عضو، و سطر چهار برای مجمموعه با سه عضو است. ستون یک برای جمع صفر، ستون دو جمع یک، و به همین شکل تا ستون پنج که با جمع ۴ است. در اصل، مجموعه ی اصلی را از پایین به بالا ساخته و برای هر کدام، زیرمجموعه های ممکن را در نظر میگیریم. بدین شکل، با اضافه شدن هر عنصر و حرکت به حالت بعد، تمام زیر مجموعه ها را از اول محاسبه نکرده و اثر عنصر جدید را سنجیده و امکان تشکیل زیرمجموعه با آن جمع را می‌بینیم. اگر مثلا جمع ۲ میخواستیم اما در اعضا تا به حال ۵، ۷، و ۱ داشتیم نمیتوانستیم زیرمجموعه را بسازیم. اما با اضافه شدن عنصر ۲ و مقایسه آن با جمع تعیین شده متوجه میشویم که میتوانیم زیر مجموعه را بسازیم و حالت مربوطه را در ماتریس *True* میکنیم.

پس یعنی بخش تشخیص وجود چنین زیرمجموعه ای به شکل زیر است:

۱) جمع مجموعه را یافته و در صورت فرد بودن، عدم وجود داده و در صورت زوج بودن بر دو تقسیم کرده تا جمع مورد نظر $^{sum}/_{2}$ یا همان $newsum$ را بیابیم.

۲) آرایه ای با ابعاد $n + 1$ و $newsum + 1$ تشکیل میدهیم که وضعیت و امکان حالات را در آن به ازای هر i عضو و j جمع ذخیره کنیم.

۳) در عناصر و جمعها تا سقف جمع مورد نظرمان پیش رفته و برای هر کدام در ماتریس امکان یا عدم امکان را مشخص میکنیم. در نظر داریم که هر بار با اضافه شدن عنصری، به سطر بعد رفته و آن را در حلقه دوم برای جمع ها بررسی میکنیم. در هر بار، سطر جدید را از سطر قبل ساخته و در صورت موثر بودن عنصر جدید در آن حالت تغییر میدهیم.

با بررسی عنصر آخر ماتریس (سطر آخر ستون آخر، که تمام عناصر و عدد کامل جمع را در نظر گرفته) تشخیص میدهیم که آیا زیرمجموعه‌ی مورد نظر ممکن است یا خیر. اگر بله، حال بایستی با توجه به سطر آخر این ماتریس زیرمجموعه را بسازیم. سطر آخر ماتریس را که نشان میدهد اضافه شدن هر عنصر چه اثری در به دست آوردن جمع مورد نظر داشته از آخر به اول پیموده و در عناصر اضافه شده بررسی میکنیم که اضافه شدن هر عنصر آیا به ممکن شدن جمع اثر گذاشته یا خیر. هر بار با تشخیص عناصر موثر، آنها را در آرایه ای قرار داده و زمانی که پایان میرسیم آرایه را چاپ میکنیم.

کارایی زمانی این الگوریتم با مشاهده آرایه ساخته شده و سپس طی شده قابل مشاهده است. از آنجا که در بخش اصلی الگوریتم که ساختن آرایه هست هر جایگاه آرایه را یک بار طی کرده و پر میکنیم، کارایی زمانی الگوریتم برابر با ابعاد آن خواهد بود. به طور کامل تر برای کارایی زمانی الگوریتم (که چار بخش مستقل دارد داریم):

۱) **یافتن جمع و عمل تقسیم:** $O(n)$

۲) **تشکیل ماتریس:** $O(n \times newsum)$

۳) **پیمایش و پر کردن ماتریس:** $O(n \times newsum)$

۴) **پیمایش سطر آخر ماتریس:** $O(n)$

بنابراین برای کارایی داریم: $(n) + O(n \times newsum) + O(n \times newsum) + O(n) = O(n \times newsum)$

مراجع:

- https://www.interviewkickstart.com/problems/partition-equal-subset-sum
- https://www.geeksforgeeks.org/partition-problem-dp-18

۲. فرض کنیـــد که T_1 یک درخت دودویی جستجـــوی n_1 گره‌ای و با ارتـــفاع h_1 باشـــد؛ و T_2 یک درخت دودویی جستجوی n_2 گره‌ای و با ارتفاع h_2 باشد؛ و اینکه هر کلیدی که در T_1 ذخیره شده باشد، از هر کلیدی که در T_2 ذخیره شده باشد، کوچک‌تر باشد.

اگر $h = \max\{h_1, h_2\}$ باشـــد، الگوریتمی با کارایی زمانی $\mathrm{O}(h)$ طراحی کنیـــد که دو درخت T_1 و T_2 را با هم ادغام کند و یک درخت دودویی جستجوی جدید $n_1 + n_2$ گره‌ای (که حاوی کلیدهای T_1 و کلیدهای T_2 باشد) بسازد.

جواب:

در یک درخت جستجوی دودویی میدانیم که گره ها همواره بر اساس جنس تقسیم شده و همواره چپ درخت عناصر کوچک تر بوده و راست درخت بزرگترین عناصر است. از آنجا که درخت T_1 همواره عناصری کوچک تر از درخت T_2 دارد، رابطه دو درخت مانند دو نیمه یک درخت جستجوی دودویی بوده و این به ما نشان میدهد که برای جمع کردن دو درخت در یک درخت جستجوی دودویی نیاز هست هر کدام را نیمه ای از یک درخت کنیم.

