هوالعلیم

طراحی و تحلیل الگوریتم‌ها

نیمسال دوم سال تحصیلی ۱۴۰۱ – ۱۴۰۰

تمرینات نظری ۲

مریم رضائی

۱. تصـــور کنید که شـــما در بخش فناوری اطلاعات بیمارســـتانی بزرگ کار می‌کنید. افراد شـــاغل در بخش پذیرش بیمارســتان، همیشــه به شــما اعتراض می‌کنند که اجرای نرم‌افزاری که برای تســویه حســاب و ترخیص بیماران اســتفاده می‌کنند، خیلی کند است؛ آنقدر که در نزدیک به ظهر اکثر روزها، به دلیل کندی نرم‌افزار، صـف‌های طولانی از افرادی که منتظرند تا تسـویه حسـاب کنند و بیمارسـتان را ترک کنند، تشـکیل می‌شـود. وقتی که شـما برای حل این مسـأله از کارمندان می‌پرســید که آیا چنین صــف‌های طولانی برای پذیرش بیماران در بیمارســتان نیز تشــکیل می‌شــود، متوجه می‌شـوید که در قیاس با فرایند تسـویه، فرایند پذیرش بسـیار سـریع‌تر اسـت. بعد از بررسـی کد نرم‌افزاری که برای فرایند پذیرش و ترخیص افراد نوشته شده است، متوجه می‌شوید که اطلاعات بیمارانی که در حال حاضر در بیمارستان پذیرش شـده‌اند، در یک لیسـت پیوندی ذخیره شـده‌اند و اطلاعات بیماران جدیدی نیز که قرار اسـت پذیرش شـوند، به سـادگی به انتهای آن لیست پیوندی اضافه می‌شوند. توضیح دهید که چگونه این نرم‌افزار را تغییر می‌دهید تا بتوان از طریق آن، هم پذیرش بیماران و هم ترخیص آنها را با سرعت انجام داد. زمان اجرای راه‌حل الگوریتمی خود را هم برای ثبت پذیرش‌ها و هم برای ثبت ترخیص‌ها مشخص کنید.

جواب:

ابتدا به شـناسـایی مشـکل موجود می‌پردازیم. مشـکل در تفاوت کارایی زمانی اضافه کردن عنصـر به لیسـت پیوندی و پیدا کردن و حذف عنصـر از آن اسـت، زیرا عملیات اول (که در پذیرش بیمارسـتان اتفاق می‌افتد) زمان ثابت و عملیات دوم (که در ترخیص اتفاق می‌افتد) زمان خطی دارد. این به علت ساختار لیست پیوندی‌ست، و با این ساختار داده‌ی استفاده شده نمی‌توان آن را بهینه کرد. پس برای حل مشکل نرم‌افزار بیمارستان، نیاز به تغییر ساختار داده داریم.

بهترین ســاختار داده برای ذخیره‌ی اطلاعات که به ســرعت قابل جســتجو و تغییر باشــند، جدول درهم‌ســازی یا هش (*hash table*) می‌باشـد. در این سـاختار داده، اطلاعات به صـورت مقادیر (*value*) در جایگاه‌هایی مناسـب در جدولی از پیش تعیین شده ذخیره می‌شوند که به کمک کلیدهایی (*key*) قابل دسترسی هستند. به این علت، کارایی زمانی اعمال جسـتجو، ثبت، و حذف در این سـاختار داده در بهترین حالت و حالت آنالیز اسـتهلاکی (و در کل در حالت میانگین برای داده‌های زیاد) زمان ثابت $O(1)$ است.

لازم به ذکر اســـت که کارایی زمانی بدترین حالت این ســـاختار داده $O(n)$ اســـت و زمانی اتفاق می‌افتد که تابع هش انتخاب شـده کلیدها را به جایگاه‌های مناسـب هش نکند و مقادیر در جایگاه‌های یکسـان قابل دسـترسـی باشـند، یا اندازه جدول در نظر گرفته مناسب نباشد و نیاز شود که جدول از نو ساخته شود. این دو حالت، که در اصل نشانگر ایراد در انتخاب دو بخش اصلی این ساختار داده که برای کارا بودن آن نیاز هستند است، نتیجه در تصادف (*collision*) می‌دهند. برای پیشگیری از این اتفاق، باید تابع هش و اندازه جدول مناسبی در نظر گرفته شود.

