Homework 5

Daoyu Wang PB21030794

October 10

Contents

1 7.1-2 **1**

2 7.4-5 **1**

3 8.2-4 **2**

4 8.4-2 **3**

5 9.1-1 **3**

6 9.3-6 **5**

1 7.1-2

1. If the elements in $A[p..r]$ equal:

   The return value **q** is **r**, since **if**$A[j] \leq x$ at function 'PARTITION' in Para 170.

2. modify function PARTITION To make PARTITION return $q = \lfloor (p + r)/2 \rfloor$.

   We also let the last element x = A[r] as **pivor element**. However, it will be exchanged as other normal ones in the circulation. Firstly, start from the left side of the sequence and search to the right to find the first value that is smaller than the pivot element. Exchange it with the pivot element. Then start from the left and search to the right, find the first value that is larger than the pivot element, and exchange it with the pivot element. This process stops until the two search directions meet.

   The pseudocode is as follows:

```
PARTITION(A, p, r)
    while i < j then
        if i < j && A[i] <= key
            exchange A[i] with A[r]
        i = i + 1
        if i < j && A[j] >= key
            exchange A[j] with A[r]
        j = j - 1
    end
    return i
```

2 7.4-5

According to what were said in class about **quick sort performance**, its time complexity can be interpreted by the following derivation:

Firstly, assume that c_n represents the numbers of comparisons in first stage: stop when the array length is under k

$$\begin{aligned} c_n &= n - 1 + \frac{1}{n}\sum_{k=1}^{n}(c_{k-1} + c_{n-k}) \\ &= n - 1 + \frac{2}{n}\sum_{k=1}^{n} c_k \end{aligned}$$

However, here is a detail different from what was taught in class: the end point of the recursion is not $c_1 = 0$ but $c_k = 0$. After the same derivation as in class, we can achieve as follows:

$$\frac{c_n}{n+1} - \frac{c_{n-1}}{n} = \frac{4}{n+1} - \frac{2}{n} and \frac{c_k}{k+1} = 0$$

$$\frac{c_n}{n+1} = 2(H(n) - H(k)) + \frac{4}{n+1} - \frac{2}{k+1}$$

$H(n)$ is **harmonic series**, and $H(n) = O(log(n))$, then:

$$c_n = O(n(log(n) - log(k))) = O(nlog(\frac{n}{k}))$$

After this process, the array is devided into $\frac{n}{k}$ areas. We can call **insertion sort function** now. Thanks to the quick sort, we can find that the array is increasing by area. Originally, when using insertion sort, each element was compared with all the elements before it, but now each element only needs to be compared with the elements in the same area before it. Therefore, assume d_n is the time complexity now d_n is as follows:

$$d_n = \sum_{j=1}^{k-1} j \cdot \frac{n}{k} = \frac{(k-1)n}{2} = O(kn)$$

Finally, the total time complexity is $c_n + d_n = O(nk + nlog(\frac{n}{k}))$.

From a theoretical perspective, we expect a **k** to minimize $O(nk+nlog(\frac{n}{k}))$. This means that we need to derive the expression with respect to **k** into $An - \frac{Bn}{k}$(A, B is constants). So $k = \Theta(1)$ can make the time complexity lowest.

From a practical perspective, however, we would use a larger **k** because the coefficient of the expression $\Theta(1)$ will not as larger as 1 . As a result, the

expected **k** is not necessarily an integer. Therefore, in practical selection, k is often larger than the theoretical value.

3 8.2-4

The algorithm processs similar to what **counting sort** does in line 1 to 9.After this algorithm, array $C[i]$ contains the number of elements less than or equal to i. Back to the quary: *how many integers fall within the interval [a: b]*, we can easily find that the answer is $C[b] - C[a-1]$.

The time complexity of preprocessing is same as the **counting sort**: $O(n+k)$, and the time complexity of finding how many integers is $O(1)$

4 8.4-2

In the worst case, we would have a bucket which contains all n values of the input sequence. The **insertion sort** will take $O(n^2)$ running time. To avoid this, the **insertion sort** can be replaced by **merge sort** which has the worst running time $O(nlog(n))$.

