دانشکده‌ی علوم ریاضی

ساختمان داده ۱۲ آبان ۹۲

جلسه‌ی ۱۳آ: درخت مقایسه، پایداری

مدرّس: دکتر شهرام خزائی *نگارنده: فرزاد جعفررحمانی*

۱ مقدمه

تاکنون چندین الگوریتم مرتب‌سازی[۱] که می‌تواند n عدد را در زمان $O(n \lg n)$ مرتب کند بیان شده است، مانند مرتب‌سازی هرمی و مرتب‌سازی سریع. این الگوریتم‌ها در ویژگی مقایسه‌ای اشتراک دارند. به همین دلیل تحت عنوان الگوریتم‌های مرتب‌سازی مقایسه‌ای از آنها یاد می‌شود.
در این جلسه قرار است که نشان دهیم که هر الگوریتم که از روش مقایسه استفاده می‌کند نمی‌تواند پیچدگی زمانی کمتر از $O(n \lg n)$ داشته باشد. برای این کار نشان می‌دهیم که هر مرتب‌سازی مقایسه‌ای برای مقایسه n عدد در بدترین حالت باید $n \lg n$ مقایسه انجام دهد.

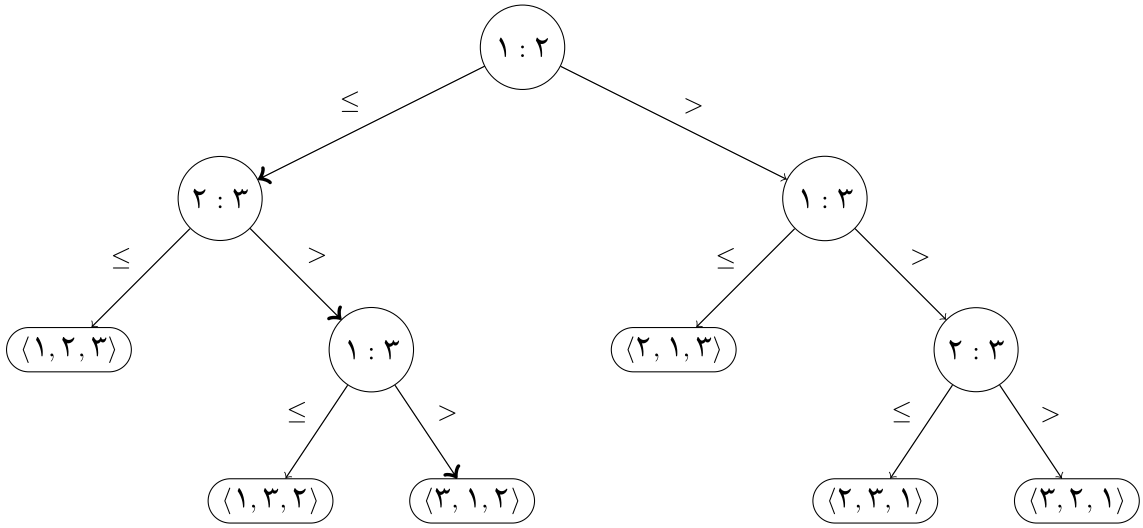
۲ حدهای پایین برای مرتب‌سازی

در مرتب‌سازی مقایسه‌ای تنها از مقایسه دو عنصر برای دست‌یابی به اطلاعات درمورد ترتیب نهایی آنها در آرایه مرتب‌شده از آرایه ورودی $\langle a_1, a_2, ..., a_n \rangle$ استفاده می‌کنیم.
در این بخش بدون از دست دادن کلیت مساله فرض می‌کنیم همه عناصر ورودی متفاوت هستند. برای رسیدن به هدف مورد نظر لازم است ابتدا درخت مقایسه را بیان کنیم.

۳ درخت مقایسه

درخت مقایسه یک درخت دودویی است که ترتیب مقایسه بین عناصر را در روند مرتب کردن آرایه ورودی نمایش می‌دهد. هر مرتب‌سازی مقایسه‌ای را می‌توان به طور خلاصه برحسب یک درخت مقایسه در نظر گرفت که از جنبه‌های دیگر الگوریتم مانند جابه‌جایی صرف نظر شده است. در شکل زیر درخت مقایسه معادل با الگوریتم مرتب‌سازی درجی روی یک دنباله ورودی سه عضوی نشان داده شده است.

