Practical Work 2

Algorithms 1

Federal University of Minas Gerais
Department of Computer Science
Henry Tamekloe
henrany@ufmg.br

INTRODUCTION:

The problem at hand is to implement an algorithm for a game called "**the game of diamonds".** This problem is to practice the knowledge gained from programming paradigms learnt form class specifically dynamic programming which is an optimization approach of solving problem. The idea of dynamic programming is to optimize a problem in which the normal approach of using brute force will take a longer time before execution. But there is a price to pay for dynamic programming as we trade space for time.

PROBLEM:

A goldsmith by the name Adriana, who after receiving a bag of diamonds, do the same operations over and over before she creates her jewels.This is what she does, assuming a collection of diamonds of different weights, two diamonds of say p1 and p2 can be combined based on some operations that she does. First she compares the two stones p1 and p2 and if p1 is equal to p2, she destroys both stones but if p1 is greater than p2, she subtracts the weight of p2 from p1 and return the weight to her collection and destroy p2. She does this operation until there is at most a stone left in her bag. The objective of doing this is to get the minimum weight possible between all the combination of the stones, so choosing of the stones should be optimal.

DEVELOPMENT ENVIRONMENT:

The problem was developed mainly in c++ using the object oriented programming as is the best way of organizing the code for better readability. The compiler used is the GCC compiler

To compile the program, run ***>make*** and to execute the program, you should have a file with all the test cases to run against and type ***>./tp2 tests.txt***

DATA STRUCTURES AND ALGORITHMS USED:

The data structure and algorithm used to solve this problem is array and also using the dynamic programming paradigm. We have a brute force approach of solving the problem and also a dynamic programming paradigm of solving it also. For the brute force, recursion was used to solve for the problem. Below is the pseudocode of the brute force approach

```
bruteForce(array, n, set1, set2):
    If n is empty:
        return abs(set1 - set2)
    Endif
    include = bruteForce(array, n-1, set1 + array[n],set2)
    exclude = bruteForce(array, n-1, set1, set2 + array[n])
    return min(include , exclude)
```

For the dynamic approach, we store each solved problem in a matrix for later consultation. The bottom up approach was used where each new problem depends on the lower calculated problem. Below is the pseudocode for the dynamic approach.

```
dynamicApproach(array, n):
    Sum = 0
    For I in range n:
        Sum = sum + array[i]
    Endfor
    lookUp[n+1][sum+1] = false
    For I in range n:
        lookUp[i][0] = true
        For j in range (1, sum) and i>0:
            lookUp[i][j] = lookUp[i-1][j]
            If j >= array[i-1]:
                lookUp[i][j] |= lookUp[i-1][j-array[i-1]]
            Endif
        Endfor
    Endfor
    J = sum/ 2
    While j>=0 and lookUp[n][j] is true :
```

Decrement j
Endwhile
return sum - j*2
End

IMPLEMENTATION:

An array was created for getting the entries of the program. The implementation was done using classes in c++ to organize the code. The class Diamonds declares a number for the total entry size of the array and an array to store the weights of the elements.

The public class has the following functions Diamonds(with parameter), addToVector(index, value), bruteForce(n, set1, set2), dynamicApproach(n), print(n), and ~Diamonds().

Diamonds(int number) is the constructor used to initialized the values. The number in the private class is initialize by the number passed in the constructor. The array diaArray was initialized to zero.

addToVector was parameterized with the index and the value to be stored in the array.

BruteForce was used to solve the problem using the recursive form. The idea behind the recursive solution is, divide the array into two and then consider each element in the array one at a time. Where in this case, there are two possibilities, that is, we add the element we are looking at to the first set and do a recursion for the remaining elements or not to add the element in the first set but then add this element in the second set and do recursion for the remaining elements in the array. We then return the minimum difference we get by including current item in the first set and the second set. At the end of this, we will have sum of the first set and sum of the second set and when there are no more elements left in the array, we return the absolute difference between the two sums. This solution considers all the possible combination and returns the optimal one.

The dynamicApproach was based on the recursive approach whereby each new solution calculates the solution form scratch.

Here we use the bottom up approach where we solve the problem form the basis to the top. So we start off by calculating the sum of all the element in the array. We then initialize a boolean matrix lookUp where we will store the solution of the subproblems. We initialized it with false values. All elements with sum 0 is true as this is the absolute minimum that we will get. We then loop through again, and remove the i'th element in the array. We then check whether the element in our original array is less than some index j of sum, if so, we then include this in the lookup array. This is then until we have filled the whole array. The sum of the array is then divided into two and assign it to the a value j and then scan through the lookup array. This is done to get the maximum value of j between 0 and sum/2 where the last row is true. The solution is then returned by the subtracting the double of j from the sum. This will then give us the optimal solution. This was a variation of the knapsack problem where we include ou not include an item

print is used to output the results of the dynamic approach.

