The Knapsack Problem - an Introduction to Dynamic Programming
Different Problem Solving Approaches

Greedy Algorithms
- Build up solutions in small steps
- Make local decisions
- Previous decisions are never reconsidered
- We will solve the Divisible Knapsack problem with a greedy approach

Dynamic Programming
- Solves larger problem by relating it to overlapping subproblems and then solves the subproblems
  - Important to store the results from subproblems so that they aren’t computed repeatedly
- We will solve the Indivisible Knapsack problem with dynamic programming

Backtracking
- Solve by brute force searching the solution space, pruning when possible
The Knapsack Problem

We are given:
- A collection of n items
- Each item has an associated non-negative weight, $w_i$
- Each item has an associated value (cost), $c_i$
- And we are given a knapsack that can hold total weight $W$

Our task is:
- Determine the set $S$ of items of maximum total value (cost) that can be contained in the knapsack subject to the constraint that the total weight is no greater than $W$
The Knapsack Problem

A first version: the Divisible Knapsack Problem
- Items do not have to be included in their entirety
- Arbitrary fractions of an item can be included
- This problem can be solved with a GREEDY approach
- Complexity - $O(n \log n)$ to sort, then $O(n)$ to include, so $O(n \log n)$

```
KNAPSACK-DIVISIBLE(n,c,w,W)
1. sort items in decreasing order of $c_i/w_i$
2. $i = 1$
3. currentW = 0
4. while (currentW + $w_i < W$) {
5.     take item of weight $w_i$ and cost $c_i$
6.     currentW += $w_i$
7.     i++
8. }
9. take $W$-currentW portion of item i
```
The Indivisible Knapsack Problem

We are given:
- A collection of \( n \) items
- Each item has an associated non-negative weight, \( w_i \)
- Each item has an associated value (cost), \( c_i \)
- And we are given a knapsack that can hold total weight \( W \)

Our task is:
- Determine the set \( S \) of items of maximum total value that can be contained in the knapsack subject to the constraint that the total weight is no greater than \( W \)
- Items must be included in their entirety or not at all
The Indivisible Knapsack Problem

Possible Solutions:

- **Greedy approaches**
  - Sort by cost, and include from highest on down until full
  - Sort by cost per unit weight, and include from highest on down until full
  - Sort by weight, and include from lightest upward until full

- **No known greedy approach is optimal**
  - For each greedy algorithm, we can design at least one case in which it fails to produce the optimal result

- **Backtracking** - consider all possible solutions
  - How big is the solution space - all possible subsets of n items

- **Dynamic Programming**
Dynamic Programming

**General Idea:**
- Solves larger problem by relating it to overlapping subproblems and then solves the subproblems
- It works through the exponential set of solutions, but doesn’t examine them all explicitly
- Stores intermediate results so that they aren’t recomputed
Dynamic Programming

For dynamic programming to be applicable:
- At most polynomial number of subproblems (else still exponential-time solution)
- Solution to original problem is easily computed from the solutions to the subproblems
- There is a natural ordering on subproblems from “smallest” to “largest” and an easy to compute recurrence that allows solving a subproblem from smaller subproblems
Dynamic Programming – A First Example

Fibonacci Numbers
- 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...
- F(0) = 0, F(1) = 1
- F(n) = F(n-1) + F(n-2)

Computing the Fibonacci Numbers
- Each n\textsuperscript{th} number is a function of previous solutions
- A recursive solution:

```
Fib(n)
1. if n < 0 then RETURN “undefined”
2. if n ≤ 1 then RETURN n
3. RETURN Fib(n-1) + Fib(n-2)
```

What’s the drawback to this solution?
- Complexity is exponential
Dynamic Programming - A First Example

Computing Fibonacci Numbers - Can we do better than exponential?

- Yes - “Memoization”
- Each time you encounter a new subproblem and compute the result, store it so that you never need to recompute that subproblem

- Each subproblem is computed just once, and is based on the results of smaller subproblems
  - This leads naturally to converting the recursive solution to an iterative solution

```
FibDynProg(n)
1. Fib[0] = 0
2. Fib[1] = 1
3. for i=2 to n do
4.     Fib[i] = Fib[i-1] + Fib[i-2]
5. RETURN Fib[n]
```
Dynamic Programming - Returning to the Knapsack Problem

How can we solve the Knapsack Problem using Dynamic Programming?

