CS 603: Dynamic Programming: Subset Sums and the Knapsack Problem

Ellen Veomett

University of San Francisco

Outline

1. Subset Sums
2. Generalization: the Knapsack Problem

Recall: Weighted Interval Scheduling

- List of n tasks to be completed
- Each task has start time $s(i)$ and finish time $f(i)$, $i = 1, 2, \ldots, n$.
- Each task has a weight/value associated to it: v_i. (Can think of this as profit for completing task i).
- Goal: pick subset $S \subset \{1, 2, \ldots, n\}$ of tasks with *nonoverlapping* times such that

$$\sum_{i \in S} v_i$$

is as large as possible.

Now: Total Scheduling Problem

- List of n tasks to be completed.
- Each task has time/weight associated to it: w_i (an integer)
- There's a total amount of time/weight that the resource (such as a printer) can be used: W (also an integer)
- Goal: pick subset $S \subset \{1, 2, \ldots, n\}$ of tasks of highest weight, such that the resource is used as much as possible. That is,

$$\sum_{i \in S} w_i$$

is maximized, subject to:

$$\sum_{i \in S} w_i \leq W$$

Dynamic Programming reminders

Template for Solving a Problem using Dynamic Programming

- Solution to full problem can be deduced (easily) from solutions to sub-problems.
- Number of sub-problems is small (polynomial in n, the original problem size).
- There is a natural ordering of sub-problems, from smallest to largest.

Dynamic Programming reminders

Template for Solving a Problem using Dynamic Programming

- Solution to full problem can be deduced (easily) from solutions to sub-problems.
- Number of sub-problems is small (polynomial in n, the original problem size).
- There is a natural ordering of sub-problems, from smallest to largest.

To find a dynamic programming solution

- Describe the structure of an optimal solution.
- Use this structure to recursively define the value of an optimal solution.
- Construct solution, in a bottom-up fashion, typically storing solutions as they are found

If needed, the parts consisting of a solution can potentially be re-constructed from the stored data.

Again: consider whether solution has *n*th task or not

Again: consider whether solution has nth task or not

- If nth item is in the optimal solution, then the remaining items create an optimal solution when considering items $j = 1, 2, \ldots, n-1$, and total weight
$$W - w_n$$

Again: consider whether solution has *n*th task or not

- If *n*th item is in the optimal solution, then the remaining items create an optimal solution when considering items $j = 1, 2, \ldots, n-1$, and total weight
$$W - w_n$$
- If the *n*th item is *not* in the optimal solution, the optimal solution is the same as when considering items $j = 1, 2, \ldots, n-1$ and total weight W.

Again: consider whether solution has *n*th task or not

- If *n*th item is in the optimal solution, then the remaining items create an optimal solution when considering items $j = 1, 2, \ldots, n-1$, and total weight
$$W - w_n$$
- If the *n*th item is *not* in the optimal solution, the optimal solution is the same as when considering items $j = 1, 2, \ldots, n-1$ and total weight W.
- Notation:
$$Opt(i, w)$$
is the optimal solution when considering items $j = 1, 2, \ldots, i$ and total weight w

Again: consider whether solution has *n*th task or not

- If *n*th item is in the optimal solution, then the remaining items create an optimal solution when considering items $j = 1, 2, \dots, n-1$, and total weight
$$W - w_n$$

- If the *n*th item is *not* in the optimal solution, the optimal solution is the same as when considering items $j = 1, 2, \dots, n-1$ and total weight W.

- Notation:
$$Opt(i, w)$$
is the optimal solution when considering items $j = 1, 2, \dots, i$ and total weight w

- Recurrence relation:

$$Opt(i, w) = \begin{cases} Opt(i-1, w) & \text{if } w_i > w \text{(item } i \text{ can't be used anyway)} \\ \max\{Opt(i-1, w), w_i + Opt(i-1, w - w_i)\} & \text{otherwise} \end{cases}$$

How can we build the solution using memoization?

$$Opt(i, w) = \begin{cases} Opt(i-1, w) & \text{if } w_i > w \text{(item } i \text{ can't be used anyway)} \\ \max\{Opt(i-1, w), w_i + Opt(i-1, w-w_i)\} & \text{otherwise} \end{cases}$$

How can we build the solution using memoization?

$$Opt(i,w) = \begin{cases} Opt(i-1,w) & \text{if } w_i > w \text{(item } i \text{ can't be used anyway)} \\ \max\{Opt(i-1,w), w_i + Opt(i-1, w-w_i)\} & \text{otherwise} \end{cases}$$

