HW3 Solutions

#1 Solution
Randomly choosing one element and do the partition using this as the pivot is what 'randomized partition' means. Then, we can use this randomized partition to find out $\frac{n}{4}$-th element as follows.

SEARCH_QUARTER(A)

1. Randomly choose one element x and partition the array using x as the pivot.

2. If x's index is

   (a) greater than $\frac{n}{4}$: call **SEARCH_QUARTER(A')** where A' is the array of elements who are at the left side of x after the partition.

   (b) less than $\frac{n}{4}$: call **SEARCH_QUARTER(A')** where A' is the array of elements who are at the right side of x after the partition.

   (c) equal to $\frac{n}{4}$: x is the $\frac{n}{4}$-the element.

The complexity of randomized partition over n items is $O(n)$.

- Best case: our first-round random selection is the $\frac{n}{4}$-th element, in which case the complexity is $O(1)$.

- Worst case: our random selection at each recursion is the largest element in the array, in which case the complexity os $O(n + n - 1 + n - 2 + \cdots) = O(n^2)$.

- Average case: similar to the average case of the quick sort algorithm, the complexity in average cases (*i.e.,* the expected value of complexity) will be (i_{piv} is the index of the chosen pivot):

$$
\begin{aligned}
E\Big(T(n)\Big) &= \frac{1}{n}\Big[\sum_{i_{piv}=1}^{\frac{n}{4}-1}\Big(T(n-i_{piv})+n\Big) + \sum_{i_{piv}=\frac{n}{4}}^{\frac{n}{4}}(0+n) + \sum_{i_{piv}=\frac{n}{4}+1}^{n}\Big(T(i_{piv})+n\Big)\Big] \\
&= n + \frac{1}{n}\Big[\sum_{i_{piv}=1}^{\frac{n}{4}-1} T(n-i_{piv}) + \sum_{i_{piv}=\frac{n}{4}+1}^{n} T(i_{piv})\Big] \\
&= n + \frac{1}{n}\Big[\sum_{i_{piv}=\frac{n}{4}+1}^{\frac{3n}{4}} T(i_{piv}) + 2\cdot\sum_{i_{piv}=\frac{3n}{4}+1}^{n} T(i_{piv})\Big]
\end{aligned}
$$

We let $f(x) = \sum_{i=\frac{x}{4}+1}^{\frac{3x}{4}} T(i) + 2\cdot\sum_{i=\frac{3x}{4}}^{x} T(i)$. Since $E(E(T(n))) = E(T(n))$, we have

$$
\begin{aligned}
E\Big(T(n)\Big) &= E\Bigg(E\Big(T(n)\Big)\Bigg) \\
&= E\Bigg(n + \frac{1}{n}\Big[\sum_{i_{piv}=\frac{n}{4}+1}^{\frac{3n}{4}} T(i_{piv}) + 2\cdot\sum_{i_{piv}=\frac{3n}{4}+1}^{n} T(i_{piv})\Big]\Bigg) \\
&= n + \frac{1}{n}\Big[\sum_{i_{piv}=\frac{n}{4}+1}^{\frac{3n}{4}} E\Big(T(i_{piv})\Big) + 2\cdot\sum_{i_{piv}=\frac{3n}{4}+1}^{n} E\Big(T(i_{piv})\Big)\Big] \\
&= n + \frac{1}{n}\Bigg[\sum_{i_{piv}=\frac{n}{4}+1}^{\frac{3n}{4}}\Big[i_{piv} + \frac{1}{i_{piv}}\Big(\sum_{i=\frac{i_{piv}}{4}+1}^{\frac{3i_{piv}}{4}} T(i) + 2\cdot\sum_{i=\frac{3i_{piv}}{4}+1}^{i_{piv}} T(i)\Big)\Big] \\
&\qquad + 2\cdot\sum_{i_{piv}=\frac{3n}{4}+1}^{n}\Big[i_{piv} + \frac{1}{i_{piv}}\Big(\sum_{i=\frac{i_{piv}}{4}+1}^{\frac{3i_{piv}}{4}} T(i) + 2\cdot\sum_{i=\frac{3i_{piv}}{4}+1}^{i_{piv}} T(i)\Big)\Big]\Bigg] \\
&= n + \frac{n+1}{2} + \frac{1}{n}\Bigg[\sum_{i_{piv}=\frac{n}{4}+1}^{\frac{3n}{4}} \frac{1}{i_{piv}}\Big(\sum_{i=\frac{i_{piv}}{4}+1}^{\frac{3i_{piv}}{4}} T(i) + 2\cdot\sum_{i=\frac{3i_{piv}}{4}+1}^{i_{piv}} T(i)\Big) \\
&\qquad + 2\cdot\sum_{i_{piv}=\frac{3n}{4}+1}^{n} \frac{1}{i_{piv}}\Big(\sum_{i=\frac{i_{piv}}{4}+1}^{\frac{3i_{piv}}{4}} T(i) + 2\cdot\sum_{i=\frac{3i_{piv}}{4}+1}^{i_{piv}} T(i)\Big)\Bigg]
\end{aligned}
$$