برای این کار ابتدا باید بزرگترین عنصر درخت کوچکتر و کوچکترین عنصر درخت بزرگتر را بیابیم که آن ها را ریشه‌ی اتصال درخت ها قرار دهیم. برای این کار در راست درخت یک و چپ درخت دو پیش رفته تا بعد از عمق به ترتیب h_1 و h_2 ماکسیموم درخت یک که در کل عناصرش کوچک تر است و مینیموم درخت دو که در کل عناصرش بزرگ بر است را بیابیم. این قدم به اندازه ارتفاع دو درخت پیش رفته و وابسته به درختیست که بزرگتر است، یعنی *max* دو ارتفاع، پس کارایی زمانی آن $\mathrm{O}(h)$ است که $h = \max\{h_1, h_2\}$. در قدم ارتفاع دو درخت را نیز محاسبه میکنیم.

حال باید بنابر این نقطه نزدیکی درخت ها را بازسازی و ادغام کرده تا درخت اول که عناصرش کوچکتر است، فرزند چپ ریشه درخت جدید باشد، و درخت دوم که عناصرش بزرگتر است فرزند راست ریشه درخت جدید باشد. برای این کار با توجه به ارتفاع یافت شده برای دو درخت داریم:

اگر $h_1 < h_2$ باشد:

۱) راست ترین گره یافت شده برای درخت چپ را حذف می‌کنیم و چرخش لازم را برای متعادل نگه داشتن درخت جستجوی دودویی انجام می‌دهیم. این عمل باز در $\mathrm{O}(h)$ انجام میشود که $h = \max\{h_1, h_2\}$ و $\mathrm{O}(\log n)$ می باشد. از گره حذف شده درختی جدید ساختی و آن را ریشه قرار میدهیم.

۲) حال درخت دوم را به عنوان فرزند راست ریشه جدید قرار داده و درخت اول جدید را به عنوان فرزند چپ ریشه قرار میدهیم. پس از اینکار نیاز به دوباره متعادل سازی درخت داریم که در این نوع درخت ها بعد از درج زمان $\mathrm{O}(\log n)$ یعنی همان بیشترین ارتفاع میان دو درخت را می‌برد.

برای ارتفاعات برعکس نیز همین کار را انجام داده اما از درخت دوم ریشه را میگیریم.

بنابراین کارایی زمانی الگوریتم سه بخش مستقل $O(h)$ میشود که به معنی همان کارایی زمانی کلی $O(h)$ می باشد.

مراجع:

- https://stackoverflow.com/questions/9605968/algorithm-how-to-concatenate-two-binary-search-tree-efficiently

۳. الف) فرض کنید A و B ، دو مجموعه‌ای باشند هر یک متشکل از n عدد صحیح که هر یک از آنها در محدوده ۱ تا $2n$ واقع باشد. الگوریتمی را با کارایی زمانی $O(n)$ توصیف کنید که دو مجموعه A و B را به عنوان ورودی بگیرد و تعیین کند که آیا دو مجموعه A و B با هم برابر هستند یا خیر؛ یعنی آیا شامل عناصر کاملاً یکسانی (گرچه با ترتیب متفاوت) هستند یا خیر.

ب) فرض کنید A و B ، دو مجموعه‌ای باشند هر یک متشکل از n عدد صحیح که هر یک از آنها در محدوده ۱ تا n^4 واقع باشد. الگوریتمی را با کارایی زمانی $O(n)$ توصیف کنید که دو مجموعه A و B و عدد صحیح x را به عنوان ورودی بگیرد و تعیین کند که آیا عدد صحیح a در مجموعه A و عدد صحیح b در مجموعه B وجود دارند که رابطه $x = a + b$ برقرار باشد یا خیر.

جواب:

الف) برای طراحی چنین الگوریتم کارایی نیاز به استفاده از ساختار داده‌ای مناسب داریم که بتواند جستجو در مجموعه‌ها که نامرتبند را در زمان ثابت $O(1)$ انجام دهد. این یعنی نیاز به استفاده از ساختار داده‌ی جدول درهم ساز است. برای استفاده از این ساختار داده در نظر داریم که نیاز به تعیین دو بخش اصلی داریم: اندازه جدول، و تابع درهم‌ساز. از آنجا که دامنه اعداد تا $2n$ است، برای جدول مناسبی که در خانه‌ای دو کلید وارد نشود میتوانیم جدول را به اندازه $2n$ قرار دهیم و در تابع درهم‌ساز مقدار را برابر با خود کلید تعریف کنیم یعنی:

$$h(key) = key$$

بدین صورت ساختار داده‌ای داریم که درج و جستجو در آن قطعا در زمان ثابت رخ می‌دهد، زیرا خانه‌ای دارای بیش از یک کلید نخواهد بود؛ این به شرطی‌ست که یک عدد را بیش از یک بار وارد جدول نکنیم . خصوص درست است زیرا در ادامه اشاره میکنیم که در صورت برخورد با اعداد تکراری چطور رفتار کرده و اعداد را دوباره در جدول قرار نمی‌دهی.

