CMPS 102 — Winter Quarter 2017 – Homework 3

Solution to Problem 1 - Tokens

You're playtesting a new boardgame from one of your friends in the Game Design major, and they want your help to figure out the optimal strategy. The game has n rounds and in each round $1 \leq i \leq n$ you must acquire s_i tokens.
There are two possible ways to acquire tokens in round i:

(A) Spend r dollars per token, so $r * s_i$ in total, to acquire all tokens of round i.

(B) Spend C dollars to acquire the tokens of round i and of the next 3 rounds, independently of how many there are. (If you do this in round i, then your next choices concern rounds $i + 4$ and greater).

Example. Suppose $r = 1$, $C = 40$, and the sequence of s_i is $11, 9, 9, 12, 12, 12, 12, 9, 9, 11$. Then the cheapest way to acquire all the tokens is to choose action A for the first three rounds, then action B once, and then action A for the final three rounds.

Give a dynamic programming algorithm for finding the cheapest way to acquire all tokens, given the sequence $s_1, s_2, ..., s_n$. You must describe not only how you determine the value of the optimal sequence of actions, but also the sequence of actions itself.

OPT Identity

We define $OPT(j)$ to be the minimum cost needed to collect all the tokens from round $1...j$. These $OPT(j$values will be stored in an array of size n, where the j^{th} entry in this "OPT Array" corresponds to the j^{th} entry in the input array.

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \sum_{i=1}^{j} r \cdot s_i & \text{if } j = \{1, 2, 3\} \\ min\{(r \cdot s_i) + OPT(j-1), C + OPT(j-4)\} & \text{if } j \geq 4 \end{cases} \tag{1}$$

Proof. **For $j = 0$:**
The minimum cost must be 0, since there have been 0 rounds thus far, and we have not had the opportunity to obtain any tokens.

For $j = \{1, 2, 3\}$:
We must take option A for all choices, since option B requires at least 4 rounds. The first time B can be invoked is when $j = 4$ where option B is invoked on the first index.

For $j \geq 4$:
Our job is to consider our options between A and B, and pick the smaller one. Option A is to calculate the cost of the current round with $(r \cdot s_j)$, and add it to the running sum of minimum costs, represented by $OPT(j-1)$. Option B is to calculate the cost of four rounds C, and add it to $OPT(j-4)$, since we obviously want to avoid counting node values we shouldn't think of, and choosing Option B "occupies" four indexes. This is also where the cost C is paid. All that is left is to take these two values and take the smaller of the two, since our goal is end up with the minimum total cost and it wouldn't make sense to pick a higher cost at any point. □

Shadow Array:
We will call the shadow array BIT, and set it to size n. Then, we'll set the values like so:

$$BIT(j) = \begin{cases} 0 & \text{if Option A was chosen} \\ 1 & \text{if Option B was chosen} \end{cases} \tag{2}$$

So, we have a bitstring BIT. Notice that all values at $j = \{1, 2, 3\}$ are 0, since those correspond to the first three executions of the algorithm, where choosing B was not yet possible.

Using BIT, we can then determine the sequence of actions taken to yield the minimum total cost to collect all the tokens across n rounds.

Sequence Algorithm:

1. Set index $k = n$. Initialize a string ANS.
2. If $BIT(k) = 0$, append A to ANS and decrement k.
3. If $BIT(k) = 1$, append B to ANS and decrease k by 4.
4. Repeat this process until $k \leq 0$.
5. Reverse ANS to get a string of characters (as we recorded them in reverse order), with each character corresponding to both the choices made and the order in which they were made.

Proof. The reason why step 2 is true is because it represents choice A, which is localized to only one round. Rounds before and after a round where A was chosen are independent of each other. Remember that the goal of this algorithm is to retrace the steps taken that yield the minimum total cost. Step 3 is true because since we scroll backwards through the array, when we encounter a decision of B, then it must have been the last round in the set of 4. After writing this decision down, we can jump 4 ahead, since it's just one decision which spanned four entries in BIT. Thus, we can accurately record the choices we made in a string. □

The OPT and BIT arrays are filled at the same time as the algorithm executes n times. To find the value of the minimum cost, we wait until $OPT(n)$ is run, and take its value as the minimum cost. Afterwards, we run the Sequence Algorithm detailed above on the BIT array to get the string ANS corresponding to the decisions made and their order.

Observe that this results in two separate passes over arrays of size n. Thus, the runtime of this algorithm is $O(n)$.

Solution to Problem 2 - Monopoly

Another Game Design friend is working on a Monopoly spin-off with zoning laws! The board has n spaces, and the value of space i is $P(i)$. However, if you own space i you can't own any of spaces $i-2, i-1, i+1, i+2$.

(A) Give a dynamic programming algorithm for determining the maximum value of the spaces one can own. You do **not** need to output the corresponding set of spaces.

(B) Of course, in real Monopoly, the board wraps around, so let's take that into account as well. That is, assume that if you own space 1, you can not own spaces n or $n-1$ (besides spaces 2, 3). Solve this problem using DP (ideally, reusing your code for (a) (with tiny modifications) a few times).

Part A

We define $OPT(j)$ to be the maximum value of the first j spaces, and store each of these values in an array called OPT of size n.

