

# CSE373 Homework 2 Solutions & Hints

In this solution, we assume that the input is  $S[1..n]$  if there's no specification about the form of the input with a size  $n$ .

We also assume that  $\text{Sort}(S, i, j, 'i')$  will sort  $S[i], S[i+1], \dots, S[j]$  in an increasing order in the worst-case  $\Theta(k \log k)$  time ( $k = j - i + 1$ ); while  $\text{Sort}(S, i, j, 'd')$  will sort in a decreasing order in the worst-case  $\Theta(k \log k)$  time.

In this solution,  $\log n = \log_2 n$  if there's no specification.

Mostly, the proofs are omitted.

## 1 Exercises 4-1 and 4-2 (10 pts + 10 pts = 20)

4-1:

Sort  $S$  ( $2n$  elements in total), then return the first  $n$  elements and the last  $n$  elements.

4-2:

(a): Return the maximal and minimal elements in the array. Finding the maximum and minimum can be done in  $\left\lceil \frac{3n}{2} \right\rceil - 2$  comparisons.

(b): Return the maximal and minimal elements in the array. ( $S[1]$  and  $S[n]$  in this case)

(c): First sort the array, then return  $\min_{1 \leq i \leq n-1} |S[i+1] - S[i]|$

(d): Return  $\min_{1 \leq i \leq n-1} |S[i+1] - S[i]|$  directly.

## 2 Exercises 4-5 and 4-6 (10 pts + 10 pts = 20)

4-5:

Here if we use a hashtable, we can easily handle this in  $O(n)$  time. If we do not use a hashtable, first we sort the arraylist, and then do a linear sweep to check which numbers appear most. Details omitted.

4-6:

```
Sort(S1, 1, n, 'i');
```

```
Sort(S2, 1, n, 'i');
```

```
for (int i = 1, j = n; i <= n && j >= 1;){
```

```
    if (S1[i] + S2[j] == x) return true;
```

```
    else if (S1[i] + S2[j] > x) j--;
```

```
    else i++;
```

```
}
```

```
return false;
```

### 3 Exercises 4-12 to 4-16 (4 pts \* 5 = 20)

4-12:

We build a min-heap upon the  $n$  inputs ( $O(n)$  running time), and then call `EXTRACT_MIN()`  $k$  times. ( $k * O(\log n)$ ).

Notes: If we first build a **max**-heap upon the first  $k$  elements, and then insert the rest  $n-k$  elements one by one, (i.e., if the element is larger than or equals to the top of the heap, we ignore it; otherwise we substitute the element for the original top, and then adjust the heap) we will have an  $O(n \log k)$  algorithm, with a relatively smaller space usage.

Moreover, we could use a worst-case  $O(n)$  algorithm to find the  $k$ -th smallest element in the array, and then use the `Partition()` algorithm in QuickSort. For more information, please visit [http://en.wikipedia.org/wiki/Selection\\_algorithm](http://en.wikipedia.org/wiki/Selection_algorithm) or refer to *Introduction to Algorithms*, section 9.3, *Selection in worst-case linear time*, 2<sup>nd</sup> edition.

4-13:

(a): it doesn't matter

(b): max-heap (here we suppose that we know the pointer to the element we want to delete, and that the sorted array is not saved as a linked list)

(c): max-heap

(d): sorted array

4-14:

Like MergeSort, we keep a record of  $k$  pointers. Initially we set the pointers to 1. Firstly we build a heap upon  $S_i[1]$  ( $1 \leq i \leq k$ ). Then we extract the minimum of the heap, increase by 1 the pointer of the array from which the minimum we extracted comes, and insert the next element of the array into the heap if there is any. The running time is  $O(n \log k)$  since the size of the heap is  $k$ .

4-15:

(a):  $n + \lceil \log n \rceil - 2$  comparisons

We build a balanced tree to keep a record of the process of the 'tournament' whose purpose is to find the largest number in the array. It's clear that the height of tree is  $\lceil \log n \rceil$ , and we need  $n - 1$  comparisons to find the largest number since each comparison can 'knock out' one element. Then we find the competition path of the winner. Only those elements that are directly 'beaten' by the winner could be the second largest number. So we substitute the leaf which represents the largest number with  $-\infty$ , and update the tree. Then we can find the largest number among the remaining  $n - 1$  elements in  $\lceil \log n \rceil - 1$  comparisons.