حال با استفاده از این ساختار داده می‌توانیم الگوریتم را طراحی کنیم. به این طور که تنها در زمان $O(n)$ ابتدا یکبار مجموعه اول را طی کرده و اعضایش را در جدول درهم‌سازی قرار می‌دهیم. سپس باز در زمان خطی $O(n)$ در مجموعه دوم پیش رفته و هر عنصر را در جدول تشکیل شده جستجو می‌کنیم. اگر تا انتهای مجموعه دوم پیش رفتیم و تمام عناصر را در جدول یافتیم، از آنجا که عناصر وارد شده در جدول به تعداد یکسان با مجموعه دو بودند (هر دو مجموعه n عضو دارند) پس دو مجموعه برابرند.

نکته‌ی قابل توجه این است که در سوال ذکر نشده است که مجموعه‌ها اعضای تکراری ندارند. و این مورد در جستجو باعث مشکل می‌شود زیرا، جدا از مشکل در ساختار داده به خاطر شکل تابع درهم‌ساز، اگر در مجموعه دوم یک عدد دوبار موجود باشد و در مجموعه اول یک بار، در جستجو در جدول همواره عدد یافت می‌شود با این تفاوت که تعداد یکسان نیست. برای رفع این مشکل کافی‌ست شمارنده‌ای در ساختار داده برای هر عنصر تعریف کنیم.

بدین صـورت که در زمان ایجاد جدول مجموعه‌ی اول، برای هر عنصـر بررسـی کنیم که آیا قبل آن را دیده بودیم (یعنی آیا آن را در جدول وارد کرده بودیم) و اگر این طور بود آن را دوباره در جدول وارد نکرده و تنها به شـمارنده‌ی آن عنصر جدول، یک عدد اضافه کنیم. بدین صـورت زمانی که در مجموعه دوم پیش رفته و در جدول به دنبال عناصر می‌گردیم، هر بار در صـورت یافتن، شـمارنده آن را نیز بررسـی کرده؛ اگر شـمارنده یک یا بیشـتر بود از آن یکی کم کنیم، اما در صورت صفر بودن شمارنده آنگاه یعنی تعداد تکرار عنصر در مجموعه دوم بیشتر از اول است، و دو مجموعه برابر نیستند.

شـبه کد این الگوریتم (بدون در نظر گرفتن طراحی سـاختار داده و در اصـل شـامل شـکل دادن جدول و جسـتجو) به صـورت زیر اسـت. برای کارایی زمانی آن مشـاهده می‌کنیم که دو بار به طور مسـتقل در یک مجموعه n عضـوی پیش میرویم (زمان خطی) و برای هر بار عملی با زمان ناچیز (زمان ثابت) انجام می‌دهیم. پس برای کارایی زمانی الگوریتـــم طراحی شده داریم:

$$O(n) \times O(1) + O(n) \times O(1) = 2O(n) = O(n)$$

Algorithm: *SetSimilarity*(*A*, *B*)

// Utilizes hash table to determine if two unsorted sets are equal
// Input: Two unsorted sets *A* and *B*
// Output: *True* if sets are similar, *False* if not

hashset ← *Hash*() // creates new object from predefined data structure
for *element* **in** *A* **do**
  if *hashset.hasKey*(*element*) **then** // add to count of existing key
    hashset.Key(*element*).*count* ← *hashset.Key*(*element*).*count* + 1
  else then // add new key in the table and make count 1
    hashset.addKey(*element*)
    hashset.Key(*element*).*count* ← 1
for *element* **in** *B* **do**
  if *hashset.hasKey*(*element*) **and** *hashset.Key*(*element*).*count* > 0 **then**
    hashset.Key(*element*).*count* ← *hashset.Key*(*element*).*count* – 1
  if *hashset.hasKey*(*element*) **and** *hashset.Key*(*element*).*count* = 0 **then**
    return *False*
  else then // element of *B* is not in first set *A*
    return *False*
// if search ends successfully then all exist and sets are similar
return *True*

ب) این بار باید بررســی کنیم آیا یک عدد داده شــده x می‌تواند از جمع یک عنصــر از هر مجموعه داده شــده A و B تشکیل شود یا خیر. برای اینکار باز به شکل قبل به ساختار داده مناسب نیاز داریم که باز جدول درهم‌ساز است، زیرا نیاز ما به جستجو و قرار دادن در زمان ثابت است. با استفاده از جدول درهم‌ساز میتوانیم باز یک مجموعه را به صورت هش در آوریم (پیشگیری از تکرار دیگر ضروری نیست و شمارنده تعبیه نمی‌کنیم)، سپس در مجموعه دوم پیش رویم و برای هر عضو، از مقدار عدد x کم کنیم تا ببینیم آیا عدد باقیمانده عضوی از جدول درهم سازی A هست یا خیر.

