**CSCI570 Homework 3**

1. **SOLUTION:**

* Here we are given n voters and k candidates
* We create an array of size n for all the voters and fill it up with values of the candidates voted by the respective voters (0, 1, …, k)
* If we have no candidates standing for the election then there is no winner. If we have 1 candidate then we know that he is the winner. If we have 2 candidates we directly check for the winner by comparing the votes. If we have 3 or more than 3 we check for the winner recursively in the following way by performing function findWinner():
  + We first divide our array into two halves and run findWinner() on both halves recursively
  + The winner of the left subarray and the winner of the right subarray are stored separately in variable leftWinner and rightWinner. We check of the following 4 conditions to find the winner of every recursive step.

1. If both sides (leftWinner and rightWInner) have the same winner then he is the winner.
2. If leftWinner and rightWinner are two different candidates we check for which one has more votes in total when the arrays are combined. This has to be more than n/2 for there to have a winner.
3. After having two different winners in the right-side and left-side, if none of the candidates have more than half of the votes, we have no winner.
4. If left side as well as right sides have no winners individually, we can say that there is exists no winner.

* After performing these steps, we finally get the winner of the election or there is no winner if none of the candidates have no more than n/2 votes. The time complexity for the same is as we are dividing the arrays into subarrays recursively and performing the traversal n times.
* As we are only using a few defined variables such as leftWinner and rightWinner we are using only which is constant space to store the data at each step.

1. **SOLUTION:**

* Here, we have been given a number n and we need to find its square root.
* Along with this we have to check if n is a perfect square root.
* We can solve this problem by following these steps:
  + First take an array of size n starting from 0 to n
  + Then perform a binary search operation on the following array and find the term in the middle of the array by dividing the array in half
  + Extracting the middle element, we can check if the square of the middle element is greater than n, equal to n or lesser than n
  + If square of the middle term is greater than n, then we search in the left side of the array and if the square of the middle term is lesser than n, we search the right side of the array.
  + We perform this recursively until we get the square root of the number n
  + If we are not able to find the square root of the number using this method, the number n is not a perfect square.
* Time Complexity of this algorithm is as it is the time complexity of binary search.

1. **SOLUTION:**

* We can solve this problem of finding the largest area rectangular mural possible by using the divide and conquer strategy. We can follow these steps to do the same:
* Divide the buildings into two equal parts using the middle of the buildings in a row as the midpoint
* We now need to recursively find the largest area which will either be from the maximum area on the left or the maximum area from the right or it starts from the left and ends in the right
* When we follow the recursive process for the left we get the left max area and following the right most step we get the right max area
* Calculating for middle area we do the following:
* Create an array Arr[ ] which will store the base heights which are common.
* Now, traversing from the middle to the left most end we will update the array values to the minimum value that occurs
* Now, traversing from the middle to the right most end we will update the array values to the minimum value that occurs
* Now, we will find all the elements that have common base heights. We will multiply this with every value we get in this array.
* We will return the maximum of the three areas which we got
* Here we use divide and conquer strategy and divide the array in 2 parts every time and merging the subarrays take O(n) time
* We get a recurrence relation as . By using master’s theorem, we get

1. **SOLUTION:**
2. Here,

This represents case 1, where leaves dominate.

i.e.,

Therefore,

1. Here,

This represents case 3, where internal nodes dominate.

i.e.,

Therefore,

1. Here,

This represents case 2, where internal nodes and leaves have the same time complexity and neither dominate.

i.e., then

Therefore,

1. Here, substituting , we get

This represents case 1, where leaves dominate.

i.e.,

Therefore,

1. **SOLUTION:**
2. **Subproblem:**

* Let i be the length of the rod and pi be the dollars earned after selling the rod, 0 <= i <= N
* Let OPT[i] be the maximum amount that is achievable by strategically cutting the rod and maximizing profit.
* Our choices are as follows:
  + We do not break the rod, OPT [ i ] = pi ,
  + We break the rod into smaller pieces, MAX (OPT [ 1 ] + OPT[i-1], OPT[2] + OPT[i-2], OPT[3] + OPT[i-3], … , OPT[ i /2 ] + OPT[ i - i/2])

1. **Recurrence relation:**

* OPT [ i ] = MAX (pi, OPT[ 1 ] + OPT[i-1], OPT[2] + OPT[i-2], OPT[3] + OPT[i-3], … , OPT[ i /2 ] + OPT[ i - i/2])

1. **Pseudocode:**

maxCost(p[], int N)

int Opt[n+1]

Int Opt[0] = p[0]

for(i=1; i<= N; i++){

t =p[i]

for (j =1; j<= i/2; j++){

t = max (t, Opt(j)+ Opt(i-j)

}

Opt[i]=t

}

Return Opt[N]

1. **Base Cases and where to find final answer:**

* **Base Case:**

OPT [ 1] = p1,

OPT [ 0] = 0

* **We can find the answer at the end of the array at OPT[N]**

1. **Complexity of the solution:**

* Runtime complexity of the followingis **O(n2)**

1. **SOLUTION:**
2. **Subproblem:**

* We take an array of marbles from i to j and 0 <= i <= j<= N-1 where N is the total number of marbles.
* Let us say that OPT[i][j] is the optimal difference in the scores of the two players Tommy and Bruiny.

