DESIGN AND ANALYSIS OF ALGORITHMS

G. Srilatha
Asst. Prof.
Department of CSE
Jyothishmathi Institute of Technology & Science

0-1 Knapsack problem

Given some items, pack the knapsack to get the maximum total value. Each item has some weight and some value. Total weight that we can carry is no more than some fixed number W. So we must consider weights of items as well as their values.

Item #	Weight	Value
1	1	8
2	3	6
3	5	5

Knapsack problem

There are two versions of the problem:

1. "0-1 knapsack problem"
   - Items are indivisible; you either take an item or not. Some special instances can be solved with *dynamic programming*

2. "Fractional knapsack problem"
   - Items are divisible: you can take any fraction of an item

0-1 Knapsack problem

- Given a knapsack with maximum capacity W, and a set S consisting of n items
- Each item i has some weight w_i and benefit value b_i (all w_i and W are integer values)
- Problem: How to pack the knapsack to achieve maximum total value of packed items?

0-1 Knapsack problem

- Problem, in other words, is to find

$$\max \sum_{i \in T} b_i \text{ subject to } \sum_{i \in T} w_i \leq W$$

- The problem is called a "*0-1*" problem, because each item must be entirely accepted or rejected.

Defining a Subproblem

If items are labeled *1..n*, then a subproblem would be to find an optimal solution for S_k *= {items labeled 1, 2, .. k}*

- This is a reasonable subproblem definition.
- The question is: can we describe the final solution (S_n) in terms of subproblems (S_k)?
- Unfortunately, we <u>can't</u> do that.

Defining a Subproblem

w_1 =2 b_1 =3	w_2 =4 b_2 =5	w_3 =5 b_3 =8	w_4 =3 b_4 =4	

?

Max weight: W = 20

For S_4:

Total weight: 14

Maximum benefit: 20

w_1 =2 b_1 =3	w_2 =4 b_2 =5	w_3 =5 b_3 =8	w_5 =9 b_5 =10

For S_5:

Total weight: 20

Maximum benefit: 26

Item #	Weight w_i	Benefit b_i
1	2	3
2	4	5
3	5	8
4	3	4
5	9	10

S_4 S_5

Solution for S_4 is not part of the solution for S_5!!!

Defining a Subproblem

- As we have seen, the solution for S_4 is not part of the solution for S_5

- So our definition of a subproblem is flawed and we need another one!

Defining a Subproblem

- Given a knapsack with maximum capacity W, and a set S consisting of n items
- Each item i has some weight w_i and benefit value b_i (all w_i and W are integer values)
- Problem: How to pack the knapsack to achieve maximum total value of packed items?

Defining a Subproblem

- Let's add another parameter: *w*, which will represent the maximum weight for each subset of items

- The subproblem then will be to compute *V[k,w], i.e.,* to find an optimal solution for S_k = *{items labeled 1, 2, .. k} in a knapsack of size w*

Recursive Formula for subproblems

- The subproblem will then be to compute *V[k,w], i.e.,* to find an optimal solution for S_k = *{items labeled 1, 2, .. k} in a knapsack of size w*

- Assuming knowing V[i, j], where i=0,1, 2, … k-1, j=0,1,2, …w, how to derive V[k,w]?

Recursive Formula for subproblems (continued)

Recursive formula for subproblems:

$$V[k,w] = \begin{cases} V[k-1,w] & \text{if } w_k > w \\ \max\{V[k-1,w], V[k-1,w-w_k]+b_k\} & \text{else} \end{cases}$$

It means, that the best subset of S_k that has total weight w is:

1) the best subset of S_{k-1} that has total weight $\leq w$, **or**

2) the best subset of S_{k-1} that has total weight $\leq w-w_k$ plus the item k

Recursive Formula

$$V[k,w] = \begin{cases} V[k-1,w] & \text{if } w_k > w \\ \max\{V[k-1,w], V[k-1,w-w_k]+b_k\} & \text{else} \end{cases}$$

- The best subset of S_k that has the total weight $\leq w$, either contains item k or not.
- First case: $w_k > w$. Item k can't be part of the solution, since if it was, the total weight would be $> w$, which is unacceptable.
- Second case: $w_k \leq w$. Then the item k can be in the solution, and we choose *the case with greater value*.

0-1 Knapsack Algorithm

```
for w = 0 to W
    V[0,w] = 0
for i = 1 to n
    V[i,0] = 0
for i = 1 to n
    for w = 0 to W
        if w_i <= w // item i can be part of the solution
            if b_i + V[i-1,w-w_i] > V[i-1,w]
                V[i,w] = b_i + V[i-1,w- w_i]
            else
                V[i,w] = V[i-1,w]
        else V[i,w] = V[i-1,w]  // w_i > w
```

Running time

```
for w = 0 to W
    V[0,w] = 0
for i = 1 to n
    V[i,0] = 0
for i = 1 to n
    for w = 0 to W
        < the rest of the code >
```

$O(W)$

Repeat n times

$O(W)$

What is the running time of this algorithm?

O(n*W)

Remember that the brute-force algorithm takes $O(2^n)$