1 Problems

1. Recurrences: Solve the following recurrences exactly or asymptotically. You may assume any convenient form for \( n \).

(a)

\[
\begin{align*}
T(1) &= 1 \\
T(n) &= T(\sqrt[3]{n}) + 1, \ n > 1
\end{align*}
\]

(b)

\[
\begin{align*}
T(1) &= 0 \\
T(n) &= 4T(\frac{n}{2}) + n^2 \cdot \log n, \ n > 1
\end{align*}
\]

Solution:

(a) Put \( n = 3^k \). Accordingly, the recurrence can be restated as:

\[
\begin{align*}
T(3^0) &= 1 \\
T(3^k) &= T(3^{\frac{k}{3}}) + 1, \ k > 0
\end{align*}
\]

Let \( G(k) \) denote \( T(3^k) \). Accordingly, the above recurrence can be represented as:

\[
\begin{align*}
G(0) &= 1 \\
G(k) &= G(\frac{k}{3}) + 1, \ k > 0
\end{align*}
\]

Using one of the many techniques discussed in class (expansion, induction, the Master Theorem), it is easily seen that \( G(k) = \log_3 n \), from which it follows that \( T(n) = \log_3 \log_3 n \).

(b) We use the Master Theorem to solve this recurrence. As per the pattern discussed in class, \( a = 4 \), \( b = 2 \) and \( f(n) = n^2 \log n \). It is clear that \( f(n) \in \Theta(n^{\log_2 4} \log \frac{1}{n}) \), from which it follows that \( T(n) \in \Theta(n^{\log_2 4} \log \frac{1}{n}) \).

\( \square \)

2. Binary Trees: Let \( T \) denote a proper binary tree with \( n \) internal nodes. We define \( E(T) \) to be the sum of the depths of all the external nodes of \( T \); likewise, \( I(T) \) is defined to be the sum of the depths of all the internal nodes of \( T \). Prove that \( E(T) = I(T) + 2 \cdot n \).

Solution: We use induction on the number of internal nodes in \( T \).
Base case: $n = 1$. In this case, $T$ consists of a root node with a left child and right child node. The root node is the only internal node and hence $I(T)$ is 0; its two children are the only external nodes and hence $E(T)$ is $1 + 1 = 2$. Since $E(T) = I(T) + 2 \cdot 1$, the conjecture is proven in the base case.

Assume that if $T$ is a proper binary tree with $i$ internal nodes, where $i \leq k$ then $E(T) = I(T) + 2 \cdot i$.

Now consider a proper binary tree $T$ having exactly $k + 1$ internal nodes. Let $h$ denote the height of this tree. Since $T$ is proper, there are at least two external nodes, which are children of the same internal node. Splice out these external nodes to get a new tree proper binary tree $T'$ having $k$ internal nodes (since a node that was internal in $T$ has now become external). As per the inductive hypothesis, we must have $E(T') = I(T') + 2 \cdot k$.

Observe that in $T'$ two external nodes at depth $h$ in $T$ have been removed and one node which was internal in $T$ at depth $h - 1$ has been added; hence, $E(T') = E(T) - 2 \cdot h + (h - 1)$.

Likewise, a node which was internal in $T$ at depth $h - 1$ is now external in $T'$ and hence $I(T') = I(T) - (h - 1)$.

We thus have,

$$
E(T') - I(T') = E(T) - I(T) - 2 \cdot h + (h - 1) + (h - 1)
$$

$$
\Rightarrow E(T) - I(T) = E(T') - I(T') + 2
$$

$$
\Rightarrow E(T) - I(T) = 2 \cdot k + 2
$$

$$
\Rightarrow E(T) = I(T) + 2 \cdot (k + 1)
$$

Thus, using the second principle of mathematical induction we can conclude that the conjecture is true for all proper binary trees regardless of the number of internal nodes. $\square$

3. Greedy: Assume that you are given a set $S$ of $n$ activities $\{a_1, a_2, \ldots, a_n\}$. Associated with activity $a_i$ are its start time $s_i$ and finish time $f_i$; if activity $a_i$ is selected then it must start at $s_i$ and finish before $f_i$. Two activities $a_i$ and $a_j$ are compatible, if $s_i \geq f_j$ or $s_j \geq f_i$; otherwise, they are conflicting. Design an algorithm that outputs the largest set of compatible activities.

