ساختمان داده‌ها (۲۲۸۲۲)

مدرس: حسین بومری

[بهار ۱۴۰۰]

جلسه : دوازدهم

نگارنده: غزل فراهانی

در جلسه گذشته درمورد مفهوم stable بودن الگوریتم‌های مرتب‌سازی صحبت کردیم. همچنین insertion sort و bubble sort را پیاده‌سازی کردیم و در مورد radix sort صحبت کردیم که در این جلسه آن را پیاده‌سازی می‌کنیم.

۱ Radix sort

در الگوریتم radix sort اعداد را بر اساس کم‌ارزش‌ترین رقم تا پرارزش‌ترین رقم مرتب می‌کنیم.(در ابتدا تمام اعداد در numbers[0] هستند.)

```
RadixSort()
// numbers and buffers are DoublyLinkedList
// pow supposed to keep the value of the current power of 10
pow = 1
while pow < max * 10
    for digit = 0 to 10
        while !numbers[digit].isEmpty()
            value = numbers[digit].popFront()
            nextDigit = (value/pow)%10
            buffers[nextDigit].pushBack(value)
    temp = numbers
    numbers = buffers
    buffers = temp
    pow *= 10
for i = 0 to values.size()
    values.set(i, numbers[0].popFront())
```

در الگوریتم بالا شرط pow < max*10 یعنی همه ارزش مکانی‌ها را چک کردیم و در نهایت یک ارزش مکانی جلوتر رفته‌ایم درنتیجه تمام اعداد، مرتب شده، در numbers[0] وجود دارند.

مقایسه با سایر الگوریتم‌های مرتب‌سازی

این الگوریتم را اجرا کردیم و با زمان ۵ میلی ثانیه سریع‌ترین الگوریتم در مقایسه با سایر الگوریتم‌هایی است که پیاده‌سازی کردیم. الگوریتم radix sort نسبت به اعداد بزرگ دارای حساسیت است. برای مثال ممکن است در قسمت $\text{pow} < \text{max}*10$ عددی داشته باشیم که از integer بزرگ‌تر باشد. برای حل این مشکل می‌توانیم بازه اعداد را محدود کنیم. همچنین اگر اعداد بسیار بزرگ باشند با توجه به مرتبه زمانی الگوریتم ($O(n * log(Max))$) نمیتوان عبارت $\log(\text{Max})$ را $constant$ فرض کرد. درنتیجه این الگوریتم برای اعداد بزرگ عملکرد خوبی ندارد. همچنین radix sort برای نوع داده Integer تعریف شده و برای نوع داده‌های دیگر معنی ندارد.در این الگوریتم از عملگرهای تقسیم بسیار استفاده می‌شود که نسبتا عملگرهای سنگینی هستند. البته می‌توان با محاسبه آن‌ها در مبنای ۲ این سربار عملگر تقسیم را برداشته و آن را سریع‌تر هم کرد.

۲ Count sort

الگوریتم count sort معمولا برای نوع داده Integer استفاده می‌شود. این الگوریتم از آرایه دیگری با اندازه $\text{max} + 1$ استفاده می‌کند به این ترتیب که به ازای هر کلید r در آرایه ورودی، اعدادی را که مقدار آن‌ها برابر r باشد را می‌شمارد و در خانه مربوطه در آرایه کمکی قرار می‌دهد.

$\text{CountSort}(ArrayList < Integer > \quad values)$

$counts = new\ int[max]$

for $i = 0$ **to** $values.size()$

    $counts[values.get(i)] + +$

for $i = 0, count = 0$ **to** $count.length$

    for $j = 0$ **to** $counts[i]$

        values.set(count++, i)