- For any i, $Opt(i,0) =$

How can we build the solution using memoization?

$$Opt(i, w) = \begin{cases} Opt(i-1, w) & \text{if } w_i > w \text{(item } i \text{ can't be used anyway)} \\ \max\{Opt(i-1, w), w_i + Opt(i-1, w-w_i)\} & \text{otherwise} \end{cases}$$

- For any i, $Opt(i, 0) = 0$

How can we build the solution using memoization?

$$Opt(i,w) = \begin{cases} Opt(i-1,w) & \text{if } w_i > w \text{(item } i \text{ can't be used anyway)} \\ \max\{Opt(i-1,w), w_i + Opt(i-1, w-w_i)\} & \text{otherwise} \end{cases}$$

- For any i, $Opt(i,0) = 0$
- $Opt(0,w)$ assumes we include 0 of the items, so $Opt(0,w) = 0$

How can we build the solution using memoization?

$$Opt(i, w) = \begin{cases} Opt(i-1, w) & \text{if } w_i > w \text{(item } i \text{ can't be used anyway)} \\ \max\{Opt(i-1, w), w_i + Opt(i-1, w-w_i)\} & \text{otherwise} \end{cases}$$

- For any i, $Opt(i, 0) = 0$
- $Opt(0, w)$ assumes we include 0 of the items, so $Opt(0, w) = 0$
- We can build a 2-dimensional array, indexed by i and w, to store $Opt(i, w)$.

How can we build the solution using memoization?

$$Opt(i,w) = \begin{cases} Opt(i-1,w) & \text{if } w_i > w \text{(item } i \text{ can't be used anyway)} \\ \max\{Opt(i-1,w), w_i + Opt(i-1, w-w_i)\} & \text{otherwise} \end{cases}$$

- For any i, $Opt(i,0) = 0$
- $Opt(0,w)$ assumes we include 0 of the items, so $Opt(0,w) = 0$
- We can build a 2-dimensional array, indexed by i and w, to store $Opt(i,w)$.
- Then use the recursive relation to fill out the array.

```
public int subsetSum(int[] w, int W){
   // initialize int[] m of size (n+1) x (W+1) to have all 0s
   // where either index is 0
   int n = w.length;
   for (int i=1; i<= n; i++){
      for (int w=1;w<=W; w++){
         if (w[i] > w){
            m[i][w] = m[i-1][w];
         }
         else {
            m[i][w] = Math.max(m[i-1][w], w[i]+m[i-1][w-w[i]]);
      }
   }
   return m[n][W];
}
```

Example

$$Opt(i, w) = \begin{cases} Opt(i-1, w) & \text{if } w_i > w (\text{item } i \text{ can't be used anyway}) \\ \max\{Opt(i-1, w), w_i + Opt(i-1, w-w_i)\} & \text{otherwise} \end{cases}$$

$$w_1 = 4, w_2 = 3, w_3 = 1, w_4 = 2 \qquad W = 6$$

n\ W	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0						
2	0						
3	0						
4	0						

Example

$$Opt(i,w) = \begin{cases} Opt(i-1,w) & \text{if } w_i > w (\text{item } i \text{ can't be used anyway}) \\ \max\{Opt(i-1,w), w_i + Opt(i-1, w-w_i)\} & \text{otherwise} \end{cases}$$

$$w_1 = 4, w_2 = 3, w_3 = 1, w_4 = 2 \qquad W = 6$$

n\ W	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	0	0	4	4	4
2	0						
3	0						
4	0						

Example

$$Opt(i,w) = \begin{cases} Opt(i-1,w) & \text{if } w_i > w \text{(item } i \text{ can't be used anyway)} \\ \max\{Opt(i-1,w), w_i + Opt(i-1, w-w_i)\} & \text{otherwise} \end{cases}$$

$$w_1 = 4, w_2 = 3, w_3 = 1, w_4 = 2 \qquad W = 6$$

n\ W	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	0	0	4	4	4
2	0	0	0	3	4	4	4
3	0						
4	0						

Example

$$Opt(i,w) = \begin{cases} Opt(i-1,w) & \text{if } w_i > w (\text{item } i \text{ can't be used anyway}) \\ \max\{Opt(i-1,w), w_i + Opt(i-1, w-w_i)\} & \text{otherwise} \end{cases}$$

$$w_1 = 4, w_2 = 3, w_3 = 1, w_4 = 2 \qquad W = 6$$

n\ W	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	0	0	4	4	4
2	0	0	0	3	4	4	4
3	0	1	1	3	4	5	5
4	0						

