**Homework #2**

**CMSC351H - Spring 2019**

PRINT Name: ___________________________

- Grades depend on neatness and clarity.
- Write your answers with enough detail about your approach and concepts used, so that the grader will be able to understand it easily. You should ALWAYS prove the correctness of your algorithms either directly or by referring to a proof in the book.
- Write your answers in the spaces provided. If needed, attach other pages.
- The grades would be out of 16. Four problems would be selected and everyone’s grade would be based only on those problems. You will also get 4 bonus points for trying to solve all problems.

1. [Prob 3.4,Pg 56] By using Theorem 3.1 in the book (similar theorem is mentioned in the class), prove that $(\log_2 n)^{100} = O \left( \frac{1}{n^{10}} \right)$.

Let $f(n) = \frac{\log_2 n}{10}$. The left side is $(\log_2 n)^{100} = \left( 10f(n) \right)^{100} = 10^{100} \times (f(n))^{100} = \Theta \left( (f(n))^{100} \right)$ and the function in the right side is $n^{\frac{1}{10}} = 2^{\log_2 n \times \frac{1}{10}} = 2^{f(n)}$. By theorem 3.1: $(f(n))^{100} = O \left( 2^{f(n)} \right)$ therefore $(\log_2 n)^{100} = O \left( \frac{1}{n^{10}} \right)$.
2. Prove that $T(n) = O(n \log n)$, where $T(n)$ is defined by the following recurrence relation ($c$ is some constant)

$$T(n) = \begin{cases} 
  c & \text{if } n = 1, \\
  10T\left(\left\lfloor \frac{n}{10} \right\rfloor \right) + cn & \text{if } n > 1.
\end{cases}$$

This problem can be solved using Theorem 3.4. Another approach is to use induction. See problem 3 as an example.
3. [Prob 3.8, Pg 56] Prove that $T(n)$, which is defined by the recurrence relation

$$T(n) = \begin{cases} 4 & \text{if } n = 2, \\
2T\left(\left\lfloor \frac{n}{2} \right\rfloor \right) + 2n \log_2 n & \text{if } n > 2.
\end{cases}$$

satisfies $T(n) = O(n \log^2 n)$.

We assume that there exist a constant $c$ such that $T(n) \leq c \ n \log^2 n$, for every $n \geq 2$. We prove this (and find the exact value of $c$) by strong induction.

The base case is for $n = 2$. We should have $4 \leq c \ 2 \log^2 2 = 2c$. Thus we should choose $c$ to be greater than or equal to 2.

Now assuming that for all $m < n$ we have $T(m) \leq c \ m \log^2 m$, we want to prove that $T(n)$ is also less than or equal to $c \ n \log^2 n$ (for every $n$ greater than 2). We have:

$$T(n) = 2T\left(\left\lfloor \frac{n}{2} \right\rfloor \right) + 2n \log_2 n \leq 2c \left\lfloor \frac{n}{2} \right\rfloor \log^2 \left(\left\lfloor \frac{n}{2} \right\rfloor \right) + 2n \log n$$

$$\leq 2c \frac{n}{2} \log^2 \left(\frac{n}{2} \right) + 2n \log n = cn(\log n - 1)^2 + 2n \log n$$

$$= cn(\log^2 n - 2 \log n + 1) + 2n \log n = cn \log^2 n - (2c - 1)n \log n - 1)$$

Again by assuming $c \geq 2$ (or even greater than $\frac{7}{6}$), the term $(2c - 1)n \log n - 1$ would be always positive and therefore $T(n)$ would not be greater than $cn \log^2 n$.

[This proof is complete. However, since now we know that $c \geq 2$ is sufficient, one may find it easier to rewrite all the proof by putting 2 instead of $c$.]
4. [Prob 3.22-a, Pg 58] Solve the following recurrence relation. It is sufficient to find the asymptotic behavior of $T(n)$. (Hint: Substitute another variable for $n$)

$$T(n) = \begin{cases} 
1 & \text{if } n = 2, \\
4T(\lfloor \sqrt{n} \rfloor) + 1 & \text{if } n > 2.
\end{cases}$$

We define a new function $F(k)$ which is equal to: $F(k) = T(2^{(2^k)})$.

