به نام خداوند بخشایشگر مهربان

تمرین اول الگوریتم پیشرفته محمد تنهایی ۸۷۲۰۲۴۴۶ نرم افزار

۱. ابتدا ثابت می کنیم که MC متعلق به Np است . اگر یک جواب برای مسئله مانند <G,T,δ> داشته باشیم مـی تـوانیم در زمان چند جمله ای جواب را چک کنیم . به این صورت که ابتدا چک می کنیم که مجموع یال هـایی کـه در T قـرار دارد هزینه کمتر از δ دارد . این کار با جمع کردن کلیه یال های موجود در T قابل انجام است . حـداکثر تعـداد یالهـای موجود در T برابر E است ، پس چک کردن مجموع یالها حداکثر از درجه O(e) است . حال باید ثابت کنیم کـه از راس های s به راس های t مسیری وجود ندارد . از الگوریتم Floid استفاده کرده و طول کل مسیر هـای گـراف را بـه دسـت می آوریم . این الگوریتم در زمان $O(n^3)$ اجرا می شود . حال باید از هر راس به راس دیگر که در T قرار دارد مسیر بی نهایت باشد . چک کردن این درایه ها در آرایه ای که الگوریتم فلوید ایجاد کرده در زمان O(n) قابل انجام است . بنا بـر این Certification در زمان $O(n^3 + n + e)$ قابل انجام است . بنا بر این MC متعلق به ان پی است.

حال یک reduction از VC به MC معرفی می کنیم . فرض کنیم که VC را در گراف G محاسبه کـرده ایـم . نشـان می دهیم که می توان با تبدیل (چند جمله ای) MC را در گراف G' که شیوه ساخت آن را بیان خواهیم کرد پیدا کرد. فرض کنیم گراف G را داریم . در گراف G' کلیه راس های G را عیناً نگهداشته و راس جدیدی مانند X اضافه می کنیم . راس X را به کلیه رئوس قبلی متصل می کنیم (وزن این یالها را برابر یک قرار می دهـیم) . مجموعـه T را در گـراف G' برابر مجموعه یال ها در گراف G قرار می دهیم ، همچنین δ را برابر k می گیرم. . این reduce در زمان چند جمله ای و به سادگی قابل انجام است $VC \leq_p MC$. حال نشان می دهیم که حل مسئله VC در گراف <G,E,k> معـادل بـا حل مسئله MC در گراف <G',T,δ> است . فرض کنیم که VC را در گراف G به دست آورده ایم . اگر یالهای متصل به رئوس مجموهه VC را در گراف G' را حذف کنیم ، یک MC در گراف G' یافته ایم . برای نمونه به شکل زیر نگاه کنیم:

در این مثال در گراف G رئوس ۲ و ۳ ، رئوس VC هستند که تعداد آنها از یک k مفروض کمتر است . می خـواهیم یـک MC با حداکثر هزینه δ با مجموعه رئوس { (1,2) ، (2,3) و (3,4) }=T را در گراف G' پیدا کنـیم . در ایـن حالـت اگر یالهای متصل به ۲ راسی که در VC یافته ایم را در گراف G' حذف کنیم . یک MC با مجموعه رئوس { (1,2) ، (2,3) و (3,4) }=Tداریم که حداکثر هزینه آن ها کمتر از مقدار δ (که همان k است) ، می باشد .

اگر برای یک مسئله تصمیم گیریVC به صورت <G,E,k> جـواب yes داشـته باشـیم . آنگـاه مسـئله MC نیـز دارای جواب yes خواهد بود ، زیرا δ=k و اگر یالهای متصل به رئوسی که در مجموعه راس های جواب VC است را در گـراف متناظر G' حذف کنیم یک MC بین رئوسی داریم که در گراف G دو سر یال هستند! (**به عبارت راحت تر VC یک سـری راس را به عنوان جواب انتخاب می کند ، ما تمام یال هایی که در گراف G' به این رئوس وصل هستند را حذف می کنیم . حـال در این گراف یک MC داریم که مجموعه T آن برابر است با راس های دو سر هر یال در G**)

در حالت برعکس اگر برای یک MC به صورت <G',T,δ> جواب yes داشته باشیم ، در گراف G متناظر با آن یـک VC داریم که تعداد رئوس آن کمتر از δ است (k= δ) و همه رئوس بـه آن متصـل هسـتند.(یعنـی جـواب <G,E,k> yes است) در این حالت رئوس VC برابر اند با رئوس دو سر یالهای مجموعه T منهای x .

