Problem 1: Given a set $A$ of $n$ distinct integers we want to find the $m^{th}$ smallest element of $A$

- Give a deterministic $\Theta(n \log n)$ algorithm to find the median of $A$ [2pts];
- Give a randomized algorithm that finds the median in expected $O(n)$ time. [8pts]

Solution:

- A $\Theta(n \log n)$ algorithm can be achieved by first sorting the array $A$ (using say MERGE-SORT) and then selecting $A[m]$.

- Our algorithm works like randomized QUICKSORT. It chooses a random "pivot" $p$ and then divides the array in two parts: $A_S$ of the elements smaller than the pivot and $A_L$ of those larger than the pivot. If $|A_S| = m - 1$ then it returns the pivot as the $m^{th}$ element. If $|A_S| > m - 1$ then obviously, the $m^{th}$ element in $A$ is also the $m^{th}$ element in $A_S$. Finally if $|A_S| < m - 1$ then the $m^{th}$ element in $A$ is $\ell^{th}$ element in $A_L$ where $\ell = m - 1 - |A_S|$. The last two cases are dealt by calling the algorithm recursively on the appropriate input.

\[
\text{Statistic}(A, m) \\
u \leftarrow \text{Random}(1, n) \\
p \leftarrow A[u] \\
A_S \leftarrow \{a \in A : a < p\}; \\
A_L \leftarrow \{a \in A : a > p\}; \\
\text{If } |A_S| > m - 1 \text{ Then Return } \text{Statistic}(A_S, m); \\
\text{If } |A_S| = m - 1 \text{ Then Return } p; \\
\text{If } |A_S| < m - 1 \text{ Then} \\
\ell \leftarrow m - 1 - |A_S|; \\
\text{Return } \text{Statistic}(A_L, \ell);
\]

To analyze the running time we note that the first four lines of the algorithm take $f(n) = \Theta(n)$ time since we must read the entire array $A$ to divide it into $A_S$ and $A_L$.

We then note that for every $j$ between $m$ and $n - 1$ we recurse on an array of size $j$ with probability $1/n$ (this is when we recurse on $A_S$). Also for every $j$ between $1$ and $m - 1$ we recurse on an array of size $n - m$ with probability $1/n$ (this is when we recurse on $A_L$). Finally with probability 1/n we do not recurse at all. So the expected running time $\tilde{T}(n)$ of \text{Statistic}(A, j)$ is

\[
\tilde{T}(n) = \frac{1}{n} f(n) + \frac{1}{n} \sum_{j=m}^{n-1} [\tilde{T}(j) + f(n)] + \frac{1}{n} \sum_{j=1}^{m-1} [\tilde{T}(n - j) + f(n)]
\]

or

\[
\tilde{T}(n) = \frac{1}{n} \sum_{j=m}^{n-1} \tilde{T}(j) + \frac{1}{n} \sum_{j=1}^{m-1} \tilde{T}(n - j) + f(n)
\]

Let $c_1, n_0$ be the constants such that $f(n) < c_1 n$ for $n > n_0$. Then let’s assume that $\tilde{T}(j) < c_2 j$ for $j < n$. By using the substitution method we get that for $n > n_0$

\[
\tilde{T}(n) < \frac{c_2}{n} \left[ \sum_{j=m}^{n-1} j + \sum_{j=1}^{m-1} (n - j) \right] + c_1 n < \frac{c_2}{n} \left[ \sum_{j=1}^{n-1} j + (m - 1)(n - m) \right] + c_1 n
\]
where the left end side is derived by noting that
\[
\sum_{j=m}^{n-1} j + \sum_{j=1}^{m-1} (n - j) = \sum_{j=1}^{n-1} j - \sum_{j=1}^{m-1} j + \sum_{j=1}^{m-1} (n - j) = \sum_{j=m}^{n-1} (j - m) + \sum_{j=1}^{m-1} (n - j) = \\
\sum_{j=1}^{n-1} j + \sum_{j=1}^{m-1} (n - m) = \sum_{j=1}^{n} j + (m - 1)(n - m)
\]

Note that \((m - 1)(n - m)\) is maximized by \(m = n/2\) and that \(\sum_{j=1}^{n-1} j = n(n - 1)/2\) so we get
\[
\bar{T}(n) < c_2 (n - 1) + \frac{c_2}{2} (\frac{n}{2} - 1) + c_1 n = (c_1 + \frac{3}{4} c_2)n - c_2 < c_2 n
\]
provided that \(c_2 > 4c_1\).

The above proves that \(\bar{T}(n) < c_2 n\) for \(n > n_0\) and therefore that \(\bar{T}(n) = O(n)\).

