Greedy vs Dynamic Programming Approach
Outline

- Compare the methods
- Knapsack problem
  - Greedy algorithms for 0/1 knapsack
  - An approximation algorithm for 0/1 knapsack
  - Optimal greedy algorithm for knapsack with fractions
  - A dynamic programming algorithm for 0/1 knapsack
Greedy Approach VS Dynamic Programming (DP)

- Greedy and Dynamic Programming are methods for solving optimization problems.
- Greedy algorithms are usually more efficient than DP solutions.
- However, often you need to use dynamic programming since the optimal solution cannot be guaranteed by a greedy algorithm.
- DP provides efficient solutions for some problems for which a brute force approach would be very slow.
- To use Dynamic Programming we need only show that the principle of optimality applies to the problem.
The 0/1 Knapsack problem

- Given a knapsack with weight capacity $W > 0$.
- A set $S$ of $n$ items with weights $w_i > 0$ and benefits $b_i > 0$ for $i = 1,\ldots,n$.
- $S = \{ (item_1, w_1, b_1), (item_2, w_2, b_2), \ldots, (item_n, w_n, b_n) \}$
- Find a subset of the items which does not exceed the weight $W$ of the knapsack and maximizes the benefit.
0/1 Knapsack problem

Determine a subset $A$ of $\{1, 2, \ldots, n\}$ that satisfies the following:

$$\max \sum_{i \in A} b_i \text{ where } \sum_{i \in A} w_i \leq W$$

In 0/1 knapsack a specific item is either selected or not
Variations of the Knapsack problem

- Fractions are allowed. This applies to items such as:
  - bread, for which taking half a loaf makes sense
  - gold dust
- No fractions.
  - 0/1 (1 brown pants, 1 green shirt…)
  - Allows putting many items of same type in knapsack
    - 5 pairs of socks
    - 10 gold bricks
  - More than one knapsack, etc.
- We will first cover the 0/1 knapsack problem followed by the fractional knapsack problem.
Brute force!

- Generate all $2^n$ subsets
  - Discard all subsets whose sum of the weights exceed $W$ (*not feasible*)
  - Select the maximum total benefit of the remaining (feasible) subsets

- What is the run time?
  - $\Omega(2^n)$
Example with “brute force”

\[ S = \{ (item_1, 5, \$70), (item_2, 10, \$90), (item_3, 25, \$140) \} \]
and \( W = 25 \)

- Subsets:
  1. \{\}
  2. \{ (item_1, 5, \$70) \}  \quad \text{Profit=\$70}
  3. \{ (item_2, 10, \$90) \}  \quad \text{Profit=\$90}
  4. \{ (item_3, 25, \$140) \}  \quad \text{Profit=\$140}
  5. \{ (item_1, 5, \$70), (item_2, 10, \$90) \}.  \text{Profit=\$160 ****}
  6. \{ (item_2, 10, \$90), (item_3, 25, \$140) \} exceeds \( W \)
  7. \{ (item_1, 5, \$70), (item_3, 25, \$140) \} exceeds \( W \)
  8. \{ (item_1, 5, \$70), (item_2, 10, \$90), (item_3, 25, \$140) \} exceeds \( W \)
Greedy approach for 0/1 Knapsack?

- It falls short! We will see examples in the following slides.
Greedy 1: Max benefit first – Counter example

\[ S = \{ (item_1, 5, \$70), (item_2, 10, \$90), (item_3, 25, \$140) \} \]
Greedy 2: Minimum weight first – Counter example

\[ S = \{ (item_1, 5, \$150), (item_2, 10, \$60), (item_3, 20, \$140) \} \]
Greedy 3: Max weight first
Counter Example

\[ S = \{ (item_1, 5, \$150), (item_2, 10, \$60), (item_3, 20, \$140) \} \]
Greedy 4: Maximum benefit per unit item -- Counter Example

\[ S = \{ (item_1, 5, $50), (item_2, 20, $140), (item_3, 10, $60) \} \]
Approximation algorithms

- Approximation algorithms are not guaranteed to provide an optimal solution, but yields that are reasonably close to optimal solutions.

- Let ApproxAlg represent a solution provided by an approximate algorithm. How far is the solution ApproxAlg away from the optimum OPT in the worst case?

- Many criteria are used. We use OPT/ApproxAlg for maximization, and attempt to establish OPT/ApproxAlg ≤ K where K is a constant (ApproxAlg/OPT for minimization)
Approximation algorithms

- The following slides show that the “best” greedy algorithm for 0/1 knapsack
  - Greedy 4 does not satisfy $\text{OPT}/\text{ApproxAlg} \leq K$
  - Often greedy4 gives an optimal solutions, but for some problem instances the ratio can become very large
  - A small modification of greedy4, however, guarantees that $\text{OPT}/\text{ApproxAlg} \leq 2$
  - This is a big improvement
Approximation algorithms

• Use greedy 4: Select the item with *maximum benefit per unit first*

• Example where greedy4 provides a very poor solution:
  • Assume a 0/1 knapsack problem with \( n=2 \)
  • Very large \( W \).
  • \( S=\{( \text{item1, 1, $2$}, \ ( \text{item 2, } W, $1.5W) \} \)
  • The solution to greedy4 has a benefit of $2
  • An optimal solution has a benefit of $1.5W$.
  • If we want the best investment and we have \( W=10,000 \). We should choose the 2nd one with a profit of $15,000, and not the first with a profit of $2.
Approximation Continued

- Let $B_{Opt}$ denote the optimal benefit for the 0/1 knapsack problem.

