CS170 Discussion Section 2: *1/25*

1. Recurrence Relations

1. $T(n) = 4T(n/2) + 42n$

   Use the master theorem. Or:

$$\begin{tikzcd}
 & & & 42n \arrow[dlll] \arrow[dl] \arrow[dr] \arrow[drrr] & & & \\
21n & & 21n & & 21n & & 21n
\end{tikzcd}$$

$$\vdots$$

   The first level sums to $42n$, the second sums to $84n$, etc. The last row dominates, and we have $\log n$ rows, so we have $42 \cdot 2^{\log n} \cdot n = \Theta(n^2)$.

2. $T(n) = 4T(n/3) + n^2$

   Use the master theorem (the case $d > \log_b a = \log_3 4$), or expand like the previous question. The answer is $\Theta(n^2)$.

3. $T(n) = 2T(2n/3) + T(n/3) + n^2$

   This one is a bit tricky:

$$\begin{tikzcd}
 & & & & n^2 \arrow[dlll] \arrow[d] \arrow[drrr] & & & & & \text{Sum: } n^2 \\
 & (\frac{2}{3}n)^2 \arrow[dl] \arrow[d] \arrow[dr] & & & (\frac{2}{3}n)^2 \arrow[dl] \arrow[d] \arrow[dr] & & & (\frac{1}{3}n)^2 \arrow[dl] \arrow[d] \arrow[dr] & & \text{Sum: } n^2 \\
(\frac{2}{3}\frac{2}{3}n)^2 & (\frac{2}{3}\frac{2}{3}n)^2 & (\frac{2}{3}\frac{1}{3}n)^2 & (\frac{2}{3}\frac{2}{3}n)^2 & (\frac{2}{3}\frac{2}{3}n)^2 & (\frac{2}{3}\frac{1}{3}n)^2 & (\frac{1}{3}\frac{2}{3}n)^2 & (\frac{1}{3}\frac{2}{3}n)^2 & (\frac{1}{3}\frac{1}{3}n)^2 & \text{Sum: } n^2
\end{tikzcd}$$

$$\vdots$$

   So the answer is: $\Theta(n^2 \log n)$.

4. $T(n) = 3T(n/4) + n \log n$

   We end up with $\sum_{i=0}^{\log_4 n} (3/4)^i n \log(n/4^i)$. We can lower-bound this by $n \log n$ by taking the first term, and upper-bound it by $n \log n$ be replacing $\log(n/4^i)$ by $\log n$, so this is $\Theta(n \log n)$.

2. Counting inversions

This problem arises in the analysis of *rankings*. Consider comparing two rankings. One way is to label the elements (books, movies, etc.) from 1 to n according to one of the rankings, then order these labels according to the other ranking, and see how many pairs are "out of order".
We are given a sequence of n distinct numbers $a_1, \cdots, a_n$. We say that two indices $i < j$ form an inversion if $a_i > a_j$ that is if the two elements a_i and a_j are "out of order". Provide a divide and conquer algorithm to determine the number of inversions in the sequence $a_1, \cdots, a_n$ in time $O(n \log n)$ (*Hint:* Modify merge sort to count during merging)

(i) **Main idea**
There can be a quadratic number of inversions. So, our algorithm must determine the total count without looking at each inversion individually.
The idea is to modify merge sort. We split the sequence into two halves $a_1, \cdots, a_m$ and $a_{m+1}, \cdots, a_n$ and count number of inversions in each half while sorting the two halves separately. Then we count the number of inversions (a_i, a_j), where two numbers belong to different halves, while combining the two halves. The total count is the sum of these three counts.

(ii) **Psuedocode**

```
procedure COUNT($A$)
    if length[$A$] $= 1$ then
        return $A, 0$
    $B, x \leftarrow$ COUNT(first half of $A$)
    $C, y \leftarrow$ COUNT(rest of $A$)
    $D \leftarrow$ empty list
    $z \leftarrow 0$
    while $B$ is not empty and $C$ is not empty do
        if $B$ is empty then
            Append $C$ to $D$ and remove elements from $C$
        else if $C$ is empty then
            Append $B$ to $D$ and remove elements from $B$
        else if $B[1] < C[1]$ then
            Append $B[1]$ to $D$ and remove $B[1]$ from $B$
        else
            Append $C[1]$ to $D$ and remove $C[1]$ from $C$
            $z \leftarrow z +$ length[$B$]
    return $D, x + y + z$
```

(iii) **Proof of correctness** Consider now a step in merging. Suppose the pointers are pointing at elements b_i and c_j. Because B and C are sorted , if b_i is appended to D, no new inversions are encountered, since b_i is smaller than everything left in list C, and it comes before all of them. On the other hand, if c_j is appended to D, then it is smaller than all the remaining elements in B, and it comes after all of them, so we increase the count of inversions by the number of elements remaining in B.

(iv) **Running time analysis** In each recursive call, we merge the two sorted lists and count the inversions in $O(n)$. The running time is given by $T(n) = 2T(n/2) + O(n)$ which is $O(n \log n)$ by the master theorem.

3. Find the missing integer

An array A of length N contains all the integers from 0 to N except one (in some random order). In this problem, we cannot access an entire integer in A with a single operation. The elements of A are represented in binary, and the only operation we can use to access them is "fetch the jth bit of $A[i]$". Using only this operation to access A, give an algorithm that determines the missing integer by looking at only $O(N)$ bits. (Note that there are $O(N \log N)$ bits total in A, so we can't even look at all the bits). Assume the numbers are in bit representation with leading 0s.

(i) **Main idea** Look at least significant bits and compare the 0s and 1s. Discard the numbers whose least significant bit is of the larger set. The bit of the missing number at this position will be the bit of the smaller set. Recursively apply the algorithm and build the missing number at each bit position.

