Dixita Bhanderi

0-1 Knapsack problem solving using dynamic programming, brute force and branch and bound techniques

Submission

Submit on Canvas your code and a report in which you list and discuss the results. The report should include the following:

1. **The above table with your results in it. If you do extra credit, add a column showing the results of your best B&B algorithm.**

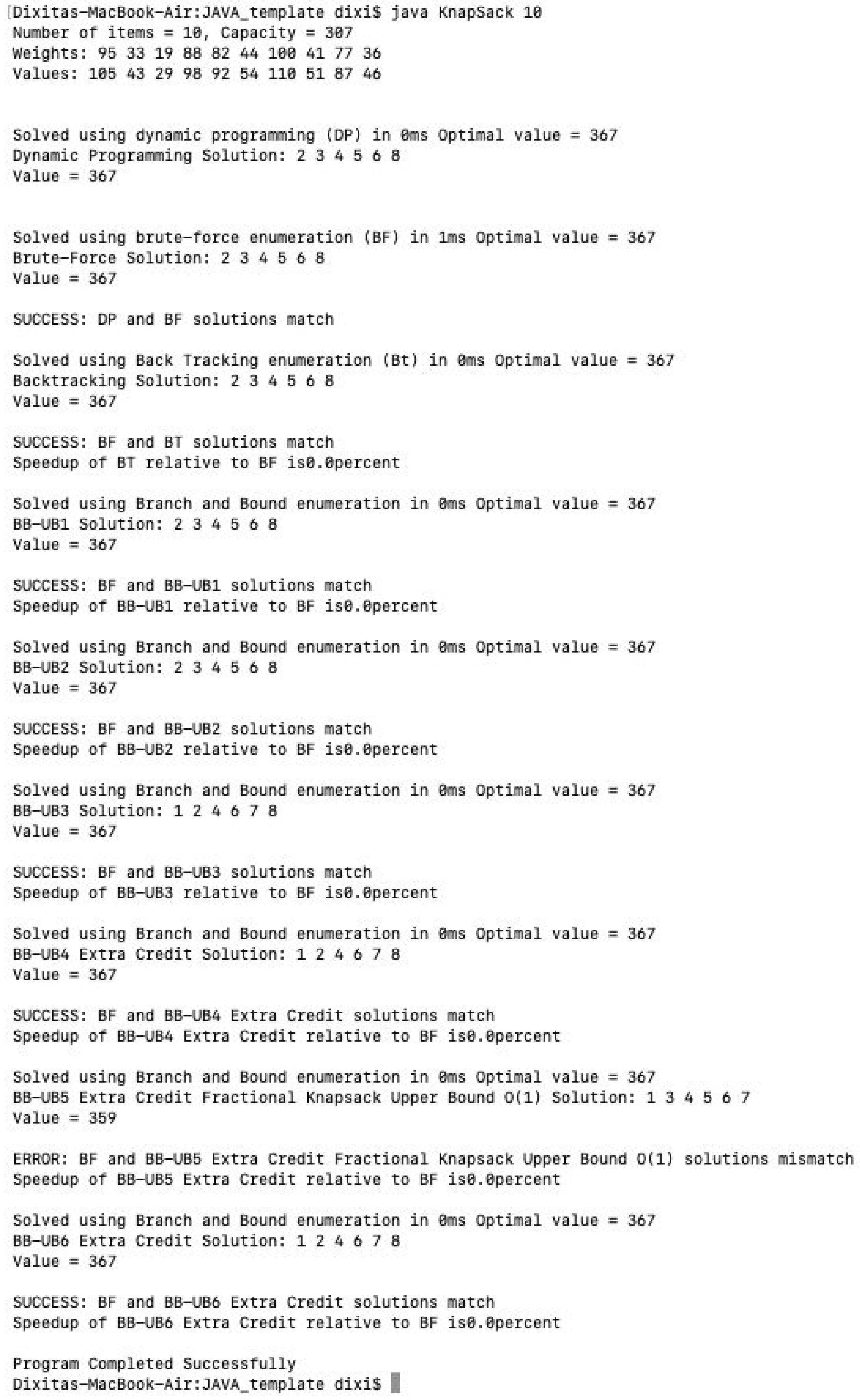
	Brute Force	Backtracking	B&B UB1	B&B UB2	B&B UB3	Dynamic Programming	B&B UB4	B&B UB5	B&B UB6
N = 10	1ms	0ms	0ms	0ms	0ms	0ms	0ms	0ms	0ms
N = 20	83ms	20ms	10ms	8ms	0ms	1ms	18ms	3ms	14ms
N = 30	65638 ms	7590ms	703ms	508ms	0ms	3ms	4940 ms	6ms	4446 ms
N = 40	More than 30min	More than 30min	886300 ms	844343 ms	0ms	5ms			
Largest Input Solved in 10ms	N = 16	N = 18	N = 20	N = 23	N = 500	N = 85	N = 19		N = 20

