مقدمه‌ای بر طراحی و تحلیل الگوریتم‌ها

علی نوراله


بسم الله الرحمن الرحیم

فهرست مطالب

پیش گفتار

درس طراحی و تحلیل الگوریتمها از دروس اصلی دوره کارشناسی مهندسی کامپیوتر و فن‌آوری اطلاعات محسوب میشود که نقش بسیار اساسی در درک و فهم دروس دیگر را نیز بازی میکند. همچنین در دوره کارشناسی ارشد نیز دروسی مبتنی بر این درس وجود دارند که از آن جمله میتوان به دروسی نظیر هندسه محاسباتی، الگوریتمهای پیشرفته، الگوریتمهای ژنتیک و پردازش تکاملی، الگوریتمهای طراحی فیزیکی مدارات VLSI و الگوریتمهای گراف اشاره کرد.

در این کتاب با روشهای طراحی الگوریتمها و اصول اساسی و مبنایی تحلیل الگوریتمها آشنا میشویم. همچنین جهت آشنایی با روشهای طراحی الگوریتمها مثالهایی از الگوریتمهای هندسی نیز بیان شده است که در میان فصول کتاب به آنها اشاره شده است. این کتاب مشتمل بر ۸ فصل است که در فصل ۱ یادآوری مختصری از درس ساختمان داده ها، در فصل ۲ اصول تحلیل الگوریتمها و نمادهای مجانبی، فصل ۳ تا ۷ نیز روشهای طراحی الگوریتمها را در بر گرفته است. در فصل آخر نیز مقدمه ای بر تئوری NP بیان شده است.

در اینجا لازم میدانم که از آقای محمدرضا باقری و خانم فرزانه صبوحی که در جمع آوری و تنظیم اولیه کتاب زحمات فراوانی را متحمل شدند تشکر نمایم.

۱- یادآوری

تعریف الگوریتم: هر دستوالعملی که مراحل مختلف انجام کاری را به زبان دقیق و با جزئیات کافی بیان نماید به طوریکه ترتیب مراحل و شرط خاتمه عملیات در آن کاملاً مشخص باشد الگوریتم نام دارد.

مطالعه الگوریتمها در برگیرنده موارد زیر است:

۱- طراحی الگوریتم
۲- معتبر سازی یا اثبات درستی الگوریتم
۳- بیان یا پیاده سازی الگوریتم
۴- تحلیل الگوریتم

که در این کتاب ما موارد اول و چهارم را مورد بررسی قرار میدهیم.

۱-۱- مروری بر روشهای مرتب سازی و پیچیدگی آنها

۱-۱-۱-مرتب سازی درجی (Insertion Sort)

در مرحله j ام این الگوریتم فرض بر این است که عناصر اول تا j-1 ام آرایه مرتب هستند و عنصر j ام در عناصر قبل از خود در محل مناسب درج میشود. حال به ازای j از ۲ تا n این عمل انجام میشود.

مرحله ۱: x[1] خودش به تنهایی بطور بدیهی مرتب است.

مرحله۲: x[2] را یا قبل از یا بعد از x[1] درج می کنیم طوریکه x[1] و x[2] مرتب شوند.

مرحله۳: x[3] را در مکان صحیح در x[1] وx[2] درج می کنیم به گونه ای که x[3],x[2],x[1] مرتب شده باشند.

...

مرحلهn: x[n] را در مکان صحیح خود در x[1] ,x[2],… x[n-1] به گونه ای درج می کنیم که کل آرایه مرتب شده باشد.

```
For j← 2 to n do
   k← x[j]
   i←j-1
   while x[i]>k and  i>0 do
      x[i+1] ← x[i]
      i ← i-1
   repeat
   x[i+1] ← k
repeat
```

به راحتی قابل ملاحظه است که الگوریتم بالا در بدترین وضعیت (عناصر آرایه نزولی باشند) در مرحله jام بـه ازای عنصر j ام j-1 مقایسه انجام میدهد و لذا دارای مرتبه زمانی $O(n^2)$ خواهد بـود!. و در بهتـرین وضـعیت (عناصـر آرایه از قبل صعودی باشند) در مرحله j ام به ازای عنصر j ام یک مقایسه با عنصر j-1 ام انجام میدهد و لذا دارای مرتبه زمانی $O(n)$ خواهد بود. این الگوریتم بطور متوسط نیز دارای مرتبه زمانی $O(n^2)$ خواهد بـود!. همچنـین از لحاظ مصرف حافظه کمکی نیز چون آرایه بدون نیاز به حافظه کمکی مرتـب میشـوند لـذا میـزان مصـرف حافظـه کمکی برابر با $O(1)$ خواهد بود.

نکته: مرتب سازی درجی را میتوان به صورت دیگری بازنویسی کرد که آن را مرتب سازی درجی دودویـی مینامنـد که در آن عمل جستجو برای پیدا کردن محل عنصر j ام توسط جستجوی دودویی انجام میشود ولی زمان اجرای آن تفاوتی نخواهد کرد.

۱-۱-۲-الگوریتم مرتب سازی ادغامی(Merge Sort)

اگر مجموعه ای از اعداد داشته باشیم و این مجموعه را به دو بخش تقسیم کنیم و هر بخش را جداگانه مرتب کنـیم و حاصل را به گونه ای ادغام کنیم که رعایت ترتیب شود آنگاه کل مجموعه اعداد مرتب خواهند شد.

در این روش بعد از تقسیم کردن داده های اولیه به دو بخش، هر بخش را نیز بـه همـین ترتیـب، یعنـی تقسـیم بـه مجموعه های کوچکتر و مرتب کردن و ادغام کردن آنها با هم مرتب می کنیم. اما شکستن زیر لیسـت‌هـا را تـا چـه زمانی انجام می دهیم؟ تا زمانی که تعداد عناصر هر لیست برابر با یک شود. آنگاه دو لیست کوچک تک عنصری را ادغام کرده و به صورت بازگشتی عمل می کنیم، مراحل کار این الگوریتم به صورت خلاصه در ذیل آمده است:

اگر تعداد داده ها یکی یا کمتر است نیازی به مرتب کردن نیست، برگرد.

در غیراینصورت وسط داده ها را پیدا کن

نیمه اول داده ها را به روش merge Sort (همین روش) مرتب کن

نیمه دوم داده ها را به روش merge Sort (همین روش) مرتب کن

دونیمه مرتب شده را به گونه ای ادغام کن که حاصل مرتب باشد.

```
Procedure mergesort(A,l,u)
   If l<u then
      mid ← (l+u) / 2
      Mergesort(A,l, mid)
      Mergesort(A,mid+1, u)
      Merge(A, l, mid, u)
   endif
end.
```

$$
\begin{aligned}
T(n) &= 2T(\frac{n}{2}) + n \\
&= 2[T(\frac{n}{4}) + \frac{n}{2}] + n = 4T(\frac{n}{4}) + 2n \\
&= 8T(\frac{n}{8}) + 3n \\
&= 2^i T(\frac{n}{2^i}) + in \\
&= 2^{\log n} T(1) + n.\log n = c.n + n.\log n = \mathrm{O}(n.\log n)
\end{aligned}
$$

از این رو الگوریتم مرتب سازی ادغامی در همه حالات دارای مرتبه زمانی O(n. log n) میباشد. همچنین بدلیل بازگشتی بودن به اندازه O(log n) از حافظه پشته استفاده میکند. ولی با عیر بازگشتی نوشتن الگوریتم میتوان حافظه مصرفی را به O(1) کاهش داد ولی زمان واقعی الگوریتم بیشتر خواهد شد!.

نکته: مرتب سازی ادغامی بازگشتی را میتوان به صورتهای متفاوتی از جمله مرتب سازی ادغامی طبیعی (Natural Merge Sort) و یا مرتب سازی ادغامی غیر بازگشتی بازنویسی کرد.

۱-۱-۳-مرتب سازی سریع(Quick Sort)

براساس این الگوریتم نیز مجموعه اعداد یا به طورکلی داده ها به دو بخش تقسیم می شوند و هر بخش جداگانه مرتب می شوند. تفاوت های عمده بین روش مرتب سازی و مرتب سازی ادغام به صورت زیر هستند:

اول اینکه در روش مرتب سازی سریع یکی از عناصر مجموعه را به عنوان عنصر محوری انتخاب می‌کنیم که فرقی نمی کند کدام عنصر از مجموعه باشد.

دوم اینکه مجموعه درواقع به سه قسمت شکسته می شود الف) عنصر محوری ب) عنصر کوچکتر از عنصر محوری ج) عناصر بزرگتر از عنصر محوری

سوم اینکه در هر مرحله محل عنصر محوری در آرایه ثابت می شود و در واقع مکانش پیدا می شود و چهارم اینکه احتیاجی به ادغام هر مجموعه نیست. بلکه در حقیقت در هر مرحله اجراء محل یک عضو از آرایه ثابت می شود. (همان عضو محوری)

بعد از یافتن محل ثابت عضو محوری (Partition Phase) هر یک از دو بخش دیگر نیز به روش فوق به صورت بازگشتی مرتب می شوند(Recursion Phase).

به راحتی قابل ملاحظه است که الگوریتم بالا در بدترین وضعیت (عناصر آرایه از قبل مرتب باشند) در هر مرحله لیست را به دو قسمت با اندازه های کاملاً دور از هم تقسیم میکند و لذا دارای مرتبه زمانی $O(n^2)$ خواهد بود!. و در بهترین وضعیت (عناصر آرایه به گونه‌ای باشند که در هر مرحله عنصر میانه به عنوان قلم محوری انتخاب شود) در هر مرحله لیست به دو قسمت با اندازه نزدیک به هم تقسیم میشود و لذا دارای مرتبه زمانی O(n. log n) خواهد بود!. این الگوریتم بطور متوسط نیز دارای مرتبه زمانی O(n. log n) خواهد بود!. همچنین از لحاظ مصرف حافظه کمکی نیز به دلیل بازگشتی بودن الگوریتم در بدترین حالت برابر با O(n) و در بهترین حالت و حالت متوسط برابر با O(log n) خواهد بود.

```
Procedure Quick_Sort( A, l, u)
   If l < u then
      Pivot ← A[l]
      i←l
      j←u;
      repeat
         Do i ← i + 1 until A[i] > pivot
         Do j ← j - 1 until A[j] < pivot
         If i < j then
            Swap(A[i], A[j])
         Endif
      Until i ≥ j
      Swap(A[l], A[j])
      Quick_Sort(A, l, j - 1)
      Quick_Sort(A, j + 1, u)
   endif
End.
```

۱-۱-٤-مرتب سازی توده ای (Heap Sort)

قبل از توضیح الگوریتم باید دو الگوریتم مورد نیاز را یادآوری کنیم:

- Adjust: در این الگوریتم فرض بر این است که درختی وجود دارد که از Heap بودن زیر درخت چپ و راست آن مطمئن هستیم ولی تظمینی بر Heap بودن کل درخت وجود ندارد و لذا شبیه عمل حذف عنصر بیشینه از یک MaxHeap عمل میکنیم و ریشه را با مقایسه با فرزندانش (در صورت وجود) تا جای مورد نیاز به سمت برگها پیش میبریم. و لذا دارای مرتبه زمانی O(log n) خواهد بود.
- Heapify: در این الگوریتم یک درخت دودویی کامل به یک Heap تبدیل میشود. با شروع از عنصر n/2 (آخرین عنصر دارای فرزند) به سمت عنصر سر لیست پیش میرویم و آن را در عناصر بعد از خود adjust میکنیم. مرتبه زمانی الگوریتم برابر با O(n) میباشد.

```
Procedure Adjust(A, i, n)
   j← 2*i
   item←A(i)
   while j≤n do
      if j<n and A(j)<A(j+1) then j←j+1 endif
      if item ≥A(j) then
         exit
      else
         A(⌊j/2⌋)←A(j)
         j←2*j
      endif
   repeat
   A(⌊j/2⌋)←item
end.
```

```
Procedure Heapify(A, n)
   for i← ⌊i/2⌋ to 1 do
      call adjust(A,i,n)
   repeat
end.
```

نکته: الگوریتم Heapify به دلیل زیر دارای مرتبه زمانی O(n) است!.

$$\sum_{i=1}^{k}(k-i)2^{i-1} \overset{j=k-i}{=} \sum_{j=0}^{k-1} j.2^{k-j-1} = 2^{k-1}\sum_{j=0}^{k-1}\frac{j}{2^j} \le n\sum_{j=0}^{k-1}\frac{j}{2^j} \le n\sum_{j=0}^{\infty}\frac{j}{2^j} = 2n = O(n)$$

که در آن i نشان دهنده سطح گره های درخت، k برابر با عمق درخت ($\lceil \log(n+1) \rceil$) میباشد. واضح است که تعداد گره های سطح i ام درخت حداکثر برابر با 2^{i-1} میباشد. در مرتب‌سازی توده‌ای ابتدا لیست یا آرایه را به یک Heap تبدیل میکنیم (به کمک الگوریتم Heapify) و سپس بعد از Heap شدن لیست عناصر سر درخت (عنصر ماکزیمم) را با عنصر انتهای لیست جابه‌جا کرده و دوباره بدون در نظر گرفتن عنصر جابه‌جا شده (در انتها) درخت را با کمک الگوریتم adjust به یک Heap تبدیل میکنیم. این عمل n-1 بار صورت میگیرد و به این ترتیب عناصر از بیشینه به کمینه از انتهای آرایه به سمت ابتدای آرایه چیده خواهد شد.

برای مرتب‌سازی لیست، ابتدا یک heap بزرگ با استفاده از فرخوانی متوالی adjust ایجاد می‌کنیم و سپس n-1 گذر روی لیست داریم. در هرگذر اولین رکورد heap را با آخرین رکورد تعویض می‌کنیم. چون اولین رکورد شامل بزرگترین کلید است، این رکورد اکنون در موقعیت مرتب شدن خودش است پس‌اندازه heap را کاهش می‌دهیم و دوباره آن را تراز می‌کنیم، به عنوان مثال، در اولین گذر، رکوردی با بزرگترین کلید را در n اُمین محل می‌گذاریم، در گذر دوم رکوردی با دومین کلید بزرگی را در محل n-1 می‌گذاریم و i اُمین گذر رکوردی با i اُمین کلید بزرگی را در محل n-i+1 می‌گذاریم.

از این رو الگوریتم مرتب سازی توده‌ای در همه حالات دارای مرتبه زمانی O(n. log n) میباشد!. همچنین بدلیل غیر بازگشتی بودن دارای حافظه مصرفی O(1) خواهد بود.

```
Procedure HeapSort(A, n)
   call Heapify(A,n)
   for i←n to 2 do
      Swap(A(i), A(1))
      call Adjust(A,1,i-1)
   repeat
end.
```

۱-۲- درخت پوشای مینیمم

۱-۲-۱-الگوریتم راشال (Kruskal)

در این روش درخت پوشای با کمترین هزینهT، لبه به لبه ساخته می شود. لبه های مورد استفاده درT، به ترتیب صعودی وزنها می باشد. یک لبه در T خواهد بود، اگر با لبه های قبل که درT بوده اند تشکیل حلقه ندهد.

۱-۲-۲-الگوریتم پریم (Prim)

در این روش درخت پوشای با کمترین هزینهT ، راس به راس به T که در ابتدا شامل یکی از رئـوس (بـه دلخـواه) است، اضافه میشوند. در هر مرحله یال با کمترین هزینه که یک راس مجاور آن داخـل T و راس مجـاور دیگـرش خارج از T باشد به درخت اضافه میشود.

دو الگریتم بالا به طور کامل در فصل مربوط به روش طراحی حریصانه شرح داده خواهند شد.

۱-۳- پیمایش و جستجوی گراف‌ها

۱-۳-۱-جستجو و پیمایش عمقی (DFS)

در این روش پیمایش گره ها در امتداد یک شاخه تا انتهای آن ادامه پیدا می کند و با پایان یافتن شاخه‌ها یک مرحله به عقب برگشت کرده و مجدداً بررسی میکنیم. در این روش با مشاهده یک گره جدید، پیمایش سایر فرزندان گـره قبلی به تعویق افتاده و گره جدید بررسی می شود. در واقع در این روش با گذر از هر گره به سمت عمـق درخـت گره فعلی در پشته قرار داده میشود که با بازگشتی طراحی کردن الگوریتم پشته مورد نظر به صـورت پنهـان سـاخته خواهد شد.

```
Procedure DFS(v)
   Visit(v)
   For each vertex w adjacent to v do
      If not visited w then
         DFS( w)
      Endif
   repeat
end
```

در الگوریتم بالا در بدترین وضعیت (گراف همبند باشد) هر یال حداکثر دو بار پیموده میشـود (!) و هـر راس نیـز یکبار ملاقات میشود و لذا مرتبه زمانی الگوریتم O(n+e) است.

۱-۳-۲-جستجو و پیمایش ردیفی(BFS)

اگر در الگوریتم قبلی به جای پشته یک صف در نظر بگیریم به الگوریتم جدیدی خواهیم رسید که BFS نـام دارد. برای پیمایش کلیه گره ها گره آغاز را انتخاب می کنیم. در این روش بعد از دیدن هرگره کلیه فرزنـدان همـان گـره ملاقات شده و سپس فرزندان اولین فرزند گره اصلی و بعد فرزندان دومین فرزند گره اصلی و تا انتها پیش می رود.

بنابراین روش کار این است که نخست دومین گره را ملاقات می کنیم و سپس کلیه فرزندان آن را ملاقـات کـرده و در انتهای صف قرار می دهیم و سپس از داخل صف، اولین گره را برداشــته و فرزنـدان آن را ملاقـات کـرده و بـه انتهای صف می افزاییم.

```
Procedure BFS(v)
   Visit(v)
   AddQueue(v)
   While Queue is not empty do
      DeleteQueue(v)
      For each w adjacent to v do
         If not visited w then
            Visit(w)
            AddQueue(w)
         Endif
      Repeat
   Repeat
end
```

طبق دلیل مطرح شده در الگوریتم DFS لذا مرتبه زمانی این الگوریتم نیز O(n+e) است.
توسط الگوریتمهای DFS و BFS میتوان یک گراف غیر همبند را نیز پیمایش کرد. به این صـورت کـه هـر بـار از یک گره ملاقات نشده شروع کرده و آن را DFS یا BFS میکنیم و لذا میتوان به الگوریتمهایی مبتنـی بـر پیمـایش عمقی یا ردیفی رسید.

۲- تحلیل الگوریتمها

در این بخش به تعریف نمادهای مجانبی مورد نیاز در تحلیل الگوریتمها و بیان چند قضیه قضیه در مورد آن میپردازیم.

۲-۱- نمادهای مجانبی

تعریف: $f(n)\in O(g(n))$ اگر و فقط اگر ثابتc و ثابت $n_\circ$ وجود داشته باشند که برای همه مقادیر $n \ge n_\circ$، داشته باشیم.

$$\forall n \ge n_0 : |f(n)| \le c|g(n)|$$

تعریف: $f(n)\in \Omega[g(n)]$ اگر و فقط اگر ثابتc و ثابت صحیح $n_\circ$ وجود داشته باشد که:

$$\forall n \ge n_0 : |f(n)| \ge c|g(n)|$$

در این صورت کران پایین تعداد اعمال لازم جهت یک الگوریتم به $c|g(n)|$ محدود گردیده است. و بدین صورت خوانده می شود: $f(n)$ اُمگای بزرگ $g(n)$ است.

تعریف: $f(n)\in \theta[g(n)]$ اگر و فقط اگر ثابتهای c_1 و c_2 و ثابت صحیح $n_\circ$ وجود داشته باشد به گونه ای که برای همه مقادیر $n \ge n_\circ$ داشته باشیم.

$$\forall n \ge n_0 : c_1|g(n)| \le |f(n)| \le c_2|g(n)|$$

قضیه (۱):

$$if\ \ f(n) = a_m n^m + \cdots + a_1 n + a_0\ \ then\ \ f(n) = \begin{cases} O(n^m) \\ \Omega(n^m) \\ \Theta(n^m) \end{cases}$$

قضیه (۲):

$$f(n) = O(g(n)) \iff g(n) = \Omega(f(n))$$

قضیه (۳):

$$f(n)=\Theta(g(n)) \Leftrightarrow \begin{cases} f(n)=O(g(n)) \\ f(n)=\Omega(g(n)) \end{cases}$$

تعریف: f(n)∈o(g(n)) اگر و فقط اگر ثابت c و ثابت $n_\circ$ وجود داشته باشند که برای همه مقادیر $n \geq n_\circ$، داشته باشیم:

$$\forall n \geq n_0 : |f(n)| < c.|g(n)|$$

نماد o را میتوان به گونه دیگری نیز تعریف کرد:

$$f(n)=o(g(n)) \Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)}=0$$

مثال:

$$\frac{n}{\log n}=o(n)$$

تعریف: f(n)∈ω(g(n)) اگر و فقط اگر ثابت c و ثابت $n_\circ$ وجود داشته باشند که برای همه مقادیر $n \geq n_\circ$، داشته باشیم:

$$\forall n \geq n_0 : |f(n)| > c.|g(n)|$$

نماد ω را میتوان به گونه دیگری نیز تعریف کرد:

$$f(n)=\omega(g(n)) \Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)}=\infty$$

نکته: مقایسه نمادهای مجانبی دو تابع f و g را با دو عدد حقیقی a و b میتوان بصورت زیر بیان کرد:

$$f(n)=O(g(n)) \quad \approx \quad a \leq b$$
$$f(n)=o(g(n)) \quad \approx \quad a < b$$
$$f(n)=\Omega(g(n)) \quad \approx \quad a \geq b$$
$$f(n)=\omega(g(n)) \quad \approx \quad a > b$$
$$f(n)=\Theta(g(n)) \quad \approx \quad a = b$$

نکته:

$$2^n \neq O(2^n)$$

۲-۲- تحلیل حالت متوسط الگوریتم

تحلیل حالت متوسط یک الگوریتم دارای اهمیت بیشتری نسبت بدترین حالت و بهترین حالت الگوریتم میباشد. اگر تعداد تکرار یک دستورالعمل در یک الگوریتم با n ورودی را در حالت متوسط با A(n) نشان دهیم آنگاه یک رابطه ساده جهت بدست آوردن حالت متوسط بصورت زیر میباشد:

$$A(n)=\sum_{i\in S}P(i).F(i)$$

که در آن S مجموعه حالات ممکنه، P(i) احتمال رخ دادن حالت i ام و F(i) تعداد تکرار دستور مورد نظر در حالت i ام میباشد.

مثال: متوسط تعداد تکرار دستورالعمل (×) را در برنامه زیر بدست آورید:

```
Procedure   max (A,n,j)
    j← 1
   for  i←2   to  n   do
          if   A(i) > A(j)    then
                j←i          (*)
          endif
    repeat
end.
```

تعداد تکرار دستور× در یک آرایه n تایی را بطور متوسط برابر با A_n در نظر میگیریم و لذا:

$0 \le A_n < n-1$

تعداد تکرار دستور (*) در این زیربرنامه در صورتی صفر است که بزرگترین عنصر در ابتدای آرایه باشد.

$$\begin{cases} A_1=0 \\ A_n=\dfrac{n-1}{n}A_n+\dfrac{1}{n}(1+A_{n-1}) \end{cases}$$

$$\Rightarrow A_n=\frac{1}{n}+A_{n-1}=\frac{1}{n}+\frac{1}{n-1}+\cdots+\frac{1}{2}+A_1=\underbrace{\frac{1}{n}+\frac{1}{n-1}+\cdots+\frac{1}{2}+\frac{1}{1}}_{\substack{=H_n \\ =\ln n+\gamma+\Theta(\frac{1}{n}) \\ \gamma=0.5772}}-1=\Theta(\log n)$$

تعداد تکرار دستور $A(i)>A(j)$

$$\frac{1}{n}(n-1)+\frac{1}{n}(n-1)+\frac{1}{n}(n-1)+\cdots+\frac{1}{n}(n-1)=n-1$$

پس همیشه n-1 بار تکرار می شود.

مثال: متوسط تعداد تکرار دستورالعمل(×) را در برنامه زیر بدست آورید؟

```
Function Linear_search(A, n, x)
   for   i←1   to   n   do
       if   A(i)=x     then        (*)
             return i
       endif
   repeat
end.
```

$$\text{متوسط تکرار} = \frac{1}{n}\times 1+\frac{1}{n}\times 2+\frac{1}{n}\times 3+\cdots+\frac{1}{n}\times n=\frac{1}{n}\times n=\frac{1}{n}(1+2+3+\cdots+n)$$

$$\Rightarrow \frac{1}{n}\frac{(n+1)n}{2}=\frac{n+1}{2}$$

نکته: در الگوریتم قبلی در هر صورت حلقه تا n اجرا می شد ولی در این مثال هرگاه که شرط برقرار باشد از حلقه خارج می شویم.

مثال: میانگین زمان اجرای الگوریتم QuickSort را بدست آورید:

$$T(n)=\begin{cases} 0 & ;n=1 \\ (n-1)+\dfrac{2}{n}\displaystyle\sum_{i=1}^{n-1}T(i) & ;n>1 \end{cases}$$

$$\Rightarrow T(n+1)=n+\frac{2}{n+1}\sum_{i=1}^{n}T(i)$$

$$\Rightarrow \begin{cases} nT(n)=n(n-1)+2\displaystyle\sum_{i=1}^{n-1}T(i) \\ (n+1)T(n+1)=n(n+1)+2\displaystyle\sum_{i=1}^{n}T(i) \end{cases} *(-1)+$$

$$\Rightarrow (n+1)T(n+1)-nT(n)=2n+2T(n)$$

$$\Rightarrow (n+1)T(n+1)=2n+(n+2)T(n)$$

$$\Rightarrow T(n+1)=\frac{2n}{n+1}+\frac{n+2}{n+1}T(n)$$

$$\overset{\overbrace{n\ge1\Rightarrow\frac{2n}{n+1}\le2}}{\Rightarrow}\quad T(n+1)\le 2+\frac{n+2}{n+1}T(n)$$

$$\Rightarrow T(n)\le 2+\frac{n+1}{n}T(n-1)\Rightarrow T(n)\le 2+\frac{n+1}{n}[2+\frac{n}{n-1}T(n-2)]$$

$$\Rightarrow T(n)\le 2\frac{n+1}{n+1}+2\frac{n+1}{n}+\frac{n+1}{n-1}T(n-2)$$

$$\Rightarrow T(n)\le 2(n+1)(\frac{1}{n+1}+\frac{1}{n})+\frac{n+1}{n-1}[2+\frac{n-1}{n-2}T(n-3)]$$

$$\Rightarrow T(n)\le 2(n+1)(\frac{1}{n+1}+\frac{1}{n}+\frac{1}{n-1})+\frac{n+1}{n-2}T(n-3)$$

$$\Rightarrow T(n)\le 2(n+1)\underbrace{(\frac{1}{n+1}+\frac{1}{n}+\frac{1}{n-1}+\cdots+\frac{1}{3})}_{H_{n+1}-\frac{1}{2}-1}+\frac{n+1}{2}\underbrace{T(1)}_{0}$$

$$\Rightarrow T(n)=O(n.\log n)$$

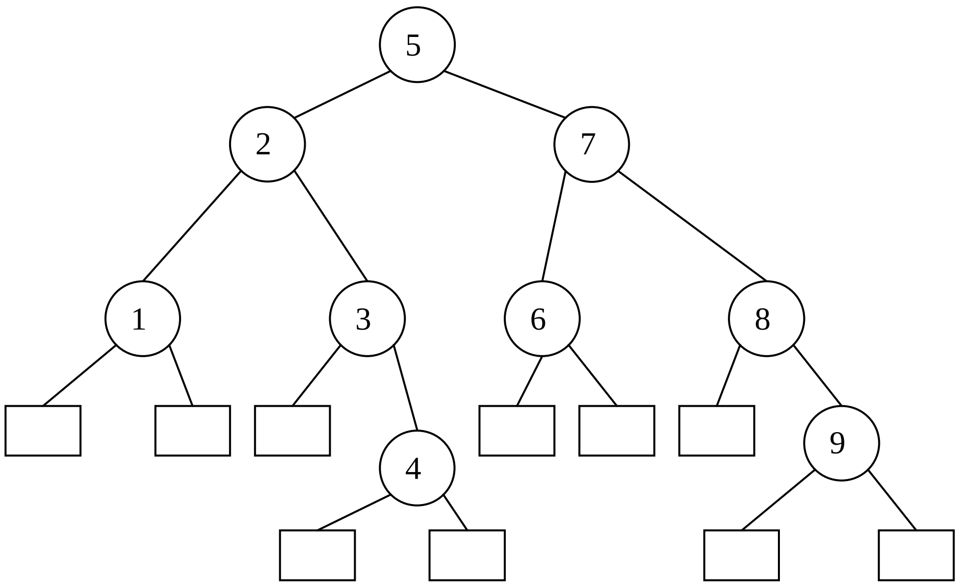
نتیجه: بدترین حالت میانگین زمان اجرای الگوریتم QuickSort برابر O(n. log n) میباشد.

