To: Professor Timothy Kearns
From: Anh Nguyen
Cal Poly username: anguy183
CPE 349 - section 9
Date 12/02/2015
Subject: Knapsack Algorithms Compare write up

I. Algorithm 1 - Brute Force

1. Approaching :

Search for all possible candidate solutions. Compute the weight and value of each candidate then select the solution with most value and fit in the sack.

2. Pros:

- Easy to implement. Re-use the code from generating GrayCode to generate all possible bit string.
- Since this approaching method is straightforward, and it searched for all possible solutions, the solution obtaining from this solution is guaranteed to be absolute correct.
- Even though this algorithm take so much time, but it is practical in real life. For a problem size < 5, any human can compute the result within their head.

3. Cons:

Complexity $O(2^n)$, which is very insufficient. Not recommended for solving problem with problem size > 30.

4. Idea for Improvement

It is too naive to compute all possible candidate solutions. Most of the time, the candidate solution which has couple items will not be the final solution. Instead of computing all candidate solution, we can eliminate some candidate using this method:

- Search for the 2 heaviest items. If their total weight is less than capacity, eliminate all candidate solution which has only 1 item. This step cost O(n).
- Search for 2^{nd} and 3^{rd} heaviest item. If their total weight is less than (capacity - the heaviest item), eliminate all candidate which has 2 items. This step cost O(n).

Keep doing it until the 2 next heaviest item does not weights less than (capacity - set of the heaviest items). At the end, we are left with a subset of candidate solutions which ensure to contain the final solution.

II. Algorithm 2 - Greedy

1. Approaching:

Sort the list of item in order of ascending ratio of (value/weight). Walk down through that sorted list and keep picking items until the sack is full. I chose to sort the list by (value/weight) because it represent how dense that item it. The more density, the higher value per unit of weight.

2. Pros:

- Easy to implement. Just having the item implements Comparable, than we can use Arrays.sort to sort the list in O(nlogn).
- Fast and stable: Since sorting time is O(nlogn) and picking up item is O(n), the total complexity is O(n). This is very fast.
- Using space efficiently. Does not require extra space, which means we can compute very large problem size (>1000).
- For large problem size and does not require absolute solution (require approximate solution only), I do strongly recommend this approaching for time and space efficient.
- If all items has the same weight, greedy algorithm will compute the fastest optimal solution.

3. Cons

Since we must pick the item as a whole, not fractional of it. This approaching does not give the optimal solution.

III. Algorithm 3 - Dynamic

1. Approaching

Walk down through the list of item. At each item, if that item fit in the empty sack, calculate the Max(item's value + most possible value of the leftover space , value before coming to this item).

2. Pros

- Provide the optimal solution.
- Fast enough to use for medium problem size (.3s for hard200.txt test).
- Recommend using this approaching method for medium problem size (< 5000)

3. Cons

Space consuming O(item * capacity). Could run out of memory for very large problem size.

IV. Algorithm 4 - Branch & Bound

1. Approaching

Sort the list in ascending order of (value/weight) so that we will search on the better density item first. Having a Priority Queue which put Node with highest upperBound first. Start with the root Node, put in the Queue. Keep doing these steps:

- Remove an item from the Queue, which has largest upperBound.
- Check if it is solution (it is the solution if it is the leaf of tree)..
- If it is solution, then return it.
- If it is not the solution, make left & right child(if possible) and put them into the Queue.

Following this approaching, I will be constructing the tree as I traversing the tree. Since the Priority Queue will pop the highest upperBound, I will traverse the tree in frontier first search order.

2. **Pros**

This approaching provide a dynamic view of problem. We can see the max possible profit of each path as we are constructing the tree. We can apply this approaching to a problem where we know the list of all item, but we are not the one who has authority to decide to pick the item or not. After someone else, or some other factors decide to pick the item or not, we can change the path to which has max possible profit.

3. **Cons**

- Time consuming $O(n2^n)$. For large and complex problem like hard200.txt test, it take several minutes to complete.
- Space consuming. There is possible case that we have to build the whole complete tree in order to find the solution. In that case, we need at least 2^n extra space.
- Not recommended for problem with size > 50 because it take time and may run out of memory in regular computer.

V. Result Table

Below is my recorded table for all run. For hard200.txt test on Branch and Bound, I half the program after 1 minute.

	easy20 value(weight) *runtime(ms)*	easy50 value(weight) *runtime(ms)*	hard50 value(weight) *runtime(ms)*	easy200 value(weight) *runtime(ms)*	hard200 value(weight) *runtime(ms)*
Enum Brute Force	726 (519) *107*	X	X	X	X
Greedy Algorithm	692 (476) *0.15*	1115 (247) *0.16*	16538 (10038) *0.11*	4090 (2655) *0.2*	136724 (111924) *0.25*
Dynamic Programing	726 (519) *1.3*	1123 (250) *2.1*	16610 (10110) *22.4*	4092 (2658) *10.8*	137448 (112648) *256*
Brand & Bound	726 (519) *1.6*	1123 (250) *0.6*	16610 (10110) *0.3*	4092 (2656) *2.9*	136059 (111359) *60.6 s*