CSCI 2300 — Introduction to Algorithms
Homework 5 (document version 1.0) — DUE April 7, 2020
Dynamic Programming

- This homework is due by 11:59PM on the due date above and must be submitted in Submitty as a single PDF called `upload.pdf`.
- Just like the labs, please avoid using Google to find suggestions or solutions. You are allowed to use the textbooks and class notes as references. We strongly encourage you to use the office hours, labs/recitations, and Discussion Forum to successfully complete this assignment.
- Also, you are allowed to consult with your classmates, collaborating in line with the policies set forth in the syllabus. You must write up your own separate solution; do not copy-and-paste anything.
- Remember the goal is to use your own brain to work these problems out, which will help you develop the skills to do well on the exams and, more importantly, become a substantially better computer scientist.
- For your solutions and proofs, please write clearly and concisely, and use rigorous formal arguments and pseudocode that mimics the style used in the Dasgupta textbook. Be consistent.
- In general, if you are asked to provide an algorithm, the best thing to do is to provide a clear description in English (e.g., "Use BFS, but with the following change..." or "Run BFS two times as follows...").
- Be as concise as you can in your answers. Long answers are difficult to grade.

Instructions for all Problems

All of the homework problems are dynamic programming problems. Therefore, for each problem, be sure to explicitly state the following:

1. What your subproblems are and what they mean, e.g., "$L[i]$ is the length of the shortest path ending at i" is fine, whereas "$L[i]$ is optimum at i" is insufficient.
2. Your recurrence and algorithm, for which a few lines of pseudocode should suffice.
3. One or two sentences explaining why your recurrence is correct (no need for a formal proof).
4. The runtime of your algorithm.

Warm-Up Problems (do not hand in for grading)

Work out the problems below as practice problems before going on to the next section. Note that warm-up problems might sometimes show up on an exam!

1. DPV Problem 6.1 – **SOLUTIONS:**

   Subproblems: Define array of subproblems $D(i)$ for $0 \leq i \leq n$ with $D(i)$ being the largest sum of a (possibly empty) contiguous subesequence ending exactly at position i.

   Algorithm and Recursion: The algorithm initializes $D(0) = 0$ and updates each $D(i)$ in ascending order according to:

   $$D(i) = \max\{0, D(i-1) + a_i\}$$

   The largest sum is then given by the maximum element in array D at index k, with the contiguous subsequence of maximum sum terminating at k. Its beginning will be at the first index $j \leq k$ such that $D(j-1) = 0$, as this implies that extending the sequence before j will only decrease its sum.

   Correctness: The contiguous subsequence of largest sum ending at i will either be empty or contain a_i. If empty, the value of the sum will be 0; otherwise, the value of the sum will be the sum of a_i and the best sum we can get ending at $i-1$, i.e., $D(i-1) + a_1$. Since we are looking for the largest sum, $D(i)$ will be the maximum of these two possibilities.

   Runtime: The runtime for this algorithm is $O(n)$ since we have n subproblems and the solution of each can be computed in constant time. Further, the identification of the optimal subsequence only requires a single $O(n)$ time pass through array D.

2. DPV Problem 6.2 – **SOLUTIONS:**

   Subproblems: Define subproblem $D(i)$ for $0 \leq i \leq n$ with $D(i)$ being the mininum total penalty to get to hotel i.

   Algorithm and Recursion: The algorithm initializes $D(0) = 0$ and computes the remaining $D(i)$ values in ascending order using:

   $$D(i) = \min_{0 \leq j < i} \{(200 - a_j)^2 + D(j)\}$$

   To recover the optimal itinerary, we keep track of a maximizing j for each $D(i)$ and use this information to backtrack from $D(n)$.

   Correctness: To solve $D(i)$, we consider each possible hotel j we can stay at on the night before reaching hotel i; for each of these possibilities, the minimum penalty to reach i is the sum of the cost of a one-day trip from j to i and the minimum penalty necessary to reach j. Since we are interested in the minimum penalty to reach i, we take the minimum of these values over all possible j values.

