Problem 1: Given a set $A$ of $n$ distinct integers we want to find the median of $A$, i.e. the element $a \in A$ such that

$$|\{ x \in A : x \leq a \}| - |\{ x \in A : x > a \}| \leq 1$$

(the above is a fancy way to state that $a$ is the element such that $A$ has the same number of elements which are smaller than $a$ and larger than $a$, with the difference of 1 allowed to account for an odd number $n$ of total elements).

- 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 the median as $A[k]$ where $k = \frac{n+1}{2}$ if $n$ is odd, and $k = \frac{n}{2}$ if $n$ is even.
- We are going to show an algorithm $\text{Statistic}(A, j)$ that finds the $j^{th}$ element in the array, i.e. the element $a$ such that $|\{ x \in A : x \leq a \}| = j$. Then it will be sufficient to invoke $\text{Statistic}(A, k)$ where $k$ is defined as above.

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 $|A_S| = j - 1$ then it returns the pivot as the $j^{th}$ element. If $|A_S| > j - 1$ then obviously, the $j^{th}$ element in $A$ is also the $j^{th}$ element in $A_S$. Finally if $|A_S| < j - 1$ then the $j^{th}$ element in $A$ is the $\ell^{th}$ element in $A_L$ where $\ell = j - 1 - |A_S|$. The last two cases are dealt by calling the algorithm recursively on the appropriate input.

$$\text{Statistic}(A, j)$$

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

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 $m$ between $j$ and $n - 1$ we recurse on an array of size $m$ with probability $1/n$ (this is when we recurse on $A_S$). Also for every $m$ between $1$ and $j - 1$ 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

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

or

$$\tilde{T}(n) = \frac{1}{n} \sum_{m=j}^{n-1} \tilde{T}(m) + \frac{1}{n} \sum_{m=1}^{j-1} \tilde{T}(n - m) + f(n)$$
Let $c_1, n_0$ be the constants such that $f(n) < c_1 n$ for $n > n_0$. Then let’s assume that $\tilde{T}(m) < c_2 m$ for $m < n$.

By using the substitution method we get that for $n > n_0$

$$\tilde{T}(n) < \frac{c_2}{n} \left[ \sum_{m=j}^{n-1} m + \sum_{m=1}^{j-1} (n-m) \right] + c_1 n = \frac{c_2}{n} \left[ \sum_{m=1}^{n-1} m + (j-1)(n-j) \right] + c_1 n$$

where the left end side is derived by noting that

$$\sum_{m=j}^{n-1} m + \sum_{m=1}^{j-1} (n-m) = \sum_{m=1}^{n-1} m - \sum_{m=1}^{j-1} m + \sum_{m=1}^{j-1} (n-m) = \sum_{m=1}^{n-1} m + \sum_{m=1}^{j-1} (m-j) + \sum_{m=1}^{j-1} (n-m)$$

Note that $(j-1)(n-j)$ is maximized by $j = n/2$ and that $\sum_{m=1}^{n-1} m = n(n-1)/2$ so we get

$$\tilde{T}(n) < \frac{c_2}{2} (n-1) + \frac{c_2}{2} \left( \frac{n}{2} - 1 \right) + c_1 n = \left( c_1 + \frac{3}{4} c_2 \right) n - c_2 < c_2 n$$

provided that $c_2 > 4c_1$.

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

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

SHIFT-PERMUTE$(A)$

\[ n \leftarrow |A|; \]
\[ s \leftarrow \text{RANDOM}(1, n); \]
\[ \text{FOR } i = 1 \text{ TO } n \]
\[ j \leftrightarrow s + i \mod n; \]
\[ B[j] \leftarrow A[i]; \]
\[ \text{RETURN } B \]

What is the expected running time of PERMUTE-QUICK-SORT if you use SHIFT-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 SHIFT-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 a simple shift), 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 \( \frac{1}{n} \) \textsc{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

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

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

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

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

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

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-Permутe} 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: Let \( a \) and \( b \) be two \( n \) bit numbers (assume for simplicity that \( n \) is a power of 2).

• Describe the ”grade school” algorithm to multiply \( a \) and \( b \) and show that it requires \( O(n^2) \) steps; [2pts]

• Describe a divide-and-conquer algorithm with an asymptotically faster running time. [8pts]

Solution:

• The grade school algorithm requires pair-wise multiplications of all the digits in the two numbers, which is why it requires \( O(n^2) \) steps (since there are \( n^2 \) possible pairs). More specifically let \( a_0 \ldots a_{n-1} \) and \( b_0 \ldots b_{n-1} \) be bits such that \( a = \sum_i a_i 2^i \) and \( b = \sum_i b_i 2^i \). Let \( c_{ij} = a_j b_i \). We construct \( n \) numbers, each \( n \)-bit long

\[
c^{(k)} = \sum_i c_{ki} 2^i
\]
and
\[ ab = \sum_k c^{(k)}2^k \]
The computation of the \( c_{ji} \) takes \( O(n^2) \).

- Split \( a \) and \( b \) into the top and bottom half of the bits. Then
\[ a = \hat{a} + \bar{a}2^{n/2} \quad \text{and} \quad b = \hat{b} + \bar{b}2^{n/2} \]
where \( \hat{a}, \bar{a}, \hat{b}, \bar{b} \) are \( n/2 \)-bit numbers. We have that
\[ ab = \alpha + 2^{n/2}\beta + 2^n\gamma \]
where
\[ \alpha = \hat{a}\hat{b} \quad \beta = \hat{a}\bar{b} + \bar{a}\hat{b} \quad \gamma = \bar{a}\bar{b} \]
A trivial implementation of this algorithm would recurse four times on inputs of size \( n/2 \) to compute the four cross products. Since the additions can be done in \( \Theta(n) \) time we would have the recurrence
\[ T(n) = 4T(n/2) + \Theta(n) = \Theta(n^2) \]
by the Master Method. So this approach does not improve on the grade school algorithm.

However notice what happens if we compute
\[ \delta = (\hat{a} + \bar{a})(\hat{b} + \bar{b}) = \alpha + \beta + \gamma \]
Which means that we can recurse only three times on input of size \( n/2 \) to compute \( \alpha, \beta, \delta \) and then compute \( \beta = \delta - \alpha - \gamma \) to obtain \( ab \). Since all additions can be computed in \( \Theta(n) \) the running time of this algorithm satisfies the recurrence
\[ T(n) = 3T(n/2) + \Theta(n) = \Theta(n^\log_3 3) \]
which is asymptotically faster than \( \Theta(n^2) \).