CS 270 Homework 6

Neel Gupta

March 1, 2023

Problem 1. Now let us consider the knapsack problem where you have infinitely many items of each kind. Namely, there are n different types of items. All the items of the same type i have equal size w_i and value v_i. You are offered with infinitely many items of each type. Design a dynamic programming algorithm to compute the optimal value you can get from a knapsack with capacity W.

a. Define subproblems to be solved.

b. Write a recurrence relation for the subproblems.

c. Make sure you specify

   i. base cases and their values

   ii. where the final answer can be found

d. What is the complexity of your solution?

Answer:

a. For a given capacity W, find the maximum value that can be obtained through choosing some various items amongst the first i items.

b. Let *mem* be an array for conducting dynamic programming on. Then $mem[i][w]$ will be the maximum capacity of the first i items with a knapsack capacity of w. At every level, we have two choices.

- Do not include any items of type i, then $mem[i][w] = mem[i-1][w]$
- Include 1 or more items of type i, then $mem[i][w] = \max(mem[i][w], mem[i][w - k * w_i] + k * v_i)$ where k is the number of items of type i that are incluided in the knapsack.

Therefore, the recurrence relation can be expressed as

$mem[i][w] = \max(mem[i-1][w], mem[i][w - k * w_i] + k * v_i)\text{when } k * w_i \leq W$

c.

```
procedure INFINITEKNAPSACK($w, v, W$)
    Intitialize $mem[0][w]$ $\forall w = 0, 1, ..., W$
    for $i = 1, 2, ..., n$ do
        for $w = 0, 1, ..., W$ do
            $mem[i][w] = mem[i-1][w]$
            for $k = 1, 2, ..., \lfloor \frac{w}{w_i} \rfloor$ do
                $mem[i][w] = \max(mem[i][w], mem[i][w - k * w_i] + k * v_i)$
            end for
        end for
    end for
    return $mem[n][W]$
end procedure
```

The base case is that none of each item has no value. The answer can be found in $mem[n][W]$.

d. The time complexity of the algorithm is $\Theta(nW^2)$, where n is the number of types of items and W is the capacity of the knapsack. For each of the n iterations, the first inner loop does W iterations and the second inner loop does upto $\frac{W}{w_i}$ iterations. Thus, the total runtime is $\Theta(nW^2)$. The total space complexity of this algorithm is $\Theta(nW)$ for the DP table.

Problem 2. Suppose we want to replicate a file over a collection of n servers, labeled $s_1, s_2, ..., s_n$. To place a copy of the file at server s_i results in a *placement cost* c_i, for an integer $c_i > 0$.

Now, if a user requests the file from seriver s_i, and no copy of the ile is present at s_i, then the servers $s_{i+1}, s_{i+2}, s_{i+3}, ...$ are searched in order until a copy of the file is finally found, say at server s_j, where $j > i$. This results in an *access cost* of $j - i$. The access cost is 0 if s_i holds a copy of the file. We will require that a copy of the file be placed at server s_n, so that all such searches will terminate, at the latest, at s_n.

We'd like to place copies of the files at the servers so as to minimize the sum of placement and access costs. Formally, we say that a *configuration* is a choise, for each server s_i with $i = 1, 2, ..., n - 1$of whether to place a copy of the file at s_i or not. (Recall that a copy is always placed at s_n.) The *total cost* of a configuration is the sum of all placement costs for servers with a copy of the file, plus the sum of all acccess costs associated with all n servers.

a. Define subproblems to be solved.

b. Write a recurrence relation for the subproblems.

c. Make sure you specify

   i. base cases and their values

   ii. where the final answer can be found

d. What is the complexity of your solution?