گراف G' که ما ساخته ایم یک درخت است . به همین خاطر MC در داخل درخت نیز Np کامل است .

---

۲. ابتدا ثابت می کنیم HALF-CLIQUE ان پی است . برای این منظور یک الگوریتم برای Certificate کـردن معرفـی می کنیم . چک کردن HALF-CLIQUE شامل چک کردن تعداد راس ها که باید بیشتر از نصف باشد و همچنین چک کردن این که گراف حاصل یک گراف کامل باشد ، باید چک کنیم که هر راس به همه راس های موجـود در جـواب یـک یال داشته باشد ، این کار به راحتی در زمان چند جمله ای قابل انجام است O(n)، حال باید اثبـات کنـیم کـه HALF-CLIQUE ان پی سخت است برای این منظـور یـک reduction از CLIQUE بـه HALF-CLIQUE معرفـی مـی کنیم :

a. با ورودی <G(V,E),k > کارهای زیر را انجام می دهیم
b. یک گراف جدید به نام G'(V',E') معرفی می کنیم که شامل همه راس ها و یال های گراف G است.
c. اگر |V|<2k باشد به اندازه |V|-2k راس به گراف G' اضافه می کنیم . این راس ها بـه هـیچ راسـی متصـل نمی کنیم. و ایزوله هستند.
d. اگر |V|>2k به اندازه |V|-2k راس به گراف جدید اضافه می کنیم . راس های جدید را به همه رئوس قبلـی موجود در G' متصل می کنیم .
e. حال G' گراف جدیدی است که ساخته ایم.

باید نشان دهیم که گراف G' در زمان چند جمله ای ساخته می شود . افزودن رئوس به سادگی قابل انجام است . همچنـین اتصـال این رئوس در بخش d به رئوس قبلی در زمان چند جمله ای قابل انجام است. $Clqiue \leq_p HalfClqiue$

حال اثبات می کنیم که این reduction درست است .

در حالتی که |V|<2k باشد. اگر گراف G دارای یک CLIQUE به اندازه k باشـد در ایـن صـورت G' دارای یـک گراف کامل به اندازه k است (زیرا رئوس اضافی به هیچ راسی متصل نیستند پـس در گـراف کامـل G' هـم حضـور

نخواهند داشت) در اینجا $V'=2k-|V|+|V|$ به عبارت بهتر $k=|V'|/2$. در جهت برعکس اگر G' دارای یک گراف کامل به اندازه $|V'|/2$ باشد در این صورت گراف G نیز دارای یک گراف به اندازه k است. زیرا به وضوح مشخص است که هیچ کدام از رئوس اضافه شده در این گراف حضور ندارند.

در حالتی که $|V|>=2k$ باشد . اگر G دارای یک گراف کامل به اندازه k باشد ، G' دارای یک گراف به اندازه $V' = k+|V|-2k$ است . این عبارت برابر است با $|V'|/2$ (زیرا $|V'|= 2|V|-2k$) . در حالت برعکس اگر G' دارای یک گراف کامل باشد ، این گراف کامل شامل همه رئوس اضافه شده به G' است (زیرا این رئوس به همه رئوس قبلی متصل شده اند) . اگر این رئوس را از گراف کامل کم کنیم . $|V|-k-(|V|-2k) = k$ راس خواهیم داشت . به عبارت بهتر G دارای CLIQUE است.

با توجه به reduction انجام شده و با توجه به قضایا ، HALF-CLIQUE ان پی کامل است.

.۳

a. برای اینکه نشان دهیم مسئله Np است ابتدا یک الگوریتم Certificate ارائه می دهیم . فرض کنیم که برای یک عبارت مفروض که تعداد تکرار هر متغیر بیشتر از 3 نیست یک جواب داریم . به سادگی با جاگذاری مقدار هر متغیر در clause های مربوط به آن می توانیم تشخیص دهیم که جواب مسئله صحیح است یا نه . این کار به سادگی در زمان چند جمله ای قابل انجام است.