مثال: فرض کنید یک آرایه مرتب (صعودی) و کلید x داده شده باشند. متوسط زمان اجرای الگوریتم BinarySearch را محاسبه کنید.

```
Function BinarySearch( A, n, x)
   low ← 1
   high←n
   while low≤high do
      mid←⌊(low+high)/2⌋
      if x=A(mid) then
         return mid
      elsif x>A(mid) then
         low←mid+1
      else
         high←mid-1
      endif
   repeat
   return 0
end.
```

در این الگوریتم اگر x داخل آرایه موجود باشد تعدادی مقایسه با عناصر آرایه صورت میگیرد و نهایتاً جستجو موفق خواهد بود و اگر x داخل آرایه موجود نباشد، تعدادی مقایسه با عناصر آرایه صورت میگیرد و نهایتاً جستجو ناموفق خواهد بود. مقایسه ها بصورت جستجو در یک درخت جستجوی دودویی انجام میشوند برای نمونه به آرایه زیر و درخت حاصل از جستجوی آن دقت کنید:

A	1	2	3	4	5	6	7	8	9
مقادیر	6	10	16	19	21	42	66	81	93
تعداد مقایسات در حالت موفق	3	2	3	4	1	3	2	3	4



در درخت بالا گره های خارجی با مستطیل و گره های داخلی با دایره نشان داده شده اند.درخت بالا را درخت تصمیم دودویی مینامند. جستجوهای موفق به گره های داخلی و جستجوهای ناموفق به گره های خارجی ختم میشوند.

نکته: اگر تعداد گره های خارجی را با n_0 نشان داده و تعداد گره های داخلی را با n نشان دهیم آنگاه $n_0=n+1$

میانگین جستجوهای موفق=(3+2+3+4+1+3+2+3+4)/9=25/9=2.77

نکته: فرض کنید احتمال موجود بودن x داخل عناصر مختلف آرایه و همچنین در صورت موجود نبودن x در آرایه احتمال قرار داشتن x بین عناصر آرایه مساوی است(به این معنی که احتمال قرار داشتن x در مجموعه [5,9] و [11,15] و ... مساوی است(کلاسها متساوی الاحتمالند).

A	1	2	3	4	5		6	7	8	9
تعداد مقایسات در حالت ناموفق	3	3	3	4	4	3	3	3	4	4

میانگین جستجوهای ناموفق=(3+3+3+4+4+3+3+3+4+4)/10=37/10=3.7

قضیه: اگر $n\in[2^{k-1},2^k)$ باشد، آنگاه الگوریتم جستجوی دودویی مستلزم حداکثر k مقایسه برای جستجوی موفق و k-1 یا k مقایسه برای جستجوی ناموفق است.

نتیجه: جستجوی ناموفق دارای همیشه دارای مرتبه زمانی $\Theta(\log n)$ است. جستجوی موفق در بهترین وضعیت $\Omega(1)$ و در بدترین وضعیت $O(\log n)$ میباشد.

حال متوسط تعداد مقایسات در جستجوی موفق را محاسبه میکنیم:

اگر v یک گره داخلی در درخت تصمیم باشد و سطح آن را با Level(v) نشان دهیم، آنگاه تعریف میکنیم:

$$d(v)=level(v)-1$$

$$E=\sum_{v\in External\ \ Nodes} d(v)$$

$$I=\sum_{v\in Internal\ \ Nodes} d(v)$$

قضیه: E=I+2n

راهنمایی اثبات: با استقرا اثبات کنید.

حال اگر میانگین تعداد مقایسات در جستجوی موفق در یک درخت تصمیم با n گره داخلی را با S(n) و میانگین مقایسات در جستجوی ناموفق را با U(n) نشان دهیم، آنگاه:

$$S(n)=\frac{\sum_{v\in Internal\ \ Nodes}[d(v)+1]}{n}=\frac{I+n}{n}$$

$$U(n)=\frac{\sum_{v\in External\ \ Nodes} d(v)}{n+1}=\frac{E}{n+1}$$

حال با استناد به قضیه بالا و دو رابطه به دست آمده میتوان نتیجه گرفت که:

$$\Rightarrow S(n)=\frac{E-2n+n}{n}=\frac{(n+1)U(n)-n}{n}=(1+\frac{1}{n})U(n)-1$$

و چون قبلاً دیدیم که $U(n)=\Theta(\log n)$ در نتیجه:

$$S(n)=\Theta(\log n)$$

۲-۳- روابط بازگشتی

در این قسمت به طور مختصر از حل روابط بازگشتی سخن خواهیم گفت. روابط بازگشتی را میتوان از روی درجه آن دسته بندی کرد:

۲-۳-۱-روابط بازگشتی درجه ۱

این روابط به سادگی از طریق روش جایگذاری قابل حل میباشند. به این معنی که با باز کردن رابطه بصورت تو در تو میتوان رابطه را حل نمود.

مثال:

$$T(n)=\begin{cases}2T(n-1)+1 & ;n>0\\ 1 & ;n=0\end{cases}$$

$$\begin{aligned}\Rightarrow T(n)&=2T(n-1)+1\\ &=2[2T(n-2)+1]+1=4T(n-2)+2+1=2^2T(n-2)+2^1+2^0\\ &=2^iT(n-i)+2^{i-1}+\cdots+2^1+2^0\\ &=2^nT(0)+2^{n-1}+\cdots+2^0=\sum_{i=0}^{n}2^i=2^{n+1}-1=O(2^n)\end{aligned}$$

۲-۳-۲-روابط بازگشتی درجه ۲ (همگن)

این روابط به روش محسبه معادله مشخصه قابل حل هستند.

مثال: تعداد فراخوانی های تابع مقابل را بدست آورید؟

```
function   f(n)
      if   n<2     then
             return      n
      else
             return    f(n-1)+f(n-2)
      end if
end.
```

A_n= تعداد فراخوانی های تابع f با ورودی n

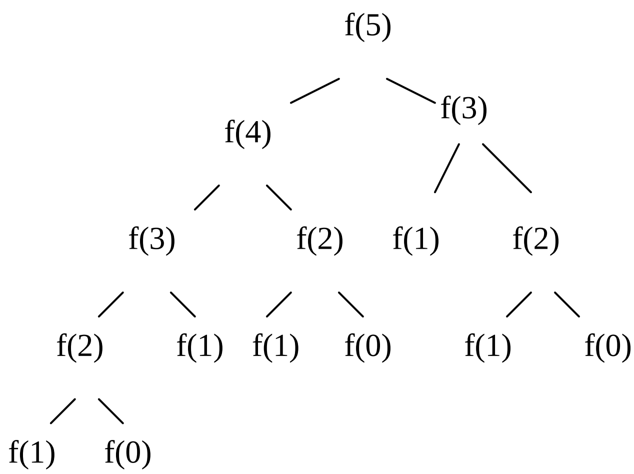
$$\begin{cases}A_n=A_{n-1}+A_{n-2}+1\\ A_0=1\\ A_1=1\end{cases}$$

راه حل:

$$A_n = A_{n-1} + A_{n-2} + 1$$
$$A_n + 1 = A_{n-1} + A_{n-2} + 1 + 1$$
$$*B_n = A_n + 1$$
$$* \Rightarrow B_n = B_{n-1} + B_{n-2}$$
$$B_0 = 2 \qquad B_1 = 2$$
$$B_n = r^n \Rightarrow r^n = r^{n-1} + r^{n-2}$$
$$r^2 - r - 1 = 0$$
$$r_1 = \frac{1+\sqrt{5}}{2} \qquad r_2 = \frac{1-\sqrt{5}}{2}$$
$$B_n = C_1 r_1^n + C_2 r_2^n$$
$$\Rightarrow \quad B_0 = C_1 r_1^0 + C_2 r_2^0 = C_1 + C_2 = 2 \quad if \quad n=0 \quad ①$$
$$\Rightarrow \quad B_1 = C_1 r_1^1 + C_2 r_2^1 = C_1\left(\frac{1+\sqrt{5}}{2}\right) + C_2\left(\frac{1-\sqrt{5}}{2}\right) = 2 \quad if \quad n=1 \quad ②$$
$$① \text{و} ② \quad \Rightarrow \quad C_1 = \frac{1+\sqrt{5}}{\sqrt{5}} \qquad C_2 = \frac{1-\sqrt{5}}{\sqrt{5}}$$
$$B_n = \frac{2}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^{n+1} - \left(\frac{1-\sqrt{5}}{2}\right)^{n+1}\right]$$
$$A_n = B_n - 1 \cong (1/6)^n$$

روش حل پروسیجر قبلی تقسیم و حل میباشد و یک درخت را به شکل زیر پویش می کند این الگوریتم مقادیری را چندین بار بصورت تکراری محاسبه می کند و به همین علت مرتبه زمانی الگوریتم نمائی می باشد.

مثال درخت فراخوانی f(5) :



این مسئله را به وسیله یک حلقه برگ‌ها به سمت ریشه با حفظ مقادیر هم می‌توان حل کرد که این روش پایین به بالا (bottom-up) است در این روش حالت‌های تکراری حذف می‌شود. (روش برنامه سازی پویا)

٢-٤- قضیه اصلی (Master Theorem)

در فصل بعدی با یکی از روشهای حل مسئله به نام روش تقسیم و حل آشنا خواهیم شد که در تحلیل مرتبه زمانی الگوریتمهای طراحی شده به آن روش به قضیه‌ای به نام قضیه اصلی نیاز خواهیم داشت و لذا در این قسمت به بیان آن خواهیم پرداخت.

قضیه اصلی:

$$T(n) = aT\left(\frac{n}{b}\right) + Cn^k \quad if$$

که در آن c,a,b و k اعداد ثابت و b>1 است آنگاه:

$$T(n) = \begin{cases} \theta\left(n^{\log_b^a}\right) & a > b^k \\ \theta\left(n^k \log n\right) & a = b^k \\ \theta\left(n^k\right) & a < b^k \end{cases}$$

مثال:

زمان اجرای Binary search را به کمک قضیه اصلی به دست آورید.

$$T(n) = 1 + T\left(\frac{n}{2}\right)$$

$$\begin{cases} a = 1 & \\ b = 2 & a = b^k \Rightarrow 1 = 2^0 \\ k = 0 & \theta\left(n^k \log n\right) = \theta(\log n) \end{cases}$$

نکته: در قضیه اصلی مقدار c در زمان نهایی تأثیری ندارد.

مثال: زمان اجرای مثال‌های زیر را به کمک قضیه اصلی محاسبه کنید؟

$$T(n) = T\left(\frac{3n}{2}\right) + n^2 \Rightarrow$$

$$\begin{cases} a = 1 & \\ b = \frac{2}{3} & a < b^k \\ k = 2 & \theta\left(n^k\right) = \theta\left(n^2\right) \end{cases}$$

مثال:

$$T(n) = 2T\left(\sqrt{n}\right) + \log n$$

$$n = 2^m \Rightarrow m = \log_2^n \qquad ①$$

$$T\left(2^m\right) = T\left(2^{\frac{m}{2}}\right) + m$$

$$T\left(2^m\right) = s(m)$$

$$s(m) = 2s\left(\frac{m}{2}\right) + m$$

$$\theta(m \log m), \quad ① \quad \Rightarrow \theta(\log n . \log\log n)$$

مثال:

$$T(n) = 3T\left(\frac{n}{4}\right) + n\log n$$

$$\begin{cases} a = 3 & n\log n = n^{1/x} & n < n\log n < n^2 \\ b = 4 & \theta\left(n^{1/x}\right) = \theta(n\log n) \\ k = 1/x \end{cases}$$

۳-روش حریصانه (Greedy)

روش حریصانه، روش آزمند و یا روش دستاوردهای قدم به قدم نیز نامیده می‌شود. این روش جزو رده بزرگتری به نام روشهای بهینه‌سازی است در این روش:

۱- دنباله‌ای از انتخابها را در پیش رو داریم که باید از بین این دنباله انتخابهایی را که منجر به بهینه شدن جواب نهایی گردد انتخاب کنیم.

۲- در هر مرحله از مراحل اجرایی الگوریتم باید بخشی از جواب یا به عبارت درست‌تر مؤلفه‌ای از مؤلفه‌های جواب را به دست آوریم.

۳- نتیجه نهایی یک الگوریتم آزمند مجموعه‌ای از داده‌هاست که ممکن است ترتیب آنها نیز اهمیت داشته باشد. این مجموعه داده‌ها بیشتر مواقع زیرمجموعه داده‌های ورودی است.

۴- جواب نهایی باید تابع هدف یک عبارت را بهینه (ماکزیمم و یا مینیمم) نماید. البته باتوجه به اینکه در روش‌های آزمند آینده‌نگری وجود ندارد و به وضعیت جاری بیشتر توجه می‌شود واضح است که این بهینگی ممکن است کلی نباشد و محلی باشد و به عبارتی max و min فعلی حاصل می‌شود نه ماکزیمم و مینیمم سراسری.

۵- تصمیم اتخاذ شده در مورد انتخاب یا عدم انتخاب یکی از داده‌های ورودی به عنوان مؤلفه‌ای از جواب، قطعی و غیرقابل برگشت است.

۶- در این روش باید بصورت محلی بهترین انتخاب را انجام داد و امیدوار بود که بهترین انتخاب محلی منجر به بهترین انتخاب سراسری نیر خواهد شد (اگر بتوان این موضوع را اثبات کرد میتوان مسئله را به این روش حل کرد).

نکته: مسئله مهم در استفاده از روش حریصانه اثبات مورد پنجم از موارد بالا میباشد.

اجزای الگوریتم‌های حریصانه عبارتند از:

۱- مجموعه اولیه که مؤلفه‌های جواب از بین آنها انتخاب می‌شود.

۲- مجموعه مؤلفه‌هایی که تا به حال انتخاب شده‌اند (به عنوان بخشی از جواب).

۳- تابعی برای انتخاب مؤلفه بعدی.

٤- تابعی برای تعیین اینکه با انتخاب جاری و مجموعه انتخاب‌های قبلی امکان رسیدن به جواب وجود دارد یا خیر.

۵- تابع هدف برای ارزش دهی به جواب مسأله که هدف نهایی بهینه کردن این تابع است.

٦- یک تابع برای بررسی اینکه مشخص کند در نهایت جواب حاصل شده یا خیر.

مسائلی نظیر زمانبندی برنامه‌ها، درخت پوشای مینیمم، ادغام دودویی و بهینه فایلها، انتخاب فعالیتها و ... را میتوان توسط روش حریصانه حل نمود. مدل کلی روش حل مسئله با روش حریصانه بصورت زیر میباشد.

```
function    Greedy(A,n)
            solution  ← ∅
            for  i←1   to  n  do
                 x ← select (A)
                 if  feasible(x,solution)    then
                    solution ←  union (solution,x)
                 endif
            repeat
            return   solution
end .
```

۳-۱- مسأله کوله‌پشتی ساده یا کسری (*Knapsack*)

در این مسئله فرض بر این است که تعداد n کیسه از اجناس مختلف به عنوان ورودی داده شده است. هر کیسه دارای وزن و سود مشخصی است. شخصی با یک کوله پشتی قصد انتخاب این اجناس را دارد. هدف، انتخاب اجناس و قرار دادن آنها در کوله پشتی به گونه‌ای است که سود حاصله بیشینه گردد.

ورودی:

n: تعداد کیسه ها

P_i: سود حاصل از انتخاب کل کیسه i اُم

W_i: وزن کل کیسه i اُم

M: گنجایش کوله پشتی

هدف: بیشینه کردن سود حاصل از انتخاب اجناس یعنی $Max\sum_{i=1}^{n} x_i . p_i$

شرایط مسئله:

وزن همه کیسه‌ها روی هم از وزن کوله‌پشتی بیشتر است زیرا اگر کمتر باشد یعنی می‌توانیم همه را برداریم و انتخابی وجود ندارد همچنین مجموع وزن اجسامی که انتخاب کردیم از وزن کل کوله‌پشتی نباید بیشتر شود.

۱- $$\sum_{i=1}^{n} w_i > M$$

۲- $$\sum_{i=1}^{n} x_i w_i \le M$$

خروجی: کسری از کیسه i اُم که انتخاب می‌شود داخل عنصر i ام آرایه قرار می‌گیرد.

X_i = کسری از کیسه i ام که انتخاب میشود. $(0 \le X_i \le 1)$ $(i=1,\ldots,n)$

راه حل: توسط مثالهای نقضی میتوان نشان داد که انتخاب کیسه ها بصورت صعودی وزن و یا نزولی سود ممکن است منجر به جواب بهینه نگردد! و لذا راه حل بهینه به گونه ای است که باید کیسه ها را بصورت نزولی سود بر وزنشان انتخاب کرد.

راه‌حل بهینه این می‌باشد که سود حاصل از هرکیلو از اجسام را محاسبه کرده و از با ارزش‌ترین جسم شروع به انتخاب کنیم مادامی که کوله‌پشتی فضا برای جسم جدید دارد، در غیراین صورت کسری از آن جسم را انتخاب می‌کنیم.

به بیان دیگر ابتدا اجسام را بصورت $\frac{P_1}{W_1} \ge \frac{P_2}{W_2} \ge \ldots\ldots \ge \frac{P_n}{W_n}$ مرتب میکنیم و سپس به ترتیب انتخاب میکنیم.

در الگوریتم Greedy- knapsack وزن اجسام مرتب شده براساس بیشترین سود هر کیلوی آنها می‌باشد.

در کوله‌پشتی ساده $0 \le x_i \le 1$. (x_i درایه i ام آرایه X میباشد). الگوریتم مورد نظر در زیر بیان شده است.

```
Procedure Greedy- knapsack ( W, n, M, X )
        cu   ←   M
        X  ←  0
        for i←1  to   n   do
                 ┌ if   w(i)  > cu    then
 (feasibility) ┤                      Exit
                 └ endif
                x(i)   ← 1
                cu    ← cu- w(i)
        repeat
        x(i)   ←  cu/w(i)
end
```

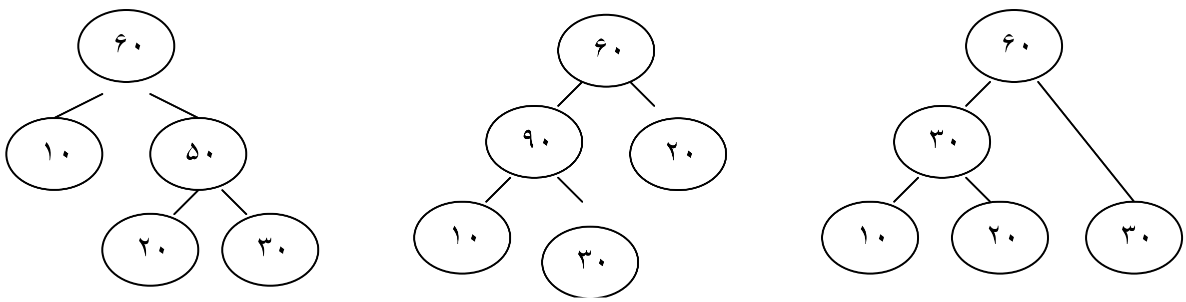
اگر شرط مقابل برقرار نباشد جسم i اُم انتخاب می‌شود

اگر شرط برقرار شود، کسری از جسم i اُم را انتخاب می‌کنیم تا بیشترین سود را داشته باشیم و نیز کوله کاملاً پر شود.

۳-۲- مسئله ادغام دودویی و بهینه فایلها(یا آرایه های مرتب)

یادآوری: میدانیم که برای ادغام دو آرایه یا فایل مرتب که هر کدام به ترتیب دارای m_1 و m_2 رکوردیا عنصر باشند نیاز به زمان $O(m_1+m_2)$ میباشد. حال اگر هزینه ادغام این دو آرایه را برابر با m_1+m_2 در نظر بگیریم. با فرض داشتن n فآیل مرتب، هدف پیدا کردن روشی جهت ادغام دودویی و بهینه (کمترین هزینه) این n فایل میباشد. میدانیم که n فایل را به طرق مختلفی میتوان ادغام کرد که هر طریق ادغام دارای درخت ادغام مخصوص به خود است. الگوریتمی برای ساخت بهترین درخت دودویی ادغام که دارای کمترین مقایسه جهت رسیدن از برگهای درخت به ریشه آن می‌باشد مورد نظر است.

مثال: اگر سه فایل به اندازه های ۱۰، ۲۰ و ۳۰ داشته باشیم به ۳ طریق میتوان آنها را ادغام کرد که کمترین هزینه ادغام برابر با ۹۰ میباشد. در زیر هر سه درخت ادغام را میبینیم:



۱۰+۲۰
۱۰+۲۰+۳۰
۲×۱۰+۲×۲۰+۳۰=۹۰

۱۰+۳۰
۱۰+۳۰+۲۰
۲×۱۰+۲×۳۰+۲۰=۱۰۰

۲۰+۳۰
۲۰+۳۰+۱۰
۲×۲۰+۲×۳۰+۱۰=۱۱۰

سوال: تعداد طرق ادغام n فایل به صورت دودویی را محاسبه کنید:

$A_2=1$

$$A_n=\binom{n}{2}*A_{n-1}=\binom{n}{2}\binom{n-1}{2}\binom{n-2}{2}......\binom{2}{2}=$$

$$=\frac{n!\ (n-1)!\ (n-2)!\ ....2!\ 3!\ 2!}{2!\ (n-2)\ !\ 2!\ (n-3)\ !\ 2!\ (n-4)\ !\ ....2!\ 2!\ 1!\ 2!\ n!}=\frac{n!\ (n-1)!}{2^{n-1}}$$

لذا اگر بخواهیم همه درختهای ادغام را تولید کرده و کم هزینه ترین آنها را انتخاب کنیم هزینه بسیار زیادی را باید پرداخت کرد! لذا نیاز به الگوریتمی است که در زمان چند جمله‌ای بهترین درخت ادغام را تولید کند.

الگوریتم ساخت درخت دودویی با کمترین هزینه:

تعریف: فاصله از ریشه $dist(v_i)=level(v_i)-1$

تعریف: مقدار هزینه‌ای که باید صرف کرد تا n فایل بصورت دودویی ادغام شوند (WEPL=Weighted External Path Length).

$$Wep1=\sum_{i=1}^{n} dist(v_i)*\cos t(v_i)$$

توضیح الگوریتم: برای بدست آوردن درخت دودویی با کمترین هزینه یک نود جدید میسازیم و سپس کم ارزش‌ترین نود را انتخاب کرده و به عنوان فرزند چپ (left child) گره جدید درج میکنیم و آن را از لیست حذف می‌کنیم سپس دوباره همین کار را برای فرزند سمت راست(rightchild) انجام می‌دهیم. سپس گره جدید را به لیست اضافه میکنیم. تا مادامی این کار را انجام می‌دهیم که همهٔ نودهای لیست (به جز آخری) از لیست حذف شوند. کد الگوریتم مورد نظر را در زیر می‌بینیم:

```
Function construct_tree(L,n)
   For i←1 to  n-1  do
     Get node(T)
     Lchild(T)← least(L)
     Rchild(T) ← least(L)
     Cost(T) ← cost(Lchild(T))+cost(Rchild(T))
      Insert(L,T)
   Repeat
   Return least(L)
end .
```

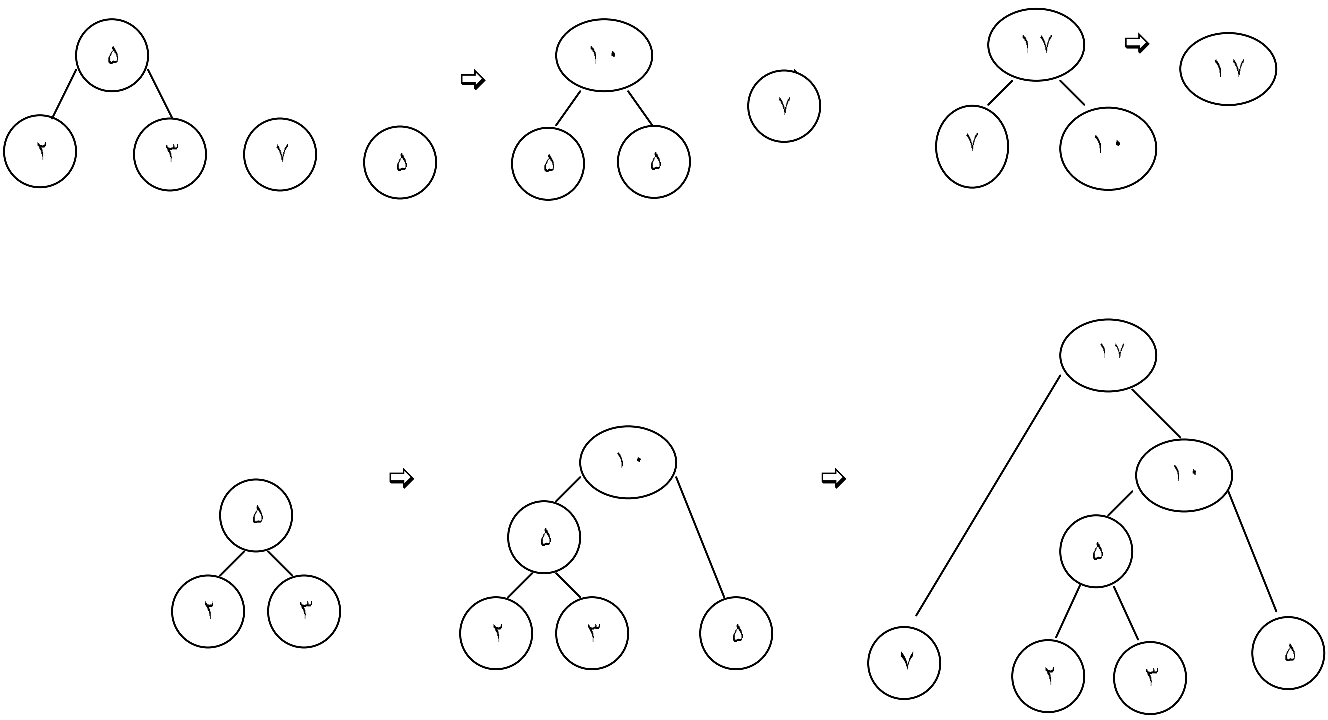
رویه least(L): در لیست کمترین مقدار را پیدا کرده آن را حذف کرده و آدرسش را برمی‌گرداند.

