Lab 3- Wakanda and Knapsack problem

Dhruv Sandesara

DJS3967

Algorithms

Part1:

For the program to find the amount of resources to be allocated to each project to get the maximum growth, the logic is as follows:

We build an array of size projects*number of Vibranium. Each row is the first I projects considered to which the resources may be allocated to. The n rows denote how much Vibranium is allocated the the projects. And the max size will be the input project numbers and total vibranium ores. Looking at these end values will return us the value for max growth given the number of projects and the vibranium units.

We initialize the array by setting the growth values of 0-i projects given 0 vibranium to 0. We also initialize the growth given 0 projects and 0-n vibranium to 0. Then look at the first project and store the value of growth given 1-n vibranium. We then move on the next row. Over here when computing the max growth value for 2 projects given j vibranium, we compute all combinations of 0-j vibranium to the 2nd project and the remaining vibranium to the first one and find the max growth value and store it for that index value. This way we find the value of the maximum growth of all combinations of resources allocation between the 2 projects. We continue doing this with the remaining projects by computing all combinations of resource allocation to the next added project and the previous most optimal solution. This can be denoted by Optimum(N,V) which is the optimum deployment for n projects and V vibranium as

Optimum(N,V)= Max(given i=0->N)(Growth by project N given I resouurces + + Optimum(N-1, V-I))

This problem has optimal substructure as it computes the optimal solution at every stage and uses it to compute the next best optimal state as described above.

Next thinking about the how to recreate the optimal solution of distribution of the Vibranium to each project. My though is that for the array that we use to store the maximum growth number, instead of using integers, we use nodes which have one field as the max growth value and other fields as the number of resources allocated to each project. And then each time we compute the next optimal solution we use the values from the previous best solution that we found and add the remaining resources for the project of which row we are working on. This way at each step not only are we keeping track of the optimal solution but also how the resources are being distributed.

Part 2:

For the Knapsack problem with multiple constrains of volume and weight my optimal solution is as follows:

For the recursive definition is as follows Optimum(I,W,V) is optimum price stolen given first I items, W weight constrain and V volume constrain.

Opt(I,W,V)= Opt(I-1,W-I's weight,V-I's volume) and we do this for each item for the entire array from the items own weigh& volume to the maximum weight and volume constrains.

My Algorithms: We first initialized an array of size W and V which are the weight and the Volume constrains and each value of the array gives the maximum stolen price for those constrains. We then iteratively consider each item and consider all values from the maximum constrains to the items own constrains. We then compute the maximum value of the current entry and the sum of the items price + entry of current constrains — the current item's constrains. When we do this iteratively, we get the optimal price at each

constrain given the first I items. And thus adding each item just has to measure the correct combination of the current item and the previous optimal solution. Thus this problem has optimal substructure.

To recreate the optimal solution we can tweak the array by instead of storing just the int max price, we store a node with fields of max price and the currently considered set. And this way when we are recomputing the optimal solution, we just add the current item to the set of the previous set of the optimal stolen items. Thus it works.

Testing:

The approach to testing that I took was mostly to use the given test cases and check with other classmates of Piazza whether the solution is correct. Other than the given test cases, I did change the input files to have interesting cases as all 0 input values, only 0 Volume constrains(which turns problem into the only one constrain knapsack problem) and vice versa. Also I checked for the case when none of the items fit in the bag and I got expected results in all of them. I although did check for unusual inputs such as negative weighs and volumes which broke the code but I did not bother fixing it as they are unexpected inputs. Thus I believe that I have provided adequate coverage of the implementation. There was one interesting bug I had found when testing my knapsack problem. This was arising as I was overwriting the previous optimal solution from the items constrains to the maximum constrains. This was giving me problems as when it came to constrains twice the capacity of the item, the algorithms attempted to store the item twice. This was fixed by me by just going the other way from max to mix constrains as this removed the case of overwriting values and using them again.