- Let $B_{Greedy4}$ be the benefit calculated by greedy4.
  - For last example $B_{Opt} / B_{Greedy4} = \frac{1.5W}{2}$
  - Note: $W$ can be arbitrarily large

- We would like to find a better algorithm $Alg$ such that $B_{Opt} / Alg \leq K$ where $K$ is a small constant and is independent of the problem instance.
A Better Approximation Algorithm

- Let $\text{maxB} = \max\{ b_i | i = 1, \ldots, n \}$

- The approximation algorithm selects, either the solution to Greedy4, or only the item with benefit $\text{MaxB}$ depending on $\max\{ \text{BGreedy4}, \text{maxB} \}$.

- Let $\text{APP} = \max\{ \text{BGreedy4}, \text{maxB} \}$

- What is the asymptotic runtime of this algorithm?

- It can be shown that with this modification the ratio $\frac{\text{BOpt}}{\text{APP}} \leq 2$ (Optimal benefit at most twice that of APP)
An Optimal Greedy Algorithm for Knapsack with Fractions (KWF)

In this problem a fraction of any item may be chosen
The following algorithm provides the optimal benefit:

- The greedy algorithm uses the maximum benefit per unit selection criteria
  1. Sort items in decreasing \( \frac{b_i}{w_i} \).

  2. Add items to knapsack (starting at the first) until there are no more items, or the next item to be added exceeds \( W \).

  3. If knapsack is not yet full, fill knapsack with a fraction of next unselected item.
Let $k$ be the index of the last item included in the knapsack. We may be able to include the whole or only a fraction of item $k$

**Without item $k$**

\[ \text{totweight} = \sum_{i=1}^{k-1} w_i \]

**profit**

\[ \text{profitKWF} = \sum_{i=1}^{k-1} p_i + \min\{ (W - \text{totweight}), w_k \} \times \left( \frac{p_k}{w_k} \right) \]

\[ \min\{ (W - \text{totweight}), w_k \} \], means that we either take the whole of item $k$ when the knapsack can include the item without violating the constraint, or we fill the knapsack by a fraction of item $k$. 
Example of applying the optimal greedy algorithm for Fractional Knapsack Problem

\[ S = \{ (item_1, 5, $50), (item_2, 20, $140) (item_3, 10, $60) \} \]
Greedy Algorithm for Knapsack with fractions

- To show that the greedy algorithm finds the optimal profit for the fractional Knapsack problem, you need to prove there is no solution with a higher profit (see text)

- Notice there may be more than one optimal solution
Dynamic programming approach for the 0/1 Knapsack problem

- Show principle of optimality holds
- Discuss the algorithm
Principle of Optimality for 0/1 Knapsack problem

- **Theorem**: 0/1 knapsack satisfies the principle of optimality

- **Proof**: Assume that item \( i \) is in the most beneficial subset that weighs at most \( W \). If we remove item \( i \) from the subset the remaining subset must be the most beneficial subset weighing at most \( W - w_i \) of the \( n - 1 \) remaining items after excluding item \( i \).

- If the remaining subset after excluding item \( i \) was not the most beneficial one weighing at most \( W - w_i \) of the \( n - 1 \) remaining items, we could find a better solution for this problem and improve the optimal solution. This is impossible.
Dynamic Programming Approach

- Given a knapsack problem with $n$ items and knapsack weight of $W$.

- We will first compute the maximum benefit, and then determine the subset.

- To use dynamic programming we solve smaller problems and use the optimal solutions of these problems to find the solution to larger ones.
Dynamic Programming Approach

• What are the smaller problem?
  • Assume a subproblem in which the set of items is restricted to \{1, \ldots, i\} where \( i \leq n \), and the weight of the knapsack is \( w \), where \( 0 \leq w \leq W \).
  • Let \( B[i, w] \) denote the maximum benefit achieved for this problem.
  • Our goal is to compute the maximum benefit of the original problem \( B[n, W] \).
  • We solve the original problem by computing \( B[i, w] \) for \( i = 0, 1, \ldots, n \) and for \( w = 0, 1, \ldots, W \).
  • We need to specify the solution to a larger problem in terms of a smaller one.
Recursive formula for the “smaller” 0/1 Knapsack Problem only using \textit{item}_1 \text{ to } \textit{item}_i \text{ and knapsack weight at most } w

1. If there is no item in the knapsack or } W \text{ is 0, then the benefit is 0

2. If the weight of \textit{item}_i \text{ exceeds the weight } w \text{ of the knapsack then }
\textit{item}_i \text{ cannot be included in the knapsack and the maximum benefit is } B[i-1, w]