---

Example:

First round(7 comparisons):

```
      8
     / \
    8   5
   / \ / \
  3  8 5  6
 / \ / \ / \
3  1 8 7 2 5 4 6
```

Find the path; substitute (only 7, 3 and 5 could be the second largest one):

(Here 0 means  $-\infty$ ; ? means the numbers we want to update)

```

      ?
     ?      5
    3      7      5      6
3  1  0  7  2  5  4  6

```

Update (2 comparisons):

```

      7
     7      5
    3      7      5      6
3  1  0  7  2  5  4  6

```

Thus 7 is the second largest number.

---

(b):  $n + 2\lceil \log(n - 1) \rceil - 3$  comparisons

Use a similar strategy. First, we pick out the first  $n - 1$  elements, and use  $n - 2$  comparisons to find the largest number among the first  $n - 1$  elements. Now the height of the tree is  $\lceil \log(n - 1) \rceil$ . We know that the largest number we picked out could not be the 3<sup>rd</sup> largest. Then we substitute the  $n$ -th elements for the leaf which represents the winner, update the tree, and then find the largest number in  $\lceil \log(n - 1) \rceil$  comparisons. (This number is not  $\lceil \log(n - 1) \rceil - 1$  because the new leaf is an element, not  $-\infty$ ). We know this number could not be the 3<sup>rd</sup> largest number either. Finally, we use  $\lceil \log(n - 1) \rceil - 1$  comparisons to find the largest number in the rest  $n - 2$  numbers, and this number we picked out is the 3<sup>rd</sup> largest number.

---

Example:

$S = \{A, B, C, D, E\}$

```

      A
     A  C
    A  B  C  D

```

3 comparisons. A could be the 1<sup>st</sup> or 2<sup>nd</sup> one. (Maybe  $E > A$ )

```

      Y
     X  C
    E  B  C  D

```

2 comparisons. Y could be the 1<sup>st</sup> or 2<sup>nd</sup> one. (Maybe  $A > Y$ )

Finally we find the path of Y, substitute the leaf with  $-\infty$ , update the tree, and find the largest number among the rest 3 elements, which is the 3<sup>rd</sup> largest among the original array. (1 comparison only)

---

$$\begin{aligned} \text{Total \# of comparisons} &= (n - 2) + \lceil \log(n - 1) \rceil + (\lceil \log(n - 1) \rceil - 1) \\ &= n + 2\lceil \log(n - 1) \rceil - 3 \end{aligned}$$

Here we don't know which number is the largest one, or which is the second largest. We only

know which two are the first two largest elements.

Criteria:

You get full credits if your answer is no larger than  $n + 2\lceil \log n \rceil - 3$ .

You get +4 bonus if you came up with the above algorithm and answered the yes/no question correctly. (It seems that nobody got this bonus, hmmm)

Notes:

Let  $V_t(n)$  denote the smallest worst-case comparisons if we want to find the t-th largest element in n elements.

Let  $W_t(n)$  denote the smallest worst-case comparisons if we want to find the first t-th largest elements (i.e., 1<sup>st</sup>, 2<sup>nd</sup>, ..., and t<sup>th</sup>) in n elements.

We have:

$$\begin{aligned}V_t(n) &\leq W_t(n) \\V_2(n) = W_2(n) &= n + \lceil \log n \rceil - 2 \\V_t(n) &= V_{n-t}(n) \\W_t(n) &= W_{n-t}(n) \\V_t(n) &\leq n - t + (t - 1)\lceil \log_2(n + 2 - t) \rceil \\W_t(n) &\leq n - t + \sum_{j=n+2-t}^n \lceil \log_2 j \rceil \\V_3(5) &= 6; W_3(5) = 7\end{aligned}$$

In order to determine the t-th largest element in an array, we have to know which t elements are the first t largest elements.

The algorithm we used is called Tournament Algorithm, and the data structure we used is called Loser Tree.