Answer:

a. The subproblems would be to find the minimum cost solution to get the file to a server i.

b. Consider an optimal solution O_i which has the minimum cost for getting the file to server i. To get to server j from server i given that $j > i$, we must pay the access cost c_j as well as all the access costs for servers $i+1$ to $j-1$.

$$O_j = c_j + O_i + (j-i-1) + (j-i-2) + ... + 1$$
$$O_j = c_j + O_i + \frac{(j-i)(j-i-1)}{2}$$

Therefore, a recurrence relation for the problem would be:

$$T[j] = c_j + \min\left(\frac{(j-i)(j-i-1)}{2} + T[i]\right) \; \forall j = i+1, i+2, ..., n$$

c.

```
procedure REPLICATEFILE($costs$,$n$)
    $T[0] \leftarrow 0$
    for $i = 1, 2, ..., n$ do
        for $j = i+1, i+2, ..., n$ do
            $T[j] = c_j + \min(\frac{(j-i)(j-i-1)}{2} + T[i])$
        end for
    end for
    return $T[n]$
end procedure
```

The base case is that there is no access placement cost to access the first server that the file begins in. The final answer will be in $T[n]$.

d. The total runtime is $\Theta(n^2)$ since there are upto n iterations at each iteration. The total space complexity is $\Theta(n)$ for the table.

Problem 3. Given n balloons, indexed from 0 to $n-1$. Each balloon is painted with a number on it represented by array nums. You are asked to burst all the balloons. If you burst balloon i, you will get $nums[left] * nums[i] * nums[right]$ coins where $left$ and $right$ are adjacent indices of i. After bursting the balloon, the $left$ and $right$ then become adjacent. You may assume $nums[-1] = nums[n] = 1$ and they are not real therefore you can't burst them. Design a dynamic programming algorithm to find the maximum coins you can collect by bursting the balloons wisely.

a. Define subproblems to be solved.

b. Write a recurrence relation for the subproblems.

c. Make sure you specify

   i. base cases and their values

   ii. where the final answer can be found

d. What is the complexity of your solution?

Answer:

a. The subproblems involves bursting a subset of balloons such that the maximum amount of coins can be collected.

b. Consider the following example where $n = 4, nums = [3, 1, 5, 8]$. At every level, we want to define a subproblem that increases the subset of the balloons popped, so we can store the value of the maximum coins per range, but since the order of popping is what matters, we will store the last popped balloon as well as the value within our table.

Let rows represent the starting index i and columns represent the ending index j. When $i = j$, the diagonals of the matrix will be filled with the value of beginning the popping at that index and will store that index as well as the last, in this case first as well, popping. Consider the following table after going through the diagonals and bursting that one first with the example.

	0	1	2	3
0	(3,0)	-1	-1	-1
1	-1	(15,1)	-1	-1
2	-1	-1	(40,2)	-1
3	-1	-1	-1	(40,3)

Then to find all the subsets of length two we will have to use the values found before and then iterate through all the possible values in the new subset of size $length + 1$ to find which one optimally pops ballons in the best order. When $i = 0, j = 1$, setting k to 0 would mean popping that one last so popping the balloon at index 1 would mean the value at index (1,1) added to what would happen when popping the 0-index ballon last given that the 1-index balloon has already been popped. So, $k = 0$, has total value $15 + 1 * 3 * 5 = 30$. Doing something similar for $k = 1$, the total value is 3 (from the table at index 0, 0) + 15 = 18, so popping the 0-index balloon after the 1-index balloon in an array of length two is more optimal, therefore we will store 30 along with the best last balloon bursting (30,0) at index (0,1). After doing this same task for all subarrays of length two, we get the following table.

	0	1	2	3
0	(3,0)	(30,0)	-1	-1
1	-1	(15,1)	(135,2)	-1
2	-1	-1	(40,2)	(48,3)
3	-1	-1	-1	(40,3)

To fill in the value at the index (0,2), we will start with $k = 0$ meaning the 0-index is popped last, therefore we use the table to lookup what the max value that was attained from a length of size 2 problem from index 1 to 2, which is exactly the value stored at index (1,2), thereofre the value at (0,2) = value at (1,2) + $1 * 3 * 8 = 159$ and ending at index 0, we will do this as well for $k = 1, 2$ to see that 0 being the last balloon to burst in this group maximizes the value. Thus, we store (159,0) at index (0,2). Repeating this for index (1,3), we see that when popping the 3-index balloon last we also get 159.

	0	1	2	3
0	(3,0)	(30,0)	(159,0)	-1
1	-1	(15,1)	(135,2)	(159,3)
2	-1	-1	(40,2)	(48,3)
3	-1	-1	-1	(40,3)

For the last subproblem of length 4, for $k = 0, 1, 2, 3$, we get the values of of 162, 52, 57, and 167. Therefore, we store (167,3) at index (0,3).