Select: هر بار دو min را انتخاب میکند.

Feasibility: ندارد

Union: بعد از پیوند به گره جدید به لیست اولیه اضافه می‌کند.

شکل زیر روند ساخته شدن درخت دودویی ادغام بهینه را بر روی داده های ورودی ۲، ۳، ۵، ۷ و ۱۰ نشان میدهد.



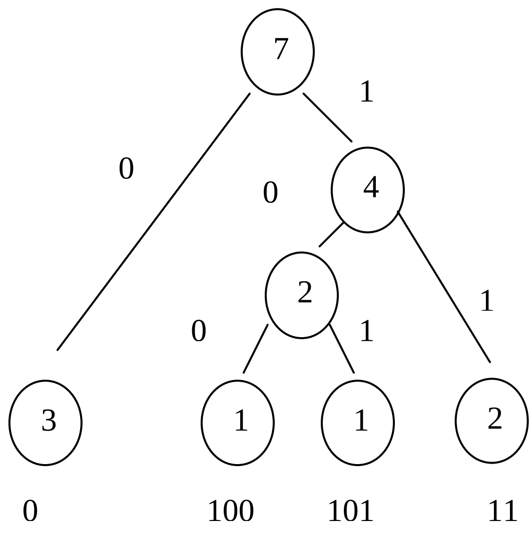
٣-٣- کدینگ **Huffman**

یکی از کاربردهای مسئله ادغام دودویی فایلها در کدینگ هافمن است که جهت فشرده سازی فایلها یا ارسال کم حجمتر اطلاعات بر روی خطوط شبکه مورد استفاده قرار میگیرد. در ابتدا یک جدول درنظر می‌گیریم در ستون ابتدایی مقادیر (داده‌ها) را وارد می‌کنیم در ستون دوم تعداد تکرار داده‌ها را محاسبه کرده و براساس آن درخت دودویی را طبق الگوریتم Constract-tree می‌سازیم سپس یال‌های چپ را صفر و یال‌های راست را یک در نظر می‌گیریم در ستون آخر جدول کدهای جدید را که برچسب یالهای درخت حاصله از ریشه تا هر برگ است را درج می‌کنیم. این برچسبها را کدینگ هافمن مینامیم. برچسبهای جدید کدهای جایگزین کدهای قبلی خواهند شد که در فشرده سازی فایلها حجم کمتری را به خود اختصاص میدهد. جهت بازگرداندن فایل به حالت اولیه نیز از برعکس همین روش میتوان استفاده کرد(بدیهی است که باید اطلاعاتی در فایل فشرده شده قرار داد که از روی آنها بتوان مجدداً درخت کدینگ را ساخت)!.

فایل اولیه						
١٠٠	٠١٠	١٠٠	٠١١	٠١١	١١٠	١٠٠

داده	تعداد تکرار	کدینگ هافمن
٠٠٠	٠	----
٠٠١	٠	----
٠١٠	١	١٠٠
٠١١	٢	١١
١٠٠	٣	٠
١٠١	٠	----
١١٠	١	١٠١
١١١	٠	----

فایل نهایی						
٠	١٠٠	٠	١١	١١	١٠١	٠



۳-۴- درخت پوشای مینیمم

مسئله پیدا کردن درخت پوشای با کمترین هزینه بر روی یک گراف همبند وزندار از جمله مسائل پـر کـاربردی در الگوریتمهای گراف محسوب میشود. گراف همبند وزندار G(V,E) مفروض است، هدف پیدا کردن زیرگرافـی از G است که دارای خواص زیر باشد:

۷- شامل همه گره های V باشد

۸- همبند و بدون حلقه باشد

۹- در بین کلیه زیرگرافهایی که شرط ۱ و ۲ را دارند دارای کمترین هزینه باشد

برای این مسئله الگوریتمهای متعددی وجود دارد که از بین آنها به بیان دو الگوریتم که روش طراحی آنها حریصـانه است میپردازیم.

نکته: اگر گراف ورودی همبند نباشد، دارای درخت پوشا نیست.

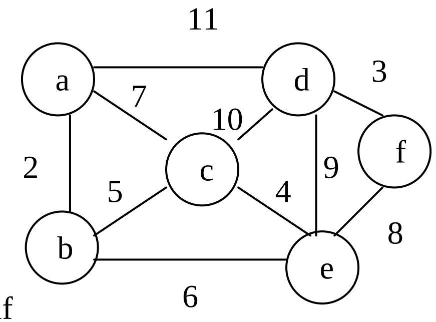
نکته: درخت پوشای یک گراف با n راس دقیقاً دارای n-1 یال میباشد. اگر یک یال به آن اضافه کنیم شـرط بـدون حلقه بودن آن نقض میشود و اگر یک یال از آن حذف کنیم شرط همبنـد بـودن آن نقـض خواهـد شـد. در نتیجـه میتوان درخت پوشای یک گراف با n راس را به این صورت تعریف کرد: زیر درختی از گراف کـه دارای n-1 یـال میباشد.

۳-٤-۱-الگوریتم راشال (kruskal)

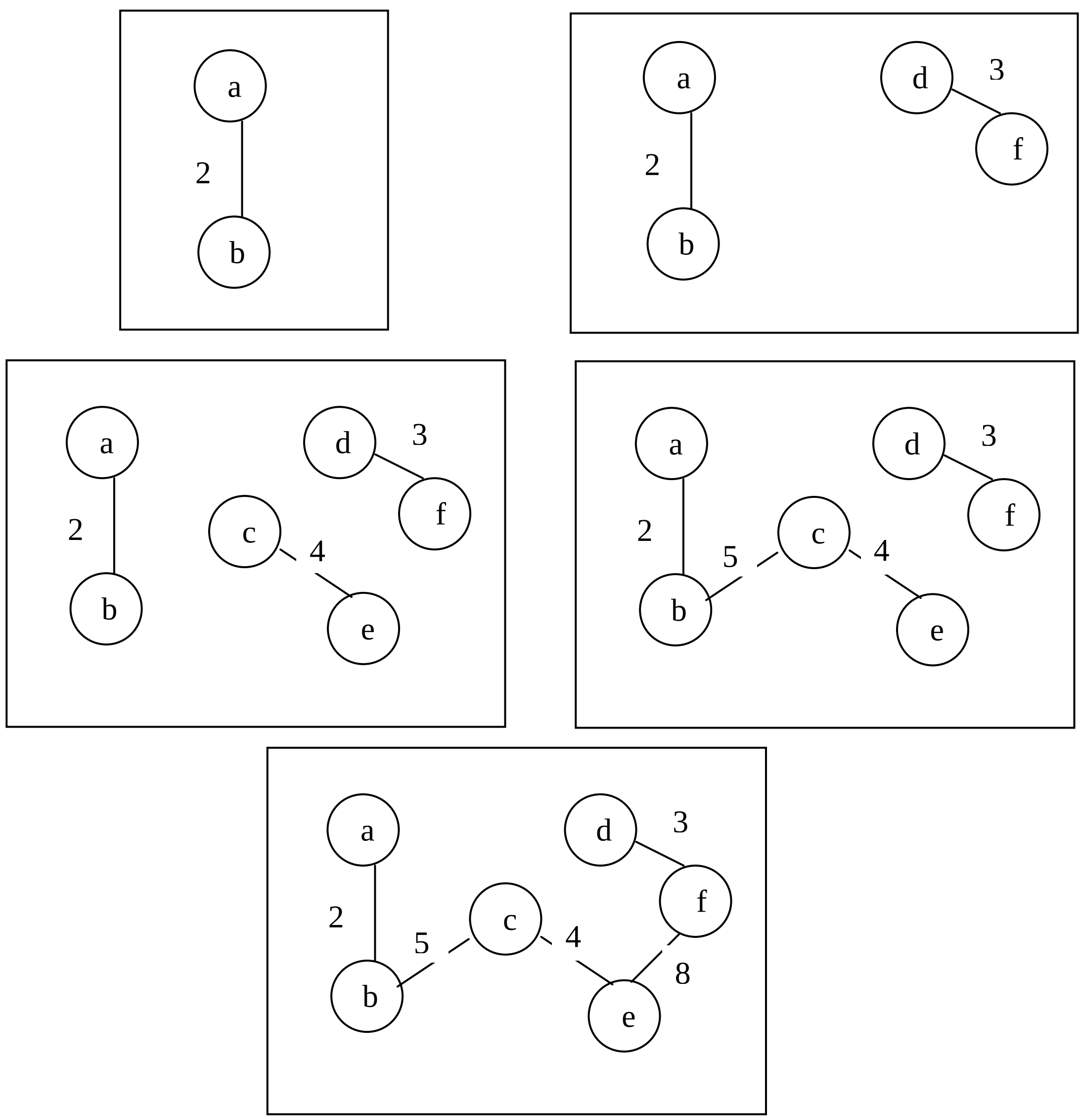
در این الگوریتم در هر مرحله یک یال با کمترین هزینه از گراف ورودی انتخاب شده و از گراف حذف شـده و در صورتیکه اضافه کردن آن به درخت موجب ایجاد حلقه نشود آن را به درخت مورد نظـر (کـه در ابتـدای الگـوریتم

خالی است) اضافه میکنیم. این عمل تا زمانیکه تعداد n-1 یال به درخت اضافه نشده است ادامه میابد. شبه کد الگوریتم را در شکل زیر میبینیم.

```
Procedure kruskal
   T  ←  ∅
   While ( |T| < n-1 ) and ( |E| ≠ 0 )   do
      Select   edge e    from E with minimum cost
      Delete   e  from   E
      If  e  does not  create a cycle in T then
          add e  to T
      endif
   repeat
   if |T| < n-1   then write ('no spanning Tree ') endif
end.
```



مراحل ساخت درخت:



مرتبه زمانی الگوریتم در بهترین پیاده سازی برابر با O(e. log n) میباشد!.

۳-٤-۲-الگوریتم Prim

الگوریتم پریم، مانند الگوریتم راشال در هر زمان یک لبه از درخت پوشای با کمترین هزینه را می‌سازد. در الگوریتم پریم از یک راس دلخواه شروع کرده و در هر مرحله یال با کمترین هزینه که یک راس مجاور آن در درخت بوده و راس دیگرش در درخت نباشد را به آن اضافه میکنیم. در هر مرحله، مجموعه لبه‌های انتخاب شده یـک درخـت را تشکیل می‌دهند و در هر مرحله درخت گسترش میابد. درمقابل، مجموعه لبه‌های انتخاب شده در الگـوریتم راشـال در هرمرحله یک جنگل را تشکیل می‌دهند. شبه کد الگوریتم را در شکل زیر میبینیم.

```
Procdure prim(v,E,T)
   T←∅
   Tv←{1}
   While |T| < n-1    do
      Select e= (u,v)    with   minimum  cost   from  E such that u∈Tv and v∉Tv
      if there is no any such edge then                يك رأس يال موردنظر در داخلTv باشد و رأس ديگر نباشد.
           exit                                        اين شرط در صورتي برقرار مي‌شود كه گراف اصلي ما همبند نباشد.
      endif
      add  e  to   T
      add  v  to   Tv
   repeat
   if |T|<n-1    then  write  ('no spanning tree')  endif
end.
```

مرتبه زمانی الگوریتم در بهترین پیاده سازی برابر با O(e. log n) میباشد!.

دراین الگوریتم:

هزینه کمتر با شرایط خاص   : Select

به علت نحوه انتخاب یال‌ها وجود ندارد :   feasibility

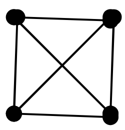
۳-٤-۳-مقایسه الگوریتم Kruskal و Prim

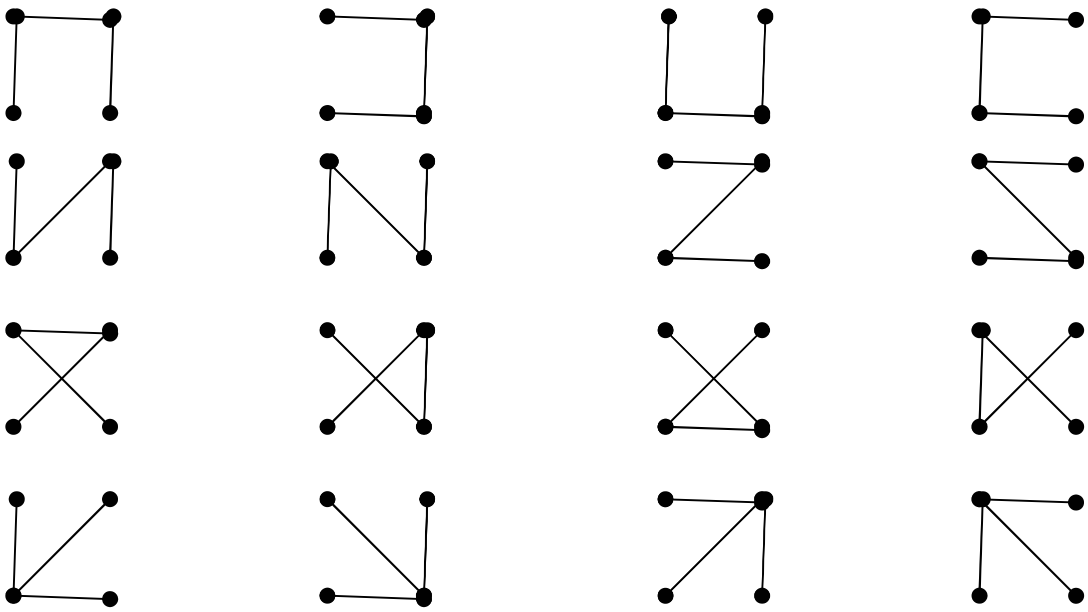
دو الگوریتم prim و kruskal درصورتی دو درخت متفاوت تولید میکنند که چندین یال با هزینـه‌هـای مسـاوی داشته باشیم ولی همیشه هزینه درختهای تولید شده برابر و کمینه است. در الگوریتم Prim در ابتدا یک گره بوده و کم کم در حین الگوریتم گسترش میابد تا به درخت پوشای کمینه تبدیل شود درحالیکه در الگوریتم kruskal چند درخت (جنگل) وجود دارد که در انتهای الگوریتم درختهای جنگل به هم پیوند خورده تا تبدیل به درخـت پوشـای کمینه شوند.

۳-٤-٤-تعداد درختهای پوشای K_n

در این بخش به بیان قضیه مهمی در مورد تعداد درختهای پوشا میپردازیم.

مثال: درختهای پوشای K_4 را بدست آورید:





قضیه: ثابت کنید تعداد درختهای پوشای k_n (گراف کامل با n راس) برابر n^{n-2} است.

اثبات: از روش زیر برای اثبات قضیه بالا استفاده می‌کنیم:

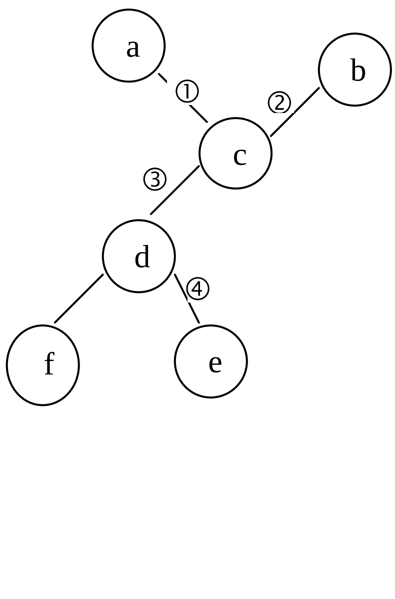
ابتدا نشان میدهیم که هر درخت پوشا با n رأس در تناظر یک به یک با یک رشته n-2 حرفی روی n حرف است و سپس چون تعداد رشته های n-2 حرفی روی n حرف برابر با n^{n-2} است قضیه ثابت میشود.

باید از هردرخت پوشا به یک رشته برسیم و برعکس. با الگوریتم زیر میتوان برای هر درخت پوشا یک رشته n-2 حرفی یکتا تولید کرد.

۱- به ازای i از ۱ تا n-2 مرحله ۲ را تکرار کنید:

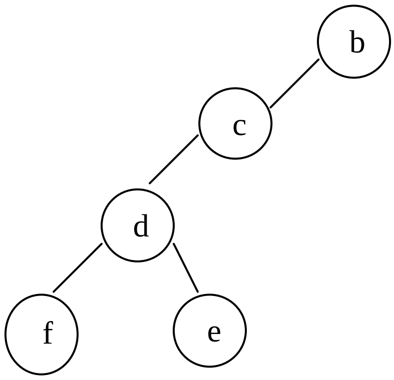
۲- کوچکترین (از لحاظ ترتیب الفبایی یا عددی) برگ (راس با درجه ۱) را از درخت حذف کرده و راس مجاور یال مربوط به آن را به عنوان i امین عنصر رشته محسوب میکنیم.

۱	۲	۳	٤
c	c	d	d



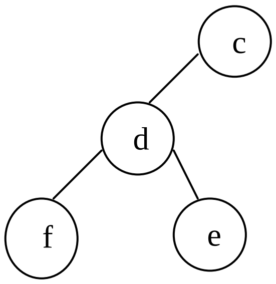
مرحله اول:

c			



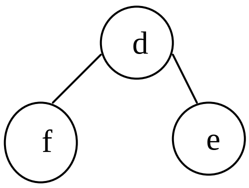
مرحله دوم:

c	c		



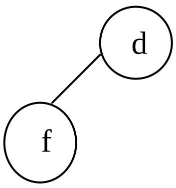
مرحله سوم:

c	c	d	



مرحله چهارم:

c	c	d	d



عکس روش بالا (رسیدن از رشته به درخت):

باتوجه به مثال بالا طول رشته موردنظر را با دو جمع کرده تعداد رئوس بدست می‌آید سپس در مرحله بعد جدولی از کلیه حروف تهیه می‌کنیم حال تعداد تکرار هر حرف در رشته را به اضافه یک می‌کنیم و در جدول قرار

می‌دهیم(قابل ذکر است که درجه هر گره در درخت پوشا برابر با یکی بیشتر از تعداد دفعات ظهـور گـره در رشـته است).

۱- به ازای i از ۱ تا n-2 مرحله ۲ و ۳را تکرار کنید:

۲- کوچکترین راس با درجه ۱ در جدول را انتخاب کرده و یالی بین آن راس و حرف i ام رشته برقرار کنید.

۳- از درجه دو راسی که در مرحله ۲ بین آنها یالی برقرار شد یک واحد کم کنید

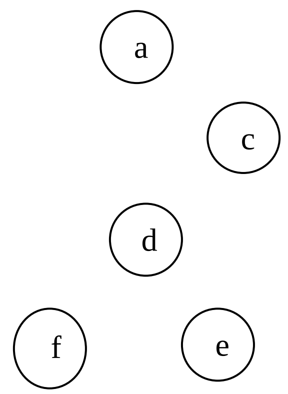
۴- یالی بین دو راس باقیمانده برقرار کنید.

مثال:

a	b	c	d	e	f
1	1	3	3	1	1

c	c	d	d

شروع کار:

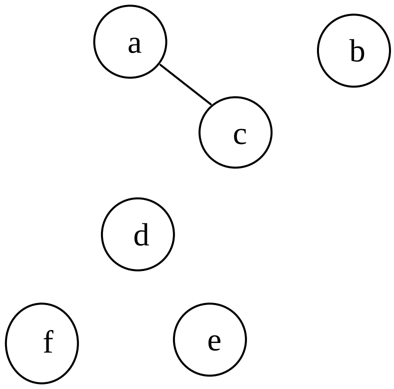


مرحله اول:

با حرکت از سررشته (کهc می‌باشد) و باتوجه به جدول کوچکترین برگ را (a) انتخاب می‌کنـیم حـال از درجـه c یکی کم کرده و همینطور از درجه برگ که(a) می‌باشد و این کار را برای مراحل بعد نیز انجام می‌دهیم.

a	b	c	d	e	f
1 0	1	3 2	3	1	1

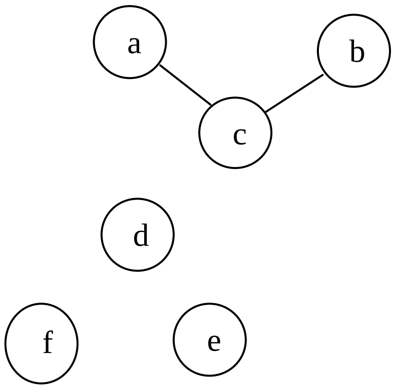
c	c	d	D



مرحله دوم:

a	b	c	d	e	f
1 0	1 0	3 2 1	3	1	1

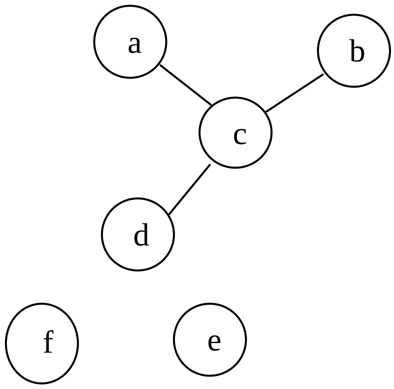
c	c	d	d



مرحله سوم:

a	b	c	d	e	f
1 0	1 0	3 2 1 0	3 2	1	1

c	c	d	d

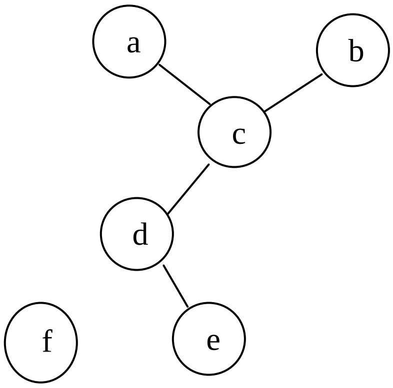


مرحله چهارم:

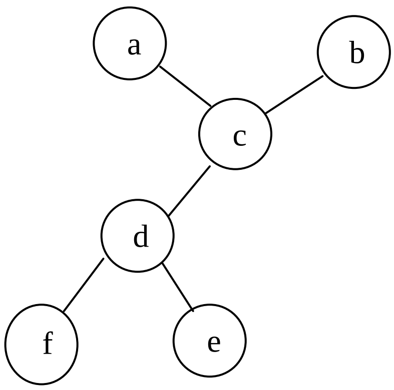
در این مرحله دو نود انتهایی را به یکدیگر وصل می‌کنیم.

a	b	c	d	e	f
1 0	1 0	3 2 1 0	3 2 1	1 0	1

c	c	d	d



مرحله آخر:



۳-۵- کوتاهترین مسیرهای هم مبدا

در این قسمت به بررسی مسئله کوتاهترین مسیرهای هم مبـدا (Single Source Shortest Paths) یـا SSSP میپردازیم. الگوریتمهای متعددی برای محاسبه طول کوتاهترین مسیر بین دو گره در گراف وزندار وجود دارد که در این میان الگوریتم Dijkstra به روش حریصانه طراحی شده است لذا در این قسمت به بیان آن میپردازیم. در ایـن الگوریتم هدف پیدا کردن طول کوتاهترین مسیر از یک گره مبدا نظیـر v بـه گـره‌هـای دیگـر مـی‌باشـد. الگـوریتم دایکسترا بر روی گرافهای وزنداری قابل اجراست که هزینه یالها نامنفی باشد.

فرضیات:

طول کوتاهترین مسیر از راس v به راس i	Dist[i]:
تعداد رئوس	n:
مبدأ:	V:
ماتریس هزینه‌ها:	cost:
راسی قبل از راس i بر روس کوتاهترین مسیر	p[i]:

در این الگوریتم کمترین فاصله مبدا به هر راس کم کم محاسـبه شـده و در هـر مرحلـه کمتـر میشـود تـا در پایـان الگوریتم به کمترین حد خود برسد. در حین اجرای الگوریتم هر راسی که فاصله مبدا بـه آن بطـور کامـل محاسـبه شده بود و مقدار آن به کمترین حد خود رسیده باشد برچسب دائمی شدن (s[w]=1) خواهـد خـورد. در ابتـدای کار همه رئوس به غیر از مبدا دارای برچسب موقتی هستند(s[w]=0).

توضیح: الگوریتم دارای ۲ فاز میباشد(مرحله ۲ و ۳):

۱- مراحل ۲ و ۳ را n-2 بار تکرار کنید:

۲- در هر مرحله از میان رئوسی که هنوز برچسب دائمی شدن نخورده‌انـد راسـی کـه دارای کمتـرین فاصـله نسبت به مبدا است (کمترین Dist) را انتخاب میکنیم و آن را برچسب دائمی میزنیم.

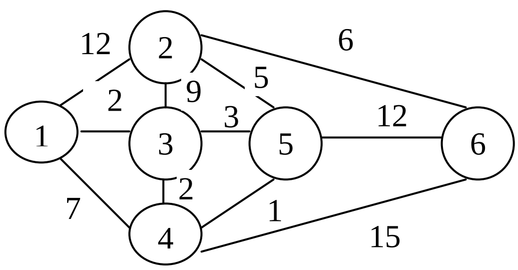
۳- مقدار Dist مابقی رئوسی را که هنوز موقتی هستند با توجه به راس دائمی شده در مرحله ۱ بروز میکنیم.

شبه کد الگوریتم در زیر آورده شده است:

```
Procedure (G, n, v, cost, dist)
   For   i←1    to    n   do
      dist[i]  ←  cost (v,i)
      p[i] ←  v
      s[i] ←   false
   repeat
   s[v]  ← True
   for  i←1  to  n-2  do
      select  vertex  u  such  that  dist (u)= min {dist(w) | s(w)= false}
      s[u]  ←  true
      for all vertices  w  in which  s(w) = false  do
         if  dist (u) +cost (u,w) <dist (w)  then
            dist (w)  ← dist (u) + cost (u,w)
            p[w]  ←  u
         end if                                              (*)
      repeat
   repeat
end.
```

مثال:

اگر در گراف زیرا راس شماره ۱ را به عنوان مبدا انتخاب کنیم آنگاه الگـوریتم تغییراتـی را در آرایـه Dist و P بـا توجه به جداول زیرا بوجود خواهد آورد. باتوجه به ستون dist و p میتوان کوتاهترین مسیرها را رسم کرد.



مرحله شروع:

Vertex	S	Dist	P
1	1	0	1
2	0	12	1
3	0	2	1
4	0	7	1
5	0	∞	1
6	0	∞	1

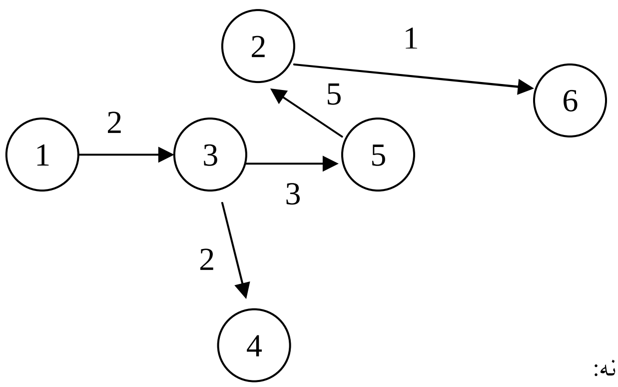
Vertex	S	Dist	P
1	1	0	1
2	0	11	3
3	1	2	1
4	0	4	3
5	0	5	3
6	0	∞	1

Vertex	S	Dist	P
1	1	0	1
2	0	11	3
3	1	2	1
4	1	4	3
5	0	5	3
6	0	21	4

Vertex	S	Dist	P
1	1	0	1
2	0	10	5
3	1	2	1
4	1	4	3
5	1	5	3
6	0	17	5

Vertex	S	Dist	P
1	1	0	1
2	1	10	5
3	1	2	1
4	1	4	3
5	1	5	3
6	0	16	2

در سطر۲ آرایه p رأس ۵ قرار دارد، و این بدان معنی است که کمترین هزینه برای رفتن به رأس۲ از طریق رأس ۵، می‌باشد در نتیجه کوتاهترین مسیر از راس ۱ به راس ۲ بصورت ($2\rightarrow5\rightarrow3\rightarrow1$) میباشد. از روی آرایـه p میتـوان درختی تولید کرد که به نام درخت پوشای کوتاهترین مسیرها به مبدا (ریشه) v معروف است (SPST)(اگر گـراف اولیه همبند باشد درخت حاصل از کوتاهترین مسیرها به ریشـه v نیـز پوشـا اسـت). شـکل زیـر درخـت پوشـای کوتاهترین مسیرها به مبدا راس ۱ را نشان میدهد.



