CS 3510: Design & Analysis of Algorithms (Sections A & B)

Homework 4: Dynamic Programming

Professors Dana Randall and Gerandy Brito **Due: 10/4/2022 11:59pm**

Name: Anthony Wong

1.) (30 points) A "contiguous subsequence of a list S" is a subsequence made up of consecutive elements of S. For instance, if $S = [2, 0.2, 4, 0.5, 2.2]$, then $[0.2, 4, 0.5]$ is a contiguous subsequence of S, but $[2, 0.5, 2.2]$ is not. Design a dynamic programming algorithm that takes input S, a list of positive real numbers, and returns the contiguous subsequence with the maximum product. For the example above, the answer would be $[4, 0.5, 2.2]$ because it has the maximum product of 4.4.

(a.) Define the entries of your table in words, e.g. $T[i]$ or $T[i, j]$ is ...

Solution:

We define $T[i]$ to be the max product possible from the list $S[1 \dots n]$ up to the index i including $T[i]$ where $1 \leq i \leq n$.

(b.) State your base cases and the recurrence for entries of your table in terms of smaller subproblems. Briefly explain in words why it is correct.

Solution:

Let us first state our base cases which are:

$T[0] = 0$, $T[1] = S[1]$

In defining the recurrence for smaller subproblems, we have the following:

$T[i] = \max\{S[i], S[i] * T[i-1]\}$ where $2 \leq i \leq n$.

We know that this propogates the correct answer because each element in the array is the max product up to the index of that element. We can either add onto this product with the number at our current index, or if the number at the current index is larger than the current number times the previous max product we just populate the current element as we should start a new contiguous product subsequence. This ensures that we are always producing the max possible product at any index.

(c.) Write pseudocode for your algorithm to solve this problem.

Solution:

MaxProduct(S=$S[1 \ldots n]$)

```
T[0] = 0
T[1] = S[1]
maxproduct = -1
left_index, right_index = -1, -1

for i = 2 to n:
    T[i] = max(S[i], S[i] * T[i - 1])
    if max(S[i], S[i] * T[i - 1]) > maxproduct:
        if S[i] > S[i] * T[i - 1]:
            maxproduct = S[i]
            left_index = i
            right_index = i
        else if S[i] * T[i - 1] > S[i]:
            max_product = S[i] * T[i - 1]
            right_index = i

return S[left_index to right_index + 1]
```

(d.) Analyze the running time of your algorithm.

Solution:

In this algorithm, we iterate through the given array once and populate the left and right indices within that same pass through. Therefore the runtime is $O(n)$.

2.) (30 points) You are consulting for a small investment company. They give you the price of Google's shares over the last n days. Let $S[1 \ldots n]$ be the array of share prices, where $S[1]$ is the price of the stock on the first day in this time period. During this window, the company wanted to buy a fixed number of shares on some day and sell all these shares on some later day (that is, only one purchase and one sale). The company wants to know when they should have bought and when they should have sold in order to maximize their profit. If there was no way to make money during the n days, you should report "Impossible".

For example, if $n = 3$ and $S = [9, 1, 5]$, then you should return "buy on day 2 and sell on day 3"

(a.) Define the entries of your table in words, e.g. $T[i]$ or $T[i, j]$ is ...

Solution:

We define the $T[i]$ as the maximum profit possible from the share price array $S[1 \ldots n]$ from the first day to day i where $1 \leq i \leq n$.

(b.) State your base cases and the recurrence for entries of your table in terms of smaller sub-problems. Briefly explain in words why it is correct.

Solution:

Let us first state our base cases which are:

$T[0] = -1$, $T[1] = S[1]$

In defining the recurrence for smaller subproblems, we have the following:

$T[i] = \max\{T[i-1], S[i] - \mathit{minelement}\}$ where $2 \leq i \leq n$ and $\mathit{minelement}$ is the smallest element we've come across.

We know this continually propagates the correct answer because we are essentially turning this problem into finding the max difference within the given array where each subproblem is finding the largest calculated distance up to that index. If we come across an element that cannot produce a larger difference than we've calculated, we simply propogate the previous element since the largest difference from the last time step hasn't changed. If we find a larger difference at our current time step we will propogate that difference since we've found a new max possible profit.

(c.) Write pseudocode for your algorithm to solve this problem.

