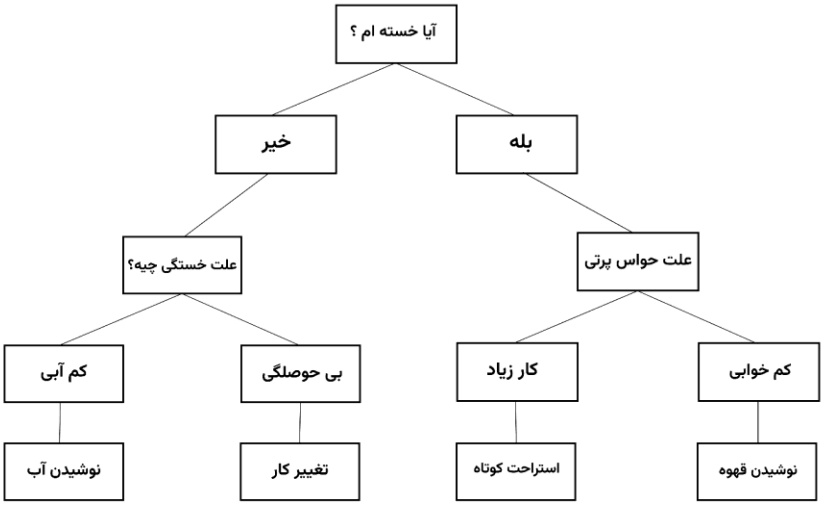
Data Structures and algorithms
Instructor : **Dr.Amin Eskandari**
Head-Ta's : Hamid Namjoo , Amir Hossein Hemati , Ali Mojahed
Ta's : Amir Mohammad Asadjoo , Padina razmjooi , Armin Janfaza , Alireza Heydari , Mahdi Torabi ,
Maryam Ghorbani , Fatemeh Keshavarz , MohammadReza Fasli , Abolfazl Tadrisi

Week 05 Answers

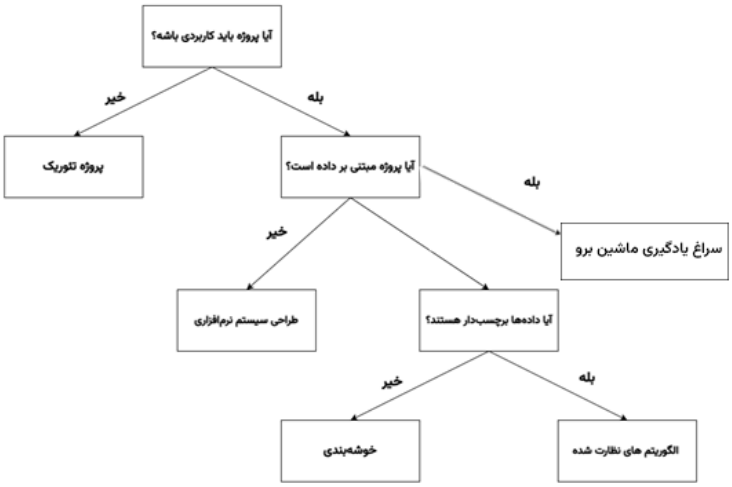
پاسخنامه تکالیف هفته ۰۵

پاسخ سوال اول:

پاسخ بخش الف :

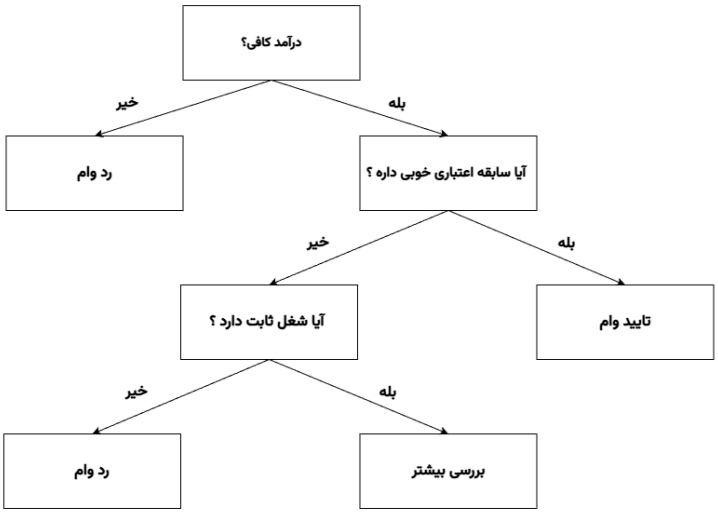


پاسخ بخش ب :