Note: For the N = 40, I set the timer for 30 mins and Brute Force and Backtracking exceeded the timer.
B&B UB4, B&B UB5 and B&B UB6 I have done for extra credit but B&B UB4 and B&B UB6 could not get better performance than B&B UB3 and B&B UB5 It could not get correct solution sometimes gives solution mismatch sometimes and some times gives correct solution.
Here, Blank cells implies I did not run that implementation for those values of N.

2. **A screenshot of the output for input size 10 showing a *match* between each solution and the brute-force solution. Note that the program will print the actual items taken in the solution for n=10.**

Dixita Bhanderi

0-1 Knapsack problem solving using dynamic programming, brute force and branch and bound techniques



```
[Dixitas-MacBook-Air:JAVA_template dixi$ java KnapSack 10
Number of items = 10, Capacity = 307
Weights: 95 33 19 88 82 44 100 41 77 36
Values: 105 43 29 98 92 54 110 51 87 46


Solved using dynamic programming (DP) in 0ms Optimal value = 367
Dynamic Programming Solution: 2 3 4 5 6 8
Value = 367


Solved using brute-force enumeration (BF) in 1ms Optimal value = 367
Brute-Force Solution: 2 3 4 5 6 8
Value = 367

SUCCESS: DP and BF solutions match

Solved using Back Tracking enumeration (Bt) in 0ms Optimal value = 367
Backtracking Solution: 2 3 4 5 6 8
Value = 367

SUCCESS: BF and BT solutions match
Speedup of BT relative to BF is0.0percent

Solved using Branch and Bound enumeration in 0ms Optimal value = 367
BB-UB1 Solution: 2 3 4 5 6 8
Value = 367

SUCCESS: BF and BB-UB1 solutions match
Speedup of BB-UB1 relative to BF is0.0percent

Solved using Branch and Bound enumeration in 0ms Optimal value = 367
BB-UB2 Solution: 2 3 4 5 6 8
Value = 367

SUCCESS: BF and BB-UB2 solutions match
Speedup of BB-UB2 relative to BF is0.0percent

Solved using Branch and Bound enumeration in 0ms Optimal value = 367
BB-UB3 Solution: 1 2 4 6 7 8
Value = 367

SUCCESS: BF and BB-UB3 solutions match
Speedup of BB-UB3 relative to BF is0.0percent

Solved using Branch and Bound enumeration in 0ms Optimal value = 367
BB-UB4 Extra Credit Solution: 1 2 4 6 7 8
Value = 367

SUCCESS: BF and BB-UB4 Extra Credit solutions match
Speedup of BB-UB4 Extra Credit relative to BF is0.0percent

Solved using Branch and Bound enumeration in 0ms Optimal value = 367
BB-UB5 Extra Credit Fractional Knapsack Upper Bound O(1) Solution: 1 3 4 5 6 7
Value = 359

ERROR: BF and BB-UB5 Extra Credit Fractional Knapsack Upper Bound O(1) solutions mismatch
Speedup of BB-UB5 Extra Credit relative to BF is0.0percent

Solved using Branch and Bound enumeration in 0ms Optimal value = 367
BB-UB6 Extra Credit Solution: 1 2 4 6 7 8
Value = 367

SUCCESS: BF and BB-UB6 Extra Credit solutions match
Speedup of BB-UB6 Extra Credit relative to BF is0.0percent

Program Completed Successfully
Dixitas-MacBook-Air:JAVA_template dixi$ 
```

Dixita Bhanderi

0-1 Knapsack problem solving using dynamic programming, brute force and branch and bound techniques

N=10

Note : In BB-UB3, BB-UB4 and BB-UB6 Solution set differs than the other solution set because in them Knapsack Instance array is sorted by descending order of their value-to-weight ratios initially. Hence, indexes of the original values-weight pair changes and prints here differently. However, if we map the values of that indexes with original knapsack instance than they matches.

