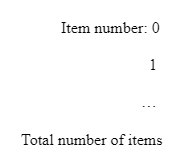
**Exploring Knapsack Problem and Usage of Different Algorithms**

The program implements three different algorithms for solving the knapsack problem; Implemented algorithms are: dynamic programming, breadth-first branch and bound, and best-first branch and bound.

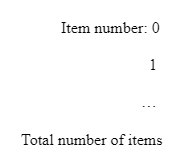
**Dynamic Programming**

For the dynamic programming algorithm, we construct a matrix that has dimensions of a number of items + 1 by total weight limit + 1. The reason why we include +1 is that we consider the cases starting from 0 until the total number of items and until the weight limit. Overall, the graph looks like this (three dots represent additional values until the weight limit and the total number of items):

Weight: 0 1 2 … weight limit 

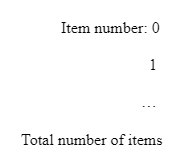
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

To start filling the table, items should be ordered in nonincreasing order according to their ratio of profit/weight. Our code sorts the array of items, so the items on inputted file do not have to be ordered. Afterward, we store the values at each of the optimal solutions at each square. Let’s start with the simple case when weight and item numbers are 0. In this case, the output or the values of the matrix would also be 0 because in 0-weighted knapsacks no amount of object can be fitted that has a profit.

Weight: 0 1 2 … weight limit 

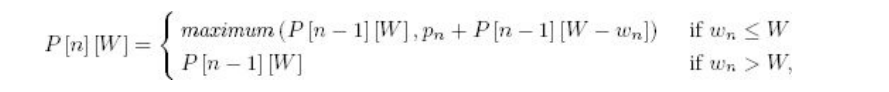
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 0 | 0 |
| 0 |  |  |  |  |
| 0 |  |  |  |  |
| 0 |  |  |  |  |

Then we start filling the matrix, while only considering cases at the same line and above. We start at item 1. Suppose weight of item 1 is 2 and the profit is 4, in this case, we cannot have knapsack weight 1 and have this item in it, so we input 0 for the [1][1] in the matrix. Then, we can fit item 1 at the knapsack which can carry a weight of 2, so for the [1][2], we put the profit value of the item. As we increase the weight capacity of the knapsack, the profit value stays the same, so we keep rewriting the profit value for the other [1][n] squares for the matrix.

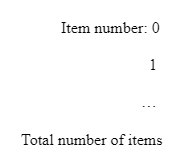
Weight: 0 1 2 … weight limit 

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 4 | 4 | 4 |
| 0 |  |  |  |  |
| 0 |  |  |  |  |

On line 2, we still keep the pattern: only consider items in line 2 or above. The way to determine whether or not to input a new item combined with the other items or leave the previous one is by using the following formula from the book:



Input the value of the maximum of these two numbers: value at the row above to the current value or the combination of maximum profit/weight valued item in the row above + profit of the new line. The algorithm works because items are ordered, thus, as we keep filling the table, and keep going bottom and right, we get the optimal profit.

Weight: 0 1 2 … weight limit 

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 4 | 4 | 4 |
| 0 | 0 | 4 | 6 | 6 |
| 0 | 0 | 4 | 6 | 7 |

I just inputted the random numbers in the matrix to show that the element in the last row and column would give the maximum profit. Thus, the program outputs the element of the last row and the column.

Furthermore, we need to output the elements which were chosen for the knapsack problem. To do it, we do the following: Starting from the element from the last row and the last column, if the element equals the element above it, then we check the element, locating right to that element. If there was an occurrence when an element at a certain location does not equal the element above it, we decrement the weight of it to the total weight limit and store the item index in the ArrayList.

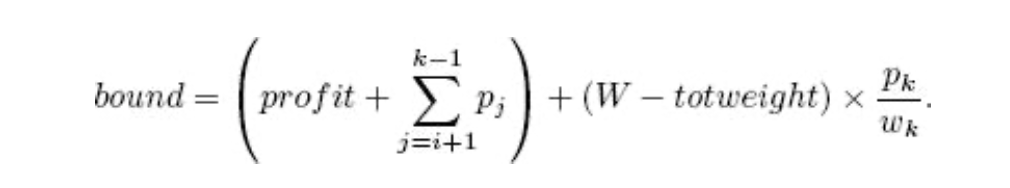
Once we know the indexes of the objects, we retrieve their weights and sum them up, so that we can get the optimal weight.