   Runtime: The runtime is $O(n^2)$ since we have n subproblems with each subproblem taking $O(n)$ to solve, as we need to compute the minimum of $O(n)$ values. Further, backtracking takes $O(n)$.

3. DPV Problem 6.3 – **SOLUTIONS:**

   Subproblems: Define subproblem $D(i)$ to be the maximum profit that Yickdonald's can obtain from locations 1 to i.

   Algorithm and Recursion: The algorithm initializes $D(0) = 0$ and computes the remaining $D(i)$ values using:

   $$D(i) = \max\{D(i-1), p_i + D(i^*)\}$$

   Here, i^* is the largest index j such that $m_j \leq m_i - k$, i.e., the first location preceding i and at least k miles away from it.

   Correctness: If location i is not used, then maximum profit $D(i)$ simply equals $D(i-1)$; otherwise, if location i is used, the best we can hope for is the sum of p_i and the maximum profit from the remaining locations we are still allowed to open before i, i.e., the $D(i^*)$ term.

   Runtime: This algorithm solves n subproblems; each subproblem requires finding index i^*, which can be done in $O(\log n)$ time using binary search on the ordered list of locations, and computing a maximum of two values, which can be done in constant time. Therefore, the runtime is $O(n \log n)$.

Required Problems (hand in for grading)

Work out the problems below and submit for grading before the deadline.

1. DPV Problem 6.4

   SOLUTIONS AND GRADING GUIDELINES:

   $\Longrightarrow$ 2 points for clearly described and correct subproblems
   $\Longrightarrow$ 2 points for clearly described and correct algorithm and recurrence
   $\Longrightarrow$ 2 points for showing correctness (no need for formal proof)
   $\Longrightarrow$ 2 points for correct runtime
   $\Longrightarrow$ 2 points for part (b) correctness

   (a) *Subproblems:* Define array of subproblems $S(i)$ for $0 \leq i \leq n$, where $S(i) = 1$ if $S[1 \cdots i]$ is a sequence of valid words and $S(i) = 0$ otherwise.

   Algorithm and Recursion: Initialize $S(0) = 1$. Update $S(i)$ in ascending order according to:

   $$S(i) = \max_{0 \leq j < i} \{S(j) : \texttt{dict}(s[j+1 \cdots i]) = \texttt{true}\}$$

   Then, string s can be reconstructed as a sequence of valid words if and only if $S(n) = 1$.

   Correctness and Runtime: Consider $S[1 \cdots i]$. If it is a sequence of valid words, there is a last word $s[j \cdots i]$ that is valid such that $S(j) = 1$ and the update will cause $S(i)$ to be set to 1. Otherwise, for any valid word $S[j \cdots i]$, $S(j)$ must be 0 and $S(i)$ will be set to 0. This runs in quadratic time $O(n^2)$ as there are n subproblems, each of which takes linear time $O(n)$ to be updated from the solution obtained from smaller subproblems.

   (b) Every time an $S(i)$ is updated to 1, keep track of previous item $S(j)$ that caused the update because $s[j+1 \cdots i]$ was a valid word. At termination, if $S(n) = 1$, trace back the series of updates to recover the partition in words.

   For each subproblem, this only adds a constant-time amount of work; therefore, this adds a total of $O(n)$ to pass over the array at the end. Hence, the overall runtime remains $O(n^2)$.

2. DPV Problem 6.8

SOLUTIONS AND GRADING GUIDELINES:

$\Longrightarrow$ 2 points for clearly described and correct subproblems
$\Longrightarrow$ 2 points for clearly described and correct algorithm and recurrence
$\Longrightarrow$ 2 points for showing correctness (no need for formal proof)
$\Longrightarrow$ 2 points for correct runtime

Subproblems: For $1 \leq i \leq n$ and $1 \leq j \leq m$, define subproblem $L(i,j)$ to be the length of the longest common substring of strings x and y that *terminate* at x_i and y_j. The recursion is then:

$$L(i,j) = \begin{cases} L(i-1,j-1)+1 & \text{if } x_i = y_j \\ 0 & \text{otherwise} \end{cases}$$

The initialization is, for all $1 \leq i \leq n$ and $1 \leq j \leq m$, simply $L(0,0) = L(i,0) = L(0,j) = 0$.