5 9.1-1

1. Separate the array to two parts having equal size. if **n** is odd, let the middle item alone.

2. Select element pair that are symmetrical along the middle line and compare with each other. Store the smaller one into another array of length **n**. Meanwhile, store the index of the larger one to prepare for the following traceback.

3. Repeat this process until the length of extra array is 2 or 3. return the **min** and **sub-min**.

4. Since we would pair the **min** and **sub-min** at some time in recursion, and then neglect the item **sub-min**. So after the recursion, we need to traceback: compare the **min** in the returned pair with the larger neglected value stored in advance in the second step.

The recursion formula is as follows:

$$T(n) = T(\lceil \frac{n}{2} \rceil) + \lfloor \frac{n}{2} \rfloor + 1$$

Now, we can prove that $T(n) = O(n + \lceil log(n) \rceil - 2)$. Firstly, we acknowledge that $T(2) = 1 = 2 + \lceil log(2) \rceil - 2$. Assume that when $m < n$, $T(m) \leq m + \lceil log(m) \rceil - 2$.

$$\begin{aligned} T(n) &= T(\lceil \frac{n}{2} \rceil) + \lfloor \frac{n}{2} \rfloor + 1 \\ &\leq \lceil \frac{n}{2} \rceil + \lceil log(\lceil \frac{n}{2} \rceil) \rceil - 2 + \lfloor \frac{n}{2} \rfloor + 1 \\ &= n + \lceil log(\lceil \frac{n}{2} \rceil) + 1 \rceil - 2 \\ &= n + \lceil log(2\lceil \frac{n}{2} \rceil) \rceil - 2 \end{aligned}$$

we can prove that: $\lceil log(2\lceil \frac{n}{2} \rceil) \rceil = \lceil log(n) \rceil$
if n is even, the equation is obvious. When n is odd, let $n = 2k + 1$.

$$\begin{aligned} LEFT &= \lceil log(2k + 2) \rceil \\ RIGNT &= \lceil log(2k + 1) \rceil \end{aligned}$$

It looks like $LEFT \geq RIGNT$, but only when $2k + 1 = 2^m$ and $2k + 2 = 2^m + 1$, $LEFT > RIGNT$ because $\lceil log(t) \rceil = m$ only when $2^{m-1} + 1 \leq t \leq 2^m$. However, $2k + 1$ is odd, comflict with 2^m which is even.

So $T(n) \leq n + \lceil log(n) \rceil - 2$. Now, we can assert that: in the worst case, the running time of finding the sub-smallest element is $n + \lceil log(n) \rceil - 2$.

The C++ code is as follows:

```
1   /**
2    * @brief Recursion to find the min_index and sub_index of
       the vector
3    *
4    * @param arr input vector
5    * @param len length of the input vector
6    * @return pair<size_t, size_t> min_index, sub_index
7    * @brief the arr "index" has the same length of "next_arr",
       which store a
8    * mirror: index[i] is the index of the smaller one in pair
       in "arr" i is the
9    * index where the smaller one stay in "next_array" So use "
       index[ret.first]",
10   * we can find the index of "min" in upper recursion
       Meanwhile,
11   * arr[index[ret.first]] is obviously "min"
12   */
13  pair<size_t, size_t> Submin::SelectSubminIndex(vector<int>
       arr, unsigned len) {
14      if (len == 2) {
15          if (arr[0] > arr[1]) {
```

```
16                  return make_pair(static_cast<size_t>(1),
                        static_cast<size_t>(0));
17              } else {
18                  return make_pair(static_cast<size_t>(0),
                        static_cast<size_t>(1));
19              }
20          }
21          vector<int>      next_arr;
22          vector<size_t> index;
23          for (size_t i = 0; i < len / 2; i++) {
24              if (arr[i] < arr[len - 1 - i]) {
25                  next_arr.push_back(arr[i]);
26                  index.push_back(i);
27              } else {
28                  next_arr.push_back(arr[len - 1 - i]);
29                  index.push_back(len - 1 - i);
30              }
31          }
32          if (len % 2 == 1) {
33              next_arr.push_back(arr[len / 2]);
34              index.push_back(len / 2);
35          }
36          auto    ret     = SelectSubminIndex(next_arr, next_arr.size
                ());
37          size_t min_index = index[ret.first];
38          size_t sub_index = index[ret.second];
39          // int min = arr[index[ret.first]];
40          int sub       = arr[index[ret.second]];
41          int compare_min = arr[len - 1 - min_index];
42          if (sub > compare_min && min_index != len - 1 - min_index
                ) {
43              sub_index = len - 1 - min_index;
44          }
45          return make_pair(min_index, sub_index);
46      }
```

