16. Greedy Algorithms
16.1 An activity--selection problem

- Select a maximum-size subset of mutually compatible activities.
  - An activity set $S = \{a_1, a_2, ..., a_n\}$
  - Each activity $a_i$ has a start time $s_i$ and a finish time $f_i$, where $0 \leq s_i < f_i < \infty$
  - Compatible: if the intervals $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap

- Resource Scheduling
Example

\[ \begin{array}{ccc}
   k & s_k & f_k \\
   1 & 1 & 4 \\
   2 & 3 & 5 \\
   3 & 0 & 6 \\
   4 & 5 & 7 \\
   5 & 3 & 8 \\
   6 & 5 & 9 \\
   7 & 6 & 10 \\
   8 & 8 & 11 \\
   9 & 8 & 12 \\
   10 & 2 & 13 \\
   11 & 12 & 14 \\
\end{array} \]
Dynamic programming version

- Optimal substructure
  - \( S_{ij} = \{ a_k \in S : f_i \leq s_k < f_k \leq s_j \} \)
  - \( a_0 : f_0 = 0, \quad a_{n+1} : s_{n+1} = \infty \)
  - \( S = S_{0,n+1} \)
  - Let us assume that the activities are sorted in increasing order of finish time
    - \( f_0 \leq f_1 \leq f_2 \leq \ldots \leq f_n < f_{n+1} \)
  - \( S_{ij} = \emptyset, \) whenever \( i \geq j \)
  - Given an optimal solution \( A_{ij} \) to \( S_{ij} \) and \( a_k \in A_{ij} \)
    - \( A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}. \)
  - \( S_{0,n+1} : \) an optimal solution to the entire problem
Dynamic programming version (Cont.)

- **Recursive equation**
  - \( c[i,j] \) : the number of activities in a maximum-size subset of mutually compatible activities in \( S_{ij} \)

\[
C[i,j] = \begin{cases} 
0 & \text{if } S_{ij} = \emptyset \\
\max_{i<k<j} \{ c[i,k] + c[k,j] + 1 \} & \text{if } S_{ij} \neq \emptyset
\end{cases}
\]
Greedy solution

Theorem 16.1
- Consider any nonempty subproblem $S_{ij}$, and let $a_m$ be the activity in $S_{ij}$ with the earliest finish time:
  \[ f_m = \min \{ f_k : a_k \text{ in } S_{ij} \} \]
- Then
  - Activity $a_m$ is used in some maximum-size subset of mutually compatible activities of $S_{ij}$.
  - The subproblem $S_{im}$ is empty.

Due to Theorem 16.1,
- Only one subproblem is used in an optimal solution.
- During the solution of subproblem, consider only one choice: the one with the earliest finish time in $S_{ij}$.
- We can solve each subproblem in a top-down fashion.
Greedy algorithms – Recursive version

RECURSIVE-ACTIVITY-SELECTOR(s, f, i, j)
1. $m \leftarrow i+1$
2. while $m < j$ and $s_m < f_i$
   3. do $m \leftarrow m+1$
   4. if $m < j$
   5. then return $\{a_m\} \cup$ RECURSIVE-ACTIVITY-SELECTOR(s, f, m, j)
   6. else return $\emptyset$
Greedy algorithms – Recursive version example

<table>
<thead>
<tr>
<th>k</th>
<th>s_k</th>
<th>f_k</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>9</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>11</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>12</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>13</td>
</tr>
<tr>
<td>11</td>
<td>12</td>
<td>14</td>
</tr>
<tr>
<td>12</td>
<td>∞</td>
<td>-</td>
</tr>
</tbody>
</table>

Diagram showing the recursive version example.
Greedy algorithms – Iterative version

GREEDY- ACTIVITY- SELECTOR(s, f)
1  n ← length[s]
2  A ← \{a_i\}
3  i ← 1
4  for m ← 2 to n
5      do if s_m ≥ f_i
6          then A ← A U \{a_m\}
7                i ← m
8  return A
16.2 Elements of the greedy strategy
Elements of the greedy strategy

- How can one tell if a greedy algorithm will solve a particular optimization problem?
  - There is no way in general
  - But the greedy-choice property and optimal substructure are the two key ingredients
  - If we can demonstrate that the problem has these properties, then we are well on the way to developing a greedy algorithm
Greedy-choice property

- A globally optimal solution can be arrived at by making a locally optimal greedy choice.
- Make the choice that looks best in the current problem, without considering results from subproblems.
Greedy-choice property

- Dynamic Programming
  - The choice at each step usually depends on the solutions to subproblems
  - Consequently, typically solve dynamic-programming problems in a bottom-up manner
  - Progress from smaller subproblems to larger subproblems
Greedy-choice property

