Warmup Question

① Given a vertex v in a directed graph G, describe an algorithm to decide if there is a cycle in G that contains v.

② How check if a graph is bipartite or not using Depth first search.

The Knapsack Problem

A thief is robbing a store and can carry a maximal weight of W into his knapsack. There are n items available in the store and weight of i^{th} item is W_i and its value is V_i. The problem is to choose a subset of the items of maximum total value that will fit in the knapsack.

Two Variants of Knapsack

1. 0-1 Knapsack [This class]

2. fractional Knapsack

0-1 Knapsack Problem

Input: n items $\{1, 2, \ldots n\}$

item i worth v_i and weight w_i

Total weight W

Output: A subset $S \subseteq \{1, 2, \ldots n\}$ such that

$\sum_{i \in S} w_i \le W$ and $\sum_{i \in S} v_i$ is maximized.

Example:

$W = 45$

item	weight	value
1	10	20
2	20	65
3	30	50

Q: What is the optimal solution?

Bruteforce Solution

- Try all possible 2^n subsets of items.

- Can we do better than bruteforce?

Warmup: Some greedy Strategies

1. Greedy by highest value v_i

2. Greedy by least weight w_i

3. Greedy by largest value density $\frac{v_i}{w_i}$

Example:

$W = 5$

i	v_i	w_i	v_i/w_i
1	6	1	6
2	10	2	5
3	12	3	4

Greedy by value density v_i/w_i

takes item 1 & 2, Value = 16, weight = 3

Optimal soln = item 2 & 3, Value = 22, weight = 5.

Recap of yesterday's class

The Idea of Developing a DP Algorithm

Step1: Structure: Characterize the structure of an optimal solution.

– Decompose the problem 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.

Step2: Principle of Optimality: Recursively define the value of an optimal solution.

– Express the solution of the original problem in terms of optimal solutions for smaller problems.

The Idea of Developing a DP Algorithm

Step 3: Bottom-up computation: Compute the value of an optimal solution in a bottom-up fashion by using a table structure.

Step 4: Construction of optimal solution: Construct an optimal solution from computed information.

Steps 3 and 4 may often be combined.

opt soln to the Problem can be obtained by finding opt soln to subproblems.

The knapsack problem exhibits the optimal substructure property:

Let i_k be the highest-numberd item in an optimal solution $S = \{i_1, \ldots, i_{k-1}, i_k\}$, Then

1. *$S' = S - \{i_k\}$ is an optimal solution for weight $W - w_{i_k}$ and items $\{i_1, \ldots, i_{k-1}\}$*
2. *the value of the solution S is*

$$v_{i_k} + \text{the value of the subproblem solution } S'$$

Define

$c[i, w] =$ value of an optimal solution for items $\{1, \ldots, i\}$
and maximum weight w.

<u>Goal</u>: To find $c[n, W]$

Express $c[i, w]$ interms of subproblems

Case 1: if $w_i > W$

<u>Case2</u>:   If $w_i \leq W$

In summary,

$$c[i,w] = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \\ c[i-1,w] & \text{if } i > 0 \text{ and } w_i > w \\ \max\{v_i + c[i-1, w-w_i], c[i-1,w]\} & \text{if } i > 0 \text{ and } w_i \le w \end{cases}$$

Pseudocode

```
Knapsack(v, w, n, W)

    for w = 0 to W
        c[0,w] = 0

    for i = 1 to n
        c[i,0] = 0

    for i = 1 to n
        for w = 0 to W
            If w_i ≤ w
                c[i,w] = max{ c[i-1,w],
                              v_i + c[i-1, w - w_i] }
            else
                c[i,w] = c[i-1,w]

    return c[n,W]
```

Running time: $\Theta(nW)$

Example:

i	v_i	w_i
1	6	1
2	10	2
3	12	3

$W = 5$

Fill the DP table.

i	v_i	w_i
1	6	1
2	10	2
3	12	3

$W = 5$

$i \backslash w$	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	6	6	6	6	6
2	0	6	10	16	16	16
3	0	6	10	16	18	22

Optimal Value = $c[3,5] = 22$

The above algorithm does not tell which subset gives the optimal solution.