Example

$$Opt(i,w) = \begin{cases} Opt(i-1,w) & \text{if } w_i > w (\text{item } i \text{ can't be used anyway}) \\ \max\{Opt(i-1,w), w_i + Opt(i-1, w-w_i)\} & \text{otherwise} \end{cases}$$

$$w_1 = 4, w_2 = 3, w_3 = 1, w_4 = 2 \qquad W = 6$$

n\ W	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	0	0	4	4	4
2	0	0	0	3	4	4	4
3	0	1	1	3	4	5	5
4	0	1	2	3	4	5	6

Do we really need to fill out *every* slot in the table?

Do we really need to fill out *every* slot in the table? No! (Though filling out only the needed slots won't change the general complexity calculation)

```
public int subsetSumRecursive(int i, int W, int[] w){
   // int[] m of size (n+1) x (W+1) is already initialized to
       have all -1s
   if (i == 0 || W ==0){
      m[i][W] = 0;
      return m[i][W];}
   if (m[i-1][W] == -1){
      m[i-1][W] = subsetSumRecursive(i-1,W, w);}
   if (w[i] > W){
      m[i][W] = m[i-1][W];
      return m[i][W];}
   if (m[i-1][W-w[i]] == -1){
      m[i-1][W-w[i]] = subsetSumRecursive(i-1, W-w[i], w);}
   m[i][W] = Math.max(m[i-1][W], w[i] + m[i-1][W-w[i]]);
   return m[i][W];
}
```

Example

$$w_1 = 4, w_2 = 3, w_3 = 1, w_4 = 2 \qquad W = 6$$

n\ W	0	1	2	3	4	5	6
0	-1	-1	-1	-1	-1	-1	-1
1	-1	-1	-1	-1	-1	-1	-1
2	-1	-1	-1	-1	-1	-1	-1
3	-1	-1	-1	-1	-1	-1	-1
4	-1	-1	-1	-1	-1	-1	?

$$Opt(4, 6)$$

Example

$$w_1 = 4, w_2 = 3, w_3 = 1, w_4 = 2 \qquad W = 6$$

n\ W	0	1	2	3	4	5	6
0	-1	-1	-1	-1	-1	-1	-1
1	-1	-1	-1	-1	-1	-1	-1
2	-1	-1	-1	-1	-1	-1	-1
3	-1	-1	-1	-1	?	-1	?
4	-1	-1	-1	-1	-1	-1	?

$Opt(3, 6)$
$Opt(3, 4)$
$Opt(4, 6)$

Example

$$w_1 = 4, w_2 = 3, w_3 = 1, w_4 = 2 \qquad W = 6$$

n\ W	0	1	2	3	4	5	6
0	-1	-1	-1	-1	-1	-1	-1
1	-1	-1	-1	-1	-1	-1	-1
2	-1	-1	-1	?	?	?	?
3	-1	-1	-1	-1	?	-1	?
4	-1	-1	-1	-1	-1	-1	?

$Opt(2,4)$

$Opt(2,3)$

$Opt(2,6)$

$Opt(2,5)$

$Opt(3,6)$

$Opt(3,4)$

$Opt(4,6)$

Example

$$w_1 = 4, w_2 = 3, w_3 = 1, w_4 = 2 \qquad W = 6$$

n\ W	0	1	2	3	4	5	6
0	-1	-1	-1	-1	-1	-1	-1
1	-1	?	?	?	?	?	?
2	-1	-1	-1	?	?	?	?
3	-1	-1	-1	-1	?	-1	?
4	-1	-1	-1	-1	-1	-1	?

$Opt(1,6)$

$Opt(1,2)$

$Opt(1,5)$

$Opt(2,4)$

$Opt(2,3)$

$Opt(2,6)$

$Opt(2,5)$

$Opt(1,1)\,Opt(3,6)$

$Opt(1,4)\,Opt(3,4)$

$Opt(1,3)\,Opt(4,6)$

Example

$$w_1 = 4, w_2 = 3, w_3 = 1, w_4 = 2 \qquad W = 6$$

n\ W	0	1	2	3	4	5	6
0	?	?	?	?	?	?	?
1	-1	?	?	?	?	?	?
2	-1	-1	-1	?	?	?	?
3	-1	-1	-1	-1	?	-1	?
4	-1	-1	-1	-1	-1	-1	?

