فصل پنجم:

روش حریصانه

فصل پنجم
روش حریصانه

مفهوم روش حریصانه

نام این روش از شخصیت معروف اسکروج گرفته شده است. اسکروج به جای آنکه به گذشته و آینده فکر کند، تنها انگیزه هر روز او به دست آوردن طلای بیشتر بود. الگوریتم حریصانه (Greedy) نیز مانند شیوه اسکروج می‌باشد.

الگوریتم‌های حریصانه یا آزمند شبیه روش‌های پویا اغلب برای حل مسائل بهینه‌سازی استفاده می‌شوند. در شیوه حریصانه در هر مرحله عنصری که بر مبنای معیاری معین «بهترین» به نظر می‌رسد، بدون توجه به انتخاب‌هایی که قبلاً انجام شده یا در آینده انجام خواهد شد، انتخاب می‌شود. الگوریتم‌های حریصانه اغلب راه‌حل‌های ساده‌ای هستند. در روش حریصانه بر خلاف روش پویا، مسأله به نمونه‌های کوچک‌تر تقسیم نمی‌شود.

با یک مثال ساده مفهوم این روش را شرح می‌دهیم :

مثال ۱ : خریداری از یک فروشگاه یک جنس ٦٤ تومانی می‌خرد و ۱۰۰ تومان به فروشنده می‌دهد و فروشنده باید ۳٦ تومان به او برگرداند. اگر فروشنده سکه‌های ۱، ۵ ، ۱۰، ۲۵ و ۵۰ تومانی (از هر کدام حداقل یک نمونه) داشته باشد چگونه می‌تواند بقیه پول خریدار را برگرداند به نحوی که تعداد سکه‌ها (در کل) کمترین مقدار ممکن باشد؟

یک راه‌حل حریصانه به این صورت است : در ابتدا هیچ سکه در مجموعه جواب نداریم. از بین سکه‌های موجود بزرگترین ممکن یعنی ۲۵ تومانی را انتخاب می‌کنیم. این مرحله از الگوریتم حریصانه را روال انتخاب (selection procedure) گویند. اگر یک سکه ۲۵ تومانی دیگر را انتخاب کنیم حاصل از ۳٦ تومان بیشتر شده، لذا آن را کنار گذاشته به سراغ سکه ۱۰ تومانی می‌رویم. حال بررسی می‌کنیم اگر این سکه ۱۰ تومانی را به مجموعه انتخابی قبلی اضافه کنیم حاصل از ۳٦ تومان بیشتر می‌شود یا خیر. این مرحله را تحقیق عملی بودن (feasibility check) می‌نامند. حال اگر این ۱۰ تومان را به ۲۵ تومان اضافه کنیم جمع مجموعه انتخاب شده ۳۵ می‌شود که هنوز به ۳٦ نرسیده است. این مرحله را تحقیق حل شدن (Solution check) می‌گوئیم. در ادامه سکه‌های دیگر را به ترتیب مقایسه می‌کنیم و در نهایت با انتخاب سکه یک تومانی در کل با ۳ سکه (۲۵ تومانی و ۱۰ تومانی و یک تومانی) ۳٦ تومان به دست می‌آید و این حداقل تعداد سکه ممکن است. توجه کنید در الگوریتم فوق ملاک انتخاب، برای انتخاب بهترین سکه در هر مرحله (بهینه محلی) ارزش سکه است و در

هنگام انتخاب سکه در هر مرحله به انتخاب‌های قبلی و بعدی کاری نداریم. در این شیوه اجازه فکر کردن درباره یک انتخاب انجام شده را نداریم یعنی هنگامی که سکه‌ای پذیرفته شد به طور دائم جزو حل به حساب می‌آید و هنگامی هم که سکه‌ای رد می‌شود به طور دائم از حل کنار گذاشته می‌شود.

همان‌طور که مشاهده کردید این روش بسیار ساده است ولی اصلی‌ترین نکته آن است که آیا این روش الزاماً به یک حل بهینه می‌رسد؟ در رابطه با مسأله خاص فوق می‌توان اثبات کرد که جواب بهینه است ولی با مثال زیر نشان می‌دهیم که ممکن است اینگونه نباشد.

مثال ۲ : فرض کنید قرار است به فردی ۱٦ تومان پس دهیم. سکه‌های موجود ۲۵، ۱۲، ۱۰، ۵ و ۱ تومانی است (با اینکه ما در ایران سکه ۱۲ تومانی نداریم ولی این فرض را بکنید که داریم).

با الگوریتم حریصانه فوق به این نتیجه می‌رسیم که باید به آن فرد یک سکه ۱۲ تومانی و ٤ سکه یک تومانی بدهیم یعنی جمعاً ٥ سکه. در حالی که حل بهینه مسأله فوق یک سکه ۱۰ تومانی، یک سکه ٥ تومانی و یک سکه یک تومانی است یعنی در کل ۳ سکه.

از مثال فوق نتیجه می‌گیریم که هر الگوریتم حریصانه الزاماً حل بهینه را نمی‌دهد و برای هر مسأله خاص باید اثبات کنیم که آیا الگوریتم حریصانه برای آن، جواب بهینه می‌دهد یا خیر و این موضوع اغلب سخت‌ترین مرحله کار است.

با توجه به مثال‌های فوق می‌توان مراحل روش حریصانه را به این صورت بیان کرد :

کار با یک مجموعه تهی شروع شده و به ترتیبی خاص عناصری به مجموعه اضافه می‌شوند. هر دور تکرار الگوریتم شامل بخش‌های زیر است :

۱ـ **روال انتخاب :** این روال عنصر بعدی را طبق یک معیار حریصانه انتخاب می‌کند. این انتخاب یک شرط بهینه را در همان برهه برآورده می‌سازد.

۲ـ **تحقیق عملی بودن :** در این مرحله مشخص می‌شود که آیا مجموعه جدید به دست آمده، برای رسیدن به حل عملی است یا خیر.

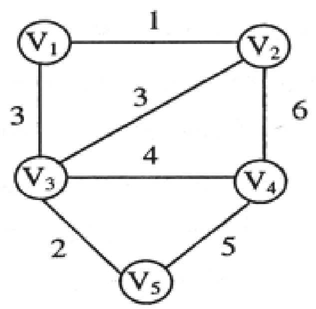
۳ـ **تحقیق حل :** مشخص می‌سازد که آیا مجموعه جدید، نمونه مورد نظر را حل کرده است یا خیر. در ادامه مثال‌هایی را بیان می‌کنیم که با این روش حل می‌شوند. در آخر نیز مثالی را می‌آوریم که با روش حریصانه قابل حل نبوده و باید روش پویا برای حل آن استفاده شود و بدین‌ترتیب روش پویا و حریصانه را با یکدیگر مقایسه می‌کنیم تا دریابیم در هر مورد بهتر است از کدام روش استفاده گردد.

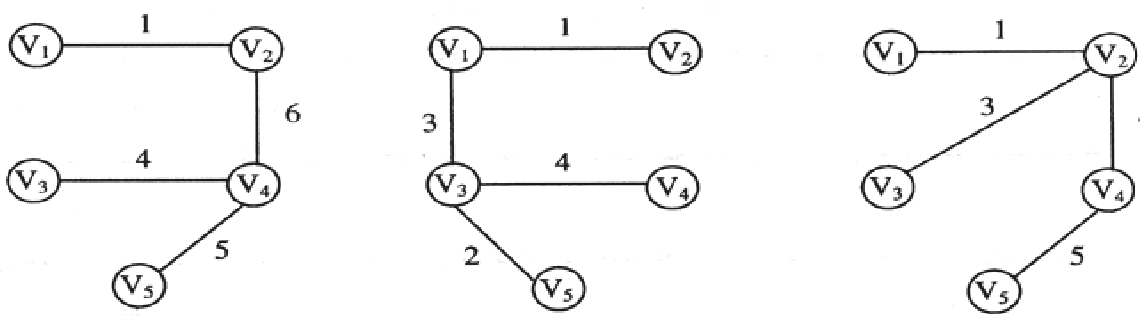
مثال اول : الگوریتم پریم (Prim)

در درس ساختمان داده‌ها خوانده‌ایم که درخت یک گراف بدون جهت، متصل و بی‌چرخه است. به عبارتی دیگر درخت، یک گراف بدون جهت است که در آن بین هر جفت از رئوس فقط و فقط یک مسیر وجود دارد.