Recursively, we can again apply the expectation $E(\cdot)$ to the both sides and then get the next equation, which will be approximately $n + \frac{n}{2} + \frac{n}{4} + \frac{1}{n}\Big[\cdots\Big]$ with exponentially decreasing negligible constants, and the next equation will show $E(T(n))$ will be approximately equal to $n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \frac{1}{n}\Big[\cdots\Big]$. By induction, we can show that $E\Big(T(n)\Big) = O(n)$. Therefore, the average case complexity is $O(n)$.

#2 Solution

When the input array is already sorted in increasing order, MAX-HEAPIFY takes $\log n^*$ steps every time at each iteration, where n^* is the number of elements remaining in the array at the iteration. Then, the number of steps we have in the heapsort algorithm is

$$\begin{aligned}
&\log n + \log n - 1 + \cdots + \log \frac{n}{2} + 1 + \log \frac{n}{2} + \cdots + \log 3 + \log 2 \\
\geq &\log n + \log n - 1 + \cdots + \log \frac{n}{2} + 1 + \log \frac{n}{2} \\
\geq &\frac{n}{2} \log \frac{n}{2} = \frac{n}{2}(\log n - 1) \geq \frac{1}{4} n \log n \quad \text{for all } n \geq 4
\end{aligned}$$

Therefore, such arrays require $\Omega(n \log n)$ steps, which implies it requires $cn \log n$ steps for a constant c.

#3 Solution

We will also store the additional data of which of the k sorted lists a number (X) belongs to, along with the keys, and augment the key value as (X,L). L is the sorted list(1...k) X belongs to. ($\approx O(n)$)

1. Extract the minimum elements from each of the k lists and put it in one heap H. This takes O(k*log k) time.

2. while H is not empty;(n times $\approx O(n)$)

(i) Extract-min element M from H and store it in the result list.($\approx O(logk)$)

(ii) Insert the next element from the heap L in (M,L), and insert it into H.($\approx O(logk)$)

Therefore, total time taken $= O(n) + O(klogk) + O(n * 2logk) \approx O(n * logk)$

#4 Solution

In radix sort we start with the least significant digits first. So at the i^{th} iteration of radix sort, the numbers are sorted with respect to the i^{th} least

significant digits.
So, we can output any number as soon as we consider all its digits we can put it in the right place in the sorted array.
k is the total number of digits on all numbers.
n is the size of the list.
To output the numbers whose $length = i$ after i^{th} iteration, we need to check the length of each number before putting it in a bucket at round$i+1$.
If number length is less than i then move the number to the result list.
So the time complexity is $O(n+k)$

#5 Solution
To get the lower bound on the solution we need to consider a binary tree with permutations of n elements as leaf nodes, out of which only one is the correct sorting result. Now, we only need to count the number of different permutations. Each sub-sequence has $(\log n)!$ permutations, and we have $(n/\log n)$ sub-sequences. That means we have $\Big((\log n)!\Big)^{n/\log n}$ different permutations in our problem. The permutation is not $n!$ because we already know the order of sub-sequences. Any sorting algorithm over n items would need to 'go through' the binary tree to reach one of the many leaf nodes, and each 'hop' in the tree is one necessary 'step' in the comparison-based sorting, which implies the depth of the tree is the lower bound of all sorting algorithms over those n items because not all algorithms will be so efficient that they only make one 'step' for each 'hop'. The depth of the tree containing $\Big((\log n)!\Big)^{n/\log n}$ leaf nodes is

$$\begin{aligned}
\log\left(\Big((\log n)!\Big)^{n/\log n}\right) &= \frac{n}{\log n}\log\left(\Big((\log n)!\Big)\right) \\
&= \Theta\Big(\frac{n}{\log n}\log n \log(\log n)\Big) \\
&= \Theta(\frac{n}{\log}\log n)
\end{aligned}$$