3. **A discussion of your results. Your discussion should explain the results reported in the table.**

Dynamic Programming:

To implement Dynamic Programming problem, I used the pseudo-code given in class.
DP uses more space, but it computes knapsack solution faster compare to other approaches here except UB3(Fractional Knapsack way). Dynamic programming works well if the knapsack capacity is a reasonably small integer.

In DP, the problem is decomposed into smaller problems and find a relation between the structure of the optimal solution of the original problem and the solutions of the smaller problems and we use the older calculated value. Hence, the performance is improved as we are not recalculating previously calculated partial solutions and but directly take that older calculated value.

DP is giving the faster performance compare to Brute Force and other approaches Back tracking, BB-UB1, BB-UB2 for smaller values of n. It performs 85 items within 10ms. and 10-15% compare to the BB-UB1. In above table we can see that for n = 30 also, Brute force is 65638ms, BT is 7590ms, UB1 is 703ms, UB2 is 508ms and DP is 3ms.

Brute Force Approach:

For Brute Force problem solution was given to us. In this approach, it is calculating each and every possibility(2^n) and compare with a best solution so far. It is taking more time as the input size increases. Running time increases exponentially. It computes the optimal solution in 2^n time. It is 2^n because either we take the item or we don’t take.

The maximum number of leaves we will get is 2^n in which it also contains the combinations of items which are exceeding the capacity. Apart from this, we need to traverse the whole path to reach the optimal solution. As this approach considers each and every possibility of the knapsack solution, complexity of this algorithm is O(2^n).

As it is an exponential approach, Brute Force gives slowest performance than any other approaches mentioned here. As the number of items increases it get slower exponentially. Here, for N = 30, running time was 65638ms and for N = 40 it exceeded the time of 30min it could be 1hour also or more than that for just N = 40.

Dixita Bhanderi

0-1 Knapsack problem solving using dynamic programming, brute force and branch and bound techniques

Backtracking Approach:

It is better approach as compare to the Brute force, where we do not proceed towards the leaf if the total weight of selected items is exceeds the capacity. Here we check feasibility (going to fit in capacity) and infeasibility (not fit in the capacity) of solution on early stages. If we are getting infeasibility solution on early stages then we are not proceeding toward leaf we backtrack in that condition. Hence we are saving a lot of computations.

I modified the brute force implemented in such a way that it checks the current solution total taken weight if it exceeds the capacity than we backtrack and prune that subtree. This does not allow early pruning of subtree and goes deeper in the tree comparitively. Hence it gives slower performance than other Upper Bound pruning techniques BB-UB1, BB-UB2 and BB-UB3.

By this approach, we get less number of leafs than 2^n and it allows early pruning of the subtree. Hence, several paths will not be transverse and that is why it gives faster performance than the brute force approach. It gives better performance than the Brute Force for any value of N. For N = 40 it exceeded the timer of 30min

Upper Bound 1:

Here Upper Bound 1 is the addition of The **sum of taken item values and undecided item values.** As described in class, I have compute this Upper Bound 1 at each node by subtracting the sum of untaken item values from the sum of all item values. The sum of untaken item values I have computed incrementally by updating this sum at each node and its complexity is θ(1).

UB1 approach is better than Backtracking. In this approach, we are calculating the upper bound at each node and if it does not exceed than the best solution so far, we prune that subtree. Following is the formula I am using to calculate the Upper Bound 1:

UB1 = **SumOfTotalValues – SumOfNotTakenValues**(O(1)) = Taken Item Values (current solution value so far)+ total value of Undecided Items

Here, To get total value of Undecided Items in linear time, **SumOfTotalValues – SumOfNotTakenValues**(O(1)) this formula is used, as it will compute that value in O(1) time by just subtracting. **SumOfTotalValues** I am computing at the beginning itself, while generating the instance and **SumOfNotTakenValues** I am computing incrementally at every node. This way, it is all computed in O(1) time.

If UB1 is greater than best solution than we proceed further with that subtree for more calculations of those nodes, otherwise we prune it. Such pruning of whole subtree saves the operations and so as time to find the optimal solution than the backtracking or brute force methods.

This is improving the performance drastically about 90% compare to Brute Force. It takes θ(1) time to calculate upper bound at each node and it gives less number of leaves in the tree relative to BackTracking approach even. In above table we can see that for n = 30 also, Brute force is 65638ms, BT is 7590ms but UB1 its 703ms.

