CSCI 570 - HW 3

1. Suppose we define a new kind of directed graph in which positive weights are assigned to the vertices but not to the edges. If the length of a path is defined by the total weight of all nodes on the path, describe an algorithm that finds the shortest path between two given points A and B within this graph.

Solution: *Given* - A directed graph *(G)* with weights on vertices.

To Do: Find a shortest path between two points A and B in this graph.

We can use Djikstra's algorithm to find the shortest path in this Graph, by modifying the input by creating an edge weighted graph from a vertex weighted graph.

We know that every path on our graph will traverse a particular vertex if it traverses an edge pointing to that vertex. We thereby modify the input to an edge-weighted graph by assigning to each the weight of the vertex it travels to.

Let S be the set of explored nodes.

We maintain the distances in an array called *dist[]*

SHORTEST_PATH_WEIGHT_ON_VERTICES(G):

1. Initially we set *S={A}*. We perform the below operations until we don't reach the destination *B*.
2. We select a node *v* that doesn't belong to *S* that has at least one edge from source A and has a minimum distance from the set of vertices processed.
3. Mark the above vertex *v* as visited and add it to *S* . We then update the distance values of adjacent nodes of the current node *v* as below:
    a. For each new adjacent node, if *dist(v) + weight(v) < dist(u),* if there is a new minimal distance found for *u,* then we *update dist[u]*
    b. If no new minimum distance is found, then we don't update *dist(u)*.
4. To store the path traversed we make use of another array *r[]* to store the path traversed as *r[v]=u.* To get the total weight of the path we make use of the *dist[]* and we add the weight on the source vertex to that in order to get the final path cost.

Time Complexity:

We initially build a heap of vertices that takes Θ(V). Then in the loop each time, we are calling deleteMin() for V nodes that gives us a time complexity of O(V logV) . We also update the dist[] in order for it to reflect the shortest distance to a destination and we run decreaseKey() that gives us a time complexity of O(ElogV). In order to output the fastest path, we traverse through r[] that has a complexity of O(V) in the worst case. Together, we have our combined complexity as O(V logV + ElogV *)*

2. For each of the following recurrences, give an expression for the runtime T(n) if the recurrence can be solved with the Master Theorem. Otherwise, indicate that the Master Theorem does not apply.

   - $T(n) = 3T(n/2) + n^2$

     Solution: For this recurrence we have, $a = 3,\ b = 2,\ c = log_2 3 = 1.584\ and f(n) = n^2$. We can apply **Case 3** of master's theorem, since $f(n) = \Omega(n^{c+\varepsilon})$ ie. n^{c} is polynomially smaller than $f(n)$ for any $\varepsilon > 0$, concluding that, $T(n) = \Theta(n^2)$

   - $T(n) = 4T(n/2) + n^2$

     Solution: For this recurrence we have, $a = 4,\ b = 2,\ c = log_2 4 = 2\ and f(n) = n^2$. We can apply **Case 2** of master's theorem, as we can see that both $f(n)$ and $n^c are$ equal concluding that $f(n) = \Theta(n^c log^{k} n)$ and thus the solution to the recurrence is $T(n) = \Theta(n^{c} log^{k+1} n)$ which is finally, $T(n) = \Theta(n^2 logn)$

   - $T(n) = T(n/2) + 2^{n} - n$

     Solution: For this recurrence we have, a=1, b=2, $c = log_2 1 = 0$ and f(n)= $2^n - n$

     We **cannot apply** Master's theorem here as *f(n)* is not polynomially larger than n^c and hence falls in the gap of case 2 and case 3.

- $T(n) = 2^n T(n/2) + n^n$

  Solution: For this recurrence we have, $a = 2^n$ and for applying master's theorem we know that a should be a constant and hence we can conclude that **we cannot apply** master's theorem.

- $T(n) = 16T(n/4) + n + 10$

  Solution: For this recurrence we have, $a = 16,\ b = 4,\ c = log_4 16 = 2\ and f(n) = n + 10$. We can apply **Case 1** of master's theorem as $f(n)$ is polynomially smaller than n^c for some $\varepsilon > 0$ and hence we can see that, $f(n) = O(n^{c-\varepsilon})$ and hence conclude that, $T(n) = \Theta(n^2)$

- $T(n) = 2T(n/2) + nlogn$

  Solution: For this recurrence we have, a=2, b=2, $c = log_2 2 = 1$ and f(n)=$nlogn$

  We can apply **Case 2** of master's theorem as n^c approximates to n^1 *which is equal to* n and hence see that both $f(n)$ and $n^c are$ equal(dropping the log terms) concluding that $f(n) = \Theta(n^c log^k n)$ and thus the solution to the recurrence is $T(n) = \Theta(n^c log^{k+1} n)$ which is finally, $T(n) = \Theta(nlog^2 n)$ where k *is assumed to be* 1.