We note that $k = \log_2 \log_2 (2^{(2^k)})$. By the definition of $T(n)$, we have

$$F(k) = \begin{cases} 
1, & \text{if } k = 0 \\
4F(k - 1) + 1, & \text{if } k > 0
\end{cases}$$

One case easily show that $F(k) = \Theta(4^k)$ and therefore $T(n) = \Theta(4^{\log \log n}) = \Theta((2^{\log \log n})^2) = \Theta(\log^2 n)$.

It is worth mentioning that you can also solve this problem by substituting $n$ by $2^k$. 
5. The sieve of Eratosthenes is an algorithm for finding all the prime numbers \( p \) such that \( p \leq n \). The algorithm starts with marking every number \( i \leq n \) except 1 as prime numbers. Next, it iterates on \( i \) from 2 to \( \sqrt{n} \) and if \( i \) is still marked as a prime number, marks every number which \( i \) divides as a not prime number. Since for each composite number \( c \leq n \), there is a prime number \( p \leq \sqrt{n} \) such that \( p < c \) and \( p \) divides \( c \), number \( c \) is marked as not prime at some step before the for loop reaches number \( c \). This way, all numbers marked as prime numbers at the end of the algorithm are the prime numbers from 2 to \( n \). For more details see the [Wikipedia page](https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes). The running time of the algorithm equals

\[
T(n) = \sum_{p \leq \sqrt{n}} O\left(\frac{n}{p}\right)
\]

a) The running time of the algorithm is upper-bounded by \( T(n) \leq \sum_{i=1}^{n} O\left(\frac{n}{i}\right) \). Prove that \( T(n) = O(n \log n) \) by showing \( \sum_{i=1}^{n} O\left(\frac{n}{i}\right) = O(n \log n) \).

\[
\sum_{i=1}^{n} \frac{n}{i} = n + n \sum_{i=2}^{n} \frac{1}{i} \leq n + n \int_{1}^{n} \frac{1}{t} = O(n \log n).
\]

Note that we need to separate the case of \( i = 1 \) in order to upper-bound the summation with the integral. The integral doesn’t converge otherwise.

b) Use the following facts to prove \( T(n) = O(n \log \log n) \).

- Let \( \pi(n) \) be the number of primes numbers \( p \leq n \). Then, \( \pi(n) = \Theta\left(\frac{n}{\log n}\right) \) holds for all \( 1 < n \).

- \( \sum_{i=3}^{n} \frac{1}{\log i} = O(\log \log n) \). Why?

Although we iterate over primes up to \( \sqrt{n} \), we show the statement holds even if we take primes greater than \( \sqrt{n} \) into account.

\[
\sum_{p \leq \sqrt{n}} \frac{n}{p} \leq \sum_{p \leq n} \frac{n}{p} = n \sum_{p \leq n} \frac{1}{p}.
\]

We can replace \( \sum_{p \leq n} \frac{1}{p} \) by \( \sum_{i=2}^{n} \frac{\pi(i) - \pi(i-1)}{i} \). In this summation, the numerator equals 1 for primes and 0 otherwise. By using the facts, we have

\[
\sum_{p \leq n} \frac{1}{p} = \sum_{i=2}^{n} \frac{\pi(i) - \pi(i-1)}{i} = \frac{\pi(2) - \pi(1)}{2} + \frac{\pi(3) - \pi(2)}{3} + \ldots + \frac{\pi(n) - \pi(n-1)}{n} = \\
= \frac{\pi(n)}{n} - \frac{\pi(1)}{2} + \sum_{i=2}^{n-1} \frac{\pi(i)}{i} \left(\frac{1}{i} - \frac{1}{i+1}\right) = \frac{1}{\log n} + \sum_{i=2}^{n-1} O\left(\frac{i}{\log i}\right) \frac{1}{i(i+1)} = \\
= O(1) + \sum_{i=2}^{n-1} \frac{1}{(i+1) \log i} \leq O(1) + \sum_{i=3}^{n} \frac{1}{i \log i} = O(\log \log n).
\]
6. [Prob 3.12, Pg 57] Solve the following full-history recurrence relation:

\[ T(n) = \begin{cases} 
1 & \text{if } n = 1, \\
n + \sum_{i=1}^{n-1} T(i) & \text{if } n > 1.
\end{cases} \]

For \( n > 2 \), we have

\[ \begin{align*}
T(n) &= n + \sum_{i=1}^{n-1} T(i) \\
T(n - 1) &= n - 1 + \sum_{i=1}^{n-2} T(i)
\end{align*} \]

Subtracting the second equality from the first one, we have \( T(n) - T(n - 1) = 1 + T(n - 1) \).