Upper Bound 2:

Upper Bound 2 is the **sum of taken item values and the values of the undecided items that fit** in the remaining capacity at each node. As described in class, I have computed this in O(n) time at each node by checking all the remaining items and adding the value of every item whose weight is less than or equal to the remaining capacity.

In this case, I consider all the remaining undecided items that fit in the remaining capacity. Here, this computation take O(n) time as I have to traverse through the remaining items to check whether each of them individually fits in the remaining capacity or not. Following is the formula I am using to calculate the Upper Bound 2:

UB2 = Taken Item Values (current solution value so far)(O(1)) + Total value of Undecided Items that fits(O(n))

UB2 gives tighter upper bound than UB1, as we are adding a constraint in the undecided items value consideration. In UB1 we use total value of Undecided Items whereas in UB2 fit constraint is added to the Undecided Items. This way we dont consider the items that does not fit in the remaining capacity and hence it gives smaller total of values than in the case of UB1 making UB2 tighter. Hence, generally, UB2 allows early pruning of subtree as compare to UB1.

If UB2 is greater than best solution than we proceed further with that subtree for more calculations of those nodes, otherwise we prune it. Such pruning of whole subtree saves the operations and so as time to find the optimal solution than the backtracking, brute force methods and also the UB1 in most of the cases.

In some cases UB1 may give better performance than UB2, Reason behind is, if the all the remaining undecided items are the one which fits in the remaining capacity, than UB2 computation time will exceed than UB1 computation time, and if same thing happens at each remaining nodes or most of the remaining nodes than BB - UB2 algorithm takes somewhat more time than UB1 algo because UB2 computation time is O(n). However, it is a rare possibility.

UB2 is improving the performance by 95% compare to Brute Force and 10-15% compare to the UB1. It takes θ(n) time to calculate upper bound at each node but it allows early pruning and it gives less number of leaves in the tree relative to Brute Force approach or backtracking. In above table we can see that for n = 30 also, Brute force is 65638ms, BT is 7590ms, UB1 is 703ms and UB2 is 508ms.

Upper Bound 3:

In Upper Bound 3 I solve the remaining sub-problem at each node as a Fractional Knapsack problem to use as a pruning technique. As described in class, this is computed in O(n) time at each node as I have sorted the items by descending order of their value-to-weight ratios before start of the search (in the preprocessing step).

For the pruning condition, I add the fractional Knapsack solution for remaining undecided items to the value of current node and if is less than or equal to the best solution value so far than I prune than subtree. Fractional Knapsack smaller/comparatively tighter upper bounds. Hence allows the subtree earliest at very shallow level of the tree. This way without traversing at the deep level pruning of the subtree is done.

Here, running time at each node for the computation could be O(n), but as it is pruning the subtrees early that n would be small. As we go deeper in the subtree that n gets smaller and smaller in lower depths as remaining capacity will decrease at each taken node. As UB1 is computed in O(1) time and UB2 computation time is O(n), in UB3, UB3 is tighter upper bound than the UB1 and UB2, hence despite of O(n) (n would be small most of the time as we go deep in tree) time spent at each node to compute Fractional Knapsack solution this approach gives faster performance than UB1 and UB2 because of the smaller/tighter upper bounds and overall computation time is smaller.

UB3 is improving the performance drastically by 300-400% approx as compare to Brute Force. As mentioned in the table above, it gives faster performance even with larger values of n. For n=500 solution is calculating with 10ms time. UB3 performance also better than any other technique namely, UB1, UB2, DP, Backtracking. For Backtracking, UB1 and UB2 maximum number of items computed within 10ms are 18 and 20 respectively which is far less than 500 items within 10ms. It takes O(n) time to calculate upper bound at each node but it allows very early pruning and it gives far less number of leaves in the tree relative to other approach. In above table we can see that for n = 30 also, Brute force is 65638ms, BT is 7590ms, UB1 is 703ms, UB2 is 508ms and UB3 is 0ms.

4. **If you do the extra work, describe your additional enhancements, including any pruning technique that you have added. Then explain your results.**

Extra Credit BB-UB4: Modified Upper Bound 2 approach.

In this, as I am calling the take item method before the don't take item method, I get the best solution for take items call with UB2 Upper Bound. After that I have put another upper bound for don't take items, so if we don't take item than remaining sum of values should greater than the best solution which will prune the subtree and will save the recursion calls. But this approach did not performed well against the BB-Upper Bound 3 method.

