Homework 01 (due 09/26)
COMP 5030: Algorithms
Computer Science Department, UMass Lowell
Fall 2023

Solved problem. (From Jeff Erickson) A **shuffle** of two strings X and Y is formed by interspersing the characters into a new string, keeping the characters of X and Y in the same order. For example, the string BANANAANANAS is a shuffle of the strings BANANA and ANANAS in several different ways.

BANANAANANAS BANANAANANAS BANANAANANAS

Similarly, the strings PRODGYRNAMAMMIINCG and DYPRONGARMAMMICING are both shuffles of DYNAMIC and PROGRAMMING:

PRODGYRNAMAMMIINCG DYPRONGARMAMMICING

Given three strings $A[1..m]$, $B[1..n]$, and $C[1..m+n]$, we wish to determine whether C is a shuffle of A and B.

(a) Define subproblems and give a recurrence that computes a subproblem based on smaller subproblems. (English description: 1pt; which subproblem corresponds to final answer: 1pt; base case: 1pt; recursive cases: 3pts.)

Solution.

(English description) We define a boolean function $Shuf(i, j)$, which is `true` if and only if the prefix $C[1..i+j]$ is a shuffle of the prefixes $A[1..i]$ and $B[1..j]$. (Final answer) The value of $Shuf(m, n)$ is what we want to compute.

$$Shuf(i,j) = \begin{cases} \texttt{true} & \text{if } i = j = 0 \text{ (Base case)} \\ Shuf(0, j-1) \wedge (B[j] = C[j]) & \text{if } i = 0 \text{ and } j > 0 \\ Shuf(i-1, 0) \wedge (A[i] = C[i]) & \text{if } i > 0 \text{ and } j = 0 \\ Shuf(i-1, j) \wedge (A[i] = C[i+j]) & \\ \vee Shuf(i, j-1) \wedge (B[j] = C[i+j]) & \text{if } i > 0 \text{ and } j > 0 \end{cases}$$

(b) Describe and analyze the dynamic programming algorithm. (Describe table: 1pt; dependency DAG: 2pt; time analysis: 1pt)

Solution.

(Table description) We can memoize all function values into a two-dimensional array $Shuf[0..m][0..n]$. (Dependency DAG) Each array entry $Shuf[i, j]$ depends only on the entries immediately below and immediately to the right: $Shuf[i-1, j]$ and $Shuf[i, j-1]$. Thus, we can fill the array in standard row-major order. (Runtime) There are $O(mn)$ subproblems, each can be computed in $O(1)$ time. Thus, the total runtime is $O(mn)$.

1. (10pts) Solve the following variant of the coin change problem. As input you are given an array $V = \langle v_1, \ldots, v_k \rangle$ with the values of k types of coins, an array $Q = \langle q_1, \ldots, q_k \rangle$ with the available quantity of each type of coin, and a value t which is the target change. The objective is to fin the minimum number of coins whose values sum up to exactly t. In other words, we want to find an array $S = \langle s_1, \ldots, s_k \rangle$ such that $s_i \leq q_i$, $\sum_{i=1}^{k} s_i v_i = t$, and $\sum_{i=1}^{k} s_i$ is minimized. Try to find a solution that runs in $O(tk \max_{i \in \{1,\ldots,k\}}(q_i))$.

Solution. We define the subproblem $C[t'][i]$ as the minimum number of coins with values in the prefix $\langle v_1, \dots, v_i \rangle$ of V that make up the change amount t' exactly.

$$C[t'][i] = \begin{cases} 0 & \text{if } t' = 0 \text{ (Base case)} \\ \infty & \text{if } t' < 0 \text{ or } i < 1 \text{ (Base case)} \\ \min_{q' \in \{1,\dots,q_i\}}(N[t' - q' \cdot v_i][i-1] + q') & \text{otherwise} \end{cases}$$

There are $t \cdot k$ subproblems each can be solved in $O(\max_{i \in \{1,\dots,k\}}(q_i))$ time. Thus the runtime is $O(tk \max_{i \in \{1,\dots,k\}}(q_i))$.

- English definition of the subproblem **1pt**

  "We define the subproblem $C[t'][i]$ as the minimum number of coins with values in the prefix $\langle v_1, \dots, v_i \rangle$ of V that make up the change amount t' exactly"

- Position of the final answer **1pt**

  "The final position will be at $C[t][k]$"

- Base cases **1pt** (.5pt each)

  1/2) 0 if $t' = 0$

  1/2) ∞ if $t' < 0$ or $i < 1$

- Recursive Case **5pt**

  Math: $\min_{q' \in \{1,\dots,q_i\}}(N[t' - q' \cdot v_i][i-1] + q_i)$

  Explanation: We try all possible quantities q_i of a given coin type v_i plus the solutions to all previous values of i and find the minimum

  Grading: Only the math is needed, if their math isn't correct, then give partial credit based on how close they get to the spirit of the recursion

  Note: There are more efficient solutions, there is a solution that uses binary encodings of q_i to reduce the runtime of this one by a log factor. There is also a *"very complicated solution" - Hugo 2023* that is linear. So if their case is different check the runtime and make sure it's not a better answer. If it's the linear answer be wary of a copy-pasted answer.

- Size of the table **1pt**

  $t \times k$

- Runtime **1pt**

  $O(tk \max_{i \in \{1,\dots,k\}}(q_i))$, maybe $O(tk \max_{i \in \{1,\dots,k\}} \log(q_i))$

2. (10pts) Solve exercise 15.1-3 from CLRS. (English description: 1pt; which subproblem corresponds to final answer: 1pt; base case: 1pt; recursive cases: 6pts; Describe table: 1pt; dependency DAG: 2pt; time and space analysis: 1pt.)

   Consider a modification of the rod-cutting problem in which, in addition to a price p_i for each rod, each cut incurs a fixed cost of c. The revenue associated with a solution is now the sum of the prices of the pieces minus the costs of making the cuts. Give a dynamic-programming algorithm to solve this modified problem.

Solution. We can modify the recurrence from the book by simply adding the cost of each cut as follows:

$$r_n = \begin{cases} 0 & \text{if } n = 0 \text{ (Base case)} \\ \max(p_n, \max_{1 \le i \le n-1}(p_i + r_{n-i} - c)) & \text{otherwise.} \end{cases}$$

Note that the case when we don't cut and sell the piece for p_n must be handled separately since there is no cutting cost.

- English definition of the subproblem **1pt**

  We define r_i as the maximum profit for cutting a rod of length i

- Subproblem of the final answer **1pt**

  The subproblem r_n where n is the target rod length

- Base cases **1pt**)

  0 if $n = 0$ (this was given in the solution template)

- Recursive Case **4pt**

  Math: $\max(p_n, \max_{1 \le i \le n-1}(p_i + r_{n-i} - c))$

  Explination: "modify the recurrence from the book by simply adding the cost of each cut + the case when we don't cut and sell the piece for p_n must be handled separately"

- Describe table **1pt**

  The table is a one-dimensional array

- Dependency Dag **1pt**

  Really interested in traversal of the table, in this case in each element of our DP table relies solely, on the element to its left

  Each subproblem r_n depends on all smaller subproblems r_i, $i \in \{1, \ldots, n-1\}$. Then, in a bottom-up DP we can fill in the table from left to right.

- Runtime **1pt**

  There are n entries in the table and they take $O(n)$ time to compute hence $\Theta(n^2)$