مرحله بعد انجام reduction از یک مسئله است که np-hard بودن آن برای ما مشخص است ، می باشد . برای این منظور از مسئله صدق پذیری استفاده می کنیم . ما مسئله sat را به مسئله حاضر که آنرا r-k-sat می نامیم کاهش می دهیم . (r-k-sat = restricted k sat)

فرض کنیم عبارت p را داریم . این عبارت هیچ محدودیتی در تعداد متغیر ها ندارد . الگوریتمی ارائه می دهیم که هر عبارت p را با هر تعداد تکرار به یک عبارت با حداکثر ۳ تکرار برای هر متغیر تبدیل کند.

I. یکی از متغیر هایی که تعداد تکرار آن بیشتر از ۳ است را می گیریم به عنوان مثال متغیر A ، اگر چنین متغیری وجود ندارد به مرحله آخر می رویم.

II. متغیر جدیدی تعریف کرده (مثلاً Y) و در همه عبارات به جز ۲ عبارت ، متغیر قبلی (یعنی A) را با متغیر جدید جایگزین می کنیم .

III. عبارت $(A \leftrightarrow Y)$ را به مجموعه clause های این sat اضافه می کنیم.(and می کنیم)

IV. به مرحله I می رویم .

V. پایان. حال یک عبارت منطقی داریم که حداکثر تعداد تکرار هر متغیر در آن ۳ است .

به عنوان نمونه عبارت زیر را به شکل محدود به ۳ تکرار در می آوریم :

$$(A \vee B) \wedge (A \vee C \vee D \vee B) \wedge (A \vee D) \wedge (A \vee B')$$
$$(A \vee B) \wedge (A \vee C \vee D \vee B) \wedge (Y \vee D) \wedge (Y \vee B') \wedge (Y \leftrightarrow A)$$

ممکن است تکرار متغیر های جدیدی که به عبارت منطقی اضافه می شود بیشتر از ۳ باشد ، که با بازگشت الگوریتم به مرحله اول تشخیص داده می شوند . اگر ورودی الگوریتم ما عبارتی مانند m باشد ، این الگوریتم در زمان حداکثر m متوقف می شود . زیرا پس از تعویض با متغیر جدید ، سایر تکرار های باقی مانده از آن متغیر قبلی (در اینجا A) دیگر جایگزین نمی شوند . بدترین حالت زمانی است که در یک عبارت به طول m تنها ۲ متغیر وجود داشته باشد که تکرار می شوند . در این حالت تعداد متغیرهای جدید اضافه شده برابر است با m-1 مثلاً اگر ۸ تکرار از A داشته باشیم ۷ متغیر جدید به مجموعه فوق (و به طبع آن ۷ عبارت جدید) اضافه می شود. پس در بدترین حالت با ورودی m دارای پیچیدگی O(m) هستیم . یعنی الگوریتم چند جمله ای است. $Sat \leq_p RKSat$

اگر با یک ورودی به طول m جواب مسئله تصمیم گیری Sat ، yes باشد ،در عبارت RKSat متناظر می توانیم به جای متغیر های Y که به صورت $(Y \leftrightarrow A)$ است ، مقدار A را جایگزین کنیم . و Clause $(Y \leftrightarrow A)$ را حذف کنیم با این کار که در زمان چند جمله ای قابل انجام است یک عبارت عیناً مانند عبارت sat می سازیم بنابراین اگر مسئله تصمیم گیری sat جواب yes داشته باشد مسئله تصمیم گیری RKSat هم جواب yes می دهد.

در حالت بر عکس اگر یک جواب برای مسئله RKSat داشته باشیم ، بدون در نظر گرفتن متغیر های اضافه شده یک جواب برای مسئله sat داریم . به عبارت بهتر اگر RKSat به مسئله تصمیم گیری جواب yes بدهد ، Sat نیز جواب yes می دهد.

