CSCI 570 Homework 3
Spring 2023

Due Date: Feb. 25, 2023 at 11:59 P.M.

1. Suppose you are responsible for organizing and determining the results of an election. An election has a winner if one person gets at least half of the votes. For an election with n voters and k candidates, design a divide and conquer algorithm to determine if the election has a winner and finds that winner. Your algorithm should run in $O(nlogn)$ and is allowed to use $O(1)$ extra memory.

   Solution:

   If there is a winner in this election then the $\frac{n}{2}th$ element in the sorted array is the number assigned to the winner and the $\frac{n}{2}$ element in the sorted list of votes is the winner. We use the merge sort and merge the list of votes in $O(nlogn)$. We check the $\frac{n}{2}th$ element and iterate through the list and count the number of time it appears in the list in $O(n)$. If it appeared more than $\frac{n}{2}$ then it is the winner of the election. Otherwise the election does not have a winner. The algorithm takes $O(nlogn)$

2. Alice has learned the sqrt function in her math class last week. She thinks that the method to compute sqrt function can be improved at least for perfect squared numbers. Please help her to find a divide and conquer algorithm to find perfect squared numbers and their squared root. The input to the algorithm is n and the output is its squared root if it is a perfect squared number.

   Solution:

   The solution is based on binary search algorithm. We run the binary search algorithm to find the squared root of the number.

   if n is the number, we start the binary search and compute $k = (\frac{n}{2})^2$ and compare n with it.

   - if $n = k$, n is the perfect squared root of k and the algorithm returns k
   - if $n > k$ then we continue the binary search from $\frac{n}{2}$to n.

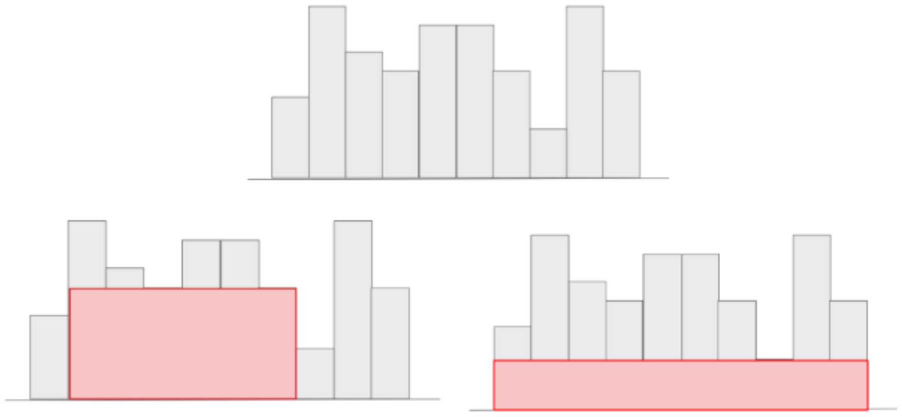

Figure 1: The example in Q3

- if $n < k$ we continue the binary search on 1 to $\frac{n}{2}$.
- if the upper bound and lower bound of the array are similar then the algorithm terminates and n is not a perfect squared number.

Similar to the binary search algorithm the time complexity is:

$$T(n) = T(\frac{n}{2}) + O(1) \rightarrow T(n) = O(logn)$$

3. Alice has recently started working on the Los Angeles beautification team. She found a street and is proposing to paint a mural on their walls as her first project. The buildings in this street are consecutive, have the same width but have different heights. She is planning to paint the largest rectangular mural on these walls. The chance of approving her proposal depends on the size of the mural and is higher for the larger murals. Suppose n is the number of buildings in this street and she has the list of heights of buildings. Propose a divide and conquer algorithm to help her find the size of the largest possible mural and analyze the complexity of your algorithm. The picture below shows a part of that street in her proposal and two possible location of the mural:

   Solution:

   divide step: Let a be the array of buildings heights. Find the element with minimum height in the array and split the array to subarray of the right side of the minimum and subarray of the left side of the minimum. Continue the process for the two subarrays and find the size of the largest rectangle on each subarray.

conquer step: The size of the largest rectangle on the two subarrays of a is computed in the dividing step. The size of largest rectangle in a is either of these cases:

- The largest rectangle found in the right side array S_{right}
- The largest rectangle found in the left side array S_{left}
- The rectangle with width of all of the buildings in a and height of minimum element in a: $len(a) \times min(a)$

return the maximum value from the three possible cases

Complexity analysis: let T(n) be the complexity of the algorithm for an array of length n. The complexity of finding minimum is O(n) therefore:

$$T(n) = T(n - i - 1) + T(i - 1) + O(n)$$

Where the value of i depends on the location of minimum in the array and in the worst case scenario:

$$T(n) = T(n - 1) + O(n) \rightarrow T(n) = O(n^2)$$

4. Solve the following recurrence using the master theorem.

a) $T(n) = 8T(\frac{n}{2}) + n^2log(n)$

b) $T(n) = 4T(\frac{n}{2}) + n!$

c) $T(n) = 2T(\frac{n}{4}) + \sqrt{n}$

d) $T(2^n) = 4T(2^{n-1}) + 2^{\frac{n}{2}}$

Solution

a) $a = 8, b = 2 \rightarrow log_b^a = 3 \rightarrow f(n) = O(n^{3-e})$

Case 1: T(n) = O(n^3)

b)a = 4, b = 2 $\rightarrow log_b^a = 2 \rightarrow f(n) = \Omega(n^2)$

Case 3: T(n) = $\theta(n!)$

c)a = 2, b = 4 $\rightarrow log_b^a = 0.5 \rightarrow f(n) = \theta(\sqrt{n})$

Case 2: T(n) = $\sqrt{n}log(n)$

$d)$k = $2^n \rightarrow T(k) = 4T(\frac{k}{2}) + \sqrt{k}$
Case 1: T(k) = O(k^2)

5. 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 (in plain English) subproblems to be solved.

   b. Write a recurrence relation for the subproblems

   c. Using the recurrence formula in part b, write pseudocode to find the solution.

   d. Make sure you specify

      i. base cases and their values

      ii. where the final answer can be found

   e. What is the complexity of your solution?

Solution:

a. Let r [0], . . . , r [n] be the array where r [i]: is the maximum money we can get for a rod of length i.

b. Considering the recurrence relation of r [i]: At each remaining length of the rod, we can choose to cut the rod at any point, and obtain points for one of the cut pieces and compute the maximum points we can get for the other piece.
r[i] = max(r[i], r[j] + p[i-j]) for ($0 \leq$ j $\leq$ i $\leq$ n)

c. Input: an integer array r and the length of this array n.
Initialize: set r [0] = 0 and Ans = 0.

```
Let r[0, ..., n] be a new array
r[0] = 0
for i = 1 to n do
    r[i] = 0
```

```
        for j = 0 to i do
            r[i] = max(r[i], r[j] + p[i - j])
        end for
    end for
    return r[n]
```

d. i. Base case is : r[0] = 0

   ii. Solution can be found at : r[n]

e. The time complexity of this proposed algorithm is $O(n^2)$ as there are n subproblems and each subproblem costs O(n) to compute in worst case.

6. Tommy and Bruiny are playing a turn-based game together. This game involves N marbles placed in a row. The marbles are numbered 0 to N-1 from the left to the right. Marble *i* has a positive value m_i. On each player's turn, they can remove either the leftmost marble or the rightmost marble from the row and receive points equal to the sum of the remaining marbles' values in the row. The winner is the one with the higher score when there are no marbles left to remove.

   Tommy always goes first in this game. Both players wish to maximize their score by the end of the game.

   Assuming that both players play optimally, devise a Dynamic Programming algorithm to return the difference in Tommy and Bruiny's score once the game has been played for any given input. Your algorithm must run in $O(N^2)$ time.

   a. Define (in plain English) subproblems to be solved.

   b. Write a recurrence relation for the subproblems

   c. Using the recurrence formula in part b, write pseudocode to find the solution.

   d. Make sure you specify

      i. base cases and their values

      ii. where the final answer can be found

   e. What is the complexity of your solution?

Solution:

a. Define OPT(i, j) as the maximum difference in score achievable by the player whose turn it is to play, given that the marbles from index i to j (inclusive) remain.

b. We first calculate a prefix sum for the marbles array. This enables us to find the sum of a continuous range of values in O(1) time.
If we have an array like this: [5, 3, 1, 4, 2], then our prefix sum array would be [0, 5, 8, 9, 13, 15].
Then, the recurrence relation becomes
$\text{if\_take}_{\text{i}} = \text{prefix\_sum}_{\text{j+1}} - \text{prefix\_sum}_{\text{i+1}} - \text{OPT}_{\text{i+1,j}}$
$\text{if\_take}_{\text{j}} = \text{prefix\_sum}_{\text{j}} - \text{prefix\_sum}_{\text{i}} - \text{OPT}_{\text{i,j-1}}$
$\text{OPT}_{\text{i,j}} = \max(\text{if\_take}_{\text{i}}, \text{if\_take}_{\text{j}})$
for all i, j ($0 \leq i \leq N-1$, $i+1 \leq j \leq N-1$)