لازم به ذکر است که در طراحی ساختار داده در اینجا نمیتوانیم هر کلید جنس عدد را در جایگاه خود در جدول بگذاریم زیرا اعداد میتوانند تا n^4 بروند. در اینجا تابعی بهتر تابع درهم ساز معروف پیمانه‌ایست که به صورت زیر است:

$$h(key) = key \bmod table\ size$$

دقت می‌کنیم که اندازه جدول در عملکرد این تابع و مقدار یکســان یافت شــدن برای دو کلید متفاوت اثر دارد. اگر از درهم سازی بسته استفاده کنیم، درصورت نتیجه یکسان برای دو کلید متفاوت، کلیدها در لیستی پیوندی در همان خانه در کنار هم ذخیره میشوند. باید با تعیین اندازه مناسب جدول از ایجاد لیست‌های پیوندی زیاد و بزرگ جلوگیری کنیم و در عین حال جدول را زیاد بزرگ نگیریم. برای اینکار میدانیم که دو حالت مناسب برای جلوگیری از نیاز جدول به بزرگ شــدن یا تکرار خانه ها، دو برابر گرفتن اندازه جدول نسبت به ورودی یا گرفتن عدد اول بعدی آن اســت زیرا اندازه اول برای انجام عمل پیمانه‌ای پاسخ‌های مناسب‌تری دارد. از آنجا n عدد در بازه‌ی ۱ تا n^4 داریم و احتمال بد عمل کردن پیمانه سازی زیاد است، پس می‌توانیم نزدیک ترین عدد اول دو برابر مقدار n را بگیریم که برای محاسبه بهتر به توانی از دو نیز نزدیک نباشد.

حال با توجه به این ساختار داده می‌توانیم با الگوریتم، زیر مجموعه اول را درهم سازی کرده و سپس با پیش رفتن در مجموعه دوم و جستجو در جدول درهم سازی مجموعه اول، سوال را حل کنیم. شبه کد الگوریتم به شکل زیر است.

Algorithm: *NumSum*(*A*, *B*, *x*)

// Utilizes hash table to check if an int can be made of elements of two unsorted sets
// Input: Two unsorted sets *A* and *B* with integer *x*
// Output: *True* if *x* can be made from *A* and *B*, *False* if not

hashset ← *Hash*() // creates new object from predefined data structure

for *a* **in** *A* **do**

hashset.addKey(*element*)

for *b* **in** *B* **do**

a ← *x* – *b*

if *hashset.hasKey*(*a*) **then return** *True*

return *False* // if search ends and no two *a* and *b* are found then *x* cannot be made

برای کارایی زمانی الگوریتم نیز به مانند قبل مشـــاهده می‌کنیم که دو حلقه مســـتقل با تکرار n داریم. با در نظر گرفتن زمان $O(1)$ برای عمل درج و اعمال تفریق و جستجو به طور میانگین، برای کارایی زمانی مینویسیم:

$$O(n) \times O(1) + O(n) \times O(1) = 2O(n) = O(n)$$

لازم به ذکر اســـت که اگر اعداد محاســـبه شـــده در تفریق به طور قطعی با n رابطه مســـتقیم داشـــتند و همواره بزرگ می‌شـــدند، عمل تفریق را نمی‌توانســـتیم به طور میانگین ناچیز در نظر گیریم و هر عمل تفریق به تنهایی کارایی زمانی خطی $O(n)$ می‌داشت.

مراجع:

- https://stackoverflow.com/questions/245509/algorithm-to-tell-if-two-arrays-have-identical-members
- https://www.geeksforgeeks.org/given-two-unsorted-arrays-find-pairs-whose-sum-x

۴. الف) ثابت کنید که در هر گراف همبندی، رأسی وجود دارد که برداشتن آن (و همه یال‌های واقع بر آن) گراف باقیمانده را ناهمبند نخواهد کرد.

ب) الگوریتمی کارا را برای یافتن چنین رأسی ارائه کنید و کارایی زمانی الگوریتم خود را اندازه بگیرید.

جواب:

الف) بنا بر تعریف گراف هم‌بندی می‌دانیم که برای یک گراف هم‌بندی میتوان درخت پوشایی رسم کرد که تمامی رئوس گراف را با کمترین تعداد یال به هم متصل می‌کند. حال می‌دانیم که یک درخت همواره یک برگ دارد، زیرا سه حالت زیر وجود دارد:

۱) تنها یک گره دارد: خود برگ است؛
۲) دو گره متصل دارد: هر دو برگند؛
۳) سه یا بیشتر گره دارد: بنا بر فرض خلف داریم که اگر فرض کنیم تمامی گره‌ها درجه بالای یک دارند (یعنی بیش از یک یال به آنها متصل است)، آنگاه باید دوری در آن موجود باشد که این ضد تعریف درخت است و باید گره‌ای با یک یال موجود باشد که برای رسیدن از دو گره‌ی دیگر به هم، نیازی از عبور از این گره نباشد.

حال بنابر این اثبات میدانیم که برای درخت پوشایی یک گراف همبند نیز این امر صدق میکند؛ یعنی همواره حداقل یک گره وجود دارد که یک یال دارد و حذف آن اتصال درخت را از بین نمیبرد. بنابراین در گراف همبند متقابل درخت نیز میتوانیم این رأس را با یال‌هایش حذف کنیم و باز هم‌بندی گراف آسیب نبیند و از رأسی همواره راهی دیگر به یک رأس دیگر باشد، زیرا این امر در درخت صحیح بود و درخت، درخت پوشایی گراف مورد نظر بود که حداقل تعداد یال‌های ممکن را داشت. پس اثبات کامل است.

ب) حال می‌خواهیم برای تعیین رأس اثبات شده الگوریتمی بهینه بیابیم. به زبانی ساده‌تر، می‌خواهیم یک نقطه که نقطه برشی گراف نیست را پیدا کنیم. برای این کار، از الگوریتم‌های قبلا مطالعه شده استفاده می‌کنیم و با استفاده از جستجوی عمقی گراف، نقاط برشی الگوریتم و متقابلاً رئوس غیر برشی (یعنی تمامی رئوس دیگر که می‌توانند بدون شکستن گراف حذف شوند) را میابیم.