How to find the set of items present in optimal solution?

The set of items to take can be deduced from the c-table by starting at $c[n, W]$ and tracing where the optimal values came from as follows:

- If $c[i, w] = c[i-1, w]$, item i is not part of the solution, and we continue tracing with $c[i-1, w]$.
- If $c[i, w] \neq c[i-1, w]$, item i is part of the solution, and we continue tracing with $c[i-1, w-w_i]$.

Example 2: We have $n = 9$ items with

- value $= v = [2, 3, 3, 4, 4, 5, 7, 8, 8]$
- weight $= w = [3, 5, 7, 4, 3, 9, 2, 11, 5]$;
- Total allowable weight $W = 15$

DP generates the following c-table:

i/w	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2	2	2	2	2	2	2	2	2	2	2
2	0	0	0	2	2	3	3	3	5	5	5	5	5	5	5	5
3	0	0	0	2	2	3	3	3	5	5	5	5	6	6	6	8
4	0	0	0	2	4	4	4	6	6	7	7	7	9	9	9	9
5	0	0	0	4	4	4	6	8	8	8	10	10	11	11	11	13
6	0	0	0	4	4	4	6	8	8	8	10	10	11	11	11	13
7	0	0	7	7	7	11	11	11	13	15	15	15	17	17	18	18
8	0	0	7	7	7	11	11	11	13	15	15	15	17	17	18	18
9	0	0	7	7	7	11	11	15	15	15	19	19	19	21	23	23

Notation:

Let $V[i, w]$ maximum total value can be obtained by picking subset of items $\{1, 2, .. i\}$ of Combined size w.

We compute $V[i, w]$ for each $i \in \{1, 2, -- n\}$
$w \in \{0, 1, -- W\}$

Goal: To Compute $V[n, W]$

What is $V[0, w] = ?$ for $0 \leq w \leq W$

$V[i, w] = ?$ for $w < 0$

<u>Recursion</u>:

Express $V[i, w]$ interms of subproblems.

$$V[i, w] = ? + ?$$

Developing a DP Algorithm for Knapsack

[redacted] Recursively define the value of an optimal solution in terms of solutions to smaller problems.

Initial Settings: Set

$V[0, w] = 0$	for $0 \leq w \leq W$,	no item
$V[i, w] = -\infty$	for $w < 0$,	illegal

Recursive Step: Use

$$V[i, w] = \max(V[i-1, w], v_i + V[i-1, w-w_i])$$

for $1 \leq i \leq n$, $0 \leq w \leq W$.

Correctness of the Method for Computing $V[i, w]$

Lemma: For $1 \le i \le n$, $0 \le w \le W$,

$$V[i, w] = \max(V[i-1, w], v_i + V[i-1, w-w_i]).$$

Proof: To compute $V[i, w]$ we note that we have only two choices for item i

Leave item ~~file~~ i**:** The best we can do with files $\{1, 2, \ldots, i-1\}$ and weight limit w is $V[i-1, w]$.

Take item ~~file~~ i (only possible if $w_i \le w$): Then we gain Value v_i ~~of computing time~~, but have spent w_i ~~bytes~~ weight of our ~~storage~~ knapsack. The best we can do with remaining items ~~files~~ $\{1, 2, \ldots, i-1\}$ and ~~storage~~ weight $(w - w_i)$ is $V[i-1, w-w_i]$.
Totally, we get $v_i + V[i-1, w-w_i]$.

Note that if $w_i > w$, then $v_i + V[i-1, w-w_i] = -\infty$ so the lemma is correct in any case.

Developing a DP Algorithm for Knapsack

Step 3: Bottom-up computing $V[i, w]$ (using iteration, not recursion).

Bottom: $V[0, w] = 0$ for all $0 \leq w \leq W$.

Bottom-up computation: Computing the table using

$$V[i, w] = \max(V[i-1, w], v_i + V[i-1, w-w_i])$$

row by row.