توجه کنید در این تعریف هیچ گره‌ای را به عنوان ریشه درنظر نمی‌گیریم ولی در **درخت ریشه‌دار** (rooted tree) یکی از رأس‌ها ریشه می‌باشد و اغلب منظور از درخت، درخت ریشه‌دار است. البته در این قسمت فرض می‌کنیم با درخت‌هایی سر و کار داریم که ریشه آنها برای ما مهم نیست.

درخت پوشا (Spanning tree) برای یک گراف G، یک زیرگراف متصل می‌باشد که حاوی همه رأس‌های گراف G بوده و همچنین یک درخت است. به عبارت دیگر درخت پوشا، شامل همه رئوس و برخی یال‌های گراف است به نحوی که متصل بوده و چرخه نیز ندارد. مثلاً برای گراف بدون جهت وزن‌دار زیر، ۳ درخت پوشا را در پایین آن رسم کرده‌ایم :





(الف) (ب) (ج)

همان‌طور که مشاهده می‌کنید یک گراف ممکن است چندین درخت پوشا داشته باشد. گراف فوق درخت‌های پوشای بیشتری از آنچه در بالا رسم کرده‌ایم دارد. وزن کلی درخت‌های فوق عبارتند از :

الف) $1 + 4 + 5 + 6 = 16$

ب) $1 + 2 + 3 + 4 = 10$

ج) $1 + 3 + 5 + 6 = 15$

به درخت (ب) که حداقل وزن ممکن را بین درخت‌های پوشا دارد، درخت پوشای کمینه گوئیم و هدف ما د این قسمت به دست آوردن این درخت کمینه است. البته یک گراف می‌تواند بیش از یک درخت پوشای کمینه داشته باشد. مثال فوق این‌گونه است و شما یک درخت پوشای کمینه دیگر برای آن رسم کنید. اگر هم درختهای پوشا را در نظر گرفته و سپس کمینه آنها را بیابیم، این الگوریتم از حالت نمایی بوده و لذا باید روش مؤثرتری پیدا کنیم.

یکی از کاربردهای این مسأله هنگامی است که وزارت راه و ترابری بخواهد بین چند شهر جاده‌هایی را بکشد به نحوی که از هر شهر به شهر دیگر مسیری (مستقیم یا غیر مستقیم) وجود داشته باشد و جمع طول این جاده‌ها کمینه باشد.

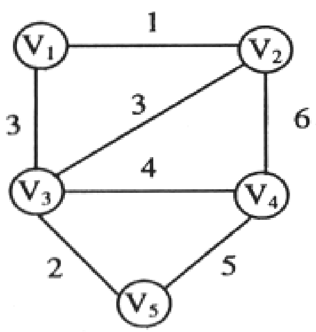
همان‌طور که می‌دانید یک گراف بدون جهت را می‌توان به صورت دو مجموعه V و E نشان داد : $G = (V , E)$ که V مجموعه‌ی رئوس و E مجموعه‌ی یال‌ها می‌باشند. مثلاً برای گراف مثال قبلی داریم :

$V = \{v_1, v_2, v_3, v_4, v_5\}$
$E = \{(v_1 , v_2) , (v_1, v_3), (v_2, v_3) , (v_2, v_4), (v_3, v_4), (v_3, v_5) , (v_4, v_5)\}$

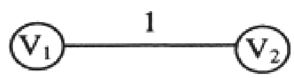
در گراف بدون جهت (v_1 , v_2) با (v_2 , v_1) یکسان است ولی قرار داد می‌کنیم که هنگام نوشتن ابتدا رأسی که اندیس کوچکتر دارد بیاوریم. درخت پوشای T برای گراف $G = (V , E)$ شامل همه رئوس V می‌باشد ولی برخی از یال‌های E را دارد که آنها را با F نمایش می‌دهیم. پس :

$T = (V , F) \quad , \quad F \subseteq E$

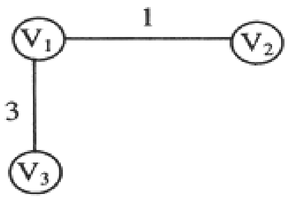
در الگوریتم پریم (prim) ابتدا مجموعه‌ی یال‌های F را برابر تهی و مجموعه‌ی رئوس Y را نیز برابر یک رأس دلخواه (اغلب v_1) قرار می‌دهیم. سپس نزدیکترین رأس به Y را از مجموعه $V - Y$ انتخاب می‌کنیم، بدیهی است که این رأس توسط یالی با کمترین وزن به رأسی از مجموعه Y متصل است. مثلاً در گراف زیر، v_2 انتخاب کرده که کمترین یال (برابر ۱) را دارد. بدین ترتیب v_2 را به Y و (v_1 , v_2) را به F اضافه می‌کنیم:



$F = \{(v_1, v_2)\} , Y = \{v_1 , v_2\}$

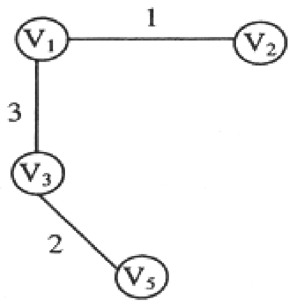


در قدم بعدی v_3 را به مجموعه Y و (v_1 , v_3) را به مجموعه F اضافه می‌کنیم :

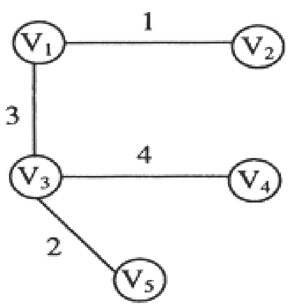


$$F = \{(v_1, v_2), (v_1, v_3)\} \ , \ Y = \{v_1, v_2, v_3\}$$

این عملیات افزودن نزدیکترین رئوس را آن قدر تکرار می‌کنیم تا Y = V شود، البته باید عدم ایجاد چرخه نیز بررسی می‌گردد :



در مرحله فوق فاصله $V - Y = \{v_4, v_5\}$ را با گره‌های موجود در $Y = \{v_1, v_2, v_3\}$ مقایسه کردیم و دیدیم که فاصله v_5 با گره v_3 از همه کمتر است لذا v_5 را به Y و (v_3 , v_5) را بــه F اضافه کردیم. در مرحله آخر فاصله v_4 را با گره‌های موجود در $Y = \{v_1, v_2, v_3, v_5\}$ مقایسه کرده و گره v_4 را به Y و $(v_3 \ , v_4)$ را بــه F اضافه می کنیم و درخت پوشای کمینه به دست می‌آید :



الگوریتم فوق را به صورت زیر می‌توان بیان کرد :

```
F = φ
y = {v1} ;
while  (the instance is not solved)
{
   select a vertex in V - Y that is nearest to Y;   // selection procedure and feasibility check
   add the vertex to Y;
   add the edge to F;
   if  (Y == V)  the instance is solved;   //solution check
}
```

البته در هنگام انتخاب رأس از V - Y باید دقت کرد که حلقه ایجاد نشود.

الگوریتم فوق را به سادگی به کمک ماتریس همجواری W می‌توان پیاده‌سازی کرد که به عنوان تمرین بر عهده دانشجویان می‌گذاریم. لیست برنامه مذکور را می‌توانید در کتاب نیپولیتان مشاهده کنید.

تذکر : الگوریتم پریم را به این صورت نیز می‌توان بیان کرد : ابتدا گره‌ای به دلخواه انتخاب می‌شود و سپس بین یال‌های متصل به آن، یالی با کمترین وزن انتخاب می‌شود به گونه‌ای که حلقه ایجاد نکند. در هر مرحله یالی انتخاب می‌شود که حتماً یکی از دو سر آن جزو مسیر جواب بوده و وزن حداقل داشته باشد. پس الگوریتم پریم دو محدودیت در هر مرحله داریم یکی آن که جنگل ایجاد نشود و دوم آنکه حلقه پدید نیاید.