c. Let n be the length of the marbles array
Let prefix sum be the calculated prefix sum array for the array marbles (takes $\theta(n)$ time)

```
Let OPT[[0, ..., 0], ..., [0, ..., 0]] be a new n*n
array with values initialized to 0

for i = n − 2 to 0 do
    for j = i + 1 to n − 1 do
        $\text{if\_take}_i$ = $\text{prefix\_sum}_{j+1} - \text{prefix\_sum}_{i+1} - \text{OPT}_{i+1,j}$
        $\text{if\_take}_j$ = $\text{prefix\_sum}_{j} - \text{prefix\_sum}_{i} - \text{OPT}_{i,j-1}$
        $\text{OPT}_{i,j}$ = max($\text{if\_take}_i$, $\text{if\_take}_j$)
    end for
end for
return $\text{OPT}_{0,n-1}$
```

d. i. Base case is : $\text{OPT}_{i,j} = 0$ for all i, j ϵ 0, . . ., n-1
ii. Solution can be found at : $\text{OPT}_{0,n-1}$

e. The time complexity of this proposed algorithm is $O(n^2)$ as there are n^2 subproblems and each subproblem costs O(1) to compute.

7. The Trojan Band consisting of n band members hurries to lined up in a straight line to start a march. But since band members are not positioned by height the line is looking very messy. The band leader wants to pull out the minimum number of band members that will cause the line to be in a formation (the remaining band members will stay in the line in the same order as they were before). The formation refers to an ordering of band members such that their heights satisfy $r_1 < r_2 < ... < r_i > ... > r_n$, where $1 \leq i \leq n$.

   For example, if the heights (in inches) are given as

$$R = (67, 65, 72, 75, 73, 70, 70, 68)$$

   the minimum number of band members to pull out to make a formation will be 2, resulting in the following formation:

$$(67, 72, 75, 73, 70, 68)$$

   Give an algorithm to find the minimum number of band members to pull out of the line.

   Note: you do not need to find the actual formation. You only need to find the minimum number of band members to pull out of the line, but you need to find this minimum number in $O(n^2)$ time.

   a. Define (in plain English) subproblems to be solved.
   b. Write a recurrence relation for the subproblems
   c. Using the recurrence formula in part b, write pseudocode to find the solution.
   d. Make sure you specify
      i. base cases and their values
      ii. where the final answer can be found
   e. What is the complexity of your solution?

   Solution:

a. Let $\mathrm{OPT}_{\mathrm{left(i)}}$ be the maximum length of the line to the left of band member i (including i) which can be put in order of increasing height (by pulling out some members)

   Note: the problem is symmetric, so we can flip the array R and find the same values from the other direction. Let's call those values $\mathrm{OPT}_{\mathrm{right(i)}}$.

b. The recurrence relations are:

   $\mathrm{OPT}_{\mathrm{left(i)}} = \max(\mathrm{OPT}_{\mathrm{left(i)}}, \mathrm{OPT}_{\mathrm{left(j)}} + 1)$ such that $\mathrm{r_i} > \mathrm{r_j}$ $\forall$ $1 \leq \mathrm{j} < \mathrm{i}$

   $\mathrm{OPT}_{\mathrm{right(i)}}$ = same once the array is flipped

   (Below paragraph not required as part of solution)

   This problem performs a Longest - Increasing - Subsequence operation twice. Once from left to right and once again, but from right to left. Once we have the values for the longest subsequence we can make after we "pull out" a few band members, we can iterate over the array and determine what minimum number of pull-outs will cause our line of band members to satisfy the height order requirements.

c. Let n be the length of the input array and R be the input array

```
for i = 1 to n do
    $OPT_{left}(i) = OPT_{right}(i) = 1$ # Base case
end for

for i = 2 to n do
    for j = 1 to i − 1 do
        if $r_i > r_j$ then
            $OPT_{left}(i) = \max(OPT_{left}(i), OPT_{left}(j) + 1)$
        end if
    end for
end for

for i = 2 to n do
    for j = 1 to i − 1 do
```

```
            if $r_{n-i+1} > r_{n-j+1}$ then
                $OPT_{right}(i) = \max(OPT_{right}(i), OPT_{right}(j) + 1)$
            end if
        end for
    end for

    result $= \infty$
    for i = 1 to n do
        result $= \min($result, n $-$ $(OPT_{left}(i)+OPT_{right}(n-i+1)-1)$
    end for
    return result
```

d. i. Base case is : OPT_{left}(i) = OPT_{right}(i) = 1 for all i, j ϵ 1, . . ., n

   ii. Solution can be found at : min(n - (OPT_{left}(i)+OPT_{right}(n-i+1)-1) for i ϵ 1, . . ., n

e. The runtime of this algorithm is dominated by the nested for loops used to calculate the LIS both ways. This takes $O(n^2)$ each time. Therefore, the time complexity of the solution is $O(n^2)$.