6 9.3-6

Assume that $T(n,k)$ represents the running time of finding k^{th} quantiles in array of length n. Firstly, we can separate $k-1$ quantiles into three parts: $\lfloor \frac{k}{2} \rfloor - 1$, 1, $\lceil \frac{k}{2} \rceil - 1$. Obviously, $k-1 = \lfloor \frac{k}{2} \rfloor - 1 + 1 + \lceil \frac{k}{2} \rceil - 1$ and $\lfloor \frac{k}{2} \rfloor - 1 \leq \lceil \frac{k}{2} \rceil - 1$

At the same time, we can separate the array into three parts: $\lceil \frac{n}{2} \rceil - 1$, 1, $\lfloor \frac{n}{2} \rfloor$. Obviously, $n = (\lceil \frac{n}{2} \rceil - 1) + 1 + \lfloor \frac{n}{2} \rfloor$ and $\lceil \frac{n}{2} \rceil - 1 \leq \lfloor \frac{n}{2} \rfloor$

The purpose of this simplified model is to make the **1** of the two separations appoximately **correspond**. We assume that the $\lceil \frac{n}{2} \rceil^{th}$ position of the array is exactly the $\lfloor \frac{n}{2} \rfloor^{th}$ quantile point. However, a more rigorous proof may need to discuss the parity of n and k. It can prove that this correction will not affect the time complexity, because the rigorous discussion itself will only have a constant level difference from the simplified model here.

Then, we can achieve the recursion formula:

$$T(n,k) = T(\lceil \frac{n}{2} \rceil - 1, \lfloor \frac{k}{2} \rfloor - 1) + T(\lfloor \frac{n}{2} \rfloor, \lceil \frac{k}{2} \rceil - 1) + O(n)$$

Finally, we prove that $T(n) = O(nlog(k))$: Firstly, we assume that the extra $O(n)$ is under dn, d is a given positive number. And when k = 2, According to **worst time O(n) selection algorithm**, $T(n,2) = O(n)$. This initial case is correspond to inductive hypothesis: $O(nlog(2))$

Then, assume that $\forall m < n, l < k$, $T(m,l) \leq cmlog(l)$.

The induction process is as follows:

$$\begin{aligned}
T(n,k) &= T(\lceil \frac{n}{2} \rceil - 1, \lfloor \frac{k}{2} \rfloor - 1) + T(\lfloor \frac{n}{2} \rfloor, \lceil \frac{k}{2} \rceil - 1) + dn \\
&= c(\lceil \frac{n}{2} \rceil - 1)log(\lfloor \frac{k}{2} \rfloor - 1) + c\lfloor \frac{n}{2} \rfloor log(\lceil \frac{k}{2} \rceil - 1) + dn \\
&\leq c(\lceil \frac{n}{2} \rceil)log(\frac{k}{2}) + c\lfloor \frac{n}{2} \rfloor log(\frac{k}{2}) + dn \\
&\leq cnlog(\frac{n}{2}) + dn \\
&= cnlog(n) - (c-d)n
\end{aligned}$$

As you can see, if $c \geq d$, $T(n,k) \leq cnlog(n)$. Therefore, the time complexity of this algorithm is $O(nlog(k))$.