نکته : در یک گراف کامل K_n با n رأس به تعداد n^{n-2} درخت پوشا، وجود دارد.

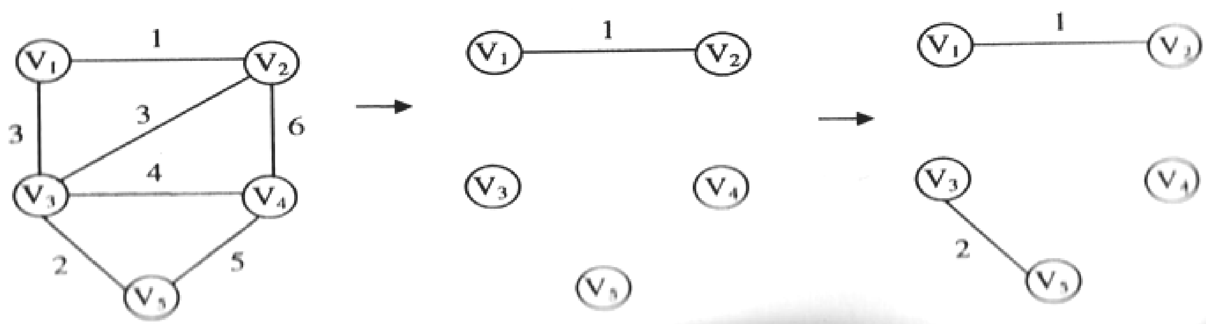
از آنجا که در الگوریتم پریم در هر مرحله فاصله هر گره با گره‌های قبلی مقایسه می‌شود پس بدیهی است که مرتبه $\theta(n^2)$ می‌باشد که n تعداد رئوس گراف است :

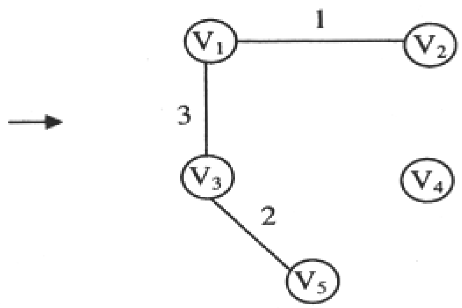
$\theta(n^2)$ = مرتبه اجرای الگوریتم پر

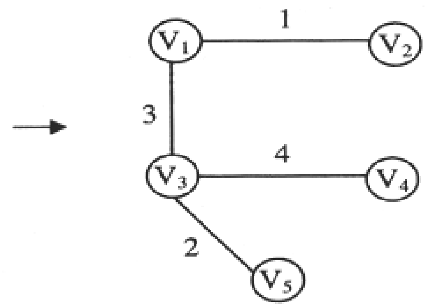
هر چند که الگوریتم‌های حریصانه اغلب ساده‌تر از الگوریتم‌های پویا هستند ولی معمولاً مشخص کردن اینکه آیا این الگوریتم حریصانه همواره جواب بهینه را می‌دهد دشوار بوده و نیاز به اثبات رسمی دارد. در کتاب نیپولیتان قضیه‌ای اثبات شده است که نشان می‌دهد الگوریتم پریم همواره یک درخت پوشای کمینه را تولید می‌کند.

مثال دوم : الگوریتم کروسکال (Kruskal)

این الگوریتم نیز مشابه الگوریتم پریم برای یافتن درخت پوشای کمینه‌ی یک گراف به کار می‌رود. در این الگوریتم ابتدا یال‌ها از کمترین وزن به بیشترین وزن مرتب می‌گردند سپس یال‌ها به ترتیب انتخاب شده و اگر یالی ایجاد حلقه کند، کنار گذاشته می‌شود. عملیات هنگامی خاتمه می‌یابد که تمام رأس‌ها به هم وصل شوند یا اینکه تعداد یال‌های موجود در F برابر n - 1 شود که n تعداد رأس‌ها است. در برخی کتاب‌ها این روش با نام راشال مطرح شده است. شکل زیر مراحل کار را برای یک گراف فرضی نشان می‌دهد.







بیشتر زمان در الگوریتم کروسکال مربوط به مرتب‌سازی یال‌هاست. پس اگر تعداد یال e باشد زمان این الگوریتم از مرتبه $\theta(e\lg e)$ خواهد بود.

ممکن است به نظر برسد که این زمان بستگی به n (تعداد رئوس) ندارد. ولی همان‌طور که می‌دانید در بهترین حالت تعداد یال‌ها برابر n - 1 و در بدترین حالت که گراف کامل باشد و بین هر دو رأس یک مسیر مستقیم داشته باشیم تعداد یال‌ها برابر $\frac{n(n-1)}{2}$ خواهد بود، یعنی :

$$n-1 \le e \le \frac{n(n-1)}{2}$$

پس اگر e نزدیک به کران پائین باشد یعنی گراف نسبتاً خلوت بوده و یال کمی داشته باشد: الگوریتم کروسکال از مرتبه روبه‌رو است :

$$\theta(e\lg e)=\theta(n\lg n)$$

و اگر e نزدیک به کران بالا باشد، یعنی گراف نسبتاً پر باشد و یال‌های زیادی داشته باشد :

$$\theta(e\lg e)=\theta(n^2\lg n^2)=\theta(n^2.2\lg n)=\theta(n^2\lg n)$$

خلاصه آنکه :

پیچیدگی الگوریتم پریم $=\theta(n^2)$

پیچیدگی الگوریتم کروسکال $=\theta(e\lg e)=\begin{cases}\theta(n\lg n) & \text{برای گراف خلوت}\\ \theta(n^2\lg n) & \text{برای گراف شلوغ}\end{cases}$

پس اگر گرافی یال‌های کمی دارد بهتر است از روش کروسکال و اگر یال‌های زیادی دارد بهتر است از روش پریم استفاده کنیم.

تذکر : می‌توان اثبات کرد که الگوریتم کروسکال همواره یک درخت پوشای کمینه ایجاد می‌کند.

مثال سوم : الگوریتم دایجکسترا (Dijkstra)

در فصل قبلی الگوریتم فلوید، جهت تعیین کوتاه‌ترین مسیر از هر رأس به همه رئوس دیگر در یک گراف موزون جهت‌دار را شرح دادیم و دیدیم که آن الگوریتم از مرتبه $\theta(n^3)$ بود. ولی اگر بخواهیم تنها کوتاه‌ترین مسیر از یک رأس به بقیه رئوس دیگر را بیابیم می‌توان از روش حریصانه دایجکسترا استفاده کرد که از مرتبه $\theta(n^2)$ می‌باشد. در ابتدا فرض می‌کنیم که از رأس مورد نظر به سایر رئوس دیگر، حتماً مسیری وجود دارد و اگر اینگونه نباشد الگوریتم به قدری اصلاح نیاز دارد.

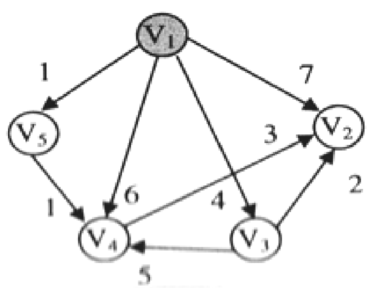
الگوریتم دایجکسترا که به نام کوتاه‌ترین مسیر تک منبع (Single Source shortest path) نیز معروف است مشابه الگوریتم پریم می‌باشد. در ابتدا مقدار مجموعه Y را برابر رأسی قرار می‌دهیم که می‌خواهیم کوتاه‌ترین مسیرهای آن را تعیین کنیم (مثلاً v_1)و مقدار مجموعه F (مجموعه یال‌ها) را نیز تهی می‌دهیم. ابتدا نزدیکترین رأس به v_1 (مثلاً v) را انتخاب می‌کنیم و آن را به مجموعه Y و یال $<v_1 , v>$ را به F اضافه می‌کنیم.