**Problem 2:** A different way to randomize Quick-Sort is to use the deterministic version of Quick-Sort over a 'randomized' array, according to the following pseudo-code

**Permute-Quick-Sort(A)**

\[
B \leftarrow \text{Random-Permute}(A); \\
\text{Return Quick-Sort}(B)
\]

- Under what conditions on the procedure **Random-Permute** will **Permute-Quick-Sort** run in \(O(n \log n)\) steps? [2pts]
- Consider the following procedure

**Block-Permute(A)**

\[
n \leftarrow |A|; \\
s \leftarrow \text{Random}(1, n); \\
\text{For } i = 1 \text{ To } n - s \text{ Do } B[i] \leftarrow A[s + i]; \\
\text{For } i = 1 \text{ To } s \text{ Do } B[s + i] \leftarrow A[i]; \\
\text{Return } B
\]

What is the expected running time of **Permute-Quick-Sort** if you use **Block-Permute** in place of **Random-Permute**? [4pts]

- Give your own implementation of **Random-Permute** that will make **Permute-Quick-Sort** run in \(O(n \log n)\) steps. [4pts]

**Solution:**

- A sufficient condition for **Permute-Quick-Sort** to run in \(O(n \log n)\) steps is that the output of **Random-Permute** is uniformly distributed among all possible permutations of the array \(A\). In other words given a permutation of the elements of \(A\), that permutation is output by **Random-Permute** with probability \(\frac{1}{n!}\).

- We note first of all that **Block-Permute** does not output all possible \(n!\) permutations of \(A\) (each with probability \(\frac{1}{n!}\)). Rather, it outputs only \(n\) of those permutations (the one resulting from swapping the 2 blocks of the array, the ones before and after the pivot), each with probability \(\frac{1}{n}\). This lack of "entropy" in the distribution of the output will not "destroy" a worst-case input causing a running time of \(\Theta(n^2)\) on some inputs.

Consider the case of an array which is already sorted. For \(i = 1, \ldots, n\), with probability \(1/n\) **Shift-Permute** will bring the \(m^{th}\) element to the first slot of the array, i.e. the pivot position. That means
that we will partition the array into two arrays of size \( m - 1 \) and \( n - m \) respectively. Because of the way \textsc{Partition} works those arrays will also be already sorted: that’s because \textsc{Partition} reads the array \( A \) in order and appends elements to the appropriate array as it reads them. Notice that when the algorithm recurses it does not randomize the array anymore: \textsc{Permute-Quick-Sort} makes only one random choice at the beginning and then calls the deterministic \textsc{QuickSort} algorithm. So the running time of the recursive calls will be quadratic. The expected running time is therefore

\[
\hat{T}(n) = \frac{1}{n} \sum_{m=1}^{n} [T_{\text{QS}}(m - 1) + T_{\text{QS}}(n - m) + \Theta(n)]
\]

Where \( T_{\text{QS}}(m) \) is the running time of deterministic quicksort on an array of size \( m \). As we saw in class this is the same as

\[
\hat{T}(n) = \frac{2}{n} \sum_{m=1}^{n-1} T_{\text{QS}}(m) + \Theta(n)
\]

We know that \( T_{\text{QS}}(m) = \Theta(m^2) \) and therefore

\[
\hat{T}(n) = \frac{2}{n} \sum_{m=1}^{n-1} \Theta(m^2) + \Theta(n) = \Theta(m^2)
\]

The last step follows from the fact that

\[
\sum_{m=1}^{n} m^2 < cn^3
\]

for a constant \( c \), which can be proven by induction. Indeed this is true for \( n = 1 \) since \( \sum_{m=1}^{1} m^2 = 1 < cn^3 = c \) for \( c > 1 \). Then assume it’s true for \( n - 1 \), we have that

\[
\sum_{m=1}^{n} m^2 = n^2 + \sum_{m=1}^{n-1} m^2 < n^2 + c(n - 1)^3 = cn^3 - (3c - 1)n^2 + 3cn - 1 < cn^3
\]

provided that

\[
(3c - 1)n^2 - 3cn + 1 > 0
\]

which is true for sufficiently large \( n \).

• Following the answer to the first question, we need to make sure that \textsc{Random-Permute} outputs a permutation chosen uniformly at random among all possible permutations of \( n \) elements. A way to do this is to select \( n \) random element between 1 and \( n \) without repetition. So choose the first one at random, then choose the second one at random among the remaining ones and so on.

**Problem 3:** At the end of the academic year CUNY will issue a sorted list of all its Computer Science students, ranked by their score on the Algorithms course. For each section of the course, instructors have been asked to return a sorted list of the students in that section according to their score. At the end of the year CUNY will have \( k \) sorted lists (one for each section), with \( n \) students in total. Show how to produce the total sorted list in \( O(n \log k) \) steps.

**Solution:** Let \( A \) be the total list and \( A_i \) each of the \( k \) sorted lists.

A simple algorithm would start with an empty \( A \), compare the minimum of each sorted list \( A_i \), choose the smallest of them, remove it from its list and append it to \( A \). Do this until the sorted lists are empty. But this algorithm takes \( O(nk) \) steps because it takes \( O(k) \) to select the smallest among the local minima and the process repeats \( O(n) \) times.

A more efficient solution is to create a heap with the \( k \) minima. This takes \( O(k) \). Then at each step we extract the minimum from this heap and append it to the list \( A \) – this step costs \( O(\log k) \). Say that this element belonged to the sorted list \( A_j \). Then we take the next element from \( A_j \) and insert it in the heap – this step costs \( O(\log k) \) as well. Repeat the above until all elements are in the list \( A \). The total cost of this algorithm is therefore \( O(n \log k) \).