برای تعیین نقاط برشی میدانیم که رأس زمانی برشی‌ست که یا ریشه درخت پیمایش عمقی بوده و حداقل دو فرزند داشته باشد (یعنی بدین صورت دو بخش گراف را به هم به تنهایی متصل کرده است) و یا ریشه نبوده اما در زیر درخت فرزند آن هیچ یالی به اجداد آن نباشد. برای حالت دوم و تشخیص اینکه آیا یالی بازگشتی وجود دارد باید در جستجوی عمقی، علاوه بر گذاشتن علامت بازدید شده بر گره‌ها، زمان بازدید هر گره را ذخیره کنیم که بتوانیم نقاط مسیر را با هم مقایسه کنیم. همچنین در صورت یافتن یالی عقب گرد از رأس جدید به رأسی بازدید شده، باید در نظر گیریم که کمترین زمان ممکن رسیدن به این رأس تغییر کرده و میتواند کمتر و از راه رأس بازدید شده باشد؛ پس نیاز است

کمترین زمان ممکن را هم (که مینیموم زمان حال برخورد با رأس و زمان اصلی برخورد با رأس بازدید شده است) را نیز ذخیره کنیم. بهترین راه ذخیره این یافته ها که در ادامه قابل دسترسی با زمان کم باشند، تعبیه ویژگی‌ای برای رئوس گراف است؛ برای مثال، از پیش همه رئوس را غیر برشی و بازدید نشده در نظر گرفته و در صورت بازدید، آن را بازدید شده تعیین کرده و در صورت یافتن هر رأس به عنوان برشی، آن را برشی علامت میزنیم.

در آخر در الگوریتم برای راحت‌تر بودن حذف بر اساس رأس یافت شده، در رئوس گراف باز پیش رفته و از میان رئوسی که ویژگی آن‌ها غیر برشی‌ست، رأسی که کمترین تعداد یال متصل دارد را یافته و خروجی میدهیم. همچنین لازم به ذکر است که قبل از یافتن نقاط برشی برای بهینگی الگوریتم باید در نظر گیریم که اگر در ابتدا رأسی با یک یال موجود بود، آن رأس برگ بوده و مناسب حذف بدون پیمایش گراف و یافتن نقاط برشی‌ست. زیرا رأسی یک یاله نمی‌تواند دو بخش از گراف را به یک دیگر وصل کند (کمتر از دو یال دارد) و حذف آن و یالش هم‌بندی گراف را از بین نمی‌برد.

بنابراین، الگوریتم سه مرحله‌ای ذکر شده با شبه کد به شکل زیر در صفحه بعد نمایش داده می‌شود. در شبه کد دو بخش موجود است: بدنه‌ی کلی که مرحله یک (بررسی وجود برگ در گراف)، مرحله دو (فراخوانی پیمایش عمقی برای تعیین نقاط برشی و غیر برشی)، و مرحله سه (انتخاب نقطه غیر برشی با کمترین تعداد یال) را شامل است، و تابع بازگشتی تعیین نقاط برشی با استفاده از جستجوی عمقی که در بدنه فراخوانی می‌شود.

Algorithm: ***FindDisposableVertex*(*G*)**

// Utilizes function *CheckArticulation* to find a vertex that can be deleted
// Input: Undirected connected graph *G*
// Output: Best vertex that can be deleted without disconnecting graph

for *v* **in** *G.Vertices*() **do** // check existence of leaf using number of its edges
  if *v.degree* < 2 **then return** *v*
// if no leaf existed, find best disposable vertex using DFS
global *time* ← 0 // global variable to store DFS visit time
for *v* **in** *G.Vertices*() **do** // check articulation to find cut vertices and trivial ones
  if *v.visited* = *False* **then** *CheckArticulation*(*G*, *v*)
// now that all vertices are marked as cut vertex or not, find best trivial one
best ← *None* **and** *bestdegree* ← 0
for *v* **in** *G.Vertices*() **do** // see which trivial vertex has least degree
  if *v.cutvertex* = *False* **then**
    if *bestdegree* = 0 **then** *best* ← *v* **and** *bestdegree* ← *v.degree*
    elif *v.degree* < *bestdegree* **then** *best* ← *v* **and** *bestdegree* ← *v.degree*
return *best*

```
Algorithm: CheckArticulation(G, v)

    // Utilizes DFS to determine which vertices of graph are cut vertices and which not
    // Input: Undirected connected graph G and vertex v to start DFS from
    // Output: Returns nothing but modifies graph attributes to note cut vertices

    v.visited ← True   // mark vertex as visited
    v.visittime ← time  and  v.mintime ← time  and  time ← time + 1
    for u in G.Edges(v, u) do   // perform DFS for vertices adjacent to starting v
        if u.visited = False then
            u.parent ← v   // store parent in DFS visit, it was automatically None
            v.childcount ← v.childcount + 1   // it was automatically 0
            CheckArticulation(G, u)   // perform recursive DFS for this subtree
            // now time of visit for u is recorded in recursion so update v.mintime
               based on u's checked subtree that could have a back-edge to pre v
            v.mintime ← min(v.mintime, u.mintime)
            // go over conditions of being a cut vertex
            // if v is a root and has two or more children found so far
            if v.parent = None and v.childcount ≥ 2 then v.cutvertex ← True
            // if v is not root and time of u is not less than v (no back-edge)
            elif v.parent ≠ None and u.mintime ≥ v.visittime then v.cutvertex ← True
        // if new vertex was visited and is not parent then there is a back-edge
        elif u.visited = True and u ≠ v.parent then   // update mintime due to back-edge
            v.mintime ← min(v.mintime, u.visittime)

    // recursive function is over and all nodes in the entered subtree that are cut vertices
    are marked as True and all that are not are False
```

