دانشگاه صنعتی شریف
دانشکدهٔ ریاضی

ساختمان‌داده‌ها
بهار ۱۴۰۰
استاد: حسین بومری
گردآورنده:علی سرائر

جلسهٔ یازدهم

مفهوم stable بودن

در جلسه قبل به merge sort و selection sort پرداخته شد. یک نکته‌ای که می‌توان در مورد سورت‌ها مطرح کرد این است که آن‌ها می‌توانند stable باشند یا نباشند. معنی stable بودن نیز این است که اگر در اعداد ورودی، دو عدد مساوی هم باشند، و یکی از دیگری زودتر آمده باشد، پس از سورت کردن هم آن یکی که زودتر آمده بود، از عدد دیگر عقب تر باشد. به عنوان مثال، اگر بخواهیم اعداد زیر را سورت کنیم

۲۱۳۱۵

پس از سورت خواهیم داشت:

۱۱۲۳۵

دقت کنید که ۱ قرمز، قبل و پس از سورت کردن همواره عقب تر از ۱ سیاه قرار دارد. بنابراین، این سورت stable است.
مثلا الگوریتم merge sort را می‌توانیم بگوییم که stable است، به طوری که در الگوریتمی که در جزوه جلسه پیش وجود دارد، ارزش‌های leftPointer و rightPointer را به این صورت مقایسه می‌کنیم که اگر اعداد پوینتر سمت چپ(ارزش مولفه leftPointer در ArrayList)، از اعداد پوینتر سمت راست کمتر و یا مساوی بود، آن را وارد ArrayList نهایی می‌کنیم. به این صورت، عددی که در سمت چپ قرار دارد، همواره سمت چپ اعداد مساوی خود که در راست آن هستند قرار می‌گیرد و سورت stable می‌شود. کد مربوط به این نکته نیز در زیر آورده شده

```
if(left.get(leftPointer).compareTo(right.get(rightPointer)) <= 0){
    result.add(left.get(leftPointer));
    leftPointer++;
}
```

برای کد کامل merge sort به جزوهٔ جلسه پیش رجوع کنید.
اما selection sort نمی‌تواند stable باشد. برای روشن شدن مطلب مثال زیر را در نظر بگیرید.
فرض کنید می‌خواهیم اعداد زیر را با selection sort مرتب کنیم.

۹۳۴۵۹۱

با اجرای الگوریتم، در مرحلهٔ اول، جای ۹ قرمز رنگ با ۱ عوض می‌شود و اعداد به صورت زیر در می‌آیند

۱۳۴۵۹۹

بنابراین واضح است که selection sort نمی‌تواند stable باشد.

بررسی سورت‌های دیگر

insertion sort

در این نوع سورت کردن، یک پوینتر در سمت چپ لیست قرار می‌دهیم. سپس هر دفعه، جای عدد سمت راست پوینتر را در سمت چپ پوینتر پیدا می‌کنیم، به این صورت که آن را جایی قرار می‌دهیم که تمام اعداد سمت چپش،از خودش کوچک‌تر و یا با آن مساوی باشند. به نحوه سورت کردن اعداد در مثال زیر توجه کنید.

مثال:

|۳۴۶۸۴۲۷۹۲
۳|۴۶۸۴۲۷۹۲
۳۴|۶۸۴۲۷۹۲
۳۴۶|۸۴۲۷۹۲
۳۴۶۸|۴۲۷۹۲
۳۴۴۶۸|۲۷۹۲
۲۳۴۴۶۸|۷۹۲
۲۳۴۴۶۷۸|۹۲
۲۳۴۴۶۷۸۹|۲
۲۲۳۴۴۶۷۸۹|

به چهار قرمز شده در مثال بالا دقت کنید. واضح است که الگوریتم طوری طراحی شده،که ترتیب اعداد مساوی با هم پس از سورت شدن تغییر نمی‌کند؛ در نتیجه این سورت stable است.
حال کد آن را در جاوا پیاده سازی می‌کنیم

```
static <T extends Comparable<T» void insertionSort(ArrayList<T> values){
    for(int i=0;i<values.size();i++){
        T temp = values.get(i);
        int j=i-1          for(;j>=0 && values.get(j).compareTo(temp)>0;j--){
           values.set(j+1, values.get(j));
        }
        values.set(j+1, temp);
    }
}
```