	0	1	2	3
0	(3,0)	(30,0)	(159,0)	(167,3)
1	-1	(15,1)	(135,2)	(159,3)
2	-1	-1	(40,2)	(48,3)
3	-1	-1	-1	(40,3)

We can then backtrack on the indices ending with 3, then checking the value at (0,2) to see that it ends in 0, then seeing the value of (1,2) to see that it ends at index 2, and then finally left with index 1. Therefore, the correct order of popping would be 1, 2, 0, 3 to maximize profit.

Therefore, we can write the following recurrence relation where *mem* will be our DP table.

$$mem[i][j] = \max(mem[i][j], mem[i][k-1] + mem[k+1][i] + nums[i-1] * nums[k] * nums[j+1])$$

$mem[i][k-1]$ is the value before the k-index balloon is popped. $mem[k+1][i]$ is the value after the k-index balloon is popped. $nums[i-1]$ and $nums[j+1]$ are the numbers just outside of the range which will multiply with the k-index ballon after the before and after balloons are all popped.

c.

```
procedure MAXBALLOONPOPPINGCOINS($nums$)
    $n \leftarrow$ len($nums$)
    Intitialize $mem$ to matrix of size $nxn$
    for $len = 1$ to $n$ do
        for $i = 1$ to $n - len$ do
            $j \leftarrow i + len - 1$
            for $k = i$ to $j$ do
                if $i = 0$ then $left \leftarrow 1$
                else $left \leftarrow nums[i-1]$
                end if
                if $j = n - 1$ **then** $right \leftarrow 1$
                **else** $right \leftarrow nums[j + 1]$
                **end if**
                $mem[i][j] = \max(mem[i][j],$
$mem[i][k - 1] + mem[k + 1][i] + left * nums[k] * right)$
            **end for**
        **end for**
    **end for**
    return $mem[0][n - 1]$
**end procedure**
```

For the base case of only having 1 balloon, the following algorithm would still work since left and right would be 1, so the return value of would be in $mem[0][0]$.

d. The overall runtime is $\Theta(n^3)$ since we can have up to n iterations for each subset and there are n^2 of them. The overall space complexity is $\Theta(n^2)$ for the DP table.

Problem 4. Suppose you have a rod of length N, and you want to cut up the rod and sell the pieces in a way that maximizes the total amount of money you get. A piece of length i is worth p_i dollars. Devise a dynamic programming algorithm to determine the maximum amount of money you can get by cutting the rod strategically and selling the cut pieces.

a. Define subproblems to be solved.

b. Write a recurrence relation for the subproblems.

c. Make sure you specify

   i. base cases and their values

   ii. where the final answer can be found

d. What is the complexity of your solution?

Answer:

a. Given the prices of each piece, the subproblems can be said to be findng the maximum amount of money that is recieved after cutting a rod of length i.

b. The recurrence relation for the problem would be

$$rev(i) = \max(price[j] + rev(i-j)), \forall j = 1, 2, ..., i$$

where $price[j]$ is the price of a rod of length j and $rev(i-j)$ is the maximum possible revenue that can be recieved from a rod of size $i-j$.

c.

```
procedure MAXREVENUE(prices,$n$)
    revenue[0] ← 0
    for $i = 1, ..., n$ do
        currMax ← -1
        for $j = 1, ..., i$ do
            currMax ← max(currMax, prices[$j$]+revenue[$i-j$])
        end for
        revenue[i] ← currMax
    end for
    return revenue[$n$]
end procedure
```

d. The base case for when a rod of length 0 is that the maximum revenue will be 0, so revenue[0] = 0. The final answer can be found in revenue[n].

e. The time complexity of the solution is $O(n^2)$ since we have n iterations with upto n iterations each. The space complexity of the solution would be $\Theta(n)$.

Problem 5. You are trying to run a large computing job in which you need to simulate a physical system for as many discrete *steps* as you can. The lab you're working in has two large supercomputers (which we'll call A and B) which are capable of processing this job. However, you're not one of the priority users of these supercomputers, so at any given point in time, you're only able to use as many spare cycles as these machines have available

Here's z the problem you face. Your job can only run on one of the machines machines in any given minute. Over each of the next n minutes, you have a "profile" of how much processing power is available on each machine. In minute i, you would be able to run $a_i > 0$ steps of the simulation if your job is on machine A, and $b_i > 0$ steps of the simulation if your job is on machine B, You also have the ability move your job from one machine to the other, but doing this costs you a minute of time in which no processing is done on your job.