برای کارایی زمانی الگوریتم به بررسی هر مرحله بدنه اصلی که مستقل است می‌پردازیم و سپس برای کارایی کل براساس قوانین محاسبه کارایی آن‌ها را با هم جمع کنیم و در اصل کارایی مرحله‌ای که بیشترین کارایی زمانی را دارد کارایی کل میگیریم.

در مرحله اول به تعداد رئوس پیش رفته تا درجه هر رأس را بررسی کرده و رأسی با درجه یک در صورت وجود بیابیم؛ این مرحله در بدترین حالت در زمان $O(|V|)$ انجام می‌شود که تمامی رئوس بررسی شوند. در بخش دوم با پیمایش

عمقی گراف طی شده و نقاط برشی تعیین می‌شوند؛ این بخش کارایی زمانی‌ای مانند الگوریتم پیمایش عمقی دارد که برای گراف نمایش داده شده با لیست مجاورتی $O(|V| + |E|)$ می‌باشد و به علت وجود ویژگی بازدید شده یا نشده و در نظر گرفتن آن در شروط، هر گره و یال را تنها یک بار بازدید می‌کند. در مرحله آخر در لیست رئوس گراف، باز کامل پیشرفته و تک تک رئوس را بررسی می‌کنیم که آیا غیر برشی‌اند، و در این صورت رأس با کمترین درجه را انتخاب می‌کنیم؛ این بخش کارایی زمانی‌ای همواره $O(|V|)$ دارد.

از آنجا که در هر بخش اعمال با زمان ثابت انجام شده‌اند که ناچیزند، آن اعمال در نظر گرفته نشده و برای کارایی زمانی سه مرحله کلی الگوریتم داریم: $O(|V|) + O(|V| + |E|) + O(|V|) = O(3|V| + |E|) = O(|V| + |E|)$

مراجع:

- **https://math.stackexchange.com/questions/1553759/prove-connected-graph-minus-one-vertex-still-connected**
- **https://www.geeksforgeeks.org/articulation-points-or-cut-vertices-in-a-graph**

۵. فرض کنید شما مشاور شرکتی هستید که تجهیزات رایانه‌ای می‌سازد و محصولات خود را به نمایندگی‌های خود در سرتاسر کشور می‌فرستد. شرکت، برای هر یک از n هفته آینده، تولید s_i کیلوگرم تجهیزات را طرح‌ریزی کرده است. این میزان از تجهیزات باید توسط یک شرکت باربری هوایی به نمایندگی‌ها انتقال داده شود.

تولیدات هر هفته، می‌تواند توسط یکی از دو شرکت باربری هوایی A یا B حمل شود:

- شرکت A برای انتقال هر کیلوگرم از تجهیزات، مبلغ ثابت r را مطالبه می‌کند؛ یعنی انتقال s_i کیلوگرم از تجهیزات در یک هفته توسط شرکت A ، $r \cdot s_i$ هزینه را برای شرکت رایانه‌ای در بر خواهد داشت.
- شرکت B مستقل از آنکه وزن تجهیزات چقدر باشد، با دریافت مبلغ ثابت c در هر هفته قرارداد می‌بندد، اما هر قرارداد آن، برای چهار هفته متوالی است.

شرکت رایانه‌ای، به دنبال یک زمانبندی برای انتقال تجهیزات خود است؛ یعنی می‌خواهد با در نظر گرفتن این محدودیت که در صورت انتخاب شرکت B ، باید برای چهار هفته پیوسته با B قرار ببندد، شرکت باربری هوایی را برای هر یک از n هفته آینده انتخاب کند. هزینه زمانبندی، مبلغ کلی است که شرکت رایانه‌ای، طبق نکات مذکور، به شرکت‌های A و B پرداخت خواهد کرد.

مثلاً اگر $r = ۱$ باشد، $c = ۱۰$ باشد و دنباله مقادیر تولید تجهیزات (در ۱۰ هفته آینده) عبارت باشد از

$$۱۱, ۹, ۹, ۱۲, ۱۲, ۱۲, ۱۲, ۹, ۹, ۱۱$$

آنگاه زمانبندی بهینه (یعنی زمانبندی با کمترین هزینه) این خواهد بود که برای سه هفته نخست، شرکت A را انتخاب کنیم، سپس برای چهار هفته متوالی بعد از آن، شرکت B را انتخاب کنیم و سپس برای سه هفته پایانی، شرکت A را انتخاب کنیم.

الگوریتمی کارا ارائه کنید که دنباله مقادیر تولید $s_1, s_2, \cdots, s_n$ در n هفته آینده را بگیرد و زمانبندی بهینه را برگرداند. کارایی زمانی و کارایی فضایی الگوریتم خود را نیز تعیین کنید.