**The time complexity of Dynamic Programming**

In the greedy method, we fill two arrays with item weight and profit. This takes a time of 2n. Afterward, we fill the matrix that was described above. We use a nested for loop to do it. One goes from 0 to n and one inside goes from 0 to m, where m is the maximum weight limit, making up the time of mn. Afterward, we print the matrix, but because it is only for testing purposes, we can neglect the time the printing takes. Then, we store the indexes of the used items for which we use the while loop. The worst-case scenario, when all the items are being used, would result in this algorithm running at nm time. Finally, we take the items and add their weight up. Making the time complexity linear. Overall, if we consider that m to be any constant, 2n+n+n+n = 5n. Thus, the time complexity is O(n).

**Breadth-First and Best-First Branch and Bound**

Because these two algorithms have very similar implementations, we will be going through the only Breadth-first branch and bound method and then we will be exploring the differences between these two implementations.

The breadth-first method starts by declaring and initializing ArrayList of items, queue q, root node u, maximum profit, and optimal weight variables. Afterward, the root node is enqueued in the queue. Then, we go through the while loop which takes us to the deepest node of the tree. While exploring the tree, we keep track of profit and weight. If during the exploration of branches, we exceed the value of already stored profit and weight limit, we update profit and weight until we get the maximum values and all of the nodes of the tree would be checked. To calculate the best case profit at a particular node we use the bound method using the next formula:

We check if the sum of all items is equal to maximum profit, otherwise, we delete the element second to last because it was not a necessary element.

The best-first branch and bound have the same implementation as the breadth-first algorithm, but instead of using an ordinary queue, we use a priority queue, which prioritizes items based on their bounds, calculated by the same formula without any changes. In other words, a priority queue helps us to explore only the children of promising nodes to get the solution. However, since we do not build the whole tree, the optimal solution is not guaranteed.

Overall, best-first results give us an approach to find a solution fast, even though it might not be optimal; however, breadth-first solution results in the optimal solution while taking more time.

**Time Complexities of Breadth-First and Best-First Branch and Bound Methods**

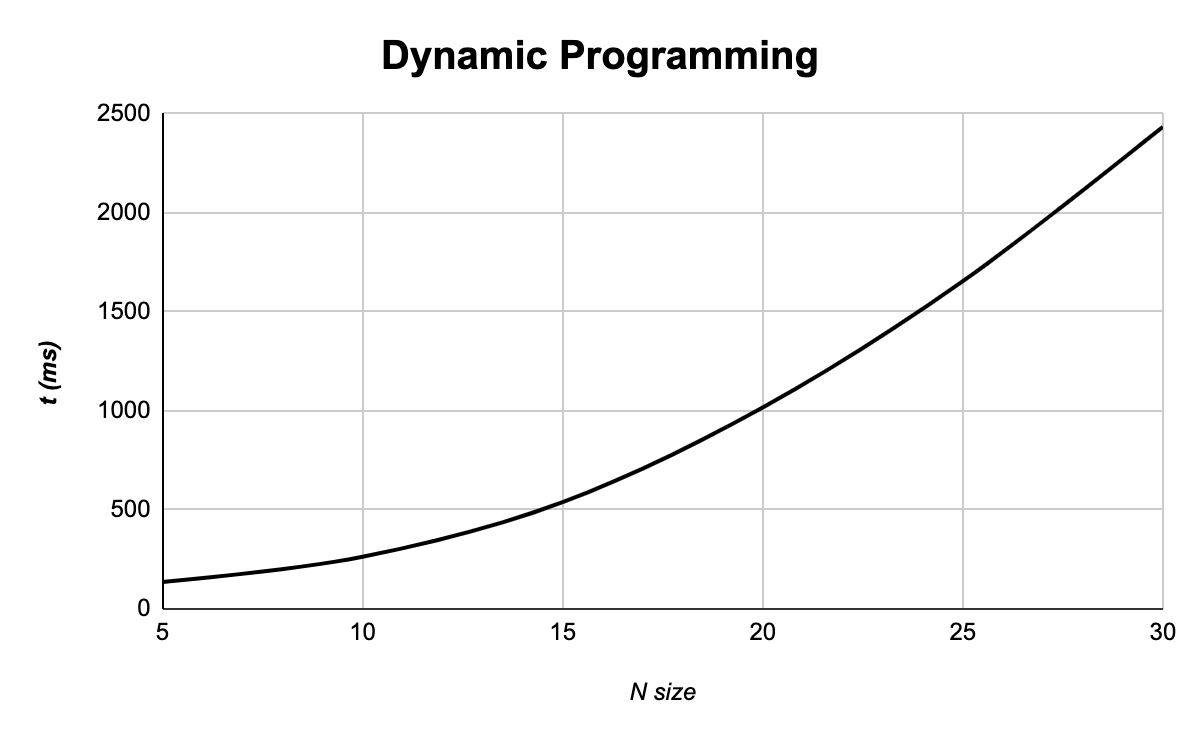
The breadth-First method goes through every possible child of the tree and chooses the best solution out of it. Because the number of children of the tree and input size n (number of items) is exponentially correlated, time complexity O(2n). We construct the binary tree, having a depth of n.