V[i,w]	w=0	1	2	3	...	...	W	
i= 0	0	0	0	0	...	...	0	bottom
1								
2								
⋮								
n								
								up

Example of the Bottom-up computation

Let $W = 10$ and

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

$V[i,w]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50
3	0	0	0	0	40	40	40	40	40	50	70
4	0	0	0	50	50	50	50	90	90	90	90

Remarks:

- The final output is $V[4, 10] = 90$.
- The method described does not tell which subset gives the optimal solution. (It is $\{2, 4\}$ in this example).

The Dynamic Programming Algorithm

KnapSack(v, w, n, W)
{
  for ($w = 0$ to W) $V[0, w] = 0$;
$O(nW)$ for ($i = 1$ to n)
    for ($w = 0$ to W)
      if ($w[i] \leq w$)
        $V[i, w] = \max\{V[i-1, w], v[i] + V[i-1, w - w[i]]\}$;
const time
      else
        $V[i, w] = V[i-1, w]$;
  return $V[n, W]$;
}

Time complexity: Clearly, $O(nW)$.

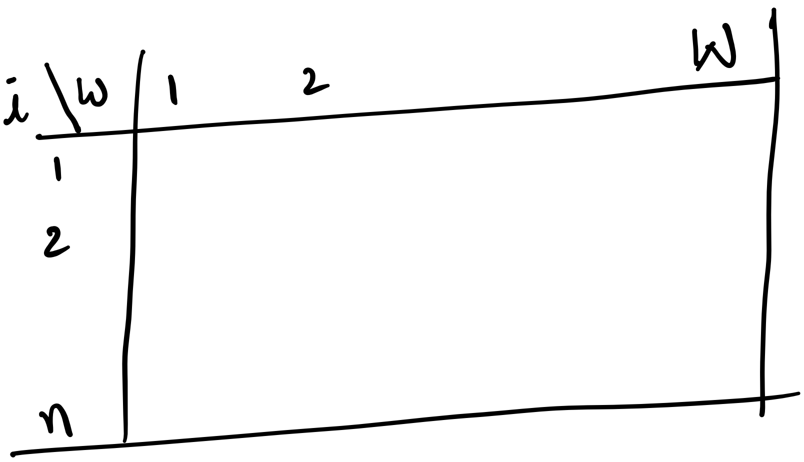

table size $O(nW)$

Constructing the Optimal Solution

- The algorithm for computing $V[i,w]$ described in the previous slide does not keep record of which subset of items gives the optimal solution.

- To compute the actual subset, we can add an auxiliary boolean array $keep[i,w]$ which is 1 if we decide to take the i-th item in $V[i,w]$ and 0 otherwise.

Question: How do we use all the values $keep[i,w]$ to determine the subset T of items having the maximum Value?

Constructing the Optimal Solution

Question: How do we use the values $keep[i, w]$ to determine the subset T of [redacted] items having the maximum [redacted] value?

If keep$[n, W]$ is 1, then $n \in T$. We can now repeat this argument for keep$[n-1, W-w_n]$.
If keep$[n, W]$ is 0, the $n \notin T$ and we repeat the argument for keep$[n-1, W]$.

Therefore, the following partial program will output the elements of T:

$K = W$;
for ($i = n$ downto 1)
    if (keep$[i, K] == 1$)
    $\{$
        output i;
        $K = K - w[i]$;
    $\}$

The Complete Algorithm for the Knapsack Problem

KnapSack(v, w, n, W)
{
  for ($w = 0$ to W) $V[0, w] = 0$;
  for ($i = 1$ to n)
    for ($w = 0$ to W)
      if (($w[i] \leq w$) and ($v[i] + V[i-1, w-w[i]] > V[i-1, w]$))
      {
        $V[i, w] = v[i] + V[i-1, w-w[i]]$; item i is Picked
        keep[i, w] = 1;
      }
      else
      {
        $V[i, w] = V[i-1, w]$; item i is not picked
        keep[i, w] = 0;
      }
  $K = W$;
  for ($i = n$ downto 1)
    if (keep[i, K] == 1)
    {
      output i;
      $K = K - w[i]$;
    }
  return $V[n, W]$;
}

Gives the subset of items with max value having weight $\leq W$