PSET 3

Due on **Tuesday October 28, 2024 at 11:59 pm ET**

CS 601: Algorithms and Complexity, Fall 2024 (Akash Kumar) Version: **1.0**

Submitted by <**name**>

Grading Policies:

(a) **Total points: 100**

(b) There are **five** problems in this pset (each of which have one or more subparts). Please solve/attempt them all.

(c) You are allowed –indeed you are encouraged– to, collaborate with other students in the class. Please list all the people you collaborated with.

(d) **Quite a few problems should be available online.** You should feel free to consult online resources, but please use those only after you spend some serious time thinking about the problems.

(e) I cannot emphasize this point enough. **With all the freedom given in the points above, it is imperative that you write your solutions yourself.** Please do not copy/paste what your friends wrote or what electronic material you find online. This defeats the purpose of solving a problem set. Indeed, **if you are caught engaging in unseemly practices, severe action will be taken.**

(f) Please write your pseudocode as you see those written in textbooks. Confusing/inscrutable psuedocodes will be penalized with no room for a regrade.

(g) Please write your mathematical claims rigorously and unambiguously. Again, follow the style laid out in textbook or in the class if you are unsure. Ambiguous or unclear claims/proofs will be penalized. Your TAs are not here to debug your proofs/algorithms. If your writeup does not convince them of the correctness, you will rightfully lose points.

1 (15 PTS.) PRODUCING TARGET SUMS USING DP

We call a sequence of n integers $x_1, \ldots, x_n$ valid if each x_i is in $\{1, \ldots, m\}$.

(a) (5 PTS.) Give a dynamic programming-based algorithm that takes in n, m and a "target" T as input and outputs the number of distinct valid sequences such that $x_1 + x_2 + \ldots + x_n = T$. Your algorithm should run in time $O(m^2n^2)$.

(b) (10 PTS.) Give an algorithm for the problem in part (a) that runs in time $O(mn^2)$.

Solution:

(a) Let us use $f(i, s)$ to denote the number of sequences with length i and sum s. The value i can take on n possible values. The number of possible values for s is mn. This means you are looking at a collection of mn^2 many subproblems. Alright, let us roll our sleeves and figure out a Dynamic Programming based approach.

For the base case, you note that

$$f(i,s) = \begin{cases} 0 \text{ if } s < 0; \\ 1 \text{ if } s = 0; \\ 1\text{if } i = 1 \text{ and } 1 \leqslant s \leqslant m \end{cases}$$

Now, let us think about the recursive part. So, you want to think about how many i-length valid sequences I can cook up which add up my favorite value s. Suppose you knew the number of $i-1$-length valid sequences which add up to $s-1, s-2, s-3, \ldots, s-m$. You can just add all of these up to obtain the expression for $f(i, s)$. This is exactly what the DP recursion says. Namely, you have

$$f(i,s) = \sum_{j=1}^{m} f(i-1, s-j).$$

You note that to fill out the next cell, you need to look at m preceding cells. So, you spend $O(m)$ time per subproblem and this means you spend a total of $O(m^2n^2)$ time this way.

Remark: I must add in a remark. If you fiddled around with this problem a bit, you would discover that you do not want to use a recursive formula that considers valid sequences with all values between 1 and i where you let i run from 1 to m. This is indeed kind of wasteful and unnecessarily slow. And yes, you just return $f(n, T)$.

(b) The key idea in this problem is to figure out whether we can solve each of the mn^2 many subproblems fast – like in $O(1)$ time. This can be done by looking at the DP recursion and massaging it carefully. So, you see

$$\begin{aligned} f(i,s) &= \sum_{j=1}^{m} f(i-1, s-j) \\ &= \sum_{j=0}^{m-1} f(i-1, s-j-1) \\ &= \left[\sum_{j=1}^{m} f(i-1, s-j-1)\right] + \left[f(i-1, s-1) - f(i-1, s-m-1)\right] \\ &= \left[f(i, s-1)\right] + \left[f(i-1, s-1) - f(i-1, s-m-1)\right] \end{aligned}$$

Here, the last step follows by the DP recurrence we established in part (a) above. But note that this derivation suggests something cool – we can indeed derive the value in the (i, s)-th cell of the DP table by looking at **only three!** previously filled out cells. Truly remarkable but that also means you can solve each of the mn^2 subproblems in $O(1)$ time which gives you the desired speedup.

2 (30 pts.) Scheduling Jobs on Supercomputers

You're running a massive physical simulation, which can only be run on a supercomputer. You've got access to two (identical) supercomputers, but unfortunately you have a fairly low priority on these machines, so you can only get time slots on them when they'd otherwise be idle. You've been given information about how much idle computing power is available on each supercomputer for each of the next n one-hour time slots: you can get a_i seconds of computation done on supercomputer A in the ith hour if your job is running on A at that point, or b_i seconds of computation if it running on supercomputer B at that point. During each hour your job can be scheduled on only one of the two supercomputers. You can move your job from one supercomputer to another at any point, but it takes an hour to transfer the accumulated data between supercomputers before your job can begin running on the new supercomputer, so a one-hour time slot will be wasted where you make no progress.