بنابر این ، این مسئله NP-Complete است.

b. برای اثبات چند جمله ای بودن این مسئله راهی جز طراحی یک الگوریتم که این مسئله را در زمان چند جمله ای حل کند نداریم . راه حلی که ما ارائه می کنیم مقداری متفاوت با راه حل مستقیم مسئله یعنی جایگزینی هر یک از متغیر ها است .
فرض کنیم که یک لیست از متغیر ها داریم .
برای سادگی یک الگوریتم سه مرحله ای ارائه می دهیم :

مرحله اول (حذف clause) :

با توجه به اینکه حداکثر ۲ تکرار از هر متغیر داریم کل حالاتی که متغیر نمونه A می تواند داشته باشد به صورت آمدن A و A ، آمدن A و A' ، آمدن A' و A' و یا یک بار تکرار به صورت A یا A' است . با توجه به ماهیت مسئله در صورتی که ما یک بار تکرار داشته باشیم می توانیم clause مربوطه را با ست کردن همان متغیر ارضا کنیم و clause از مجموعه clause ها حذف شود .همچنین با تکرار مشابه (مانند A و A یا A' و A') نیز می توانیم چنین کاری را انجام دهیم و در این حالت دو clause حذف می شود .

این کار با $O(n^2)$ که n تعداد متغیر هاست قابل انجام است . زیرا در این الگوریتم هر متغیر را بررسی می کنیم و حذف clause صورت می گیرد . البته ممکن است که حذف clause باعث ایجاد متغیر هایی با یک بار تکرار شود (و یا Clause هایی با یک متغیر شود) که باید دوباره بررسی متغیر ها انجام شود . (با دو حلقه For این بررسی قابل انجام است)

مرحله دوم (ارضا متغیر ها) :

با توجه به شکل عبارت می توانیم از متغیر اول شروع کنیم وClause های حاوی آن متغیر را با ادغام کنیم . به عنوان مثال عبارت $(A \vee B)\wedge(\bar{A} \vee C)$ معادل عبارت $(B \vee C)$ است. با این کار متغیر نمونه A حذف شده است و با عبارتی که حاوی یک Clause کمتر است ادامه می دهیم . در پایان کار یک Clause باقی می ماند که حاوی چند متغیر است که با هم or شده اند ، به سادگی می توان این عبارت را ارضا کرد.این مرحله از الگوریتم $O(m)$ قابل انجام است. که در این جا m تعداد Clause هاست . بدیهی است که $m<2n$ زیرا حداکثر تعداد Clause ها برابر تعداد متغیرها ضرب در تعداد تکرار آنهاست.

بدیهی است که در صورت رسیدن به تناقض عبارت قابل ارضا نبوده است . تناقض ما به صورت عبارتی مانند AA' نمایان می شود که قابل ارضا نیست.

و در صورت ارضا شدن همه Clause ها الگوریتم نشان داده است که عبارت قابل ارضا است.

چون که تعداد تکرار متغیرها محدود است ، تعداد Clause ها نیز محدود می شود . حداکثر تعداد clause ها برابر دو برابر تعداد متغیر هاست (این بد بینانه ترین حالت است) .

پیچیدگی الگوریتم برابر است با $O(m+n^2)$ که m تعداد Clause هاست در حالت کلی پیچیدگی برابر است با $O(n^2)$.

---

۴. در ۱۷.۱ بودن یا ۱-۱۷ بودن شک کردم ، هر دو سوال رو حل کردم!

۱۷-۱

a)

```
int A[n];
For(int i=0; i<n; i++)
{
      A[i] = revs(A[i])
}

int revs(int A)
{
int B;
int k = log n;
For( int i = 0; i<k; i++)
      {
       B[k-i] = A[i];
```

```
        }
Return B;
}
```

B)

```
Bit-Reveresed-Inc(int * A)
{
bool t= true;
int counter = 0;
while(t)
        {
                t = shift_left(A);
                counter++;
        }
shift_right(A,1); **
while(counter>0)
        {
                shift_right(A,0);
        }
}
```

حال باید نشان دهیم که این دو الگوریتم (A و B) درست عمل می کنند .

در الگوریتم قسمت اول ما یک عمل را k بار انجام می دهیم بنابراین در بهترین حالت و در بدترین حالت O(k) را خواهیم داشت. چون n بار این عمل را روی اعضای آرایه انجام می دهیم پیچیدگی آن برای کل آرایه برابر o(nk) خواهد بود.