We are given:
- A collection of $n$ items
- Each item has an associated non-negative weight, $w_i$
- Each item has an associated value (cost), $c_i$
- And we are given a knapsack that can hold total weight $W$

How can we break the problem down so that the overall solution is related to overlapping subproblems

We need to do two things:
- Define what our subproblems are
- Define a recurrence relation that links them to the original problem
Dynamic Programming – Returning to the Knapsack Problem

How can we define subproblems?:
- Consider an optimal solution
- Consider the items: 1,2,3,...n
- Either item n is in the solution or not
  - If n is in solution: \( Knapsack(n, W) = c_n + Knapsack(n-1, W-w_n) \)
  - If n is not in solution: \( Knapsack(n, W) = Knapsack(n-1, W) \)

How do we ultimately decide if item n is in the optimal solution?
- Solve the subproblems first
- Then choose which option (include or not) works out better

- \( Knapsack(n, W) = \max(c_n + Knapsack(n-1, W-w_n), Knapsack(n-1, W)) \)
Dynamic Programming - Returning to the Knapsack Problem

A Recursive Algorithm Solution

\[
\text{KNAP-IND-REC}(n, c, w, W) \\
1. \text{if } n \leq 0 \\
2. \quad \text{return } 0 \\
3. \text{if } W < w_n \\
4. \quad \text{withLastItem } = -1 \quad // \text{ undefined} \\
5. \text{else} \\
6. \quad \text{withLastItem } = c_n + \text{KNAP-IND-REC}(n-1, c, w, W-w_n) \\
7. \text{withoutLastItem } = \text{KNAP-IND-REC}(n-1, c, w, W) \\
8. \text{return } \max\{\text{withLastItem, withoutLastItem}\}
\]

NOTES:
• \(n\) is the number of items being considered (we’re working our way backwards)
• \(c\) is the vector of costs associated with the items
• \(w\) is the vector of weights associated with the item (assume integer)
• \(W\) is the capacity of the knapsack

Slides based on Kevin Wayne / Pearson-Addison Wesley
Dynamic Programming – Returning to the Knapsack Problem

What do we need to store?
- The solution to all of our subproblems

What are the subproblems?
- The solution considering every possible combination of remaining items and remaining weight
- Let \( S[k][v] := \) the solution to the subproblem corresponding to the first \( k \) items and available weight \( v \)
  - i.e. \( S[k][v] = \) the maximum cost of items that fit inside a knapsack of size (weight) \( v \), choosing from the first \( k \) items

\[
S[k][v] = \max(c_k + S[k-1][v-w_k], S[k-1][v])
\]

Note - we’re only considering \( S[k-1][v-w_k] \) if it can fit (i.e. \( v \geq w_k \)). If there isn’t room for it, the answer is just \( S[k-1][v] \).
Dynamic Programming – Returning to the Knapsack Problem

Converting to an Iterative Solution
- Build up an (n+1) x (W+1) array of subproblem solutions
- Computational Complexity: $O(nW)$
  - Referred to as pseudo-polynomial
  - The size of the problem grows exponentially with the size (number of digits) of $W$

```
KNAPSACK-INDIVISIBLE(n,c,w,W)
1. init S[0][v]=0 for every v=0,…,W
2. init S[k][0]=0 for every k=0,…,n
3. for v=1 to W do
4.     for k=1 to n do
5.         S[k][v] = S[k-1][v]
6.         if ($w_k \leq v$) and
7.             ($S[k-1][v-w_k]+c_k > S[k][v]$)
8.             then
9.                 S[k][v] = S[k-1][v-w_k]+c_k
10.            RETURN S[n][W]
```
Knapsack Example

<table>
<thead>
<tr>
<th>Increasing</th>
<th>( n )</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \phi )</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>{ 1 }</td>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>{ 1, 2 }</td>
<td></td>
<td>0</td>
<td>1</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>{ 1, 2, 3 }</td>
<td></td>
<td>0</td>
<td>1</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>18</td>
<td>19</td>
<td>24</td>
<td>25</td>
<td>25</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>{ 1, 2, 3, 4 }</td>
<td></td>
<td>0</td>
<td>1</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>18</td>
<td>22</td>
<td>24</td>
<td>28</td>
<td>29</td>
<td>29</td>
<td>40</td>
</tr>
<tr>
<td>{ 1, 2, 3, 4, 5 }</td>
<td>0</td>
<td>1</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>18</td>
<td>22</td>
<td>28</td>
<td>29</td>
<td>34</td>
<td>35</td>
<td>40</td>
<td></td>
</tr>
</tbody>
</table>

Increasing \( W \)

\[
\text{OPT: } \{ 4, 3 \} \\
\text{value} = 22 + 18 = 40
\]

\( W = 11 \)

<table>
<thead>
<tr>
<th>Item</th>
<th>Value</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>18</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>22</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>28</td>
<td>7</td>
</tr>
</tbody>
</table>

Slides based on Kevin Wayne / Pearson-Addison Wesley
Dynamic Programming - Returning to the Knapsack Problem

How do we Recover the list of Items actually included?

- Trace backwards through the matrix
- We know item n is included if:
  - \( S[k-1][W-w_n] + c_n \geq S[k-1][W] \)
- After determining the status of item n, continue working backwards through the remaining items, adjusting for what is already known