سوال یک :　　محمد طاها مجلسی　　۸۱۰۱۰۱۵۰۴

برای این سوال به این صورت عمل می‌کنیم

چون نیاز داریم با کمترین تعداد سکه‌ها باقی‌مانده‌ی پول مشتری را به او پس [بدهیم]

در ابتدا سکه‌ها را به صورت براساس مقدار Value آن‌ها به ترتیب Sort می‌کنیم. (فرض کنیم که از هر سکه (از هر نوع ارزش) بی‌شمار داریم)

حال از بزرگ‌ترین سکه آن که مقدار آن از مقدار پول اصلی کمتر باشد شروع می‌کنیم و هر تعداد که می‌توانیم از آن برمی‌داریم.

و بعد تا جایی که مقدار باقی‌مانده‌ی پول ما از مقدار Value آن سکه کمتر شود بعد به سراغ سکه‌ی بعدی می‌رویم.

و همین کار را بعد از آن ادامه می‌دهیم. (یعنی به صورت بازگشتی دوباره برای سکه‌هایی که Sort شده بودند)

این کار را ادامه می‌دهیم تا جایی که یا باقی‌مانده سکه‌ها تمام شود و یا اگر نتوان با آن مقدار سکه‌ها

مقدار باقی‌مانده‌ی پول را داد ← در این حالت نمی‌توان آن پول را با آن سکه‌ها داد.

که ما این حالت را در نظر نمی‌گیریم یعنی فرض می‌کنیم که این حالت رخ نخواهد داد.

به طور کلی کد آن به این صورت خواهد شد:

```
def min_coins ( coins , amount )

    coins.sort()  ------>

    remain = amount
    coin_count = 0 ->
    for coin in coins
        if remaining_count == 0 :
            break.
        num_coins = remaining // coin  ------>
        coin_count += num_coins
        remain -= num_coin * coin

    if remain != 0
        return -1
    return coin_counts.
```

در این حالت بیشترین زمان اجرا را داریم که $O(V\log V)$

در بهترین حالت خواهد شد.

تعداد سکه‌های استفاده شده.

* سه آمدن سکه‌ها از یک شروع شده تا n که به یک اضافه شده باشند هر کدام.

1, 2, ... n

مقدار آن کمتر از $V\log V$

و در غیر این صورت می‌شود که باز هم از $O(V\log V)$ خواهد بود

تا جایی که کمتر از coin شود.

①

• اثبات سوال یک :

حال باید اثبات شود که الگوریتم به درستی کار می‌کند پس به این صورت عمل می‌کنیم :

• ایده اصلی الگوریتم حریصانه :

ایده اصلی این الگوریتم این است که هر بار بزرگ‌ترین سکه را در نظر بگیریم که از مقدار باقی مانده کمتر یا مساوی بوده باشد سپس این مقدار را از باقی مانده کم می‌کنیم و این را تا زمانی که باقی مانده صفر شود ادامه خواهیم داد.

اثبات بهینه بودن : حال اگر فرض شود که الگوریتم حریصانه جوابی با تعداد k سکه را به ما داده است اما یک جواب k' وجود دارد که بهینه است و $k' < k$

• حال با توجه به این که ما داریم در هر مرحله از ما داریم بزرگ‌ترین سکه را از میان سکه‌هایی که می‌توان برداشت انتخاب می‌کنیم پس اگر k' ای وجود داشته باشد که جواب بهینه باشد و $k' < k$ باشد باید یکی از دو حالت زیر نقض شده باشد :

① با استفاده از بزرگ‌ترین سکه ممکن منجر به افزایش تعداد سکه‌ها می‌شود که این اشتباه است چون اگر سکه ای با ارزش کمتر را انتخاب کنیم مقدار باقی مانده بیشتر خواهد شد و تعداد سکه‌های بیشتری می‌توان برداشت چون مقدار باقی مانده بیشتری داریم. پس اگر بزرگ‌ترین سکه را که می‌توانیم برداریم مقدار باقی مانده که می‌ماند کمترین مقدار ممکن است که می‌تواند باشد و حداقل تعداد سکه‌ها انتخاب می‌شود.