Thus we can rewrite the relation in a simpler form:

\[ T(n) = \begin{cases} 
1, & n = 1 \\
3, & n = 2 \\
1 + 2T(n - 1), & n > 2
\end{cases} \]

Now using induction, one can easily show that \( T(n) = 2^n - 1 \).
7. [Prob 5.22,Pg 116] Towers of Hanoi: There are \( n \) disks of different sizes arranged on a peg in decreasing order of sizes. There are two other empty pegs. The purpose of the puzzle is to move all the disks, one at a time, from the first peg to another peg in the following way. Disks are moved from the top of one peg to the top of another. A disk can be moved to a peg only if it is smaller than all other disks on that peg. In other words, the ordering of disks by decreasing sizes must be preserved at all times. The goal is to move all the disks in as few moves as possible.

a) Design an algorithm (by induction) to find a minimal sequence of moves that solves the towers of Hanoi problem for \( n \) disks.

b) How many moves are used in your algorithm? Construct a recurrence relation for the number of moves, and solve it.

c) Prove that the number of moves in part b is optimal; that is, prove that no algorithm can use fewer moves (use induction).

a) Consider \( move(n, i, j) \) as the set of moves which can move the top \( n \) disks of peg \( i \) to peg \( j \), assuming that all the previous disks in peg \( j \) and peg \( k \) (the third peg) are bigger than the first \( n \) disks in peg \( i \). Now we can find this set of moves by this recursive definition (why?):

\[
move(n, i, j) = move(n - 1, i, k).move(1, i, j).move(n - 1, k, j).
\]

Where \( move(1, i, j) \) is just moving the topmost disk from \( i \) to \( j \).

b) Note that the number of moves in \( move(n, i, j) \) depends only on \( n \). We call this \( h(n) \). Using the recursive definition of the algorithm, we have \( h(n) = 2h(n - 1) + 1 \) in which \( h(1) = 1 \). Solving this recursive formula we get \( h(n) = 2^n - 1 \).

c) Let \( opt_n \) denote the minimum number of moves for moving \( n \) disks from one peg to another. We use induction to show \( opt_n \geq 2^n - 1 \). The base case is obvious. Suppose that we know the optimum movement scheme. To prove the inductive step, WLOG, we assume that at the end of optimum movement all the disks would be in peg 3. At some point, we have to move the biggest disk to peg 3 from some peg, say peg 1. One crucial observation is that in order to move the biggest disk from peg 1 to peg 3, we need to first move all the other \((n-1)\) disks from peg 1 to peg 2, hence using at least \( opt_{n-1} \) moves. Then we move the biggest disk and after that we again need to move the disks on peg 2 back to peg 3, thus paying another \( opt_{n-1} \).

Therefore:

\[
opt_n \geq opt_{n-1} + 1 + opt_{n-1} \geq (by\ induction) 2(2^{n-1} - 1) + 1 = 2^n - 1.
\]
8. Finding the majority: We have \( n \) numbers such that one number has appeared at least \( \left\lfloor \frac{n}{2} \right\rfloor + 1 \) times. Design an algorithm that finds this number by at most \( n \) comparisons between given numbers. (Hint: Use a similar idea as in Celebrity Problem).