تحلیل مرتبه زمانی:
اگر اعداد از اول مرتب باشد از مرتبهٔ O(n) می‌شود(بهترین حالت)،چرا که در هیچ مرحله‌ای وارد for دومی نمی‌شویم.
حال اگر اعداد دقیقا با ترتیب برعکس قرار گرفته شده باشند، کل اعداد for دوم هر دفعه پیمایش می‌شود و مرتبه زمانی از $O(n^2)$ می‌شود(بدترین حالت).
این کد را سر کلاس اجرا کردیم و مدت زمان حدودی آن تقریبا ۲۰۰۰ میلی‌ثانیه است. مرج سورت با ۵۰ میلی ثانیه هنوز کمترین زمان را می‌گیرد.
این سورت stable است. اگر به کد دقت کنیم، می‌بینیم که اگر دو عدد با هم مساوی باشند، جای آن‌ها تغییر نمی‌کند.

bubble sort

مثال: فرض کنید می‌خواهیم اعداد زیر را با bubble sort مرتب کنیم.

۱۵۳۷۴۲۸۴

در گام اول،هر دو عدد پشت سر هم(که با قرمز نشان دادیم)، در هر iteration با هم مقایسه می‌شوند. اگر ترتیب آن‌ها درست نبود جای آن‌ها با هم عوض می‌شود.

۱۵۳۷۴۲۸۴
۱۵۳۷۴۲۸۴
۱۳۵۷۴۲۸۴
۱۳۵۷۴۲۸۴
۱۳۵۴۷۲۸۴
۱۳۵۴۲۷۸۴
۱۳۵۴۲۷۸۴
۱۳۵۴۲۷۴۸

یعنی ماکسیمم که ۸ بود جای خود را پیدا کرد. در مرحلهٔ بعدی بزرگترین عدد بعدی، که عدد ۷ است جای خود را پیدا می‌کند(در جایگاه دومین بزرگترین عدد قرار می‌گیرد).به این صورت هر دفعه بزرگترین عدد بین اعداد، به جز اعدادی که تا مرحلهٔ قبلی جای خود را پیدا کردند، به سمت راست حرکت می‌کند(مانند حباب که به سمت بالا حرکت می‌کند).
پیاده سازی کد آن در جاوا نیز به صورت زیر است

```
static <T extends Comparable<T>» void bubbleSort(ArrayList<T> values){
    for(int i=0;i<values.size()-1;i--){
        for(int j=0;j<i ;j++){
            if(values.get(j).compareTo(values.get(j+1))<0){
                T temp = values.get(i);
                    values.set(j+1, values.get(j));                values.set(j, temp);          }          val-
ues.set(j+1, values.get(j));
        }
        values.set(j+1, temp);
    }
}
```

مرتبهٔ زمانی bubble sort از $O(n^۲)$ است؛ چون در هر حالت باید تمام اعداد با هم دو به دو مقایسه شوند و هر دو تا for همیشه اجرا می‌شوند. در نتیجه در بهترین حالت و بدترین حالت همواره مرتبهٔ زمانی $O(n^۲)$ است.
یک نکتهٔ ظریف: در کدهای بالا می‌توانستیم متغیر temp را بیرون for تعریف کنیم و درون for فقط مقدار آن را تغییر دهیم، تا حافظهٔ کمتری مصرف کنیم.
پس از اجرا کردن کد، زمانی که برای bubble sort به دست آمد حدود ۲۲۰۰ میلی ثانیه بود.
این سورت نیز stable است. چرا که در این الگوریتم، فقط زمانی جای دو عدد کنار هم عوض می‌شود، که سمت راستی از چپی کوچک‌تر باشد. اگر مساوی باشند جای آن‌ها عوض نمی‌شود.

radix sort

با یک مثال این سورت را توضیح می‌دهیم. فرض کنید می‌خواهیم اعداد زیر را مرتب کنیم.