بدیهی است که $<v_1, v>$ کوتاه‌ترین مسیر بین v_1 و v می‌باشد. سپس مسیرهایی از v_1 به رئوس موجود در V-Y را بررسی می‌کنیم که فقط رئوس موجود در Y را به عنوان رئوس واسطه مجاز می‌دارند، کوتاه‌ترین این مسیرها را یافته و رأسی که در انتهای این مسیر است را به Y و یال شامل آن رأس (در این کوتاه‌ترین مسیر) را به F اضافه می‌کنیم. این عمل را آن قدر تکرار می‌کنیم که مجموعه Y برابر V (یعنی همه رئوس گراف اولیه) شود. بدین ترتیب F جواب بوده و شامل یال‌های موجود در کوتاه‌ترین مسیر است. بنابراین الگوریتم دایجکسترا را در سطح بالا، می‌توان به صورت زیر بیان کرد :

```
Y = {v1};
F = φ
while  (the instance  is not solved)  {
       select  a vertex  v from  V - Y ,  that  has  a // selection procedure
       shortest  path from  v1,  using only vertices  // and feasibility  check
       in Y as intermediates;
      add   the new vertex    v to Y;
      add  the edge  (on the shortest  path)    that touches   v to F,
      if (Y == V)
              the  instance  is solved;   //solution   check
}
```

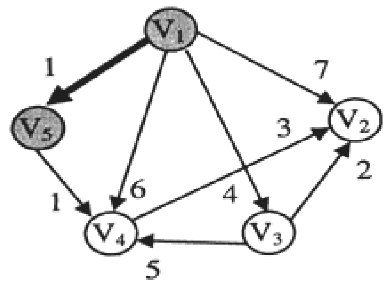
شکل زیر مراحل الگوریتم فوق را برای یک گراف فرضی نشان می‌دهد :

۱ـ محاسبه کوتاه‌ترین مسیر از v_1 :



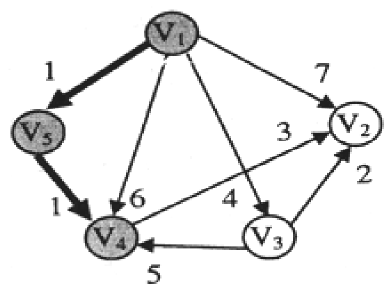
$Y = \{v_1\}$;
$F = \phi$;

۲ـ ابتدا رأس v_5 را انتخاب می‌کنیم چرا که نزدیکترین رأس به v_1 می‌باشد و بدین دلیل v_5 را به Y و $<v_1,v_5>$ را به F اضافه می‌کنیم :



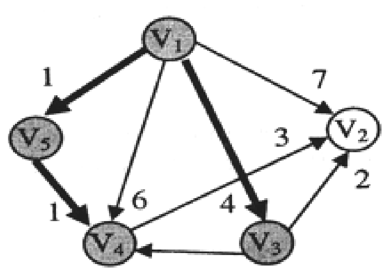
$Y = \{v_1, v_5\}$;
$F = \{<v_1,v_5>\}$;

۳ـ حال با در نظر گرفتن گره v_5 به عنوان واسطه گره v_4 را انتخاب می‌کنیم چرا که v_4 کوتاه‌ترین مسیر تا v_1 را (به کمک گره v_5) دارد.



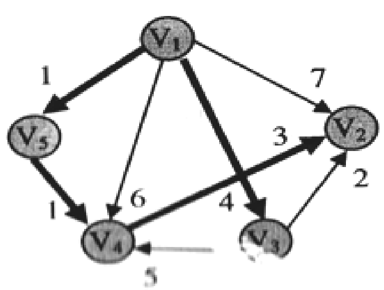
$Y = \{v_1, v_5, v_4\}$;
$F = \{<v_1,v_5>, <v_5, v_4>\}$;

۴ـ رأس v_3 را انتخاب می‌کنیم چرا که با واسطه‌گری $\{v_4, v_5\}$ کوتاه‌ترین مسیر از v_1 را دارد.



$Y = \{v_1, v_5, v_4, v_3\}$;
$F = \{<v_1,v_5> , <v_5 , v_4> , <v_1, v_3>\}$;

۵ـ در آخر نیز v_2 را انتخاب می‌کنیم و می‌بینیم که کوتاه‌ترین مسیر از v_1 به v_2 مسیر $[v_1,v_5,v_4, v_2]$ است. لـذا v_2 را به Y و آخرین یال مسیر مذکور یعنی $<v_4, v_2>$ را به F اضافه می‌کنیم.



$Y = \{v_1, v_5, v_4, v_3, v_2\}$;
$F = \{<v_1,v_5> , <v_5 , v_4> , <v_1, v_3> , <v_4, v_2>\}$;

فصل پنجم : روش حریصانه

از آنجا که در الگوریتم فوق در هر بار، فاصله هر گره بـا گره‌هـای قبلـی مقایسـه می‌شـود، الگوریتـم از $\theta(n^2)$ می‌باشد، که n تعداد رئوس گراف است :

$\boxed{\theta(n^2) \; = }$ پیچیدگی الگوریتم دایجـ

می‌توان اثبات کرد که الگوریتم داجکسترا، همواره کوتاه‌ترین مسیر را می‌دهد، توجه کنید که این موضوع نبوده و باید اثبات شود که در اینجا از آن صرف‌نظر می‌کنیم.

توجه کنید الگوریتم پریم برای موردی کاربرد دارد که مثلاً وزارت راه بخواهد بین چند شهر جـاده بکشـ نظر دارد طول آسفالت ریزی حداقل بوده و از هر شهری بتوان به شهر دیگر رفت ولی در الگوریتم دایجـ هدف آن است که مثلاً می‌خواهیم فقط از تهران به یک سری شهر سفر کنیم و می‌خواهیم طـول مسیرهـ این مسافرت‌ها حداقل باشد تا در مصرف بنزین صرفه‌جویی گردد.

تذکر ۱ : الگوریتم دایجکسترا برای گراف بدون جهت نیز قابل استفاده است.

تذکر ۲ : الگوریتم دایجکسترا برای گرافی درست کار می‌کند که یال منفی نداشته باشد.

تذکر ۳ : الگوریتم دایجکسترا را می‌توان با هرم (Heap) یا هرم فیبوناچی پیاده‌سازی کرد. پیاده‌سازی هرم به $\theta(e \lg n)$ و پیاده‌سازی هرم فیبوناچی آن به $\theta(e + n \lg n)$ زمان نیاز دارد که n تعـداد رئـوس و e یال‌ها است.

مثال چهارم : الگوریتم هافمن

فرض کنید می‌خواهیم n عنصر اطلاعاتی $A_1, A_2 \; ... \; A_n$ را بـه وسـیله رشـته‌هایی از بیت‌ها بـه صـورت درآوریم. یک راه ساده برای این منظور آن است که هر عنصر را به وسیله یک رشته r بیتی کدگذاری کنیـ در آن : $\boxed{2^{r-1} < n \le 2^r}$

در اینحالت طول کد همه عناصر با هم یکسان است.

مثال : برای کدگذاری 48 کاراکتر حداقل به چند بیت نیاز داریم؟

$$2^5 < 48 \le 2^6 = 64 \Rightarrow r = 6$$

پس حداقل به ٦ بیت نیاز داریم.

