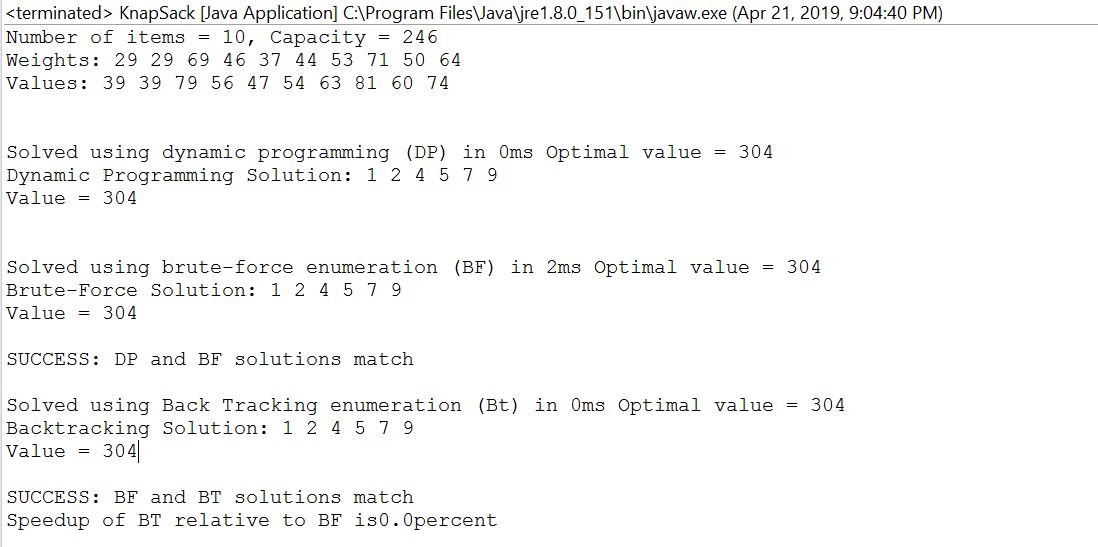
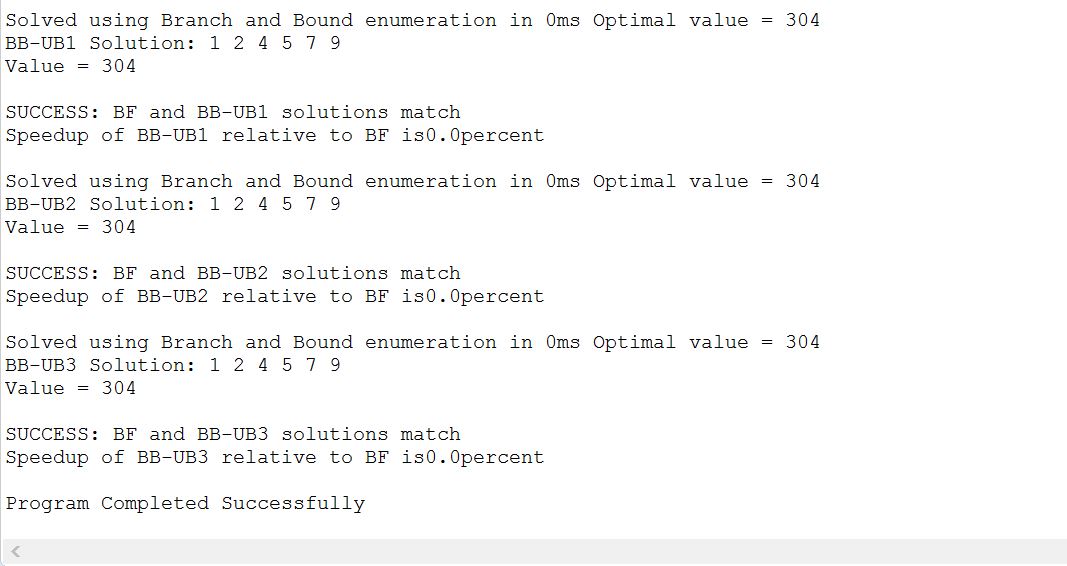
**Result Table :** ms = Milisecond, s = Second, m = Minute

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Brute Force | Backtracking | B&B  UB1 | B&B  UB2 | B&B  UB3 | Dynamic Programming |
| N = 10 | 2ms | 0ms | 0ms | 0ms | 0ms | 0ms |
| N = 20 | 109ms | 57ms | 24ms | 17ms | 3ms | 1ms |
| N = 30 | 63.9s | 42s | 3.11s | 2.297s | 42ms | 6ms |
| N = 40 | Timeout | Timeout | Timeout | 14.24m | 365ms | 12ms |
| Largest Input  Solved in 10s | 27 | 28 | 31 | 32 | 60 | java.lang.OutOfMemoryError |

****

****

**Brute Force :**

* Brute Force takes 2^n time to run for n inputs. That’s why according to upper table, it takes time in millisecond when input size is 10. However, when the input size is increased to 20 and 30, it takes seconds and minutes respectively to run on. Moreover, when the input size is 40, it goes timeout means it will take hours to run on. And in 10 seconds it can solve around 27 items. However Brute-Force is working fully exponentially.
* While Brute Force algorithm checks for the infeasible value of weights at the leaf node. And ignores weights which are infeasible. That is why when the input size will increase it will take hours or thousand of hours to run on, which is practically infinity, That is why it is Brute Force.

