1. Using a decision tree method to prove that the problem of searching for a key in a given list of \( n \) elements has time complexity lower bound \( \Omega(\log_2 n) \) for the comparison-based models, regardless the given list is sorted or not.

**Answer:**

First, we should assume computation is 'comparison-based'.

Second, decision tree method has to be used.

**Proof Sketch:**

Regardless the input list is sorted or not, there are \( n + 1 \) possible outcomes from any algorithm that does key search on a list of \( n \) elements. In particular, the algorithm outputs one of the \( n \) positions or 0 that indicates the non-existence of the key in the list.

So the decision tree model for all such search algorithms has at least \( n + 1 \) leaves, with at least one leaf to accommodate one of the \( n + 1 \) outcomes. Because the longest path in the shortest binary tree of \( n + 1 \) leaves is \( \Omega(\log_2(n+1)) \), which is \( \Omega(\log_2 n) \), the decision tree model runs in time \( \Omega(\log_2 n) \). So all algorithms have to take \( \Omega(\log_2 n) \) time to do the search correctly.
2. Consider the problem SortBinaryBits that sorts a list of binary bits (where each element \(x_i\) in the list is a single binary bit, either 0 or 1).

(a) Design an algorithm (comparison-based only) for SortBinaryBits where the input is \(\{x_1, \ldots, x_n\}\), with \(x_i \in \{0, 1\}\). Derive a time complexity upper bound for your algorithm.

(b) (Graduate students only, bonus for undergraduate students) Prove a time complexity lower bound for the problem SortBinaryBits using a decision tree based method. Consider the upper bound you derived for your algorithm in part (1) and explain if your algorithm is the optimal or not.

**Note:** you need to put in effort in algorithm design and lower bound proof. A trivial algorithm or lower bound would not deserve any point.

**Answers:**

(a) If you "ignored" the requirement that algorithms have to be "comparison-based", which disallows a comparison directly to value 1 or 0, such as if \(x_i > 0\), you can only earn at most half credit this question.

Now an algorithm idea. But it is not the only idea. The algorithm repeatedly compares the two elements at the head and tail of the list, and moves the head and tail indexes accordingly. Using a new list to store sorted result – i.e., larger element is placed at the tail and smaller at the head of the new list, moving the head and tail indexes of the new list accordingly. The algorithm ends when the head and tail indexes coincide in the input list. Because there are \(n\) elements, so at most \(n\) comparisons are needed.

(b) (Graduate students only, bonus for undergraduates)

We prove the lower bound \(\Omega(\log_2 n)\). The idea is similar to the one used to answer Q1. Use the decision tree model and estimate the minimum number of outcomes from any algorithm for this sorting problem. The sorting outcomes for an algorithm has to accommodate are all potential outputs:

\[(0, \ldots, 0, 0), (0, \ldots, 1, 0), \ldots, (1, \ldots, 1, 1)\]

which are \(n + 1\) in number. Using the same argument for Q1, the sorting problem has \(\Omega(\log_2 n)\).
IF you estimates $2^n$ many possible outcomes based on the same number of inputs, you may derive $\Omega(\log_2 2^n) = \Omega(n)$ as a lower bound. You should get at least half of the credit for this solution as long as usage of the decision tree method is correct.
3. Consider the following sorting problem where the elements in the input list \( \{x_1, \ldots, x_n\} \) always have only \( k \) distinct values, for \( k < n \). Prove that this sorting problem can be solved in worst case time \( O(n \log_2 k) \).

**Hint:** consider a quick sort type of algorithm but with a good procedure that allows some “balanced” partition. Also consider the fact that there are only \( k \) distinct values in the given list.

**Answers:**

**Proof idea:**

Using the hint, we use the Select function to find the median \( x \) of the input list of size \( n \) as the pivot in linear time \( O(n) \). The pivot is used to partition the list into two sublists of size \( \frac{n}{2} \) in each. Then for both sublists, swap all elements of the same value as the median \( x \) to become the nearest neighbors of \( x \). Then the list is reduced to two sublists by taking out elements of the same value as \( x \). We recursively apply the process on the two reduced sublists until the sublists contains a single element or empty.

This algorithm has the following recurrence for its time complexity \( T(n) \)

\[
T(n) \leq 2T\left(\frac{n}{2}\right) + cn, \quad \text{with } T(1) = a, \ T(0) = b
\]

for some constants \( a, b, c \). Using the recursive tree method, it is easy to see that \( T(n) = O(n \log_2 n) \).

But because there are \( k \) distinct values for the elements, there are at most \( k \) different pivot elements and the unfolding process will only going through at most \( \log_2 k \), instead of \( \log_2 n \) rounds. So using the recursive tree method we can prove the upper bound \( O(n \log_2 k) \).
4. Consider the following graph-theoretic problem:

**Minimum Vertex Cover** problem

**Input:** graph $G = (V, E)$;

**Output:** a subset $C \subseteq V$ such that

1. $\forall u, v \in V,$ if $(u, v) \in E$, then $u \in C$ or $v \in C$
2. $|C|$ is the minimum.

Intuitively, the found cover $C$ should cover every edge in $G$ on at least one of its two end vertices. The size of $C$ should be the smallest among all such covers.

Use the idea of search tree, design an exhaustive algorithm for the Minimum Vertex Cover problem which it has the running time $O(\gamma^n)$ for some $\gamma < 2$. The following are the requirements.

(a) Design an exhaustive search algorithm that has the time recurrence

$$T(n) \leq T(n - 1) + T(n - 2) + cn^2$$

for some constant $c > 0$;

(b) (Graduates only, bonus for undergraduates) Carefully design your algorithm so that it yields the recurrence

$$T(n) \leq T(n - 1) + T(n - 3) + cn^2$$

**Answers:**

(a) Such an exhaustive search algorithm would work with the following steps recursively:

Preprocess the input graph $G$ so that all isolated vertices (i.e., those with degree $= 0$) are removed. Assume the resulted graph has $n$ vertices.