- $T(n) = 2T(n/4) + n^{0.51}$

  Solution: For this recurrence we have, $a = 2,\ b = 4,\ c = log_4 2 = 0.5\ and f(n) = n^{0.51}$.

  We can apply **Case 3** of master's theorem as $f(n) = \Omega(n^{c+\varepsilon})$ ie. n^c is polynomially smaller than $f(n)$ for any $\varepsilon > 0$, concluding that, $T(n) = \Theta(n^{0.51})$.

- $T(n) = 0.5T(n/2) + 1/n$

  Solution: From the recurrence we have, $a = 0.5,\ b = 2\ and f(n) = 1/n$. To apply master's theorem, a needs to be ≥ 1. Hence, Master Theorem **cannot be applied** to this question.

- $T(n) = 16T(n/4) + n!$

  Solution: For this recurrence we have, $a = 14,\ b = 4,\ c = log_4 16 = 2\ and f(n) = n!$

  We can apply **Case 3** of Master's theorem as $f(n) = \Omega(n^{c+\varepsilon})$ ie. n^c is polynomially smaller than $f(n)$ for any $\varepsilon > 0$, concluding that, $T(n) = \Theta(n!)$

- $T(n) = 10T(n/3) + n^2$

  Solution: For this recurrence we have, $a = 10,\ b = 3,\ c = log_3 10 = 2.09\ and f(n) = n^2$

We can apply **Case 1** of master's theorem as $f(n)$ is polynomially smaller than n^c for some $\varepsilon > 0$ and hence we can see that, $f(n) = O(n^{c-\varepsilon})$ and hence conclude that, $T(n) = \Theta(n^{log_3 10})$

3. Suppose that we are given a sorted array of distinct integers A[1, ..., n] and we want to decide whether there is an index i for which A[i] = i. Describe an efficient divide-and-conquer algorithm that solves this problem and explains the time complexity.

   Solution: *Given* - Sorted array of distinct integers A[1,....n] and we need to check if there is an index i for which A[i] = i.

   To Do: Design a Divide and Conquer algorithm for the above problem specification.

   l - lower bound of the array A, h - upper bound(n) of array A

   CHECK_INDEX_ELEMENT_EQUALITY(A,l,h):

   1. We run a loop through where we check if $l \leq h$ and calculate the middle index by using this formula: $mid \leftarrow (l + h)/2$
   2. We check for 3 conditions once we get the middle index. Firstly we check if *A[mid]=mid,* then output *"Yes, there does exist such an index".*
   3. If the first condition doesn't match then we check for $A[mid] < mid$, if so we update l to $l \leftarrow mid$
   4. If the above two conditions don't match then, we update $h \leftarrow mid$
   5. At the end we return *"No such element exists if none of the conditions match"*, if *A[mid]=mid* condition is not met.

Recurrence Equation: $T(n) = T(n/2) + 1$

At each step we are dividing the array into two halves each time we set b=2 and f(n) is 1 because the work done at the combining step is *O(1)* is for comparison.

For the recurrence relation above, we have $a = 1,\ b = 2,\ c = log_2 1 = 0\ and\ f(n) = 1$

We can apply **Case 2** of Master's theorem as n^c is equal to $f(n)$ concluding that $T(n) = \Theta(n^c log^{k+1} n)$

which would be , $T(n) = \Theta(logn)$.

4. We know that binary search on a sorted array of size n takes O(logn) time. Design a similar divide-and-conquer algorithm for searching in a sorted singly linked list of size n. Describe the steps of your algorithm in plain English. Write a recurrence equation for the runtime complexity. Solve the equation by the master theorem.

Solution: *Given - A sorted singly linked list of size n.*

To Do: Design a Divide and Conquer algorithm to perform binary search on a singly sorted linked list of size *n.*

BINARY_SEARCH_LINKED_LIST(head_ptr, search_key):

1. We begin with initializing the start to the *head_ptr* and end to *null.* Below we perform a set of steps, until *end!=start* or *last==null.*
2. We need to find the middle element in the sorted linked list and hence for that we call another function pasing *start* and *end* as parameters to that function.
3. Within this function of finding the middle element, we check first if *start is null,* if so we return *Null* as a return value. We assign two pointers, a slow pointer that moves by 1 element, whereas a fast pointer that moves by 2 elements. Hence, when the fast pointer reaches the end, the slow pointer would point to the middle of the sorted linked list.
4. Once we get the middle element from the above function, we compare that to the *search_key.* If they match we return the middle element, otherwise if the middle element's data component $\geq$ *search_key* we update $lt\, last \leftarrow mid$. Else, if the middle element's data component is $\leq$*search_key we update start* $\leftarrow$ *mid.next*
5. If nothing matches then we return *Null.*

Recurrence Equation: $T(n) = 1.T(n/2) + O(n)$

We need *O(1)* for comparison and *O(n)* to find the middle element as a linked list doesn't have continuous memory locations.