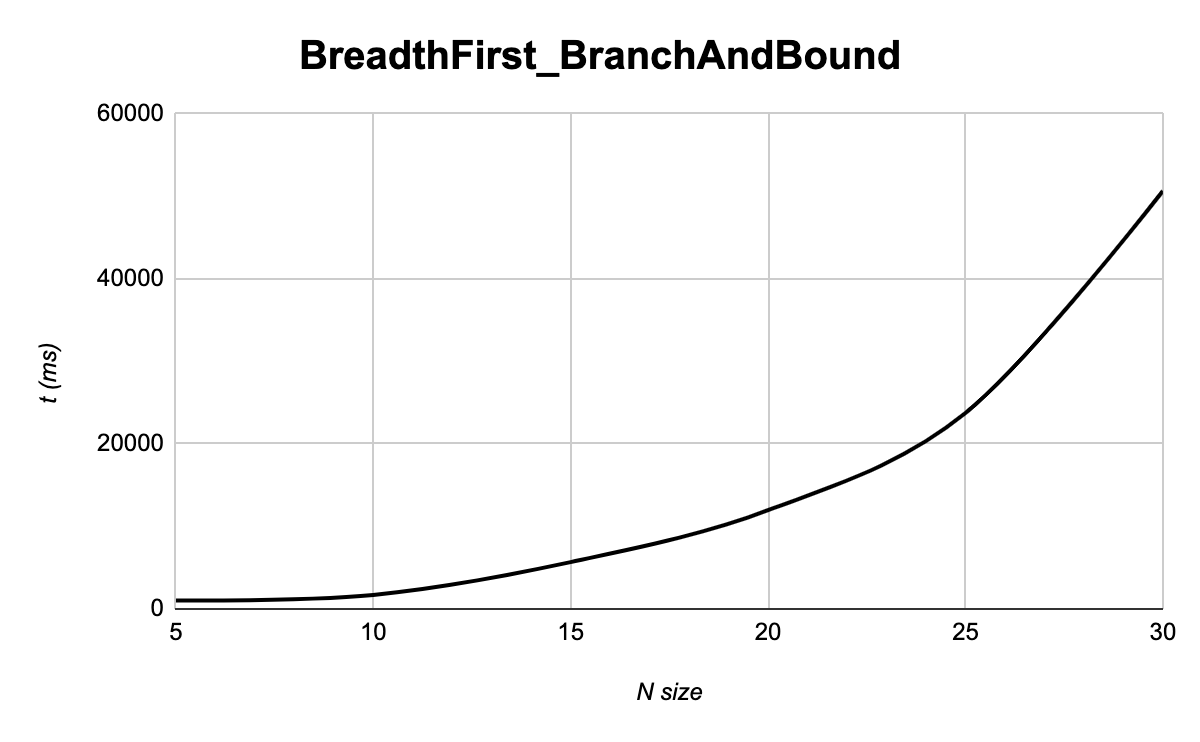
Because best-first only goes through the promising nodes, the number of nodes explored would be the same as the number of n (inputted items). Thus, the time complexity would be linear: O(n).

