پاسخ تمرین ۶

۱- الف) درست است. چرا که در یک سطر، در هر مرحله ظرفیت کوله‌پشتی یک واحد زیاد می‌شود و تعداد آیتم‌ها نیز ثابت است. در نتیجه بیش‌ترین ارزشی که می‌توانیم در کوله‌پشتی قرار دهیم، نمی‌تواند کاهش یابد. (چون ظرفیتش رو به افزایش است!)

ب) درست است. چرا که هر بار برای به‌روز رسانی dp به شکل زیر عمل می‌کنیم:

$$dp_{i,j} = max(dp_{i-1,j}, dp_{i-1,j-value\ of\ i-th\ item})$$

که به این معنیست که با ثابت بودن یک ستون، اگر سطر به سطر جلو برویم، هر سطر حداقل مقدار سطر قبلی خودش را دارد. چرا که با شیوه‌ای که به‌روز رسانی می‌کنیم نمی‌توان کاری کرد که با ظرفیت ثابت j و امکان انتخاب از $i-1$ شیء اول، ارزشی که وارد چمدان می‌شود از ارزشی که با قابلیت انتخاب i شیء اول وارد چمدان می‌شود بیشتر باشد.

۲-

	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	0	25	25	25	25
2	0	0	20	25	25	45	45
3	0	15	20	35	40	45	60
4	0	15	20	35	40	55	60
5	0	15	20	35	40	55	65

۳- مسأله همان مسأله کوله‌پشتیست. فقط این بار در صورتی که کارت i ام را برداریم، بجای رجوع به dp[i – 1][j – value of card[i]] بایستی به dp[i – 2][j – value of card[i]] رجوع نماییم. چون کارت قبلی حذف می‌شود. پس اگرکارت i را برنداریم :

$$dp_{i,j} = dp_{i-1,j}$$

و در صورتی که کارت i را برداریم، خواهیم داشت:

$$dp_{i,j} = dp_{i-2,j-value\ of\ card_i} + value\ of\ card_i$$

در نتیجه :

$$dp_{i,j} = max(dp_{i-2,j-value\ of\ card_i} + value\ of\ card_i, dp_{i-1,j})$$

پایه‌های dp هم به این شکل پر خواهند شد:

$$dp_{0,j} = 0 \ \forall \ \ 0 \le j \le k$$

$$dp_{i,0} = 0 \ \forall \ \ 0 \le i \le n$$

در ادامه شبه کد این راه حل آورده شده است:

```
dp: int[n + 1][k + 1]
card_values: int[n]

function calc_max_possible_value():
    for j from 0 to k:
        dp[0][j] = 0

    for i from 0 to n:
        dp[i][0] = 0

    for i from 1 to n:
        for j from 0 to k:
            if card_values[i] <= j:
                dp[i][j] = max(dp[i - 1][j], (dp[i - 2][j - card_values[i]] + card_values[i] if i > 1))
            else :
                dp[i][j] = dp[i - 1][j]

    return dp[n][k]
```

مرتبه زمانی الگوریتم: O((n + 1)(k + 1))

۴- برای حل این سؤال ابتدا تمام زیر مجموعه‌های ممکن n نمایش را به دست می‌آوریم که این کار با استفاده از الگوریتم بیت ماسک در $O(2^n \times n)$ ممکن است. الگوریتم بیت ماسک به این صورت است که برای هر زیرمجموعه یک Mask ایجاد می‌کند. برای یک مجموعه با n عضو، Mask زیر مجموعه i ام آن مجموعه برابر با نمایش دودویی i توسط n بیت است. در واقع در Mask(i) اگر بیت k ام برابر 0 باشد، یعنی عضو k ام مجموعه در این زیرمجموعه حضور ندارد و اگر بیت k ام 1 باشد، یعنی عضو k ام در این زیرمجموعه حاضر است. شبه کد الگوریتم بیت ماسک در ادامه آمده است:

```
n := length of set
for mask from 0 to 2^n - 1 : /// (1<<n) = 2^n
    for i from 0 to n-1 :
        if the i'th bit in mask is 1 :
            print set[i]
    go to next line of output
```