[۱] sorting algorithm



در این درخت هر برگ با جایگشت $\langle \sigma(۱), \sigma(۲), ..., \sigma(n) \rangle$ نشان داده‌می‌شود. اجرای الگوریتم روی یک ورودی، متناظر با طی‌کردن مسیری از ریشه تا برگ می‌باشد. در هر گره داخلی، مقایسه $a_i \leq a_j$ صورت می‌گیرد که با $i : j$ در شکل نشان داده شده‌است. آنگاه زیر درخت سمت چپ مسیر را برای حالت $a_i \leq a_j$ و زیر درخت سمت راست مسیر را برای حالت $a_i > a_j$ ادامه می‌دهد. وقتی که به یک برگ می‌رسیم الگوریتم مرتب‌سازی، ترتیب نهایی را تعیین کرده است. چون هر الگوریتم مرتب‌سازی صحیح باید هر جایگشت از ورودی‌ها را مرتب کند، پس هر یک از $n!$ جایگشت از n عدد ورودی باید در یک برگ از درخت مذکور باشد. به عبارت دیگر از ریشه باید مسیری به این برگ‌ها باشد یا اصطلاحا همه برگ‌ها قابل دست‌یابی باشند.
برای مثال فرض کنید ترتیب عناصر در آرایه ورودی به صورت $a_۱ < a_۳ < a_۲$ باشد. از ریشه که شروع کنیم، چون $a_۱ < a_۲$ پس به سمت زیر درخت چپ می‌رویم. سپس چون $a_۲ > a_۳$ به سمت راست می‌رویم. نهایتاً چون $a_۱ < a_۳$ به گره آخر می‌رسیم که نشان می‌دهد اگر ترتیب $\langle ۱, ۳, ۲ \rangle$ در آرایه ورودی اعمال شود آرایه مورد نظر مرتب می‌شود.

حد پایین برای بدترین حالت: طول طولانی‌ترین مسیر از ریشه تا برگ تعداد مقایسه‌هایی را که یک الگوریتم مرتب‌سازی در بدترین حالت انجام می‌دهد نشان می‌دهد. بنابراین حداکثر تعداد مقایسه‌های یک الگوریتم مرتب‌سازی مقایسه‌ای برابر است با ارتفاع درخت مقایسه متناظر با آن. لذا حد پایین ارتفاع درخت مقایسه برابر با حد پایین برای زمان اجرای هر الگوریتم مرتب‌سازی مقایسه‌ای است.

قضیه ۱ *هر الگوریتم مرتب‌سازی مقایسه‌ای در بدترین حالت به $\Omega(n \lg n)$ مقایسه نیاز دارد.*
برهان. *با توجه به مطالب ذکر شده کافی است ارتفاع درخت مقایسه را تعیین کنیم. درخت مقایسه با ارتفاع h و تعداد برگ‌های l را متناظر با یک مرتب‌سازی مقایسه‌ای در نظر بگیرید. می‌دانیم که همه $n!$ جایگشت روی n ورودی به عنوان برگ این درخت هستند. بنابراین $n! \leq l$ و از طرفی یک درخت با ارتفاع h بیشتر از ۲^h تعداد برگ ندارد. پس:*

$$n! \leq l \leq ۲^h$$

با گرفتن لگاریتم از دوطرف داریم:

$$\begin{aligned} h &\geq \lg n! \\ &> \lg(n/e)^n \\ &= \Omega(n \lg n) \end{aligned}$$

(همچنین با استفاده از رابطه استرلینگ نیز واضح است $n! = \sqrt{۲\pi n}(n/e)^n\left(۱ + \Theta(\frac{۱}{n})\right)$.) ■

۴ مرتب‌سازی شمارشی

در این قسمت می‌خواهیم یکی از الگوریتم‌هایی که از روش مقایسه استفاده نمی‌کند معرفی کنیم:
در مرتب‌سازی شمارشی فرض می‌کنیم که هر یک از n عنصر ورودی یک عدد صحیح در بازه $[۰, k]$ می‌باشد که مقداری صحیح است. اگر $k = O(n)$ باشد آنگاه این الگوریتم در زمان $\Theta(n)$ اجرا می‌شود.
ایده اصلی این مرتب‌سازی به این صورت است که برای هر عنصر ورودی x تعداد عناصر کوچکتر از x را تعیین می‌کند، که در نهایت با این کار می‌توان مکان مناسب x را پیدا کرد. برای مثال اگر ۱۰ عنصر کوچکتر از x باشند آنگاه x درمکان یازدهم قرار می‌گیرد. تنها نکته‌ای که باقی می‌ماند این است که اگر چندین عنصر با یک مقدار در عناصر ورودی باشند باید این طرح را به نحو دقیقی اصلاح کنیم تا چند عنصر در یک مکان نباشند.
ابتدا به بیان الگوریتم ساده‌تری برای پیاده‌سازی مرتب‌سازی شمارشی بیان می‌کنیم و سپس به دلیل برقرار نشدن دو ویژگی که در ادامه گفته خواهد شد الگوریتم مناسب‌تری را ارایه می‌دهیم.