نکته همیشه درخت تصمیم دودویی کامل نیست و ممکن به این شکل هم باشند

پاسخ بخش ج :



پاسخ سوال دوم :

🚀 گزارش فنی سیستم جدید: "پروتکل تریاژ پایدار"

مخاطب: فرماندهی ناو اِتِلگارد (و هر کسی که می‌خواهد منطق پشت کد را درک کند)

کاپیتان، مشکل قبلی ما این بود که وقتی هزاران خرابی همزمان رخ می‌داد، سیستم قدیمی ما (لیست خطی) آنقدر کند بود که تا بخواهد مهم‌ترین مشکل را پیدا کند، کشتی منفجر می‌شد. ما به سیستم Heap (هیپ) رو آوردیم که مثل یک درخت جادویی، همیشه "خطرناک‌ترین" مشکل را در نوک قله نگه می‌دارد.

اما یک باگ خطرناک وجود داشت: سیستم "بی‌عدالت" بود! اگر دو خرابی با شدت یکسان (مثلاً ۸۰) رخ می‌داد، سیستم تضمین نمی‌کرد که اولی زودتر تعمیر شود.

کدی که نوشتیم، این مشکل را با یک ترفند هوشمندانه به نام «سیستم نوبت‌دهی بانکی» حل کرده است.

۱. راز جادوی کد: سه‌تایی‌های هوشمند (The Smart Tuple)

در سیستم استاندارد، ما فقط دو چیز را ذخیره می‌کردیم: (میزان خطر، نام خرابی).

اما در سیستم جدید، ما یک متغیر مخفی به نام entry_count (شمارنده ورود) اضافه کردیم. تصور کنید جلوی در ورودی سیستم پردازش، یک دستگاه نوبت‌دهی گذاشته‌ایم.

هر هشداری که وارد می‌شود، تبدیل به یک بسته سه تایی می‌شود:

[میزان خطر، شماره نوبت، نام خرابی]

- میزان خطر (Priority): عددی که شما تعیین می‌کنید (مثلاً ۱۰۰ برای خطر انفجار).
- شماره نوبت (Entry Count): یک عدد که خودکار بالا می‌رود (۱، ۲، ۳...). هر هشداری که زودتر بیاید، عددش کوچکتر است.
- نام خرابی (Item): متن پیام (مثلاً "خرابی سپر").

۲. قانون مقایسه: چطور برنده را انتخاب می‌کنیم؟

وقتی سیستم (Heap) می‌خواهد تصمیم بگیرد کدام هشدار مهم‌تر است و باید بالاتر قرار بگیرد، به این روش عمل می‌کند (این منطق در توابع _is_higher_priority کد پیاده شده است):

قدم اول (زورآزمایی):

به میزان خطر نگاه کن. اگر هشدار A خطرش ۱۰۰ است و هشدار B خطرش ۵۰ است، A برنده است. تمام.

قدم دوم (رعایت نوبت):

اگر میزان خطر برابر بود (مثلاً هر دو ۸۰)، چه کنیم؟

اینجاست که سیستم ناپایدار قدیمی گیج می‌شد. اما سیستم جدید به شماره نوبت نگاه می‌کند. کسی که شماره نوبت کوچکتری دارد (یعنی زودتر رسیده)، برنده است.

مثال واقعی در کشتی:

۱. ساعت ۱۲:۰۰:۰۱ -> خرابی طبقه ۴ (خطر ۸۰) وارد می‌شود. سیستم به آن نوبت #۱ می‌دهد.

۲. ساعت ۱۲:۰۰:۰۵ -> خرابی طبقه ۵ (خطر ۸۰) وارد می‌شود. سیستم به آن نوبت #۲ می‌دهد.