Solution:

MaxPossibleProfit(S=$S[1 \ldots n]$)

```
T[0] = -1
T[1] = S[1]
```

```
minelement = S[1]
left_index, right_index, trailing_min_index = -1, -1, -1

# A trailing index is used to ensure that the index of a new
# minimum we find in the array is used in the calculation for
# max profit. left_index = start day and right_index = end day

for i = 2 to n:
    T[i] = max(T[i-1], S[i] - minelement)
    if S[i] < minelement:
        minelement = S[i]
        trailing_min_index = i
    else if S[i] > minelement:
        if S[i - 1] <= S[i] - minelement:
            right_index = i
            left_index = trailing_min_index

return "Impossible" if T[n] = -1 else
    return "buy on day {left_index} and sell on day {right_index}"
```

(d.) Analyze the running time of your algorithm.

Solution:

In this algorithm, we iterate through the given array once and find the start day, end day, and profit within that same iteration. Therefore we can sat the runtime of this algorithm is $O(n)$.

3.) (40 points) You are given coins of denominations $\{x_1, x_2, ..., x_n\}$, and there are **exactly two** coins of each denomination. Your task is to check if it is possible to make change for a value v. For example, if your denominations are $\{1, 2, 5\}$, you can make change for $v = 15$ using both coins of denominations 5 and 2 and one coin of denomination 1, but you cannot make change for $v = 20$. Design a Dynamic Programming algorithm to solve this problem.

(a.) Define the entries of your table in words, e.g. $T[i]$ or $T[i, j]$ is ...

Solution:

Because we can use each denomination twice, lets re-define our list of denominations as $S[x_1, x_1, x_2, x_2, ..., x_n, x_n]$ where the first element has index 1 and the last element has index $2n$. Each entry in $T[i, j]$ tells us if can make change for j (1 for Yes and 0 for No) using the denominations up to index i including $S[i]$ where $1 \leq i \leq 2n$ and $0 \leq j \leq v$.

(b.) State your base cases and the recurrence for entries of your table in terms of smaller subproblems. Briefly explain in words why it is correct.

Solution:

Let us first state our base cases which are:

$T[i, 0] = 0$ where $1 \leq i \leq 2n$, $T[0, j] = 0$ where $0 \leq j \leq v$

In defining the recurrence for smaller subproblems, we have the following:

$T[i, j] = \max\{T[i-1, j], T[i-1, j - S[i]\}$ where $1 \leq i \leq 2n$ and $0 \leq j \leq v$

This propagates the correct answer because each entry tells us if can make change for j using the denominations up to index i including $S[i]$. If we've already been able to make change for j with denominations up to index i, we know we can make change for all i after including i with the same j. If not, we look back at previous subproblems not including our current i and with a j value of our current j minus $S[i]$ (or the denomination value of our current i). This way we know if we can fit our current element into the sum by checking if we could fit a previous collection of denominations into some j without our current element value.

(c.) Write pseudocode for your algorithm to solve this problem.

Solution:

makesChange(S=$\{x_1, x_2, ..., x_n\}$, v=v)

```
# duplicates each denomination
S = [x for x in S for _ in range(2)]
# make S 1-indexing
S.insert(0, 0)
```

```
T[i, 0] = 0
T[0, j] = 0

for j = 0 to v:
    for i = 1 to len(S):
        # solve initial population
        if S[i] == j:
            T[i, j] = 1
        else:
            if j - S[i] > 0:
                T[i, j] = max(T[i - 1, j],T[i - 1, j - S[i])
            else:
                T[i, j] = 0

return S[len(S), v]
```

(d.) Analyze the running time of your algorithm.

Solution:

Given we are iterating a 2D array, we need to determine the dimensions, which is $2n$ x v. Thus we can say our runtime is $O(2nv) = O(nv)$. We also have $O(n)$ for doubling the array but this is overpowered by the term $O(nv)$.

(e.) Suppose your coin denominations are $\{1, 4, 7\}$. Create and fill out the DP table you would use to determine if you can make change for $v = 13$.

Solution:

S = [0, 1, 1, 4, 4, 7, 7]
Rows = numbers up to v
Columns = Indices of S

	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	1	1	1	1	1	1
2	0	0	1	1	1	1	1
3	0	0	0	0	0	0	0
4	0	0	0	1	1	1	1
5	0	0	0	1	1	1	1
6	0	0	0	1	1	1	1
7	0	0	0	0	0	1	1
8	0	0	0	0	1	1	1
9	0	0	0	0	1	1	1
10	0	0	0	0	1	1	1
11	0	0	0	0	0	1	1
12	0	0	0	0	0	1	1
13	0	0	0	0	0	1	1

Given that the bottom right corner is a 1, we can conclude that you can make change for $v = 13$ with the given denominations.