طراحی الگوریتم

جزوه سوم - الگوریتم های حریصانه

الگوریتم‌های حریصانه یکی از روش‌های پرکاربرد و مشهور در حوزه علوم کامپیوتر هستند که برای حل مسائلی با ویژگی‌های خاص مورد استفاده قرار می‌گیرند. این الگوریتم‌ها در مسائل مختلفی از جمله کمینه‌سازی، برش و بسته‌بندی، و ترتیب‌دهی کاربرد گسترده‌ای دارند. ویژگی اصلی الگوریتم‌های حریصانه این است که در هر مرحله از حل مسئله، بدون توجه به تأثیر تصمیم فعلی بر مراحل بعدی، بهترین تصمیم ممکن را انتخاب می‌کنند. این رویکرد، سرعت بالای این الگوریتم‌ها در مقایسه با روش‌های دیگر را تضمین می‌کند و آن‌ها را برای حل سریع برخی از مسائل مناسب می‌سازد.

از مزایای مهم الگوریتم‌های حریصانه، سادگی پیاده‌سازی است. هنگامی که شرایط مسئله امکان انتخاب‌های حریصانه را فراهم می‌کند، استفاده از این الگوریتم‌ها انتخابی معقول و کارآمد خواهد بود. با این حال، الگوریتم‌های حریصانه همیشه به جواب بهینه نمی‌رسند و در برخی موارد ممکن است نتایج نادرست یا زیر بهینه بدهند؛ به ویژه زمانی که تصمیمات هر مرحله بر نتیجه کلی تأثیر زیادی داشته باشد.

حل مسائل با استفاده از الگوریتم‌های حریصانه معمولاً دو مرحله دارد. در مرحله اول، باید ایده اصلی الگوریتم را به‌درستی طراحی کنیم و معیار انتخاب تصمیم‌های حریصانه را مشخص نماییم. در مرحله دوم که اهمیت ویژه‌ای دارد، نیاز است بهینه بودن الگوریتم پیشنهادی را اثبات کنیم. این اثبات می‌تواند به روش‌های مختلفی از جمله برهان خلف صورت گیرد تا اطمینان حاصل شود که الگوریتم به جواب مطلوب و بهینه دست پیدا می‌کند.

۱. Fractional Knapsack Problem

مسئله کوله‌پشتی را از فصل برنامه‌نویسی پویا به یاد دارید. در این حالت خاص از این مسئله، می‌توانیم از هر آیتم، یک بخش را برداریم. همچنان هدف این است که لوازم به گونه‌ای برداشته شوند که مجموع وزن آنها کمتر از وزن کوله‌پشتی شود، در حالی که بیشترین ارزش را دارند.

پاسخ :

انتخاب حریصانه این است که از آیتم با بیشترین نسبت ارزش به وزن، بیشترین مقدار را برداریم و اگر هنوز هم کوله‌پشتی جا داشت، سراغ آیتم باقی‌مانده با بیشترین نسبت ارزش به وزن می‌رویم. حال با استفاده از برهان خلف به اثبات بهینگی الگوریتم می‌پردازیم.

فرض کنید آیتم‌ها بر اساس نسبت ارزش به وزن خود مرتب شده‌اند. $ALG = \{p_1, p_2, ..., p_n\}$ را جواب الگوریتم حریصانه ما به مسئله در نظر بگیرید که بهینه نیست و در عین حال جواب $OPT = \{q_1, q_2, ..., q_n\}$ وجود دارد که بهینه است. توجه شود که p_i ها و q_i ها نشان دهنده مقدار آیتم i در کوله‌پشتی می‌باشند. فرض کنید j اولین اندیسی باشد که مقدار آن در ALG و OPT با هم متفاوت می‌باشند. طبق الگوریتم حریصانه می‌دانیم که $p_j > q_j$ و به همین علت یک k وجود دارد که در آن $p_k < q_k$ باشد. حال ما می‌توانیم یک جواب جدید به نام OPT' ایجاد کنیم که نسبت به OPT مقداری از آیتم j بیشتر و به همان مقدار از آیتم k کمتر دارد. به این ترتیب وزن کوله‌پشتی ثابت می‌ماند اما ارزش آن بیشتر می‌شود. در ابتدا گفته شد که OPT جواب بهینه است اما OPT' جواب بهتری از آن بود پس به تناقض برخوردیم.