(ii) **Psuedocode**

```
procedure findMissing($A$)
    return findMissingNum($A, 0$)
procedure findMissingNum($A, m$)
    if length[$A$] = 0 then
        return $m$
    $B \leftarrow$ values of $A$ with LSB of 0        ▷ LSB stands for least significant bit
    $C \leftarrow$ values of $A$ with LSB of 1
    if length[$B$] $\leq$ length[$C$] then
        $B \leftarrow B$ with LSB of all numbers removed
        $m \leftarrow m$ with 0 append to least significant bit
        return findMissingNum($B, m$)
    else
        $C \leftarrow C$ with LSB of all numbers removed
        $m \leftarrow m$ with 1 append to least significant bit
        return findMissingNum($C, m$)
```

(iii) **Proof of correctness** Removing a number, m, creates an imbalance of 0s and 1s. If N was odd, the number of 0s in least significant bit position should equal the number of 1s. If N was even, then number of 0s should be 1 more than the number of 1s. Thus if all numbers are present, COUNT(0s) $\geq$ COUNT(1s). We have four cases:

- If LSB of m is 0, removing m removes a 0
  - If N is even, COUNT(0s) $=$ COUNT(1s)
  - If N is odd, COUNT(0s) $<$ COUNT(1s)
- If LSB of m is 1, removing m removes a 1

– If N is even, COUNT(0s) $>$ COUNT(1s)
– If N is odd, COUNT(0s) $>$ COUNT(1s)

Notice that if COUNT(0s) $\leq$ COUNT(1s) m's least significant bit is 0. We can discard all numbers with LSB of 1 because removing m does not the count of 1s. If COUNT(0s) $>$ COUNT(1s), m's least significant bit is 1. Likewise we can discard all numbers with LSB of 0.
Assume that this condition applies for all bit positions up to k. If we look at the $k+1$-th bit position, the condition above holds true. The elements at the $k+1$-th bit position have the same bit at the k-th position as m and thus are the only elements we are interested in at the $k+1$ position.

(iv) **Running time analysis** Since we care about the number of bits seen, creating the auxiliary arrays looks at $O(N)$ bits. In the worst case, the problem is reduced in half, so we have the recurrence $T(N) = T(N/2) + O(N)$, which, by master's theorem, gives us $T(N) = O(N)$.
As for runtime, creating the auxiliary arrays and counting the number of 0 bits and 1 bits takes $O(N \log N)$ time because each number has at most $\log N$ bits. In the worst case, the problem is reduced in half, so we have the recurrence $T(N) = T(N/2) + O(N \log N) = O(N \log N)$.

4. K-Largest Elements

Give an efficient algorithm to determine the k-largest elements of an unsorted list of integers of length n, in any order. Your algorithm should run in $O(n)$ expected time for a fixed k. You may assume $n \geq k$.

(i) **Main idea**

We can construct an algorithm very similar to quicksort, but the key idea here is we don't have to recursively call our function on *both* sub-lists, but only *one*.

(ii) **Psuedocode**

```
procedure QUICKMAX($L, k$)
    if length[$L$] = 1 then
        return $L$
    $x \leftarrow RandomChoice(L)$                          ▷ a random element from $L$
    $S \leftarrow$ all elements in $L$ that are $\leq x$
    $B \leftarrow$ all elements in $L$ that are $\geq x$
    if $k <=$ length[$B$] then
        return QUICKMAX($B$, $k$)
    else                                                    ▷ $i$-th element must be in D
        return QUICKMAX($S$, $k-$length[$B$]) + $B$
```

(iii) **Proof of correctness**

We will prove this in two pieces. First, we will prove the algorithm always returns exactly k values (remember, we can assume $n \geq k$). Then, we will prove that for a given L_i, L_j, if we have $L_i < L_j$ then the algorithm will discard L_i before L_j.

We will prove this by induction. We must be careful with our induction, here: we are inducting on two variables simultaneously. We require *strong* induction.

Proof of correctness By induction:

Induction hypothesis Let $P(n, k)$ be the proposition that **quickmax** finds the top k elements from a list of length n, or returns all n elements if $n < k$.

Base Case If $n = 1$, for any k, the algorithm returns the only element. If $k = 1$ then the algorithm reduces to quickselect and this was proved correct in the book.

Assume that $P(n, k)$ is true for all $n < N$ and $k < K$. Then we must prove that $P(N, K)$ is true.

Induction step $P(N, K)$ asks if **quickmax**(N, K) returns the correct answer on N and K. We know that **quickmax**(n, k) is correct if either $n < N$ or $k < K$.

We will split our induction on the conditional $K \leq \text{length}(B)$. If it is true, then clearly the k largest elements of L must be equal to the k largest elements of B since every element of B is larger than any element of S. We know by the induction hypothesis that the algorithm is correct when applied to a problem of size length(B) and K, so this case is true.

On the other hand, if $K > \text{length}(B)$, then B a strict subset of the top k elements in L. The remaining elements must come from S. We need our result to be of size k, so we must get the length$(B) - k$ top elements of S. But this is a smaller sub-problem, again, and by the induction hypothesis it is true.

Therefore, since $P(1, k)$ is true for all k and $P(n, 1)$ is true for all n, and $P(N, K)$ is true whenever $P(k, n)$ is true for all $k < K$ and $n < N$ we have that $P(K, N)$ is true for all K, N.

(iv) **Running time analysis** There is one recursive call to a problem of size $n/2$ on average, and $O(n)$ time to construct the two other calls, so the expected running time is $T(n) = T(\frac{n}{2}) + O(n) = O(n)$.

In the worst case, this algorithm can take $O(n^2)$ time if we are extremely unlucky, and the randomly chosen pivots consistently chooses the max/min element.