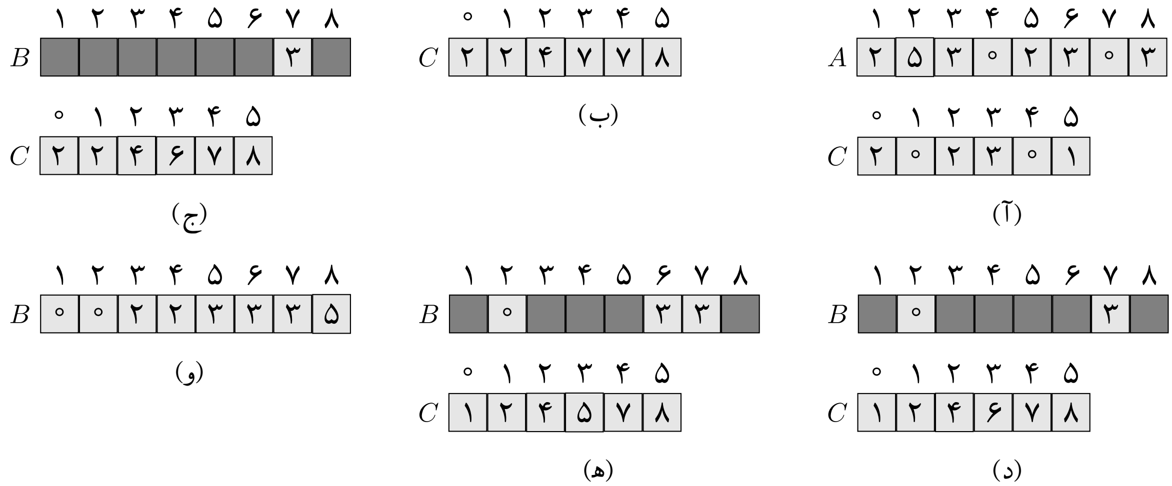
Algorithm 1 NaiveCountingSort

```
function NaiveCountingSort(A[1...n], k)
    //assumes 0 ≤ A[i] ≤ k
    let C[0···k] be a new array
    for  i = 0  to k do
        C[i] ← 0
    for  j = 1 downto n do
        C[A[j]] ← C[A[j]] + 1
    //C[i] now contains the number of elements equal to i.
    ctr ← 1
    for  i = 0  to k do
        for  j = 0  to C[i] do
            A[ctr] ← i
            ctr ← ctr + 1
```

این الگوریتم را نمی‌توانیم به صورت نوشته شده برای رکوردها مورد استفاده قرار دهیم. برای برقراری این ویژگی مهم الگوریتم را به صورت زیر تغییر می‌دهیم که ویژگی مطلوب پایداری را نیز داراست.
در شبه‌کد مرتب سازی‌شمارشی فرض می‌کنیم که آرایه ورودی $A[۱...n]$ باشد. به دو آرایه دیگر احتیاج داریم: آرایه $B[۱..n]$ و آرایه $C[۱...n]$ که به ترتیب یکی برای خروجی مرتب شده و دیگری به عنوان حافظه کاری موقتی استفاده می‌شوند.

Algorithm 2 COUNTINGSORT

```
function COUNTINGSORT(A[1...n], k)
    //assumes 0 ≤ A[i] ≤ k
    let C[0···k] be a new array
    for  i = 0  to k do
        C[i] ← 0
    for  j = 1 downto n do
        C[A[j]] ← C[A[j]] + 1
    //C[i] now contains the number of elements equal to i.
    for  i = 1  to k do
        C[i] ← C[i] + C[i − 1]
    //C[i] now contains the number of elements less than or equal to i.
    for  j = n downto 1 do
        B[C[A[j]]] ← A[j]
        C[A[j]] ← C[A[j]] − 1
```



بعد از مقداردهی اولیه در اولین حلقه for هر عنصر ورودی در دومین حلقه for بررسی می‌شود. اگر مقدار یک عنصر ورودی i باشد $C[i]$ یک واحد افزایش داده می‌شود؛ بنابراین، بعد از خط ۵، $C[i]$ تعداد عناصر ورودی برابر با i را نگه می‌دارد. در خطوط ۷ – ۶ با نگهداری یک مجموع در حال اجرای آرایه C، برای هر $i = ۰, ۱, ..., k$، تعدا عناصر ورودی کوچکتر یا مساوی با i را مشخص می‌کند.

در انتها، در آخرین حلقه for، هر عنصر $A[j]$ را در موقعیت مرتب شده صحیح خود، در آرایه خروجی B قرار می‌دهد.

