CS 256: Algorithm Design and Analysis

Problem Set 5 (due 04/09/2025 9:59pm)

Instructor: Sam McCauley

Problem 1. The following word problem is *related* to divide and conquer, but the final solution is not necessarily a divide and conquer algorithm like the ones we've seen. This question is more about divide and conquer intuition.

One of your Amazon packages has been lost. You suspect that it's in a specific warehouse, so you take your tracking number and drive over to the warehouse to find your package.

Unfortunately, on arrival, you find that there are no people at the warehouse, only a fairly unhelpful robot. The robot will only answer questions of the form: "what is the ith largest tracking number in the warehouse?" The robot will either give you the ith largest tracking number, or answer that there are fewer than i packages in the warehouse (despite its limited capabilities, the robot always answers these questions correctly).

(a) Assume for a second that you know that there are n packages in the warehouse. Describe in 1-2 sentences how to determine if your package is in the warehouse (in other words, if one of the packages in the warehouse has your tracking number) by asking the robot $O(\log n)$ questions.

Solution.

□

(b) In reality, you have no bound on the number of packages in the warehouse n.[1] Describe how you can, nonetheless, determine if your package is in the warehouse by asking the robot $O(\log n)$ questions. Please **briefly justify the running time.**

Hint: If you obtain a bound on n via your questions, you can essentially use your answer from part (a).

Solution.

□

[1]To be clear, due to the efficiency of Amazon, assume you have *no* upper bound on the number of packages in the warehouse; it could potentially be larger than any number you come up with. The only way you can bound the number is via questions to the robot.

Problem 2. (Erickson 3.3) Suppose you are given a 1-indexed array $A[1, \ldots, n]$ of numbers, which may be positive, negative, or zero, and which are not necessarily integers.

Describe and analyze an algorithm that finds the largest sum of elements in a contiguous subarray $A[i \ldots j]$. So if the array consists of only positive numbers, then the largest sum of contiguous elements would just be the sum of all the elements. If the array is only made up of negative numbers, then the largest subarray is made up of the number closest to 0. (To simplify things, we are not allowing empty subarrays).

As an example consider an array $A = [-6, 12, -7, 0, 14, -7, 5]$, then the contiguous subarray with the largest sum is $[12, -7, 0, 14]$ with the sum of 19. If the subarray is $B = [-4, -1]$, then the contiguous subarray with the largest sum is $[-1]$ with the sum of -1.

Note. This problem has been a standard computer science interview question since at least the mid-1980s and can be solved via a variety of approaches. We will use a dynamic-programming approach for this question.

This dynamic programming problem is similar to the longest increasing subsequence problem we discussed in class.

- **Subproblem definition:** Let $M(i)$ denote the optimal (largest) subarray sum ending at index i of the array (and including $A[i]$).
- **Recurrence:**

  Solution. Students, fill in the recurrence □

- **Base Case:**

  Solution. Students, fill in the base case □

- **Final solution:** Given by taking the maximum over all the $M(j)$'s that is, $\max_{1 \leq j \leq n} M(j)$.
- **Memoization structure:** We can store the values of $M[1, \ldots, n]$ in a linear size array.
- **Evaluation order:** the evaluation of the dynamic program proceeds left to right, starting from $j = 1$ and going up to $j = n$.
- **Time and space analysis:**

  Solution. Students, fill in the time and space analysis □

Problem 3. (Kleinberg Tardos 6.1) Let $G = (V, E)$ be an undirected graph with n nodes. A subset of the nodes is called an *independent set* if no two of them are joined by an edge. Finding large independent sets is difficult in general; but here we'll see that it can be done efficiently if the graph is simple enough.

Call a graph $G = (V, E)$ a path if its nodes can be written as $v_1, v_2, ..., v_n$, with an edge between v_i and v_{i+1}, for $i \in \{1, 2, \ldots, n-1\}$. With each node v_i, we associate a positive integer weight w_i. The problem we want to solve is the following: Find an independent set in a path G whose total weight is as large as possible.

For example, the maximum weight of an independent set in the path in Figure 1 is 14.

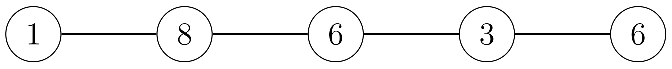

Figure 1: The maximum weight of an independent set is 14 in this example.

(a) Give a counterexample (a simple counterexample suffices!) to show that the following "pick the heaviest weight" greedy algorithm does not always work. You should say what the greedy algorithm returns, and what the correct answer is.

- Start with $S = \emptyset$
- While some node remains in G
  - Pick a node v_i of maximum weight and v_i to S
  - Delete v_i and its neighbors from G
- Return S

Solution. Students, give your counterexample here

□

(b) Give a dynamic-programming algorithm that takes an n-node path G with weights and returns the *value* of the independent set of maximum total weight.

- **Subproblem definition:**

  Solution. Students, fill in the subproblem definition □

- **Recurrence:**

  Solution. Students, fill in the recurrence □

- **Base Case:**

  Solution. Students, fill in the base case □

- **Final solution:**

  Solution. Students, fill in the final solution □

- **Memoization structure:** We can store the values in a linear size array.

- **Evaluation order:** the evaluation of the dynamic program proceeds left to right, starting from $j = 1$ and going up to $j = n$.
- **Time and space analysis:**

  Solution. Students, fill in the time and space analysis $\square$

Problem 4. (**Extra Credit (10pts)**) We have seen that the solution to the recurrence $T(n) = T(n/2) + 1$, $T(1) = 1$ is $T(n) = O(\log n)$.

We have also seen that the solution to the recurrence $T(n) = T(\sqrt{n}) + 1$, $T(1) = 1$ is $T(n) = O(\log \log n)$.

Give a recurrence of the above form that solves to $T(n) = O(\log \log \log n)$. (That is to say: your recurrence should be of the form $T(n) = T(f(n)) + 1$, $T(1) = 1$, for some $f(n)$).

Hint. The answer is not $T(n) = T(\log n) + 1$. As we saw during our Union-Find discussion, this solves to $T(n) = O(\log^* n)$.

Solution. $\square$