برای تابع هش مناســب، توابع متفاوتی ارائه شــده‌اند، برای مثال تابع هش پاول شــی (*Paul Hsieh*) یا تابع باب جنکین (*Bob Jenkin*) که هردو توابع پیچیده و طولانی‌ای هســتند، و یا تابع هش ضـربی نوث (*Knuth*) که سـاده‌تر اسـت. برای

هدف این مسئله، تابع هش پیمانه‌ای که پر استفاده‌ترین تابع است کفایت می‌کند. فرم کلی این تابع به شکل زیر است:

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

برای به کارگیری این تابع در ساختار داده‌ی بیمارستان، باید کلید و اندازه جدول مناسبی انتخاب کنیم. اگر کلید را کد ملی افراد در نظر بگیریم، مطمئنیم که کلیدی خاص بوده و از تکرار پیشگیری می‌کند. همچنین برای اندازه جدول ثابت شده است که بهترین پاسخ زمانی یافت می‌شود که اندازه جدول عددی اول باشد که نزدیک به توانی از دو نیز نباشد. برای تعیین دقیق این جدول برای بیمارستان، نیاز به بررسی داده‌های آماری بیمارستان داریم تا با در نظر گرفتن میزان مراجعه میانگین روزانه و همچنین موقعیت جغرافیای بیمارستان و امکان نیاز اضطراری به پشتیبانی حادثه، اندازه‌ای مناسب بیابیم که در شرایط سخت کمبود پیدا نکرده و در عین حال حافظه اضافی مصرف نکند.

نکته‌ی قابل به ذکر این است که با وجود در نظر گرفتن این موارد و بهینه‌سازی ساختار داده، همواره امکان وجود حالت استثنا وجود دارد که کارایی را برهم زند. این بدین علت است که در دو الگوریتم رایج استفاده از جدول هش (یعنی باز یا بسته) امکان برخورد با جایگاه بیش از یک بار مصرف شده هست. در درهم‌سازی باز موقع درج امکان برخورد با جایگاه پر شده وجود دارد که با تغییر جایگاه و امتحان جایگاه کناری باز تا آخر به جایگاه‌های پر برخورد کرده و کارایی زمانی درج و سپس یافتن آن خطی می‌شود. در درهم‌سازی بسته نیز هم اگر با استفاده از لیست پیوندی برای جایگاه‌ها اجازه‌ی قرار دادن داده را سر جایش دهیم تا جایگاه داده‌ای دیگر را نگیریم، باز امکان کم ایجاد لیستی بزرگ برای یک خانه وجود دارد. به همین علت، می‌توانیم از الگوریتم سومی به نام درهم‌سازی فاخته (*Cuckoo Hashing*) برای حرکت در ساختار استفاده کنیم که کارایی زمانی بدترین حالت حذف آن نیز ثابت $O(1)$ است.

الگوریتم درهم‌سازی فاخته به طور ساده اجازه‌ی استفاده از بیش از یک تابع هش (به طور منوال ۲ عدد تابع) را می‌دهد. بدین صورت که هر داده‌ی ورودی در دو جایگاه (به زبانی دو جدول) می‌تواند قرار گیرد، و اگر در درج با استفاده از تابع هش اولیه جایگاهی را برای داده یافتیم که با داده‌ای دیگر پر است، داده جدید را جایگزین داده قبل کرده و داده‌ی قبل را با استفاده از تابع هش دوم به جدول دوم می‌بریم. حتی اگر در بدترین حالت نیاز به چند جایگزینی داشته باشیم، نه تنها زمان درج به مانند قبل در آنالیز استهلاکی ثابت است، بلکه عملیات جستجو و حذف در هر حالت (حتی بدترین) همواره ثابت $O(1)$ خواهند بود زیرا تنها نیاز به بررسی دو خانه برای داده برحسب کلید وجود دارد؛ این متفاوت با درهم سازی باز است که برای یافتن داده، در حالت نبود آن در جایگاه کلید خود باید تک تک خانه‌های کناری را بگردیم.

بنابراین، با استفاده از ساختار داده و راهکارهای بهینه‌سازی ارائه شده، می‌توانیم سرعت کار بیمارستان را بالا ببریم. به طور کلی، در پذیرش بیمارستان با گرفتن داده و کلید مربوطه (که کد ملی خاص فرد است)، نرم‌افزار کلید را هش کرده و اطلاعات را در زمان ثابت با درهم‌سازی فاخته به ساختار داده وارد می‌کند. و در ترخیص، با گرفتن کد ملی نرم افزار اطلاعات فرد را به کمک درهم‌سازی فاخته در زمان تضمین شده‌ی خطی یافته و از داده‌ها حذف می‌کند.

مراجع: ---

۲. فرض کنید دنباله‌ای از n مقدار $x_1, x_2, \cdots, x_n$ به ما داده شده است و ما به دنبال راهی هستیم که سریعاً و مکرراً پرسش‌هایی را پاسخ دهیم که به این شکل باشند: دو مقدار i و j را بگیرید و کوچک‌ترین مقدار در محدوده $x_i, \cdots, x_j$ را بیابید.

الف) ساختارداده‌ای را طراحی کنید که فضای مصرفی آن $O(n^2)$ باشد و بتوان با کمک آن، در زمان $O(1)$ به هر یک از پرسش‌ها پاسخ داد.

ب) ساختارداده‌ای را طراحی کنید که فضای مصرفی آن $O(n)$ باشد و بتوان با کمک آن، در زمان $O(\log n)$ به پرسش‌ها پاسخ داد.