تحلیل پیچیدگی الگوریتم دوم از راه amortized analysis قابل انجام است . برای این منظور از تحلیل حسابداری استفاده می کنیم . برای هر عمل ما از ۴ واحد استفاده می کنیم . یک واحد برای شیفت به چپ استفاده می شود . یک واحد برای شیفت به راست ، ۲ واحد را هم روی بیتی که در ** به راست شیفت شده است می گذاریم . برای هر بار عمل Inc بیت ها به چپ شیفت پیدا می کنند ، اگر این بیت ۱ باشد شیفت ادامه پیدا می کند . از طرفی هر بیتی که ۱ است قبلاً ۲ واحد روی آن گذاشته شده است . بنابراین با دو عمل شیفت چپ و راست اعتبار خود را مصرف می کند . بدیهی است که اگر بیتی که شیفت به چپ می خورد صفر باشد الگوریتم با هزینه ۲ آنرا ۱ کرده و ۲ واحد هم روی آن می گذارد . بنا بر این

هزینه n عمل Inc برابر است با O(4n) که از درجه n است.

C) بله ، الگوریتمی که ما در قسمت B توضیح دادیم دقیقاً با همین فرض نوشته شده است . و اثبات آن نیز ارائه شده است (یک تیر و دو نشان!)

۱۷.۱-۱ بله ، همچنان پیچیدگی برابر O(1) باقی می ماند . Multipush(S,T,k) به صورت زیر تعریف می شود .

Multipush(S,T,k)

```
while T not empty and i<k
        do push(T[i]);
        i++;
```

این عمل k آیتم از T به پشته push می کند . دلیل O(1) بودن Multipush این واقعیت است که هر آیتم حداکثر یک بار push می شود و حداکثر هم یک بار pop می شود . بنابراین اگر یک دنباله از push ، pop ، Multipush و Multipop داشته باشیم چون که هر آیتم حداکثر یک بار push یا pop می شود بنابراین پیچیدگی برابر O(n) است از طرفی تعداد n عمل انجام داده ایم . بنابراین پیچیدگی برابر O(n)/n خواهیم داشت .

۲-۱۷.۱ در بدترین حالت هر عمل INCREMENT می تواند پیچیدگی O(k) داشته باشد ، هر عمل DECREMENT نیز می تواند حداکثر پیچیدگی O(k) داشته باشد . می توانیم counter را مجبور کنیم که روی مقداری مانند $2^k - 1$ مدام عمل DECREMENT و سپس INCREMENT انجام دهد . در این حالت هر بار k بیت صفر یک می شوند و k بیت یک صفر می شوند و این بدترین اتفاق ممکن است . یعنی پیچیدگی می تواند در O(nk) باشد.

۳-۱۷.۱ ابتدا مسئله را تعریف می کنیم :

فرض کنیم که مقدار هزینه هر مرحله را $Cost_i$ بنامیم . یک فرمول دو بخشی تعریف می کنیم :

$$Cost_i = \begin{cases} i & i \text{ is an exact power of } 2 \\ 1 & \text{other} \end{cases}$$

حال این تابع را از ۱ تا n جمع می زنیم :

$$\sum_{i=1}^{n} Cost_i = n + \sum_{i=1}^{\lfloor log\ n \rfloor} 2^i - \lfloor log\ n \rfloor = n + 2^{\lfloor log\ n \rfloor + 1} - \lfloor log\ n \rfloor \leq n + 2n - 1 < 3n$$

هزینه کل را بر تعداد قدم های الگوریتم که n است تقسیم می کنیم به عبارت 3n/n می رسیم که برابر است با ۳ بنابراین هزینه سرشکن الگوریتم برابر است با O(3)

۳۴.۵-۳

ابتدا ثابت می کنیم که ILP ان پی است . فرض کنیم که یک بردار n تایی به نام x جواب مسئله باشد . حال می توانیم چک کنیم که ضرب بردار اولیه A که m*n است در این بردار کمتر از بردار m تایی b است . این کار در زمان چند جمله ای قابل انجام است ، زیرا می توانیم ضرب بردار m*n در یک بردار n تایی را در زمان O(m*n) انجام دهیم . همچنین چک کردن کوچک تر بودن حاصل این ضرب ماتریس از B در زمان m قابل انجام است بنابراین می توان در زمان O(m*n+m) جواب ها را Certificate کرد. بنا بر این مسئله np است.