Let \( A[1], ..., A[n] \) be the input. Assume that for some \( i \), there are exactly \( i/2 \) appearances of element \( A[1] \), in the range \( A[1], ..., A[i] \). One can easily prove that if \( v \) is the element with majority in the whole input, then \( v \) is still the majority in the range \( A[i+1], ..., A[n] \). Thus we can completely ignore the first \( i \) elements and process the rest of elements similarly, i.e., find the first position that \( A[i+1] \) has the same occurrences as all other elements together. Ignore this part and continue to the rest of elements. In any of these steps, if we couldn’t find such position, it means that the first of the remaining elements is the majority. The sudo-code is presented below:

```
Input: Array A[1..n]
Output: key
key ← A[1] //key is the first element of each part as mentioned above.
num ← 0 //num is the number of occurrences of key in the remaining elements as mentioned above.

for (i ← 1 to n) do
    if (A[i] == key)
        num ← num+1 //found another occurrence of key
    else
        num ← num-1 //found an element different from key
        if (num==0)
            key ← A[i+1] //here we have found a position where key has the same occurrences. Thus we should update the key.
end if
end if
Print key
```
9. A corrupted linked list is a linked list in which the pointer of its last element points to another element in the list instead of being NIL. Given a pointer to the first element of a linked list of size $n$, design an algorithm with $O(1)$ space and $O(n)$ running time which prints "YES" if the given linked list is corrupted and prints "NO" otherwise. Note that you cannot change the data in the linked list.

There are few different algorithms for this problem. One using only two pointers $p$ and $q$ is given below (if $p$ is a pointer pointing to an element of the linked list, we denote the next element of the list by $next(p)$). Furthermore, assume that $next(nil) = nil$:

- Initialize $p$ and $q$ to the first element of the list.
- Repeat:
  - $p \leftarrow next(p)$ // Move $p$ by one element
  - $q \leftarrow next(next(q))$ // Move $q$ by two elements
- Until either $p == q$ or $q == nil$.
- If ($p == q$) : Print YES
- Else : Print NO

If the linked list is not corrupted, then our algorithm outputs the correct answer. We need to show that if the list is corrupted, our algorithm ends in $O(n)$ time. Since the list is corrupted, the last element (i.e., $n^{th}$ element) points to some element, say $t^{th}$ element (note that obviously both $n$ and $t$ are unknown to the algorithm). Let $C$ be the length of the loop in the list, i.e., $C = n - t - 1$. Consider the $x^{th}$ iteration of the algorithm for any $x \geq t$. Since $p$ moves one by one, it points to $t + (x - t - 1) \mod C$. However, $q$ moves by two elements at each step, thus it points to $t + (2x - t - 1) \mod C$. Now observe that when $x$ is a multiple of $C$ (and not less than $t$), $t + (x - t - 1) \mod C = t + (2x - t - 1) \mod C$, hence both $p$ and $q$ point to the same element and our algorithm terminates. Clearly there is always a multiple of $C$ between $t$ and $t + C$ and thus if the list is corrupted, our algorithm terminate after at most $t + C = n - 1$ iterations.
10. Given a sequence of (not necessary positive) integers \(x_1, x_2, \ldots, x_n\), find a subsequence \(x_i, x_{i+1}, \ldots, x_j\) (of consecutive elements) such that the sum of the numbers in it is **even and maximum** over all subsequences of consecutive elements with an even sum of elements. For example if the sequence is 5, −1,4,−10,3,3,3, the maximum even consecutive subsequence is 5, −1,4. The running time of your algorithm should be in \(O(n)\).

We say a subsequence of consecutive elements is “even” if the sum of elements in it is even, otherwise we say it is “odd”. Define \(E_i\) as the maximum sum of even subsequences ending at \(i\). Similarly, define \(O_i\) as the maximum sum of odd subsequences ending at \(i\). Clearly the answer to the problem is \(\max_i E_i\). We only need to show how we can compute \(E_i\) s and \(O_i\) s in \(O(n)\). One can easily show that the following recursive formula holds, which would directly give us the desired algorithm:

\(E_i\) and \(O_i\) are easily computable. For \(i \geq 2\):

- If \(x_i\) is even \(\Rightarrow E_i = \max \{0, E_{i-1} + x_i\}\) and \(O_i = O_{i-1} + x_i\)
- If \(x_i\) is odd \(\Rightarrow E_i = \max \{0, O_{i-1} + x_i\}\) and \(O_i = E_{i-1} + x_i\)