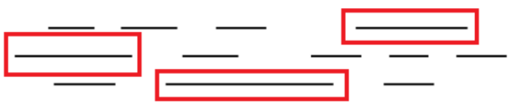
۲. Activity Selection Problem

مسئله انتخاب بهینه فعالیت‌ها، یکی از مسائل کلاسیک الگوریتم‌های حریصانه است که کاربرد زیادی در مسائل برنامه‌ریزی زمان و مدیریت منابع دارد. در این مسئله، تعدادی فعالیت وجود دارد به طوری که هر فعالیت i، یک زمان آغاز (s_i) و یک زمان پایان (f_i) دارد. هدف این است که بیشینه تعداد فعالیت را انجام دهیم به گونه‌ای که هیچ دو فعالیتی که انجام می‌دهیم، اشتراک زمانی نداشته باشند.

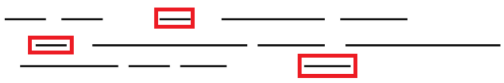
پاسخ :

در ابتدای مواجهه با مسئله، ممکن است روش‌های مختلفی ارائه شود، که آن‌ها را به ترتیب بررسی می‌کنیم.

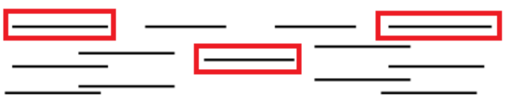
روش اول) یکی از روش‌ها می‌تواند انتخاب بر اساس ترتیب زمان شروع فعالیت‌ها باشد. شکل زیر مثال نقضی برای این روش است، زیرا با این روش تنها امکان انجام ۳ فعالیت داریم، در حالی که جواب بهینه ۶ است.



روش دوم) در این روش، در هر مرحله، فعالیتی که طول انجام آن کوتاه‌تر است را انتخاب می‌کنیم. شکل زیر نشان دهنده مثال نقضی برای این روش می‌باشد.



روش سوم) در این روش، فعالیتی که با تعداد کمتری از دیگر فعالیت‌ها اشتراک زمانی دارد، انتخاب می‌شود. شکل زیر نشان دهنده مثال نقضی برای این روش می‌باشد.



پس از بررسی روش‌های نادرست، حال روش حریصانه صحیح را بیان و سپس بهینه بودن آن را اثبات می‌کنیم. روش صحیح، انتخاب فعالیتی است که زمان پایان آن از بقیه فعالیت‌ها زودتر باشد. در این روش ابتدا باید فعالیت‌ها را بر اساس زمان پایانشان مرتب کرده و سپس فعالیتی که زمان پایان زودتر دارد را انتخاب کنیم؛ حال از فعالیت‌هایی که با آن اشتراک زمانی دارند گذر می‌کنیم و به همین ترتیب ادامه می‌دهیم. شبه کد الگوریتم به صورت زیر خواهد بود: مرتبه زمانی این الگوریتم، $O(n \times logn)$ است.

```
ActivitySelector(s, f) {
    n = s.length
    A = {a1}
    k = 1
    for m = 2 to n {
        if s[m] ≥ f[k] {
            A = A ∪ {am}
            k = m
        }
    }
    return A
}
```