OPT Identity

$$OPT(j) = \begin{cases} max_{1,2,3}\ P(j) & \text{if } j = \{1,2,3\} \\ max\{OPT(j-1), OPT(j-3) + P(j)\} & o.w. \end{cases} \tag{3}$$

Proof. **For $j = \{1,2,3\}$:**
For the first three spaces, it is impossible to pick more than one of them at a time, since all of them fall within two units of any other square in either direction. Thus, we must pick the largest of the three values to maximize the total value.

For all other values of $j \leq n$: The two options left are whether or not we take the space, or not. If we don't, then $OPT(j)$is indistinguishable from$OPT(j-1)$, so we choose that if that yields a greater value. If instead we choose to take the space, then we find the value of the current square $P(j)$, and add it to the only viable space (more than 2 spaces away) $OPT(j-3)$. By calculating these values and comparing them, we select the one which yields the greatest value.

Observe that these are the only options we can make at every j, and thus the identity holds. □

To actually calculate the maximum value of the spaces one can own, observe that by definition of $OPT(j)$, we seek the value of $OPT(n)$. Thus, we simply run $OPT(j)$ on all values $1...n$, and output the value of $OPT(n)$ value, which can be found in the n^{th} index of our OPT array. We don't need a shadow array for this problem, since the decisions we made to reach this output isn't necessary.

Each execution of $OPT(j)$ runs in $O(1)$ time, and we call $OPT(j)n$ times. Thus, the runtime of this algorithm is $O(n)$.

Part B

Solution to Problem 3 - Happy Customers

You are working on n different projects, but in m hours you are going on vacation. Imagine that for each project i, you had a function f_i that told you how happy the people paying for project i would be if out of your n available hours you devoted $0 \leq x \leq m$ to project i. Imagine further that each such function has maximum value at most s_i, corresponding to the project being fully finished (and thus the clients being perfectly happy).

Give a dynamic programming for determining the most happiness you can generate for your clients by splitting your m hours among the n projects.

- Because of book-keeping rules, you can't spend fractional hours on a project.
- The functions f_i are non-decreasing, i.e., for every project i, if $t_1 \leq t_2$, then $f_i(t_i) \leq f_i(t_2)$
- The running time of your algorithm must be $O(n^a m^b s^c)$ for some fixed integers $a, b, c \geq 0$.

Solution to Problem 4 - The Food Bin

A person is shopping at a store that carries n different food items as packaged goods. A box of the i-th item will bring the shopper a benefit of b_i and cost p_i dollars, where b_i and p_i are integers. The shopper would like to receive maximum benefit, subject to the requirement that they can't spend more than P dollars in total. So, the problem is which boxes to take (they can take multiple boxes of the same item. This is known as the integral shopping problem.

In the fractional shopping problem, the same n items are sold in bulk, so the shopper can take as much or as little as they like of each food item, instead of having to buy a whole box or nothing.

1. Suppose that the items have the same order if we sort them by price from low to high, and if we sort them by benefit from high to low. Give an efficient algorithm to find an optimal solution to the integral shopping problem in this special case, and argue that your algorithm is correct.
   Hint: Describe the algorithm in English (1 line). Argue correctness in 3 lines.

2. Prove that the fractional version has the following property: the optimal solution can be found by making a series of locally optimal choices. Therefore, give a greedy algorithm to solve this version and prove it's correct.
   Hint: Ditto

Part 1

Algorithm: Sort the items either by ascending price, or descending benefit (essentially benefit/price ratio, since we know that sorting by ascending price == sorting by descending benefit). Buy as many items starting from index = 1, moving on when you run out of items to buy of that type or if buying one more unit puts you over, until you hit or would go over P (assuming P is greater than the total cost of all items).

Proof. Let some supposedly better solution be O, and let the output of this algorithm to be S. Go to the first instant where the solutions differ. Since our solution is based on a sorting of benefit/price ratio, at any index i where the solutions differ means that they either moved on to a different item while they could still have purchased the previous one (of greater value), or stopped purchasing items while they still could have. In either case, simply either exchange their purchase decision (set $O(i)$ =S(i)) with the algorithms (since the benefit is guaranteed to be higher as the algorithm always purchase with highest value possible). This algorithm runs in $O(nlogn)$ to sort the items, then purchasing costs $O(n)$, which gets absorbed into $O(nlogn)$. □

Part 2

Algorithm: Sort the items based on benefit/price ratio, then purchase items until you spend exactly P dollars, using divisibility to guarantee you can purchase any dollar amount of an item.

Proof. Given a supposedly more optimal solution O and the algorithms solution S, let i be the index where the decisions first differ. Since $S(i)$ has the highest benefit/price ratio possible at i, that means $O(i)$ chose to purchase some item that was not the highest price ratio it could have purchased at the time. Thus, we can swap $O(i)$ with $S(i)$. If we do this for all indexes where O and S differ, then eventually $O = S$, as we can guarantee that at every point, swapping the choices made won't worsen the solution. Normally when simply

choosing by benefit/price ratio we can't guarantee optimality because such an algorithm is too short-sighted, and can't account for combinations that can use the P dollars more efficiently. However, with the power of divisibility, we can guarantee that we always spend P dollars, so we are simply concerned with making every dollar count by picking items with the greatest value i.e. benefit/price ratio. □