Source: Similar approach I discussed with one of my classmate, it differed in the way that approach was appling this pruning technique while calculating the take item calls and it was for UB 1, I modified it this way for UB2 modification.

Extra Credit BB-UB6: Modified Upper Bound 3 approach.

Similarly, In this modification, I put the same condition before Item don't take calls. Once, I get the best solution for take items call with UB3 Upper Bound. After that I have put another upper bound for don't take items, so if we don't take item than remaining sum of values should greater than the best solution which will prune the subtree and will save the recursion calls. But this approach did not performed well against the BB-Upper Bound 3 method.
Source: Similar approach I discussed with one of my classmate, it differed in the way that approach was appling this pruning technique while calculating the take item calls and it was for UB 1, I modified it this way for UB3 modification.

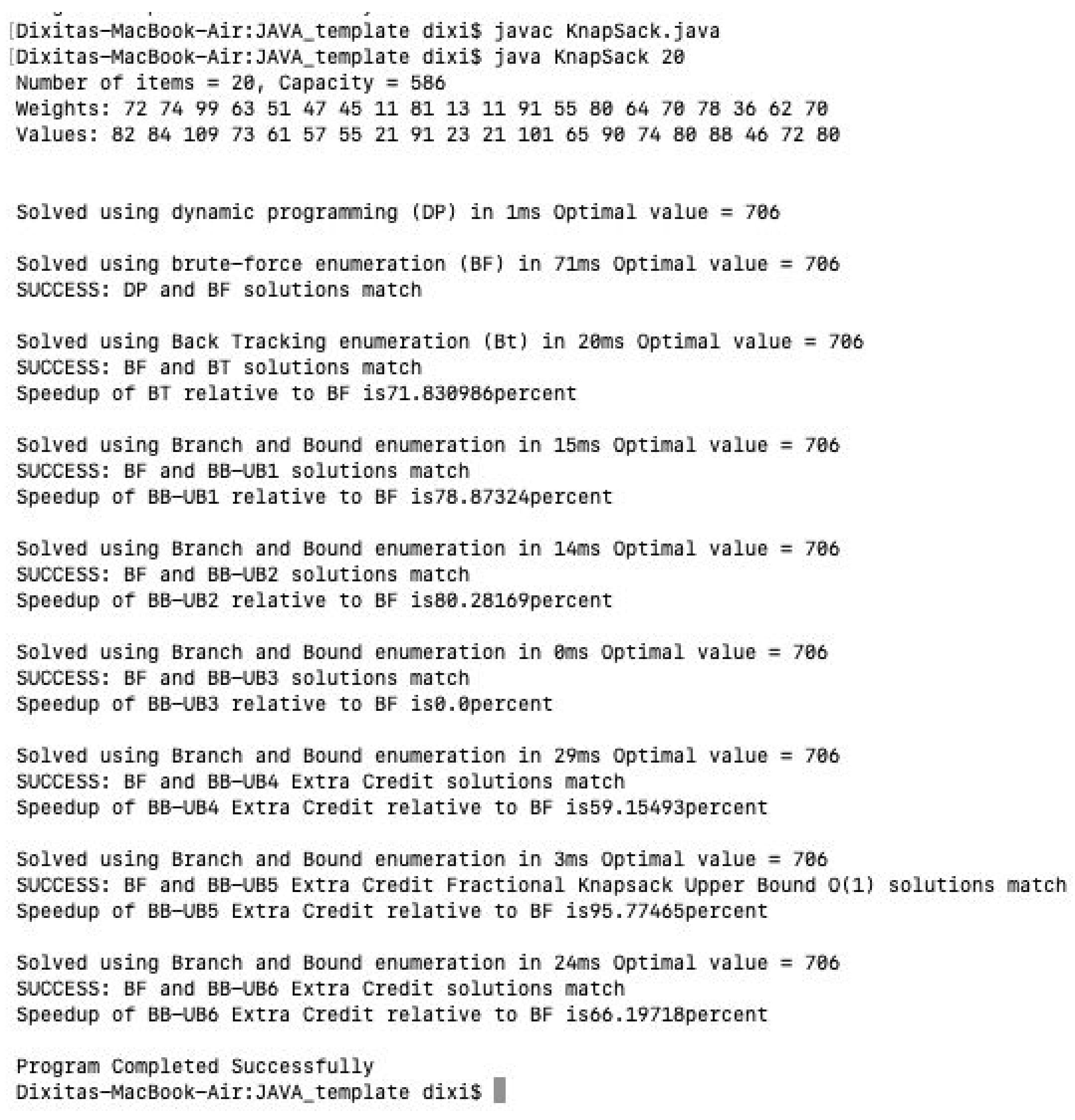
Extra Credit BB-UB5: Modified Upper Bound 3 approach Fractional Knapsack Solution computation with O(1) Average Amortized time per node.