So you need to come up with a schedule, where for each one-hour time slot your job either runs on super- computer A, runs on supercomputer B, or "moves" (it switches which supercomputer it will use in the next time slot). If your job is running on supercomputer A for the $(i-1)$th hour, then for the ith hour your only two options are to continue running on A or to "move." The value of a schedule is the total number of seconds of computation that you get done during the n hours. You want to find a schedule of maximal value. Design an efficient algorithm to find the value of the optimal schedule, given $a_1, a_2, \dots a_n$ and $b_1, b_2, \dots, b_n$.

Solution:

There are many ways to solve this problem. My preferred way is to reduce this to Longest Path Problem in DAGs – a problem we know how to solve from class. *So, we are going to create a graph from thin air – how cool is that!*

So, I will produce a directed graph on $2n$ vertices. I will add edges in this graph that only go from left to right to capture the way I want the computation to proceed. Let us have n vertices, one for each hour come from supercomputer A and another n of them coming from supercomputer B. These vertices will be written as $A_1, A_2, \dots A_n, B_1, B_2, \dots, B_n$.

Now, if I wish to run the job on supercomputer A during the i-th hour, I will put an edge between vertices A_i and A_{i+1} with weight a_i to denote the amount of computation I was able to do on A during this hour. If I wish to migrate from A to B in the i-th hour, I perform no computation in this hour and therefore I put an edge from A_i to B_{i+1} with weight 0 to represent this wasted hour. Similarly, I define the edges from B_i to capture the decision made during the i-th hour if I started off at supercomputer B during the i-th hour.

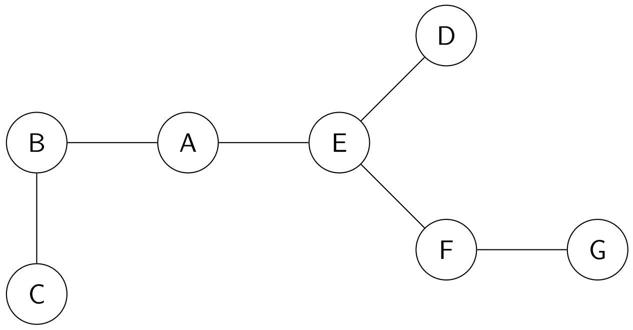
The longest path problem correponds to the optimal schedule. To get rid of the inconvenince in choosing start vertices (A_1 or B_1), I can always add a fake start vertex s and connect it with edges of weight zero to A_1 and B_1. Similarly, I can also add a sink node t which has edges of weight zero entering it from A_n and B_n. The running time of this procedure is $O(|V| + |E|)$ where V represents the vertex set of this DAG and E represents its edge set. This is easily seen to be $O(n)$.

3 (20 pts.) Vertex Cover

A vertex cover of a graph $G = (V, E)$ is a subset of vertices $S \subseteq V$ that includes at least one endpoint of every edge in E . Give a linear-time algorithm for the following task:

Input: An undirected tree $T = (V, E)$.

Output: The size of the smallest vertex cover of T. For instance, in the following tree, possible vertex covers include $\{A, B, C, D, E, F, G\}$ and $\{A, C, D, F\}$ but not $\{C, E, F\}$. The smallest vertex cover has size 3: $\{B, E, G\}$.



Solution:

It is convenient in this problem to choose a root for the tree. Pick an arbitrary vertex r and let us root the tree at r. Define $C(i)$ to mean the size of the minimum vertex cover in the tree rooted at u. You want to find $C(r)$. Let's take a pause here and make sure that sinks in.

All good so far? Okay, let's proceed. For a leaf vertex u, note that $C(u) = 0$ – this is the base case! Moreover, one also notes that if a vertex $u \in T$ is not a part of the vertex cover, then all of the neighbors must be included in the vertex cover. This suggests the following recurrence.

$$C(u) = \min \left(1 + \sum_{v:(u,v)\in E} C(v), \sum_{v:(u,v)\in E} (1 + \sum_{w:(v,w)\in E} C(w)) \right).$$

The first term accounts for the size of the min vertex cover which includes u. You find the min vertex cover in all the subtrees rooted at neighbors of u and then you add 1 to it to account for u being in the cover.

The second term looks at the situation when u is excluded from the cover. Now you go over all subtrees rooted at neighbors of u. In such a subtree, the root is already in the cover (accounted for by the plus 1 summand in the second term) and the second summand accounts for the min vertex cover in these subtrees. This is easily seen to be an $O(n)$ time solutin as each $C(w)$ value is accessed at most twice – one while computing $C(v)$ (parent of w) and second while computing $C(u)$ (grandparent of w).