جواب:

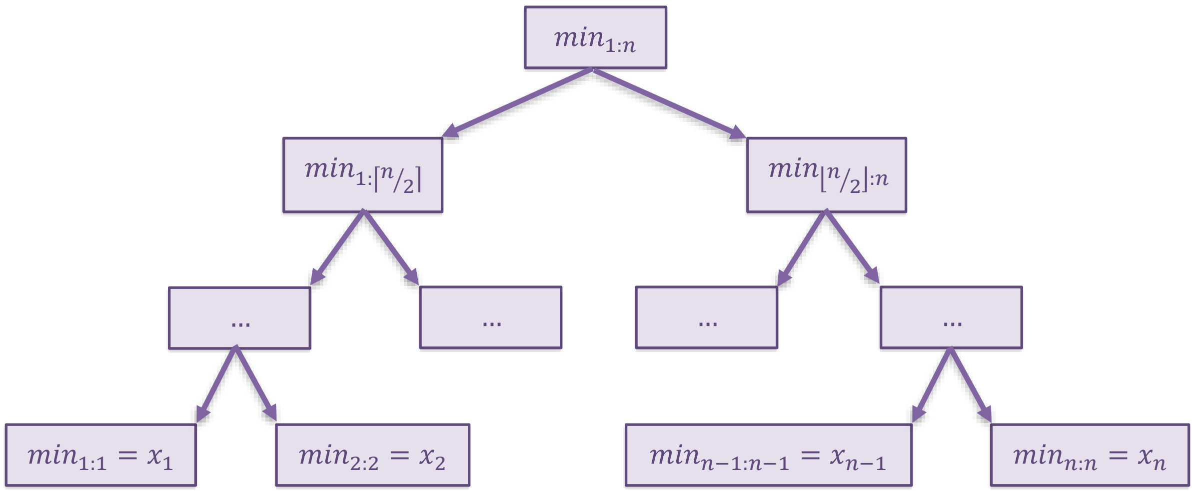
الف) می‌خواهیم ساختار داده‌ای برای پاسخ به پرسش در زمان $O(1)$ طراحی کنیم. از آنجا که ساختار داده فضای مصرفی $O(n^2)$ باید داشته باشد، می‌توانیم آن را آرایه‌ای دو بعدی در نظر بگیریم که اندیس x و y آن متناظر اندیس در دنباله اصلی مقادیر باشند. سپس در این آرایه، برای هر بازه از یک اندیس به اندیس دیگر در دنباله، کوچک‌ترین مقدار آن بازه را در خانه‌ی متقابل آن در آرایه دو بعدی ذخیره می‌کنیم. برای مثال برای $i = 2$ و $j = 5$ به خانه‌ی $(2, 5)$ در آرایه می‌رویم و مقدار $min_{i:j}$ (یعنی کوچک‌ترین مقدار که اندیس آن میان i تا j باشد) را در زمان ثابت $O(1)$ استخراج می‌کنیم. آرایه به طور کلی به شکل زیر نمایش داده می‌شود:

$$\begin{bmatrix} min_{1:1} & \cdots & min_{1:n} \\ \vdots & \ddots & \vdots \\ min_{n:1} & \cdots & min_{n:n} \end{bmatrix}$$

آرایه دو بعدی بوده و هر ضلع آن به طول دنباله یعنی n است و بنابراین کارایی فضایی آن $O(n^2)$ می‌باشد. همچنین جستجو در این آرایه تنها نیاز به داشتن مختصات نقطه دارد که همان i و j ورودی پرسش هستند، و بنابراین با زمان ثابت $O(1)$ به مقدار درون خانه‌ی مورد نیاز در آرایه دسترسی میابیم.

ب) این بار استفاده از آرایه کفاف پاسخ به این پرسش را نمی‌دهد و نیاز به طراحی ساختار داده‌ای متفاوت داریم. بهترین ساختار داده برای این کار، درخت بازه‌ها (*segment tree*) معروف است. درخت بازه‌ها یک درخت دودویی کامل است که گره‌هایش یا دو فرزند دارند یا هیچ. در این درخت، گره‌ها دارای مقادیری هستند که نماینده هر بازه‌ای از آرایه اصلی هستند (یعنی نشانگر یک مقادری از آن بازه، مثلا جمع یا در اینجا کوچکترین عضو آن). از بالا، ریشه نماینده کل آرایه، و فرزندان آن نماینده هر نیمه‌ی بازه می‌باشند؛ این الگو پیش رفته تا جایی که اعضای اصلی دنباله، برگ‌های درخت را شکل بدهند (یعنی نماینده بازه‌ی تک عضوی برای هر عضو آرایه باشند). درخت از چپ به راست منظم است.

شکل کلی درخت بازه‌های طراحی شده برای این سوال در صفحه‌ی بعد قابل مشاهده است:



برای محاسبه‌ی کارایی فضایی این ساختار داده از خواص درخت دودویی استفاده می‌کنیم تا کارا بودن آن را مشاهده کنیم. می‌دانیم که یک درخت دودویی کامل با n برگ، همواره $2n$ گره دارد، پس درخت بازه‌های یک آرایه n عضوی با محاسبه هدر رفتن فضا، در حالت کلی فضای خطی $O(n)$ نیاز دارد.