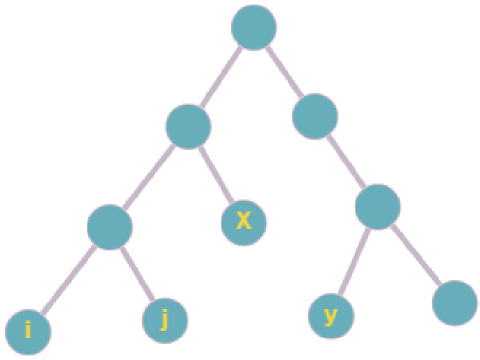
اثبات بهینگی الگوریتم با استفاده از برهان خلف:
فرض کنید که $ALG = \{p_1, p_2, ..., p_n\}$ جواب الگوریتم حریصانه ما به مسئله باشد که بهینه نیست و در عین حال جواب $OPT = \{q_1, q_2, ..., q_n\}$ جواب بهینه‌ای باشد که بیشترین شباهت را به ALG دارد. همچنین می‌دانیم که اعضای ALG و OPT بر اساس زمان پایان فعالیت‌ها مرتب شده‌اند. فرض کنید p_i و q_i اولین فعالیتی باشند که ALG و OPT در آن متفاوتند. حال می‌توانیم جواب OPT' را به گونه‌ای تعریف کنیم که همه فعالیت‌های OPT به جز q_i را داشته باشد و p_i را جایگزین آن کند. تعداد اعضای OPT' برابر OPT است و همچنین از آنجا که در ALG ما همواره فعالیت با نزدیک‌ترین زمان پایان را انتخاب می‌کنیم، قطعا زمان پایان p_i نزدیک‌تر از q_i می‌باشد پس OPT' علاوه بر بهینه بودن، معتبر نیز می‌باشد. حال می‌توانیم همین روند را آنقدر ادامه دهیم تا ALG و OPT یکی شوند و این با فرض ابتدایی ما در تضاد است پس ALG جواب بهینه ما است.

۳. Huffman Codes

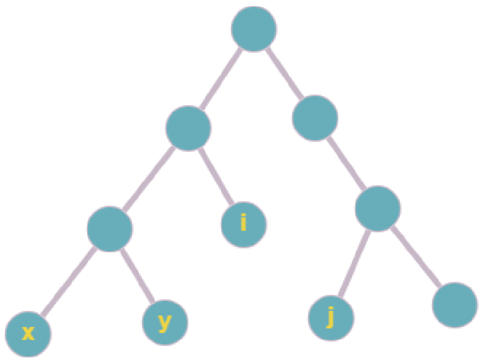
کدهای هافمن یک روش فشرده‌سازی داده است که برای کاهش حجم اطلاعات به کار می‌رود. این تکنیک به ویژه در زمینه فشرده‌سازی فایل‌های متنی استفاده می‌شود. اما کاربرد الگوریتم‌های حریصانه در این عمل فشرده سازی چیست؟ در روش کدهای هافمن برای هر کاراکتر یک کد باینری معادل در نظر می‌گیریم. سپس با توجه به تعداد تکرار هر کاراکتر در متن به گونه‌ای کد باینری‌ای را به آن نسبت دهیم که متن decode شده، کمترین حجم ممکن را داشته باشد. همچنین باید توجه کنیم که هیچ کدام از این کدها پیشوند دیگری نباشد. در این کار الگوریتم‌های حریصانه به کمک ما می‌آیند تا به بهترین روش عمل decode کردن صورت گیرد. در این مسئله ورودی ما یک فایل شامل n کاراکتر است.که در آن تعداد تکرار کاراکتر i معادل f_i می‌باشد. هدف این است که کد c_i را به طول h_i به کاراکتر i به گونه‌ای نسبت دهیم که مقدار $\Sigma f_i \times h_i$ کمینه شود و هیچ کدی پیشوند دیگری نباشد.

پاسخ :

انتخاب حریصانه ما در این حالت این است که هر بار دو کاراکتر با کمترین فرکانس را انتخاب کرده و برایشان پدر مشترک در نظر بگیریم. حال می‌خواهیم اثبات کنیم که این انتخاب ما بهینه می‌باشد. فرض کنید که ما یک راه حل بهینه داریم که در آن درختی معادل شکل ابتدای صفحه بعد ایجاد شده است.



تصور می‌کنیم که کاراکترهای x و y دارای کمترین فرکانس هستند ولی در بیشترین عمق قرار نداشته و این یک راه بهینه است. حال به یک جابه‌جایی این درخت بهینه را به صورت زیر تغییر می‌دهیم. اکنون باید نشان دهیم که هزینه این درخت جدید از درخت قبلی بیشتر نیست.