حال برای آن که مشخص کنیم آیا می‌توان بدون وقفه L دقیقه نمایش دید یا خیر، تابعی به نام dp تعریف می‌کنیم. که این تابع یک زیر مجموعه از نمایش‌ها را ورودی می‌گیرد و بیشترین زمانی که می‌توان بدون وقفه نمایش دید را خروجی می‌دهد. فرض می‌کنیم مجموعه کل نمایش‌ها S باشد در این صورت اگر داشته باشیم dp(S) > L یعنی سارا می‌تواند L دقیقه بدون وقفه به تماشای نمایش‌ها بپردازد و در غیر این صورت نمی‌تواند چنین کاری انجام دهد.

حالت پایه تابع dp برای مجموعه تهی رخ می‌دهد که مشخصا خروجی تابع 0 خواهد بود. برای به‌روز رسانی تابع dp، فرض کنید s یک مجموعه از نمایش‌ها باشد. حال روی آخرین نمایشی که دیدیم حالت‌بندی می‌کنیم. یعنی اگر آخرین نمایشی که دیده باشیم، l_s باشد، برای هر l_s ∈ s حداکثر تا چه زمانی می‌توانستیم نمایش دیده باشیم. در واقع می‌خواهیم محاسبه نماییم که بعد از زمان dp[s − l_s] ، اگر نمایش l_s را تماشا کنیم، حداکثر تا چه مدت می‌توانیم نمایش ببینیم. برای چنین کاری چک می‌کنیم که آیا در بین بازه‌هایی که نمایش l_s برگزار می‌شود، بازه‌ای هست که زمان شروع آن پیش از dp[s − l_s] باشد یا خیر. اگر چنین بازه‌ای داشتیم بازه‌ای را انتخاب می‌کنیم که زمان شروع دیرتری دارد (دیرتر تمام می‌شود) پس بین تمام حالات l_s ماکسیمم می‌گیریم و پاسخ را برابر با dp[l_s] قرار می‌دهیم. از آنجایی که کلا 2^n تا زیرمجموعه از نمایش‌ها داریم و بیشترین تعداد عضو ممکن برای یک زیرمجموعه، n است و بایستی برای هر کدام از آن‌ها یک جست‌وجوی دودویی روی لیست بازه‌های نمایش انجام دهیم، مرتبه زمانی برابر با $O(2^n \times n \times \log max(c_i))$ خواهد شد.

۵- یک آرایه $n \times k \times m$ به عنوان dp در نظر می‌گیریم. در این صورت $dp_{i,j,t}$ برابر است با هزینه رنگ‌آمیزی در صورتی که i درخت داشته باشیم، رنگ آخرین درخت t باشد و زیبایی درخت‌ها برابر j باشد. پاسخ سؤال کمینه تمام $dp_{i,j,t}$ هاست. در ابتدا تمام خانه‌های dp را برابر با بی‌نهایت قرار می‌دهیم. در ادامه به ازای هر $dp_{i,j,t}$ ای سه حالت خواهیم داشت:

حالت اول: اگر درخت i ام قبلا رنگ شده باشد و رنگش t نباشد، نمی‌شود کاری کرد که رنگ درخت آخر i شود. پس این خانه از آرایه dp همان بی‌نهایت خواهد ماند.

حالت دوم: اگر i امین درخت قبلا رنگ شده باشد و رنگش t باشد، ۲ حالت خواهیم داشت:

- حالت اول: رنگ i − 1 امین درخت هم t باشد. در این صورت هزینه رنگ‌آمیزی برابر با $dp_{i-1,j,t}$ خواهد شد.
- حالت دوم: i − 1 امین درخت رنگ m دارد که $m \neq t$. در این صورت بایستی i − 1 درخت اول زیبایی‌ای برابر با j − 1 داشته باشند. پس کمینه مقدار $dp_{i-1,j-1,m}$ برای تمام مقادیر m خواهد بود.