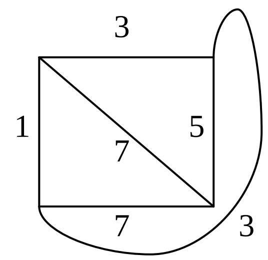
تطابق با روش حریصانه:

Select: انتخاب راس موقتی با کمترین Dist

Feasibility: -------------

Union: در برنامه (×)

نکته: با در نظر گرفتن هر یک از رئوس به عنوان مبدا و اجرای الگوریتم بیان شده میتوان به n درخت پوشای کوتاهترین مسیرها رسید که ممکن است هیچ کدام از آنها درخت پوشای کمینه نباشد. گراف زیر نشان دهنده این موضوع میباشد.



تحلیل زمانی: براحتی ملاحظه میشود که الگوریتم بالا را میتوان بصورتی پیاده سازی کرد که مرتبه زمانی آن $O(n^2)$ گردد!.

۳-۶- انتخاب بهینه فعالیتها (*Activity Selection*)

مسئله دیگری که برای آن میتوان به راحتی الگوریتمی به روش حریصانه تولید کرد، مسئله انتخاب فعالیتها نام دارد. فرض کنید n سخنران وجود دارند که هر کدام زمان شروع و پایان سخنرانی خود را به عنوان ورودی مسئله اعلام کرده‌اند. هدف انتخاب بیشترین تعداد سخنران به گونه‌ای است که هیچ دو سخنرانی با هم اشتراک بازه زمانی نداشته باشند.

نکته: جهت عدم تداخل در زمان شروع یا پایان میتوان بدون کم شدن از کلیت مسئله فرض کرد که پایان بازه سخنرانیها باز میباشند یعنی بصورت (…] میباشند.

ورودی:

S_i: شروع فعالیت i اُم

F_i: پایان فعالیت i اُم

n: تعداد فعالیتها

هدف: انتخاب بیشترین تعداد فعالیت به قسمی که اشتراک بازه زمانی نداشته باشند.

توضیح الگوریتم:

۱- درابتدا فعالیتها را براساس زمان پایان آنها مرتب کردهو سپس اولین فعالیت (زودترین زمان پایان) را انتخاب میکنیم.

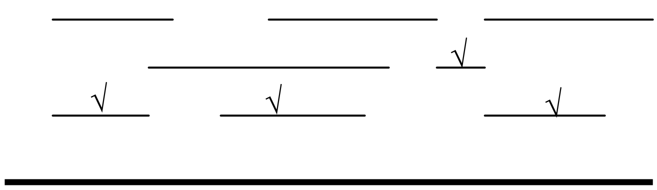
۲- به ازای i از ۲ تا n مرحله ۳ را تکرار میکنیم:

۳- اگر فعالیت i ام با آخرین فعالیت انتخاب شده تداخل بازه زمانی نداشت آن را انتخاب میکنیم.

شبه کد الگوریتم را در زیر میبینیم(فرض بر این است که فعالیتها بر حسب صعودی زمان پایانشان مرتب شده‌اند):

```
Procedure Activity_selector (A, S, F, n)
   A← {1}
   j←1
   for  i←2   to n   do
      if  Si ≥ Fj then
         A   ←  A ∪ {i}
         j← i
      endif
   repeat
end.
```

شکل زیر نمونه ای از یک مثال را نشان میدهد.



تطابق با روش حریصانه:

select : فعالیتی که زودترین زمان پایان را داشته باشد

feasibility: با آخرین فعالیت انتخاب شده تداخل زمانی نداشته باشد

Union: اضافه کردن آن به مجموعه انتخاب شده ها

تحلیل زمانی: بیشترین زمان در الگوریتم مربوط به مرتب سازی ابتدای کار است و لذا الگوریتم دارای مرتبه زمانی $O(n \log n)$ می باشد.

مثال:

selection	s_i	f_i
✓	1	4
-	3	5
-	0	6
✓	5	7
-	3	8
-	5	9
-	6	10
✓	8	11
-	8	11
-	2	13
✓	12	14

۴-روش تقسیم و حل (Divide & Conquer)

در این روش ابتدا از مسأله اصلی شروع کرده و آن را به مسائل کوچکتر تقسیم می‌کنیم. سپس هر یک از زیر مسائل را بصورت بازگشتی حل کرده و جواب حل زیر مسائل را با هم ترکیب میکنیم. این روش منطبق بر الگوی بالا به پایین می‌باشد(Top Down)، به این معنی که فضای حل مسئله از بالا به پایین ساخته شده تا به کوچکترین زیر مسئله برسیم و سپس از کوچکترین زیر مسائل حل شده و با هم ترکیب میشوند. مشکل این روش در این است که ممکن است زیر مسائل تکراری محاسبه شوند و لذا منجر به بالا رفتن زمان حل مسئله گردد. در هنگام تقسیم، مسأله بزرگتر به مسائل کوچکتر شکسته می شود اما همیشه اینطور نیست، باید تعداد تقسیمات را به گونه ای گرفت که کارایی بیشترین مقدار باشد، هرچه اندازه مسائل کوچک به هم نزدیکتر باشد معمولاً کارآیی حاصل بیشتر است. بنابراین سعی می‌شود اندازه زیر مسائل تقریباً با هم مساوی باشند. شبه کد کلی روند حل مسائل در روش تقسیم و حل بصورت زیر میباشد:

```
Procdure D&C (p,q)
    if   small (p,q)   then
         return G(p,q)
    else
       m← divide (p,q)
      return combine (D&C(p,m) , D&C(m+1,q))
    endif
end.
```

برای ارائه الگوریتم در این روش باید جزئیات توابع small ، G ، divide و combine را مشخص کرد. Small تصمیم گیری در مورد کوچک بودن اندازه مسئله را انجام میدهد به این معنی که در این تابع مشخص میشود که اگر اندازه مسئله به حد کافی کوچک است که دیگر نیازی به فراخوانی بازگشتی الگوریتم نباشد عمل تقسیم مسئله به زیر مسائل پایان میابد. تابع G مسئله با اندازه کوچک را حل میکند. تابع divide وظیفه تقسیم مسئله به زیر مسائل کوچکتر را به عهده دارد و تابع combine هم ترکیب جواب زیر مسائل را انجام میدهد.

۴-۱- محاسبه عنصر کمینه و بیشینه یک آرایه

توسط یک روش تقسیم و حل میتوان عنصر کمینه و بیشینه یک آرایه را محاسبه کرد. کد این الگوریتم در زیر آورده شده است:

```
Procedure MinMax(A, l, u, min, max)
   if l=u then
      min←A(l)
      max←A(l)
   elsif l+1=u then
      if A(l)<A(u) then
         min←A(l)
         max←A(u)
      else
         min←A(u)
         max←A(l)
      endif
   else
      mid ← (l+u)/2
      call MinMax(A, l, mid, min1, max1)
      call MinMax(A, l, mid+1, min2, max2)
      if min1<min2 then
         min←min1
      else
         min←min2
      endif
      if max1>max2 then
         max←max1
      else
         max←max2
      endif
   endif
end.
```

$$T(n)=\begin{cases} 0 & ;n=1 \\ 1 & ;n=2 \\ 2T(\frac{n}{2})+2 & ;n>2 \end{cases}$$

$$\Rightarrow T(n)=2[2T(\frac{n}{4})+2]+2=4T(\frac{n}{4})+4+2=2^2T(\frac{n}{2^2})+2^2+2^1$$

$$=2^iT(\frac{n}{2^i})+2^i+2^{i-1}+\cdots+2^2+2^1$$

$$\overset{n=2^k}{=} 2^{k-1}T(2)+2^{k-1}+\cdots+2^1$$

$$=\underbrace{2^{k-1}}_{\frac{n}{2}}+\underbrace{2^{k-1}+\cdots+2^1}_{2^k-2}=\frac{n}{2}+n-2=\frac{3n}{2}-2$$

٤-٢- ضرب دو ماتریس به روش استراسن(Strassen)

در ضرب دو ماتریس n×n، خروجی حاصل یک ماتریس n×n می‌باشد، که این ماتریس دارای n^2 عضو است که برای هر عضو به n ضرب نیاز داریم پس تعداد ضربهای مورد نیاز برای ضرب دو ماتریس n×n، $O(n^3)$ می باشد.
در این قسمت ضرب دو ماتریس n×n را به وسیله روش تقسیم و حل انجام می‌دهیم، که مرتبه زمانی این روش هم $O(n^3)$ خواهد بود.
ابتدا هر ماتریس ورودی را از سطر و ستون به دو قسمت تجزیه میکنیم تا هر ماتریس ورودی به ۴ زیر ماتریس تجزیه شود.

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}\begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}=\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

هر کدام از زیر ماتریسها دارای n/2 سطر و n/2 ستون است که نشان میدهد هر زیر ماتریس دارای $n^2/4$ درایه میباشد. با کمی تامل میتوان دریافت کـه توسط روش زیر میتوان ۴ قسمت از ماتریس حاصلضرب را تولید کرد که این روش نیاز به ۸ فراخوانی بازگشتی تابع دارد و لذا دارای مرتبه زمانی $O(n^3)$ میباشد.

$$C_{11}=A_{11}*B_{11}+A_{12}*B_{21}$$
$$C_{12}=A_{11}*B_{12}+A_{12}*B_{22}$$
$$C_{21}=A_{21}*B_{11}+A_{22}*B_{21}$$
$$C_{22}=A_{21}*B_{12}+A_{22}*B_{22}$$
$$T(n)=8T\left(\frac{n}{2}\right)+4*\frac{n^2}{4}=0\left(n^{\log_2^8}\right)=n^3$$

روش استراسن: توسط یک ایده جالب میتوان معادلات بالا را (که نیاز به ۸ ضرب ماتریسی و ۴ جمع ماتریسی داشت)به گونه دیگری بازنویسی کرد که نیاز به ۷ ضرب ماتریسی و ۱۸ جمع ماتریسی داشته باشد.
۸ ضرب را به ۷ ضرب و ۱۸ جمع تبدیل می‌کند.

$$P_1=A_{11}(B_{12}-B_{22})$$
$$P_2=(A_{11}+A_{12})B_{22}$$
$$P_3=(A_{21}+A_{22})B_{11}$$
$$P_4=A_{22}(B_{21}-B_{11})$$
$$P_5=(A_{11}+A_{22})(B_{11}+B_{22})$$
$$P_6=(A_{12}-A_{22})(B_{21}+B_{22})$$
$$P_7=(A_{11}-A_{21})(B_{11}+B_{12})$$
$$C_{11}=P_5+P_4-P_2+P_6$$
$$C_{12}=P_1+P_2$$
$$C_{21}=P_3+P_4$$
$$C_{22}=P_5+P_1-P_3-P_7$$

بنابراین:

$$T(n)=7\,T\left(\frac{n}{2}\right)+18\frac{n^2}{4}=O\left(n^{\log_2^7}\right)=O\left(n^{2.81}\right)$$

٤-٣- تعیین نزدیکترین زوج نقاط

٤-٣-١-تعیین نزدیکترین زوج نقاط در فضای یک بعدی

درفضای یک بعدی فقط محورx را داریم که بر روی این محور تعداد n نقطه به عنوان ورودی داده شده است. و هدف پیدا کردن کمترین فاصله مابین نقاط است.

الگوریتم اول:

در این روش تک تک نقاط را با بقیه نقاط بررسی کرده و فاصله آنها را محاسبه کرده و بین فواصل کمترین را محاسبه میکنیم که در نتیجه نیاز به $O(n^2)$ محاسبه دارد.

$$o\left(n^2\right)\leftarrow(n-1)+(n-2)+......+1$$

الگوریتم دوم:

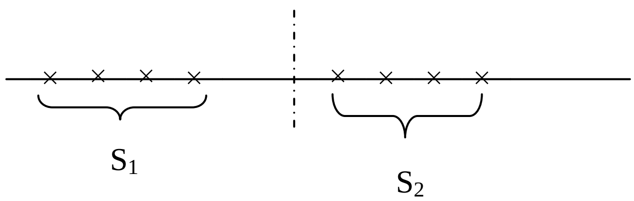
میتوان ابتدا نقاط را بر حسب مختصه x شان مرتب کرده و سپس بین فواصل نقاط متوالی کمترین فاصله را محاسبه کرد که در نتیجه الگوریتم دارای مرتبه زمانی O(n. log n) (به خاطر مرتب سازی) میباشد.

الگوریتم سوم(روش تقسیم و حل):

١- نقاط را مرتب کنید
٢- n/2 نقاط سمت چپ(p_i; i=1,…,n/2) را بصورت بازگشتی حل کنید(کمترین فاصله بین آنها را s_1 مینامیم)
٣- n/2 نقاط سمت راست(q_i; i=1,…,n/2) را بصورت بازگشتی حل کنید(کمترین فاصله بین آنها را s_2 مینامیم)
۴- کمترین فاصله نهایی از رابطه زیر قابل تولید است:

$$\min=\{s_1,s_2,\min\{q_i\}-\max\{p_i\}\}$$

$$T(n)=2T\left(\frac{n}{2}\right)+1=o(n)$$



درکل *o(n log n)* میشود (بادرنظر گرفتن *sort*)

٤-٣-٢-تعیین نزدیکترین زوج نقاط در فضای دوبعدی

الگوریتم:

۱- درابتدا نودها را در این فضا براساس مختصه x شان مرتب می‌کنیم.

۲- سپس با در نظر گرفتن یک خط فرضی مجموعه نقاط را به دو قسمت چپ و راست که هر کدام دارای n/2 عنصر هستند تقسیم میکنیم.

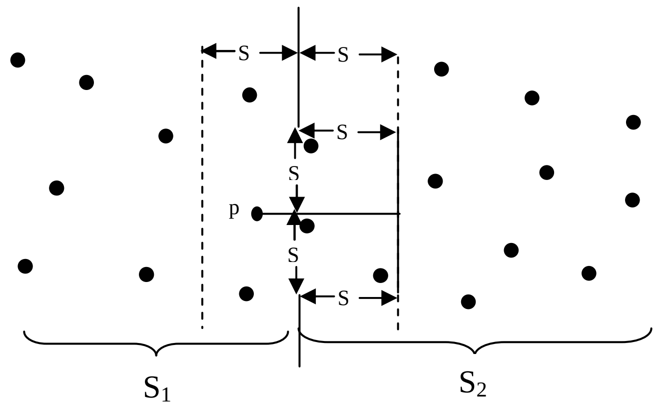
۳- دو زیر مسئله چپ و راست را حل کرده تا کمترین فاصله در هر دو زیر مسئله تولید شود و سپس جواب حل دو زیر مسئله را S_1 و S_2 مینامیم.

۴- $S=\min(S_1,S_2)$

۵- حال اگر شبیه روش یک بعدی فاصله نزدیکترین نقطه از نقاط چپ و راست خط را نیز در محاسبات دخیل کنیم ممکن است به جواب صحیح نرسیم. لذا در این مسئله با اضافه کردن یکسری شرایط سعی می‌کنیم تعداد مقایسه ها را کم کنیم. میتوان به اندازه S از خط وسط از هر دو طرف یک محدوده در نظر گرفت و نقاط داخل این باریکه را با هم مقایسه کرد که در نتیجه باز در بدترین شرایط تمام n/2 نقطه سمت چپ و n/2 نقطه سمت راست خط وسط در این باریکه ها قرار دارند و لذا مرتبه زمانی $O(n^2)$ خواهد شد ولی میتوان برای هر نقطه واقع در باریکه سمت چپ مانند p فقط نقاط واقع در مستطیل خاصی را در نظر گرفت. این مستطیل به این صورت ساخته میشود که باید پای عمود از نقطه نسبت به خط وسط را پیدا کرده از آن به اندازه S به بالا، پایین و راست حرکت کرد تا یک ناحیه مستطیلی شکل به طول 2S و عرض S حاصل شود. بدیهی است که حداکثر تعداد نقاط واقع در این ناحیه مستطیلی شکل برابر با ۶ نقطه است! و لذا برای هر نقطه در باریکه سمت چپ فقط باید فاصله آن را با ۶ نقطه در باریکه سمت راست محاسبه کرد واین فواصل را نیز در محاسبه کمترین فاصله دخیل کرد.

نکته: بدلیل اینکه پیاده سازی دایره ای با شعاع S مشکل است از یک مستطیل استفاده می کنیم.

نکته: فاصله دو نقطه (x_1,y_1) و (x_2,y_2) در فضای دو بعدی از رابطه $\sqrt{(x_1-x_2)^2+(y_1-y_2)^2}$ محاسبه میشود.



و لذا مرتبه زمانی الگوریتم بصورت زیر قابل محاسبه است:

$$T(n) = 2T(\frac{n}{2}) + 6\frac{n}{2} = \mathrm{O}(n.\log n)$$

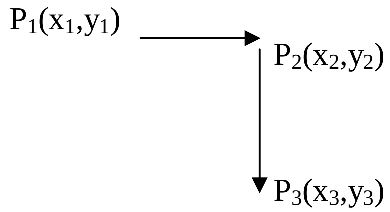
۴-۴- تعاریف و الگوریتمهای پایه در هندسه محاسباتی

چندضلعی بسته: دنباله‌ای از پاره‌خطها که انتهای هرکدام به ابتدای دیگری وصل باشد و ضمناً بسته شود یعنی انتهای آخری به ابتدای اولی متصل باشد.

چند ضلعی ساده: چند ضلعی بسته‌ای که در آن هیچ یالی، یال دیگری را قطع نکند.

چند ضلعی محدب: چندضلعی ساده‌ای که زاویه‌ای بیشتر از ۱۸۰ نداشته باشد (هردو نقطه داخل چندضلعی را به هم وصل کنیم پاره‌خط حاصل داخل چندضلعی باشد.)

جهت چرخش بین سه نقطه در فضای ۲ بعدی:



p_1, p_2, p_3 ← راست گرد

p_3, p_1, p_2 ← راست گرد

p_2, p_3, p_1 ← راست گرد

الگوریتم جهت تشخیص راست‌گرد یا چپ‌گرد بودن

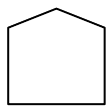
دترمینان ماتریس زیر محاسبه کرده، اگر بزرگتر از صفر بود چپ‌گرد است، اگر کوچکتر از صفر بود راست‌گرد است و اگر مساوی صفر بود ۳ نقطه هم خط هستند.

$$\begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix} > 0$$

تشخیص چپ‌گرد و راست‌گرد دارای مرتبه زمانی ← o (1)

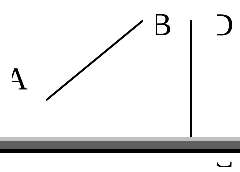
قضیه: نحوه تشخیص محدب بودن یک چندضلعی:

اگر تمامی رأس‌های متوالی نسبت به دو راس قبل از خود چپ‌گرد یا راست‌گرد باشد چندضلعی محدب است.o(n)



الگوریتم: چگونگی تشخیص متقاطع بودن دو پاره خط

دو پاره خط AB و CD به عنوان ورودی داده شده‌اند. اگر A نسبت به دو نقطه C و D راستگرد و B نسبت به دو نقطه C و D چپگرد باشد(و یا بالعکس) و همچنین C نسبت به دو نقطه A و B راستگرد و D نسبت به دو نقطه A و B چپگرد باشد(و یا بالعکس) آنگاه AB و CD متقاطع هستند و لذا باید در بدترین وضعیت ٤ بار الگوریتم تشخیص جهت (دترمینان بالا) محاسبه شود. از این رو الگوریتم دارای مرتبه زمانی O(1) میباشد.

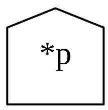


الگوریتم: نحوه پیداکردن یک نود داخل چند ضلعی محدب

سه رأس متوالی را به دلخواه انتخاب می‌کنیم میانگین x های آنها و میانگین y های آنها (x,y) نقطه‌ای داخل چند ضلعی محدب را تولید میکنند. لذا این روش دارای مرتبه o(1) میباشد.

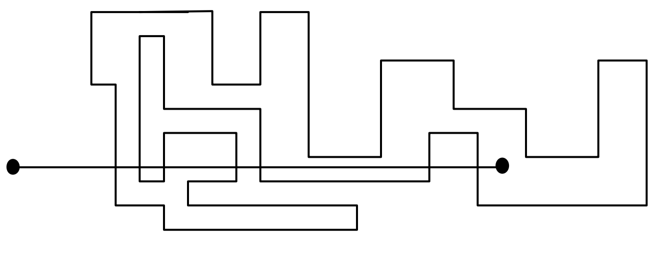
الگوریتم: تشخیص یک نقطه داخل چندضلعی محدب

اگر تمام زوج رئوس اضلاع متوالی (در جهت خلاف چرخش عقربه های ساعت) نسبت به نقطه جدید چپ‌گرد بودند نقطه داخل چندضلعی می‌باشد در غیر این‌صورت خارج آن است. و لذا الگوریتم دارای زمان o(n) میباشد.



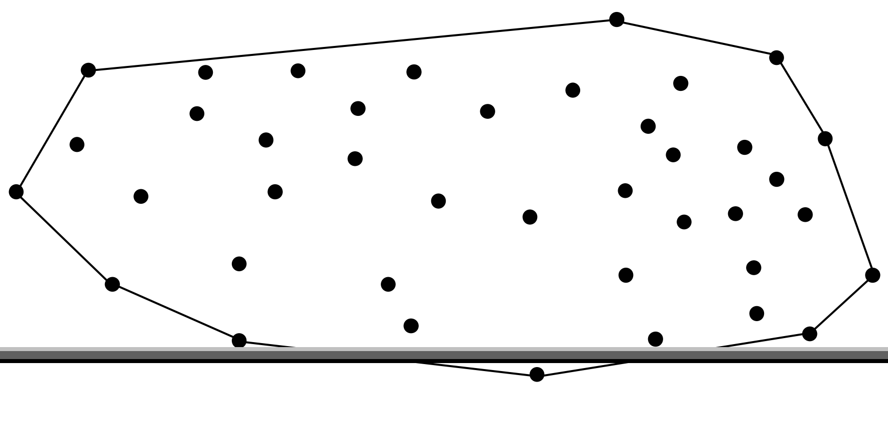
الگوریتم: تشخیص نقطه داخل چندضلعی مقعر

یک نقطه خارج چندضلعی مقعر انتخاب می‌کنیم و به نقطه موردنظر وصل می‌کنیم اگر تعداد برخوردها با یالهای چندضلعی فرد بود داخل چند ضلعی غیر محدب می‌باشد و اگر زوج بود خارج آن است. و لذا الگوریتم دارای زمان o(n) میباشد. در شکل زیر تعداد نقاط تقاطع برابر با ۷ میباشد و لذا نقطه مورد نظر در داخل شکل قرار دارد.



Convex hull: (پوسته یا پوش محدب). کوچکترین چند ضلعی محدب که شامل همه نقاط ورودی باشد.

نکته:برای تصور شکل پوسته محدب میتوان فرض کرد که تعدادی نقطه داریم و یک کش حلقوی را بازکرده دور نقاط قرار میدهیم. شکل ساخته شده توسط کش پوسته محدب را نشان میدهد.



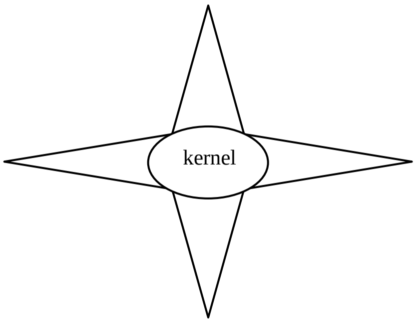
Extreme point: نقاطی را که روی convexhull هستند نقاط مرزی می‌گوییم.

نکته:

در بدترین وضعیت convex hull شامل تمام n نقطه میباشد.

در بهترین وضعیت convex hullفقط شامل ۳ نقطه میباشد.

مسئله موزه هنری: نحوه طراحی یک موزه به قسمی که به کمترین تعداد نگهبان جهت رویت احتیاج باشد.



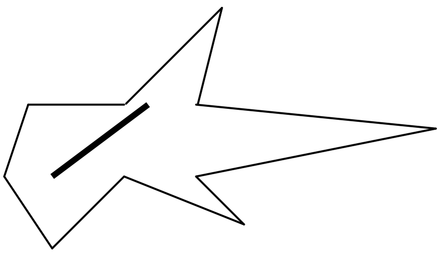
حل مسئله: در درجه اول بهتر است که convex hull باشد.

درصورتی‌که طراحی مقعر باشد باید به شکل Star shaped باشد تا یک نگهبان برای کل موزه کافی می‌باشد..

Kernel: فضایی در چند که از آن فضا کل چندضلعی دیده می‌شود.

نکته: در convex hull کل آن kernel است.

مسئله روشنایی: در convex hull یک لامپ برای روشنایی کافی است و موقعیت لامپ اهمیت ندارد در Star shaped یک لامپ برای روشنایی کافی است در صورتی که در داخل kernel قرار داشته باشد.



در شکل مقابل از یک لامپ که به شکل میله می‌باشد جهت روشن کردن کل چند ضلعی مقعراستفاده شده است.

الگوریتم محاسبه convexhull

دو روش مورد بحث قرار میگیرد:

۱- (Greedy) Graham

۲- (D & C) Shamos

٤-٥- تولید پوش محدب *(Convex Hull)*

٤-٥-١-الگوریتم Graham

١- در ابتدا نودی را که دارای کمترین y می‌باشد پیدا می‌کنیم و آن را p_1 مینامیم.

٢- مابقی نودها را نسبت به این نود براساس زاویه قطبی در جهت خلاف عقربه‌های ساعت مرتب می‌کنیم و آنها را p2 تا p_n نامگذاری میکنیم.

