1. Asymptotics (25pts)

Order the following nine functions in such a way that \( f_k = O(f_{k+1}) \). Make sure to replace \( O \) by \( \Theta \) whenever possible, like in the following example:
\[
n = O(n^2), \quad n^2 = \Theta(n^2 + \log n), \quad n^2 + \log n = O(n^3).
\]
Here are the functions you will need to arrange:

A: \( \log(n+1/n) \)
B: \( n \log n \)
C: \( \log\log n \)
D: \( 2^{n-\log n} \)
E: \( (\log n)^n \)
F: \( (5n + \log n/n)2^{(4+\log n)} \)
G: \( 3^{\log n-n} \)
H: \( (3 + \log n)! \)
I: \( n^3 + (\log n)^n \)

Solution: Let’s use the following notation: \( A \preceq B \) when \( f_A = O(f_B) \), and \( A \sim B \) when \( f_A = \Theta(f_B) \). We shall use \( A \prec B \) when \( A \preceq B \) but not \( A \sim B \).

Then the following ordering holds:
\[
G \prec C \prec A \prec B \prec F \prec H \prec D \prec E \sim I
\]

2. \( k \)-subsets (30pts)

The questions below will refer to the following piece of code that is also available on the web as a supplement.

```cpp
void print_subset(vector<unsigned>& s) {
    cout << "\{ ";
    for(int i=0; i<s.size(); ++i)
    {
        cout << s[i] << " ";
    }
    cout << "\}" << endl;
}

void rec_subset(vector<unsigned>& s, int n, int k) {
    if(n<k) {
        return;
    }
    if(k==0) {
        print_subset(s);
        return;
    }
}
```
The function `generate_subsets(int n, int k)` will print all the subsets of \{0, \ldots, n-1\} that contain \(k\) elements.

**Solution:** We shall prove the following statement:

**Lemma 1.** A single call to the function `generate_subsets(n, k)` \((0 \leq k \leq n)\) will print all the subsets of \{0, \ldots, n-1\} that contain \(k\) elements.

We prove this by induction on \(n\): Let \(S(n)\) be the statement of the theorem with some particular fixed non-negative integer \(n\). That is, \(S(n)\) is the statement for a single fixed \(n\) and all non-negative \(k\).

**Base case:** \(S(0)\) For \(n = 0\) and \(k = 0\), we return upon checking the condition of the second if statement, and `print_subset` is called with empty set subset trivially in the first zero positions. duh. For \(n = 0\) and \(k > 0\) the function returns upon checking the first if condition without ever calling `print_subset` function, just as it should since there are no nonempty subsets of an empty set.

**Induction step:** Suppose that the lemma’s statement holds for some \(n = n’\) (note that it would have to hold for all non-negative \(k\)). Let’s prove it for \(n = n’ + 1\). That is, let’s prove that for any non-negative \(k\) the `print_subset` function will be called once for every \(k\)-subset of \{0, \ldots, n’\}. Indeed, for \(k = 0\), there is a single empty 0-subset that will be called right away in the second if statement and then the function will return. So, the statement holds for \(k = 0, n = n’ + 1\).

For \(k > n\), the function returns without ever calling `print_subset` just as it should.

Now, consider the case for some \(k\) such that \(0 < k \leq n\). Denote by \(B(n’, k)\) the collection of all the \(k\)-subsets of \{0, \ldots, n’\}. It is easy to see that all the members of \(B(n’, k)\) can be split into two categories: the ones that contain \(n’\) and the ones that do not. Note that the second category is exactly what we denoted by \(B(n’ – 1, k)\). Also, note that all the subsets of the first category can be obtained by generating all the \(k – 1\)-subsets of
\{0, \ldots, n' - 1\} and adding \(n'\) as a member to all thus generated subsets. This is exactly what happens in the last three lines of \texttt{rec_subset} function.

Namely, the first recursive call \texttt{rec_subset(s, n-1, k-1)} will result in the function \texttt{print_subset} being called once for every \(k - 1\)-subset of \(\{0, \ldots, n' - 1\}\) with that subset represented by the first \(k - 1\) elements of the vector \(s\) (this follows from the induction assumption). Since we placed \(n'\) as the \(k\)-th element of the vector \(s\), we shall see \texttt{print_subset} being called once for every \(k\)-subset of \(\{0, \ldots, n'\}\) from the first category above with that subset being placed in the first \(k\) positions of \(s\).