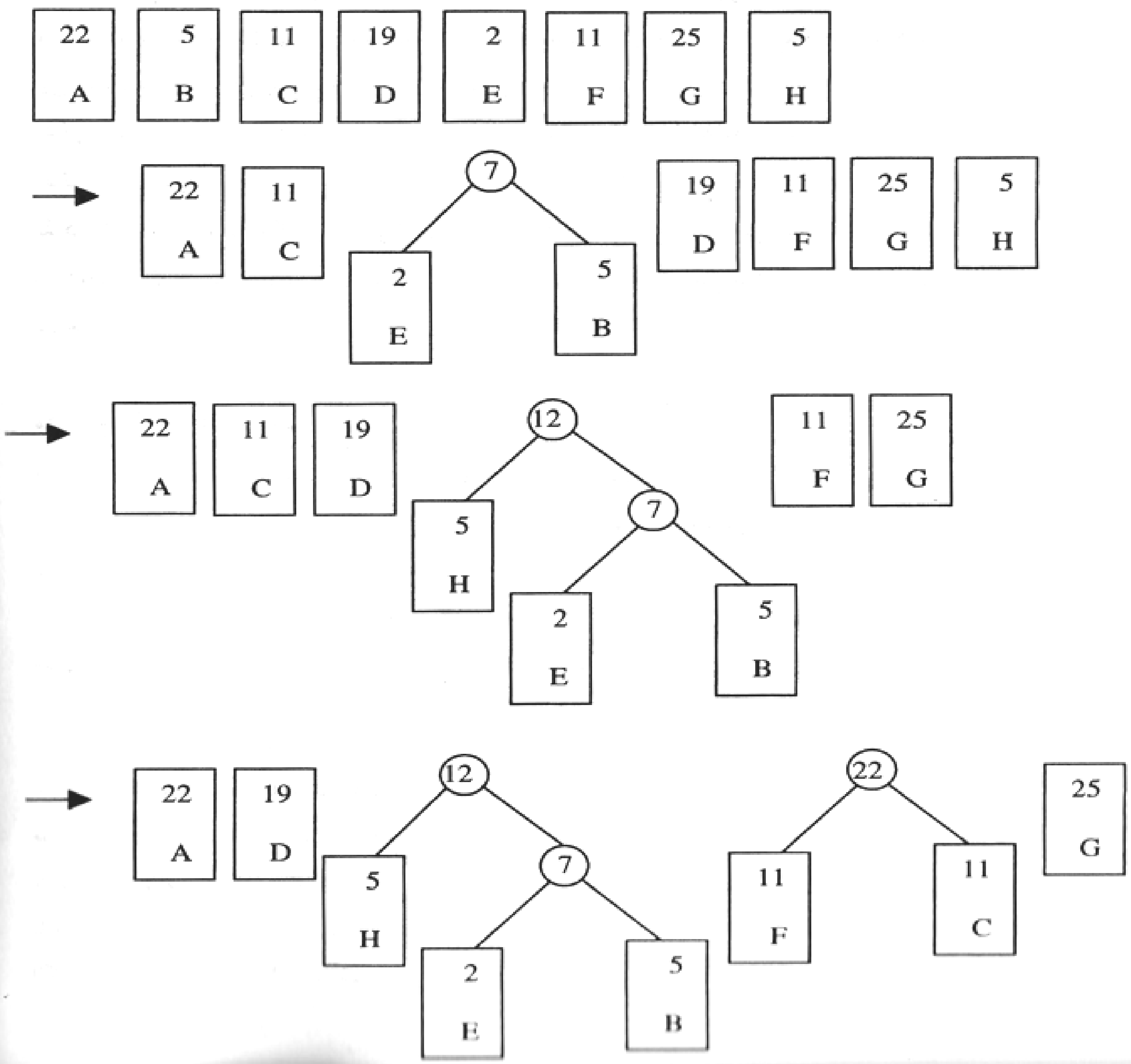
حال فرض کنید عنصرهای اطلاعاتی با احتمال مساوی اتفاق نمی‌افتند آنگاه می‌توان با اسـتفاده از رشـته‌ها طول متغیر، در مصرف حافظه صرفه‌جوئی کرد. به این ترتیب که عناصری که اغلب ظاهر می‌شوند با رشـ کوتاهتر و عناصری که به ندرت ظاهر می‌شوند با رشته‌های بزرگتر نشان داده شوند. برای پیدا کردن این با طول متغیر می‌توان از الگوریتم هافمن استفاده کرد.

الگوریتم هافمن را با مثال زیر شرح می‌دهیم.

مثال: فرض کنید A, B, C, D, E, F, G, H ، ۸ عنصر اطلاعاتی با وزنهای مشخص شده هستند :

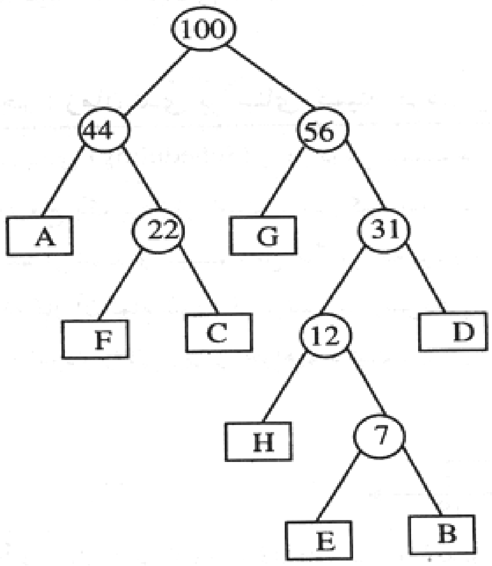
عنصر اطلاعاتی	A	B	C	D	E	F	G	H
وزن	22	5	11	19	2	11	25	5

می‌خواهیم درخت با حداقل طول مسیر وزن داده شده را با استفاده از اطلاعات بالا و الگوریتم هافمن ترسیم کنیم. شکل زیر مراحل کار را نشان می‌دهد. در هر مرحله دو درختی که ریشه مینیمم دارند را با هم ترکیب می‌کنیم (وزن آنها را با هم جمع می‌کنیم). گره با وزن کمتر سمت چپ قرار می‌گیرد :



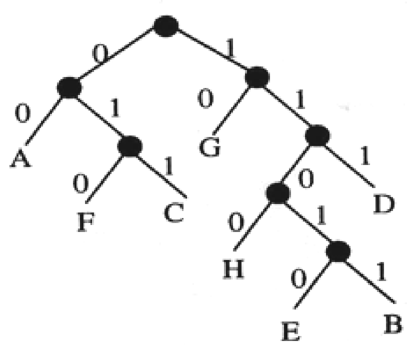
اگر همینطور الگوریتم را ادامه دهید شکل نهائی به صورت زیر می‌شود :

درخت هافمن ⇐



همانطور که مشاهده کردید درخت هافمن از پائین به بالا (از برگ به ریشه) ساخته می‌شود.

مثال : ۸ عنصر اطلاعاتی H, G, F, E, D, C, B, A مثال قبلی را در نظر بگیرید. فرض کنید وزنها نمایش درصد احتمالاتی باشند که عنصرها ظاهر می‌شوند. آنگاه درخت این عناصر با حداقل طول مسیر وزن داده شده را با الگوریتم هافمن می‌سازیم و سپس به شاخه‌های سمت چپ بیت 0 و به شاخه‌های سمت راست بیت 1 را نسبت می‌دهیم.



حال برای پیدا کردن کد هر عنصر از ریشه تا آن عنصر حرکت کرده و بیتهای مشاهده شده را می‌نویسیم :

A = 00 B = 11011 C=011 D=111

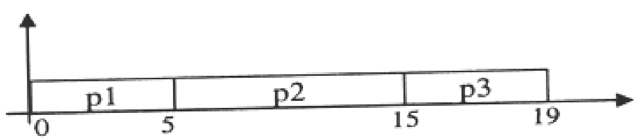
E = 11010 F = 010 G = 10 H = 1100

همانطور که مشاهده می‌کنید G که احتمال ظهور 25% دارد با رشته دو بیتی ولی B که احتمال ظهور 5% دارد با رشته ۵ بیتی نشان داده شده است.

تذکر : یک ویژگی دیگر کد هافمن آن است که نیازی به جداکننده ندارد یعنی کد هیچ حرفی به عنــوان بخـش ابتدایی کد هیچ حرف دیگری نیست، به این کدها Pefix - Free Code (رمز پیشوند آزاد) می‌گویند.