٣- P_1 و p_2 را داخل یک پشته قرار داده و سپس چپ‌گرد بودن بقیه نودها را با دو نقطه سر پشته بررسی می‌کنیم. اگر جهت چپ‌گرد بود نقطه مورد بررسی را به پشته اضافه میکنیم درغیراینصورت یک نقطه از سر پشته حذف شده و دوباره بررسی را انجام میدهیم.

```
Procedure Graham
   p1 ← find the point with minimum  y
   Sort the other points around p1 (CCW –Polar angle) and call them p2 ,…, pn
   Push (p1)
   Push (p2)
   For i←3  to n  do
      While Right (stack [top-1] , stack[top], Pi)  do
         Pop
      Repeat
      Push (pi)
   repeat
End.
```

سرشکن شدن هزینه در الگوریتم:

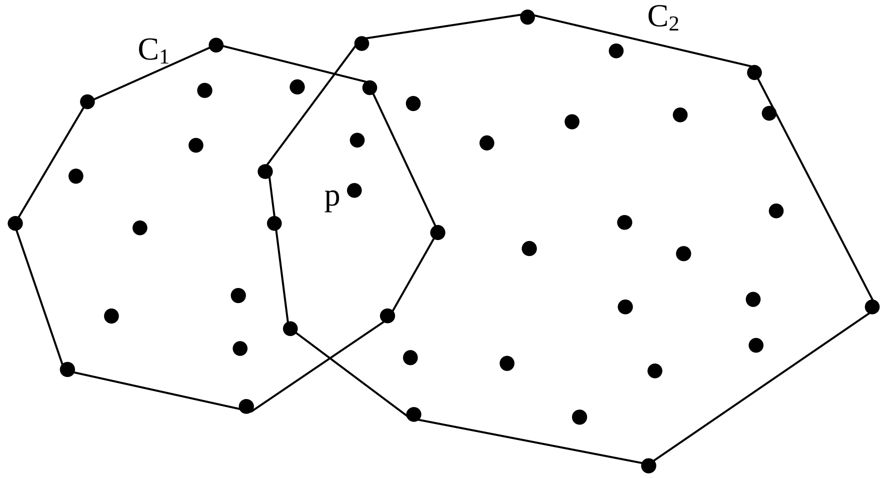
بیشترین تکرار در الگوریتم بالا مربوط به pop میباشد که برخلاف چیزی که در ظاهر نشان می‌دهد دارای مرتبه زمانی $o(n^2)$ نیست بلکه $o(n)$ است زیرا یک نقطه بیشتر از یک بار نمی‌تواند pop یا push شود و چون مرتبه زمانی مرتب سازی $O(n \log n)$ است. لذا مرتبه زمانی الگوریتم $O(n \log n)$ است.

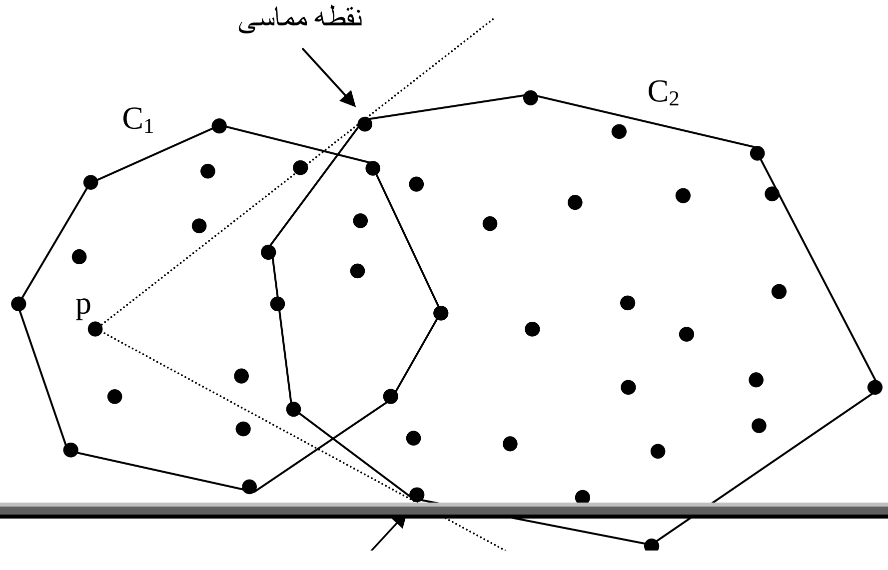
٤-۵-۲-الگوریتم Shamos

۱- نقاط را به دو دسته که هر دسته شامل n/2 نقطه است تقسیم میکنیم.
۲- هر زیر مسئله را بصورت بازگشتی حل میکنیم ولذا برای هر کدام یک پوسته محدب تولید میشود. این دو پسته را C_1 و C_2 مینامیم.
۳- یک نقطه داخل C_1 در نظر گرفته و آن را p مینامیم.
۴- اگر $p \in C_2$ آنگاه (نقطه p هم داخل C_1 و هم داخل C_2 است): دو مجموعه از نقاط C_1 و C_2 حول p که یک نقطه داخلی است مرتب هستند لذا میتوان آنها را ادغام کرده و سپس قسمت سوم الگوریتم گراهام را روی آنها اجرا کرد.
۵- اگر $p \notin C_2$ آنگاه دو نقطه مماسی از p نسبت به C_2 را محاسبه کرده لذا C_2 به دو قسمت پوسته داخلی زاویه مماسی و پوسته خارجی زلویه مماسی تقسیم میشود. نقاط قسمت داخلی زاویه مماسی از C_2 را حذف کرده و لذا نقاط قسمت خارجی زاویه مماسی از C_2 و کل نقاط C_1 حول p مرتب هستند با ادغام آنها و اجرای قسمت سوم الگوریتم گراهام میتوان پوسته محدب نهایی را تشکیل داد.

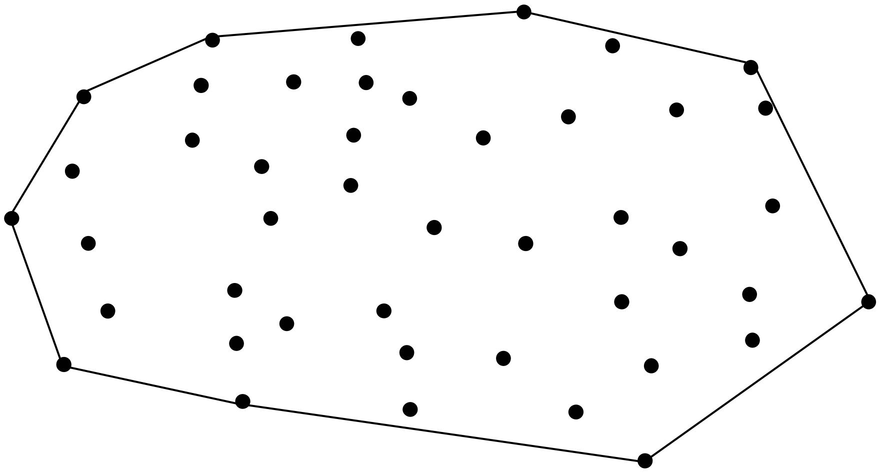
$$T(n) = 2T(\frac{n}{2}) + cn = O(n.\log n)$$

اشکال زیر دو حالت مختلف که برای نقطه p قابل تصور است را نشان میدهد.





پوسته محدب نهایی به شکل زیر خواهد بود.



۵-روش برنامه سازی پویا (Dynamic Programming)

در روش تقسیم و حل دیدیم که ابتدا از مسئله اصلی شروع کرده و آن را به زیر مسائل کوچکتر تقسیم می‌کنیم و کار را تا حد ممکن پیش می‌بریم و سپس از انتها به ابتدا مسائل را حل می‌کنیم. در واقع برای شکستن مسائل الگوی بالا به پایین (top down) و در ترکیب جوابها الگوی پایین به بالا رعایت می‌گردد که منطبق بر الگوریتم‌های بازگشتی است. اما اگر در روش تقسیم و حل لازم باشد زیر مسأله خاصی را چندین مرتبه حل کنیم این تکرار کارآیی الگوریتم را پایین می‌آورد. اگر چنین زیرمسأله‌ای را یک بار حل کرده و جواب آنها را نگهداری کنیم، می‌توانیم در مراحل بعدی از آن استفاده کنیم و این اساس کار روش حل مسائلی است که به روش برنامه‌سازی پویا حل می‌شوند.

در این روش از کوچکترین مسائل شروع و همه آنها را حل می‌کنیم و جواب آنها را نگهداری می‌کنیم. سپس به سطح بعدی می‌رویم و کلیه مسائل اندکی بزرگتر را حل می‌کنیم و سپس به حل مسائل سطح بعدی می‌پردازیم و کار را تا جایی ادامه می‌دهیم که مسأله اصلی حل شود. برای حل هریک از مسائل هر سطح می‌توانیم از حل کلیه سطوح پایین‌تر که لازم باشد استفاده کنیم. از روش برنامه سازی پویا زمانی میتوان استفاده کرد که اصل بهینگی برقرار باشد. این اصل بر این اساس است که برای حل مسئله بصورت بهینه از حل بهینه زیر مسائل آن میتوان استفاده کرد. در ادامه چند مسأله را که بدین روش حل می‌شوند بررسی می‌کنیم.

برنامه‌سازی پویا در مقایسه با روش تقسیم و حل: درخت حل مسئله را از پایین به بالا می‌سازیم و نتایج را در یک جدول نگهداری می‌کنیم تا در موقع لزوم بتوان از آنها استفاده کرد و دوباره آنها را حل نکرد.

نکته: نوعی روش برنامه سازی پویا وجود دارد که در آن زیر فضای حل مسئله از بالا به پایین است ولی زیر مسائل حل شده در جدولی نگهداری میشوند تا از حل زیر مسائل تکراری پرهیز شود که این روش به نام روش به خاطرسپاری (memoized) شناخته میشود.

اصل بهینگی: انتخاب بهینه نهایی به انتخابهای بهینه اولیه بستگی دارد.

برای ارائه الگوریتم به روش برنامه سازی پویا ۴ مرحله را باید طراحی کرد:

۱- تعریف تابعی که حل تابع منجر به حل مسئله شود.

۲- بیان شرایط مرزی

۳- بیان جواب مسئله بر حسب تابع

۴- تعریف بازگشتی تابع

۵-۱- مسئله کوله‌پشتی *0/1*

مسئله کوله پشتی صفر یا یک مدلی از مسئله کوله پشتی است که در آن اشیاء یا بطور کامل انتخاب میشوند و یا انتخاب نمیشوند و نمیتوان فقط کسری از اشیاء را انتخاب کرد.

ورودی:

n: تعداد کیسه ها

P_i: سود حاصل از انتخاب کل جسم i اُم

W_i: وزن کل جسم i اُم

M: گنجایش کوله پشتی

هدف: بیشینه کردن سود حاصل از انتخاب اجناس یعنی $Max\sum_{i=1}^{n} x_i . p_i$

شرایط مسئله:

وزن همه کیسه‌ها روی هم از وزن کوله‌پشتی بیشتر است زیرا اگر کمتر باشد یعنی می‌توانیم همه را برداریم و انتخابی وجود ندارد همچنین مجموع وزن اجسامی که انتخاب کردیم از وزن کل کوله‌پشتی نباید بیشتر شود.

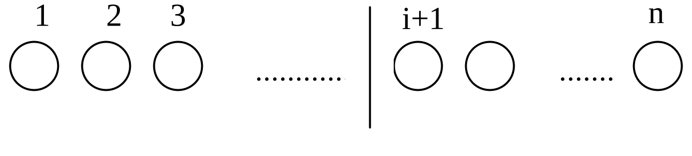
۱- $$\sum_{i=1}^{n} w_i > M$$

۲- $$\sum_{i=1}^{n} x_i w_i \le M$$

خروجی: کسری از کیسه i اُم که انتخاب می‌شود داخل عنصر i ام آرایه قرار می‌گیرد.

X_i = کسری از کیسه i ام که انتخاب میشود. (X_i =0 or 1) (i=1,…,n)

راه حل: توسط مثالهای نقضی میتوان نشان داد که انتخاب کیسه ها بر اساس معیار مطرح شده در مسئله کوله پشتی کسری ممکن است به جواب بهینه منجر نشود. توسط مثال نقضی میتوان این موضوع را نشان داد.



الگوریتم:

۱- $g_i(y)$: بیشترین سود حاصل از حل مسأله برای جسم i+1اُم تا nاُم به شرطی که گنجایش کوله y باشد.

۲- $g_0(M)$ بیشترین سود حاصل از حل مسئله برای کل اجسام به شرطی که گنجایش کوله M باشد.

۳- شرایط مرزی:

$g_n(y)=0$ ، (از جسمn+1 به بعد جسمی وجود ندارد که سودی داشته باشد)

y<0، گنجایش کوله منفی می‌باشد یعنی بیشتر از حد کوله جسم داریم در این حالت $g_i(y)$ حالت ضرر است.

$(g_i(y))=-\infty$

۴- $g_i(y)=Max\{g_{i+1}(y), p_{i+1}+g_{i+1}(y-w_{i+1})\}$

مثال: مسئله زیر را به به کمک الگوریتم بالا حل کنید؟

P	۲۰	۱۰	۳۰
w	۱۲	۱۰	۱۳

M=30

$g_0(30)=Max\{g_1(30),20+g_1(18)\}=50$

$g_1(30)=Max\{g_2(30),10+g_2(20)\}=40$

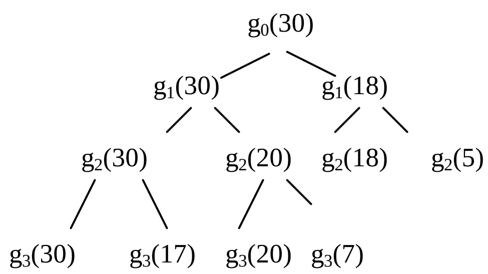
$g_1(18)=Max\{g_2(18),10+g_2(8)\}=30$

$g_2(30)=Max\{g_3(30),30+g_3(17)\}=30$

$g_2(20)=Max\{g_3(20),30+g_3(7)\}=30$

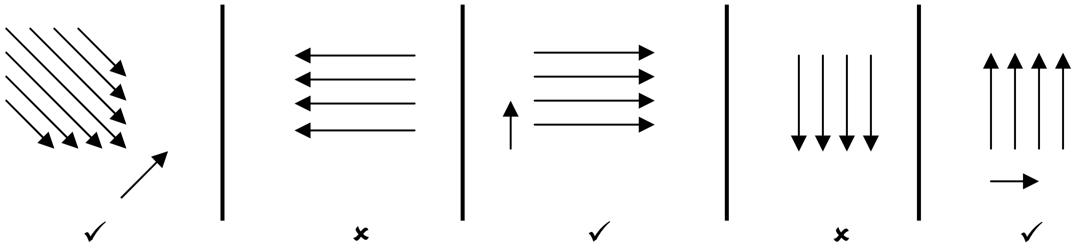
$g_2(18)=Max\{g_3(18),30+g_3(5)\}=30$

$g_2(8)=Max\{g_3(8),30+g_3(-5)\}=0$



g	-	0	1	2		Y		M-1	M
0	-∞								جواب
1	-∞								
	-∞								
i	-∞					$g_i(y)$			
	-∞								
n-1	-∞								
n	-∞	0	0	0	0	0	0	0	0

نکته اساسی در روش محاسبه ماتریس جواب در این است که به کدام یک از روشهای زیر میتوان ماتریس را پر کرد. با کمی دقت در رابطه بازگشتی میتوان به جواب رسید.



```
function DP_knapsack(W,P,n,M)
   for i←0 to M do g[n,i] ←0 repeat
   for i←0 to n do g[-,i] ←-∞ repeat
   for   i=n-1    to   0
      for   j=0     to    m
         g[i,j] =max{g[i+1,y],p_{i+1}+g[i+1,y-W_{i+1}] }
      repeat
   repeat
   return g[0,M]
end.
```

مرتبه زمانی این الگوریتم o(M.n) می‌باشد.

نکته: با اینکه مرتبه زمانی الگوریتم بالا در ظاهر چندجمله‌ای است ولی چون فقط بـه تعـداد اقـلام ورودی وابسـته نیست بلکه به حجم کوله پشتی هم وابسته است لذا به آن شبه چند جمله‌ای (pseudo polynomial) گویند.

۵-۲- مسئله همه کوتاهترین مسیرها(*APSP*)

مسئله دیگری که از دسته مسائل کوتاهترین مسیرها بر روی گراف اسـت، مسـئله همـه کوتـاهترین مسـیرها (All Pairs Shortest Paths) میباشد که هدف آن پیدا کردن طول کوتـاهترین مسـیر بـین همـه زوج رئـوس گـراف میباشد.

دراین روش ما تمام مسیرهای ممکن از هر رأسی را به هر رأس دیگر که از رأس k بگذرد یـا نگـذرد را محاسـبه می‌کنیم تا کوتاهترین مسیر بدست آید و خودk از یک تاn تغییر می‌کند. و وقتی k=n شد ما تمام حـالات ممکـن برای کوتاهترین مسیر را محاسبه کردیم. (جواب= A(i,j)) و وقتی k=0 است یعنی از هیچ رأس عبـور نمـی‌کنـیم مستقیماً از رأس i به j می‌رویم که همان cost(i,j) می‌باشد.

نکته: در صورتی‌که مسیر از i به j وجود نداشته باشد. ارزش یالهای آن برای ∞ می‌شود.

الگوریتم :

$A^k(i,j)=$

طول کوتاهترین مسیر از راس *i* به راس *j* به قسمی که از رئوس با شماره بزرگتر از *k* حق گذر نداشته باشیم

$A^n(i,j)=$

جواب

$A^0(i,j)=cost(i,j)$

عبارت زیر بیان بازگشتی توضیحات بالا می‌باشد.

$$A^k(i,j)=\min\{A^{k-1}(i,j),A^{k-1}(i,k)+A^{k-1}(k,j)\}$$

نکته: برای محاسبه $A^k(i,j)$ دو حالت پیش می‌آید. یا مسیر بهینه از راس k میگـذرد کـه در نتیجـه مسـیر بـه دو قسمت i→k و k→j قابل تقسیم است(که هیچ کدام از این دو مسیر راس با شماره بزرگتر از k-1 ندارند) ویـا از راس k نمیگذریم که در نتیجه کلیه رئوس نمیتوانند از k-1 بزرگتر باشند.

حال اگر ما بتوانیم $A^n(i,j)$ را محاسبه کنیم کوتاهترین،
مسیر از $i \rightarrow j$ را محاسبه نموده‌ایم.
الگوریتم مورد بحث به نام الگوریتم floyd معروف است که شبه کد آن را در زیر میبینیم:

```
procedure APSP_Floyd(cost,n,A)
   for i←1 to n do
      for j←1 to n do
         A[i,j] ←cost[i,j]
         p[i,j] ←0
      repeat
   repeat
   for k←1 to n do
      for i←1 to n do
         for j←1 to n do
            A[i,j] ←min{A[i,j],A[i,k]+A[k,j]}        (*)
         repeat
      repeat
   repeat
end.
```

تولید مسیر: اگر در الگوریتم بالا جمله (×) را با جمله زیر جایگزین کنیم آنگاه عنصر P[i,j] همیشه راس میانی بین دو راس i و j را دربردارد و لذا میتوان به کمک ماتریس p کوتاهترین مسیرها را تولید کرد:

```
if A[i,k]+A[k,j]<A[i,j] then
   A[i,j] ←A[i,k]+A[k,j]
   p[i,j] ←k
endif
```

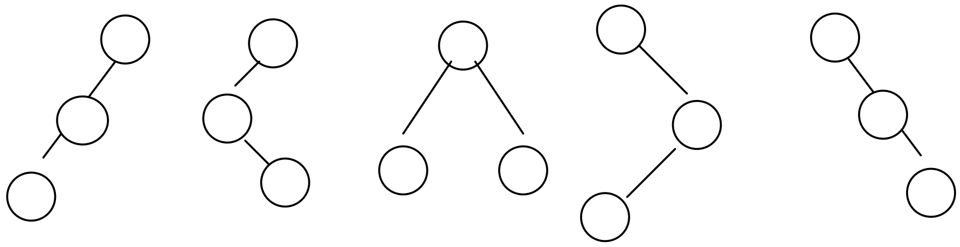
واضح است که مرتبه زمانی الگوریتم بالا $O(n^3)$ میباشد.

۵-۳- عدد کاتالان (*Catalan Number*) و مسائل وابسته

حال به بررسی چند مسئله که همگی آنها دارای پاسخ‌های یکسان می‌باشند، میپردازیم.

۱- با n گره چند درخت دودویی می‌توان ساخت؟

مثال: با سه گره چند درخت دودوئی می‌توان ساخت؟



(۱) (۲) (۳) (۴) (۵)

۲- به چند طریق می‌توان n+1 یک ماتریس را در هم ضرب کرد؟

مثال: به چندطریق می‌توان ٤ ماتریس را درهم ضرب کرد؟

$1-\left(\left(\left(m_1\times m_2\right)\times m_3\right)m_4\right)$

$2-\left(\left(m_1\times\left(m_2\times m_3\right)\right)\times m_4\right)$

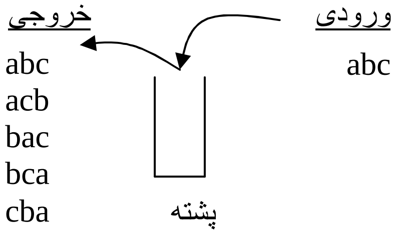
$3-\left(\left(m_1\times m_2\right)\times\left(m_3\times m_4\right)\right)$

$4-\left(m_1\times\left(\left(m_2\times m_3\right)\times m_4\right)\right)$

$5-\left(m_1\times\left(m_2\times\left(m_3\times m_4\right)\right)\right)$

۳- به چند طریق می‌توان n ورودی را به کمک یک پشته در خروجی چاپ کرد به قسمی که فقط عملیات read، push،write و pop قابل انجام است؟

مثال: ۳ ورودی a,b,c را به چند طریق می‌توان در خروجی چاپ کرد؟



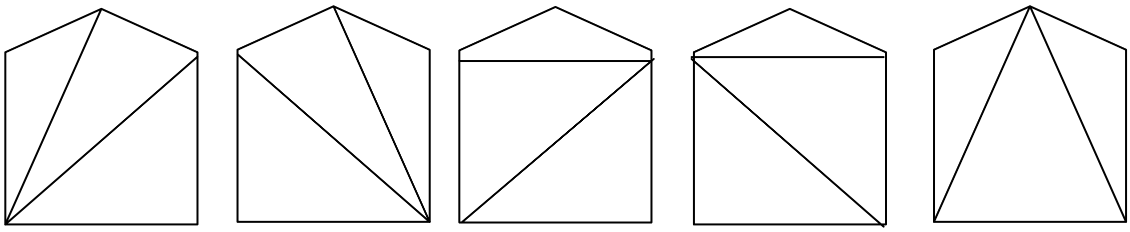
ملاحظه میشود که ۳ ورودی را به ۵ طریق میتوان در خروجی قرار داد.

٤- به چند طریق می‌توان یک n+2 ضلعی محدب را مثلث‌بندی کرد؟

ابتدا تعریفی از مثلث بندی ارائه میدهیم. اگر P یک چندضلعی محدب باشد، مثلث بندی P ، اضافه کردن قطرهایی غیر متقاطع در داخل P است به گونه‌ای که P به تعدادی مثلث افراز گردد.

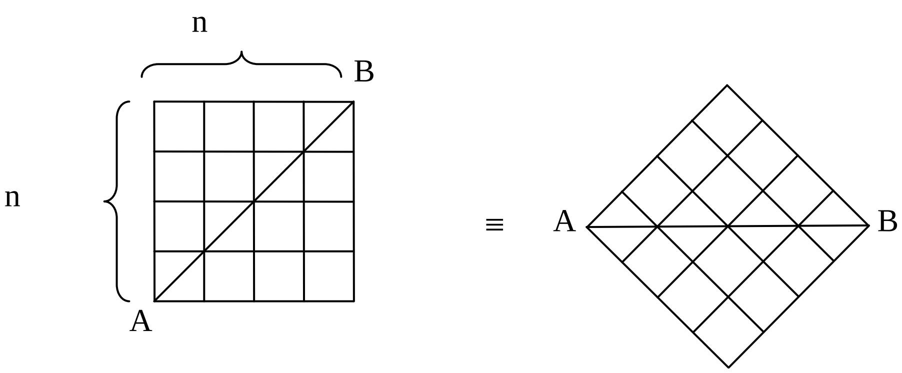
نکته: مثلث بندی P منجر به رسم بیشترین تعداد اقطار داخل P میشود به قسمی که اقطار یکدیگر را قطع نکنند.

مثال: یک ۵ ضلعی محدب را به چند طریق می‌توان مثلث‌بندی کرد؟

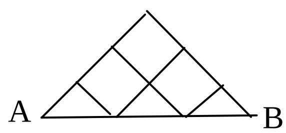


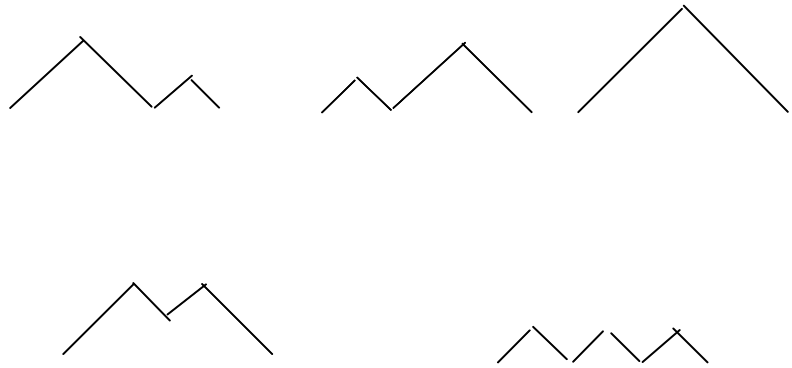
ملاحظه میشود که یک ۵ ضلعی محدب را به ۵ طریق میتوان مثلث بندی کرد.

۵- به چند طریق می‌توان در یک گرید که دارای n سطر و n ستون است، از پوشه پایین سمت چپ به گوشه بالا سمت راست رسید. به قسمتی که هیچگاه زیر قطر قرار نگیریم و فقط 90+ درجه یا 0 درجه حرکت کنیم(شکل سمت چپ) یا هیچگاه زیر پاره خط واصل بین A و B قرار نگیریم و فقط 45+ یا 45- درجه حرکت کنیم (شکل سمت راست)؟



مثال: در شکل زیر به چند طریق می‌توان از رأس A به B رسید؟
(به قسمتی که حرکت فقط 45+ و یا 45- درجه باشد.)

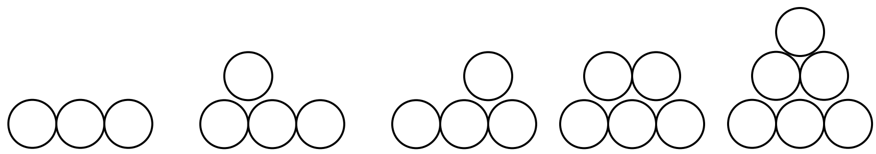




ملاحظه میشود که باز برای n=3 به ۵ طریق امکان پذیر است.

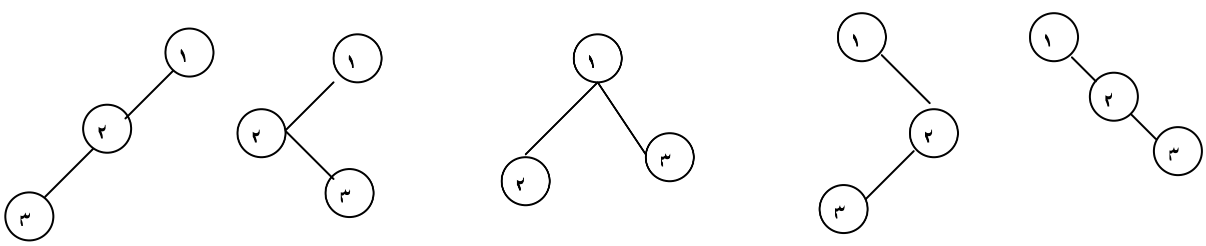
٦- به چند طریق می‌توانn سکه را در قاعده قرار داد و روی آن تعداد دلخواه سکه چید؟

مثال: به چند طریق می‌توان ۳ سکه را در قاعده قرار داد و روی آن سکه چید؟



ملاحظه میشود که باز برای n=3 به ۵ طریق امکان پذیر است.