8. From the lecture, you know how to use dynamic programming to solve the 0-1 knapsack problem where each item is unique and only one of each kind is available. Now let us consider knapsack problem where you have in- finitely 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 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 (in plain English) subproblems to be solved.

   b. Write a recurrence relation for the subproblems

   c. Using the recurrence formula in part b, write pseudocode to find the solution.

   d. Make sure you specify

   i. base cases and their values
   ii. where the final answer can be found

e. What is the complexity of your solution?

Solution:

a. Similar to what is taught in the lecture, let $OPT(k, w)$ be the maximum value achievable using a knapsack of capacity $0 \leq w \leq W$ and with k types of items $1 \leq k \leq n$.

b. We find the recurrence relation of $OPT(k, w)$ as follows. Since we have infinitely many items of each type, we choose between the following two cases:

   i. We include another item of type k and solve the sub-problem $OPT(k, w - w_k)$.

   ii. We do not include any item of type k and move to consider next type of item this solving the sub-problem OP T (k - 1, w).

   Therefore, we have
   $OPT(k, w) = max\{OPT(k - 1, w), OPT(k, w - w_k) + v_k\}$

c. Let OPT be a n*W sized 2-D array

```
OPT(0,0) = 0
for k = 1 to n do
    for w = 0 to W do
        if w $\geq$ w$_k$ then
            OPT(k,w) = max{OPT(k−1,w), OPT(k, w−w$_k$)+v$_k$}
        else
            OPT(k,w) = OPT(k−1,w)
        end if
    end for
end for

return OPT(n,W)
```

d. i. Base case is : OPT(0,0) = 0

ii. Solution can be found at : OPT(n,W)

e. The runtime of this algorithm is dominated by the nested for loops used to calculate the OPT. This takes $O(W)$ for $O(n)$ times. Therefore, the time complexity of the solution is $O(W * n)$.

9. 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. Here left and right are adjacent indices of i. After the burst, the left and right then becomes adjacent. You may assume $nums[-1] = nums[n] = 1$ and they are not real therefore you can not burst them. Design an dynamic programming algorithm to find the maximum coins you can collect by bursting the balloons wisely.

   Here is an example. If you have the $nums$ array equals $[3, 1, 5, 8]$. The optimal solution would be 167, where you burst balloons in the order of 1, 5 3 and 8. The left balloons after each step is:

$$[3,\ 1,\ 5,\ 8] \rightarrow [3,\ 5,\ 8] \rightarrow [3,\ 8] \rightarrow [8] \rightarrow []$$

   And the coins you get equals:

$$167 = 3 * 1 * 5 + 3 * 5 * 8 + 1 * 3 * 8 + 1 * 8 * 1.$$

   a. Define (in plain English) subproblems to be solved.
   b. Write a recurrence relation for the subproblems
   c. Using the recurrence formula in part b, write pseudocode to find the solution.
   d. Make sure you specify
      i. base cases and their values
      ii. where the final answer can be found
   e. What is the complexity of your solution?

   Solution:

   a. Let $OPT(l, r)$ be the maximum number of coins you can obtain from balloons $l, l + 1, ..., r - 1, r$.

b. The key observation is that to obtain the optimal number of coins for balloon from l to r, we choose which balloon is the last one to burst. Assume that balloon k is the last one you burst, then you must first burst all balloons from l to k - 1 and all the balloons from k + 1 to r which are two sub- problems. Therefore, we have the following recurrence relation:

$OPT(l,r) = \max_{l \leq k \leq r}\{OPT(l,k-1)+OPT(k+1,r)+nums[k]*nums[l-1]*nums[r+1]\}$

c.
```
Let OPT[0..n][0..n] be a new 2D array
OPT(i,j) = 0 for all i < j (0 <= i, j < n)
for len = 1 to n do
    for i = 0 to n-len do
        j = i + len - 1
        for k = i to j do
            OPT(i,j) = max(OPT(i,j), OPT(i,k-1) + \
            nums[i-1]*nums[k]*nums[j+1]+OPT(k+1,j))
        end for
    end for
end for
return OPT(0,n-1)
```

d. i. Base case is : OPT(i,j) = 0 for all i < j ($0 \leq$ i, j < n)

ii. Solution can be found at : OPT(0,n-1)

e. For running time analysis, we in total have O(n^2) and computation of each state takes O(n) time. Therefore, the total time is O(n^3).

10. **Online Questions.** Please go to DEN (https://courses.uscden.net/) and take the online portion of your assignment.