مثال پنجم : زمان‌بندی بر مبنای کمینه کردن زمان کل

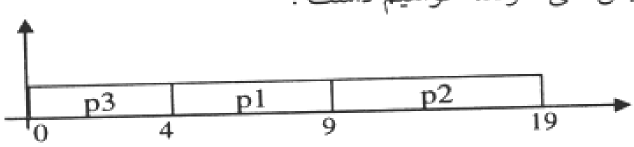
مسأله زمان‌بندی (Scheduling) را در درس سیستم‌عامل خوانده‌اید. منظور از زمان برگشت فاصله‌ی زمانی بین ورود تا خروج یک پردازش است. فرض کنید سه پردازش با زمان‌های $p1 = 5$ ، $p2 = 10$ و $p3 = 4$ همزمـان وارد سیستم می‌شوند و سیستم‌عامل هنگامی که پردازنده را به پردازشی می‌دهد، نمی‌تواند آن را پس بگـیـرد تـا هنگامی که کار آن تمام شود. اگر ترتیب سرویس‌دهی به این برنامه‌ها به ترتیب از چپ به راست [p1,p2,p3] باشد آنگاه نمودار زمانی زیر را خواهیم داشت :



یعنی مثلاً در زمان 5 پردازنده به p2 داده شده و p2 در زمان 15 تمام می‌شود. در این حال زمان کل در سیستم یا به عبارت دیگر جمع زمان برگشت ۳ برنامه برابر است با :

$5 + 15 + 19 = 39$

همان‌طور که در درس سیستم عامل خوانده‌اید اگر الگوریتم زمان‌بندی SJF (Shortest Job First) باشـد، یعنی ابتدا به کوتاه‌ترین کارها سرویس دهیم، جمع زمان برگشت (یا زمان کل در سیستم) حداقل خواهـد بـود. در مثال فوق اگر به ترتیب [p3, p1, p2] سرویس‌دهی شوند، خواهیم داشت :



$4 + 9 + 19 = 32$ = جمع زمان‌های بازگشت

و عدد 32 فوق از هر حالت ممکن دیگر کمتر بوده و این را می‌توان بر مبنای قضیه زیر اثبات کرد.

قضیه : زمان کل در یک سیستم فقط هنگامی کمینه می‌شود که کارها بر مبنای افزایش زمان ارائه خدمات مرتب شده و زمان‌بندی شوند. از آنجا که مرتب‌سازی n قلم داده، از مرتبه $\theta(n \lg n)$ است لذا داریم :

$\theta(n \lg n)$ = مرتبه زمان‌بندی بر مبنای کمینه کردن زمان کل

الگوریتم فوق در واقع یک الگوریتم حریصانه است که در هر مرحله به سادگی کوچکترین کار انتخاب می‌شود.

الگوریتم فوق را می‌توان برای چند پردازنده به صورت زیر تعمیم داد. فرض کنید m پردازنده و n برنامه داری
برنامه‌ها را به ترتیب صعودی مرتب می‌کنیم. کار شماره ۱ را به پردازنده اول، کار ۲ را به پردازنده دوم و .... ک
شماره m را به پردازنده m ام می‌دهیم. بدیهی است که پردازنده اول زودتر از بقیه کار خود را تمام می‌کند، لـ
کار $(m + 1)$ را به پردازنده اول می‌دهیم، کار $(m + 2)$ را به پردازنده دوم می‌دهیم و الی آخر.

مثال : در سیستمی برنامه‌هایی با طول‌های اجرای $\{1,4,2,6,2,13,7,8\}$ موجودند. یک زمان‌بندی با زما
اجرای کلی 26 می‌تواند به صورت زیر باشد (فرض کنید هنگامی که پردازنده به برنامه‌ای داده شد دیگر قابل
پس‌گرفتن نباشد)

ابتدا برنامه‌ها را به صورت صعودی مرتب می‌کنیم :

$\{1,2,2,4,6,7,8,13\}$

سپس اولین کار را به CPU1 و دومین کار را به CPU2 می‌دهیم. بدیهی است که CPU1 کارش را زودتر تما
می‌کند. لذا کار سوم را به CPU1 می‌دهیم و الی آخر.

CPU1	1	2	6	8
CPU2	2	4	7	13

زمان اجرای کلی $= 26 \Rightarrow$

ولی اگر ترتیب 8 و 13 را در مرحله‌ی آخر عوض کنیم :

1	2	6	13
2	4	7	8

زمان اجرای کلی $= 22$

پس آن‌گونه که الگوریتم را برای چند پردازنده تعمیم دادیم الزاماً جواب بهینه را نمی‌دهد. بحث زمان‌بندی
چندپردازنده‌ای از مباحث درس سیستم‌عامل پیشرفته است.

پس مشاهده می‌کنید این روش حریصانه برای یک پردازنده همواره جواب بهینه را می‌دهد ولی برای د
پردازنده (آن گونه که ما تعمیم دادیم) الزاماً جواب بهینه را نمی‌دهد.

مثال ششم : زمان‌بندی با مهلت معین (Scheduling with Deadlines)

فرض کنید ٤ کار داریم که همگی در یک واحد زمانی (مثلاً یک ثانیه) اجرا و تمام می‌شوند. هر یک از این
کارها (برنامه‌ها) یک مهلت و یک بهره معین دارند. اگر برنامه‌ای تا قبل از مهلت داده شده به آن، اجرا شود
بهره معادل آن حاصل می‌گردد. هدف آن است که کارها به نحوی زمان‌بندی شوند که بیشترین بهره به دست
آید و البته لازم نیست الزاماً همه‌ی کارها زمان‌بندی شوند.

مثال : چهار کار زیر که همگی طول اجرای 1 داشته با بهره و مهلت داده شده را در نظر بگیرید :

کار	مهلت	بهره	زمان اجرا
1	2	30	1
2	1	35	1
3	2	25	1
4	1	40	1

اینکه مهلت کار 3 برابر 2 است یعنی اگر کار 3 در زمان 1 یا زمان 2 شروع شود، بهره‌ی 25 معادل آن به دست می‌آید و اگر بخواهد در زمان 3 یا بیشتر آغاز شود غیر قابل قبول بوده و بهره‌ای را نمی‌دهد. در مثال فوق فرض می‌کنیم زمان صفر نداریم و شروع زمان 1 می‌باشد. برای مثال فوق انواع زمان‌بندی‌های ممکن و نیز بهره‌های به دست آمده عبارتند از :

زمان‌بندی	بهره کل
[1,3]	$30 + 25 = 55$
[2,1]	$35 + 30 = 65$
[2,3]	$35+25=60$
[3,1]	$25 + 30 = 55$
[4,1]	$40 + 30 = 70$
[4,3]	$40 + 25 = 65$

در جدول فوق زمان‌بندی‌های غیرممکن نوشته نشده‌اند. مثلاً زمان‌بندی [1,4] امکان‌پذیر نیست زیرا اگر کار 1 اجرا شود دیگر مهلت کار 4 تمام شده و نمی‌تواند اجرا گردد. هدف از این مثال به دست آوردن زمان‌بندی [4,1] با بالاترین بهره‌ی کل 70 است. اگر بخواهیم همه‌ی زمان‌بندی‌های ممکن را درنظر بگیریم الگوریتمی از مرتبه‌ی فاکتوریل خواهد بود که خیلی زمان‌بر است لذا الگوریتمی حریصانه و بسیار سریع‌تر معرفی می‌کنیم.

در این روش حریصانه ابتدا کارها را به ترتیب نزولی بهره مرتب می‌کنیم و بررسی می‌کنیم که آیا این کارها را می‌توان به زمان‌بندی اضافه کرد یا خیر. یک مجموعه از کارها را «مجموعه‌ی امکان‌پذیر» گوئیم اگر حداقل یک ترتیب امکان‌پذیر برای آن کارها وجود داشته باشد. در مثال فوق {1,2} یک مجموعه‌ی امکان‌پذیر است چرا که ترتیب [2,1] امکان‌پذیر است. ولی {2,4} امکان‌پذیر نیست چرا که هیچ ترتیبی از اعضای آن قابل زمان‌بندی نیست.

