Homework 3

Bhanu Verma

Question 1: Prove that Pygmalion is NP-Complete

We would follow the standard procedure that was discussed in class for showing that a problem is NP-complete.

Let’s define Pygmalion first, given an undirected graph G, where nodes represent towns and edges represent roads, is there a way to build k bunkers at k different towns so that every town either has its own bunker or connected (by a direct road) to a town that does have a bunker.

***Step 1***: Show that Pygmalion is in NP

To prove that NP Pygmalion is in NP, we need to show that given a certificate, we should be able to verify if it is a solution or not in polynomial time.

Let’s say we have a certificate in the form of a list of k bunkers (k bunker nodes) and we also have the knowledge of graph G (u,v) (each town is a node and each road between two towns is an edge). Now we need to verify if each town has a bunker of its own or is connected directly to a bunker town. Here is the algorithm for checking it:

* For each town, check if has a bunker.
* If yes, go to next town.
* If no, check if every neighbor of this town (i.e. other town of every road that this town is part of) has a bunker. If one of the neighboring towns has a bunker, we check the next town. If not, we return false i.e. given certificate is not a solution.

Let’s check out the time complexity of the above algorithm. We are doing checks for each town and for each town, we need to check all it’s neighbors. Each town at maximum can have n-1 neighboring towns, where n is the total number of towns. So our time complexity becomes O(n\*n-1) i.e. O(n2). Hence, a given certificate can be verified in polynomial time.

***Step 2***: Choose a NP-complete problem. We choose vertex cover as the given problem is same as vertex cover. As know vertex cover is known to be NP-complete. We will use this problem to show that Pygmalion is NP-complete as well.

Let’s define vertex cover, given a graph G, we have a set of vertices <=k such that this set set covers each edge i.e. for each edge(u,v), either u or v belongs to the set of k vertices.

**Step 3**: Now we need to prove that Vertex-Cover problem is poly-time reducible to Pygmalion. It can be done by following steps:

* Describe a procedure that converts the inputs of Vertex Cover (problem X) to inputs of Pygmalion (problem Y) in polynomial time.
* Show that if X(i) = YES then Y(i) = YES and vice-versa

X - Vertex Cover

Y - Pygmalion

So, first step is to convert inputs of X into inputs of Y. For vertex cover, we have a graph of nodes, if we consider each node as the town, then the set of k vertices becomes our k bunkers. We see that this conversion didn’t take any extra computation time which depends on the size of n and hence can be done in constant time. This takes care of the first step i.e. we have converted inputs of Vertex Cover to inputs of Pygmalion problem. **Please note that direct connection between towns means an edge for the graph problem.**

Now, we will show that ***if X(i) = YES then Y(i) = YES*** i.e. if an input of X (Vertex Cover) is a solution then we have a solution for Y (Pygmalion) as well.

We have a set of k vertices such that for each edge, either u or v lies in the set. Let’s proof that Y(i) = YES as well using proof of contradiction. Let’s say that given set of k vertices which equates to k towns which have bunkers is not the solution for Pygmalion problem. This means that there exists a town which does not have a bunker and is not connected to any town with a bunker. Using our input conversion (town ~ nodes), this equates to the case where we have a node and that node is not in the k set of vertices and it is neither part of any edge for which the other node is in k set of vertices. This means that given k set of vertices is not the solution for Vertex Cover problem and this contradicts the fact that we started with i.e. we have a solution for Vertex Cover. Hence, we have a valid solution for Pygmalion problem as well.

Now, we will show that ***if Y(i) = YES then X(i) = YES*** i.e. if an input of Y (Pygmalion) is a solution then we have a solution for X (Vertex Cover) as well.

We have a set of k towns with bumpers such that every town either has a bunker or is connected to a town with bunker. Let’s again do this with the help of contradiction. Now as we know that k towns equate to k vertices for vertex cover problem. We say that set of k vertices is not the solution for Vertex Cover problem. This means that we have a node such that it is not in the set of k vertices and this node is not a part of any edge for which the other node is part of the set. Using our input conversion (town ~ nodes), this also means that we have a town with no bunker and it is not connected to any town with a bunker. This contradicts the fact that we started with a solution for Pygmalion problem. Hence, we have a valid solution for Vertex Cover.

Hence, it is proved that Pygmalion problem is NP-Complete.

Question 2: Help Amrita in buying the car

***Divide & Conquer Algorithm***: Before we start looking at the solution, let’s describe the problem first. We are given an array which contains a sequence of interest rates and we need to find a maximum contiguous sub-array from the given array that would give us the maximum sum.

If we think intuitively, there can only be three cases that will give us the result while we are splitting up our array into equal parts:

Case 1: Maximum sum sub-array will come from left half

Case 2: Maximum sum sub-array will come from right half

Case 3: A piece of maximum sum sub-array will be come from the left and other piece will come from the right. This means that middle point will be contained in the final result.