حالا هیپ این دو را مقایسه می‌کند:

خطرها برابرند ($80 = 80$).

پس نوبت‌ها چک می‌شوند: نوبت ۱ کوچکتر از ۲ است (یعنی قدیمی‌تر است).

نتیجه: طبقه ۴ اولویت بالاتر دارد و زودتر تعمیر می‌شود. (پایداری تضمین شد)

۳. عملکرد توابع اصلی

(وارد کردن هشدار) تابع (Push)

وقتی ناوبان کل دکمه ثبت خطا را می‌زند:

۱. کد یک شماره نوبت منحصربه‌فرد به خطا می‌چسباند.

۲. آن را به انتهای درخت اضافه می‌کند.

۳. سپس عملیات Sift Up (غربال رو به بالا) شروع می‌شود: هشدار جدید مثل حباب بالا می‌رود تا جایی که مطمئن شود از هشدارهای کم‌خطرتر بالاتر قرار گرفته و اگر با کسی هم‌خطر بود، زیر دستِ قدیمی‌ترها قرار می‌گیرد.

تابع Pop (تعمیر کردن/خارج کردن)

وقتی کاپیتان می‌گوید «بعدی را تعمیر کن»:

۱. کد بلافاصله نوک قله درخت را برمی‌دارد (چون به لطف ساختار هیپ، همیشه مهم‌ترین و قدیمی‌ترین آنجاست).

۲. سپس آخرین هشدار ته لیست را موقتاً می‌گذارد نوک قله.

۳. عملیات Sift Down (غربال رو به پایین) شروع می‌شود: این هشدار موقت، سنگین است و پایین می‌رود تا جای درستش را پیدا کند و دوباره تعادل برقرار شود.

۴. چرا این روش سریع است؟ (پیچیدگی زمانی)

کاپیتان پرسید: «آیا اضافه کردن این سیستم نوبت‌دهی، سرعت ما را کم نمی‌کند؟»

پاسخ: خیر!

در علوم کامپیوتر، مقایسه دو عدد (مثلاً چک کردن اینکه آیا نوبت ۱ < نوبت ۲ است) تقریباً هیچ زمانی نمی‌برد.

- پیدا کردن مهم‌ترین هشدار: آنی ($O(1)$).
- مرتب‌سازی دوباره درخت بعد از هر ورود یا خروج: متناسب با ارتفاع درخت است که می‌شود لگاریتمی ($O(\log N)$).

یعنی حتی اگر ۱,۰۰۰,۰۰۰ هشدار داشته باشیم، سیستم فقط نیاز دارد حدود ۲۰ مقایسه انجام دهد تا جای درست یک هشدار را پیدا کند. این بسیار بسیار سریع‌تر از روش قدیمی است که باید ۱,۰۰۰,۰۰۰ هشدار را چک می‌کرد.

پاسخ سوال سوم :

میانه در پنجره لغزان با حذف تنبل

۱. درک مسئله: تعادل و جریان

تصور کنید شهر **داده‌سرا** یک سد است که آب (داده‌ها) از یک طرف وارد و از طرف دیگر خارج می‌شود. مدیران به **عدد میانی** (عددی که نیمی از داده‌ها کوچک‌تر و نیمی بزرگ‌تر از آن هستند) اهمیت می‌دهند.

چالش اصلی:

۱. **جریان مداوم:** داده‌ها دائماً در حال ورود هستند (مانند استریم).

۲. **پنجره لغزان (اندازه k):** ما فقط به kعدد **آخر** اهمیت می‌دهیم. با ورود عدد جدید، قدیمی‌ترین عدد باید حذف شود.

اگر بخواهیم هر بار kعدد را مرتب کنیم، بسیار کُند است ($O(k\log\ k)$ برای هر قدم). راه‌حل ما باید خیلی سریع باشد ($O(\log\ k)$).