ارتفاع هر نود در درخت اولیه که معادل همان طول هر رشته کد می‌باشد، به صورت زیر می‌باشد:

$$h_i = h_j = h$$

$$h_x = h'$$

$$h_y = h''$$

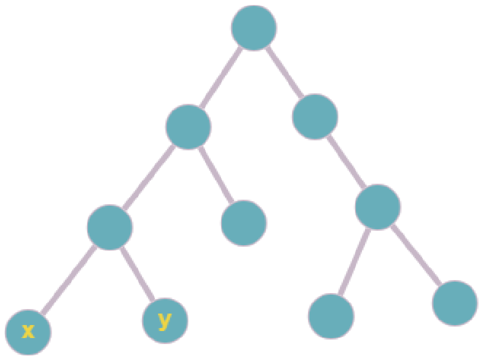
بنابراین هزینه در این درخت معادل $f_x \times h' + f_y \times h'' + f_i \times h + f_j \times h$ خواهد شد که طبق فرضمان کمترین هزینه است. حال در درخت جدید هزینه به صورت $f_x \times h + f_y \times h + f_i \times h' + f_j \times h''$ می‌باشد. ادعا می‌کنیم طبق محاسبات زیر این هزینه بیشتر از هزینه راه‌حل بهینه نخواهد بود.

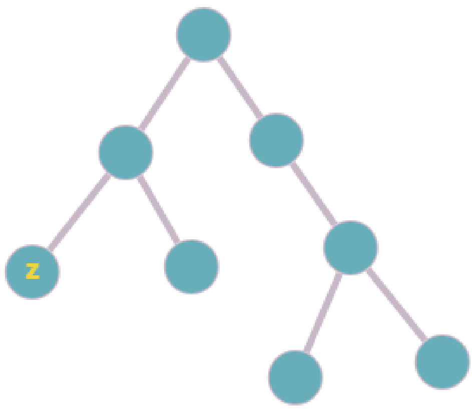
$$f_x \times h' + f_y \times h'' + f_i \times h + f_j \times h \geq f_x \times h + f_y \times h + f_i \times h' + f_j \times h''$$

$$f_x(h' - h) + f_y(h'' - h) + f_i(h - h') + f_j(h - h'') \geq 0$$

$$(f_x - f_i)(h' - h) + (f_y - f_j)(h'' - h) \geq 0$$

حال طبق فرض اولیه‌مان $f_x, f_y \leq f_i, f_j$ و h بیشترین ارتفاع است. بنابراین طبق این دو فرض، حاصل عبارت بالا همواره بزرگتر یا مساوی صفر خواهد بود. پس دیدیم که اگر ما نودهای i, j را با نودهای x, y جابه‌جا کنیم، هزینه‌ای به درخت ما اضافه نخواهد کرد و ما توانستیم با انتخاب حریصانه خود از روی جواب بهینه به جوابی برسیم که هزینه‌ای به ما اضافه نمی‌کند. حال می‌خواهیم اثبات کنیم مسئله ما خاصیت زیرمسئله‌های بهینه را دارد. فرض می‌کنیم درخت زیر درختی است که با الگوریتم حریصانه خود آن را ساخته‌ایم. می‌خواهیم ببینیم که آیا با انتخاب متداوم انتخاب حریصانه، مسئله‌ای که باقی می‌ماند همچنان ساختار بهینه را دارد یا خیر.





اگر اختلاف هزینه این دو درخت را محاسبه کنیم، به یک مقدار ثابت می‌رسیم (درخت اول معادل T و درخت دوم معادل T' است و همانطور که در شکل‌ها مینیموم نود z از مرج کردن نودهای x, y به دست آمده):

$$cost(T) - cost(T') = f_x h + f_y h - f_z(h-1) = (f_x + f_y)h - (f_x + f_y)(h-1) = f_x + f_y$$