۷- اگر پیمایش preorder درخت دودویی برابر با 1,2,3,...,n باشد چند inorder برای آن متصور است؟



فقط درختهای بالا است که پیمایش preorder آنها 1,2,3 است و لذا ۵ طریق inorder زیر امکان پذیر است:

1,2,3
1,3,2
2,1,3
2,3,1

3,2,1

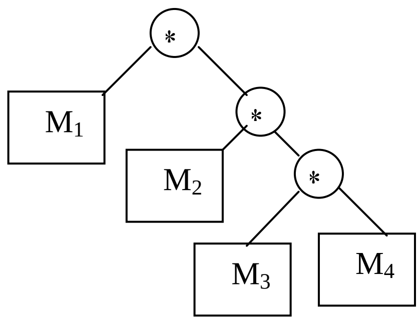
قابل اثبات است که جواب همگی این مسائل همگی برابر با عدد معروفی به نام عدد کاتالان میباشد که دارای مقدار زیر است:

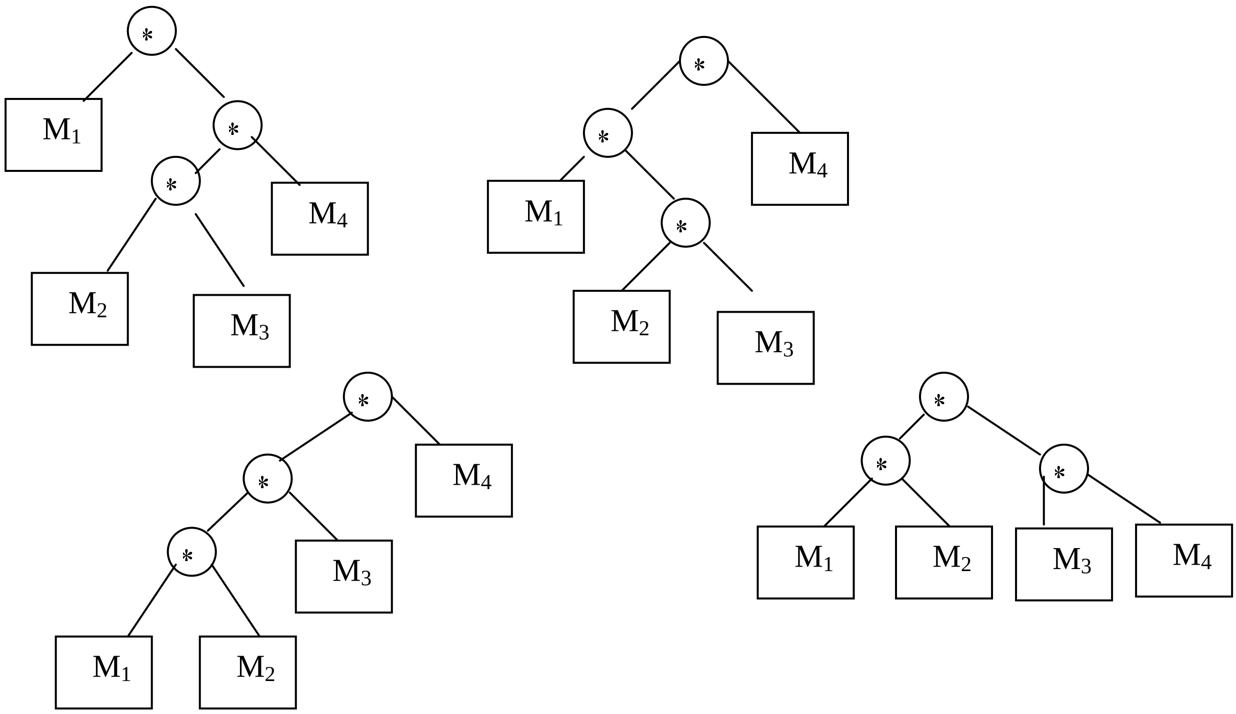
$$C_n = \frac{\binom{2n}{n}}{n+1}$$

حال نکته جالب در این است که باید بتوان ثابت کرد که این مسائل همگی ذاتاً یک مسئله هستند و در نتیجه باید بتوان یک نگاشت یک به یک بین جوابهای آنها ایجاد کرد.

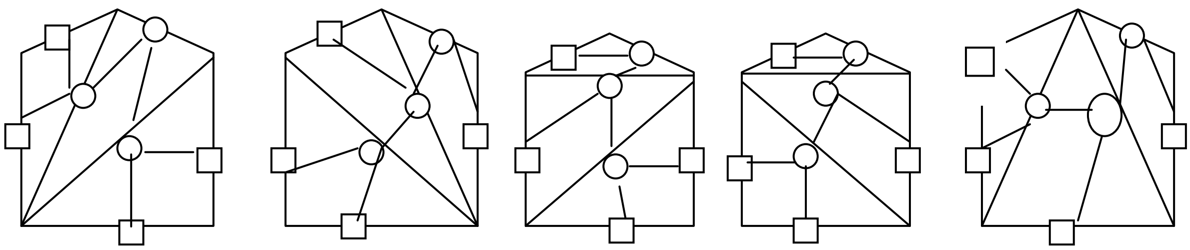
مثال: به بررسی رابطه‌های بین این مسئله‌ها می‌پردازیم(سعی کنید این روابط را از روی شکل بیابید).

رابطه‌ای بین مسئله ۱ و ۲

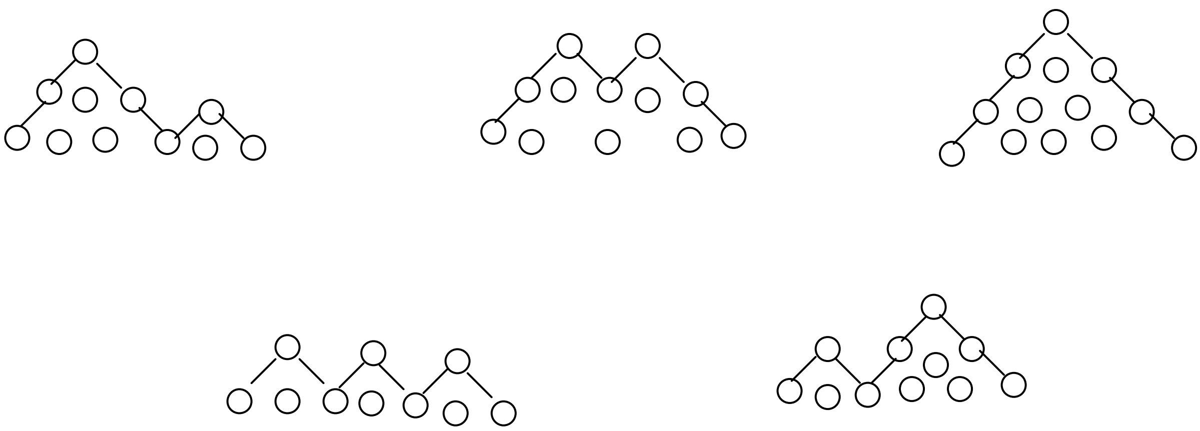




رابطه بین مسئله ٤ و ٢



رابطه بین مسئله ٥و٦:



سعی کنید رابطه بین بقیه مسائل را خودتان بدست آورید.

۵-۴- ضرب زنجیره‌ای و بهینه ماتریس‌ها

میدانیم که در ضرب دو ماتریس که اولی دارای m سطر و n ستون و دومی دارای n سطر و p ستون است باید به تعداد m*n*p ضرب انجام داد ولذا در این قسمت هزینه ضرب این دو ماتریس را برابر با m*n*p در نظر میگیریم. با این فرض به شرح صورت مسئله میپردازیم. در این مسئله فرض بر این است که n ماتریس باید در هم ضرب شوند و هدف تعیین طریقی از ضرب است که در آن هزینه کمینه گردد(میدانیم که تعداد طرق ضرب n ماتریس برابر با C_{n-1} است).

ورودی: اعداد صحیح $p_0, p_1, \ldots, p_n$

که در آن ماتریس p_{i-1} تعداد سطرها و p_i تعداد ستونها ماتریس i ام (M_i) است.

خروجی: کمترین هزینه ضرب ماتریس‌های M_1 تا M_n

$$m_1 * m_2 * m_3 * \ldots\ldots * m_n$$

روش حل:

$$\cos t\left(m_{pi\times pj} * m_{pj\times pk}\right) = p_i * p_j * p_k$$

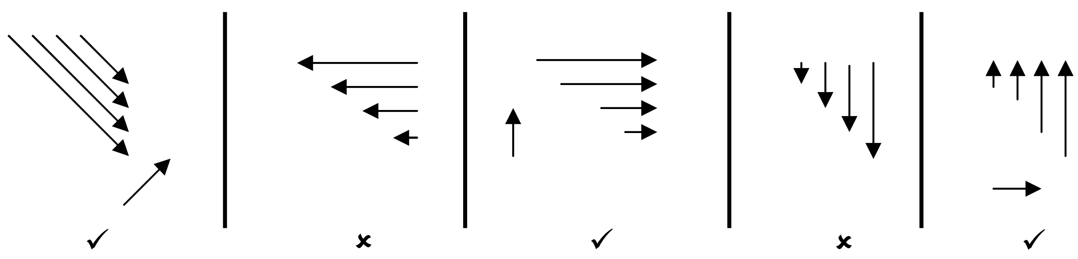
مراحل برنامه‌سازی پویا:

۱- کمترین هزینه ضرب ماتریس‌های $M_i * \ldots * M_j$ = $m[i, j]$
۲- $m[i,i]=0$
۳- جواب $= m[1,n]$
۴- $m[i, j] = \min_{i\le k<j}\{m[i,k] + m[k+1, j] + p_{i-1} p_k p_j\}$

توضیح:

با کمی دقت در رابطه بازگشتی میتوان به نحوه محاسبه عناصر ماتریس m دست یافت. همچنین چون همیشه در m[i,j] فرض بر این است که $i \le j$ لذا فقط عناصر بالای قطر اصلی در ماتریس m باید محاسبه شوند.

m	1	2		j		n
1	0					جواب
2	×	0				
	×	×	0			
i	×	×	×	0		
	×	×	×	×	0	
n	×	×	×	×	×	0



واضح است که برای محاسبه ماتریس m باید عناصر بالای قطر اصلی محاسبه شوند و برای هر درایه نیز در بدترین وضعیت نیاز به مینیمم گیری روی n-1 عضو وجود دارد(بر اساس رابطه بازگشتی در الگوریتم) و از این رو این الگوریتم دارای مرتبه زمانی $O(n^3)$ میباشد.

مثال:

محاسبه کمترین هزینه ضرب٤ ماتریس : (به ٥ طریق می‌توان ٤ ماتریس را ضرب کرد.)

ورودی:

$M_{1(2*3)}*M_{2(3*5)}*M_{3(5*4)}*M_{4(4*1)}$

$p_0=2$, $p_1=3$, $p_2=5$, $p_3=4$, $p_4=1$

$m[1,2]=min\{m[1,1]+m[2,2]+p_0p_1p_2\}=2*3*5=30$
$m[2,3]=min\{m[2,2]+m[3,3]+p_1p_2p_3\}=3*5*4=60$
$m[1,3]=min\{m[1,1]+m[2,3]+p_0p_1p_3 , m[1,2]+m[3,3]+p_0p_2p_3\}$
$=min\{60+2*3*4,30+2*5*4\}=70$
$m[3,4]=min\{m[3,3]+m[4,4]+p_2p_3p_4 \}=20$
$m[2,4]=min\{m[2,2]+m[3,4]+p_1p_2p_4 , m[2,3]+m[4,4]+p_1p_3p_4\}$
$=min\{20+3*5*1,60+3*4*1\}=35$
$m[1,4]=min\{m[1,1]+m[2,4]+p_0p_1p_4 , m[1,2]+m[3,4]+p_0p_2p_4, m[1,3]+ m[4,4] + p_0p_3p_4 \}$
$=min\{35+2*3*1, 30+20+2*5*1,70+2*4*1\}=41$

نکته: برای بدست آوردن طریق بهینه ضرب باید دقت کرد که جواب بهینه بر اساس کدام انتخاب بدست آمده است. برای مثال جواب ۴۱ در مثال بالا به خاطر کمینه بودن 35+2*3*1 بدست آمده است که این مقدار نیز همان $m[1,1]+m[2,4]+p_0p_1p_4$ است و لذا باید ابتدا M_1 را جدا کرد و جواب بهینه $M_2*...*M_4$ را بدست آورد و این دو ماتریس را در هم ضرب کرد. $m[2,4]$ نیز خود از روی $m[2,2]+m[3,4]+p_1p_2p_4$ محاسبه شده است و لذا باید ابتدا جواب بهینه M_3*M_4 را بدست آورده و در M_2 ضرب کرد. پس طریقه ضرب بهینه بصورت زیر است:

$(M_1) * ((M_2) * ((M_3) * (M_4)))$

٥-٥- مثلث بندی بهینه چند ضلعی محدب

یکی دیگر از مسئله های جالبی که توسط روش برنامه سازی پویا میتوان برای آن الگوریتم کارایی تولید کرد مسئله مثلث بندی (triangulation) بهینه یک چند ضلعی محدب نام دارد. اکنون به بیان این مسئله و روش حل آن میپردازیم.

ورودی: مختصات n+1 راس از یک n+1 ضلعی محدب که رئوس آن بصورت $V_0, V_1, ..., V_n$ میباشد.

هدف : مثلث بندی چند ضلعی به گونه ای که مجموع محیط مثلثها کمینه گردد.

قبل از ارائه الگوریتم ابتدا W(i,j,k) را بصورت هزینه مثلث V_i,V_j,V_k یا همان محیط مثلث V_i,V_j,V_k تعریف میکنیم.

الگوریتم:

شبیه دیگر مسائل باید به ارائه ۴ مرحله برنامه سازی پویا بپردازیم:

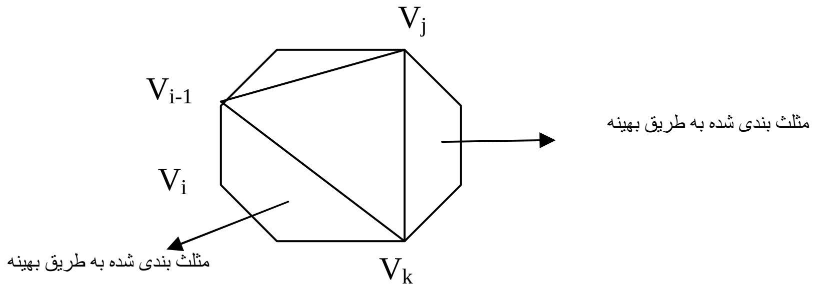
۱- کمترین هزینه مثلث بندی روی رئوس $V_{i-1}, V_i, ..., V_j$ = t[i,j]

۲- t[i,i+1]=0 (بدیهی)

۳- جواب = t[1,n]

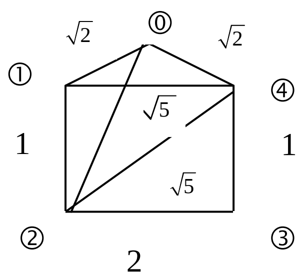
۴- رابطه بازگشتی:

$$t[i,j] = \min\{t[i,k] + t[k+1,j] + w(v_{i-1}, v_k, v_j)\}$$
$$i \le k < j$$



رابطه بازگشتی بدین صورت نوشته شده است که در طریقه مثلث بندی بهینه رئوس V_{i-1} تا V_j راسی مثل V_k وجود دارد که هم از V_{i-1} و هم از V_j به آن قطری متصل گردیده است و همچنین V_{i-1} تا V_k و V_k تا V_j بصورت بهینه مثلث بندی شده اند. بنابراین برای پیدا کردن چنین k ای باید تمام رئوس بین V_{i-1} تا V_j را بررسی کرد.

مثال: چند ضلعی زیر را به روش بالا با کمترین هزینه مثلث‌بندی کنید؟



$$t[1,2] = \min\{t(1,1) + t(2,2) + v_0v_1v_2\} = 1 + \sqrt{2} + \sqrt{5}$$

$$t[1,3] = \min\{t(1,2) + t(2,3) + v_0v_2v_3 = 3 + \sqrt{2} + 3\sqrt{5}$$
$$t(1,2) + t(3,3) + v_0v_2v_3 = 2 + \sqrt{2} + 3\sqrt{5}\}$$
$$= 2 + \sqrt{2} + 3\sqrt{5}$$

$$t[2,3] = \min\{t(2,2) + t(3,3) + v_1v_2v_3\} = 3 + \sqrt{5}$$

$$t[3,4] = \min\{t(3,3) + t(4,4) + v_2v_3v_4\} = 3\sqrt{5}$$

$$t[2,4] = \min\{t(2,2) + t(3,4) + v_1v_2v_3 = 6 + 2\sqrt{5}$$
$$t(2,3) + t(4,4) + v_1v_3v_4 = 6 + 2\sqrt{5}\}$$

$$t[1,4]=\min\{t(1,1)+t(2,4)+v_0v_1v_4=8+2\sqrt{2}+2\sqrt{5}$$ و
$$t(1,2)+t(3,4)+v_0v_2v_4=4+4\sqrt{5}+2\sqrt{2}$$ و
$$t(1,3)+t(4,4)+v_0v_3v_4\ \}=4+2\sqrt{2}+4\sqrt{5}$$

٥-٦- طولانی‌ترین زیر دنباله مشترک (LCS)

در این قسمت به مسئله طولانی ترین زیـر دنبالـه مشـترک (Longest Common Subsequence) میپـردازیم.

ابتدا تعریفهای لازم را ارائه میدهیم.

رشته: دنباله ای از کاراکترهای پشت‌سرهم با ترتیبی خاص. برای مثال در زیر رشـته X کـه دنبالـه ای از m حـرف است دیده میشود.

پیشوند i ام X: i حرف پشت سر هم از ابتدای رشته X را پیشوند i ام X گویند و با X_i نشان میدهند.

حرف i ام: h امین حرف رشته X را با x_i نشان میدهند.

$$X=<x_1,\cdots,x_m>$$
$$X_i=<x_1,\cdots,x_i>$$
$$Z=<z_1,\cdots,z_k>$$

زیر دنباله: گوییم z زیردنباله‌ای از X است اگر:

$$\exists i_1,i_2,\cdots,i_k\ |i_1<i_2<\cdots<i_k:\forall j:1\le j\le k\ \ z_j=x_{i_j}$$

مثال:

$X=<A,B,B,A,D,A,B,F>$

$Z=\ <B,A,A,F>$

$$\begin{cases} i_1=2\Rightarrow z_1=x_2\\ i_2=4\Rightarrow z_2=x_4\\ i_3=6\Rightarrow z_3=x_6\\ i_4=8\Rightarrow z_4=x_8\end{cases}$$

نکته: هر رشته m حرفی دارای 2^m زیردنباله است.

دراین مسئله در ورودی دو رشته به طول‌هـای n,m وجـود دارد. کـه هـدف مسـئله بدسـت آوردن طـولانی‌تـرین زیردنباله مشترک می‌باشد.

ورودی:

دو دنباله X و Y که اولی به طول m و دومی به طول n میباشد.

خروجی: طول طولانی ترین زیر دنباله مشترک X و Y

یک راه‌حل ساده جهت حل این مسئله استفاده از تابع زیر و یک آرایه دوبعدی $m \times n$ می‌باشد.

شبیه دیگر مسائل باید به ارائه ۴ مرحله برنامه سازی پویا بپردازیم:

۱- طول طولانی ترین زیر دنباله مشترک X_i و Y_j = c[i,j]

۲- c[i,0]=c[0,j]=0 ; ∀i,j (بدیهی)(وقتی یکی از رشته‌ها تهی باشد هیچ اشتراکی وجود نخواهد داشت)

۳- جواب = c[m,n]

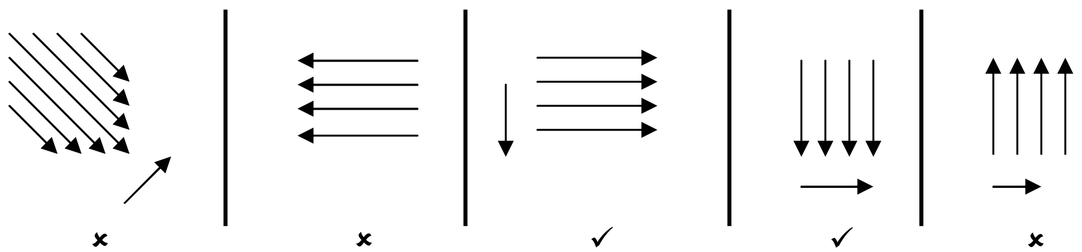
۴- رابطه بازگشتی:

$$c[i,j]=\begin{cases} c[i-1,j-1]+1 & ;x_i=y_j \\ \max\{c[i-1,j],c[i,j-1]\} & ;else \end{cases}$$

به راحتی میتوان با در نظر گرفتن یک ماتریس (m+1)*(n+1) به جای c الگوریتم بالا را پیاده سازی نمود.

c	0	1		j		n
0	0	0	0	0	0	0
1	0					
	0		$c_{i-1,j-1}$	$c_{i-1,j}$		
i	0		$c_{i,j-1}$	$c_{i,j}$		
	0					
m	0					جواب

روش محاسبه: چون $c_{i,j}$ برای محاسبه نیاز به $c_{i-1,j-1}$ و $c_{i,j-1}$ و $c_{i-1,j}$ دارد لذا روش محاسبه بصورت زیر قابل اجرا میباشد:



در زیر شبه کد الگوریتم دیده میشود:

```
Procedure LCS (X,Y,m,n)
   For  i←0   to   m    do     c[i,0] ←0   repeat
   For  j←0   to   n     do   c[0,j]  ←0  repeat
   For  i←1    to    m    do
      For  j←1    to      n     do
         If   xi=yj     then     c[i,j]  ← 1+c[i-1,j-1] , b[i,j]  ← '↘'
         elsif   c[i-1,j] > c[i,j-1]   then  c[i,j] ← c[i-1,j] , b[i,j]  ←  '↓'
         else    c[i,j]  ← c[i,j-1] , b[i,j]    ← '→'
         endif
       repeat
    repeat
end.
```

واضح است که مرتبه زمانی الگوریتم O(m.n) میباشد.

حال رویه ای بازگشتی را میبینیم که توسط آن میتوان طولانی ترین زیر رشته مشترک را چاپ کرد:

```
procedure print_LCS(X,b,i,j)
   if (i=0)  or  (j=0)   then  return endif
   if  b[i,j] = '↘'  then
      print_LCS(X,b, i-1, j-1)
      write (xi)
   elsif  b[i,j] = '→'   then
      print_LCS(X,b,i,j-1)
   else
      print_LCS(X,b, i-1,j)
   endif
end.
```

نکته: فراخوانی اولیه رویه print_LCS باید بصورت print_LCS(X,b,m,n) باشد.

نکته: کمترین تعداد فراخوانی رویه print_LCS برابر با min (m,n) و بیشترین آنها برابر با m+n میباشد.

نکته: کمترین تعداد write در رویه print_LCS برابر با صفر و بیشترین تعداد آن برابر با min(m,n) میباشد.

مثال:

ورودی: دو دنباله X به طول m و Y به طول n

خروجی: طول طولانی‌ترین زیر دنباله مشترک X و Y

X= <A,B,B,A,D,A,B,F>

Y= <B,F,A,F,D,H,B>

با اجرای الگوریتم به ماتریس زیر خواهیم رسید:

جهت وضوح بیشتر هر دو ماتریس c و b در یک ماتریس نشان داده شده است.

نکته: وقتی هر دو درایه c[i-1,j] و c[i,j-1] با هم برابر باشند هر کدام از آنها به دلخواه میتوانند به عنوان max{c[i-1,j],c[i,j-1]} انتخاب شوند الگوریتم نوشته شده در بالا بصورتی است که c[i,j-1] را به عنوان بیشینه انتخاب میکند.

c,b	-	B	F	A	F	D	H	B
-	0	0	0	0	0	0	0	0
A	0	0→	0→	1↘	1→	1→	1→	1→
B	0	1↘	1→	1→	1→	1→	1→	2↘
B	0	1↘	1→	1→	1→	1→	1→	2↘
A	0	1↓	1→	2↘	2→	2→	2→	2→
D	0	1↓	1→	2↓	2→	3↘	3→	3→
A	0	1↓	1→	2↘	2→	3↓	3→	3→
B	0	1↘	1→	2↓	2→	3↓	3→	4↘
F	0	1↓	2↘	2→	3↘	3→	3→	4↓

الگوریتم print_LCS مسیر مشخص شده به رنگ خاکستری را از آخر به اول طی کرده و دوباره بازمیگردد(به علت بازگشتی بودن رویه) و در حین بازگشت خانه هایی که در آن فلش مورب میبیند را چاپ میکند. از این رو طولانی ترین زیر دنباله مشترک X و Y بصورت زیر چاپ خواهد شد:

Z= <B,A,D,B>

نکته: بدون در نظر گرفتن ماتریس b هم الگوریتم print_LCS میتواند طولانی ترین زیر دنباله مشترک را چاپ کند!.

٥-٧- فروشنده دوره گرد

تعریف: دور هامیلتونی در یک گراف دوری است که از همه رئوس دقیقاً یکبار بگذرد.

در مسئله فروشنده دوره گرد (Traveling Salesman Problem) یا TSP ، هدف پیدا کردن دور همیلتونی با هزینه مینیمم در یک گراف وزن دار ورودی میباشد.

فرض کنید گراف ورودی بصورت G(V,E) میباشد که در آن V={1,2,...,n} است و $c_{i,j}$ نشاندهنده هزینه یال (i,j) باشد. همچنین بدون کم شدن از کلیت مسئله فرض کنید راس آغازین راس شماره ۱ باشد.