۲. راه‌حل بنیادی: استراتژی دو هیپ (Two-Heaps)

سارا راه حل درخشانی ارائه داد که بر اساس تقسیم‌بندی داده‌ها است:

۱. **Max-Heap (استخر اوبسیدین):** این هیپ فقط اعداد **کوچک‌تر** از میانه را نگه می‌دارد. چون یک Max-Heap است، بزرگ‌ترین عدد این نیمه (که نزدیک‌ترین عدد به میانه از سمت پایین است) همیشه در دسترس است (رأس هیپ).

۲. **Min-Heap (استخر کوارتز):** این هیپ فقط اعداد **بزرگ‌تر** از میانه را نگه می‌دارد. کوچک‌ترین عدد این نیمه (که نزدیک‌ترین عدد به میانه از سمت بالا است) همیشه در دسترس است (رأس هیپ).

تعادل (Balancing): برای اینکه میانه همیشه در دسترس باشد، اندازه این دو هیپ باید تقریباً برابر باشد (اختلاف اندازه حداکثر ۱).

مزیت: میانه همیشه یکی از این دو رأس است ($O(1)$ برای دسترسی).

۳. چالش جدید: حذف کارآمد (The Deletion Problem)

وقتی پنجره لغزان می‌شود، ما باید عددی را که kگام پیش وارد شده، از این دو هیپ **حذف** کنیم.

مشکل هیپ‌ها: حذف یک عنصر دلخواه (که لزوماً رأس نیست) از یک هیپ معمولاً زمان‌بر است ($O(N)$ یا $O(\log N)$با پیچیدگی اضافی). ما نمی‌خواهیم این کار را انجام دهیم.

راه‌حل سارا: حذف تنبل (Lazy Deletion)

به جای حذف فوری، ما فقط **علامت‌گذاری** می‌کنیم که یک عدد باید حذف شود.

۱. **نقشه‌ی حذف (delayed):** یک دیکشنری (نقشه) ایجاد می‌کنیم که به ازای هر عددی که باید حذف شود، تعداد دفعاتی که باید حذف گردد را ثبت می‌کند. مثلاً اگر عدد ۵ باید حذف شود، می‌نویسیم delayed[۵] = ۱.

۲. **درج و تعادل:** درج و متعادل‌سازی طبق روال عادی انجام می‌شود.

۳. **پاک‌سازی (clean):** قبل از اینکه بخواهیم میانه را محاسبه کنیم یا یک عنصر را از رأس هیپ **حذف واقعی** کنیم، همیشه بررسی می‌کنیم:

- آیا عنصری که در رأس Max-Heap است، در نقشه‌ی delayed علامت‌گذاری شده؟
- اگر بله، آن را **واقعاً حذف** می‌کنیم (heappop) و شمارنده‌اش را در delayed کم می‌کنیم، و این کار را آنقدر تکرار می‌کنیم تا به عنصری برسیم که نباید حذف شود (یا هیپ خالی شود).

پاسخ سوال چهارم :

تورنمنت خرده‌بلورها و راز الگوریتم کوییک‌سورت

۱. داستان اصلی و فلسفه مسئله

در پادشاهی دیتاون، چالش بزرگی وجود دارد: کوهی عظیم از بلورها با اندازه‌های گوناگون در اختیار داریم و هدف نهایی، مرتب‌سازی آن‌ها از کوچک به بزرگ است. لیرا، جادوگر الگوریتم‌ها، راهکاری هوشمندانه ارائه می‌دهد. او معتقد است به‌جای تلاش برای مرتب کردن کل کوه بلورها به صورت یک‌جا، بهتر است آن را به بخش‌های کوچک‌تر تقسیم کرده و هر بخش را جداگانه مدیریت کنیم. این رویکرد، دقیقاً همان فلسفه‌ی زیربنایی الگوریتم «مرتب‌سازی سریع» یا Quick Sort است.

۲. جادوی اصلی: استراتژی تقسیم و حل

لیرا برای اجرای این ایده، ابتدا یک بلور را به عنوان معیار انتخاب می‌کند که به آن «محور» یا Pivot می‌گوییم. سپس تمام بلورهای کوچک‌تر از محور را به سمت چپ و تمام بلورهای بزرگ‌تر را به سمت راست می‌فرستد. با این کار، مسئله به دو کوه کوچک‌تر تبدیل می‌شود. او همین فرآیند را برای هر کدام از این کوه‌ها تکرار می‌کند. در علوم کامپیوتر، به این روش استراتژی «تقسیم و حل» (Divide & Conquer) می‌گویند که پایه‌ی اصلی سرعت و کارایی این الگوریتم است.