For more information, you can visit

[http://en.wikipedia.org/wiki/Selection\\_algorithm#Selecting\\_k\\_smallest\\_or\\_largest\\_elements](http://en.wikipedia.org/wiki/Selection_algorithm#Selecting_k_smallest_or_largest_elements)

or refer to

Donald E Knuth, *The Art of Computer Programming*, Vol3, Section 5.3.3, *Minimum-Comparison Selection*, 2<sup>nd</sup> edition.

4-16:

We generalize the algorithm to find the k-th smallest element in an array. Arbitrarily pick up an element as the pivot, do the partition, and determine the position of this pivot in the original array. If it's in the smaller half, we search only in the larger half, and vice versa. For example, n equals to 9, and we know the pivot we picked is the 3<sup>rd</sup> smallest number in the array. Then we just recursively search the 2<sup>nd</sup> largest number in the sub-array which contains all the numbers larger than the pivot.

Now we guess the expected running time is in  $O(n)$ , but the formal proof is not trivial.

Notes: Considering the probabilistic analysis on randomized algorithms is not taught, I won't deduct marks if you do not write the proof. If you are interested in it, you may refer to *Introduction to Algorithms*, section 9.2, *Selection in expected linear time*, 2<sup>nd</sup> edition.

4 Exercises 4-17 to 4-19 (7 pts + 6 pts + 7 pts = 20)

4-17: Let  $T(n)$  be the # of comparisons that Quicksort makes in our algorithm.

(a) We have  $T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil - 1\right) + (n - 1)$  ( $n \geq 3$ );  $T(2) = 1$ ;  $T(1) = 0$

Here we prove  $T(n) = \Theta(n \log n)$

First, using mathematical induction, we can prove  $T(2^n - 1) = n2^n - 2^{n+1} + 2$

$n=1$ : trivial

Suppose when  $n = k - 1$  ( $k \geq 2$ ) this equation holds, then when  $n = k$ , we have:

$$T(2^k - 1) = 2T(2^{k-1} - 1) + 2^k - 2 = 2((k-1)2^{k-1} - 2^k + 2) + 2^k - 2 = n2^n - 2^{n+1} + 2.$$

So it also holds.

Then for any  $n \geq 1$ , when  $k = \lfloor \log(n+1) \rfloor$ , we have  $2^k - 1 \leq n < 2^{k+1} - 1$

It's easy to prove  $T(x) \leq T(y)$ , when  $x \leq y$ , if we use mathematical induction.

So we have:  $T(n) \geq T(2^k - 1) = k2^k - 2^{k+1} + 2 > k\frac{n}{2} - 2n = \lfloor \log(n+1) \rfloor \frac{n}{2} - 2n$

(since  $2^k > \frac{n}{2}$ ,  $-2^{k+1} + 2 \geq -2n$ )

Thus  $T(n) = \Omega(n \log n)$

$$\begin{aligned} \text{Also, } T(n) &\leq T(2^{k+1} - 1) = (k+1)2^{k+1} - 2^{k+2} + 2 < (k+1) \cdot 2(n+1) - 2n \\ &= 2\lfloor \log(n+1) \rfloor (n+1) + 2 \end{aligned}$$

(since  $2^{k+1} \leq 2(n+1)$ ,  $-2^{k+2} + 2 < -2n$ )

Thus  $T(n) = O(n \log n)$

So we have proved  $T(n) = \Theta(n \log n)$

(b) Similarly, we have  $T(n) = T\left(\left\lfloor \frac{2n}{3} \right\rfloor\right) + T\left(\left\lceil \frac{n}{3} \right\rceil - 1\right) + (n - 1)$  ( $n \geq 2$ );  $T(1) = T(0) = 0$

To simplify the problem, first we write  $T(n) = T\left(\frac{2n}{3}\right) + T\left(\frac{n}{3}\right) + n$  instead.