For the recurrence we have, $a = 1,\ b = 2,\ c = log_2 1 = 0\ and\, f(n) = n$

We can apply **Case 3** of master's theorem as $f(n) = \Omega(n^{c+\varepsilon})$ ie. n^{c} is polynomially smaller than $f(n)$ for any $\varepsilon > 0$, concluding that**,** $T(n) = \Theta(n)$.

5. Given n balloons, indexed from 0 to n − 1. Each balloon is painted with a num- ber on it represented by array nums. You are asked to burst all the balloons. If you burst the balloon i you will get nums[i − 1] × nums[i] × nums[i + 1] coins. Here left and right are adjacent indices of i. After the burst, the left and right then become adjacent. You may assume nums[−1] = nums[n] = 1 and they are not real therefore you cannot burst them. Design a dynamic programming algorithm to find the maximum coins you can collect by bursting the balloons wisely. Analyze the running time of your algorithm.
   Example. If you have the nums = [3,1,5,8]. The optimal solution would be 167, where you burst balloons in the order of 1, 5, 3 and 8. The array nums after each step is: [3,1,5,8] → [3,5,8] → [3,8] → [8] → []

   And the number of coins you get is 3×1×5+3×5×8+1×3×8+1×8×1 = 167.

Solution: *Given - n balloons indexed from 0 to n-1. Array nums[] consisting of the number on each balloon.*

To Do: Develop an algorithm to find the maximum coins we can collect by bursting the balloons in a particular order.

Subproblem to be solved: The subarray of *nums[]* of every length(*starting from 1 to n)* and for every subarray we find the last balloon to burst as the one that gives maximum coins for the subarray. Once we have calculated for every subarray we find the maximum value for the entire array based on finding the maximum value at each subarray of lengths ranging from *1..n.*

We start with subarrays of length 1 and find maximum value gained on bursting that balloon. Then, we go get all subarrays of length 2 and find maximum value gained on bursting the balloon in that subarray(we will have 2 choices there to find maximum). This way we fill our table up and reuse the existing(initial subarray values) to find the global maximum value gained on bursting the balloon. The assumption is that we are going to assume the balloon that we are going to burst is the last one to bursted.

Let OPT[i,j] contain the maximum coins gained when bursting a balloon from a subarray starting from index i uptill index j.

Recurrence Relation for the Subproblems:

1. Base Cases:
    a. OPT[i,j] = 0 when j>i
    b. OPT[i][k-1] = 0 when k=i
    c. OPT[k+1][j] = 0 when k=j
2. OPT[i,j] = max(OPT[i][j], nums[i-1] * nums[k]* nums[j+1] + OPT[i][k-1] + OPT[k+1][j])

Pseudo-code for the above algorithm:

```
int maximizeCoins(int[] nums){

        int n=nums.length;

        if n==0:

                return 0;

         int OPT[n][n];

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

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

                        int j=i+len;

                        for(int k=i; k<=j; k++){

                        // We find out the numbers on the left and right of the balloon to be bursted

                                int left_number = i== 0 ? 1: nums[i-1];

                                int right_number = j == n-1 ? 1: nums[j+1];

                        // We find out the numbers on the left and right optimal value for the balloon
                        that was bursted.

                                int left_optimal = k == i ? 0: OPT[i][k-1];

                                int right_optimal = k == j ? 0: OPT[k+1][j];

                                OPT[i][j] = Math.max(OPT[i][j], left_number * nums[k ]*
                      right_number +      left_optimal + right_optimal );

                }
                }
        }
        return OPT[0][n-1];
}
```

Time Complexity: The time complexity of the algorithm is $O(n^3)$ as we can see from the pseudocode that we are iterating through three loops are needed wherein we need one loop to iterate through the entire nums array and then we need two variables to check and find subarrays of all lengths and find maximum value gained from each of those.

6. Given a non-empty string str and a dictionary containing a list of unique words, design a dynamic programming algorithm to determine if str can be segmented into a sequence of dictionary words. If str ="algorithmdesign" and your dictionary contains "algorithm" and "design". Your algorithm should answer Yes as str can be segmented as "algorithmdesign". You may assume that a dictionary lookup can be done in O(1) time.