Solution:

Algorithm 1.1 represents a greedy approach to output the maximum number of mutually compatible activities.

```
Function MAX-ACTIVITY-SELECT(S)
1. Let $R$ denote a subset of mutually compatible activities.
2. Set $R = \phi$.
3. Order the activities by their finish times so that $f_1 \leq f_2 \leq \ldots \leq f_n$.
4. for $i = 1$ to $n$ do
5.    if (activity $a_i$ is compatible with the activities already in $R$) then
6.       $R = R \cup \{a_i\}$.
7.    end if
8. end for
9. return($R$)
```

Algorithm 1.1: Greedy algorithm for maximum compatible activity subset

Assume that Algorithm 1.1 is not optimal and there exists another algorithm, say $A'$, which produces a solution $R'$, such that $|R'| > |R|$.

Claim 1.1 If $a_1 \notin R'$, then $a_1$ can always be made part of $R'$, without decreasing the number of activities in $R'$.
4. Sorting: We observe that the height of a binary tree (or any tree, for that matter) is minimized, when the tree is approximately equal portions. Letting $a_i, a_j \in R'$ denote two jobs that conflict with $a_1$. Since $f_1 \leq f_i, f_j$, we must have $s_i, s_j \leq f_1$. However, this means that both $a_i$ and $a_j$ straddle $a_1$, which forces them to conflict with each other. It follows that $a_i$ and $a_j$ conflict with each other as well! Thus, there can be at most one activity in $R'$ that conflicts with $a_1$; replacing that activity with $a_1$ preserves the cardinality of $R'$. □

Let $k$ be the smallest index such that $a_k \in R$ and $a_k \notin R'$. Thrust $a_k$ into $R'$; using the same argument as before, $a_k$ can conflict with at most one activity in $R'$; replacing that activity with $a_k$ does not affect the cardinality of $R'$, but brings it one activity closer to $R$.

Working in this fashion, we can gradually transform $R'$ such that it includes all the activities in $R$, without decreasing its cardinality. Once this transformation has been carried out, we claim that there are no additional activities in $R$. Assume that there exists an activity, say $a_p \in R'$, such that $a_p \notin R$. Let $a_q$ denote the finish time of the last activity that was added to $R$.

We consider two possibilities:

(a) $s_p \geq f_q$ - In this case, the greedy algorithm would have considered $a_p$ and added it to $R$, since it does not conflict with any of the jobs already in $R$.

(b) $s_p < f_q$ - If $f_p \geq f_q$, then $a_p$ conflicts with $a_q$ and hence cannot be part of $R'$. If $f_p < f_q$, then the greedy algorithm would have considered $a_p$ before $a_q$; the fact that $a_p \notin R$ implies that it conflicted with some of the activities already chosen in $R$!

We have thus established that any optimal solution can be transformed into the greedy one, i.e., the greedy approach does produce the optimal solution.

□

4. Sorting: Analogous to the notion of worst-case running time for an algorithm, is the notion of best-case running time, which is the minimum amount of time that an algorithm needs to accomplish its task. Argue that the best-case running time of Quicksort (in terms of element-to-element comparisons) is $\Omega(n \cdot \log n)$. (It is interesting to note that the best-case running time of Insertion sort is $O(n)$.)

Solution: We focus on the computation tree of Quicksort; recall that we used the computation tree to demonstrate that the expected running time of Quicksort is $O(n \cdot \log n)$. Indeed the running time of the Quicksort algorithm is $O(n \times h)$, where $h$ is the height of the computation tree.

We observe that the height of a binary tree (or any tree, for that matter) is minimized, when the tree is balanced, i.e., external nodes occur only at level $h$ and possibly level $h - 1$.

Accordingly, for the best-case performance of Quicksort, the partition procedure must divide the array into approximately equal portions.

Letting $T(n)$ denote the best-case running time of Quicksort on an array of $n$ elements, we get,

$$
T(1) = 0 \\
T(n) = 2 \cdot T\left(\frac{n}{2}\right) + (n - 1)
$$

We argue using induction, that $T(n) \geq G(n) = \frac{1}{10}n \cdot \log n - n$.

Since $T(1) \geq G(1)$, the base case is proven.

Assume that $T(n) \geq G(n)$ for all $n \leq k$.