Using mathematical induction, we can prove  $T(n) \leq n \log_{\frac{3}{2}} n$  when  $n$  is large enough:

$$T(n) \leq \frac{2n}{3} \log_{\frac{3}{2}} \frac{2n}{3} + \frac{n}{3} \log_{\frac{3}{2}} \frac{n}{3} + n \leq n \log_{\frac{3}{2}} \frac{2n}{3} + n = n \log_{\frac{3}{2}} n$$

So  $T(n) = O(n \log n)$

On the other hand,  $T(n) \geq T\left(\frac{n}{3}\right) + n \geq T\left(\frac{n}{9}\right) + 2n \geq T\left(\frac{n}{27}\right) + 3n \geq \dots \geq n \lfloor \log_3 n \rfloor$

So  $T(n) = \Omega(n \log n)$

Thus we have  $T(n) = \Theta(n \log n)$ .

In fact, the ceiling or floor function, or the difference between  $T(n) = \dots + n$  and  $T(n) = \dots + n - 1$  doesn't influence the asymptotic complexity of the algorithm. Details omitted.

Criteria:

-1 each if you cannot formally prove the asymptotic complexity (using the recursion tree is acceptable).

4-18:

Use the idea that is used in Partition() in Quicksort. First we define a partial order relationship on colors: *red* < *white* < *blue*. Then we do a linear sweep to find the first element whose color is red. If there is such an element, we treat it as the pivot, swap it with the last element, and then call partition(A, 1, n) directly (change  $s[i] < s[p]$  to  $\text{Examine}(s, i) \leq \text{Examine}(s, p)$  of course). Then suppose we know that there are  $j$  reds in the array. We do a linear sweep to find the first element whose color is white. If there is, we swap it with the last element, and then call partition(A, j+1, n). It's clear that this algorithm is linear.

4-19:

(a) There are  $\binom{n}{2} = \frac{n(n-1)}{2}$  pairs of elements; the permutation with a reversed order has (i.e.,  $(n, n-1, n-2, \dots, 1)$ ).

(b) First we see that  $P[i]$  and  $P[j]$  are out of order  $\Leftrightarrow P^r[n+1-i]$  and  $P^r[n+1-j]$  are not out of order.

Then we define  $f(P)$  be the number of inversions of  $P$ . And for any  $i, j$ ,  $[P[i] > P[j]]$  equals to 1 if  $P[i] > P[j]$ ; otherwise it equals to 0.

When  $n \geq 2$ , we have:

$$\begin{aligned} f(P) + f(P^r) &= \sum_{i < j} [P[i] > P[j]] + \sum_{i < j} [P^r[i] > P^r[j]] \\ \sum_{i < j} [P^r[i] > P^r[j]] &= \sum_{1 \leq i < j \leq n} [P[n+1-i] > P[n+1-j]] \\ &= \sum_{1 \leq n+1-j < n+1-i \leq n} [P[j] > P[i]] = \sum_{i < j} [P[i] < P[j]] \end{aligned}$$

Here we substitute  $i$  with  $n+1-j$ ,  $j$  with  $n+1-i$ .

$$\text{So } f(P) + f(P^r) = \sum_{i < j} [P[i] > P[j]] + \sum_{i < j} [P[i] < P[j]] = \sum_{i < j} 1 = \frac{n(n-1)}{2}$$

When  $n = 1$ , this equation also holds.

(c) When  $n \geq 2$ , first we see we can divide the whole  $n!$  permutations into two groups, and each permutation and its reversal are in different groups.

Then we have:

$$E(f(P)) = \frac{E(f(P)) + E(f(P^r))}{2} = \frac{E(f(P) + f(P^r))}{2} = \frac{E\left(\frac{n(n-1)}{2}\right)}{2} = \frac{n(n-1)}{4}, \text{ for any } n \geq 1.$$

Note that when  $P$  runs over all permutations,  $P^r$  also does.

## 5 Exercises 4-21 and 4-24 (10 pts + 10 pts = 20)

4-21:

Build a balanced binary search tree. Each node of the tree records the key and its number of appearances in the array. The keys in the nodes are distinct. We first insert the integers into the tree one by one, and then do an inorder traversal to output the results. The running time is  $O(n \log k)$  since the height of the search tree is  $\log k$ .

4-24:

Using the same strategy as the previous problem, we have an  $O(n \log \log \log n)$  solution.