The main function is where we call the functions for execution. The addToVector called first for adding elements into the array and then we call the print function to calculate the results. The runtime of the program is also calculated.

TIME COMPLEXITY:

The time complexity of is the maximum bound of the program. It is found by analyzing each function(n indicate the number of elements in the array). The function constructor initialized the array to be used to 0. So the time complexity here is O(n).

The function addToVector takes a constant time to add values to the array

The function BruteForce takes an exponential time O(2^n) to get the answer. But this wont be necessary since we wont call it in the main function. This function is for experimental analysis only.

The function dynamicApproach takes O(n*sum) as time complexity where sum is the total sum of all the elements in the original

array. This is due to running a two loops for insertion of elements into the the auxiliary matrix created for storing sub problems already solved.

The print function just calls the a single function so its time complexity is the same as of the dynamicApproach which is O(n*sum).

The main function reads the number of element and the elements to be stored in a constant time O(1). And after that calls the function addToVector and adds the elements into the array in a linear time O(n). Then call the print function which takes O(n*sum) time complexity.

So the total time complexity of the whole program is the maximum time complexity of a function. This is indicated by Max(O(n), O(1), O(n*sum) = O(n*sum). So the time complexity of the program is O(n*sum).

Now the space complexity of the program is the space saved for storing of elements. This can be seen by the diaArray which stores the elements. Its space complexity is O(n) which is linear. The other saved space is the lookUp where we store all the subproblems solutions generated. It takes up a space of O(n*sum) where sum is total sum of all the elements in the diaArray. So the space complexity of the program is the Max(O(n), O(n*sum) = O(n*sum).

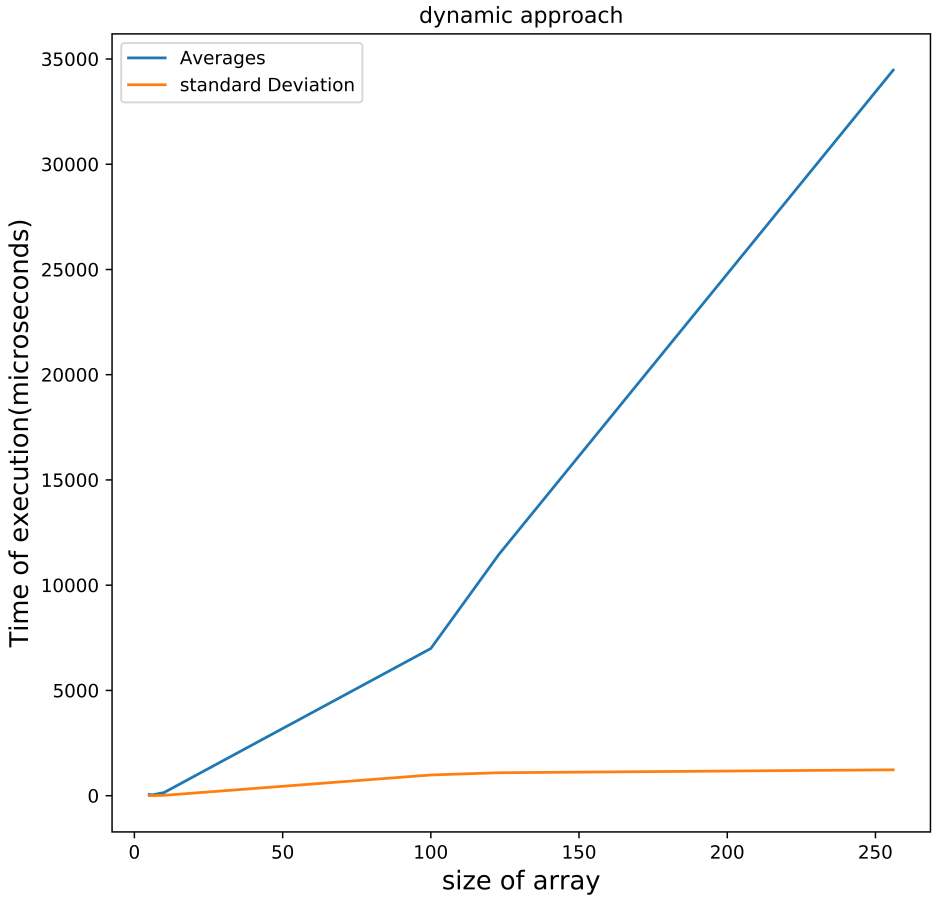
EXPERIMENTAL ANALYSIS:

To check the comportment of our program, we run some tests cases to find out if our program is working correctly. And doing so, we recorded the time taken for the program to run based on the number of elements in the array. The average and standard deviation was also calculated. Below is a table of runtime of the program of the dynamic approach method in microseconds.