۲۱۳, ۲۳۳۲۴, ۴۳۵۴, ۲۲۱۳, ۳۲۴, ۵۴۶۵, ۱۲۶, ۴۳۲۵۴۲

برای این سورت مراحل زیر را طی می‌کنیم
ابتدا برحسب یکان اعداد را مرتب می‌کنیم.

۱st : ۲۱۳, ۲۳۳۲۴, ۴۳۵۴, ۲۲۱۳, ۳۲۴, ۵۴۶۵, ۱۲۶, ۴۳۲۵۴۲
o :
۱ :
۲ : ۴۳۲۵۴۲
۳ : ۲۱۳, ۲۲۱۳
۴ : ۲۳۳۲۴, ۴۳۵۴, ۳۲۴
۵ : ۵۴۶۵
۶ : ۱۲۶
۷ :
۸ :
۹ :

→ ۴۳۲۵۴۲ ۲۱۳ ۲۲۱۳ ۲۳۳۲۴ ۴۳۵۴ ۳۲۴ ۵۴۶۵ ۱۲۶

در مرحلهٔ بعد رقم دوم(دهگان)ها را با هم مقایسه می‌کنیم.

۲۱۳, ۲۳۳۲۴, ۴۳۵۴, ۲۲۱۳, ۳۲۴, ۵۴۶۵, ۱۲۶, ۴۳۲۵۴۲
۱st : ۴۳۲۵۴۲ ۲۱۳ ۲۲۱۳ ۲۳۳۲۴ ۴۳۵۴ ۳۲۴ ۵۴۶۵ ۱۲۶
o :
۱ : ۲۱۳ ۲۲۱۳
۲ : ۲۳۳۲۴ ۳۲۴ ۱۲۶
۳ :
۴ : ۴۳۵۴۲
۵ : ۴۳۵۴
۶ : ۵۴۶۵
۷ :
۸ :
۹ :

→ ۲۱۳ ۲۲۱۳ ۲۳۳۲۴ ۳۲۴ ۱۲۶ ۴۳۵۴۲ ۴۳۵۴ ۵۴۶۵

توجه: دقت کنید که این سورت اعداد را واقعا مرتب می‌کند، پس از آن که اعداد برحسب یکان از کوچک به بزرگ مرتب شدند، در مرحلهٔ بعد که برحسب دهگان مرتب می‌کنیم، آن عددی اول در نتیجه قرار می‌گیرد که یکان کوچک‌تری داشت،چون در مرحلهٔ دوم،از لیست سورت شده برحسب یکان مرحلهٔ اول استفاده می‌کنیم که یکان کمتر زودتر از یکان بیشتر آمده است.

حال به همین شکل مراحل را برای رقم‌های بزرگ‌تر تکرار می‌کنیم.

۲۱۳, ۲۳۳۲۴, ۴۳۵۴, ۲۲۱۳, ۳۲۴, ۵۴۶۵, ۱۲۶, ۴۳۲۵۴۲
۱^{st} : ۴۳۲۵۴۲ ۲۱۳ ۲۲۱۳ ۲۳۳۲۴ ۴۳۵۴ ۳۲۴ ۵۴۶۵ ۱۲۶
۲^{nd} : ۲۱۳ ۲۲۱۳ ۲۳۳۲۴ ۳۲۴ ۱۲۶ ۴۳۵۴۲ ۴۳۵۴ ۵۴۶۵
o :
۱ : ۱۲۶
۲ : ۲۱۳ ۲۲۱۳
۳ : ۲۳۳۲۴ ۳۲۴ ۴۳۵۴
۴ : ۵۴۶۵
۵ : ۴۳۲۵۴۲
۶ :
۷ :
۸ :
۹ :