Solution: G*iven - a non-empty string str and a dictionary consisting of sequence of words.*

To Do: Develop an algorithm to check if a string can be segmented into a sequence of dictionary words.

Subproblem to be solved: The subproblem to be solved here is to check if each prefix of the string is present in the dictionary and then we recur this for the rest of the string. If each of those prefixes are present in the dictionary we return true we say "yes" the string can be segmented into a list of dictionary words, otherwise we go ahead and try the next prefix of the string. If none of them return true then we say "No" it can't be segmented into a list of dictionary words. A prefix string[0...i] is breakable only if either the whole string is contained in the dictionary or there is a sub prefix at an index j<i such that prefix string[0….j] is breakable and substring(j+1,...i) is present in the dictionary.

Let dp[I] be true if the string *str[0….i-1]* can be segmented into dictionary words, false otherwise.

Recurrence Relation for the Subproblems:

1. Base Cases:
   a. dp[0] = true
2. dp[i] = dictionaryLookUp(str.subset(i,j-i)) + dp[current_prefix] + dp [remaining_prefix from i+1 to n]

Assumption: Denoting true for "yes" and false for "no" and the function dictionaryLookup() takes O(1) time. where in it basically check if the word passed to that function is present in the dictionary.

Pseudocode:

```
bool checkSegment(string str){

/* we create our dp table to store boolean values if prefixes can be segmented into dictionary words and we initialize
all values to false initially */

bool dp[str.size()+1];

for(int i=1; i<=str.size(); i++){

if(dp[i] == false && dictionaryLookup(str.substr(0,i)))

        dp[i]=true; //We are checking if the current prefix can help make str segmentable.

        if(dp[i] == true){

                // We would now have to check all substrings starting from(i+1)th and add their respective entries
                in the table dp[].

                if( i == size)

                        return true;

                for(int j=i+1; j<=str.size(); j++){

                        //We now keep checking if the substring from the i….j-i is present in the dictionary.

                        if(dp[i] == false && dictionaryLookup(str.substr(i,j-i)) )

                                dp[j]=true;

                        if(j==str.size() && dp[j] ==true)

                                return true;

                }

            }

         }

        return false;

    }
```

Time Complexity: The time complexity of the above algorithm is $O(n^2)$ as we need one variable to loop through the entire string char by char to check if that can be segmented into a dictionary of words or not, along with that we need another variable to check if all previous substrings from *i…….j-i* are present in the dictionary and then update the *dp[]* table.

7. A tourism company is providing boat tours on a river with n consecutive segments. According to previous experience, the profit they can make by providing boat tours on segment i is known as ai. Here, ai could be positive (they earn money), negative (they lose money), or zero. Because of the administration convenience, the local community requires that the tourism company do their boat tour business on a contiguous sequence of the river segments (i.e., if the company chooses segment i as the starting segment and segment j as the ending segment, all the segments in between should also be covered by the tour service, no matter whether the company will earn or lose money). The company's goal is to determine the starting segment and ending segment of boat tours along the river, such that their total profit can be maximized. Design a dynamic programming algorithm to achieve this goal and analyze its runtime.

Solution: Given: Profit earned on a segment i is called a_i which could be positive, negative or zero

To Do: Develop an algorithm to determine starting and ending segment of boat tours along a river to maximize total profit

Subproblem to be solved: This can be viewed as a problem of finding the maximum subarray sum where in the sum is essentially the profit gained by the boat tour. Hence, at each step we either start a new sum from index *i* or we add the existing solution till *(i-1)th* index.

Let MP[i] be the maximum profit earned by the tourism company ending at index *i.*

Recurrence Relation for the Subproblems: Let the *A[i]* contain profit for each boat segment i.

1. Base Cases:
    a. best_sum = - ∞
2. best_sum = max(current_sum+A[i], A[i])

Pseudocode:

```
getMaxProfitForTour(int A[]) {
  best_sum = - ∞ ; // Handles case for all negative weights
   start_best = end_best = 0; // We assume that we start with index 0.
   current_sum = 0;
  for current_end, x in enumerate(numbers):
                if current_sum <= 0:
               # Start a new sequence at the current element
                          current_start = current_end;
                         current_sum = x;
               else:
                # Extend the existing sequence with the current element
               current_sum += x;
                if current_sum > best_sum:
                          best_sum = current_sum;
                         start_best = current_start;
                         end_best = current_end + 1;
                         print(best_sum, best_start, best_end)
                         // This prints the maximum profit gained, along with the starting and ending segments


  }
```

Time Complexity: The time complexity of the algorithm is $O(n)$ where in we are looping through the array A containing profits once and then recursively finding out the maximum profit gained based on the current element and the previous sum found.