پس هزینه رنگ‌آمیزی در این حالت به صورت زیر خواهد بود:

$$min(dp_{i-1,j-1,t}, min(dp_{i-1,j-1,m}) \forall m | m \neq t)$$

حالت سوم: اگر i امین درخت رنگ‌آمیزی نشده باشد، تنها تفاوتش با حالت قبل این است که بایستی i امین درخت را با رنگ t رنگ بزنیم. که در این صورت به هزینه محاسبه شده در حالت قبل، p_{ij} افزوده خواهد شد.

۶- شبه کد چنین کاری به شکل زیر است:

```
Generate a spanning tree T of G
Put all remaining edges that are not in T in a list L

For each edge e in L:
    Find the cycle C in the graph that is T plus edge e
    If e is lighter than the heaviest edge e' in C:
        add e to T and remove e' from T
```

برای تولید درخت پوشا به O(n) عملیات نیاز داریم. زمانی که یک یال به درخت اضافه می‌کنیم، به O(n) عملیات برای یافتن یک دور نیاز است. هم‌چنین برای یافتن سنگین‌ترین یال در دور، به O(n) عملیات نیاز است. چون کلا 1 + 8 یال باقی خواهد ماند و از آنجا که 1 + 8 یک عدد ثابت است، پس حلقه for هم به O(n) عملیات نیاز خواهد داشت. پس کلا الگوریتم از مرتبه O(n) است.

۷- مرغ‌ها را بر اساس وزن‌شان مرتب می‌کنیم. متغیری به نام last خواهیم داشت که نشان می‌دهد آخرین مرغی که وزن نهایی‌اش را مشخص کرده‌ایم، چه وزنی داشته است. در ابتدا برای این که متغیر last ، تأثیری نداشته باشد، $last = 0$ قرار می‌دهیم. حال از کم‌ترین وزن مرغ شروع می‌کنیم. هر بار اگر وزن مرغ مورد بررسی w باشد و $last < w - 1$ باشد، مرغ را رژیم می‌دهیم و وزن مرغ $w - 1$ خواهد شد. در صورتی که $last = w - 1$ باشد، مرغ در همین وزن خواهد ماند و در صورتی که $last = w$ باشد، وزن مرغ را یک واحد افزایش می‌دهیم. دقت شود که در انتهای هر تصمیم‌گیری باید متغیر last را به وزن نهایی به‌روز رسانی نماییم. تعداد مرغ‌های با وزن متفاوت برابر تعداد مرغ‌هایی خواهد بود که: $last \neq w$.

دقت شود که در این الگوریتم تنها یک مرتب کردن ویک حرکت کردن روی اعضای آرایه را داشتیم. پس مرتبه زمانی O(n log n) خواهد بود.

۸- دو اشاره‌گر یکی به اولین خانه‌ای که در آن پلیس قرار دارد و یکی به اولین خانه‌ای که در آن دزد قرار دارد در نظر می‌گیریم. اگر دزد فعلی در فاصله کم‌تر از k ازپلیس بود، پلیس دزد را دستگیر می‌کند و اشاره‌گرها را به اولین دزد و پلیس بعدی به‌روز رسانی می‌کنیم. اگر فاصله دزد و پلیس بیشتر از k بود، اشاره‌گر کوچک‌تر را به‌روز رسانی می‌کنیم تا به دزد یا پلیس بعدی اشاره کند و این کار را تکرار می‌کنیم. چون هر یک از اشاره‌گرها یک بار طول آرایه را طی می‌کنند، هزینه زمانی این الگوریتم به شکل سرشکن از مرتبه O(n) خواهد بود.

۹- حل کردن مسأله داده شده معادل با این است که به هر بازه یک رنگ اختصاص دهیم به طوری که هیچ دو بازه‌ای که اشتراک دارند، رنگ یکسانی دریافت نکرده باشند. هر رنگ را با یک عدد بزرگ‌تر از صفر نشان می‌دهیم. این الگوریتم را در نظر بگیرید؛ بازه‌ها را طبق زمان شروع (s_i) در نظر می‌گیریم و به هر بازه کوچک‌ترین رنگی که به هیچ‌کدام از بازه‌هایی که تا الان رنگ شده‌اند و با آن اشتراک دارند، داده نشده را اختصاص می‌دهیم. واضح است که این الگوریتم یک رنگ‌آمیزی صحیح از بازه‌ها ارائه می‌کند (اگر دو بازه اشتراک داشته باشند، رنگ یکسانی نمی‌گیرند.) حال اثبات می‌کنیم این الگوریتم حداقل تعداد رنگ را استفاده می‌کند:

از روی بازه‌های داده شده گراف G را به این شکل می‌سازیم که به ازای هر بازه در G یک رأس قرار می‌دهیم و بین دو رأس یک یال اضافه می‌کنیم، اگر و تنها اگر بازه‌های متناظر آن دو رأس با هم اشتراک داشته باشند. اجرای الگوریتم داده شده روی این گراف را در نظر بگیرید.

ادعا می‌کنیم اگر این الگوریتم به یک رأس، رنگ k را نسبت دهد، آنگاه این گراف یک خوشه با اندازه k دارد. رأسی را در نظر بگیرید که به آن رنگ k داده‌ایم. با توجه به این که بازه‌ها را به ترتیب شروع بررسی می‌کنیم، هر بازه‌ای که رنگ شده باشد و با این بازه اشتراک داشته باشد، حتما با نقطه شروع آن اشتراک دارد، با توجه به این که این بازه رنگ k خورده است، حداقل $k - 1$ بازه با ابتدای آن اشتراک داشته‌اند که به همراه خود این بازه تشکیل یک خوشه با اندازه k می‌دهند. با توجه به این که برای رنگ‌آمیزی رئوس یک گراف حداقل به تعداد اندازه بزرگ‌ترین خوشه آن رنگ نیاز داریم، رنگ‌آمیزی ارائه شده توسط الگوریتم بهینه است. تنها بخشی که از مسأله می‌ماند این است که روشی ارائه دهیم که برای هر بازه کوچک‌ترین رنگی که می‌توان به آن داد را در زمان مورد نیاز پیدا کند. برای انجام چنین کاری ابتدا تمام اعداد ۱ تا n را در یک درخت جست‌وجوی دودویی قرار می‌دهیم. سپس تمام نقاط (شروع و پایان) را به ترتیب در نظر می‌گیریم و یک لیست پیوندی شامل تمام بازه‌هایی که به ابتدای آن‌ها رسیده‌ایم ولی به انتهای‌شان نرسیده‌ایم را نگه می‌داریم. با دیدن هر نقطه، اگر نقطه شروع بود،

کوچک‌ترین عدد داخل درخت را حذف کرده و آن رنگ را به بازه‌ای که دیده‌ایم نسبت می‌دهیم، اگر نقطه پایان بود، آن را از لیست پیوندی حذف کرده و رنگی که به آن داده بودیم را به درخت اضافه می‌کنیم. (هر نقطه شروع و پایان به بازه نظیر خود در لیست پیوندی یک اشاره‌گر دارد.) زمان مورد نیاز برای مرتب‌سازی نقاط O(n log n) است و به ازای هر بازه یک بار به درخت اضافه، یک بار در آن جست‌وجو و یک بار از آن حذف می‌کنیم که زمان مورد نیاز برای جمع این‌ها نیز O(n log n) است.

۱۰- فرضیات:

- مؤلفه x مختصات خانه‌ها داده شده و برای هر خانه یکتاست.
- مؤلفه y مختصات تمام خانه‌ها یکسان است.

گام‌های الگوریتم:

۱- اگر خانه‌ای وجود نداشت، صفر برمی‌گردانیم.

۲- مختصات خانه‌ها را به شکل نانزولی مرتب می‌کنیم.

۳- لیستی با نام towers در نظر می‌گیریم که در آن مختصات هر دکل قرار دارد.

۴- متغیری با نام last tower loc تعریف می‌کنیم که مکان آخرین دکل را نشان می‌دهد. این متغیر در ابتدا مقدار $x[1]+8$ دارد.

۵- last tower loc را به towers اضافه می‌کنیم.

۵- متغیر دیگری با نام towers count تعریف می‌کنیم که این متغیر نشان‌دهنده حداقل تعداد دکل‌های لازم برای امکان ارائه خدمات به تمام خانه‌هاست و در ابتدا مقدار ۱ دارد.