So, given a sequence of n minutes, a *plan* is a specied by a choice of A, B, or *"move"* for each minute, with the property that choices A and B cannot appear in consecutive minutes. The *value* of a plan is the total number of steps that you manage to execute over the n minutes: so it's the sum of a_i over all minutes in which the job is on A, plus the sum of all b_i over all minutes in which the job is on B.

Given values $a_1, a_2, ..., a_n$ and $b_1, b_2, ..., b_n$, find a plan of maximum value. Note that your plan can start with either of the machines A or B in minute 1.

a. Show that the following algorithm does not correctly solve this problem, by giving an instance on which it does not return the correct answer. In your example, say what the correct answer is and also what the algorithm above finds.

```
In minute 1, choose the machine with the larger of $a_1, b_1$
$i \leftarrow 2$
**while** $i \leq n$ **do**
    What was the choice in minute $i-1$?
    **if** $A$ **then**
        **if** $b_{i+1} > a_i + a_{i+1}$ **then**
            Choose *move* in minute $i$ and $B$ in minute $i+1$
            Proceed to iteration $i+2$.
        **else**
            Choose A in minute $i$
            Proceed to iteration $i+1$
        **end if**
    **else if** $B$ **then** behave as above with roles of $A$ and $B$ reversed
    **end if**
**end while**
```

b. Give an efficient algorithm that takes values $a_1, a_2, ..., a_n$ and $b_1, b_2, ..., b_n$ and returns the value of an optimal plan.

   i. Define subproblems to be solved.

   ii. Write a recurrence relation for the subproblems.

   iii. Make sure you specify

i. base cases and their values

ii. where the final answer can be found

iv. What is the complexity of your solution?

Answer:

a. Consider the following counter example

Minutes	1	2	3	4
A	2	1	1	1
B	1	10	1	1

The above algorithm would return a_1, a_2, a_3, a_4 as the plan with value 5. A more optimal solution would be to take b_1, b_2, b_3, b_4 that has value 13.

b.

i. The subproblems are to find a sequence of instructions such that we maximize the productivity of the machines over our workload.

ii. Let O be our optimal solution and that we have $O[A][i-2]$ and $O[A][i-1]$ which are the optimal solutions at time $i-2$ and $i-1$ that end with machine A and $O[B][i-2]$ and $O[B][i-1]$ which are the optimal solutions at time $i-2$ and $i-1$ that end with machine B. At every iteration, we have the choice to stay or move. If we move, we lose a turn where we could have been making money. Therefore, we get the following recurrence relations

$$mem[A][i] = \max(mem[A][i-1] + a_i, mem[B][i-2])$$

$$mem[B][i] = \max(mem[B][i-1] + b_i, mem[A][i-2])$$

iii.
```
procedure CHOOSEMACHINES(a, b, n)
    $mem[A][0], mem[B][0] \leftarrow 0$
    $mem[A][1], mem[B][1] \leftarrow a_1, b_1$, respectively
    for $i = 2, 3, ..., n$ do
        if $mem[A][i-1] + a_i < mem[B][i-2]$ then
            $mem[A][i] \leftarrow mem[B][i-2]$
        else
            $mem[A][i] \leftarrow mem[A][i-1] + a_i$
        end if
        if $mem[B][i-1] + b_i < mem[A][i-2]$ then
            $mem[B][i] \leftarrow mem[A][i-2]$
        else
            $mem[B][i] \leftarrow mem[B][i-1] + b_i$
        end if
    end for
    return max($mem[A][n], mem[B][n]$)
end procedure
```

The basecase is that not starting the jobs means that at minute 0, no progress has been made. Therefore, $mem[A][0], mem[B][0]$ get initalized to 0. If there is only one job, then the algorithm will return the machine with the higher processing power of the two. The final answer can be found in either $mem[A][n]$ or $mem[B][n]$ depending on which machine would be more optimal to end with.

iv. The total runtime of this algorithm would be $\Theta(n)$ since at most constant time work is being done at each iteration. Therefore, the algorithm runs in linear time. The space complexity would be $O(2n)$ for storing all the progress made by the two factories which is $\Theta(n)$.