The output of the algorithm is the maximum of $L(i,j)$ over all $1 \leq i \leq n$ and $1 \leq j \leq m$,

Correctness and Runtime: The initialization is clear; therefore, it suffices to prove the correctness of the recursion. The longest common substring terminating at x_i and y_j must include x_i and y_j; therefore, it will be of length zero if these characters are different or of length $L(i-1, j=1)+1$ if they are equal.

The runtime is $O(mn)$ as we have $m \times n$ subproblems, with each subproblem taking constant time to evaluate through the recursion.

3. DPV Problem 6.13 – for this problem, assume that the cards have values in the range $[1, 13]$, with an Ace counting as 1, a Jack counting as 11, a Queen counting as 12, and a King counting as 13.

   SOLUTIONS AND GRADING GUIDELINES:

   $\Longrightarrow$ 2 points for correct sequence in part (a)
   $\Longrightarrow$ 2 points for clearly described and correct subproblems
   $\Longrightarrow$ 2 points for clearly described and correct algorithm and recurrence
   $\Longrightarrow$ 2 points for showing correctness (no need for formal proof)
   $\Longrightarrow$ 2 points for correct runtime

   (a) Consider the sequence $(2, 9, 1, 1)$. The best available card at the start has value 2, but this leads only to a score of 3 since the opponent will pick the card with value 9 in the next turn. In this case, the first player should select the card with value 1 initially, which guarantees a win with a total of 10.

   There are many other possibilities here.

   (b) *Subproblems:* Define subproblems $A(i, j)$ for $i \leq j$, where $A(i, j)$ is the *difference* between the largest total the first player can obtain and the corresponding score of the second player when playing on sequence $s_i, s_{i+1}, \cdots, s_j$. This will be positive if and only if the first player has the larger total.

   Algorithms and Recursion: We can solve all subproblems by initializing $A(i, i) = s_i$ for all i and using the following update rule for $i < j$:

$$A(i, j) = \max\{s_i - A(i + 1, j), s_j - A(i, j - 1)\}$$

   We can also keep track of the optimal move at each stage by simultaneously maintaining matrix M, where we set $M(i, j)$ to `first` if $A(i, j)$ takes the value of the first element in the maximization or to `last` otherwise.

   Correctness and Runtime: The recursion is correct since at any stage of the game, there are two possible moves for the first player: (1) choose the first card, in which case player one will gain s_i and score $-A(i+1, j)$ in the rest of the game; or (2) choose the last card, gaining s_j and scoring $-A(i, j - 1)$ from the remaining cards.

   The algorithm computes all of the subproblems and entries of M in quadratic time $O(n^2)$ since each update take constant time. At the end of the algorithm, the player can simply walk through matrix M to learn the best strategy at any point in the game.

4. DPV Problem 6.22 – as a hint, translate this problem into the Knapsack problem without repetition from Lab 5.

   SOLUTIONS AND GRADING GUIDELINES:

   $\Longrightarrow$ 2 points for clearly described and correct subproblems
   $\Longrightarrow$ 2 points for clearly described and correct algorithm and recurrence
   $\Longrightarrow$ 2 points for showing correctness (no need for formal proof)
   $\Longrightarrow$ 2 points for correct runtime

   This problem reduces to the Knapsack problem without repetition. The knapsack will have capacity t and each number a_i will be an item of value a_i and weight a_i.

   A subset adding up to t will exist if and only if a total value of t can fit in the knapsack.

   The dynamic programming algorithm then solves the problem in time $O(n \sum_i a_i)$.