ما می‌دانیم می‌توانیم درخت T را از روی یک درخت T' بسازیم. فرض می‌کنیم T' درخت بهینه ما می‌باشد. حال ما می‌توانیم در این درخت تنها با جایگذاری یک نود به نام z با دو نود به نام x, y به T برسیم، اما هنوز نمی‌دانیم که این یک درخت بهینه است یا خیر. حال فرض خلف می‌کنیم که T درخت بهینه ما نیست و یک درخت دیگری به نام T'' وجود دارد که بهینه است:

$$cost(T'') < cost(T)$$

حال بر اساس خاصیت greedy که بالاتر توضیح دادیم، می‌توانیم فرض کنیم که x, y در این درخت T'' برادر هستند (چرا که ثابت کردیم می‌توانیم بدون افزایش هزینه یک درخت بهینه را به گونه‌ای نودهایش را جابه‌جا کنیم که این خاصیت greedy حفظ شود). حال یک درخت دیگری از روی T'' می‌سازیم به نام T''' به این صورت که نودهای x, y در این درخت با نود z جایگزین شده است. بنابراین هزینه این درخت معادل زیر خواهد شد:

$$cost(T''') = cost(T'') - f_x - f_y$$

که در اینجا طبق فرضمان ($cost(T'') < cost(T)$):

$$cost(T''') < cost(T) - f_x - f_y$$

و همچنین دیده بودیم که:

$$cost(T') = cost(T) - f_x - f_y$$

بنابراین:

$$cost(T''') < cost(T')$$

که این یک تناقض است! چرا که فرض کرده بودیم که درخت T' یک درخت بهینه است. حال یک درختی ساختیم که آن هم بهینه است ولی هزینه‌اش از دیگری کمتر می‌شود که این یک تناقض است. بنابراین فرض اولیه ما اشتباه بوده و $cost(T'') < cost(T)$ برقرار نیست. پس T یک درخت بهینه است و یک کد پیشوندی بهینه به ما نمایش می‌دهد. اثبات شد که با انجام انتخاب حریصانه در هر مرحله، خاصیت بهینه بودن درخت از بین نمی‌رود.

۴. Coin Changing

می‌خواهیم مبلغ X را با استفاده از سکه‌هایی به ارزش های مختلف بدست آوریم (به عبارتی x را بین سکه‌ها تقسیم کنیم) به گونه‌ای که کمترین تعداد سکه ممکن استفاده شود.

پاسخ :

به‌طور کلی برای رسیدن به پاسخ صحیح این سوال باید از روش برنامه ریزی پویا استفاده کرد اما درستی استفاده از روش حریصانه بستگی به ارزش سکه های موجود دارد،

در ادامه، مثالی عددی از حالات مختلف می‌زنیم و سپس عملکرد کلی این الگوریتم را برای این مسئله بررسی می‌کنیم. فرض کنید ارزش سکه‌ها ۱ و ۵ و ۱۰ و ۲۵ باشد. می‌توان مسئله خرد کردن پول را به‌صورت حریصانه حل کرد:

- از بزرگترین سکه (۲۵) شروع کرده و تا جایی که ممکن است از این سکه استفاده می‌کنیم.
- سپس از سکه‌های ۱۰ برای باقیمانده مبلغ استفاده کرده و به همین ترتیب با سکه‌های ۵ و ۱ ادامه می‌دهیم.

می‌خواهیم ثابت کنیم که "این الگوریتم حریصانه همیشه جواب بهینه را ارائه می‌دهد"، یعنی حداقل تعداد سکه‌ها برای پرداخت مبلغ X را فراهم می‌کند.

اثبات: فرض کنید فقط از سکه هایی با ارزش ۱و۵و۲۵ استفاده می کنیم و در پایان خواهیم گفت چگونه از ۱۰ استفاده می کنیم

لم۱: در الگوریتم حریصانه از هر سکه کمتر از ۲۵ حداکثر چهار بار استفاده می‌شود.

اثبات لم۱: از برهان خلف کمک می‌گیریم. به برهان خلف فرض کنید یک سکه داریم که پنج بار از آن استفاده کرده‌ایم. با توجه به این فرض، وضعیتی پیش خواهد آمد که الگوریتم حریصانه از سکه‌ای با ارزش ۵k۵k۵k پنج بار استفاده کرده است،

در حالی که این امکان وجود داشت که به جای پنج بار استفاده از این سکه، یک بار از یک سکه به ارزش $5(k+1)$ استفاده کرد. با توجه به اینکه می‌توانستیم از مقدار بزرگتری استفاده کنیم بنابراین این اتفاق با حریصانه بودن الگوریتم تناقض دارد. لذا فرض غلط و ثابت می‌شود از هر سکه حداکثر چهار بار استفاده شده است.