همچنین برای جستجو در این درخت و یافتن پاسخ به پرسش بر اساس i و j ورودی، به علت مرتب بودن درخت تنها نیاز به جستجوی دودویی و هر بار شکستن درخت به دو و جستجو در یک زیر درخت حاصل است. بنابراین کارایی زمانی این جستجو در اصل بستگی به عمق درخت دارد (زیرا از هر سطح یک بار گذر می‌کنیم)، که می‌دانیم در یک درخت دودویی کامل با n برگ، این عمق $\lceil \log n \rceil$ می‌باشد. پس در بدترین حالت (یعنی زمانی که نیاز به پیش رفتن تا پایان عمیق‌ترین بخش درخت باشد) کارایی زمانی جستجو در درخت بازه‌ها $O(\log n)$ است.

مراجع: ---

۳. فرض کنید P یک چندضلعی ساده n رأسی و q نقطه‌ای در صفحه باشد. (چندضلعی P ممکن است محدب باشد یا نباشد؛ و نقطه q ممکن است در داخل چندضلعی باشد یا نباشد.) الگوریتمی با کارایی $O(n\log n)$ طراحی کنید که با آن بتوان پاره‌خطی را یافت که یک نقطه انتهایی آن q باشد و حداکثر تعداد یال‌های چندضلعی P را قطع کند. الگوریتم خود را با شبه‌کد توصیف کنید.

جواب:

در نظر می‌گیریم که چندضلعی به صورت یک گراف (با دسته‌ای از مختصات رئوس و دسته‌ای از اضلاع) در دست باشد. برای حل این مسئله نیاز است به جای دستگاه مختصات دوبعدی عادی، از دستگاه مختصات قطبی استفاده کنیم؛ زیرا تنها برای ما مهم است که در صورت ایستادن در نقطه‌ی q، چه یال‌هایی در هر زاویه در دسترس هستند تا بیشترین تعداد جمع شده در یک زاویه دید را انتخاب کنیم. بنابراین در الگوریتم حل این مسئله، به تبدیل چندضلعی به بازه‌های زاویه‌ای پرداخته و سپس بازه‌ای که بیشترین تعداد ضلع در آن موجود بودند را انتخاب می‌کنیم و در آن پاره‌خط رسم می‌کنیم. برای این کار، به طور کل الگوریتم نیازمند ۵ مرحله است:

۱. اولین قدم تبدیل مختصات نقاط رأس به مختصات قطبی‌ست؛ البته ما با مختصات کامل (r,θ) کار نداشته و فقط زاویه θ را بر حسب درجه به دست می‌آوریم. این کار را با پیش رفتن در لیست رئوس و یافتن زاویه ساعت‌گرد نسبت به مرکز q با کمک فرمول انجام می‌دهیم. از آنجا که حفظ نقاط برای مرحله‌ی بعد و تشخیص ضلع‌های متصل به هر کدام مهم است، زاویه حاصل را به صورت ویژگی‌ای دیگر از جسم رأس ذخیره می‌کنیم.

۲. سپس لازم است با رئوس درجه‌ای و پیش رفتن در اضلاع، محدوده درجه‌ی هر ضلع (یعنی بازه زاویه‌ای که این ضلع در راستای دید از نقطه q است) را به دست آوریم، و هرجا ضلع شروع می‌شود «سر» بازه، و هرجا به پایان می‌رسد «ته» بازه در نظر بگیریم. نکته این است که برای تعیین سر و ته، آن دو را مقایسه کرده و هر کدام زاویه کوچک‌تری داشت سر بگیریم و دیگری را ته. همچنین پس از این مرحله دیگر با رئوس قبل کار نداریم، پس می‌توانیم اجسام اصل را کنار گذاشته و برای راحتی آرایه‌ای تشکیل دهیم که برای هر ضلع، دو زاویه با جنسشان در آن ذخیره شود. در این مرحله قطعا رئوس بیش از یک بار به لیست اضافه می‌شوند.

۳. حال باید آرایه حاصل از مرحله‌ی ۲ را که دارای زاویه سرها و ته‌های اضلاع است مرتب کنیم تا بتوانیم تشخیص دهیم با چرخش ساعت‌گرد به ترتیب از چه سر و ته‌هایی گذر می‌کنیم. این کار را با مرتب‌سازی ادغامی (*merge sort*) انجام می‌دهیم. این الگوریتم را انتخاب می‌کنیم زیرا نسبت به الگوریتم‌های مرتب‌سازی با زمان یکسان دیگر پایدار و مطمئن‌تر است و برای آرایه‌های بزرگ بهتر است؛ اگر آرایه ما کوچک بود و یا محدودیت فضا داشتیم، از نوعی از مرتب‌سازی سریع (*quicksort*) استفاده می‌کردیم که تصادفی شده و پایدار تر از مرتب‌سازی سریع عادی‌ست.