قضیه : اگر S مجموعه‌ای از کارها باشد، این S مجموعه‌ای امکان‌پذیر است، اگر و فقط اگر مرتب‌شده‌ی کارهای درون آن براساس مهلت‌های صعودی امکان‌پذیر باشد.

با توجه به قضیه فوق الگوریتم حریصانه برای حل این مسأله به صورت زیر است.

الگوریتم حریصانه برای حل مسأله‌ی زمان‌بندی با مهلت معین : ابتدا کارها را براساس بهره به صورت نزولی (غیر صعودی) مرتب می‌کنیم و مجموعه‌ی S را در ابتدا تهی در نظر می‌گیریم. یک کار با بیشترین بهره را

انتخاب کرده و موقتاً به مجموعه‌ی S اضافه می‌کنیم. اگر این مجموعه‌ی S جدید، امکان‌پذیر بود آن کار را جزو S قرار می‌دهیم وگرنه آن را رد می‌کنیم. برای بررسی امکان‌پذیر بودن S ، کارهای موجود در S را بر حسب مهلت‌های صعودی مرتب می‌کنیم و تنها امکان‌پذیر بودن این حالت را بررسی می‌کنیم.

مثال : بهترین زمان‌بندی برای جدول زیر را جهت به دست آوردن حداکثر بهره بیان کنید.

کار	مهلت	بهره
1	3	40
2	1	35
3	1	30
4	3	25
5	1	20
6	3	15
7	2	10

حل : جدول فوق براساس بهره‌های نزولی مرتب‌شده است.

۱ـ ابتدا S برابر ϕ می‌شود.

۲ـ S برابر {1} شده و بررسی می‌گردد که ترتیب [1] امکان‌پذیر است.

۳ـ S برابر {1,2} شده و قبول می‌شود چرا که ترتیب [2,1] امکان‌پذیر می‌باشد.

٤ـ {1,2,3} رد می‌شود چرا که ترتیب $[\underset{\substack{\uparrow\\1}}{3} , \underset{\substack{\uparrow\\1}}{2} , \underset{\substack{\uparrow\\3}}{1}]$ با مهلت‌هایی که زیر آن با فلش رسم کرده‌ایم، امکان‌پذیر نمی‌باشد. پس هیچ ترتیب دیگری از {1,2,3} نیز امکان‌پذیر نمی‌باشد.

۵ـ S را برابر {1,2,4} قرار می‌دهیم و این مجموعه امکان‌پذیر است چرا که ترتیب $[\underset{\substack{\uparrow\\1}}{2} , \underset{\substack{\uparrow\\3}}{1} , \underset{\substack{\uparrow\\3}}{4}]$ امکان‌پذیر است.

٦ـ {1,2,4,5} رد می‌شود چرا که ترتیب $[\underset{\substack{\uparrow\\1}}{2} , \underset{\substack{\uparrow\\1}}{5} , \underset{\substack{\uparrow\\3}}{1} , \underset{\substack{\uparrow\\3}}{4}]$ امکان‌پذیر نیست.

۷ـ {1,2,4,6} رد می‌شود چرا که ترتیب $[\underset{\substack{\uparrow\\1}}{2} , \underset{\substack{\uparrow\\3}}{1} , \underset{\substack{\uparrow\\3}}{4} , \underset{\substack{\uparrow\\3}}{6}]$ امکان‌پذیر نیست.

۸ـ {1,2,4,7} رد می‌شود چرا که ترتیب $[\underset{\substack{\uparrow\\1}}{2} , \underset{\substack{\uparrow\\2}}{7} , \underset{\substack{\uparrow\\3}}{1} , \underset{\substack{\uparrow\\3}}{4}]$ امکان‌پذیر نیست (کار 4 در مهلت 3 تمام نمی‌شود.)

پس جواب مسأله مجموعه‌ی S = {1,2,4} و یک ترتیب ممکن برای آن [2,1,4] یا [2,4,1] با بهره‌ی کل 40+35+25=100 می‌باشد.

قضیه : الگوریتم حریصانه‌ای که در بالا شرح دادیم، همواره یک مجموعه‌ی بهینه را تولید می‌کند. اثبات قضیه فوق‌الذکر در کتاب نیپولیتان آمده است.

کارهایی که در الگوریتم فوق صورت می‌گیرد یکی مرتب کردن n کار براساس بهره است که به زمان $\theta(n \lg n)$ نیاز دارد، دیگری بررسی امکان‌پذیر بودن مجموعه‌ی پدید آمده در همه‌ی مرحله‌ها که از مرتبه $\theta(n^2)$ بوده و بر زمان $n \lg n$ برتری دارد. لذا :

زمان الگوریتم حریصانه زمان‌بندی با مهلت معین $= \theta(n^2)$

مثال هفتم : مسأله‌ی انتخاب فعالیت‌ها

n فعالیت با شماره‌های ۱ تا n می‌خواهند از یک منبع استفاده کنند و در یک زمان معین فقط یک فعالیت می‌تواند از این تک منبع استفاده نماید و فعالیت‌ها نمی‌توانند همپوشانی داشته باشند، مانند n سخنران که می‌خواهند از یک سالن اجتماعات استفاده نمایند. هر فعالیت i دارای یک زمان شروع s_i و یک زمان خاتمه f_i است و بدیهی است که $s_i \le f_i$ می‌باشد. فعالیت شماره i در صورت انتخاب شدن در فاصله زمانی $[s_i, f_i)$ اجرا خواهد شد. مسأله انتخاب حداکثر تعداد فعالیت‌هایی است که با هم همپوشانی نداشته باشند، مثلاً می‌خواهیم سخنران‌هایی را انتخاب کنیم که تعداد کل سخنران‌ها حداکثر گردد.

یک الگوریتم حریصانه برای حل این مسأله به صورت زیر است. فرض کنید فعالیت‌های داده شده به صورت صعودی بر مبنای زمان خاتمه شدنشان مرتب شده باشند، یعنی :

$$f_1 \le f_2 \le f_3 \le \ldots \le f_n$$

در غیر این صورت باید این n فعالیت را در زمان $\theta(n \log n)$ مرتب کرد در تابع زیر آرایه‌های ورودی S و F شروع و خاتمه فعالیت‌ها را نشان می‌دهند :

```
function   Activity - Selection (S[ ] , F[ ])
{
     A = {1} ;    j = 1;
     for  i = 2  to  n
        if   S[i] ≥ F[j]   then
        {
           A = A ∪ {i};
           j = i ;
        }
     return   A;
}
```

شماره فعالیت‌های انتخاب شده در مجموعه A جمع شده و توسط تابع برگردانده می‌شوند.

بدیهی است الگوریتم فوق که فقط یک حلقه for دارد از مرتبه $\theta(n)$ است، البته این به شرطی است که فعالیت‌ها از قبل مرتب شده باشند. در غیر این صورت زمان کلی الگوریتم برابر زمان مرتب‌سازی یعنی $\theta(n \log n)$ خواهد بود.

به عنوان مثال مراحل الگوریتم فوق را برای فعالیت‌های زیر بیان می‌کنیم :

i (شماره فعالیت)	1	2	3	4	5	6	7	8
S[i]	1	3	0	5	3	5	6	8
F[i]	4	5	6	7	8	9	10	11

در جدول فوق فعالیت‌ها براساس F[i] مرتب شده هستند :

i	شرط S[i] ≥ F[j]	A	j
2	غلط	{1}	1
3	غلط	-	-
4	درست	{1,4}	4
5	غلط	-	-
6	غلط	-	-
7	غلط	-	-
8	درست	{1,4,8}	8

پس فعالیت‌های با شماره 1 و 4 و 8 را انتخاب می‌کنیم.

توجه کنید که باید اثبات کنیم که الگوریتم حریصانه فوق همواره جواب درستی می‌دهد که این اثبات را در قسمت تحقیق و مطالعه بر عهده دانشجویان گذاشته‌ایم.