$Opt(0,0)Opt(1,6)$
$Opt(0,3)Opt(1,2)$
$Opt(0,4)Opt(1,5)$
$Opt(0,6)Opt(2,4)$
$Opt(0,2)Opt(2,3)$
$Opt(0,1)Opt(2,6)$
$Opt(0,5)Opt(2,5)$
$Opt(1,1)Opt(3,6)$
$Opt(1,4)Opt(3,4)$
$Opt(1,3)Opt(4,6)$

Example

$$w_1 = 4, w_2 = 3, w_3 = 1, w_4 = 2 \qquad W = 6$$

n\ W	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	-1	?	?	?	?	?	?
2	-1	-1	-1	?	?	?	?
3	-1	-1	-1	-1	?	-1	?
4	-1	-1	-1	-1	-1	-1	?

$$Opt(1,6)$$
$$Opt(1,2)$$
$$Opt(1,5)$$
$$Opt(2,4)$$
$$Opt(2,3)$$
$$Opt(2,6)$$
$$Opt(2,5)$$
$$Opt(1,1)\,Opt(3,6)$$
$$Opt(1,4)\,Opt(3,4)$$
$$Opt(1,3)\,Opt(4,6)$$

Example

$$w_1 = 4, w_2 = 3, w_3 = 1, w_4 = 2 \qquad W = 6$$

n\ W	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	0	0	4	4	4
2	-1	-1	-1	?	?	?	?
3	-1	-1	-1	-1	?	-1	?
4	-1	-1	-1	-1	-1	-1	?

$Opt(2,4)$
$Opt(2,3)$
$Opt(2,6)$
$Opt(2,5)$
$Opt(3,6)$
$Opt(3,4)$
$Opt(4,6)$

Example

$$w_1 = 4, w_2 = 3, w_3 = 1, w_4 = 2 \qquad W = 6$$

n\ W	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	0	0	4	4	4
2	-1	-1	-1	3	4	4	4
3	-1	-1	-1	-1	?	-1	?
4	-1	-1	-1	-1	-1	-1	?

$Opt(3, 6)$
$Opt(3, 4)$
$Opt(4, 6)$

Example

$$w_1 = 4, w_2 = 3, w_3 = 1, w_4 = 2 \qquad W = 6$$

n\ W	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	0	0	4	4	4
2	-1	-1	-1	3	4	4	4
3	-1	-1	-1	-1	4	-1	5
4	-1	-1	-1	-1	-1	-1	?

$Opt(4, 6)$

Example

$$w_1 = 4, w_2 = 3, w_3 = 1, w_4 = 2 \qquad W = 6$$

n\ W	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	0	0	4	4	4
2	-1	-1	-1	3	4	4	4
3	-1	-1	-1	-1	4	-1	5
4	-1	-1	-1	-1	-1	-1	6

Time Complexity of Subset Sum problem

We just filled out an $n \times W$ chart (or at least a significant percentage of it).

Complexity:

Time Complexity of Subset Sum problem

We just filled out an $n \times W$ chart (or at least a significant percentage of it).

Complexity:

$$O(nW)$$

This is *not* polynomial in n. This is *pseudo-polynomial* (it's polynomial in n and the largest input integer W).

Outline

Knapsack Problem

- List of n items to be put into a knapsack you carry.
- Each item has weight associated to it: w_i (an integer)
- Each item has value associated to it: v_i (an integer)
- There's a total amount of weight you can maximally carry in your knapsack: W (also an integer)
- Goal: pick subset $S \subset \{1, 2, \ldots, n\}$ of items to put in the knapsack such that value is highest. That is:

  $$\sum_{i \in S} v_i$$

  is maximized, subject to:

  $$\sum_{i \in S} w_i \leq W$$

Solving Knapsack Problem

- Everything is the same except that instead of maximizing $\sum_{i \in S} w_i$, we maximize

$$\sum_{i \in S} v_i$$

Solving Knapsack Problem

- Everything is the same except that instead of maximizing $\sum_{i\in S} w_i$, we maximize

$$\sum_{i\in S} v_i$$

- Use:

$$Opt(0,w) = 0$$
$$Opt(i,0) = 0$$
$$Opt(i,w) = \begin{cases} Opt(i-1,w) & \text{if } w_i > w \text{(item } i \text{ can't be used anyway)} \\ \max\{Opt(i-1,w), v_i + Opt(i-1,w-w_i)\} & \text{otherwise} \end{cases}$$