شبیه بقیه الگوریتمهایی که به روش برنامه سازی پویا ارائه شد باید ۴ مرحله را طراحی کرد:

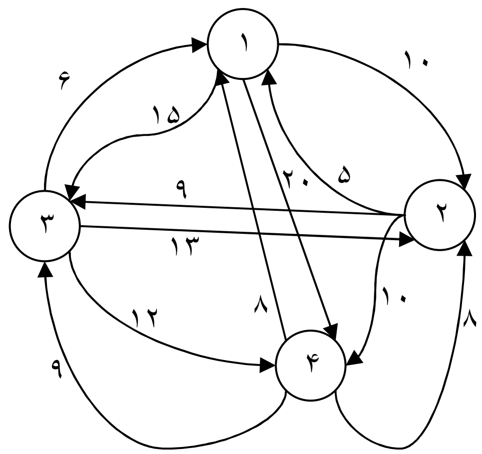
۱- طول کوتاهترین مسیری که از راس i شروع شده و از کلیه رئوس مجموعه S دقیقاً یکبار بگذرد و به راس ۱ ختم شود = $g(i,S)$ $(i \notin S\ ,\ 1 \notin S)$

۲- $g(i, \varnothing)=c_{i,1}$ (بدیهی)

۳- جواب = $g(1, V-\{1\})$

۴- $g(i,S)= \min\{c_{i,j} + g(j, S - \{j\})\ \}$ $\quad (j \in S, 1 \notin S, i \notin S)$

مثال: در گراف زیر دور هامیلتونی با هزینه کمینه را پیدا کنید:



در این گراف ماتریس هزینه ها به صورت زیر میباشد:

c	1	2	3	4
1	0	10	15	20
2	5	0	9	10
3	6	13	0	12
4	8	8	9	0

$$|S|=0 \Rightarrow \begin{cases} g(2,\varnothing)=c_{2,1}=5 \\ g(3,\varnothing)=c_{3,1}=6 \\ g(4,\varnothing)=c_{4,1}=8 \end{cases}$$

$$|S|=1 \Rightarrow \begin{cases} g(2,\{3\})=c_{2,3}+c_{3,1}=15 \\ g(2,\{4\})=c_{2,4}+c_{4,1}=18 \\ g(3,\{2\})=c_{3,2}+c_{2,1}=18 \\ g(3,\{4\})=c_{3,4}+c_{4,1}=20 \\ g(4,\{2\})=c_{4,2}+c_{2,1}=13 \\ g(4,\{3\})=c_{4,3}+c_{3,1}=15 \end{cases}$$

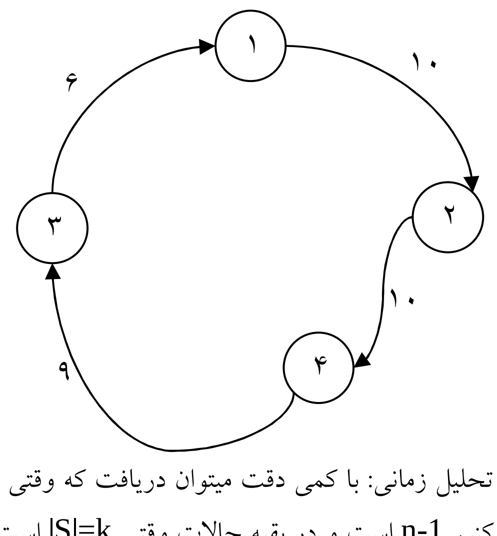
$$|S|=2 \Rightarrow \begin{cases} g(2,\{3,4\})=\min\{c_{2,3}+g(3,\{4\}),\underbrace{c_{2,4}+g(4,\{3\})}_{\min}\}=25 \\ g(3,\{2,4\})=\min\{c_{3,2}+g(2,\{4\}),c_{3,4}+g(4,\{2\})\}=25 \\ g(4,\{2,3\})=\min\{c_{4,2}+g(2,\{3\}),c_{4,3}+g(3,\{2\})\}=23 \end{cases}$$

$$|S|=3 \Rightarrow \begin{cases} g(1,\{2,3,4\})=\min\{\underbrace{c_{1,2}+g(2,\{3,4\})}_{\min},c_{1,3}+g(3,\{2,4\}),c_{1,4}+g(4,\{2,3\})\}=35 \end{cases}$$

تولید دور کمینه: حال با توجه به اینکه مقدار ۳۵ بر اساس کمینه بودن کدام مقدار تولید شده است به سمت عقب بازمیگردیم.

$$\Rightarrow 35=c_{1,2}+\underbrace{g(2,\{3,4\})}_{c_{2,4}+\underbrace{g(4,\{3\})}_{c_{4,3}+c_{3,1}}}=c_{1,2}+c_{2,4}+c_{4,3}+c_{3,1}$$

بنابراین دور کمینه بصورت زیر است:



تحلیل زمانی: با کمی دقت میتوان دریافت که وقتی |S|=n-1 باشد تعداد مقادیری که باید روی آنها مینیمم محاسبه کنیم n-1 است و در بقیه حالات وقتی |S|=k است در رابطه g(i,S) ، تعداد حالاتی که متغیر i میتواند داشته باشد n-1 حالت و تعداد حالاتی که میتوان یک مجموعه k عضوی از n-1 عضو انتخاب کنیم برابر با ترکیب k از n-1 میباشد. از این رو مرتبه زمانی الگوریتم برابر با مقدار زیر خواهد بود:

$$(n-1)+\sum_{k=0}^{n-2}(n-1)\binom{n-1}{k}$$

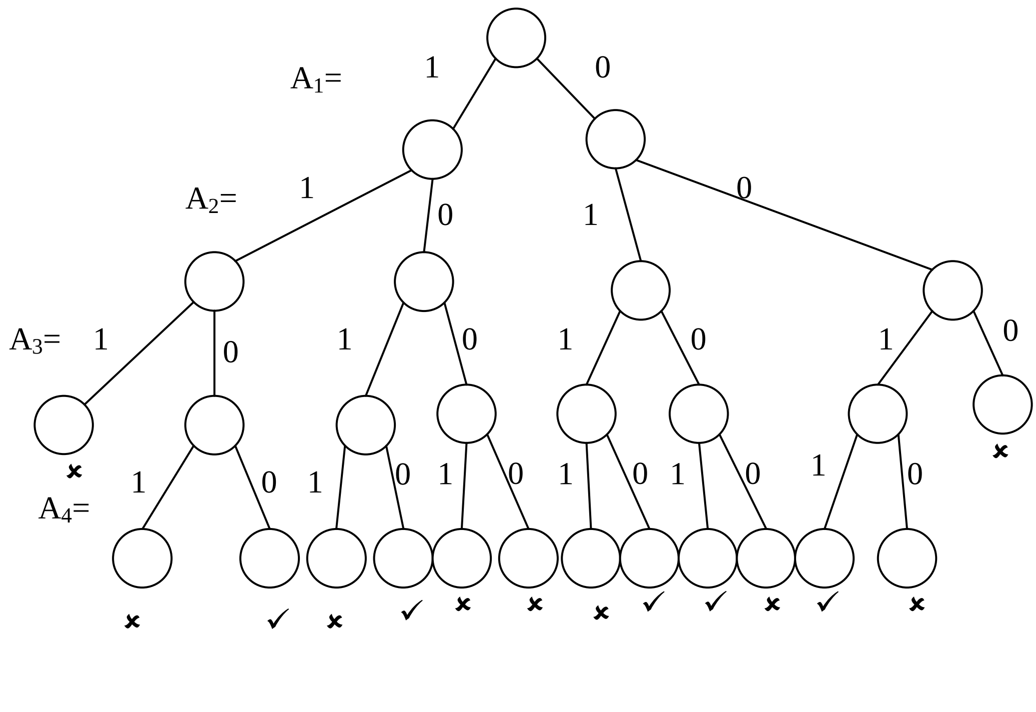
۶-روش عقبگرد (Backtracking)

در بعضی شرایط در هنگام حل مسئله نمیتوان از روش خاصی استفاده کرد و از این رو لازم میشود که فضای حالات بطور کامل جستجو شده تا جواب مسئله مشخص شود. در این شرایط از روش خاصی به نام روش پی-جویی به غقب یا روش عقبگرد استفاده میشود. در این روش درخت فضای حالت به روش "عمق اول" جستجو میشود به این معنی که تا زمانی که از اشتباه بودن یک مسیر مطمئن نشدهایم آن مسیر را ادامه داده و در صورت اطمینان از اشتباه بودن آن یک مرحله به عقب بازگشته و دوباره کار را ادامه میدهیم. این روند ادامه مییابد تا به جواب نهایی برسیم.

این روش یک روش غیرهدفمند میباشد که در حل مسئله باید تمام حالات را درنظر بگیریم در صورتیکه بتوانیم یک سری شروط بنا به مقتضیات مسئله، اضافه کنیم تا کل فضای حالت پیمایش نشود میتوان الگوریتم با زمان واقعی کمتری بدست آورد ولی مرتبه زمانی الگوریتم کاهش نمیابد.

مثال: اعداد ۴ بیتی را پیدا کنید که تعداد یکهای آنها دقیقاً ۲ تا باشد.

برای این مسئله فضای حالت را رسم میکنیم:



در فضای حالت هنگامی که مطمئن هستیم که این مسیر به جواب درست منجر نخواهد شد عمل عقبگرد صورت میگیرد که در شکل با علامت ✖ مشخص شده است. توسط کد زیر میتوان فضای حالت رسم شده در بالا را تشکیل داد و به جواب رسید:

```
procedure BT(A, i, n)
   f1←sum(A,n)
   if i>n then
      if f1= 2 then write(A) endif
   else
      if f1>2 or 2-f1>n-i then return endif        (*)
      A[i] ←1
      BT(A,i+1,n)
      A[i] ←0
      BT(A,i+1,n)
   endif
end.
```

نکته: فراخوانی اولیه رویه بالا باید بصورت BT(A,1,4) صورت گیرد.

نکته: تابع sum در کد بالا تعداد یکهای رشته A را تا مرحله فعلی (i-1 ام) محاسبه میکند.

نکته: شرط مشخص شده در (*) بررسی میکند که در صورتی که مسیر فعلی منجر به جواب درست نخواهد شد عمل بازگشت به عقب (Backtrack) را انجام میدهد. شرط اول آن نشاندهنده بیش از حد مجاز بودن تعداد یکهاست و شرط دوم آن نشاندهنده کمتر از حد مجاز بودن تعداد بیتهای مشخص نشده تا بحال میباشد که هر دو شرط دلیل غلط بودن مسیر فعلی میباشد.

نکته: مرتبه زمانی کد بالا $O(2^n)$ میباشد.

نکته: هرچه تعداد شرطهای دستور (*) بیشتر باشد یعنی در درخت فضای حالت زودتر عقبگرد میکنیم و این منجر به کم شدن زمان واقعی الگوریتم خواهد شد.

حال به ارائه مثالی دیگر در این رابطه میپردازیم.

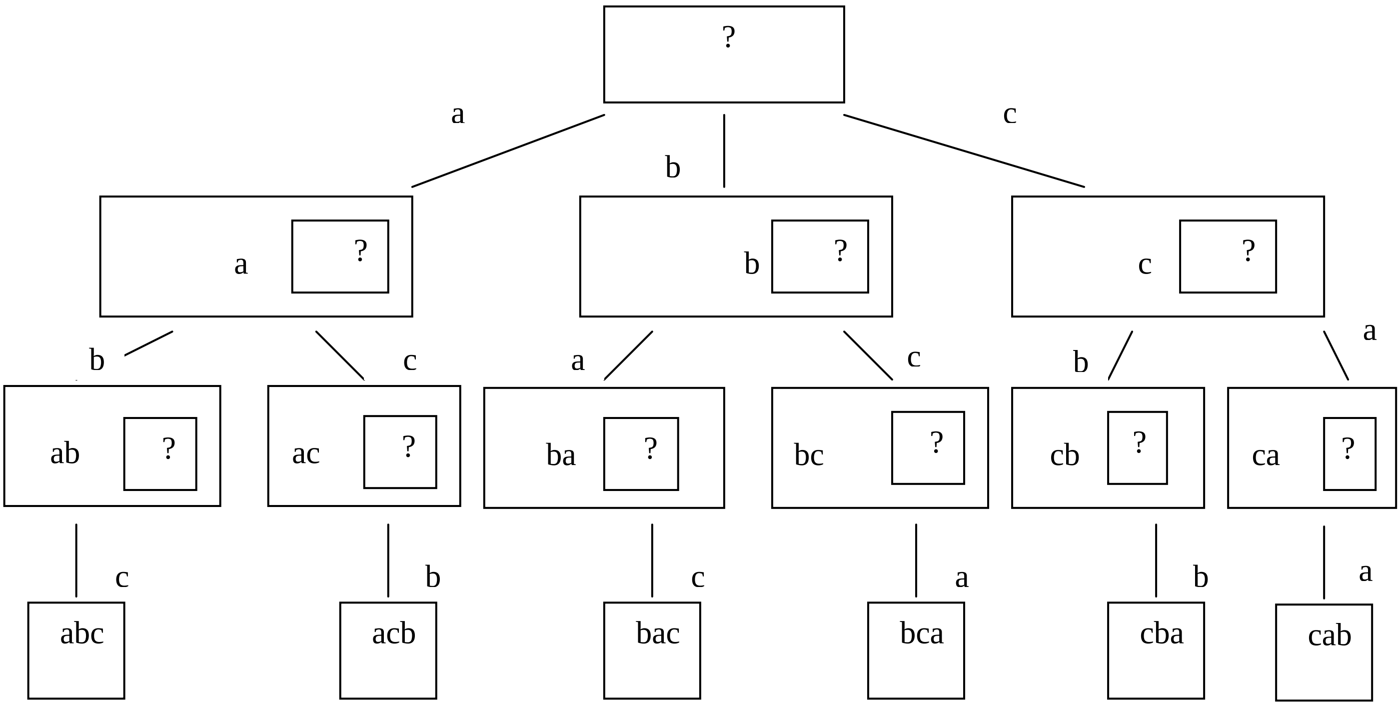
۶-۱- مولد ترکیبات

در این الگوریتم مجموعه با $n \geq 1$ عضو وجود دارد و تصمیم داریم تمام ترکیبات ممکن آن را داشته باشیم. این روش دارای مرتبه زمانی (n!) می‌باشد. برای حل مسئله ما از روش Backtracking استفاده می‌کنیم.

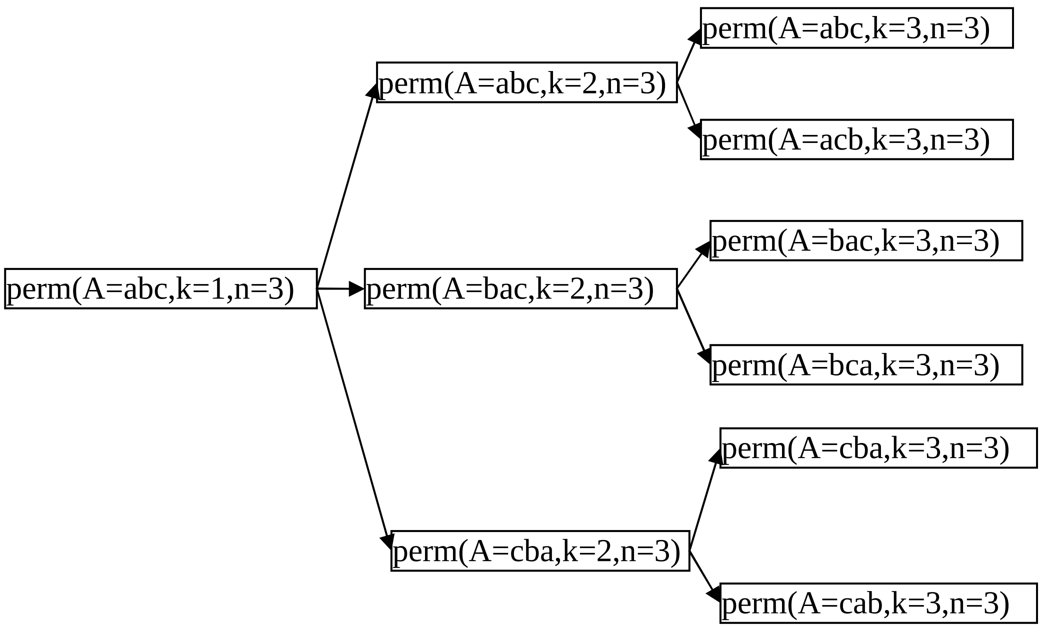
```
Procedure perm(A, k, n)
   if   k=n  then
       write(A)
   else
      for i←k  to   n    do
         swap(A[k],A[i])
         perm(A,k+1,n)
      repeat
   endif
end.
```

نکته: در فراخوانی اولیه رویه perm ابتدا باید عناصر 1 تا n را در آرایه A قـرار داده و سـپس perm(A,1,n) را فراخوانی کنیم.

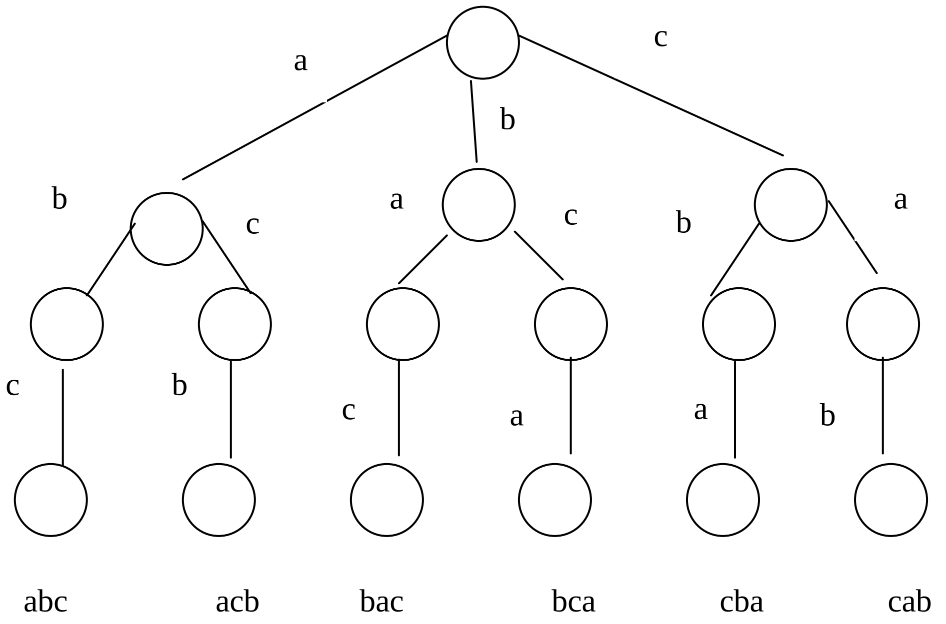
مثال: تمام ترکیبات ممکن a,b,c را به وسیله پروسیجر perm تولید کنید؟



نحوه فراخوانی بازگشتی رویه بصورت شکل زیر میباشد:



یا شکل ساده‌تر



۶-۲- مسئله ***n*** وزیر

حال به طرح مسئله‌ای میپردازیم که در ارائه الگوریتم برای آن از روش عقبگرد استفاده میکنیم. در این مسئله هدف قرار دادن n مهره وزیر در صفحه شطرنج (n*n) به گونه‌ای است که هیچیک از وزیران دیگری را تهدید نکند. در مسئله n وزیر فرض در هیچ سطر و ستونی بیش از یک وزیر قرار نگیرد. حال با این فرض تمام جایگشت‌های ممکن را تولید می‌کنیم یعنی همان درخت حل مسئله یا فضای حالت را بوجود می‌آوریم سپس در هرمرحله بررسی می‌کنیم که مسیر فعلی در درخت با این نحوه‌ای چیدمان وزیرها آیا دارای شرایط مسئله می‌باشد یعنی وزیرها یکدیگر را تهدید می‌کنند یا خیر. برای مثال برای n=4 مسئله دارای دو جواب بصورت زیر است.

جواب اول:

	×		
			×
×			
		×	

که توسط آرایه زیر قابل نمایش است:

۲	۴	۱	۳

جواب دوم:

×			
		×	
			×
	×		

که توسط آرایه زیر قابل نمایش است:

۳	۱	۴	۲

اگر n برابر ۵ باشد ۵ جواب برای مسئله وجود دارد و برای n=8 ۹۲ جواب وجود دارد.

نکته: چون در هر لحظه در یک سطر و یک ستون نمیتواند بیش از یک وزیر قرار گیرد لذا استفاده از یک ماتریس مورد لزوم نیست و فقط یک آرایه یک بعدی n عنصری کفایت میکند. در این آرایه نیز اگر فرض زیر را انجام دهیم:

A[i] = ستون قرار گرفتن وزیر i ام

آنگاه عناصر تکراری در آرایه نباید وجو داشته باشد و از این رو میتوان از همان رویه perm برای تولید همه جایگشتها در آرایه استفاده کرد و فقط هنگام چاپ خروجی شرطی را برای چاپ در خروجی قرار می‌دهیم که همان تابع test می‌باشد که کد در زیر آورده شده است. بررسی اینکه آیا وزیرهای قرار گرفته در آرایه همدیگر را تهدید میکنند یا خیر به راحتی توسط رابطه زیر قابل انجام است. به این معنی که اگر عناصر قرار گرفته در آرایه دارای این خاصیت باشند هیچگاه دو وزیر یکدیگر را تهدید نمیکنند:

$$\forall i, j \mid i \neq j \Rightarrow |A[i] - A[j]| \neq |i - j|$$

```
Procedure Queen(A, k, n)
   if   k=n  then
      if test(A,n) then write(A) endif
   else
      for i←k  to   n    do
         swap(A[k],A[i])
         perm(A,k+1,n)
      repeat
   endif
end.
```

```
function test(A,n)
   for i←1 to n do
      for j←i+1 to n do
         if | A[i] – A[j] | = | i-j | then return false endif
      repeat
   repeat
   return true
end.
```

نکته: الگوریتم Queen دارای مرتبه زمانی O(n!) می‌باشد.

٦-٣- تعیین نقاط روی محور xها از روی فواصل آنها

ورودی: مجموعه فواصل n نقطه $\left(x_1, x_2, ..., x_n\right)$

خروجی: $x_1 \le x_2 \le ..... \le x_n$

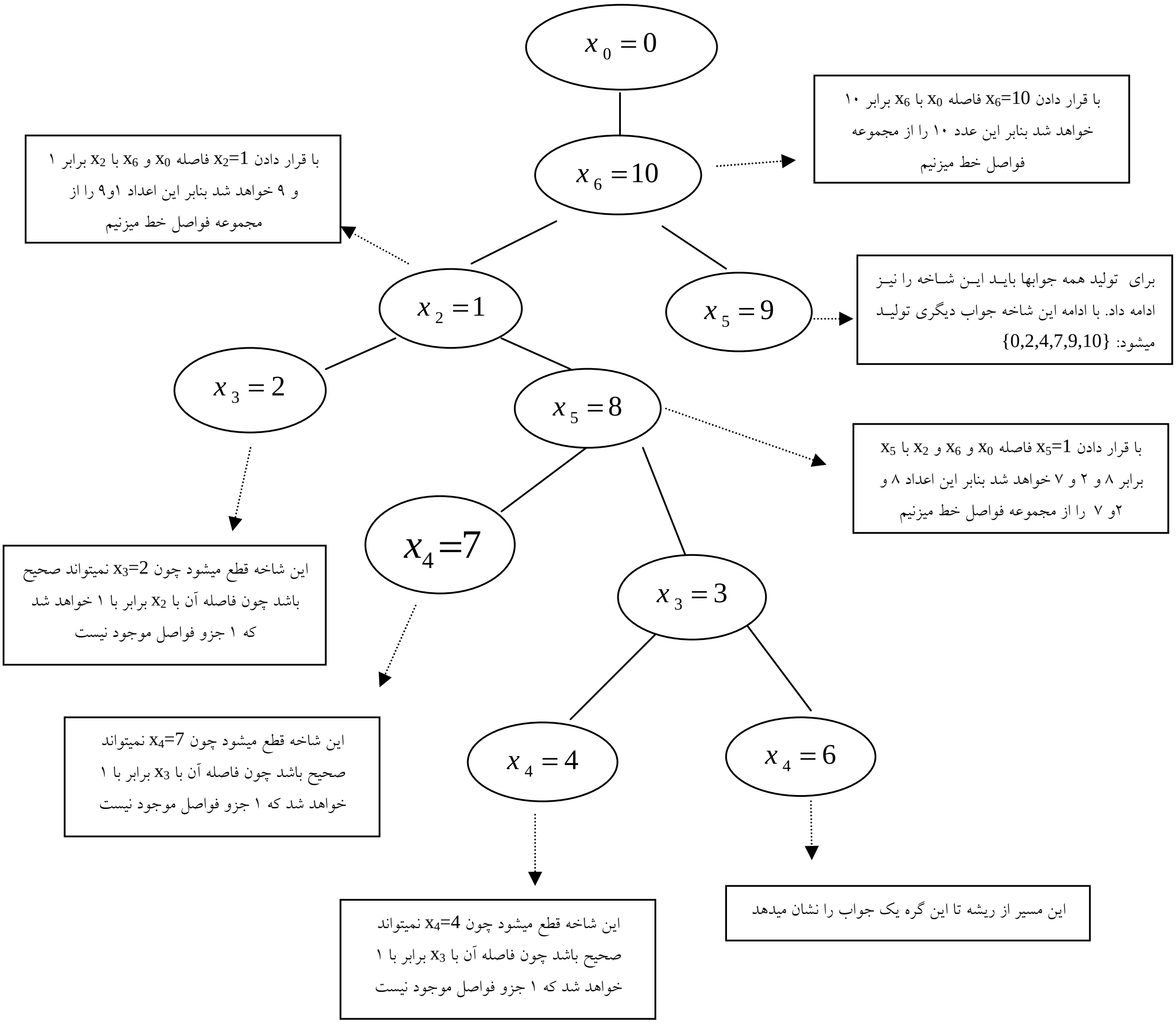
میتوان توسط رابطه زیر تعداد فواصلی که n نقطه میتوانند با یکدیگر تولید کنند را بدست آورد:

$$|D| = \frac{n\left(n-1\right)}{2} = \binom{n}{2}$$

برای جلوگیری از عدم تولید جوابهای بدیهی و منفی فرض میکنیم که $X_1=0$ و بقیه نقاط همگی مثبت هستند.

مثال: $D = \{1, 2, 2, 2, 3, 3, 4, 5, 5, 6, 7, 7, 8, 9, 10\}$

در هر مرحله بزرگترین فاصله را از مجموعه فواصل انتخاب کرده، این عدد یا فاصله اولین نقطه (x_0) با آخرین نقطه نامعلوم و یا فاصله آخرین نقطه (x_6) با اولین نقطه نامعلوم میباشد. (در هر گره مسیر از ریشه تا آن گره نشان دهنده نقاط معلوم تا آن لحظه میباشند.) با در نظر گرفتن این دو حالت به دو نقطه معلوم جدید مرسیم و فواصل بوجود آنده توسط آن نقاط را با نقاط معلوم قبلی (در صورت وجود) از مجموعه فواصل خط میزنیم. اگر این فواصل موجود نبودند شاخه قطع شده و شاخه دیگری را ادامه میدهیم.



جواب اول:

x_1	x_2	x_3	x_4	x_5	x_6
۰	۱	۳	٦	۸	۱۰

توضیح: نود اول را فرض می‌کنیم صفر است بزرگترین فاصله، فاصله نود اول و آخر می‌باشد. این فاصله را از مجموع فواصل حذف کرده و دوباره از مجموع فواصل باقی‌مانده ماکزیمم فاصله را پیدا می‌کنیم که ۲ حالت پیش می‌آید:

۱- فاصله رأس اول تا نود n-1 ام برابر این فاصله می‌باشد.

۲- فاصله رأس آخر با نود دوم برابر این فاصله می‌باشد.

و در هرمرحله هر فاصله‌ای که هر نقطه با نقاط دیگر پیدا می‌کند از مجموعه فواصل حذف می‌کنیم و این کار را تـا تمامی این فاصله‌ها حذف شوند ادامه می‌دهیم.

اگر فاصله‌ای در مجموع فواصل نبود مسیر را اشتباه آمده‌ایم شاخه را قطع کرده و شاخه دیگری را ادامه میدهیم.

نکته: مرتبه‌ای زمانی این الگوریتم $O(n^2)$ می‌باشد!.

۷-روش انشعاب و تحدید (Branch & Bound)

روش دیگری که در این قسمت به آن میپردازیم به نام روش انشعاب و تحدید یا شاخه و حد معروف است. این روش یک روش غیرهدفمند ولی هوشمند است. در این روش برخلاف روش عقبگرد تمامی حالات در فضای حالت بررسی نمی‌شود و براساس شرطی که در مسئله قرار داده می‌شود (این شروط وابسته به نوع مسئله متفاوت است) برخی از حالت‌ها بررسی نمی‌شوند.

سه تفاوت عمده بین روش غقبگرد و روش شاخه و حد وجود دارد:

۱- در روش عقبگرد درخت فضای حالت بصورت عمق اول (DFS) جستجو میشود ولی در روش شاخه و حد ابتدا کلیه فرزندان گره فعلی ساخته میشود و سپس از بین آنها یکی بسته به شرایط انتخاب میشود وتا حدودی میتوان آن را منطبق با جستجوی ردیفی (BFS) در نظر گرفت.

۲- در روش شاخه و حد یک تابع محدود کننده وجود دارد که از گسترش بیش از حد و نابجای شاخه های درخت حل مسئله جلوگیری میکند و در موقع لزوم شاخه ها را قطع کرده و مسیر فعلی را ادامه نمی دهد. درحالیکه در روش عقبگرد چنین معیاری وجود ندارد.