The second recursive call will handle the second category of \(k\)-subsets.

Thus the statement \(S(n' + 1)\) has been proven. Thus the lemma is proven as well. Invoking the lemma for \(n\) proves the needed result.

**Important:** Suppose that we remove printing commands from the body of \texttt{print_subset(s)} function, so that a call to \texttt{print_subset(s)} takes constant time. The following questions B through F will assume that.

**B**(5pts) Let \(T(n, k)\) be the running time of a call to \texttt{rec_subset(s,n,k)} function. Find a recurrence relation for \(T(n, k)\). Consider all the cases satisfying \(0 \leq k \leq n + 1\).

**Solution:** Here is the recursion:

\[
T(n, k) = \begin{cases} 
K_1, & \text{for } k = 0 \\
K_2, & \text{for } k > n \\
K_3 + T(n-1, k-1) + T(n-1, k), & \text{otherwise}
\end{cases}
\]

**C**(5pts) Introduce a new variable \(T'(n, k) = T(n, k) + C_1\) and prove that it satisfies the following recurrence relation (there was a typo in equations below, now it’s fixed):

\[
T'(n, k) = \begin{cases} 
T'(n-1, k-1) + T'(n-1, k), & \text{when } 0 < k \leq n, \\
C_2, & \text{when } k = n + 1, \\
C_3, & \text{when } k = 0.
\end{cases}
\]

What choice of the constant \(C_1\) will make it work?

**Solution:** Add \(C_1 = K_3\) to both sides of the original recurrence equations to get

\[
T(n, k) + K_3 = \begin{cases} 
K_1 + K_3, & \text{for } k = 0 \\
K_2 + K_3, & \text{for } k > n \\
K_3 + T(n-1, k-1) + K_3 + T(n-1, k), & \text{otherwise}
\end{cases}
\]

The rest is obvious. \(C_1 = K_3, C_2 = K_2 + K_3, C_3 = K_1 + K_3\).
**D(5pts)** Let $C_4 = \max(C_2, C_3)$. Prove by induction that $T'(n, n) \leq C_4(n+1)$.

**Solution:** Base case is $n = 0$, for which we get $T'(0, 0) = C_3 \leq C_4$.

Suppose that $T'(n, n) \leq C_4(n+1)$. From recurrence we get $T'(n+1, n+1) = T'(n, n) + T'(n, n+1) = T'(n, n) + C_2 \leq C_4(n+1) + C_2 \leq C_4(n + 2)$, which is exactly what we need.

---

**E(5pts)** Prove by induction that $T'(n, k) \leq C_4 \binom{n+1}{k}$.

**Solution:** Let’s be careful here. The statement $M(n)$ will be: for all $k$ such that $0 \leq k \leq n + 1$ we have $T'(n, k) \leq C_4 \binom{n+1}{k}$.

Base case is $M(0)$, so that $n = 0$ for which we get $T'(0, 0) = C_3 \leq C_4 \binom{1}{0}$.

Induction step assumes that $M(n)$ is true. Let’s prove $M(n+1)$ based on that. For $k = 0$ we have $T'(n, 0) = C_3 \leq C_4 \binom{n+2}{0}$. Similarly, for $k = n+2$ we have $T'(n+1, n+2) = C_2 \leq C_4 \binom{n+2}{n+2}$ as required. Consider $0 < k < n + 2$ now. We have:

$$T'(n+1, k) = T'(n, k-1) + T'(n, k) \leq C_4 \binom{n+1}{k-1} + C_4 \binom{n+1}{k} = C_4 \binom{n+2}{k}$$

Here we used the inductive assumption and the binomial coefficients property. Thus, $M(n+1)$ is proven.