جواب:

در این سوال در اصل میخواهیم الگوریتمی ارائه کنیم که با آن بتوانیم دنباله‌ای از تصمیمات را به بهترین شکل ممکن بگیریم، یعنی اعمال انتخاب شده که به میزان کلی‌ای اضافه میکنند، در نهایت کمترین وزن ممکن را داشته باشند. این امر در حیطه برنامه‌نویسی پویاست. در برنامه‌نویسی پویا به مانند سوال اول، قصد تقسیم یک مسئله بزرگ به حالات یا بخش‌هاییست که قدم به قدم بر هم اضافه شده و پاسخ نهایی را به طوری بدهند که بهترین تصمیم برای هر حالت باعث شود بهترین تصمیم برای مسئله‌ی کل گرفته شود. در اینجا، برنامه‌نویسی پویا قصد دارد با تقسیم برنامه‌ریزی شرکت به چند بخش و تعیین فرمول کلی برای هر حالت که بهترین پاسخ را بدهد، مسئله را حل کند.

اولین قدم تعیین حالت‌های ممکن و بهترین پاسخ برای هر حالت است.

میدانیم که شرط اصلی غالب بر برنامه‌ریزی، چهار هفته‌ای بودن قرار داد شرکت B است. این شرط حالات پایه کلی را به ۳ بخش تقسیم می‌کند: (۱) اگر دنباله تعداد هفته‌ها صفر عضو داشته باشد و قراردادی انتخاب نشود؛ (۲) اگر دنباله هفته‌ها دارای ۱ تا ۳ عضو باشد که در آن نمی‌توان با شرکت B قرارداد بست و تنها شرکت A برای انتخاب موجود است؛ (۳) اگر دنباله هفته‌ها ۴ یا بیشتر عضو داشته باشد که در این صورت قرارداد با هر دو کشور A و B در انتخابات ما وجود دارند و باید بنا بر قیمت شرکت‌ها و وزن تولیدات، بهترین انتخاب را انجام دهیم.

برای هر حالت، فرمول مناسب که بهترین خروجی برای آن حالت را بنا بر قیمت شرکت‌ها و وزن تولیدی بدهد تعریف می‌کنیم. برای این کار تابع $mincost()$ را تعریف می‌کنیم که در هر یک از این سه بازه ضابطه‌ای بهینه داشته و کمترین هزینه برای برنامه شرکت را تعیین می‌کند. بدین صورت، تعریف تابع برای هر حالت به شکل زیر است.

۱) **زمانی که دنباله صفر عضو دا شته با شد (یعنی $n = 0$)** ← در این صورت قرار داد هیچ شرکتی در نظر نیست و در نظر گرفتن وزن یا قیمتی نیاز نیست و داریم:

$$mincost(n) = mincost(0) = 0, \qquad for\ n = 0$$

۲) **زمانی که دنباله ۱ تا ۳ عضو داشته باشد (یعنی $0 < n < 4$)** ← این حالت تنها ممکن است با شرکت A که محدودیت هفته‌ای قرار نداده کار کنیم. می‌دانیم که قیمت این شرکت به طور ضرب قیمت ثابت در وزن تولید شده یعنی $r \times s_n$ محاسبه می‌شود و همچنین این فرمول برای هر هفته مجزا بوده و برای چند هفته، هزینه کل را به طور بازگشتی بر اساس جمع هزینه کل تا هفته قبل با هزینه هفته جدید داریم. پس:

$$mincost(n) = mincost(n-1) + (r \times s_n), \qquad for\ 0 < n < 4$$

۳) **زمانی که دنباله بیش از یا مساوی ۴ عضو دا شته با شد (یعنی $n \geq 4$)** ← در حالت آخر آزادی داریم از قرار داد هر دو شرکت استفاده کنیم. پس در ضابطه باید نوع محاسبه هر دو شکل را قرار داده و کمترین هزینه را انتخاب کنیم. در نظر داریم که برای محاسبه شرکت B که قرار داد آن به طور nc برای n هفته (مضرب چهار) با هر وزنیست، همواره باید دسته دسته هر چهار هفته را حساب کنیم. یعنی انتخاب برای هفته nاُم یا شرکت A است، و یا برای آن هفته و سه هفته پیشین شرکت B است. پس برای حالت ۳ فرمول زیر را داریم:

$$mincost(n) = \min\{mincost(n-1) + (r \times s_n), mincost(n-4) + 4c\}, \qquad for\ 0 < n < 4$$

حال بنابر حالت‌های ارائه شده با برنامه‌نویسی پویا، الگوریتمی طراحی می‌کنیم که این مقادیر و همچنین تصمیمات را یافته و ذخیره می‌کند تا بتوان پس از گذشتن از وزن‌های هر بازه و انتخاب، به کمترین هزینه در آخر برسیم. برای این کار، نیاز به دو ساختار داده برای دو عمل ذخیره مقادیر حداقل تا آن هفته، و ذخیره تصمیمات تا آن هفته داریم.

