|  |
| --- |
| **(S1-21\_DSECLZG519)**  **(Data Structures and Algorithms Design)**  **Academic Year 2020-2021** |
| **Assignment 2 – PS15 - [Fastfood Joint] - [Group 255]** |
| |  |  | | --- | --- | | Smitha K A | 2021fc04616@wilp.bits-pilani.ac.in | | VITHARANA GAMAGE THARINDU GAYAN | 2021fc04612@wilp.bits-pilani.ac.in | | RICHIN GUPTA | 2021FC04611@wilp.bits-pilani.ac.in | |

## Algorithm

This problem falls into the category of 0|1 knapsack problem. Greedy and divide and conquer approaches cannot be used to find the global optimal solution for this kind of problems. Hence, we must consider all possible solutions (brute force approach) to obtain the global optimal solution. To do that we could use iterative or recursive methods. As instructed, we have used recursive approach to solve this problem along with dynamic programming techniques (memorization) used to address the overlapping sub problems.

### The recursion tree:

In each level of the tree, we

* Either select a particular preparation or,
* Ignore and go to next

This creates two sub trees /recursive calls at each node/call of the tree/function as shown in the recursion tree below. When merging the results, we select the sub problem with the highest profit and return its profit, cost and selected preparations list as a tuple. If both sub problems have the same profit, then we choose the one which yields in maximum utilization of funds with more preparations(variety in food).

Selectt

Ignore

Select

Ignore

Selectt

Ignore

### Base case:

The first base case of the recursion is either no fund balance is left, or all preparations are considered for the selection. In this case we return zero profit.

Since we are using “memoization” to address overlapping subproblems, the second base case is, this sub problem is already solved. Here we return the solution from the stored memory.

### Recursive calls:

If the base cases are not satisfied, then call the two recursive function calls one with selecting the current preparation and other ignoring the current preparation. Then based on the results of those recursive calls we return the profit, cost and selected preparations list as a tuple.

## The data structure model

This problem is solved using recursive algorithm(as instructed) with overlapping sub problems. By using recursive approach **stack** is being used implicitly for the recursive function calls.

### Input details

We have used lists to store the input details to the problem. Since we are not doing any time-consuming operations like searching or sorting etc. on the input data instead of just iterating through them, lists are the simplest and best data structure for this purpose.

self.preperation\_list: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

self.cost\_list: [3.0, 1.7, 2.0, 1.0, 1.3, 1.0, 1.6, 2.5, 1.5, 1.8]

self.profit\_list: [6.0, 3.5, 5.5, 4.0, 6.6, 2.0, 3.5, 5.0, 7.0, 1.0]

### Memoization table

Some kind of key, value pair storing data structure should be used to store the intermediate results which will cut off the repetitive work done when overlapping subproblems occurs. Python dictionaries are the ideal candidate for this kind of data structures where we could look up a given key in O(1) time complexity. However, its instructed to not to use them and hence we used two lists to store the key and value pairs. Below figure demonstrates how mapping between key and value is achieved using the two lists.

|  |  |
| --- | --- |
| **Index** | **Value** |
| 0 | Value\_1 |
| 1 | Value\_2 |
| 2 | Value\_3 |
| … | … |

|  |  |
| --- | --- |
| **Index** | **Value** |
| 0 | Key\_1 |
| 1 | Key\_2 |
| 2 | Key\_3 |
| … | … |

self.mem\_indices

self.mem

key

Index

Value

Here to get the index of the value in the self.mem list, we have to get the index of the “key” in self.mem\_indices list. This takes O(n) time if the length of the list self.mem\_indices is “n”. Since the key and the value are stored in the same index of the two lists, we can directly get the value using the index of the key. This is an O(1) time operations.

#### Key generation:

The key for a stored result is generated using below python code

mem\_key = f"{n}\_{round(total\_fund,1)}"

Here “n” and “total\_fund” are the remaining no: of preparations and remaining fund in a given function call.

The total no: of keys that could be generated depends on the no: of preparations, the total fund value, and the no: of decimal places (precision) of these values.

So, the space complexity of the memorization table would be O(n\*W\*d), where;

n - no: of preparations

W - the total fund value

d - no: of decimal places (precision) of profit and cost values

##### Assumption

Here we have assumed that profit and cost values have a single decimal place and created the key string according to that assumption.

## Time complexity analysis

If dynamic programming with memoization was not used (ie: brute force recursion) the time complexity would be 2n (exponential) where n is the no: of preparations. By using the dynamic programming with memoization we are limiting the no: of recursive calls, up to a maximum of n\*W\*d, where;

n - no: of preparations

W - the total fund value

d - no: of decimal places (precision) of profit and cost values

In each recursive call the time intensive task would be the key look up in the list self.mem\_indices. Since the space complexity of the self.mem\_indices is O(n\*W\*d) as described above, the time complexity of the lookup is also O(n\*W\*d).

Therefore, the overall time complexity can be derived as below,

T = (# recursive calls) \* (time complexity of the key lookup)

= O(n\*W\*d) \* O(n\*W\*d)

= O(n2\*W2\*d2)

where;

n - no: of preparations

W - the total fund value

d - no: of decimal places (precision) of profit and cost values

## Alternate approach

The no: of decimal places (precision) of profit and cost values is not pre-defined in the problem statement. If it was pre-defined then, we could formulate a function to generate the index value of the self.mem list. This function would generate a key in O(1) time and we would be able to improve the performance of memory look up operation from O(nWd) to O(1).

Furthermore, we can initialize the memoization table with some specific value (say -1) to check whether if that is updated or not. If a particular value is not “-1”, means its already solved subproblem. This is O(1) operation compared to the O(n\*W\*d) lookup we are doing in each recursive call in the implemented program.

Hence if we know the no: of decimal places (precision) of profit and cost values beforehand then we could use this alternate method and improve the time complexity to O(n\*W\*d)\*O(1) = O(n\*W\*d)