② و یا این که تعداد کمتری از سکه‌های دیگر همان مقدار از باقی مانده را پوشش بدهند که این هم اشتباه است. چون هر بار داریم بیشترین مقداری از value را در سکه‌ها برمی‌داریم پس کمترین مقدار باقی مانده را خواهیم داشت و کمترین مقدار باقی مانده چون بازه را بزرگ‌ترین سکه ممکن کم می‌کنیم پس کمترین تعداد سکه‌ها انتخاب خواهد شد به نوعی می‌توان گفت که اگر ما بزرگ‌ترین سکه را برنداریم پس مقدار باقی مانده آن یعنی $remain' < remain$ الگوریتم حریصانه خواهد شد و در نتیجه خواهیم داشت که چون یک سکه باقی مانده مقدار $coin' > coin$ خواهد شد.

و چون فرض کرده بودیم که الگوریتم $coin' < coin_{opt}$ شده است خلاف فرض اصلی بوده و درست بودن و optimum بودن اثبات می‌شود.

سوال ② : کدهای ۳ بیتی جدیدای که می‌توان از آن ها استفاده کرد:

• تمام ۳ عدد دیگر با C تداخل دارند.

111 → C

010 →

011 → C

110 → C

پس این درست است و پیشوند هیچ کدی نیست

کدهای تک بیتی 0 و 1 رو نمی‌توان استفاده کرد به خاطر پیشوند بودن و همین طور از کدهای ۲ بیتی 00 11 10 01 هم نمی‌توان استفاده کرد

چون از پیشوند کدهای دیگر باشند. چون هیچ کدی نباید پیشوند کد دیگری باشد.

• پس از بین کدهای ۳ بیتی باقی مانده اونی که تداخل با دیگران نداشت رو انتخاب کردیم.

حال اگر بخواهیم اجرا هم نباشیم می‌توان به همین صورت الگوریتم را ادامه داد:

اول از کدهای ۳ بیتی شروع می‌کنیم و چک می‌کنیم که Prefix ای از کدهای دیگر نباشند

نتیجه برای کدهای ۳ بیتی اگر چک می‌کنیم اگر یافت نشد ۴ بیتی ها را در نظر می‌گیریم و به همین صورت تا یافتن جواب

ادامه می‌دهیم و جلو می‌رویم. (با این صورت تضمین می‌شود که کمترین تعداد بیت ممکن برای آن حرف پیدا شده بود)

• حال برای حالت ۲ حرف بودن کد دقیقا الگوریتم بالا را تکرار می‌کنیم با این تفاوت که آن حرف اول که frequency آن بیشتر بود را باید اول چک کرد که اگر فضا با بیت‌های کمتر باقی مانده بود آن را به آن اختصاص داده شود و دیگری کد با حجم کمتری گرفته باشد.

سوال (3) برای حل این سوال از شهری شروع می‌کنیم که بیشترین تعداد N_i را دارد و به تعداد N_i+1 تا شمشیر در آن رها می‌کنیم

و صبر می‌کنیم تا تمام اژدهاها نابود شوند. به این صورت همه که در ابتدا N_i+1 شمشیر تمام کند دهانه‌های آن شهر

را نابود می‌کنند چون که تعداد رستاخیز اژدهاهای که نشان تولید نرخ شمشیرهای باشند تا زمانی معین تمام آن‌ها را نابود کرده

و بعد باید به شهر بعدی که N_i آن بیشترین مقدار است برود.

یعنی باید بر اساس N_i آن‌ها شهرها را sort کرده باشیم و بعد هر بار از یک شهر به شهر دیگر برود در راستای نزدیکی بعدی اگر

N_iها حرکت کنیم این الگوریتم optimal خواهد بود چون اگر دقیقاً $max(N_i)+1$ تا شمشیر داشته باشیم

و از بیشترین اژدهاها رها شروع کنیم N_i شروع کنیم چون نرخ سر جدا کردن شمشیرها بیشتر از تولید شدن سر آن‌ها

در هر مرحله ما باشد پس تا بدین هر بار آن اژدهای که هستند نابود می‌شوند.