---

**F(5pts)** Prove that for $k \leq \lfloor n/2 \rfloor$ we have

$$T'(n, k) \leq 2C_4 \binom{n}{k}$$

**Solution:** We know that $T'(n, k) \leq C_4 \binom{n+1}{k}$.

But $\binom{n+1}{k} = \binom{n}{k} + \binom{n}{k-1}$. Moreover, $\binom{n}{k-1} \leq \binom{n}{k}$ when $k \leq \lfloor n/2 \rfloor$. It follows that

$$T'(n, k) \leq C_4 \binom{n+1}{k} = C_4 \left( \binom{n}{k} + \binom{n}{k-1} \right) \leq 2C_4 \binom{n}{k}.$$  

**Conclusion** Thus, we have proven that

$$T(n, k) \leq 2C_4 \binom{n}{k} - C_1 \leq 2C_4 \binom{n}{k},$$

that is the time per one generated $k$-subset is constant (when $k \leq \lfloor n/2 \rfloor$).

---

**EXTRA(10pts)** What happens when $\lfloor n/2 \rfloor < k < n$? Find an upper bound on the time per one generated $k$-subset. Is it $O(1)$? $O(n)$? $O(k)$?
Solution: We know that
\[ T'(n, k) \leq C_4 \binom{n + 1}{k}. \]

We can use the fact that
\[ \binom{n + 1}{k} = \frac{n + 1}{n + 1 - k} \binom{n}{k}. \]

The factor \((n + 1)/(n + 1 - k)\) is the upper bound asymptotics of runtime per single generated subset. We can say that \( (n + 1)/(n + 1 - k) = O(n) \).

3. Asymptotics (30pts)

A function \( t(n) \) is defined by recurrence relation:
\[
t(n) = \begin{cases} 
a, & \text{for } n = 1 \\ 4t(\lceil n/3 \rceil) + bn, & \text{for } n > 1 \end{cases}
\]

A.(15pts) Prove by induction that \( t(n) \) is an eventually non-decreasing function.

Solution: First of all, we will assume that \( a \) and \( b \) are positive constants throughout this exercise.

We’d like to prove that when \( n \leq n' \) then \( t(n) \leq t(n') \).

Let \( S(n) \) be the statement: for \( m \) and \( m' \) such that \( 0 < m \leq m' \leq n \) we have \( t(m) \leq t(m') \).

The base case: \( S(2) \) is trivial, since \( t(2) = 4a + 2b \geq a = t(1) \).

The inductive step: Assume \( S(n), n \geq 2. \) Let’s prove \( S(n + 1) \). It is enough to show that \( t(n + 1) \geq t(n) \).

\[
t(n + 1) - t(n) = 4(t(\lceil (n + 1)/3 \rceil)) - t(\lceil n/3 \rceil) + b,
\]

from the original recurrence, and \( 0 < \lceil n/3 \rceil \leq \lceil (n + 1)/3 \rceil \leq n \) since \( n \geq 2 \). It follows from the inductive assumption that \( t(\lceil (n + 1)/3 \rceil) \geq t(\lceil n/3 \rceil) \) so that

\[
t(n + 1) - t(n) \geq 0,
\]

which proves everything.

B.(15pts) Find the exact order of \( t(n) \) in the simplest possible form.

Solution: Using the master theorem we get \( t(n) = \Theta(n^{\log_3 4}). \)
Consider an algorithm $A$ that has average-case time complexity $O((n \log n)^2)$ and $\Omega(n \log n)$. For the following statements state whether it could or could not be true, and justify your answer.

A. $A$ has worst-case time complexity $O(n^2)$.
   **Solution:** Could be.
   Let’s cook up an example that will satisfy everything: consider an algorithm whose running time is always quadratic so that $t(n) = \Theta(n^2)$ for any instance of size $n$. Such algorithms do exist.
   Then its average and worst running time will be $\Theta(n^2)$. But surely, $\Theta(n^2) \subset O(n^2)$ and $\Theta(n^2) \subset O((n \log n)^2)$ and $\Theta(n^2) \subset \Omega(n \log n)$.

B. $A$ has worst-case time complexity $\Theta(n)$.
   **Solution:** Could not be, because the average case runtime cannot be slower than the worst case runtime thus we would have to have $t_{\text{average}}(n) = O(n)$. But then $O(n) \cap \Omega(n \log n)$ is an empty set, so there are no such algorithms.

C. $A$ has average-case time complexity $\Theta(n^2)$.
   **Solution:** Could be.
   In fact, the example from the part A works again.