اگر تمام n عنصر متفاوت باشند، آنگاه وقتی برای اولین بار به خط ۹ وارد می‌شود، برای هر عنصر $A[j]$ ، $C[A[j]]$ مکان صحیح $A[j]$ در آرایه خروجی است، زیرا به تعداد $C[A[j]]$ عنصر کوچکتر یا مساوی با $A[j]$ وجود دارد. بدلیل اینکه عناصر ممکن است متفاوت نباشند، هر بار که یک مقدار $A[j]$ را در آرایه B قرار می‌دهیم، $C[A[j]]$ را یکی کاهش می‌دهیم. در واقع با این کار باعث می‌شویم که اگر یک عنصر ورودی بعدی در صورت وجود دارای مقداری برابر با $A[j]$ باشد، به مکانی برود که بلافاصله قبل از $A[j]$ در آرایه خروجی قرار می‌گیرد.

۵ پیچدگی زمانی مرتب‌سازی شمارشی

اولین حلقه for به دلیل k بار اجرا شدن در زمان $\Theta(k)$ انجام می‌شود؛ دومین حلقه for در زمان $\Theta(n)$ صورت می‌گیرد؛ سومین حلقه for به دلیل k بار اجرا شدن در زمان $\Theta(k)$ صورت می‌گیرد و اخرین حلقه for در زمان $\Theta(n)$ انجام می‌شود. بنابراین زمان اجرای کل برابر $\Theta(n+k)$ است. در عمل از مرتب‌سازی شمارشی وقتی استفاده می‌کنیم که $k = O(n)$ است، که در این حالت پیچدگی زمانی برابر با $\Theta(n)$ می‌شود.
اگر تعداد داده‌ها خیلی کوچک نباشد، در عمل این الگوریتم برای مرتب‌سازی کلمه‌های ۸ـبیتی ($k = ۲۵۶$) مناسب است. بسته به میزان حافظه الگوریتم برای مرتب‌سازی کلمه‌های ۱۶ـبیتی ($k = ۲^{۱۶}$) ممکن است مناسب باشد. اما برای کلمه‌های ۳۲ـبیتی ($k = ۲^{۳۲}$) احتمالا مناسب نخواهد بود.

۶ مرتب‌سازی و پایداری

داده‌ها معمولاً به صورت عناصری هستند که دارای چندین مؤلفه می‌باشند. به عنوان مؤلفه‌های اطلاعات یک دانشجو می‌تواند شامل نام، نام خانوادگی، نمره، شماره دانش‌جویی و ... باشد. هنگامی که تعدادی عنصر را مرتب می‌کنیم، مرتب‌سازی را برحسب یکی از مؤلفه‌ها انجام می‌دهیم که اصطلاحاً به آن کلید گفته می‌شود. مثلا اگر بخواهیم لیست دانش‌جویان یک درس را بر اساس نمره‌ی آن‌ها مرتب نماییم، کلید، مؤلفه نمره است. در این‌صورت می‌توان چندین لیست مختلف از دانش‌جویان ارائه داد که در همه‌ی آن‌ها، لیست بر حسب نمره مرتب شده است ولی ترتیب دانش‌جویانی که نمره‌ی یکسان دارند متفاوت باشد. الگوریتم‌های **مرتب‌سازی پایدار**[۲]، به گونه‌ای لیست نهایی را مشخص می‌کنند که اگر دو داده‌ی مختلف، دارای کلید یکسانی باشند، ترتیب نسبی آن‌ها در ورودی، در خروجی هم حفظ شود. یعنی اگر عنصر x در ورودی قبل از عنصر y آمده‌است و x و y کلیدهای یکسانی دارند، در خروجی مرتب‌شده هم حتماً x قبل از y قرار بگیرد.

نکته ۱ *در دوحالت زیر پایداری الگوریتم مرتب‌سازی مهم نیست:*

- *همه عناصر فقط از یک مؤلفه، که همان کلید است، تشکیل شده باشند،*
- *کلیدهای همه عناصر متمایز باشند.*

الگوریتم‌های ناپایدار را می‌توان به گونه‌ای پیاده‌سازی کرد که پایدار شوند. یکی از راه‌های ممکن این است که هنگام مقایسه، اگر کلیدها برابر بودند، ترتیب اولیه‌ی داده‌ها در ورودی را برای آن‌ها در نظر بگیریم، هر چند نگه‌داری ترتیب اولیه ممکن است نیاز به زمان و حافظه‌ی اضافی نسبت به خود الگوریتم داشته باشد.
مطلوب این است که یک الگوریتم مرتب‌سازی خود پایدار باشد یا با تغیر اندکی در آن، بدون تحمیل زمان و یا حافظه‌ی اضافی، بتوان آن‌را پایدار کرد. مرتب‌سازی شمارشی، یک الگوریتم پایدار است. اهمیت پایداری مرتب‌سازی شمارشی استفاده آن در مرتب‌سازی مبنایی[۳] که در جلسه بعد معرفی می‌شود.

[۲] stable sort
[۳] radix sort