سر باشد حداقل $max(N_i)+1$ شمشیر اگر از هر شهری شروع کنیم تمام اژدهاها را نابود می‌کنیم چون مقدار تولید سر حداکثر

برابر اژدهاها $max(N_i)$ بود در حالی که N_i+1 شمشیر داشته بودیم پس تمام آن‌ها را می‌توانیم نابود کنیم.

سوال ④ : برای این مسئله حل الگوریتم که مقدار avarge را محاسبه کنیم. بنابراین جمع کل distance ها

تقسیم بر تعداد k . در پایان حداقل بزرگترین بخش را به دست خواهیم رسید:

• به این صورت عمل می‌کنیم: تعداد شهرها

```
def minimize_max_distance(distances, k)        فاصله‌ها.
    total_length = sum(distances)
    avarege_length = total_length / k
    segments = []        بخش‌هایی که شکل یافتند از تعداد فاصله‌ها
    current_length = 0
    for distance in distances:
        if current_length + distance >= avarage_length and len(segments) < k - 1:
            segments.append(current_length + distance)
            current_distance = 0
        else
            current_length += distance
```

یا بزرگ‌تر

اگر برابر شود یا بیشتر از avarge_length یک segment جدید اضافه می‌کنیم.

• در غیر این صورت باید این مقدار اضافه شود.

• پس در این صورت این قدر باید distance ها را با هم جمع زده باشیم تا در پایان از مقدار avarge_distance زیاد‌تر شود.

~~while len(segments)~~

$d_{max} = \max(d_1, \dots d_n)$ — • این مقدار را باید برای کمترین مقدار ممکن برسانیم.

حال آن را اثبات می‌کنیم. برهان خلف: اگر فرض کنیم که این تقسیم‌بندی بهینه نباشد و یک تقسیم‌بندی دیگر بتواند بیشترین طول را کمتر کند فرض ما به تناقض می‌رسد چون:

Avarge_length به عنوان یک شاخص کمک می‌کند که طول هر بخش تا جای ممکن به

① در الگوریتم حریصانه تقسیم شده چون هر بخش باید کم‌تر از آن تعداد باشد یا برابر ← حال اگر در یک تقسیم‌بندی دیگر یک بخش از آن کوتاه‌تر از الگوریتم حریصانه باشد نزدیک باشد

آن مسئله باید در بخش‌های دیگر بلندتر باشد زیرا مجموع طول‌ها ثابت بود. و تعداد k ها نیز ثابت

(آنان جز بخش آخر باید بیشتر avarg_dist مشخص شده چون ... • وقتی مقداری کم کنیم هر کدام از بخش‌ها کمتر از مقدار در پایان مقدار آن max_dis باید زیاد کرده باشد.

سوال ⑤: در این سوال در الگوریتم Greedy باید فایل‌ها را بر اساس ترتیب $\frac{P}{b}$ چیده شده باشد

تا به مقدار بهینه در $\text{Total Cost} = \sum_{i=1}^{n} P_i \, \text{Position}(f_i)$

رسیده باشیم. که Possition (f_i) برابر خواهد بود با فاصله ابتدای memory

تا پایان فایل. حال این را با برهان خلف ثابت می‌کنیم

فرض می‌کنیم که OPT جواب بهینه را به ما بدهد و $\text{Total } C' < \text{Total Cost}$

باید لغو بهینه‌تر از Greedy عمل کرده باشد در این صورت اگر برای دو فایل f_i و f_j

داشته باشیم که $\underbrace{f_i}_{OPT}$ اول آمده باشد و بعد از آن f_j آمده باشد حال فرض می‌کنیم که: $\frac{P_i}{b_i} < \frac{P_j}{b_j}$

بعد قدرا با ... الگوریتم OPT اما در Greedy جای این دو f_j زودتر آمده باشد مطابق با الگوریتم Greedy

در این صورت هزینه‌ی دسترسی به f_j در Greedy کمتر از OPT خواهد بود.