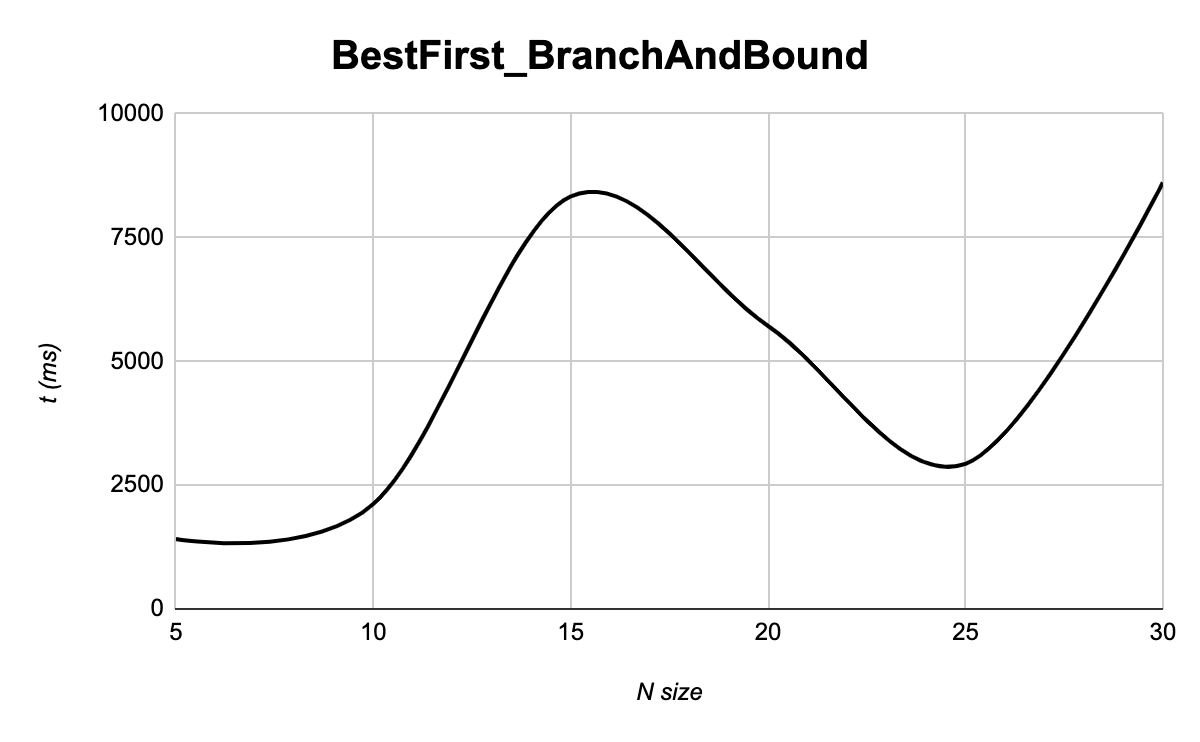
**Tested Time Complexities**

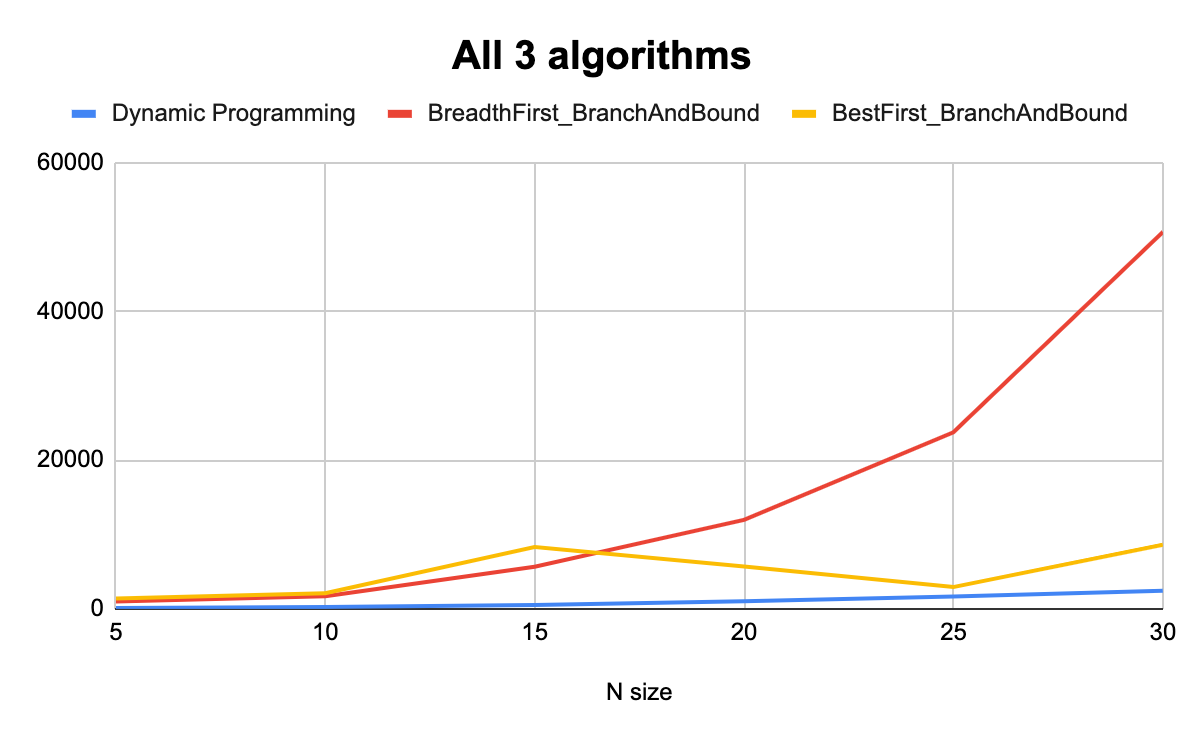
We measured the time run of all algorithms, calculating the difference between system time at the end and at the start of each of the algorithms. You can see the results in the table below:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **N size** | **Weight** | **Dynamic Programming** | **Breadth First Branch and Bound** | **Best first Branch and Bound** |
| 5 | 100 | 136 | 1010 | 1411 |
| 10 | 200 | 263 | 1689 | 2117 |
| 15 | 300 | 539 | 5673 | 8330 |
| 20 | 400 | 1017 | 11959 | 5715 |
| 25 | 500 | 1655 | 23743 | 2931 |
| 30 | 600 | 2435 | 50658 | 8613 |

As we expected, the time complexity growth for the dynamic programming algorithm is linear. It doesn’t look perfectly linear at this scale because we also change the maximum weight. 

Same with the breadth-first branch and bound algorithm. The growth is exponential. 

However, the time complexity for the best-first branch and bound algorithm is unpredictible.



**Interesting modifications and findings**

1. **K variable (Profit multiplicator)**

Branch and bound algorithms do not work properly with small numbers (tiny fractions). To solve this problem we increase all profits in K times (100000 as default), but then divide the found profit by K to have a proper answer.

1. **Smart ArrayList of collected items**