$\rightarrow$ ۱۲۶ ۲۱۳ ۲۲۱۳ ۲۳۳۲۴ ۳۲۴ ۴۳۵۴ ۵۴۶۵ ۴۳۲۵۴۲

۲۱۳, ۲۳۳۲۴, ۴۳۵۴, ۲۲۱۳, ۳۲۴, ۵۴۶۵, ۱۲۶, ۴۳۲۵۴۲
۱^{st} : ۴۳۲۵۴۲ ۲۱۳ ۲۲۱۳ ۲۳۳۲۴ ۴۳۵۴ ۳۲۴ ۵۴۶۵ ۱۲۶
۲^{nd} : ۲۱۳ ۲۲۱۳ ۲۳۳۲۴ ۳۲۴ ۱۲۶ ۴۳۵۴۲ ۴۳۵۴ ۵۴۶۵
۳^{rd} : ۱۲۶ ۲۱۳ ۲۲۱۳ ۲۳۳۲۴ ۳۲۴ ۴۳۵۴ ۵۴۶۵ ۴۳۲۵۴۲
o : ۱۲۶ ۲۱۳ ۳۲۴
۱ :
۲ : ۲۲۱۳ ۴۳۲۵۴۲
۳ : ۲۳۳۲۴
۴ : ۴۳۵۴
۵ : ۵۴۶۵
۶ :
۷ :
۸ :
۹ :

$\rightarrow$ ۱۲۶ ۲۱۳ ۳۲۴ ۲۲۱۳ ۴۳۲۵۴۲ ۲۳۳۲۴ ۴۳۵۴ ۵۴۶۵

۲۱۳, ۲۳۳۲۴, ۴۳۵۴, ۲۲۱۳, ۳۲۴, ۵۴۶۵, ۱۲۶, ۴۳۲۵۴۲

1^{st} : ۴۳۲۵۴۲ ۲۱۳ ۲۲۱۳ ۲۳۳۲۴ ۴۳۵۴ ۳۲۴ ۵۴۶۵ ۱۲۶

2^{nd} : ۲۱۳ ۲۲۱۳ ۲۳۳۲۴ ۳۲۴ ۱۲۶ ۴۳۵۴۲ ۴۳۵۴ ۵۴۶۵

3^{rd} : ۱۲۶ ۲۱۳ ۲۲۱۳ ۲۳۳۲۴ ۳۲۴ ۴۳۵۴ ۵۴۶۵ ۴۳۲۵۴۲

4^{th} : ۱۲۶ ۲۱۳ ۳۲۴ ۲۲۱۳ ۴۳۲۵۴۲ ۲۳۳۲۴ ۴۳۵۴ ۵۴۶۵

o : ۱۲۶ ۲۱۳ ۳۲۴ ۲۲۱۳ ۴۳۵۴ ۵۴۶۵

۱ :

۲ : ۲۳۳۲۴

۳ : ۴۳۲۵۴۲

۴ :

۵ :

۶ :

۷ :

۸ :

۹ :

$\rightarrow$ ۱۲۶ ۲۱۳ ۳۲۴ ۲۲۱۳ ۴۳۵۴ ۵۴۶۵ ۲۳۳۲۴ ۴۳۲۵۴۲

و واضح است که در مرحلهٔ بعدی همهٔ اعداد سورت می‌شوند و کار تمام شد.
این سورت stable است، چون در هیچ‌جایی اگر دو عدد مساوی هم باشند، جای آن‌ها را عوض نمی‌کنیم.
همیشه ده‌تا صف داشتیم که اعداد را بر اساس رقم‌هایشان در هر کدام از صف‌ها قرار می‌دادیم. یعنی ۱۰ تا صف n تایی داریم. بنابراین مرتبهٔ حافظه،از $O(n)$ است.
مرتبهٔ زمانی radix sort از $O(n * digits(Max))$ است، که $digits(Max)$ تعداد ماکسیمم رقمی است که بین اعداد وجود دارد(تا آخرین رقم باید از ۱۰ تا صف بالا هر دفعه استفاده کنیم).
سؤالی که در اینجا پیش می‌آید این است که چرا $digits(Max)$ را در مرتبهٔ زمانی در نظر گرفتیم. به این دلیل است که ممکن است تعداد رقم‌های ما حدی نداشته باشند، و اگر تعداد رقم‌ها خیلی بیشتر از تعداد اعداد باشد،مرتبهٔ زمانی را به مقدار قابل توجهی تحت تأثیر قرار می‌دهد. واضح است که می‌توانیم مرتبهٔ زمانی را به شکل زیر(که بهتر است) نیز بنویسیم
$O(n * log(Max))$ که Max بزرگ‌ترین عدد بود.