۳. چرا این روش سریع است؟

اگر تقسیم‌بندی‌ها متعادل باشند (یعنی در هر مرحله کوه تقریباً نصف شود)، تعداد مراحل تقسیم متناسب با $\log n$خواهد بود. از آنجا که در هر مرحله تمام عناصر یک‌بار بررسی می‌شوند، پیچیدگی زمانی کل الگوریتم به $O(n\log\ n)$می‌رسد. این نتیجه نشان‌دهنده‌ی سرعت بسیار بالای الگوریتم در حالت ایده‌آل است.

۴. خطر بزرگ: انتخاب محور اشتباه

اما داور دانا به نکته‌ای حیاتی اشاره می‌کند: اگر انتخاب محور هوشمندانه نباشد (مثلاً همیشه بزرگ‌ترین یا کوچک‌ترین بلور انتخاب شود)، تقسیم‌بندی نامتوازن خواهد شد. در این حالت فاجعه‌بار، یک طرف شامل $n-1$بلور و طرف دیگر خالی می‌ماند. این اتفاق باعث می‌شود عمق بازگشت بسیار زیاد شده و الگوریتم به جای نصف کردن مسئله، هر بار فقط یک عنصر را کم کند. در نتیجه، پیچیدگی زمانی به $O(n^2)$سقوط می‌کند که بسیار کند و ناکارآمد است.

۵. چالش و محدودیت: ممنوعیت تصادفی‌سازی

چالش اصلی زمانی ایجاد می‌شود که داور شرطی سخت‌گیرانه می‌گذارد: شما اجازه ندارید از انتخاب تصادفی (Random) استفاده کنید. انتخاب محور باید کاملاً قطعی (Deterministic) باشد و الگوریتم حتی روی آرایه‌هایی که از قبل مرتب شده‌اند یا چیدمان خاصی دارند نیز نباید دچار افت سرعت شود. سؤال اینجاست: چطور بدون شانس و تصادف، محوری را انتخاب کنیم که تعادل را حفظ کند؟

۶. راهکار نجات‌بخش: تکنیک میانه‌ی سه‌تایی

پاسخ هوشمندانه، روشی به نام «میانه‌ی سه‌تایی» (Median of Three) است. در این روش، لیرا به جای انتخاب کورکورانه‌ی عنصر اول یا آخر، سه بلور را بررسی می‌کند: اولین بلور، بلور وسط و آخرین بلور. سپس بلوری را که اندازه‌اش بین دوتای دیگر است (میانه)، به عنوان محور انتخاب می‌کند.

۷. چرا این روش فوق‌العاده است؟

این تکنیک تضمین می‌کند که محور انتخاب شده نه خیلی کوچک است و نه خیلی بزرگ، بنابراین تقسیم‌ها متعادل‌تر انجام می‌شوند. چه آرایه مرتب باشد، چه معکوس و چه تقریبا مرتب، این روش عمق بازگشت را کنترل کرده و مانع از سقوط الگوریتم به $O(n^2)$می‌شود. از آنجا که انتخاب همیشه از سه جایگاه مشخص انجام می‌شود و هیچ عدد تصادفی‌ای در کار نیست، شرط «قطعی بودن» مسئله نیز کاملاً رعایت می‌شود و میانگین عملکرد الگوریتم در حد مطلوب $O(n\log\ n)$باقی می‌ماند.