In this work, I used the same Fractional Knapsack Solution computation method which I created for the BB-UB3 approach, I am calling that method before I call the UB4 solution method and It calculates the greedy Fractional Knapsack Solution from the first item to the slack based on capacity. Than I use that solution globally for comparing it as the Upper Bound and modify that variable when necessary.

This approach, avoids the O(n) calculations for getting the Fractional Knapsack Solution for each current node. As this way previously calculated Fractional Knapsack Solution gets modified for each node Not Taken. For the Taken Node no modification needed in this solution. If Item 2 is not taken, than item 2 weight and value will go away, and in the last taken item (slack), we take the weight of that item same as item 2 weight if available else we take the remaining weight of that slack item and go to next item and take needed weight from that and so on. For this to work, I am maintaining a pointer to the slack item (modified the same Fractional Knapsack Solution method used of UB3 this way). And also I keep track of the remaining weight of the slack item similar way. This both variable I declared globally so it can be modified when there is a new slack item.

This approach runs O(n) for whole path of the computation, It runs O(n)/n = O(1) per node. Slack item will not get touch by the later node. Example, if S is not touch by the previous node than it will get touch by later node. At any given node, there is maximum of two items that will get touch at this node and other nodes. Items may get touch by multiple node which is at most two. In general, it is limited by total number of items. Hence, Amortised time of this approach is O(1) per node.

This approach I have attempted to implement but it gives mismatch solutions sometimes and sometimes correct-match solutions. Need to correct the coding in some context. In the above table, for this work I have write the running time of the solution I got correctly. Those time for N=20, N=30 are more than UB3 implementation because for this modification my for loop is executing more time than it should be because of the incorrect implementation probably. Which need to make correct. Here I am attaching sometimes matched solution for this implementation for N=20.



```
Dixitas-MacBook-Air:JAVA_template dixi$ javac KnapSack.java
Dixitas-MacBook-Air:JAVA_template dixi$ java KnapSack 20
Number of items = 20, Capacity = 586
Weights: 72 74 99 63 51 47 45 11 81 13 11 91 55 80 64 70 78 36 62 70
Values: 82 84 109 73 61 57 55 21 91 23 21 101 65 90 74 80 88 46 72 80


Solved using dynamic programming (DP) in 1ms Optimal value = 706

Solved using brute-force enumeration (BF) in 71ms Optimal value = 706
SUCCESS: DP and BF solutions match

Solved using Back Tracking enumeration (Bt) in 20ms Optimal value = 706
SUCCESS: BF and BT solutions match
Speedup of BT relative to BF is71.830986percent

Solved using Branch and Bound enumeration in 15ms Optimal value = 706
SUCCESS: BF and BB-UB1 solutions match
Speedup of BB-UB1 relative to BF is78.87324percent

Solved using Branch and Bound enumeration in 14ms Optimal value = 706
SUCCESS: BF and BB-UB2 solutions match
Speedup of BB-UB2 relative to BF is80.28169percent

Solved using Branch and Bound enumeration in 0ms Optimal value = 706
SUCCESS: BF and BB-UB3 solutions match
Speedup of BB-UB3 relative to BF is0.0percent

Solved using Branch and Bound enumeration in 29ms Optimal value = 706
SUCCESS: BF and BB-UB4 Extra Credit solutions match
Speedup of BB-UB4 Extra Credit relative to BF is59.15493percent

Solved using Branch and Bound enumeration in 3ms Optimal value = 706
SUCCESS: BF and BB-UB5 Extra Credit Fractional Knapsack Upper Bound O(1) solutions match
Speedup of BB-UB5 Extra Credit relative to BF is95.77465percent

Solved using Branch and Bound enumeration in 24ms Optimal value = 706
SUCCESS: BF and BB-UB6 Extra Credit solutions match
Speedup of BB-UB6 Extra Credit relative to BF is66.19718percent

Program Completed Successfully
Dixitas-MacBook-Air:JAVA_template dixi$
```