۴. پس از مرتب کردن آرایه، شمارنده‌ای تعریف کرده و در آرایه پیش می‌رویم. هر جا به یک سر رسیدیم به شمارنده اضافه کرده و هر جا به یک ته می‌رسیم از آن کم می‌کنیم. باید توجه کنیم که ما در آخر تعداد

بیشترین ضلع ممکن را نخواسته و در اصل پاره‌خط را می‌خواهیم، پس باید بازه‌های هر عدد شمارنده را حفظ کنیم. برای این کار آرایه‌ی دومی تعریف کرده و هر گاه شمارنده تغییر می‌کند و نقطه‌ی حال حاضرش با نقطه بعد یکسان نیست، بازه بین این زاویه تا زاویه بعد در لیست (که در آن تغییر ایجاد می‌شود) را در آرایه دوم به عنوان بازه‌ی این تعداد ضلع اضافه می‌کنیم. شرط را به این علت قرار داده‌ایم که نقاط در آرایه اول تکرار شده و در یک نقطه چند تغییر می‌تواند انجام شود؛ باید ابتدا تمام تغییرات ممکن در یک نقطه انجام شوند تا عدد شمارنده واقعا توصیف‌گر وضعیت در آن نقطه باشد.

۵. در مرحله آخر، در آرایه دوم پیش رفته و بیشترین مقدار شمارنده را تعیین می‌کنیم تا بازه آن را استخراج کنیم. برای این کار اولین عضو آرایه را به عنوان بیشترین ذخیره کرده و اگر به عضوی برخوردیم که شمارنده‌اش بزرگتر بود آن را به جای قبل ذخیره می‌کنیم. در انتها (پس از طی کردن آرایه دوم) از عنصر ذخیره شده، بازه مربوط به آن مقدار شمارنده را استخراج کرده و با میانگین گیری، زاویه میانی بازه را برای رسم پاره خط خروجی می‌دهیم.

شبه کد الگوریتم بالا در صفحه بعد قابل مشاهده است. از آن جا که شبه کد طولانی شده و مرتب سازی ادغامی الگوریتمی معروف است که در ترم‌های قبل مطالعه کرده‌ایم، مراحل آن را در شبه کد ذکر نکرده و تنها تابع آن را به عنوان تابعی از قبل تعریف شده استفاده می‌کنیم.

برای تحلیل کارایی زمانی الگوریتم طراحی شده، زمان هر مرحله از الگوریتم را جدا محاسبه کرده و در آخر با هم جمع می‌کنیم، زیرا این ۵ مرحله از نظر زمانی از هم مستقلند. پس برای مراحل ذکر شده داریم:

۱. عملیات‌های این مرحله شامل دو بخش مهم است: پیش رفتن در آرایه رئوس (که اندازه‌ی آن رابطه‌ی خطی با افزایش اندازه‌ی چندضلعی ورودی دارد)، و اعمال فرمول بر مختصات هر نقطه (که در زمان ثابت انجام شده و اعدادش رابطه خاصی در بزرگ شدنشان نسبت به ورودی وجود ندارد). بنابراین کارایی زمانی $O(|V|) + O(1)$ می‌شود که همان کارایی زمانی خطی $O(n)$ است.

۲. تنها عمل اصلی این مرحله پیش‌روی در اضلاع چندضلعی‌ست و اضافه کردن اعضا به آخر آرایه‌ای جدید تنها در زمان ثابت انجام می‌شود. پس زمان این مرحله $O(|E|) + O(1)$ یعنی همان $O(n)$ می‌باشد.

۳. در اینجا بزرگ‌ترین عملیات الگوریتم یعنی مرتب‌سازی آرایه‌ی اول (که دارای $2|V|$ عضو است) انجام می‌شود که به علت استفاده از مرتب‌سازی ادغامی، کارایی زمانی آن $O(2|V| \log 2|V|)$ یا همان $O(n \log n)$ است.

۴. به مانند مرحله ۱ و ۲، در یک آرایه پیش رفته و عمل ثابت انجام می‌دهیم. این بار آرایه، آرایه‌ی اول ایجاد شده است پس کارایی زمانی $O(2|V|) + O(1)$ یا همان $O(n)$ به دست می‌آید.

۵. در آخر در آرایه دوم پیش می‌رویم که زمان آن بیش از خطی یعنی $O(n)$ نخواهد بود.

پس برای زمان کلی الگوریتم داریم:

$$O(n) + O(n) + O(n \log n) + O(n) + O(n) = O(n \log n)$$

Algorithm: *IntersectMostEdges*(*P*, *q*)

```
// Finds line from point q that intersects most sides of the given polygon
// Input: Polygon P and point q
// Output: The clockwise degree at which the line should be drawn from q

for vertex in P.Vertices() do
    teta ← arctan(vertex.y ÷ vertex.x)
    vertex.angle ← (–teta + 90)mod360   // to make it clockwise with 0° as north
segments ← list()
for edge(v, w) in P.Edges() do
    if v.angle ≤ w.angle then
        segments.append([v.angle, ‘start’]) and segments.append([w.angle, ‘end’])
    elif v.angle > w.angle then
        segments.append([w.angle, ‘start’]) and segments.append([v.angle, ‘end’])
// to sort array segments based on the first element of its element pairs
MergeSort(segments)
counter ← 0
counts ← list()
for i in 0 to length(segments) – 2 do   // last i is index of last-but-one element of list
    if segments[i][1] = ‘start’ then counter ← counter + 1
    elif segments[i][1] = ‘end’ then counter ← counter – 1
    if segments[i][0] ≠ segments[i+1][0] then
        counts.append([counter, (segments[i][0], segments[i+1][0])])
highest ← counts[0]
for element in counts do
    if highest[0] < element[0] then highest ← element
bestangle ← (highest[1][0] + highest[1][1]) ÷ 2
return bestangle
```