لم۲: جواب الگوریتم حریصانه در این حالت، یکتاست.

اثبات لم۲: فرض کنیم دو جواب برای نتیجه الگوریتم حریصانه داریم، و می‌خواهیم از جواب اول به دومی برسیم. در حال حاضر از هر سکه حداکثر چهار بار استفاده شده است. با کم کردن از تعداد استفاده سکه‌هایی که یک یا چند بار استفاده شده‌اند، باید از سکه‌هایی با ارزش کمتر استفاده کرد، که این برخلاف روش حریصانه است، چرا که بزرگترین ارزش سکه ممکن انتخاب نشده است.

با زیاد کردن استفاده از سکه‌هایی که صفر تا چهار بار استفاده شده‌اند، باید از تعداد سکه‌هایی با ارزش بیشتر کم کرد تا مبلغ کل تغییر نکند. در این صورت باز هم برخلاف الگوریتم حریصانه است، چرا که در این حالت نیز بزرگترین ارزش سکه ممکن انتخاب نشده است.

اثبات بهینه بودن: فرض کنیم دنباله حریصانه و بهینه متفاوت باشند. طبق لم۲ جواب الگوریتم حریصانه یکتاست لذا الگوریتم بهینه نمی‌تواند از سکه‌های مختلف استفاده کرده باشد و همان جواب حریصانه را پیدا کند. بنابراین از یک سکه بیش از چهار بار استفاده شده است، در صورتی که طبق لم۱ می‌توانستیم به جای آن از یک سکه با ارزش پنج برابر آن استفاده کنیم. پس پاسخ بهتری از بهینه پیدا کردیم و در نتیجه به تناقض برخوردیم. لذا در این حالت الگوریتم حریصانه پاسخ بهینه را برمی‌گرداند.

حال که به جواب بهینه در این حالت رسیدیم، سکه ۱۰ را استفاده می کنیم، طبق اثبات ممکن نیست تعداد سکه ۱ استفاده شده از ۱۰ بیشتر باشد چرا که حتما کمتر از ۵ است، اما تعداد سکه های پنج میتواند بین ۰ تا ۴ باشد و بگونه ای سکه های ۵ را به ۱۰ تبدیل می کنیم که حداقل سکه استفاده شود و در این حالت از سکه ۱۰ می‌توان حداکثر دوباره استفاده کرد تا جواب بهینه پیدا شود بنابراین نشان داده شد به کمک سکه های ۱و۵و۱۰ و ۲۵ میتوان به روش حریصانه جواب بهینه را پیدا کرد.

حال تصور کنید فقط باید از سکه های ۱،۱۰ و ۲۵ باید استفاده کرد

در اینصورت روش حریصانه پاسخ بهینه نخواهد داشت و برای آن مثال نقض ارائه میکنیم

مثلا اگر $X = 30$ باشد اگر از روش حریصانه استفاده کنیم

ابتدا سکه ۲۵ را انتخاب کرده و سپس پنج سکه به ارزش ۱ انتخاب می کنیم و نهایتا با ۶ سکه تقسیم انجام میشود.

در حالیکه روش بهینه استفاده از ۳ سکه ۱۰ بود و در این حالت الگوریتم حریصانه جواب بهینه را نخواهد داد.

یک اشتباه رایج:

برای ارائه مثال نقض توجه کنید به گونه X انتخاب شود که حتما قابل تقسیم بین سکه ها باشد و در صورتی که

با روش حریصانه قابل تقسیم بود و روش بهینه دیگری وجود داشت آن موقع میتوان بهینه نبودن این راه حل را نشان داد.

مثلا اگر ارزش سکه ها ۵ و ۱۰ می‌بود و مقدار ۱۳ را می‌خواستیم تقسیم کنیم نمی‌توانستیم بگوییم چون قابل تقسیم نیست پس این روش بهینه نیست . چرا که این روش به طور کلی برای این حالت قابل استفاده نیست و این موضوع بر بهینه بودن اولویت دارد.