۶- حال از مختصات دومین خانه تا آخرین خانه شروع به پیمایش می‌کنیم:

- اگر $|x[i]-last\ tower\ loc|>8$ باشد، مقدار towers count را یک واحد زیاد می‌کنیم، last tower loc را به x[i] + 8 به روز رسانی می‌کنیم و last tower loc را به towers اضافه می‌کنیم.

۷- در پایان towers count و towers را برمی‌گردانیم.

تحلیل مرتبه زمانی: با توجه به این که یک مرتب‌سازی داریم و یک بار هم روی مختصات خانه‌ها پیمایش می‌کنیم، مرتبه زمانی الگوریتم، O(n log n) خواهد بود.

اثبات درستی الگوریتم: فرض می‌کنیم n خانه داریم. در حالتی که n = 0 باشد، هر دوی الگوریتم بهینه و الگوریتم پیشنهادی، صفر برمی‌گردانند. برای ادامه اثبات فرض می‌کنیم:

$$n \geq 1$$
$$x_1 < x_2 < x_3 < \ldots < x_n$$

حال فرض می‌کنیم جواب بهینه O باشد. در این صورت فرض می‌کنیم جواب بهینه شامل q دکل در مختصات $d_1 < d_2 < d_3 < \ldots < d_q$ است. (دقت شود که در جواب بهینه دو دکل نمی‌توانند در یک مکان باشند، اگر چنین باشد می‌توانیم یکی از دکل‌های تکراری را حذف کنیم که این امر با بهینه بودن O در تضاد است.)

حال فرض کنید $\{t_1,\ t_2,\ t_3,\ \ldots\ ,t_p\}$ خروجی الگوریتم پیشنهادی باشد. در این صورت ویژگی A(s) را به شکل زیر تعریف می‌کنیم:

$$A(s): q \geq s \quad \text{and} \quad d_s \leq t_s$$

ویژگی A(s) تضمین می‌کند که راه حل بهینه شامل حداقل s دکل می‌شود و مکان دکل s که همان d_s است از مکان دکل نظیرش در الگوریتم پیشنهادی (t_s) پایین‌تر نیست.

ادعا: $A(j)\ \ is\ \ true\ \ \forall\ \ j \in \{1, 2, ..., p\}$

قبل از اثبات ادعا اثبات می‌کنیم در صورت درستی ادعا، ادعا نشان می‌دهد که الگوریتم حریصانه جواب بهینه را برمی‌گرداند:

اگر ادعا درست باشد می‌توان گفت $A(p)$ is true پس طبق تعریف ویژگی A ، $q \geq p$. از طرفی به سادگی می‌توان تأیید کرد که مجموعه مختصات دکل‌هایی که الگوریتم حریصانه برمی‌گرداند، به گونه‌ای هستند که تمام خانه‌ها خدمات ارتباطی داشته باشند. (چرا که الگوریتم حریصانه به گونه‌ایست که به هر خانه‌ای که به خدمات دسترسی ندارد، خدمات می‌دهد و یک بار که به خانه‌ای خدمات داده شود، چون مکان دکل‌ها ثابت است، خدمات حذف نمی‌شوند.) حال از آنجا که مجموعه مختصات دکل‌هایی که توسط الگوریتم حریصانه برگردانده می‌شود، راه حلی برای مسأله فراهم کردن خدمات ارتباطی برای تمام خانه‌هاست، پس هیچ راه حل بهینه‌ای نداریم که تعداد دکل بیشتری از این راه حل ارائه دهد. یعنی $q \leq p$. از طرفی پیش‌تر نشان دادیم در صورت درستی ادعا می‌توان گفت $q \geq p$. از این دو رابطه می‌توان نتیجه گرفت در صورت درست بودن ادعا $q=p$ است و این یعنی راه حل حریصانه و راه حل بهینه یکی‌اند.