مراجع:

- https://stackoverflow.com/questions/10474287/finding-a-ray-that-intersects-a-polygon-as-many-times-as-possible

۴. الف) الگوریتم کارایی را طراحی کنید که به عنوان ورودی، اشاره‌گر به ریشه درخت دودویی T را بگیرد و تعداد برگ‌های T را به عنوان خروجی چاپ کند. الگوریتم خود را با شبه‌کد توصیف کنید و کارایی زمانی آن را تعیین کنید.

ب) الگوریتم کارایی طراحی کنید که به عنوان ورودی، اشاره‌گر به ریشه درخت دودویی T را بگیرد و تعداد گره‌های پر T را به عنوان خروجی چاپ کند. الگوریتم خود را با شبه‌کد توصیف کنید و کارایی زمانی آن را تعیین کنید.

جواب:

الف) می‌خواهیم تعداد برگ‌های درختی دودویی (یعنی گره‌هایی که فرزند ندارند) را بیابیم. می‌توانیم از پیمایش پس‌ترتیب درخت استفاده کنیم و آن را هر بار به دو قسمت راست و چپ تقسیم کنیم. از آنجا که لازم است ریشه هر زیر درخت را به مانند درخت اول به الگوریتم وارد کنیم، می‌توانیم الگوریتم را بازگشتی طراحی کنیم تا ساده پیش رود.

نکته این است که قصد شمردن برگ‌هاست؛ برای این کار در شرط یافتن یک برگ (یعنی اگر فرزند راست و چپ، هیچ بودند)، عدد یک را برمی‌گردانیم تا برای هر زیر درخت، اعداد جمع شوند و برای درخت نهایی تعداد کل را بیابیم. البته درخت در اصل به گره‌های تهی پایان میابد، و در صورتی که مثلا به گره‌ای برسیم که فرزند راست داشته باشد اما فرزند چپ نداشته باشد، الگوریتم ادامه درخت را به دو شکسته و زیر درخت چپ را با گره‌ی خالی به پایان می‌رساند. پس باید شرط خالی بودن گره‌ی در حال بررسی را قرار دهیم و الگوریتم را با بازگرداندن عدد صفر به پایان رسانیم (صفر، زیرا در جمع به تعداد اضافه نکند، اما عدد باشد تا با خطای جمع برخورد نکنیم).

بنابراین الگوریتم سه بخش اصلی دارد که در شبه کد نیز قابل مشاهده است:

۱. حالت پایه هیچ بود گره: الگوریتم عدد صفر را برمی‌گرداند.
۲. حالت پایه برگ بودن گره (فرزندهایش هر دو هیچ باشند): الگوریتم عدد یک را برمی‌گرداند.
۳. حالت کلی: الگوریتم ادامه درخت را به دو شکسته و عدد حاصل از هر دو قطعه را با هم جمع می‌کند.

```
Algorithm: LeafCounter(root)

    // Utilizes post-order traversal of binary tree recursively to count its leaves
    // Input: The root of a binary tree
    // Output: The count of the binary tree's leaves

    if root is None then
        return 0
    elif root.right is None and root.left is None then
        return 1
    else then
        return LeafCounter(root.right) + LeafCounter(root.left)
```

در بررسی کارایی زمانی الگوریتم می‌دانیم که همواره هر گره یک بار بازدید می‌شود. پس برای درختی با n گره، کارایی زمانی همواره $\Theta(n)$ خواهد بود. این مسئله در شکل کلی کارایی زمانی الگوریتم نیز قابل مشاهده است، زیرا هر بار درخت به دو بخش شکسته شده لذا داریم:

$$T(n) = T(right) + T(n - right - 1) + O(1)$$

فرمول بالا بیانگر یک سطح از بازگشت است. برای تکرار در در هر بار بازگشت، دو حالت کلی داریم: (۱) یا درخت متعادل بوده، (۲) یا درخت نامتعادل است. فرمول را برای تعداد بازگشت‌های این دو حالت محاسبه می‌کنیم.

(۱) درخت متعادل است.