چون $P_i b_j < P_j b_i$ خواهد شد. پس در حالت کلی انتقال f_j که P_j بیشتر بوده و b_j مقدار کمتر

نسبت به f_i داشته و بهینه‌تر در الگوریتم Greedy نزدیک‌تر به memory بوده پس هزینه‌ی آن

نسبت کمتری داشته

در پایان کمتر خواهد شد. پس فرض خلف اشتباه بوده و الگوریتم Greedy بهینه است.

سوال ۶ : راه حل درست می‌باشد

به این صورت که می‌توان هر بار کوچک‌ترین پروانه و کوچک‌ترین گل را با هم ست کرد و بعد آن مسئله به مسئله‌ای کوچک‌تر با $n-k$ عضو تبدیل خواهد شد که به صورت بازگشتی آن را به همین شیوه حل کرد

حال اثبات می‌کنیم که هر بار کوچک‌ترین پروانه با کوچک‌ترین گل جفت شده باشد حتماً جواب ما درست باشد

$B = \{b_1, ..., b_n\}$ لیست پروانه‌های مرتب شده

پس داریم : برهان خلف :

$F = \{f_1, ..., f_n\}$ لیست گل‌های مرتب شده

اگر دارند جفت با هم

الگوریتم ما می‌خواند : b_1 و f_1 را جفت می‌کند

• OG گرید فرض می‌کنیم الگوریتم opt ولی نگیریم که این اختلاف‌ها را کمتر از حد بهینه می‌کند

فرض کنید در جواب مطابق با الگوریتم حد بهینه نشده باشد

$Diff' \leq Diff$: داریم b_1 با f_k جفت شده و f_1 با b_x ای که بیشتر شده

یعنی b_1 با f_1 می‌شد

است که در این حالت پس داریم

$$|b_1 - f_1| \leq |b_x - f_k|$$

• پس بدان گونه بهینه‌ی opt اگر b_1 را با f_1 جفت کنیم و b_x را با f_k جفت کنیم مجموع اختلاف‌ها کمتر خواهد شد به راه حل Greedy

پس می‌توان با تبدیل راه حل opt و تبدیل آن به Greedy رسید

و Diff آن کمتر می‌شد و این خلاف فرض اصلی سوال بود.

پس الگوریتم Greedy بهینه است.

سوال ۷ : برای حل این مسئله به این صورت عمل خواهیم کرد

بهترین حالت : سنگ ها با وزن های نزدیک به هم در یک گروه قرار داده شده باشند
در این حالت گروه بندی به صورتی انجام خواهد شد که

پس به این صورت می باشد که :

```
def Calculate_Sum(weights, k)
    paired_Sum= [waitts(i) +weights(i+1) for i in range (len(weights))]
    Pair-Sums.Sort()
    min_Sum = Sum ( pair-Sums(:k-1)) +weight(0) +weight (-1)

    max_Sum = max um ( pair-sums (-(k-1):] ) + weight(0) + weight(-1)

    return min_Sum, max_Sum
```

• می دانیم که برای اینکه بتوانیم مجموع بهترین امتیاز ها را به دست آوریم که باید تاره ها را به k گروه تقسیم بندی کرد و کمترین اول و آخر عدد ها را باهم جمع زد را برای هر گروه
پس از آنکه دیدادیم بیشترین میزان دوتایی ها را که کنار هم هستند را با عنصر اول و آخر باهم جمع می زنیم
پس اول جفت ها را sort می کنیم و سپس از k-1 تا کامل آن را بررسی می کنیم که اول بیشترین مقادیر بود
که این حالت مجموع کل آن ها به حداکثر مقدار می رسد.
و برای کمترین مقادیر هم آن را نیز به همین صورت منتها کمترین مقادیر را در نظر می گیریم.

• حال که استفاده کردیم از این مسئله با استفاده از بزرگترین جفت ها را پیدا کرده بودیم که فقط از الگوریتم Greedy
خواهد شد چون اولین عنصر و آخرین عنصر آن همیشه بوده اند.