اثبات ادعا:

- اثبات درستی A(1): در الگوریتم حریصانه همواره $t_1=x_1+8$. حال با برهان خلف اثبات می‌کنیم که A(1) برقرار است. (درست است.) فرض می‌کنیم که A(1) برقرار نباشد، در این صورت یا $q=0$ و یا $d_1>t_1$. از آنجا که فرض کرده‌ایم $n\geq1$ و O یک راه حل بهینه است، پس خانه اول بایستی توسط دکلی پوشش داده شود. بنابراین در جواب بهینه حداقل یک دکل وجود دارد پس $q\neq0$. همچنین اگر $d_1>t_1$ باشد، خانه اول توسط هیچ دکلی پوشش داده نخواهد شد و چنین چیزی ممکن نیست، پس A(1) برقرار است.
- اثبات درستی $A(s)\ \ \forall 1<s\leq p$: این بار از برهان خلف روی تمام حالات ادعا استفاده می‌کنیم. فرض کنید ادعایی که کردیم درست نباشد. هم چنین فرض کنید s کوچک‌ترین اندیسی باشد که به ازای آن A(s) برقرار نیست. از حالت قبلی کاملا مشخص است که $s>1$ پس باید حالتی را بررسی کنیم که $A(s-1)$ برقرار است ولی $A(s)$ برقرار نیست.

  از آنجا که $A(s-1)$ برقرار است، پس خواهیم داشت $q\geq s-1$ و همچنین $d_{s-1}\leq t_{s-1}$. فرض کنید $k(s-1)$ نشان دهنده اندیس k ام حلقه در نقطه‌ای از الگوریتم باشد که دکل t_{s-1} واقع شده است. می‌دانیم دکل t_{s-1} برای این نصب شده است که خانه‌ای پوشش داده نشده در مکان $x_{k(s-1)}$ را پوشش دهد. طبق الگوریتم داریم: $t_{s-1}=x_{k(s-1)}+8$

  از آنجا که $s-1<p$ پس الگوریتم بعد از نصب دکل در مکان t_{s-1} به حلقه زدن روی k ادامه می‌دهد و نهایتا دکل جدیدی را در مکان t_s نصب خواهد کرد. حال اگر $k(s)$ نشان دهنده اندیس k ام حلقه در نقطه‌ای از الگوریتم باشد که دکل t_s واقع شده است، طبق الگوریتم خواهیم داشت: $t_s=x_{k(s)}+8$ و همچنین $k(s)>k(s-1)$. همچنین به ازای $k=k(s)$ در صورتی که داشته باشیم: $|x_{k(s)}-t_{s-1}|>8$ بایستی دکل جدید نصب کنیم. حال نشان می‌دهیم چگونه اطلاعات ذکر شده را ترکیب کنیم تا اثبات کنیم:

  $$x_{k(s)}>t_{s-1}+8 \qquad (1)$$

  $x_{k(s)}$ بایستی در مکانی پایین‌تر از جایی باشد که دکل t_{s-1} بتواند آن را پوشش دهد. برای نشان دادن چنین چیزی ابتدا آن چه در دو بند قبلی آمده است را به شکل زیر خلاصه می‌کنیم:

  $$|x_{k(s)}-t_{s-1}|>8$$
  $$x_{k(s-1)}=t_{s-1}-8$$
  $$x_{k(s)}>x_{k(s-1)}$$

آخرین نامساوی را از مرتب بودن مکان خانه‌ها و این که $k(s)>k(s-1)$ است، می‌توان نتیجه گرفت. در صورتی این سه شرط برای $x_{k(s)}$ برقرارند که شرط (1) برقرار باشد.
همچنین از برقرار بودن $A(s-1)$ می‌توان نتیجه گرفت که $d_{s-1}\le t_{s-1}$. بنابراین از (1) می‌توان نامساوی زیر را برای $j=s-1$ نتیجه گرفت:

$$x_{k(s)}>d_j+8 \quad \forall \quad 1\le j<s \qquad (2)$$

نامساوی (2) بیان می‌کند که خانه در مکان $x_{k(s)}$ تحت پوشش هیچ یک از (s – 1) دکل اول نیست. همچنین باقی حالات در نامساوی (2) از مرتب بودن d_j ها نتیجه می‌شوند.
از آنجا که O یک راه حل است، خانه در مکان $x_{k(s)}$ بایستی تحت پوشش حداقل یک دکل باشد، به همین خاطر حداقل یک دکل دیگر در O بایستی موجود باشد. به عبارت دیگر $q>s-1$. حال با استفاده از این فرضیات که $A(s)$ غلط است و $q>s-1$ می‌توان فهمید که $d_s>t_s$. همچنین با استفاده از مکان دکل t_s داریم: $t_s=x_{k(s)}+8$ در نهایت با استفاده از مرتب بودن d_k ها و این که $d_s>t_s$ خواهیم داشت:

$$d_j>t_s=x_{k(s)}+8 \quad \forall\, s\le j\le q \qquad (3)$$

بنابراین بر اساس روابط (2) و (3) خانه در مکان $x_{k(s)}$ بایستی خارج از پوشش تمام دکل‌های O باشد که این امر با راه حل بودن O در تضاد است و درستی ادعای ذکر شده را اثبات می‌کند.

برای دیدن اثبات با جزئیات بیشتر این الگوریتم می‌توانید به اینجا مراجعه نمایید.

۱۱- از آنجا که سکه‌ها مرتب شده‌اند، کافیست در ابتدا یک متغیر به نام max_payable که نشان‌دهنده بیشترین مبلغ قابل پرداخت است تعریف نماییم. در ادامه روی لیست سکه‌ها پیمایش می‌کنیم و اگر ارزش سکه‌ای از max_payable + 1 بیشتر بود، سکه جدیدی با ارزش max_payable + 1 به لیست سکه‌ها اضافه می‌کنیم و max_payable را هم به max_payable + 1 به‌روز رسانی می‌کنیم. در غیر این صورت max_payable را به ارزش آن سکه به روز رسانی می‌کنیم. کد پایتون این سؤال در ادامه آمده است:

```python
n, k = map(int, input().split())

coins = list(map(int, input().split()))

new_coins_count, i = 0, 0
new_coins = []
max_payable = 0

while max_payable < k and i < n :
    if coins[i] > max_payable + 1 :
        new_coins_count += 1
        n += 1
        coins.insert(i, max_payable + 1)
        new_coins.append(max_payable + 1)
        max_payable += max_payable + 1
    else :
        max_payable += coins[i]
    i += 1

while max_payable < k :
    new_coins_count += 1
    coins.append(max_payable)
    new_coins.append(max_payable + 1)
    max_payable += max_payable + 1

print(new_coins_count)
print(new_coins)
```

مرتبه زمانی الگوریتم: حلقه اول حداکثر n بار اجرا می‌شود و حلقه دوم هم چون در هر مرحله max_payable ، حدودا ۲ برابر می‌شود و تا زمانی که max_payable کمتر از k باشد این روند ادامه دارد، حداکثر log k بار اجرا می‌شود پس الگوریتم از مرتبه O(n + log k) است.

اثبات درستی الگوریتم: فرض می‌کنیم پاسخ الگوریتم حریصانه و الگوریتم بهینه متفاوت است و پاسخ الگوریتم حریصانه به صورت $\{G_1, G_2, \dots G_n\}$ و پاسخ الگوریتم بهینه به صورت $\{O_1, O_2, \dots, O_m\}$ باشد. حال فرض می‌کنیم که اولین جایی که پاسخ حریصانه و پاسخ بهینه با یکدیگر تفاوت دارند، در اندیس i است. در این صورت دو حالت خواهیم داشت:

حالت اول: $O_i < G_i$ و این یعنی $G_i > \text{max\_payable}$ که در این حالت خود max_payable را نمی‌توان ساخت که این تناقض است.

حالت دوم: $O_i \geq G_i$ در این حالت می‌توانیم در راه حریصانه تا مقدار $\text{max\_payable} + G_i - 1$ و در راه بهینه می‌توانیم تا مقدار $\text{max\_payable} + O_i - 1$ را بسازیم. که دومین مقدار واضحا از مقدار اول بزرگ‌تر است. حال در صورتی که G_I و O_i را عوض نماییم، راه بهینه همچنان درست باقی می‌ماند ولی نسبت به حالت قبلی کم‌تر شده است که این با بهینه بودنش در تضاد است.