طراحی

الگوریتم

Ǵ¶ƏƷ ǂ ǿř Ǫ¶ǽƷÌŇŬǩǂ - ǫƿY | Ʊƛ

www.toghraee.ir

۱- یادآوری

تعریف الگوریتم: هر دستوالعملی که مراحل مختلف انجام کاری را به زبان دقیق و با جزئیات کافی بیـان نمایـد بـه 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 عنصـر j ام در عناصـر 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 کـه x[3],x[2],x[1] مرتـب شـده 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!.

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

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 رئوس (به دلخواه) است، اضافه میشوند. در هر مرحله یال با کمترین هزینه که یک راس مجاور آن داخل T و راس مجاور دیگرش 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 گره قبلی به تعویق افتاده و گره جدید بررسی می شود. در واقع در این روش با گذر از هر گره به سمت عمق درخت گره فعلی در پشته قرار داده میشود که با بازگشتی طراحی کردن الگوریتم پشته مورد نظر به صورت پنهان ساخته 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 میشود (!) و هر راس نیز 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 نام دارد. برای پیمایش کلیه گره ها گره آغاز را انتخاب می کنیم. در این روش بعد از دیدن هرگره کلیه فرزندان همان گره 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 برداشـته 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.