مقایسه با سایر الگوریتم‌های مرتب‌سازی

الگوریتم count sort از $\text{O}(\text{n} + \text{Max})$ است. در نتیجه نسبت به اندازه ورودی مرتبه زمانی نمایی دارد.(برای مثال اگر بزرگ‌ترین عدد برابر با $۱۰^{۱۰۰}$ باشد، سایز ورودی برابر با $log ۱۰^{۱۰۰} = ۱۰۰$ خواهد بود.) درنتیجه الگوریتم count sort هنگامی که اندازه عددها نسبت به تعداد بسیار زیاد می‌شود عملکرد خوبی خواهد داشت. (اگر محدوده اعداد ورودی را تا ۱۰۰۰۰۰۰۰۰ بالا ببریم، سایر الگوریتم‌هایی که پیاده‌سازی کردیم دارای زمان زیر ۱۰ میلی ثانیه اما الگوریتم count sort دارای زمان ۲۵۰ میلی ثانیه است.) به عبارتی هنگامی که اندازه عددها بسیار بالا می‌رود، الگوریتم از مرتبه $\text{O}(\text{Max})$ و هنگامی که تنوع عددها کم و تعداد زیاد است از مرتبه $\text{O}(\text{n})$ است.

مفهوم stable بودن برای الگوریتم count sort معنایی ندارد زیرا برای عضوهای مشابه تفاوتی قائل نمی‌شود. البته می‌توان برای اینکه این الگوریتم stable باشد، به جای شمارش اعضا، آن‌ها را در list نگهداری کرد.

از الگوریتم‌هایی که تا به اینجا بررسی کردیم، الگوریتم merge sort اردر حافظه خوبی ندارد. الگوریتم‌های selection sort, insertion sor, bubble sort مرتبه زمانی خوبی ندارند و همچنین الگوریتم‌های radix sort, count sort وابسته به ورودی مسئله هستند. اگر بخواهیم الگوریتمی برای مرتب‌سازی پیشنهاد دهیم که مزایای الگوریتم‌های گفته شده را داشته باشد، heap sort است که در جلسه‌های بعدی بعد از آشنایی با داده ساختار heap گفته می‌شود. (در این داده ساختار اعمال درج و حذف در $O(logn)$ انجام می‌شوند در نتیجه این الگوریتم مرتب سازی از مرتبه زمانی $O(nlogn)$ و همچنین حافظه $O(n)$ است.)

۳ Quick sort

در این الگوریتم ابتدا آرایه داده شده را به دو زیر آرایه ناتهی تقسیم می‌کنیم. به این ترتیب که بر اساس یه واحد انتخاب‌شده، آرایه را به دو قسمت تقسیم می‌کنیم که اعداد بعد از واحد همگی بزرگ‌تر و اعداد قبل از واحد کوچکتر باشند. که این کار توسط دو اشاره‌گر یکی در ابتدا و یکی در انتهای آرایه قابل انجام است.

```
1   3 7 1 8 6 3 2 4 6 4 4 2 8
2   3
3   * 7 1 8 6 3 2 4 6 4 4 2 8|
4   * 7 1 8 6 3 2 4 6 4 4 2| 8
5   2 * 1 8 6 3 2 4 6 4 4 7| 8
6   2 * 1 8 6 3 2 4 6 4 4| 7 8
7   2 * 1 8 6 3 2 4 6 4| 4 7 8
8   2 * 1 8 6 3 2 4 6| 4 4 7 8
9   2 * 1 8 6 3 2 4| 6 4 4 7 8
10  2 * 1 8 6 3 2| 4 6 4 4 7 8
11  2 2 * 8 6 3 1| 4 6 4 4 7 8
12  2 2 1 * 6 3 8| 4 6 4 4 7 8
13  2 2 1 * 6 3| 8 4 6 4 4 7 8
14  2 2 1 * 6| 3 8 4 6 4 4 7 8
15  2 2 1 *| 6 3 8 4 6 4 4 7 8
16  2 2 1 |3| 6 3 8 4 6 4 4 7 8
```

الگوریتم بالا stable نیست. همچنین مرتبه زمانی آن به صورت زیر به دست می‌آید:

$$T(n) \;=\; T(k) \;+\; T(n-k-۱) \;+\; O(n)$$

بهترین حالت:

$$T(n) \;=\; T(\frac{n}{۲}) \;+\; T(\frac{n}{۲}-۱) \;+\; O(n) \;\Rightarrow\; O(nlogn)$$

بدترین حالت:

$$T(n) \;=\; T(۰) \;+\; T(n-۱) \;+\; O(n) \;\Rightarrow\; O(n^{۲})$$

تحلیل حالت میانگین به جلسه بعدی موکول شد.