آرایه اول آرایه‌ای یک بعدیست که عضو nاُم آن هزینه حداقل تا هفته nاُم بنابر تصمیمات تا به حال است؛ این آرایه طول

به اندازه دنباله (یعنی تعداد کل هفته‌ها) دارد. اما آرایه دوم آرایه‌ای دو بعدی‌ست که در آن هر انتخاب ذخیره می‌شود. مثلا برای هفته‌ی ۱، ۲، و ۳ پیش رفته و تنها حالت ۲ را داریم، اما در محاسبه برای هفته چهار حالت ۳ ممکن شده و در میابیم که شرکت B ارزان تر میابد، پس تصمیم میگیریم با اضافه شدن این هفته به محاسبه، در محاسبه هفته ۴ تصمیمات هفته‌های پیش را عوض کنیم. اما سپس در هفته ۵ با وزن جدید در میابیم بهتر بود هفته ۱ را همان تصمیم برای چهار هفته پیش نگه داشته و هفته‌های ۲ و ۳ و ۴ و ۵ را با شرکت B پیش رویم؛ در اینجا نیاز به بازگشت به تصمیمات ذخیره شده داریم تا تغییر را انجام دهیم. نمایش این آرایه دو بعدی برای مثال ذکر شده بدین شکل است:

$$\begin{bmatrix} A & & & & \\ A & A & & & \\ A & A & A & & \\ B & B & B & B & \\ A & B & B & B & B \end{bmatrix}$$

یعنی به آرایه‌ای دو بعدی با ابعاد x و y نیاز داریم که x ها شماره هر هفته که در حال محاسبه آن هستیم بوده و y ها تصمیم هر هفته پیشین را تا حال، بنابر هفته جدید ذخیره کنند. اندازه این آرایه مربعی از طول دنباله می‌شود یعنی n^2.

با توجه به این ساختار داده ها برای ذخیره تصمیمات و هزینه‌ها، و همچنین فرمول‌های حالات، در نهایت الگوریتم زیر را داریم. در این الگوریتم، حالات را پیش رفته و هفته به هفته محاسبه می‌کنیم. هر بار با محاسبه و ذخیره هزینه در آرایه تک بعدی، تصمیم جدید را در خانه آخر آرایه دو بعدی ذخیره کرده و تصمیمات قبل را با حلقه‌ای با اطلاعات هفته جدید هماهنگ میکنیم (همانطور که در مثال ذکر شد). بدین صورت پاسخ نهایی (یعنی برنامه) سطر آخر آرایه دو بعدی خواهد بود که شرکت انتخاب شده برای هر هفته را ذکر کرده است. شبه کد الگوریتم در صفحه بعد قابل مشاهده است.

برای کارایی زمانی الگوریتم می‌دانیم که شناسایی وقت‌گیرترین بخش مستقل برای محاسبه کافی ست. در این الگوریتم، این به معنی شناسایی عمقی ترین حلقه‌هاست. مشاهده می‌کنیم که در حلقه‌ی حالت ۲، حلقه‌ی بیرونی $n-3$ بار و بزرگترین حلقه درونی $i-1$ بار عملیات های ثابت انجام میدهند. پس برای کارایی زمانی داریم:

$$T(n) = \sum_{i=4}^{n}\sum_{j=1}^{i-1} 1 = \sum_{i=4}^{n}\big((i-1)-1+1\big) = \sum_{i=4}^{n}(i-1) = \sum_{i=4}^{n} i - \sum_{i=4}^{n} 1$$
$$= \frac{n(n+4)}{2} - (n-4+1) = \frac{n^2+4n-2n-6}{2} = \frac{n^2+2n-6}{2} = O(n^2)$$

همچنین برای کارایی فضایی الگوریتم میدانیم از دو آرایه با ابعاد توضیح داده شده استفاده شده. بنابراین برای کارایی فضایی کل الگوریتم از جمع این دو فضا داریم:

$$O(n+1) + O((n+1)^2) = O(n) + O(n^2) = O(n^2)$$

Algorithm: ***FindSchedule***$(r, c, s_1, s_2, \ldots, s_n)$

```
// Finds best schedule for company shipping
// Input: Cost r of company A, cost c of company B, array of weekly product weights
// Output: An array of company choices for each week

mincost ← array(n + 1)   // create array of size n + 1
mincost[0] ← 0   // make zeroth index state 1
choices ← matrix(n + 1, n + 1)   // make square matix
// initialize state 2
for 0 < i ≤ min{3, n} do   // if weeks are less than 3, do their count, else do 3
    mincost[i] ← mincost[i – 1] + r × s_i
    choices[i][i] ← 'A'   // put new company choice in current week place
    for 0 < j ≤ i – 1 do   // update choices of weeks before current week
        choices[i][j] ← choices[i – 1][j]
// initialize state 3
for 4 ≤ i ≤ n do
    mincost[i] ← min{mincost[i – 1] + r × s_i, mincost[i – 4] + 4c}
    if mincost[i – 1] + r × s_i < mincost[i – 4] + 4c then // if A is better
        choices[i][i] ← 'A'   // put new company choice in current week place
        for 0 < j ≤ i – 1 do   // update choices of weeks before current week
            choices[i][j] ← choices[i – 1][j]
    else // if B is better or both are the same
        choices[i][i] and choices[i][i – 1] and choices[i][i – 2] and choices[i][i – 3]  ← 'B'
        for 0 < j ≤ i – 4 do   // update choices of weeks before current week
            choices[i][j] ← choices[i – 4][j]
return choices[n + 1]   // return last row that is the final schedule
```

مراجع:

- https://pdfcoffee.com/csci-4020-computer-algorithms-pdf-free.html