پاسخ سوال پنجم :

زنجیره‌ی اعداد و قاضیِ سریع (Quick Sort با Linked List)

صورت مسئله و مفهوم لیست پیوندی

در این مسئله، هدف اصلی مرتب‌سازی اعداد است، اما این کار با روش‌های معمول تفاوت دارد. برخلاف تمرین‌های متداول که اعداد در یک آرایه یا لیست معمولی (مانند [7, 2, 9, 4, 1, 5]) قرار دارند، در اینجا اعداد به صورت زنجیره‌ای یا همان «لیست پیوندی» (Linked List) به یکدیگر متصل شده‌اند. در این ساختار، هر عدد مانند نوشته‌ای روی یک کاغذ است که تنها یک فلش دارد و به عدد (کاغذ) بعدی اشاره می‌کند (مثلاً $7 \rightarrow 2 \rightarrow 9 \rightarrow 4 \rightarrow 1 \rightarrow 5$). ویژگی حیاتی این زنجیره آن است که دسترسی تصادفی به اعداد ممکن نیست؛ یعنی نمی‌توان ناگهان به وسط زنجیره پرید و باید حتماً گره‌به‌گره از ابتدا حرکت کرد. همچنین، در فرآیند مرتب‌سازی، شکستن زنجیره برای ساخت گره‌های جدید یا کپی‌کردن اعداد ممنوع است و تنها مجاز به تغییر جهت فلش‌ها (Linkها) هستیم.

الگوریتم قاضی سریع و استراتژی تقسیم و غلبه

برای حل این مسئله از الگوریتم «قاضیِ سریع» یا همان **Quick Sort** استفاده می‌شود. این الگوریتم بر پایه‌ی ایده‌ی مهم «تقسیم و غلبه» (Divide and Conquer) بنا شده است؛ به این معنا که اگر حل کل مسئله به صورت یک‌جا دشوار باشد، آن را به مسائل کوچک‌تر تقسیم کرده و هر کدام را جداگانه حل می‌کند. در این الگوریتم، همیشه یک عدد به عنوان «محور» یا **Pivot** انتخاب می‌شود که در این سناریو، Pivot همیشه اولین گره‌ی زنجیره است (برای مثال در زنجیره‌ی بالا، عدد ۷ به عنوان محور انتخاب می‌شود).

نحوه عملکرد و بازگشت (Recursion)

قاضیِ سریع با یک بار پیمایش کل زنجیره، آن را بر اساس مقدار Pivot به سه بخش منطقی تقسیم می‌کند: بخش اول شامل اعداد کوچک‌تر از Pivot (زنجیره‌ی چپ)، بخش دوم خود Pivot (وسط) و

بخش سوم شامل اعداد بزرگ‌تر یا مساوی Pivot (زنجیره‌ی راست). نکته‌ی بسیار مهم این است که در این فرآیند هیچ عددی کپی نمی‌شود و هیچ گره‌ی جدیدی ساخته نمی‌شود، بلکه صرفاً اتصالات یا فلش‌های بین گره‌ها جابه‌جا می‌شوند. پس از این تقسیم‌بندی، الگوریتم متوجه می‌شود که زنجیره‌های چپ و راست هنوز مرتب نیستند؛ بنابراین با استفاده از مفهوم «بازگشت» (Recursion)، تابع دوباره خودش را برای مرتب‌سازی زنجیره‌ی چپ و سپس زنجیره‌ی راست صدا می‌زند. این روند تا زمانی که همه‌ی زیرزنجیره‌ها به طول یک یا صفر برسند، تکرار می‌شود.

تحلیل پیچیدگی زمانی

دلیل محبوبیت Quick Sort این است که اگر Pivotها به درستی انتخاب شوند و زنجیره تقریباً به دو نیم تقسیم شود، سرعت اجرا بسیار بالا خواهد بود و پیچیدگی زمانی آن برابر با $O(n\log\ n)$می‌شود؛ یعنی با افزایش تعداد اعداد، زمان اجرا رشد معقولی دارد. اما در بدترین حالت، اگر زنجیره از قبل تقریباً مرتب باشد (مثلاً $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$) و Pivot همیشه اولین عنصر باشد، یک طرف تقسیم‌بندی خالی مانده و طرف دیگر شامل کل زنجیره می‌شود. این موضوع باعث عمیق شدن بیش‌ازحد بازگشت‌ها شده و پیچیدگی زمانی را به $O(n^2)$می‌رساند که منجر به کند شدن برنامه می‌گردد.