If to find the final weight we can just update it exactly in the same way we update the final profit, the list of collected items needs a more serious approach. After adding an item to the list, we check if the sum of all items is equal to all profits. If it is not equal, we delete the item before the last one, which was a node cancelled due to a better solution. After that the list of collected items is right, and the sum of their profits is equal to the maximum profit. Another approach would be to store ArrayLists of items in each node.

1. **Double vs Float**

We decided to use double for bounds instead of float, but there is no difference using the instances provided in class and made by our own.

1. **Stable growth of the time run?**

The real time run (not time complexity) seems to be unpredictable in the best-first search branch and bound algorithm. Aside from that, breadth-first one behaves faster with small instances, but becomes very slow with the larger number of n. The best-first one behaves very fast when it comes to big instances (compared to the former), but sometimes fails to give an optimal solution. In any case, dynamic programming would be the most optimum algorithm considering its time complexity and the observed time run.

**Resources Used**

Bari, A. (2018, February 20). *4.5 0/1 Knapsack - Two Methods - Dynamic Programming*. YouTube. https://youtu.be/nLmhmB6NzcM.

Bari, A. (2018, February 26). *7.2 0/1 Knapsack using Branch and Bound*. YouTube. https://youtu.be/yV1d-b\_NeK8.

Bari, A. (2018, March 4). *4.5.1 0/1 Knapsack Problem (Program) - Dynamic Programming*. YouTube. https://youtu.be/zRza99HPvkQ.

Neapolitan, R. E. (2015). *Foundations of algorithms*. Jones & Bartlett Learning.