جمع بندی:

یکی از حالاتی که برای این مسئله الگوریتم حریصانه جواب دارد این است که ارزش سکه ها توان های متوالی یک عدد طبیعی باشند

بررسی کنید دیگر چه حالاتی وجود دارد که این الگوریتم بهینه است؟

۵. گاو آهن

در مزرعه‌ای n گاو نر داریم، که هر کدام از آن‌ها زوری به اندازه s_i دارند. می‌خواهیم که این گاوها را به دسته‌های دوتایی تقسیم کنیم تا بتوانند گاوآهن را بکشند. در صورتی که گاو i و j در یک گروه باشند، باید $s_i + s_j \geq P$ باشد تا بتوانند گاوآهن را بکشند (P وزن گاوآهن می‌باشد). الگوریتمی حریصانه از مرتبه $O(nlogn)$ برای بیشینه کردن تعداد گروه‌ها ارائه دهید و درستی آن را اثبات کنید.

پاسخ :

راه حل حریصانه: قوی‌ترین و ضعیف‌ترین گاو را در نظر می‌گیریم. اگر این گروه شرایط مدنظر را برآورده کنند، آن‌ها را در یک تیم قرار می‌دهیم. در غیر این صورت، ضعیف‌ترین گاو را حذف می‌کنیم. این الگوریتم را به صورت بازگشتی روی گاهای باقی مانده اجرا می‌کنیم تا تمامی گروه‌ها تشکیل شوند. برای پیدا کردن گروه‌ها بر حسب قدرت گاوها، آن‌ها را بر حسب قدرتشان مرتب می‌کنیم که در $O(nlogn)$ قابل انجام است. برای تشکیل داده گروه‌ها نیز کافی است روی مجموعه مرتب‌شده، از اول و آخر آن شروع کنیم و این پوینترها را آپدیت کنیم که در $O(n)$ قابل انجام است. حال ثابت می‌کنیم که این روش، درست است: لم۱: فرض کنید s قوی‌ترین گاو و w ضعیف‌ترین گاو است. در صورتی که $s + w < P$ باشد، نمی‌توان w را در هیچ گروهی قرار داد.

اثبات: اگر در $Teams_2$ ، هردوی s, w در تیم‌هایی حضور نداشته باشند، $Teams_1$ را همان $Teams_2$ در نظر می‌گیریم، با این تفاوت که تیم‌هایی که s یا w عضو آن هستند را حذف می‌کنیم و تیم (s, w) را نیز اضافه می‌کنیم. چون حداکثر یک تیم حذف شده و یک تیم هم اضافه شده، خواهیم داشت: $|Teams_2| \leq |Teams_1|$.

در غیر این صورت فرض کنید در $Teams_2$ ، s با x و w با y در یک گروه قرار گرفته باشند، در این صورت داریم: $Teams_1 = Teams_2 - \{(s,x),(w,y)\} \cup \{(s,w),(x,y)\}$ در این مجموعه، s, w در یک گروه قرار گرفته‌اند و با توجه به اینکه قدرت x حداقل به اندازه w است، اندازه دو مجموعه با هم برابر بوده و در مجموعه جدید نیز تمامی گروه‌ها می‌توانند گاوآهن را بکشند. در هر دو حالت لم اثبات شد.

حال با استفاده از این دو لم، درستی روش را اثبات می‌کنیم.

قضیه: هیچ مجموعه‌ای از تیم‌های مجاز مثل $Teams_2$ وجود ندارد که اندازه آن بزرگتر از مجموعه تیم‌های تولید شده توسط این روش باشد.

این قضیه را با استقرای قوی روی تعداد گاوهای نر (n) اثبات می‌کنیم. فرض کنید به ازای این الگوریتم به ازای

تعداد گاوهای کمتر از n صحیح است. برای تعداد گاوهای نر برابر با n، فرض کنید مجموعه‌ای از تیم‌ها وجود دارد که اندازه آن بزرگتر از مجموعه تیم‌های تولید شده توسط این روش است. فرض کنید s قوی‌ترین گاو و w ضعیف‌ترین گاو است. در صورتی که $s + w < P$، با توجه به لم اول، در هر دو مجموعه هیچ تیمی وجود ندارد که w یکی از آن‌ها باشد. طبق استقرا، پاسخ الگوریتم حریصانه برای مجموعه گاوها $-w$ درست است و لذا برای مجموعه گاوها هم درست خواهد بود.