مثال هشتم : مسأله‌ی کلاسیک کوله پشتی (Knapsack)

دزدی با کوله‌پشتی خود وارد جواهر فروشی می‌شود. کوله پشتی او تحمل حداکثر وزن W را دارد و بیشتر از آن پاره می‌شود. هر قطعه جواهر وزن (w_i) و ارزش (p_i) معینی دارد. مسأله مهم دزد انتخاب قطعاتی است که حداکثر ارزش را داشته و نیز جمع وزن آنها از حداکثر W بیشتر نشود و این مسأله‌ی کلاسیک کوله‌پشتی است.

یک راه‌حل ساده ولی غیرهوشمندانه آن است که همه‌ی زیر مجموعه‌های ممکن از n قطعه را در نظر بگیریم و با مقایسه کردن همه‌ی آنها، حالتی را در نظر بگیریم که بیشترین ارزش را داشته و جمع وزن آنها W باشد. از آنجا که تعداد این زیر مجموعه‌ها 2^n است لذا این الگوریتم از مرتبه نمایی است و ما دنبال راه‌حل سریع‌تری هستیم.

ممکن است به نظر برسد یک راه‌حل حریصانه آن است که قطعاتی با ارزش بیشتر را زودتر برداریم تا هنگامی که جمع وزن آنها به W برسد. ولی با مثال‌های ساده‌ای می‌توان نشان داد که این روش غلط است. مثلاً سه قطعه با وزن و ارزش‌های زیر را در نظر بگیرید :

قطعه	1	2	3
وزن قطعه (کیلوگرم) = w_i	25	10	10
ارزش قطعه (میلیون تومان) = p_i	10	9	9

فرض کنید حداکثر W= 30 کیلوگرم باشد. در این حال دزد فقط قطعه اول را برمی‌دارد که ۱۰ میلیون تومان بدست می‌آورد. در حالی که اگر قطعات دوم و سوم را برمی‌داشت صاحب ۱۸ میلیون تومان می‌شد.

یک راه‌حل حریصانه‌ی دیگر آن است که قطعات را به ترتیب افزایش ارزش آنها به ازای واحد وزن مرتب کرده و سپس آنها را به ترتیب انتخاب کنیم.

برای جدول زیر و برای حداکثر W= 30 :

قطعه	اول	دوم	سوم
وزن قطعه = w_i	5	10	20
ارزش قطعه = p_i	50	60	140
نسبت $\frac{p_i}{w_i}$	$\frac{50}{5}=10$	$\frac{60}{10}=6$	$\frac{140}{20}=7$

دزد باید قطعه اول و سوم را بردارد و صاحب 190= 140 + 50 میلیون تومان شود. ولی حل بهینه شامل قطعات دوم و سوم با 200 میلیون تومان است، لذا الگوریتم فوق نیز بهینه نیست. مشکل الگوریتم فوق آن است که پس از انتخاب قطعه اول و سوم، 5 کیلوگرم از ظرفیت کوله‌پشتی باقی می‌ماند که قابل استفاده نیست و دزد مجبور است یک قطعه را به طور کامل بردارد و نمی تواند آن را به اجزای کوچکتر بشکند و این مسأله را «کوله پشتی صفر و یک» می‌گویند. یعنی یک قطعه یا باید کاملاً برداشته شود یا کاملاً کنار گذاشته شود. پس مسأله‌ی کوله‌پشتی صفر و یک را با الگوریتم حریصانه فوق نمی‌توان حل کرد.

ولی اگر جواهرات مثلاً به صورت کیسه‌های خاک طلا و نقره باشند، آنگاه دزد می‌تواند بخشی از کیسه‌های خاک طلا یا نقره را بردارد. در این حالت می‌توان اثبات کرد الگوریتم حریصانه فوق حتماً جواب بهینه را می‌دهد. به این مسأله «کوله‌پشتی کسری» می‌گوئیم و با روش حریصانه به سادگی حل می‌شود.

در مثال فوق دزد ابتدا کیسه اول و سوم را کامل برمی‌دارد و سپس ظرفیت ۵ کیلوگرم باقی‌مانده را با کیسه‌ی دوم پر می‌کند و بهره‌ی کل و حداکثر زیر را به دست می‌آورد :

$$50+140+\frac{5}{10}\times 60=190+30=220$$

پس الگوریتم حریصانه مسأله‌ی کوله‌پشتی کسری را با موفقیت و به سادگی حل می‌کند ولی نمی‌تواند مسأله کوله‌پشتی صفر و یک را حل کند. اگر n تعداد قطعات باشد، زمان الگوریتم حریصانه فوق برای مسأله‌ی کوله‌پشتی کسری مربوط به زمان مرتب‌سازی آن n قطعه بوده و از مرتبه‌ی $O(n \lg n)$ می‌باشد.

مقایسه روش حریصانه با روش پویا و حل مسأله کوله‌پشتی صفر و یک با تکنیک پویا

روش‌های حریصانه و پویا دو روش جهت حل مسائل بهینه‌سازی هستند. اغلب روش‌های حریصانه برای حل یک مسأله ساده‌تر و سریع‌تر از روش‌های پویا هستند ولی عموماً اثبات این موضوع که الگوریتم حریصانه حتماً یک راه‌حل بهینه را در همه‌ی حالات تولید می‌کند، سخت است.

مسأله کوله‌پشتی جهت مقایسه‌ی این دو تکنیک، نمونه‌ی مناسبی است. همان‌طور که دیدید، الگوریتم حریصانه به سادگی مسأله کوله‌پشتی کسری را حل می‌کند ولی نمی‌تواند کوله‌پشتی صفر و یک را حل کند. مسأله‌ی کوله‌پشتی صفر و یک را می‌توان با برنامه‌نویسی پویا حل کرد. ابتدا باید نشان دهیم اصل بهینگی در اینجا صادق است.

اگر A یک زیر مجموعه‌ی بهینه از n قطعه باشد دو حالت امکان‌پذیر است : یا A حاوی قطعه n ام هست یا نیست. اگر A حاوی قطعه n ام نباشد A با زیر مجموعه‌ای بهینه از (n - 1) قطعه اول برابر است. اگر A حاوی قطعه n ام باشد، منفعت کل برابر p_n به علاوه منفعت بهینه است هنگامی که قطعات را از 1 از $n-1$ قطعه اول انتخاب می‌کنیم (البته با رعایت این اصل که وزن کل از $W - w_n$ بیشتر نشود) لذا اصل بهینگی صادق است.

اگر به ازای $i > 0$ و $w > 0$ ، عنصر p[i][w] منفعت بهینه‌ی حاصل از انتخاب i قطعه‌ی اول باشد به گونه‌ای که وزن کل از w تجاوز نکند داریم :

$$p[i][w] = \begin{cases} \text{Max}(p[i-1][w], p_i + p[i-1][w - w_i]) & : w_i \le w \text{ اگر} \\ p[i-1][w] & : w_i > w \text{ اگر} \end{cases}$$

و منفعت حداکثر که به دنبال آن هستیم برابر p[n][W] می‌باشد و این مقدار با استفاده از آرایه‌ی دو بعدی p با سطرهای 0 تا n و ستون‌های 0 تا w و فرمول‌فوق به دست می‌آید. در آرایه‌ی p مقادیر سطر صفر و ستون صفر را مساوی 0 قرار می‌دهیم.

روشن است که تعداد عناصر محاسبه شده در الگوریتم فوق nW بوده و لذا روش پویای فوق از مرتبه‌ی $\theta(nW)$ می‌باشد که n تعداد قطعات و W حداکثر وزن قابل قبول است.

تذکر ۱ : اگر W در مقایسه با n بسیار بزرگ باشد $\theta(nW)$ از الگوریتم غیرهوشمند بامرتبه‌ی $\theta(2^n)$ نیز بدتر خواهد شد. در کتاب نیپولیتان الگوریتم پویای فوق به گونه‌ای بهبود یافته است که در بدترین حالت از $\theta(2^n)$ باشد و غالباً نیز از آن بسیار بهتر است پس داریم :

$$\text{زمان روش پویا برای حل کوله‌پشتی صفر و یک} = O(\text{Min}(2^n, nW))$$