۳- روش شاخ و حد در مقایسه با روش عقبگرد هوشمندانه تر عمل میکند و در هنگام انتخاب شاخه جهت گسترش درخت حل مسئله، شاخه‌ای که احتمال بیشتری برای تولید جواب دارد انتخاب می‌شود.

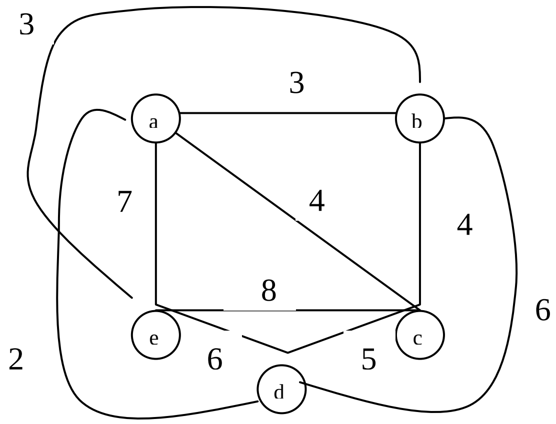
حال به ارائه الگوریتمهایی مبتنی بر این روش می‌پردازیم.

۷-۱- فروشنده دوره گرد

در این بخش به ارائه روشی مبتنی بر شاخه و حد برای مسئله فروشنده دوره گرد میپردازیم.

در یک گراف جهت‌دار تعداد دورهای هامیلتونی $(n-1)!$ است و در یک گراف غیرجهت‌دار تعداد دورهای هامیلتونی $\frac{(n-1)!}{2}$ است!.

مثال: پیدا کردن کوتاهترین دور هامیلتونی در یک گراف با ۵ رأس

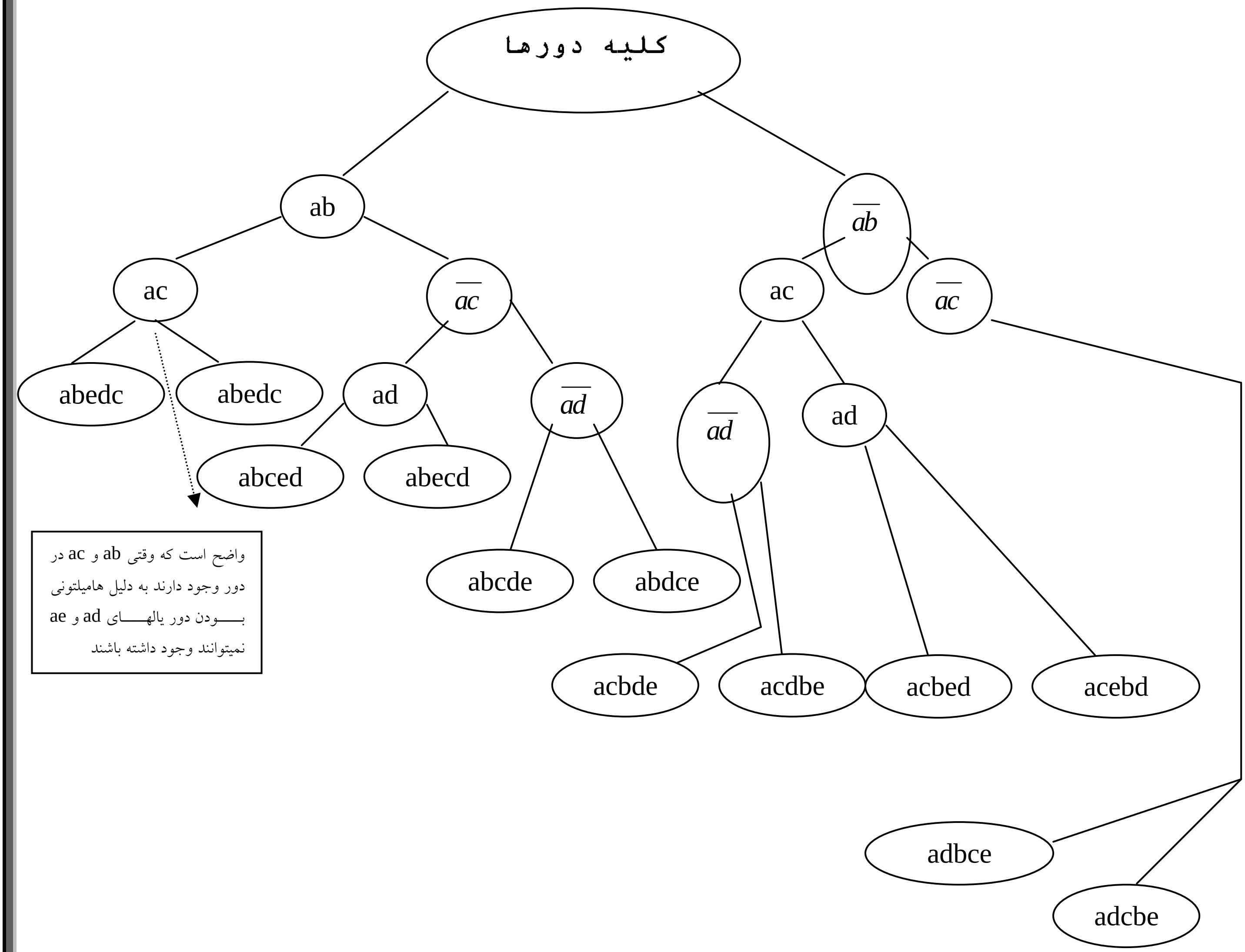


تابع محدود کننده:

$$f(G)=\frac{1}{2}\sum_{v\in V}\{c(u,v)+c(v,w):\ (u,v),(v,w)\ are\ two\ least\ \cos t\ \ edges\ \ adjacent\ to\ \ vertex\ v\}$$

واضح است که مقدار تابع بالا در گراف G، حد پایینی برای هزینه دورهای هامیلتونی خواهد بود چون برای هر راس دو یال با کمترین هزینه که مجاور به آن راس هستند را انتخاب کرده است. لذا در گسترش درخت حل مسئله هر دوری که هزینه آن به مقدار تابع بالا نزدیکتر بود زودتر جهت گسترش انتخاب میشود.

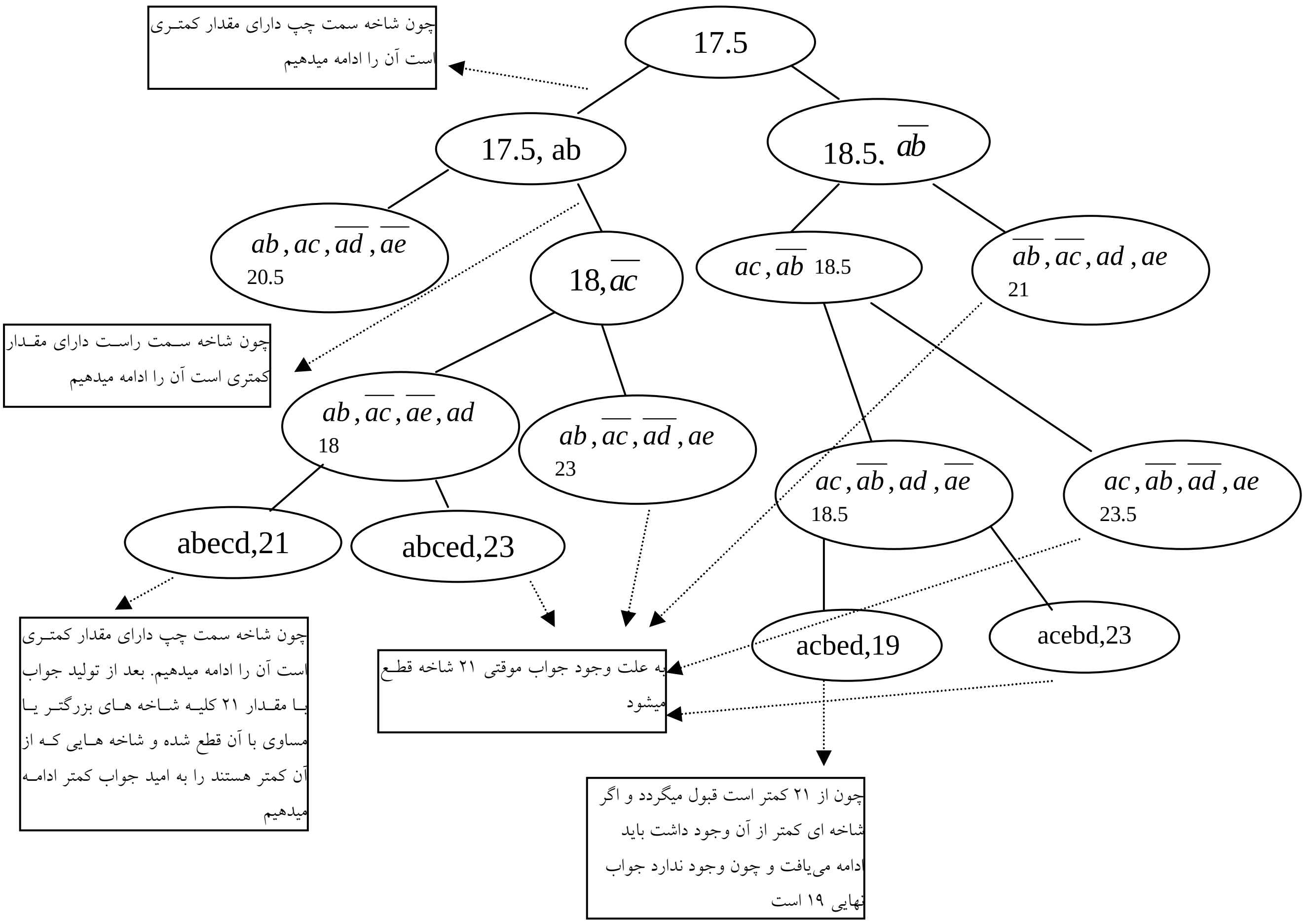
درخت فضای حالات برای گرافی با ۵ رأس در حالت کلی به شکل زیر می‌باشد:



توضیح:

دریک گراف برای هر رأس ۲ یال با کمترین هزینه را انتخاب می‌کنیم مجموع هزینه‌ها را محاسبه کرده سـپس بـردو تقسیم می‌کنیم تا کمترین هزینه‌ای که ممکن است بدست آید.

راس	اولـــین مینیمم	دومین مینیمم
a	2	3
b	3	3
c	4	4
d	2	5
e	3	6
	35/2=17/5	



طریقه محاسبه ارزش هرگره:

بر اساس تابع محدود کننده که در بالا شرح داده شد باید با توجه به محدودیتهای هر مسیر از ریشه تا آن گره تابع را مجدداً محاسبه کرد. به بیان دیگر با اضافه شدن شرایط جدید مقدار تابع بیشتر یا مساوی مقدار گره پدر خواهد بود.

وقتی این رویه را تا انتها ادامه دهیم دور هامیلتونی با کمترین هزینه بدست می‌آید ولی برای هوشمند شدن الگوریتم کل فضای حالات را پویش نمی‌کنیم بلکه از بین نودهایی که داریم نودی را که دارایی تفاوت کمتری تا گره ریشه است را گسترش میدهیم تا به برگ برسیم، در هر مرحله کوچکترین گره را ادامه میدهیم و با رسیدن به یک برگ (جواب موقتی) کلیه گره های بیشتر یا مساوی آن را قطع میکنیم.

نکته: الگوریتم‌های Branch & Bound از لحاظ مرتبه زمانی تفاوتی با الگوریتم‌های Backtraking ندارند ولی از زمان واقعی کمتری برخوردارند.

نکته: در درخت حل مسئله بالا فقط ۴ دور مختلف محاسبه شدند درحالی که در روش عقبگرد باید تمام ۱۲ دور مختلف را تولید میکردیم.

۷-۲- جمع زیرمجموعه‌های یک مجموعه

حال به بیان مسئله دیگری که توسط روش شاخه و حد میتوان برای آن الگوریتم جالبی ارائه کرد میپردازیم. در این مسئله هدف پیدا کردن زیر مجموعه‌ای از یک مجموعه مفروض است که مجموع هعضای آن مقدار مشخصی باشد.

ورودی:

مجموعه $S=\{s_1, s_2, \ldots, s_n\}$

عدد مفروض M

خروجی:

$X=\{x_1, x_2, \ldots, x_n\}$

$x_i = 0$ یا 1 $(1\le i \le n)$

$$\sum_{i=1}^{n} x_i . s_i = M$$

فرض: عناصر آرایه S صعودی هستند.

تابع محدود کننده:

عناصر آرایه X انتخاب یا عدم انتخاب عناصر متناظر در S را نشان میدهند. فرض کنید که عناصر آرایه X تا k-1 امین عنصر یعنی x_{k-1} مشخص شده اند و نوبت به مشخص شدن k امین عضو از X یعنی x_k رسیده است. باید شرایط زیر برقرار باشند. به بیان دیگر اگر در شاخه‌ای شرایط زیر برقرار نبود آن شاخه به جواب منجر نخواهد شد و باید شاخه قطع شود.

1	2	…	k-1	k		n
x_1	x_2	…	x_{k-1}			

۱- (به دلیل صعودی بودن عناصر S) $\sum_{i=1}^{k-1} x_i.s_i + s_k \le M$

۲- $\sum_{i=1}^{k-1} x_i.s_i + \sum_{i=k}^{n} s_i \ge M$

توضیح:

شرط ۱: بیان کننده این مطلب می‌باشد که اگر مجموع وزن عناصری که تاکنون انتخاب شده‌اند به اضافه‌ای وزن عنصر بعدی، از M بیشتر شود چون عناصر صعودی هستند وزن عناصر بعدی از عنصر k بیشتر است پس ادامه نمی‌دهیم.

شرط ۲: بیان کننده این مطلب می‌باشد که اگر مجموع وزن عناصری که تاکنون انتخاب شده‌اند به اضافه‌ای وزن کل عناصر باقی مانده کمتر از مقدارM باشد اگر تمام عناصر انتخاب کنیم به مقدار M نمی‌رسیم پس ادامه نمی‌دهیم.

مثال:

$S = \{7,10,12,13,15,18\}$ M=30

n=6

S: مجموع وزن عناصر انتخاب شده تا به حال

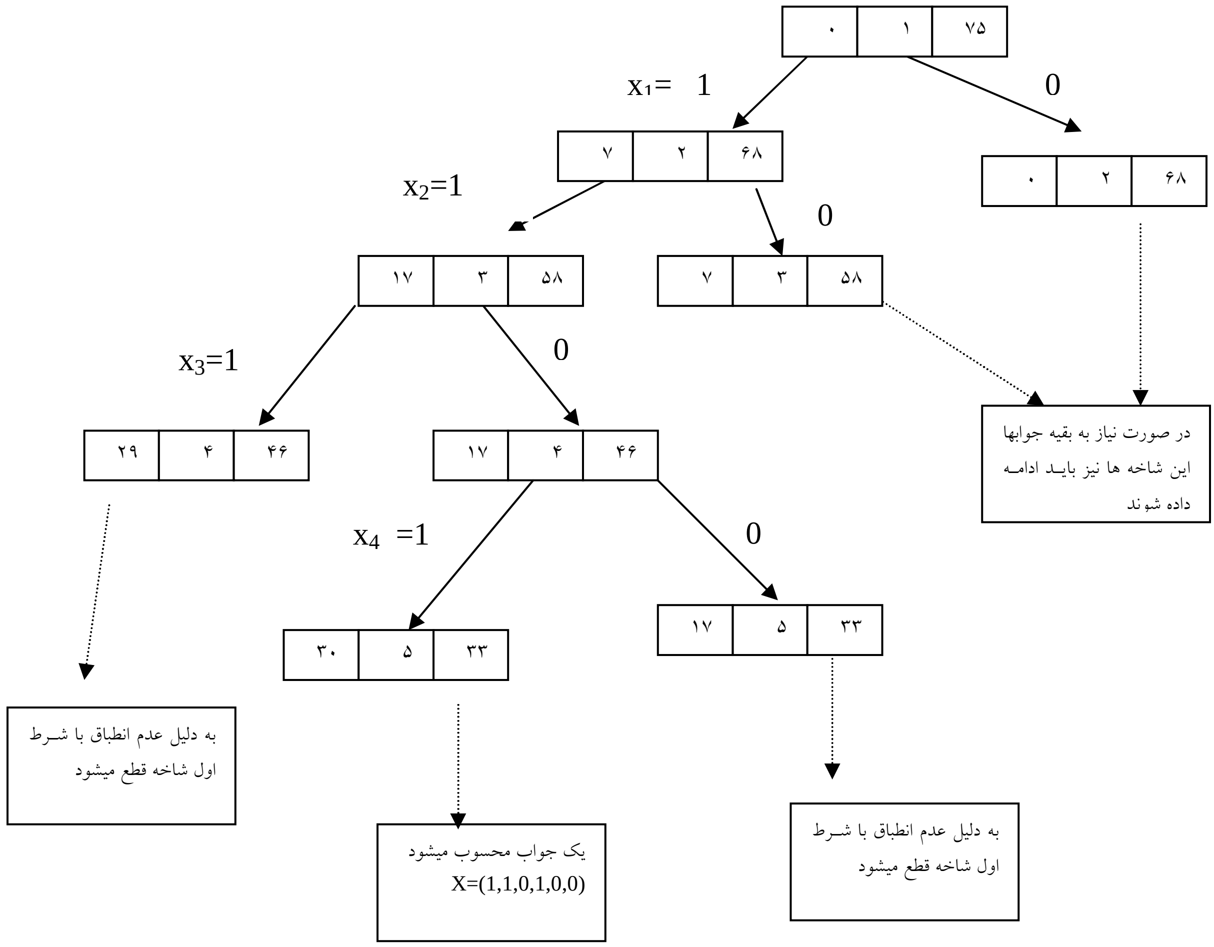
K: عنصری که روی آن در حال تصمیم گیری هستیم

r: مجموع هزینه‌های از رأس k به بعد

نکته: هر گره از درخت حل مسئله را بصورت زیر فرض میکنیم:

s	k	r

$$s=\sum_{i=1}^{k-1} s_i . x_i \qquad r=\sum_{i=k}^{n} s_i$$



نکته: عدد یک در شاخه ها نشان‌دهنده انتخاب شدن عنصر و عدد صفر نشان دهنده انتخاب نشدن عنصر میباشد.

نکته: مرتبه زمانی الگوریتم $O(2^n)$ میباشد!

۸- پیچیدگی محاسبات

در این بخش به دسته بندی مسائل از لحاظ سادگی و سختی میپردازیم. ابتدا چند تعریف ارائه میدهیم:

مسائل تصمیم گیری: مسائلی که خروجی آنها دارای دو حالت باشد.

مثال: گراف وزن داری مفروض است. تشخیص اینکه آیا هزینه دور هامیلتونی مینیمم این گراف از مقدار k کمتر است یا خیر یک مسئله تصمیم گیری محسوب میشود.

کلاس P: مجموعه مسائل تصمیم گیری که برای آنها یک الگوریتم قطعی با زمان چند جمله‌ای وجود دارد.

کلاس NP: مجموعه مسائل تصمیم گیری که برای آنها یک الگوریتم غیرقطعی با زمان چند جمله‌ای وجود دارد.

مثال:

تشخیص مرتب بودن یک آرایه

تشخیص اینکه یک رشته مفروض شامل زیر رشته خاصی هست یا خیر

تشخیص همبند بودن یک گراف

همگی جزو کلاس P هستند.

مثال:

تشخیص مرتب بودن یک آرایه

تشخیص اینکه یک رشته مفروض شامل زیر رشته خاصی هست یا خیر

تشخیص همبند بودن یک گراف

تشخیص اینکه آیا بیشترین سود حاصل در مسئله کوله پشتی صفر و یک از مقدار خاصی بیشتر است

تشخیص اینکه یک مجموعه مفروض را میتوان به دو قسمت به گونه ای افراز کرد که مجموع دو قسمت مساوی باشد(این مسئله به نام PARTITION شناخته میشود).

تشخیص اینکه میتوان در یک مجموعه مفروض یک زیر مجموعه پیدا کرد که مجموع عناصر آن برابر با عدد داده شده M باشد(این مسئله به نام Sum of Subset شناخته میشود).

همگی جزو کلاس NP هستند.

نکته: هر مسئله P متعلق به NP نیز میباشد به بیان دیگر همیشه $P \subseteq NP$. چون اگر برای مسئله ای الگوریتم قطعی چند جمله ای وجود داشته باشد میتوان همان الگوریتم را عیر قطعی هم در نظر گرفت.

هر مسئله قطعی، غیرقطعی آن نیز وجود دارد.

مسائل بغرنج (intractable): مسائلی که ثابت شده است که متعلق به NP نیستند.

مثال: مسئله تشخیص اول بودن یک عدد n بیتی بغرنج است.

کاهش (Reduce) یا تبدیل: گوییم مسئله A در زمان چند جمله ای به B تبدیل میشود ($A \propto_p B$) اگر یک الگوریتم برای B را بتوان در زمان چند جمله ای به الگوریتمی برای A تبدیل کرد و یا به بیان دیگر یک جواب برای یک نمونه مسئله از B در زمان چند جمله ای منجر به یک جواب برای یک نمونه مسئله از A شود.

مثال:

Sorting $\propto_p$ Convex Hull
PARTITION $\propto_p$ Sum of Subset

مرتب کردن n عدد توسط الگوریتم پوسته محدب: اعداد ورودی را مختصه x برای n نقطه در نظر میگیریم و به عنوان مختصه y مقدار x^2 را در نظر میگیریم. حال پوسته محدب n نقطه را میسازیم. قابل اثبات است که ترتیب x نقاط روی پوسته محدب مرتب شده اعداد اولیه هستند.

مثال:

٥	٣	٢	١	٤
٢٥	٩	٤	١	١٦

نکته : از قابل تبدیل بودن مسئله مرتب سازی به پوسته محدب میتوان اثبات کرد که بهترین الگوریتم برای تولید پوسته محدب دارای زمان $O(n. \log n)$ میباشد!. چون اگر بتوان پوسته محدب را در زمان کمتر از $O(n. \log n)$ تولید کرد پس مرتب سازی هم در کمتر از $O(n. \log n)$ امکان پذیر است که این غیر ممکن است چون مرتب سازی n عدد توسط الگوریتمهای مقایسه ای در بدترین حالت $\Omega(n. \log n)$ است.

مسئله NP-Complete:

$$A \in NPC \begin{cases} A \in NP \\ \exists B \in NPC \mid B \alpha A \end{cases}$$

مثال:

مسئله کوله پشتی ۰/۱

مسئله فروشنده دوره‌گرد

مسئله PARTITION

مسئله وجود ویا عدم وجود دور هامیلتونی در گراف

مسئله Sum of Subset

مسئله رنگ آمیزی گراف

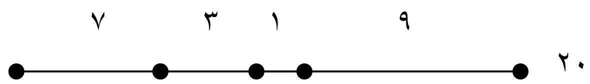
مسئله تا کردن خط کش (Ruler Folding)

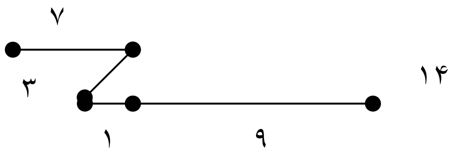
مسئله پیدا کردن مسیری به طول k در یک گراف وزن دار
همگی NP-Complete هستند.

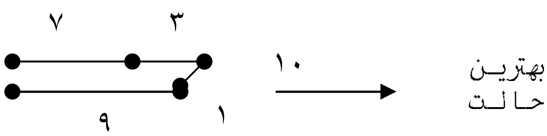
۸-۱- مسئله تا کردن خط کش

یک پاره‌خط با تعدادی مفصل بر روی آن داریم که هر پاره خط میتواند حول مفاصل دو طرفش چرخش تا ۳٦۰ درجه را داشته باشد. هدف کوتاهترین طول حاصل از تا کردن این پاره خطها بر روی خط افقی است.

مثال:



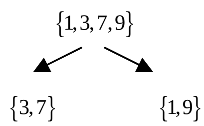




۸-۲- مسئله افراز (PARTITION)

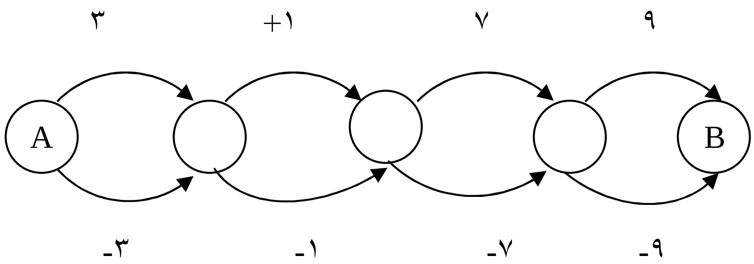
در این مسئله مجموعه‌ای از اعداد داده شده است و هدف افراز مجموعه به دو قسمت به گونه ای است که مجموع عناصر دو مجموعه مساوی باشد.

مثال: با ورودی مجموعه زیر مسئله پارتیشن به نحوه‌ای که دارای طول‌های یکسانی باشند.(هزینه‌های یکسان)

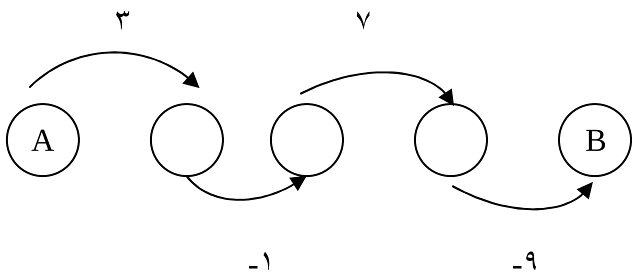


مثال: ثابت کنید مسئله افراز یک مجموعه قابل تبدیل به مسئله پیدا کردن مسیری به طول k در گـراف اسـت. بـرای این کار گرافی میسازیم که دارای n+1 راس است. بین راس i-1 ام و i ام دو یال جهتدار برقـرار میکنـیم. یـال اول دارای هزینه عنصر i ام از مجموعه اعداد داده شده و یال دوم دارای مقدار قرینه یال اول میباشـد. حـال پیـدا کـردن مسیری به طول صفر در این گراف منجر به افراز مجموعه اولیه به دو قسمت با مجموع یکسان خواهد شد!. انتخاب عناصر متناظر با یالهای اول از هر راس به راس بعدی در مسیر مورد نظر منجر به افراز مورد نظر خواهد شد.

فرض کنید هدف افراز مجموعه {3,1,7,9} میباشد. گرافی بصورت زیر سـاخته و در آن مسـیری از A بـه B بـه طول صفر را بررسی میکنیم:



دو مسیر با هزینه صفر وجود دارد که یکی از آنها در زیر نشان داده شده است:



میتوان نتیجه گرفت مجموعه قابل افراز به {3,7} و {1,9} میباشد.

پایان

نکته: خواهشمندم اشکالات موجود در مطالب را به آدرس nourollah@cic.aut.ac.ir ارسال فرمایید.

و من الله توفیق

منابع و مراجع

[1] G. Brassard and P. Bratley, "Fundamentals of Algorithmic", Prentice-Hall, 1996.

[2] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf, "Computational Geometry: Algorithms and Applications", SpringerVerlag, 1997.

[3] T. H. Corman, C. E. Leiserson, R. L. Rivest, and C. Stein, "Introduction to Algorithms", MIT Press and McGraw-Hill, 2001.

[4] E. Horowitz, S. Sahni, S. Rajasekaran, "Fundamentals of Computer Algorithms", Computer Science Press, 1996.

[5] O'Rourke, Joseph, "Computational Geometry in C", Cambridge University Press, Cambridge, UK, 1993.

[6] Preparata, F. P. and M. I. Shamos, "Computational Geometry: An Introduction", Spinger-Verlag, New York, 1985.