Observe that

$$
T(k + 1) = 2 \cdot T\left(\frac{k}{2}\right) + k \text{ as per definition} \\
\geq 2 \cdot \left[\frac{1}{10} \cdot \log \frac{k}{2} - \frac{k}{2}\right] + k \text{ as per inductive hypothesis}
$$
We then observe that,
\[
\frac{k}{10} \log k - \frac{k}{10} \geq \frac{1}{10} (k + 1) \log (k + 1) - (k + 1)
\]
\[
\Rightarrow k \log k - k \geq (k + 1) \log (k + 1) - (k + 1)
\]
\[
\Rightarrow k \log k - k \geq (k + 1) \log (k + 1) - 10(k + 1)
\]
\[
\Rightarrow 9k + 10 \geq (k + 1) \log (k + 1) - k \log k
\]
But \((k + 1) \log (k + 1) - k \log k \leq (k + 1) \log (k + 1) - k \log k = (k + 1) + \log k\). Hence, \(9k + 10 \geq (k + 1) \log (k + 1) - k \log k\), as long as \(8k + 9 \geq \log k\), which is true for all \(k\).

We have thus shown that \(T(n) \in \Omega(G(n))\); it is not hard to show that \(G(n) \in \Omega(n \cdot \log n)\); we can thus conclude that \(T(n) \in \Omega(n \cdot \log n)\).

5. **Divide and Conquer**: Design a Divide-And-Conquer algorithm to discover both the maximum and minimum of an array \(A\) of \(n\) elements using at most \(\frac{3}{2} n^2\) element-to-element comparisons. Formally prove that your algorithm makes at most \(\frac{3}{2} n^2\) element-to-element comparisons.

**Solution**: We assume that there are at least 2 elements in the array; otherwise, the problem is ill-defined. Further, we assume that the number of elements in \(A\) is an exact power of 2, in order to simplify the exposition.

Algorithm 1.2 represents a Divide-And-Conquer approach for computing both the minimum and maximum elements of the input array.

```plaintext
Function MAXMIN(A, low, high)
1: if \((high - low + 1 = 2)\) then
2: if \((A[low] < A[high])\) then
3: \(max = A[high]; min = A[low]\).
4: return((max, min)).
5: else
6: \(max = A[low]; min = A[high]\).
7: return((max, min)).
8: end if
9: else
10: \(mid = \frac{low+high}{2}\).
11: \((max_l, min_l) = \text{MAXMIN}(A, low, mid)\).
12: \((max_r, min_r) = \text{MAXMIN}(A, mid + 1, high)\).
13: end if
14: Set \(max\) to the larger of \(max_l\) and \(max_r\); likewise, set \(min\) to the smaller of \(min_l\) and \(min_r\).
15: return((max, min)).

Algorithm 1.2: Divide and Conquer algorithm for computing maximum and minimum of an array
```

Let \(T(n)\) denote the number of element-to-element comparisons carried out by Algorithm 1.2. We have,
\[
T(2) = 1
\]
\[
T(n) = 2 \cdot T\left(\frac{n}{2}\right) + 2, \quad n > 2.
\]
Substituting \( n = 2^k \) and using the expansion method discussed in class, it is straightforward to see that \( T(n) \leq \frac{3}{2}n \).

\[
T(2^k) = 2 \cdot T(2^{k-1}) + 2
= 2 \cdot [2 \cdot T(2^{k-2}) + 2] + 2
= 2^2 \cdot T(2^{k-2}) + 2^2 + 2
= 2^2 \cdot [2 \cdot T(2^{k-3}) + 2] + 2^2 + 2
= 2^3 \cdot T(2^{k-3}) + 2^3 + 2^2 + 2
= \vdots
= 2^{k-1} \cdot T(2^{k-(k-1)}) + 2^{k-1} + 2^{k-2} + \ldots + 2^2 + 2
\]

But \( T(2^{k-(k-1)}) = T(2^1) = 1 \) and hence, \( T(2^k) = \sum_{j=1}^{k-1} 2^j + 2^{k-1} \).

Note that

\[
\sum_{j=1}^{k-1} 2^j = 2 \sum_{j=0}^{k-2} 2^j
= 2 \cdot \frac{[2^0 \cdot (2^{k-1} - 1)]}{2 - 1} \text{ sum of a geometric progression}
= 2^k - 2
\]

It follows that

\[
T(n) = T(2^k)
= 2^{k-1} + 2^k - 2
= \frac{1}{2} 2^k + 2^k - 2
= \frac{3}{2} 2^k - 2
= \frac{3n}{2} - 2
\]

\( \square \)