3. Otherwise, the benefit is the maximum achieved by either not including \textit{item}_i \text{ (i.e., } B[i-1, w]) \text{ or by including } \textit{item}_i \text{ (i.e., } B[i-1, w- w_i]+b_i)

\[ B[i, w] = \begin{cases} 
0 & \text{for } i = 0 \text{ or } w = 0 \\
B[i-1, w] & \text{if } w_i > w \\
\max\{ B[i-1, w], B[i-1, w-w_i]+b_i \} & \text{otherwise}
\end{cases} \]
Pseudo-code: 0/1 Knapsack 
(n+1)*(W+1) Matrix

Input: W, \{w_1, w_2, \ldots w_n\}, \{b_1, b_2, \ldots b_n\}  
Output: B[n, W]

for w ← 0 to W do // row 0  
    B[0, w] ← 0
for k ← 1 to n do // rows 1 to n  
    B[k, 0] ← 0 // element in column 0
    for w ← 1 to W do // elements in columns 1 to W
        if (w_k ≤ w) and (B[k-1, w - w_k] + b_k > B[k-1, w])
            then  B[k, w] ← B[k-1, w - w_k] + b_k
        else  B[k, w] ← B[k-1, w]
Example:

\[ W = 30, \ S = \{ (i_1, 5, $50), (i_2, 10, $60), (i_3, 20, $140) \} \]

<table>
<thead>
<tr>
<th>Weight:</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>...</th>
<th>30</th>
</tr>
</thead>
<tbody>
<tr>
<td>MaxProfit { }</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>...</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Weight:</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>...</th>
<th>30</th>
</tr>
</thead>
<tbody>
<tr>
<td>MaxProfit { }</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>...</td>
<td>0</td>
</tr>
<tr>
<td>MaxProfit{i_1}</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>50</td>
<td>50</td>
</tr>
</tbody>
</table>


Example continued

$W = 30, S = \{ (i_1, 5, 50), (i_2, 10, 60), (i_3, 20, 140) \}$

<table>
<thead>
<tr>
<th>Weight:</th>
<th>0</th>
<th>4</th>
<th>5</th>
<th>...</th>
<th>9</th>
<th>10</th>
<th>...</th>
<th>14</th>
<th>15</th>
<th>...</th>
<th>30</th>
</tr>
</thead>
<tbody>
<tr>
<td>MaxProfit{ }</td>
<td>0</td>
<td>...</td>
<td>0</td>
<td>0</td>
<td>...</td>
<td>0</td>
<td>0</td>
<td>...</td>
<td>0</td>
<td>0</td>
<td>...</td>
</tr>
<tr>
<td>MaxProfit{$i_1$}</td>
<td>0</td>
<td>...</td>
<td>0</td>
<td>50</td>
<td>...</td>
<td>50</td>
<td>50</td>
<td>...</td>
<td>50</td>
<td>50</td>
<td>...</td>
</tr>
<tr>
<td>MaxProfit{$i_1$, $i_2$}</td>
<td>0</td>
<td>...</td>
<td>0</td>
<td>50</td>
<td>...</td>
<td>50</td>
<td><strong>60</strong></td>
<td>...</td>
<td>60</td>
<td>110</td>
<td>...</td>
</tr>
</tbody>
</table>

- $B[2,10] = \max \{ B[1,10], B[1,10-10] + b_2 \}$
  
  $= 60$

- $B[2,15] = \max \{ B[1,15], B[1,15-10] + b_2 \}$
  
  $= \max \{ 50, 50+60 \}$
  
  $= 110$
Example continued

\( W = 30, S = \{ (i_1, 5, $50), (i_2, 10, $60), (i_3, 20, $140) \} \)

<table>
<thead>
<tr>
<th>Wt:</th>
<th>0...4 5 ... 9 10...14 15... 19 20... 24 25...29 30</th>
</tr>
</thead>
<tbody>
<tr>
<td>MaxP{ }</td>
<td>0...0 0 ... 0 0 ...0 0 ... 0 0... 0 0... 0 0 ... 0 0</td>
</tr>
<tr>
<td>MaxP{(i_1)}</td>
<td>0...0 50...50 50...50 50... 50 50... 50 50... 50 50</td>
</tr>
<tr>
<td>MaxP{(i_1, i_2)}</td>
<td>0...0 50...50 60...60 110...110 110... 110 ... 110</td>
</tr>
<tr>
<td>MaxP{(i_1, i_2, i_3)}</td>
<td>0...0 50...50 60...60 110...110 140...140 190...190 200</td>
</tr>
</tbody>
</table>

  = 140

  = \max \{110, 50+140\}
  = 190

- \(B[3,30] = \max \{ B[2,30], B[2,30-20] + 140 \}\)
  = 200
It is straightforward to fill in the array using the expression on the previous slide. SO What is the size of the array?

- The array is the \((\text{number of items} + 1) \times (W + 1)\).
- So the algorithm runs in \(\Theta(nW)\). It appears to be linear BUT the weight is not a function of only the number of items. What if \(W = n!\) ? Then this algorithm is worse than the brute force method.

- No one has ever found a 0/1 knapsack algorithm whose worst case time is better than exponential AND no one has proven that such an algorithm is not possible.