**Backtracking :**

* Backtracking is an efficient way than Brute Force to run algorithm faster. This is efficient than Brute Force because Backtracking is a method which firstly checks for the sum of taken weights and current weight which are more than the capacity and ignores it if the sum of the taken weights and current weight are more than capacity. However, Brute Force will ignore infeasible weights at the leaf node but Backtracking method will do that in any node which will be greater than capacity. It means whenever taken weights sum will go greater than capacity, It will backtrack to the parent node of the child node and takes other undecided weights into consideration. This will reduce the number of operations to reach the final solution.
* That is why it will take fraction of a second for the small inputs, but for big inputs it will take minutes and hours to run on as like as Brute Force. However we are tracking the total taken weights and untaken weights, it will reduce the complexity from O(n) to O(1). And in 10 seconds it can solve around 28 items.

**Upper Bound 1 :**

* Upper Bound is a technique to make algorithm run faster than Backtracking. In this technique we are doing the same like backtracking to track the sum of taken weights and values and untaken values. But in this technique, we are assuming that all the items that are yet to be decided will be taken. So, we are performing aggregate function to find out Upper Bound 1 on the all taken values and all undecided values. As we are tracking just taken values and untaken values, So, we will subtract sum of untaken values from the total sum of values. If this value will go less than or equal to the best value found so far, we will return it.
* It means we are pruning the tree if the best value will not find out for the undecided item. Hence, we are pruning the tree more efficiently than we are pruning in Backtracking by comparing sum of taken values and undecided values to the best value found so far.
* Now this technique will take O(n) time to find the sum of taken values and undecided values if we will keep the track of it by using the for loop, But in the program we are keeping the track of it in a function like TakeItem(), DontTakeItem(), UndoTakeItem() and UndoDontTakeItem(). So, that we can compute the Taken values so far, Untaken values so far in a O(1) time. Hence, by using it we can see in out output table that it is taking fraction of a second for the small inputs and for big input like 30 and 40 it is taking seconds & hours respectively to find the best solution for the given inputs. But this is an efficient than the Backtracking. And It will solve 31 items in 10 seconds.

**Upper Bound 2 :**

* This is a same technique as Upper Bound 1 but with small difference. The difference is in the Upper Bound 1 we are assuming all item that are undecided will be taken, But in Upper Bound 2 we are assuming that Every Undecided item will be taken if they will fit to remaining capacity. For that we are also keeping the track of the remaining capacity. Like if the Undecided item’s weight is less than or equal to the remaining capacity, we will take that item and we will subtract that item’s weight from the remaining capacity and after that we checks for the best solution as like as we did it in Upper Bound 1. If best solution is less than current solution then we will take current solution, otherwise we do Undo Take then we will add that weight to the remaining capacity.
* Likewise, we do not need to traverse the whole tree to find the best solution, we just need to traverse the nodes that are feasible with our condition. However, we can decrease the number of operations by applying this condition. And it will prune the infeasible solution’s node or tree very efficiently than Upper Bound 1.
* You can see that by showing the upper table output, which shows for the larger input this technique is more efficient than Brute-Force, Backtracking, Upper Bound 1. For the input 40, Brute Force is taking hours to find the best solution, Upper Bound 2 will do that in just number of seconds. And in 10 seconds it can solve around 32 items.
* However, we are tracking the remaining capacity it will take more time to compute than Upper Bound 1 i.e. Upper Bound 1 takes O(1) time and Upper Bound 2 takes O(n) time to compute. But Upper Bound 2 allows us to do pruning in more efficient manner than it does in Upper Bound 1, So Upper Bound 2 is tighter bound than it has in Upper Bound 1.

**Upper Bound 3 :**

* In Upper Bound 3 we are using a concept of fractional knapsack to make the algorithm faster. First step of this technique is that we are sorting the items using the answer of values divide by weights in the descending order. After that we will use Fractional knapsack and Fractional knapsack is whenever remaining capacity is the part of the current item weight then we will take a fraction of that item’s weight which will completes the remaining capacity. By doing this we will take that item first which has more value/weight ratio. Hence, it will always give us the best solution. After applying this we will check whether fractional value and current item value is greater than best solution or not. If not, then we will return it. Hence, by doing this we can find the solution in a faster way.
* Now, if we sort those items inside the loop which we are using for finding solution, then it will take O(n^2) time to compute. Because sorting will take O(n) time and FindSoln will take O(n) time, it will lead the time to O(n^2). But to compute it in O(n) time we will sort the items before we start to find the solution. And in 10 seconds it can solve around 60 items.
* So, by seeing the table we can definitely say that Upper Bound 3 is more efficient than Brute-Force, Backtracking and Upper Bound 1 & 2. Because by applying fractional knapsack concept will prune the tree more efficiently, Because, we are already sorting the items in the best value manner. That’s why we do not need to worry about the best solution, because we are traversing the tree in the best value (MAX of value/weight) manner. Hence it will disregard the sub-tree or nodes in very efficient manner.

**Dynamic Programming :**

* Dynamic Programming is faster and efficient among above four technique because it has a complexity of O(nC) and here C is not fully exponential and also Dynamic Programming is not a polynomial, it’s a pseudo polynomial. That’s why it is taking very less time to compute the large inputs. But it is not useful when we have a very large number of inputs. Likewise, we are having a Out of Memory error when we are giving the very large input that will solve in just 10s as in out table. So, whenever we have a very large number of inputs then we have to solve it using above four technique.
* But for the specific amount of inputs it will work faster than any other technique which we used above. It’s because Dynamic programming is a bottom up technique and it is using two-dimensional array to store the solution found so far and it will use for loops for the items and capacity while capacity is not exponentially, that makes the algorithm faster and it will increase the efficiency of pruning. Hence, for the specific amount of inputs Dynamic Programming is much faster than above techniques, But for very large inputs it will give Out of Memory Error.
* From the table you can see that, for the input size 10, 20, 30 & 40 it taking very less time like in Milliseconds while other techniques is taking it in Milliseconds, Seconds, Minutes & hours to run on.