Let $v$ be an arbitrary vertex on the preprocessed graph. Because degree of $v$ is $\geq 1$, $v$ shares an edge with at least one other vertex, which is assumed to be $u$. Therefore, edge $(u, v)$ is in the graph. Then only two options regarding covering edge $(u, v)$ can happen.

1. $v$ is selected into the cover set $C$. In this case, the algorithm removes $v$ from the graph and recursively applies on the reduced graph (of size $n - 1$) after $v$ is removed.
(2). $v$ is not selected into the cover set $C$. In this case, the algorithm has to select $u$ into the cover set $C$ and recursively applies on the reduced graph (of size $n - 2$) after both $u$ and $v$ are removed.

Recursion stops when the graph has no edges in it.

The running time of the algorithm $T(n)$ can be defined with recurrence:

$$T(n) \leq T(n - 1) + T(n - 2) + cn^2$$

where $cn^2$ is the time sufficient for preprocessing the graph to remove vertices of degree 0 and for neighborhood checking.

(b) (Graduates only, bonus for undergraduates)

First we observe that the input graph can be further preprocessed to remove all vertices of degree = 1 before branching starts using the following greedy strategy.

If $v$ has degree 1, let $u$ be its only neighbor. Then the algorithm is “better off” by selecting $u$, instead of $v$, to the cover set $C$. (Why?)

Note that this step does not involve recursion or branching and can be done in time $cn^2$. The strategy is repeated applied on the graph until all remaining vertices have degree $\geq 2$.

On the preprocessed graph, assumed to have size $n$, the algorithm examines an arbitrary vertex $v$. Note that $v$ has at least 2 neighbors. It branches out to handle two cases:

1. select $v$ to $C$; resulting in a modified graph of size $n - 1$ (i.e., $v$ is removed); the algorithm recursively applies on to the modified graph;

2. $v$ is not selected but all of its neighbors have to be selected, resulting in a modified graph of size $\leq n - 3$ (i.e., $v$ and all its neighbors are removed); the algorithm recursively applies on to the modified graph;

So we have the recurrence for the time complexity

$$T(n) \leq T(n - 1) + T(n - 3) + cn^2$$
5. A typical dynamic programming algorithm computes the optimal solution. For example, the Assembly Line Scheduling problem is to compute the fastest path through the factory. If not only the fastest path but also the second fastest path are to be found, the dynamic programming algorithm ASSEMBLYLINE can be modified to accommodate this new goal. Discuss how you may revise the algorithm ASSEMBLYLINE for the purpose of finding both the fastest and second fastest paths through the factory.

(a) Show the details of how you would modify the algorithm to satisfy the new requirements.

(b) (Graduates only, bonus for undergraduates) Argue that the idea can be applied to finding top $k$ fastest paths through the factory, for any $k \geq 2$. What will be the time complexity of such an algorithm to find top $k$ fastest paths (in terms of both $n$ and $k$)?

**Answers:**

(a) Only discussion of an idea is needed, not formal algorithm.

Assume the fastest time function is $f_i(j)$ for $i = 1, 2$ and $j = 1, \ldots, n$. We need to add the second fastest time function $s_i(j)$ for $i = 1, 2$ and $j = 1, \ldots, n$.

The algorithm uses the same type of DP but add the step to compute the second fastest time function. That is, for $i = 1, 2$, when the $f_i(j)$ is computed from $f_1(j - 1)$ and $f_2(j - 1)$, i.e.,

\[
 f_1(j) = a_{1,j} + \min \begin{cases} 
 f_1(j - 1) \\
 f_2(j - 1) + t_{2,j-1} 
 \end{cases}
\]

\[
 f_2(j) = a_{2,j} + \min \begin{cases} 
 f_2(j - 1) \\
 f_1(j - 1) + t_{1,j-1} 
 \end{cases}
\]

Then

\[
 s_1(j) = a_{1,j} + \min \begin{cases} 
 s_1(j - 1) \\
 s_2(j - 1) + t_{2,j-1} \\
 \max \begin{cases} 
 f_1(j - 1) \\
 f_2(j - 1) + t_{2,j-1} 
 \end{cases} 
 \end{cases}
\]

Likewise, $s_2(j)$ can be recursively defined.
Table filling needs to store the pointer to where $s_1(j)$ and $s_2(j)$ are computed from.

The traceback process is similar to the case of using $f$.

(b) (Graduates only, bonus for undergraduates)
Define function $top_t(i, j)$, $t = 1, 2, \ldots, k$ for each station $(i, j)$, with $top_1$ be the fastest time, $\ldots$, $top_k$ being the $k$th fastest time. The algorithm computes all these $k$ functions, similar to the way that $f$ and $s$ are computed in (a).

The time complexity would be $O(n \times k \log k)$ because each station has to determine top $k$ values from $2k$ values and a sorting algorithm like MERGE SORT, which takes time $O(k \log k)$, can be used.
6. The casino dice decoding problem was introduced in the classroom. Complete all the 4 steps of the dynamic programming solution to this problem:

(1) Analyze the problem to show that the problem demonstrates the property of optimal substructure and property of overlapping subproblems.

(2) Define the numerical objective function and derive a recurrence for the objective function.

**Answers:**

The answers are essentially given in the lecture notes/class. Recurrence needs to include base case(s).
7. Following the questions in Q6,

(3) Give a pseudo code for your algorithm that computes the objective function.

(4) Give a traceback algorithm to recover the sequence dice used.

**Answers:**

Both iterative algorithm to compute the maximum probability and the traceback algorithm look much like those for the Assembly Line problem.