بنابراین هر بار تعداد گره‌های حدودا به دو تقسیم شده و داریم:

$$T(n) = 2T\left({}^{n}/_{2}\right) + O(1)$$

این فرمول از قضیه اصلی واکاوی الگوریتم‌ها (*Master theorem*) پیروی کرده و مشمول حالت یک آن می‌شود.

$$T(n) = aT\left({}^{n}/_{b}\right) + O(n^c) = 2T\left({}^{n}/_{2}\right) + O(1)$$
$$\xrightarrow{yields} a = 2, b = 2, c = 0 \rightarrow \log_b a = \log_2 2 = 1 > c = 0$$
$$\xrightarrow{Case\ 1} T(n) = \Theta\left(n^{\log_b a}\right) = \Theta\left(n^{\log_2 2}\right) = \Theta(n^1) = \Theta(n)$$

(۲) درخت نامتعادل است. برای مثال به چپ کج است.

بنابراین هربار درخت راست تهی (صفر) بوده و داریم:

$$\begin{aligned}
&T(n) = T(r) + T(n - r - 1) + O(1)\\
&= T(0) + T(n-1) + O(1)\\
&= 2T(0) + T(n-2) + 2O(1)\\
&= 3T(0) + T(n-3) + 3O(1)\\
&= \cdots\\
&= (n-1)T(0) + T(1) + (n-1)O(1)\\
&= nT(0) + T(0) + nO(1)\\
&= nO(1) + O(1) + nO(1)\\
&= (2n+1)O(1) = O(2n+1) = \Theta(n)
\end{aligned}$$

پس ثابت کردیم کارایی زمانی این الگوریتم در هر حالت $\Theta(n)$ به دست می‌آید.

ب) به مانند بخش (الف) کافی‌ست از پیمایش پسوندی درخت دودویی استفاده کنیم. تنها تفاوت با بخش قبل این است به جای بررسی خالی بودن فرزندهای گره باید وجود هر دو فرزند را بررسی کنیم تا گره را پر تعیین کنیم. واضح است که در این صورت شرط پر بودن گره دیگر شرط پایان الگوریتم نبوده و باید مسیر را ادامه دهیم؛ بنابراین باید تغییری در الگوریتم قبل داده و به جای قرار دادن بخش شکستن درخت به دو در یک شرط جداگانه، آن را از شرط خارج کرده و در هر صورت انجام دهیم. در این صورت تنها شرط پایه شرط اول الگوریتم قبل، یعنی شرط تهی بودن گره است (که نشانگر این است که الگوریتم با انتها این زیردرخت رسیده است). با این تغییر، به تعریف شمارنده‌ای نیاز پیدا می‌کنیم که در هر بار یافتن یک گره‌ی دارای دو فرزند به آن اضافه شود، و برای حفظ این شمارنده در بازگشت، برای هر زیر درخت صفر تعریف شده و پس از محاسبه گره‌های پر آن، جمع و برگردانده شود.

شبه کد الگوریتم به صورت زیر است:

Algorithm: *FullNodeCounter*(*root*)

// Utilizes post-order traversal of binary tree recursively to count its full nodes
// Input: The *root* of a binary tree
// Output: The count of the binary tree's full nodes

if *root* **is** *None* **then**

return 0

counter ← 0

elif *root.right* **and** *root.left* **then** // both children exist

counter ← *counter* + 1

counter ← *counter* + *FullNodeCounter*(*root.right*) + *FullNodeCounter*(*root.left*)

return *counter*

کارایی زمانی الگوریتم دقیقا مانند الگوریتم بخش (الف) بوده و بنابر اثبات صفحه قبل همواره برابر با $\Theta(n)$ خواهد بود.

مراجع:

- https://www.geeksforgeeks.org/write-a-c-program-to-get-count-of-leaf-nodes-in-a-binary-tree
- https://www.geeksforgeeks.org/count-full-nodes-binary-tree-iterative-recursive

۵. فرض کنید آرایه‌ی A شامل n پاره‌خط نامتقاطعِ $s_1, s_2, \dots, s_n$ باشد که نقاط انتهایی هر یک از آنها روی خطوط $y = 0$ و $y = 1$ قرار گرفته باشند و اینکه پاره‌خط‌ها از چپ به راست، در آرایه مرتب شده باشند. الگوریتمی طراحی کنید با کارایی زمانی $O(\log n)$ که نقطه q را که $0 < y(q) < 1$ باشد، بگیرد و یا آن پاره‌خط s_i ای در آرایه A را تعیین کند که اولین پاره‌خطِ واقع در سمت راست نقطه q باشد، یا آنکه مشخص کند که نقطه q در سمت راست تمام پاره‌خط‌ها قرار گرفته است. الگوریتم خود را با شبه‌کد توصیف کنید.

جواب:

برای جستجو در زمان $O(\log n)$ در یک آرایه منظم نیاز به جستجوی دودویی داریم. نکته مهم برای جستجو در این آرایه (که دارای دو نقطه‌های سر پاره‌خط‌هاست) باید نقطه‌ی q ورودی را با هر پاره‌خط مقایسه کنیم تا موقعیت نسبی آن را تشخیص دهیم. سپس در صورت راست بودن نقطه، جستجو را ادامه دهیم. برای این کار، از خاصیت ضرب خارجی دو نقطه استفاده می‌کنیم.