حال یک الگوریتم برای reduction ارائه می دهیم که این کار را در زمان چند جمله ای انجام دهد. می دانیم که مسئله 0-1 ILP از درجه np-hard است . از روی مسئله 0-1 ILP یک مسئله ILP می سازیم . به این صورت که تعداد m سطر به ماتریس A اضافه می کنیم . این سطر ها را به صورت گراف همانی صفر و یک مقدار می دهیم ، حال به گراف B تعداد m ستون اضافه می کنیم و به همه ستون های اضافه شده مقدار یک می دهیم . می توانیم نشان دهیم که حل 0-1 ILP در حالت اول معادل حل ILP در حالت افزوده (حالتی که به ماتریس ها سطر و ستون اضافه کردیم) است . $01\ ILP \leq_p ILP$

ثابت می کنیم که این reduction در زمان چند جمله ای قابل انجام است . تعداد سطرها و ستون هایی که ما به ماتریس A اضافه کرده ایم m*n است . همچنین تعداد ستون های اضافه شده به ماتریس B برابر m است . می دانیم چنین کاری به راحتی در زمان O(m*n + m) قابل انجام است .

برای اثبات ابتدا فرض کنیم که 0-1 ILP زیر را داریم :

$$\begin{bmatrix} a_1 & a_2 \\ a_3 & a_4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \leq \begin{bmatrix} b_1 & b_2 \end{bmatrix}$$

ماتریس افزوده را می سازیم :

$$\begin{bmatrix} a_1 & a_2 \\ a_3 & a_4 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \leq \begin{bmatrix} b_1 & b_2 & 1 & 1 \end{bmatrix}$$

اگر مسئله تصمیم گیری 0-1 ILP جواب yesبدهد ، ILP نیز جواب yes می دهد ، زیرا حل مسئله 0-1 ILP در ماتریس اولیه ، یک حل برای مسئله ILP در ماتریس افزوده است . در ماتریس افزوده باید x1*1+x2*0<=1 و x1*0+x2*1 <=1 باشد یعنی X1<=1 و X2<=1 به عبارت بهتر حل مسئله 0-1 ILP دقیقاً یک حل برای مسئله ILP است . (در اینجا شرایطی فراهم کرده ایم که k نتواند از 1 بیشتر شود)

به صورت برعکس اگر مسئله تصمیم گیری ILP جواب yes بدهد مسئله 0-1 ILP نیز جواب yes می دهد ، چون کـه دو شرط X1<=1 و X2<=1 را به صورت ضمنی در حل داشته ایم ، بنابراین جواب حاضر ، یک جـواب بـرای مسـئله 0-1 ILP نیز هست .

برای سادگی اثبات را برای حالت m=2 ارائه دادیم ولی در حالت کلی نیز چنین اثباتی برقرار هستند . دلیل آن نیز کاملاً مشخص و روشن است ، زیرا برای هر متغیر شرایطی فراهم شده است که حداکثر می تواند صفر یا یک باشد .

۴-۳۴

a. مسئله تصمیم گیری به این صورت است که : یک سری کار $A = \{a_i\}_{i=1}^{n}$ با زمان پردازش $T = \{t_i\}_{i=1}^{n}$ و با منفعت $P = \{p_i\}_{i=1}^{n}$ و یک سری deadline به صورت $D = \{d_i\}_{i=1}^{n}$ داریم این ۴ مجموعه را به همراه k بـه صورت زیر نمایش می دهیم

<A,T,P,D,k>: آیا می توان با A و T و P و D داده شده ، یک منفعت به دست آورد که بیشتر از k مفروضی باشد.

b. به راحتی می توان نشان داد که SCHEDULING ان پی اسـت . certificate مـی توانـد در زمـان O(n) انجام شود . می توانیم کل p_i ها را در جواب مفروض جمع کرده و چک کنیم که از مقدار k کمتر اسـت یـا نـه بنا براین SCHEDULING متعلق به ان پی است.