حال در صورتی که $s + w \geq P$، طبق لم ۲ یک مجموعه از تیم‌ها مثل $Teams_1$ وجود دارد که s, w در یک تیم بوده و اندازه آن بزرگتر یا مساوی $Teams_2$ است. در صورتی که مجموعه حاصل از الگوریتم را $GreedyTeams$ و مجموعه گاوها را $Oxens$ در نظر بگیریم، چون $GreedyTeams - \{(s, w)\}$ طبق استقرا یک پاسخ بهینه برای مجموعه $Oxens - \{s, w\}$ است. همچنین $Teams_1 - \{(s, w)\}$ نیز یک پاسخ قابل قبول برای مجموعه $Oxens - \{s, w\}$ است و داریم:

$$|Teams_2 - \{(s, w)\}| \leq |Teams_1 - \{(s, w)\}| \leq |GreedyTeams - \{(s, w)\}|$$

بنابراین داریم:

$$|Teams_2| - 1 \leq |Teams_1| - 1 \leq |GreedyTeams| - 1$$

لذا پاسخ الگوریتم در این حالت نیز بهینه خواهد بود و طبق استقرا، پاسخ الگوریتم به ازای هر N بهینه می‌باشد.

۶. Other Notes

الگوریتم حریصانه همیشه انتخابی را می‌کند که در لحظه بهترین به نظر می‌رسد. الگوریتم حریصانه در هر مرحله انتخابی انجام می‌دهد که در همان لحظه بهینه‌ترین یا بهترین به نظر می‌رسد و هدفش رسیدن به یک راه‌حل بهینه کلی است. در این الگوریتم، انتخاب‌ها بدون بازبینی مراحل قبلی انجام می‌شوند.

مسائلی که با برنامه‌ریزی پویا حل می‌شوند، نمی‌توانند با الگوریتم‌های حریصانه حل شوند. برنامه‌ریزی پویا برای مسائلی مناسب است که راه‌حل بهینه آنها نیاز به بررسی چندین انتخاب در هر مرحله دارد، نه اینکه در هر مرحله یک انتخاب بهینه محلی انجام دهیم. در برنامه‌ریزی پویا، برای جلوگیری از محاسبات تکراری، نتایج زیرمسئله‌ها ذخیره می‌شوند و این کار به یافتن راه‌حل بهینه کمک می‌کند.

اگر مسئله‌ای بتواند هم با روش حریصانه و هم با برنامه‌ریزی پویا بهینه حل شود، روش حریصانه معمولاً پیچیدگی زمانی کمتری خواهد داشت. دلیل این موضوع این است که الگوریتم‌های حریصانه اغلب با یک بار عبور از داده‌ها (یا تعداد کمی عبور) و با تصمیم‌گیری در همان لحظه کار می‌کنند، بدون نیاز به محاسبه و ذخیره نتایج زیرمسئله‌ها که در برنامه‌ریزی پویا مورد نیاز است.

الگوریتم برنامه‌نویسی پویا ممکن است زمانی که الگوریتم حریصانه نمی‌تواند، یک راه‌حل بهینه تولید کند. این به این دلیل است که الگوریتم حریصانه برخی از راه‌حل‌های ممکن را رد می‌کند، در حالی که الگوریتم برنامه‌نویسی پویا تمام راه‌حل‌های ممکن را بررسی می‌کند.

گرادیان نزولی یک الگوریتم حریصانه است که برای پیدا کردن مینیمم محلی یک تابع استفاده می‌شود. در این روش، در هر مرحله با حرکت در جهت مخالف گرادیان، تابع کاهش می‌یابد. این فرآیند تکرار می‌شود تا الگوریتم به مینیمم محلی برسد. با این حال، ممکن است در مینیمم‌های محلی گیر کند و نتواند به مینیمم گلوبال برسد.