می‌دانیم که حاصل ضرب خارجی دو نقطه زمانی مثبت است که زاویه بینشان با مبدأ $(0, 0)$ پادساعت‌گرد باشد، و در غیر این صورت (یعنی زمانی که ساعت‌گرد باشد) منفی‌ست. اگر نقطه‌ی پایینی پاره‌خط‌ها (یعنی نقطه‌ای که روی خط $y = 0$ می‌باشد، با نام a) را یکی از نقطه‌ها و نقطه‌ی q را نقطه‌ای دوم گرفته، و نقطه‌ی بالایی (روی خط $y = 1$، با نام b) را مبدأ مختصات در نظر گیریم، مشاهده می‌کنیم که پاره‌خط هر طور که باشد، اگر نقطه در راست آن باشد حاصل ضرب خارجی دو نقطه مثبت می‌شود، اگر چپ آن باشد منفی می‌شود، و اگر روی آن باشد صفر می‌شود.

با استفاده از این خاصیت و فرض اینکه در آرایه ورودی، هر پاره‌خط به شکل دو نقطه دو سرش داده شده که نقطه اول زوج مرتب (x, y) نقطه روی خط $y = 0$ بوده و نقطه دوم زوج مرتب (x, y) نقطه روی خط $y = 1$ باشد، و نقطه q نیز به صورت زوج مرتب (x, y) باشد، می‌توانیم به راحتی الگوریتم مورد نیاز را طراحی کنیم. بدین صورت که در جستجو در آرایه با جستجوی دودویی، برای هر پاره‌خط سه نقطه مد نظر را ابتدا بر مبنای نقطه بالایی پاره‌خط تغییر می‌دهیم تا آن نقطه به روی مبدأ مختصات جابه‌جا شود و موقعیت نسبی دو نقطه دیگر حفظ شود، سپس با ضرب خارجی دو نقطه دیگر به سنجش راست یا چپ بودن نقطه q بپردازیم.

شبه کد الگوریتم در صفحه بعد قابل مشاهده است. برای محاسبه کارایی زمانی آن می‌دانیم از آنجا که از الگوریتم جستجوی دودویی استفاده کرده‌ایم و عمل‌های اضافه انجام شده زمان بر نبوده و همتا با n تعداد پاره‌خط تغییری نمی‌یابند (که بزرگ یا کوچک شوند)، در بدترین حالت کارایی زمانی الگوریتم را $\Theta(\log n)$ داریم زیرا بنا بر قضیه اصلی واکاوی الگوریتم‌ها (*Master theorem*) برای زمان ثابت در اعمال اضافه فرم الگوریتم را داریم که:

$$T(n) = aT\left(\frac{n}{b}\right) + O(n^c) = T\left(\frac{n}{2}\right) + O(1)$$

$$\xrightarrow{yields} a = 1, b = 2, c = 0 \longrightarrow \log_b a = \log_2 1 = 0 = c = 0$$

$$\xrightarrow{Case\ 2} T(n) = \Theta(n^c \log n) = \Theta(n^0 \log n) = \Theta(\log n)$$

Algorithm: ***PointPosition*(*lines*[[$(x_{1a}, y_{1a}), (x_{1b}, y_{1b})$], … , [$(x_{na}, y_{na}), (x_{nb}, y_{nb})$]], *q*$(x, y)$)**

// Uses iterative binary search to find position of point *q* compared to lines in array
// Input: Array *lines* with two points of lines and coordinates of a point *q*
// Output: A line if one is found to the right of point, *None* if not found

start ← 0
end ← *length*(*lines*) – 1
while *start* ≤ *end* **do**
  mid ← $\lfloor (start + end)/2 \rfloor$
  a ← (*lines*[*mid*][0][0] – *lines*[*mid*][1][0], *lines*[*mid*][0][1] – *lines*[*mid*][1][1])
  point ← (*q*[0] – *lines*[*mid*][1][0], *q*[1] – *lines*[*mid*][1][1])
  product ← *a*[0] × *point*[1] – *a*[1] × *point*[0]
  if *product* = 0 **and** *mid* ≠ *length*(*lines*) – 1 **then**
    return *lines*[*mid* + 1]
  elif *product* < 0 **then**
    if *mid* = 0 **then**
      return *lines*[*mid*]
    else then // check the line to the left
      a2 ← (*lines*[*mid* – 1][0][0] – *lines*[*mid* – 1][1][0], *lines*[*mid*– 1][0][1]
        – *lines*[*mid*– 1][1][1])
      point2 ← (*q*[0] – *lines*[*mid* – 1][1][0], *q*[1] – *lines*[*mid* – 1][1][1])
      product2 ← *a2*[0] × *point2*[1] – *a2*[1] × *point2*[0]
      if *product2* > 0 **or** *product2* = 0 **then**
        return *lines*[*mid*]
      elif *product2* < 0 **then**
        end ← *mid* – 1
  elif *product* > 0 **then**
    end ← *mid* + 1
// if while loop ends and a line to the right of point is not found in array
return *None*

مراجع:

- https://www.geeksforgeeks.org/direction-point-line-segment