4 (20 PTS.) A GAME CALLED PIG

Pig is a 2-player game played with a 6-sided die. On your turn, you can decide either to roll the die or to pass. If you roll the die and get a 1, your turn immediately ends and you get 1 point. If you instead get some other number, it gets added to a running total and your turn continues (i.e. you can again decide whether to roll or pass). If you pass, then you get either 1 point or the running total number of points, whichever is larger, and it becomes your opponent's turn. For example, if you roll 3, 4, 1 you get only 1 point, but if you roll 3, 4, 2 and then decide to pass you get 9 points. The first player to get to 100 points wins.

Suppose at some point the player whose turn it is has x points, their opponent has y points, and the running total for this turn so far is z. Let's work out what the optimal strategy is.

4.A. (5 PTS.) Let $W(x, y, z)$ be the probability that the current player will eventually win if both players play optimally. What is $W(x, y, z)$ if $x + z \geqslant 100$?

4.B. (5 PTS.) Suppose the current player decides to roll, and gets a 1. What's the probability that they'll still win, in terms of the function W?

4.C. (5 PTS.) Give a recursive formula for $W(x, y, z)$.

4.D. (5 PTS.) Describe a dynamic programming algorithm to compute $W(x, y, z)$. If you needed N points to win instead of 100, what would be the asymptotic runtime of your algorithm?

Solution:

(a) It should be clear that $W(x, y, z) = 1$ as the first player can just pass.

(b) After the current player, P_1 rolls a 1, her turn is over and now it is P_2's turn. Think about the point tally – P_1 has $x + 1$ points and P_2 has y points. The "z" value has been reset to zero. Thus, the winning probability of P_2 is seen to be $q = W(y, x + 1, 0)$. The probability P_1 wins is $p = 1 - q$.

(c) As noted in the last item, if P_1 rolls a 1, she will win with probabilty p computed above. With probability $5/6$ she rolls a different face. On each of these faces (which occur with probability $1/6$ each), you have P_1 wins with probability

$$W(x, y, z + \text{ The value } P_1 \text{ rolled})$$

. She also has the option of passing which we analyze soon. But note that her winning probability can be written as

$$\mathbf{Pr}(P_1 \text{ wins}) = \max\{\text{On passing}, \text{On not passing}\}$$

To make sense of this, note

$$\mathbf{Pr}(P_1 \text{ wins on not passing}) = 1/6 \cdot \Big[1 - W(y, x + 1, 0)\Big] + 1/6 \cdot \sum_{\text{roll}=2}^{6} W(x, y, z + \text{ roll}).$$

To finish up, let us compute $\mathbf{Pr}(P_1$ wins on passing $)$. This is seen to be

$$\mathbf{Pr}(P_1 \text{ wins on passing}) = 1 - W(y, x + \max(1, z), 0).$$

(d) We have all the ammunition we need to finish this off. The base case is pretty much what you saw in part 1 – if you have a large enough running total, just pass to win. What might be unclear is how long does this process take to terminate. Wait, does it terminate?

Let's see. In the recurrence we developed for W earlier, each recursive call either increases the number of points one player has or it increases the running total. Thus, the recursion must terminate!

What you now need to compute is the runtime of this algorithm. But this can be done by focusing on what are the biggest values that either of x, y or z can take. Note that none of them could be bigger than $N + 6$. This is because if either of x, y, z is that large, the game can be happily terminated by either P_1 or P_2. Thus, the total number of possible cells you ever need to fill is just $(N + 6) \times (N + 6) \times (N + 6) = O(N^3)$. Each cell can be filled in time $O(1)$ which makes up for a total of $O(N^3)$ time.

5 (15 pts.) Something on Strings

Let x, y , and z be strings. We want to know if z can be obtained only from x and y by interleaving the characters from x and y such that the characters in x appear in order and the characters in y appear in order. For example, if $x =$ **efficient** and $y =$ **ALGORITHM**, then it is true for $z =$ **effALGiORcilenTHMt**, but false for $z =$ **efficientALGORITHMextraCHARS** (miscellaneous characters), $z =$ **effALGORITHMicien** (missing the final t), and $z =$ **randomString** (obviously wrong). How can we answer this query efficiently? Your answer much be able to efficiently deal with strings such as $x =$ **aaaaaaaaaab** and $y =$ **aaaaaaaac**.

Solution:

As a preprocessing step, we first check whether z has the appropriate length (sum of lengths of x and y). Suppose this basic test is passed by z (otherwise z clearly lacks the desired property).

Let

$$S(i,j) = \begin{cases} 1 \text{ If } x[0:i] \text{ and } y[0:j] \text{ can be interleaved to make } z[0:i+j]. \\ 0 \text{ Otherwise.} \end{cases}$$

It is fairly direct to express $S(i,j)$ in terms of smaller subproblems. You can in fact write

$$S(i,j) = \begin{cases} 1 & \text{If } z[i+j] == x[i] \text{ and } S(i-1,j) \text{ is true.} \\ \\ 1 & \text{If } z[i+j] == y[j] \text{ and } S(i,j-1) \text{ is true} \\ \\ 0 & \text{Otherwise.} \end{cases}$$

The base cases are easily seen to be $S(i,0) = 1$ if the first i characters of x are the first i characters of z and similarly for y.