1. **Recurrence relation:**

* OPT[i , j] = MAX(SUM(marble[i to j+1]) – OPT[i+01,j], SUM(marble[I to j+1])- OPT[i , j-1])
* Here SUM(marble[i to j+1]) represents the sum of marbles in subarray(i, j)

1. **Pseudocode:**

Function maxPoints(marbles){

int n = marbles.length;

for (m=2; m<= n; m++){

for(i=0; i<= n-m+1; i++){

j = i +m -1;

Opt[i][j] = max(sum(marbles[i to j+1])- Opt[i+1][j], sum(marble[i to j+1])- Opt[i][j-1])}

}

Return Opt[0][n-1]

}

1. **Base Cases and where to find the final answer:**

* **Base Case:**

Opt[i][i] = 0 where i = j that is if only one marble is remaining and if it is removed then sum of 0 is obtained.

* **Final answer can be found at Opt[0][n-1].** This gives the difference In scores of Tommy and Bruiny.

1. **Complexity:**

* Time complexity for the following problem is

1. **SOLUTION:**
2. **Subproblem:**

* OPT[i] is the minimum number of members that need to be removed from the line to create a sequence with the last member having height h[i], where i is from 1 to n. We do this so that at every possible point i is the tallest possible person in the sequence

1. **Recurrence relation:**

* OPT[i] = MIN(OPT[j] +(i-j-1) where 0<=j< i and R[j]< R[i])+ MIN(OPT[j]+(j-i-1) where 1<=j< n and R[j] < R[i]) - temp

1. **Pseudocode:**

Function minMem(heights){

Int n = heights.len;

Int Opt = new Arr[]

For (i=1; i<n; i++){

Opt[i] =i-1;

For (j=0; j<I; j++) then

If(height[j]< heights[i]{

Opt[i] = min (Opt[i], Opt [j]+ (i-j-1));

}

Int temp = Opt[i];

Opt[i] = n- i;

for(j = n-1; j>=I; j--){

if (heights [j] < heights[i]) then

Opt[i] = min(Opt[i], Opt[j]+ (j-i-1));

}

Opt[i] = Opt[i] + temp;

}

Return Opt[n-1]

}

1. **Base Cases and where to find the solution:**

* **Base Case:**

OPT[0] = 0, we remove no one when there are 0 band members

OPT[1] = 0, we remove no one where there is one member in the band

* **We can find the solution at the end of the array at OPT[n-1]**

1. **Time Complexity:**

* The time complexity of the same is as we are traversing through the array of the length n twice.

1. **SOLUTION:**
2. **Subproblem:**

* Let i be the number of items and x be the weight of the item
* Let OPT [i, x] be the max value achievable using a knapsack capacity x and items i
* Our choices are as follows:
* We do not pick the item, OPT [i,x]=OPT[i-1,x]
* We pick the item and move to the next item, OPT [ i, x] = vi +OPT [i-1, x-wi]
* We pick the item and stay on the same item, xi = 1, OPT [ i, x] = vi +OPT [i, x-wi]

1. **Recurrence relation:**

* OPT [ i, x] = MAX (OPT [ i-1, x ], vi +OPT [ i-1, x-wi ], vi +OPT [ i, x-wi ])

1. **Pseudocode:**

int modifiedKnapsack(int W, int w[], int v[], int n){

int Opt[n+1 ][W+1] ;

for (i=0; i<=n; i++) {

for (x=0; x<= W; x++) {

if (i==0|| x==0) Opt[i][x] = 0;

if (w[x]>x) Opt[i][x] = Opt[i-1 ][x];

else

Opt [i][x] = max (v[i] +Opt[i-1 ][x-w[i-1]], v[i]+ Opt[i][x-w[i-1]], Opt[i-1][x]);

}

}

return Opt[n][W];

}

1. **Base Cases and where to find the solution:**

* **Base Cases:**
* OPT [ i, x] = 0, if i =0 or x = 0
* OPT [ i, x] = OPT [i-1, x], if wi > x
* **We can find the solution at OPT [n, W]**

1. **Complexity:**

* The runtime complexity for the following problem is (table size\*the work) per cell **= O (n.W) \*O (1) = O (n.W)**

1. **SOLUTION:**
2. **Subproblem:**

* We need to find the maximum coins that can be collected by bursting balloons in the correct sequence. We are interested in the last balloon burst to build up the solution and get the most optimal solution for this problem. We solve this using dynamic programming approach.

1. **Recurrence relation:**

* OPT [ i, j] = MAX (OPT [ i, j], OPT [ i, k-1] + OPT [k+1, j] + nums[i-1]\*nums[k]\*nums[j+1] )

1. **Pseudocode:**

Int maxCoinBalloon(int [] nums){

n = nums.length;

int Opt[n][n];

for ( len = 1 to n){

for( i = 0 to n){

j = len+i-1;

for( k= i to j){

int left=1, right=1, before =0, after =0;

if ( i ! =0) then left = nums[i-1];

if ( j != 0) then right = nums[j+1];

if (i!= k) then before= Opt[i][k-1];

if (j!= k )then after = Opt[k+1][j];

Opt[i][j] = Max(left\*nums[k]\*right+ before + after, Opt[i][j]);

}

}

}

return Opt[0][n-1];

}

1. **Base Cases and where to find the solution:**

* **Base Cases:**

When only 1 balloon is left and we burst the balloon and collect

(1\*coinBalloon\*1) number of coins.

* **The final answer can be found at the end of the 2-dimensional array at OPT[0][n-1]**

1. **Time Complexity:**

* The time complexity for the same isas we are finding all possible subarrays which requires and for each one them, we are finding the last balloon that needs to be burst which requires time