حال نشان می دهیم که scheduling ان پی تمام است . برای ایـن منظـور یـک reduction از knapsack به scheduling معرفی می کنیم.این کاهش در زمان چند جمله ای قابل انجـام اسـت (ثابـت مـی کنـیم)

$$\text{KNAPSACK} \leq_p \text{SCHEDULING}$$

فرض کنیم مسئله knapsack را به صورت زیر داریم :

$$< W = \{w_i\}_{i=1}^{n}, V = \{v_i\}_{i=1}^{n}, C, T >$$

حال از روی این مسئله ، یـک مسـئله Scheduling مـی سـازیم ، تعریـف مـی کنـیم : $A = \{a_i\}_{i=1}^{n}$ و $T = \{w_i\}_{i=1}^{n}$ و $P = \{v_i\}_{i=1}^{n}$ و $D = \{C\}_{i=1}^{n}$ و k را هم مقدار T می گیریم . نتیجه این کار خروجـی زیر است <A, T, P,D, k> این کاهش در زمان چند جمله ای قابل انجام است . پیچیدگی این کار بـه انـدازه کپی چند آرایه است و چند جمله ای است.

فرض کنیم که < W, V,C, T> یک جواب yes در مسئله knapsack باشد . بنابراین مجموع همه وزن ها در جواب هستند حداکثر برابر اسـت بـا C .بـا کـاهش انجـام شـده مجمـوع کلیـه آیـتم هـایی کـه در لیسـت

schedule قرار دارد حداکثر برابر است با C . که معنای آن این است که هر وظیفه قبل از مهلت مقرر انجام شده است .همچنین کمترین میزان وزن آیتم های knapsack برابر است با T که با کاهشی که انجام شده است مجموع بهره ای که می بریم حداکثر برابر k است.

فرض کنیم که < A, T, P,D, k > یک جواب yes از مسئله Scheduling باشد . بر این اساس هر آیتمی که در این جواب Scheduling است در درون جواب Knapsack است زیرا هر وظیفه قبل از مهلت مقرر آن انجام شده است.

c. همه کارها را بر اساس مهلت تحویل مرتب می کنیم . فرض کنیم که P[i,j] سود ماکزیمم از کار i تا j باشد . یک فرمول برای محاسبه سود ماکزیمم به صورت پویا به صورت زیر است :

$$P[i,j] = \begin{cases} 0 & if\ i\ or\ j = 0 \\ P[i-1,j] & if\ t_i > j\ or\ d_i < j \\ \max\left(P[i-1,j], P[i-1,j-t_i] + p_i\right) & other \end{cases}$$

از آنجا که t_i ها همه بین ۱ تا n هستند ، بنا بر این اگر جوابی برای این مسئله وجود داشته باشد مسئله در زمان n^2 خاتمه می یابد . هدف ما به دست آوردن P[n,T] است که T کمترین مقدار از بین مقادیر d*n و (n*(n+2))/2 است . دلیل آن نیز مشخص است . زیرا مقدار T هرگز از مقدار d*n بیشتر نمی شود و از طرفی ممکن است که نیازی هم نباشد که تا این مقدار T را بزرگ بگیریم ، می دانیم که حداکثر تکرار P[i,j] برابر است با (n*(n+2))/2 این مقدار نیز با حل معادله بازگشتی قابل حل است .

پیچیدگی الگوریتم برابر است با $O(n^3)$ قبلاً این مسئله را به همین شیوه برای کوله پشتی حل کرده ایم .

d. برای مسئله بهینه سازی می توانیم یک آرایه به شیوه زیر معرفی کنیم : (علاوه بر محاسبه p[i,j] این آرایه را نیز محاسبه می کنیم)

$$A[i,j]=\begin{cases} i & if\ i\ ,j > 0\ and\ P[i-1,j] \le P[i-1,j-t_i] + p_i) \\ i-1 & if\ i\ ,j > 0\ and\ P[i-1,j] > P[i-1,j-t_i] + p_i) \\ 0 & if\ i\ or\ j = 0 \end{cases}$$

حال این آرایه شیوه انتخاب کارها را در خود نگهداری می کند . شبیه کاری که قبلاً برای مسئله کوله پشتی و غیره می کردیم می توانیم از روی این آرایه یک خروجی بهینه که جواب مسئله است را تولید کنیم .

هر درایه این ماتریس مانند j و i نشان می دهد که آخرین کاری که از زمان بندی i تا j باید انجام شود کدام است.

آخرین کاری که باید انجام شود A[i,T] است بر این اساس بقیه کارها را به صورت بازگشتی می سازیم . کار بعدی $A[i,T-t_{a[n,T]}]$ است .

پیچیدگی این الگوریتم مانند حالت C است.