So our divide and conquer algorithm can be broken down into following steps:

* Find middle point and split array into two halves using that middle point.
* Recursively split left half till we reach the base case and get the final sum for left half, called left\_sum.
* Recursively split right half till we reach the base case and get the final sum for right half, called right\_sum.
* Find the maximum sum from the sub-array overlapping both the halves, called overlapping sum.
* Find the maximum sum from these three values and return that sum while moving up.

A minor addition is required to this approach to return indices; we’ll need to return indices as well along with sum while we are moving up in our recursive algorithm.

***Time Complexity***: T(n) = 2T(n/2) + O(n), using master theorem this comes out to be O(n\*logn).

***Space Complexity***: No extra space required apart from making recursive stack calls, so from a computation perspective it is O(1).

***Dynamic Programming Algorithm***: a DP approach for the given problem has already been provided in the homework itself. Let’s have a look at this approach.

For a given day j, maximum interest rate that Amrita can obtain up till that day is either 0 i.e. sum of interest rates up till j was negative or sum of interest rates till j-1 day plus interest rate for j. Let’s write that down in terms of recurrence relation:

S[j] = 0 , if j=0

= max(S[j-1]+aj, 0) , otherwise

From an implementation perspective, I have used a bottom-up approach instead of top-down. For the bottom-up approach, we would calculate the ‘*maximum\_sum\_ending\_here’* for each index and use this value for calculating the ‘*maximum\_sum\_so\_far’* at each iteration. This way we would need only two variables that we can use throughout our iterations and can get the result in one pass.

A minor tweak to the algorithm is required to find the indices for the contiguous array which gives us the maximum sum. Our *temp\_starting* index will be updated each time our current element is more than the *maximum\_sum\_ending\_here + nums[i]*, which means for any element if our *maximum\_sum\_ending\_here* becomes negative then we update our *temp\_starting* index.

Now, whenever our *maximum\_sum\_ending\_here* becomes greater than or equal to *maximum\_sum\_so\_far*, then we set our actual starting index to temp\_starting index and end index to current index.

***Time Complexity***: As we are getting the result in one pass, our time complexity is O(n).

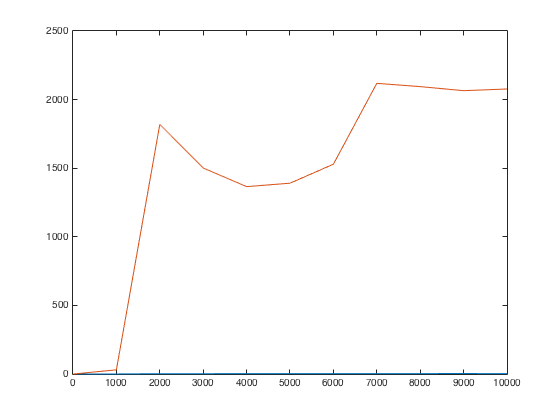
**Space Complexity**: Constant space, hence O(1).

**GRAPH PLOTTING**: Dynamic Programming (Blue) vs Divide & Conquer(Orange)

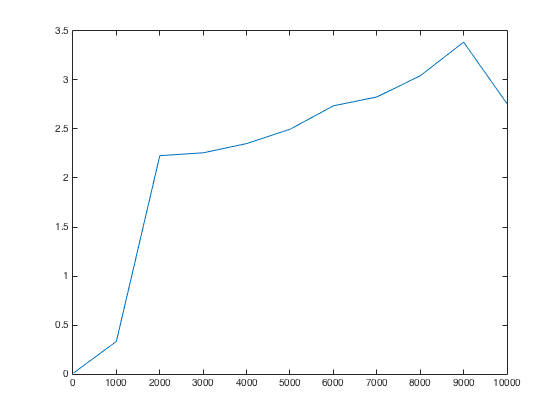
As you will see in the graph below, plots for the graphs for both the algorithms are as expected. As divide and conquer algorithm is O(n\*logn), it’s time complexity increases as the value of n increases and that increase is more than the increase for dynamic programming algorithm’s time as time for DP algorithm is linear. This is the reason we are not able to see the time for DP clearly on the graph as time for dp << time for dc.

Y Axis – Time in milliseconds

X Axis – Number of instances



Here is the graph plot of time for dynamic programming algorithm vs number of instance and it is linear as we expected it to be:



As we can see from the plots above that DP algorithm is a lot faster in comparison to Divide & Conquer approach and the difference in time keeps on increasing as we keep on increasing the problem size.

Please note that there are minor anomalies in the shapes of the graph and they are not exactly as their time complexity would suggest. This is only due to the fact that we don’t have enough observations. If we run this algorithm for sufficient number of times over small change in values of n, then graph shapes would smoothen.