Number of elements	Run 1	Run 2	Run 3	Run 4	Run 5	Average	Standard Deviation
5	54	62	51	53	55	55	4.1833

Number of elements	Run 1	Run 2	Run 3	Run 4	Run 5	Average	Standard Deviation
6	37	34	36	36	31	34.8	2.3875
10	141	174	138	140	140	146.6	15.3558
100	7997	6022	8009	6069	6853	6990	981.8915
123	11998	12171	9548	11644	11970	11466.2	1089.0905
256	36402	33820	34975	33504	33642	34468.6	1227.6888

From the table it, can be seen that, the runtime of the program increases with increase in the number of element in the array. This is, increase in number of element increases the total time taken. It can also be seen from the table that, the time complexity of the program also depends on the size of each element in the array. As increase in size of each element will result in increased total sum. This can be seen between the the array with 5 and 6 elements as the array with 5 elements has on average a higher execution time than the array with 6 elements due to the weight of the each element in the array. This means that the runtime is proportional to the size of the array and the weight of each element in the array. We plot a graph to illustrate the behavior of the above affirmation.



Visualization of average and standard deviation

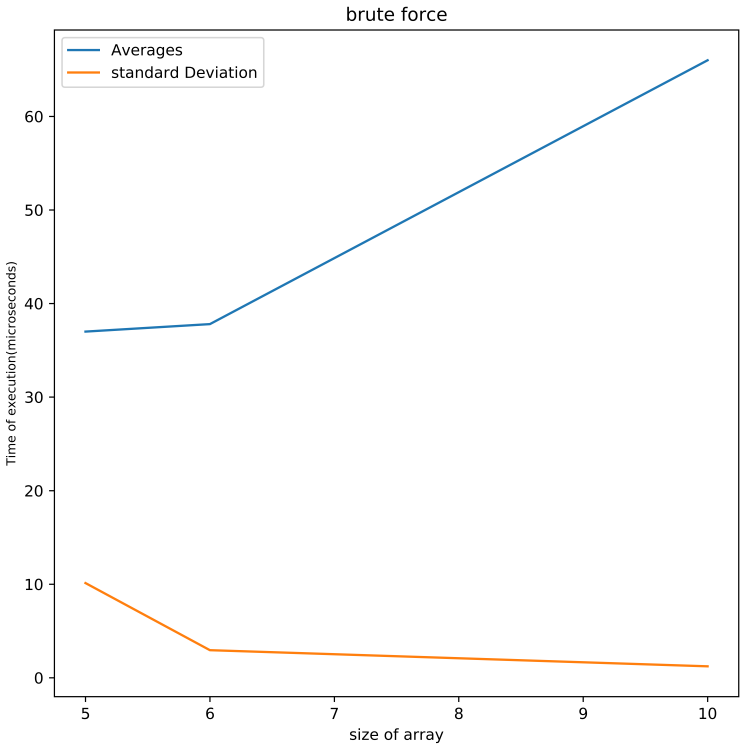
From the graph, it can be seen that the runtime increases with increase in size.

We would also like to check the variation of the dynamic approach to the brute force approach implemented and visualize if our time complexity is correct.

Number of elements	Run 1	Run 2	Run 3	Run 4	Run 5	Average	Standard Deviation
5	30	52	31	29	43	37	10.1241
6	33	41	38	38	39	37.8	2.9496
10	68	65	66	65	66	66	1.2247

Number of elements	Run 1	Run 2	Run 3	Run 4	Run 5	Average	Standard Deviation
100	>10 minutes						
123	>10 minutes						
256	>10 minutes						

From the table using the brute force approach, it can be seen that, the runtime of the program increases exponentially as the number of elements in the array increase. We could see that, the run time jumps when the entry in more than 35 as a lot fo recursive calls are made and also the same results are recalculated for different problems. This can be seen in the diagram below



Visualization of brute force approach

It can be seen from the graph that, the runtime increases exponentially with increase in the size of the array.
From here, we can really conclude that the dynamic approach solves the problem faster than that of the brute force approach but with the exception of extra memory usage.

CONCLUSION:

At the end of it all, the problem at hand brought new insight about how to solve problems optimally and understanding how to use already solved subproblems to solve new problems. Dynamic programming at first is a bit exhausting to grasp, but with a lot of practice, it can really improved how we think about problems.

REFERENCIA:

Algorithm Design , By Kleinberg, Eva Tardos & Iva Tardos
GeeksforGeeks https://www.geeksforgeeks.org/dynamic-programming/