- Greedy Algorithm
  - Make whatever choice seems best at the moment
  - And then solve the subproblem arising after the choice is made
  - Usually progress in a top-down fashion
  - Must prove that a greedy choice at each step yields a globally optimal solution
Optimal Substructure

- An optimal solution to the problem contains within it optimal solutions to subproblems
- Key ingredients of assessing the applicability of dynamic programming as well as greedy algorithms
Greedy versus dynamic programming

- The optimal-substructure property is exploited by both the greedy and dynamic-programming

- Because of this, there might be mistake to decide which approach is proper for given problem
Knapsack problem

- 0-1 and fractional knapsack problem
  - Both problems exhibit the optimal-substructure property
  - The fractional knapsack problem is solvable by a greedy strategy
  - The 0-1 knapsack problem is not solvable by a greedy strategy
  - The dynamic-programming is needed to find optimal solution for the 0-1 knapsack problem
0-1 knapsack problem

The value per pound

<table>
<thead>
<tr>
<th>Item</th>
<th>Value</th>
<th>Weight</th>
<th>Value per Pound</th>
</tr>
</thead>
<tbody>
<tr>
<td>Item 1</td>
<td>$60</td>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td>Item 2</td>
<td>$100</td>
<td>5</td>
<td>20</td>
</tr>
<tr>
<td>Item 3</td>
<td>$120</td>
<td>4</td>
<td>30</td>
</tr>
<tr>
<td>Knapsack</td>
<td></td>
<td></td>
<td>50</td>
</tr>
</tbody>
</table>
0-1 knapsack problem

\[
\begin{align*}
30 & \quad 120 \\
20 & \quad 100 \\
\quad & + \\
\quad & = 220
\end{align*}
\]

\[
\begin{align*}
20 & \quad 100 \\
10 & \quad 60 \\
\quad & + \\
\quad & = 160
\end{align*}
\]

\[
\begin{align*}
10 & \quad 60 \\
\quad & + \\
\quad & = 180
\end{align*}
\]
Fractional knapsack problem

```
<table>
<thead>
<tr>
<th>Value</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>$80</td>
<td>20</td>
</tr>
<tr>
<td>$100</td>
<td>20</td>
</tr>
<tr>
<td>$60</td>
<td>10</td>
</tr>
</tbody>
</table>

Total = $240
```
Dynamic programming for 0-1 Knapsack Problem

- Let \( c[i, w] \) = value of solution for items 1...i and maximum weight w

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

<table>
<thead>
<tr>
<th>i</th>
<th>W</th>
<th>0</th>
<th>10</th>
<th>20</th>
<th>30</th>
<th>40</th>
<th>50</th>
</tr>
</thead>
<tbody>
<tr>
<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>0</td>
<td>60</td>
<td>60</td>
<td>60</td>
<td>60</td>
<td>60</td>
<td>60</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>60</td>
<td>100</td>
<td>160</td>
<td>160</td>
<td>160</td>
<td>160</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>60</td>
<td>100</td>
<td>160</td>
<td>180</td>
<td>220</td>
<td></td>
</tr>
</tbody>
</table>
Greedy Algorithm

Huffman Codes
Huffman Codes

- Typically Huffman codes save 20% to 90% of the space.

- Binary Character Code
  In this problem, we only consider about this case.

- Fixed Length Code
  - 3 bits to represent six characters.

- Variable Length Code
  - Considerably better than a fixed length code.

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>freq</td>
<td>45</td>
<td>13</td>
<td>12</td>
<td>16</td>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td>F.C.</td>
<td>000</td>
<td>001</td>
<td>010</td>
<td>011</td>
<td>100</td>
<td>101</td>
</tr>
<tr>
<td>V.C.</td>
<td>0</td>
<td>101</td>
<td>100</td>
<td>111</td>
<td>1101</td>
<td>1100</td>
</tr>
</tbody>
</table>
Prefix Codes

- If we want to encode 16 characters using binary character code, what will be the average length of code?
- Why Prefix Codes?
- Prefix codes: codes in which no codeword is also a prefix of some other codeword.
- Encoding of prefix codes.
  - We must concatenate the codes, and we need a notion to denote concatenation.
- Decoding of prefix codes.
  - Traverse the constructed tree.
- Cost of a tree $T$ corresponding to a prefix code.
  - $B(T) = \sum f(c) d_T(c)$ and $c$ is a character.
Constructing a Huffman Code(1)

Step 1: Make frequency table and sort it.

- f: 5
- e: 9
- c: 12
- B: 13
- d: 16
- a: 45
Constructing a Huffman Code(2)

Step 2: Extract top-two elements and merge into one node.

c:12  b:13  14  d:16  a:45

0

1

f:5  e:9
Constructing a Huffman Code(3)

Step 3: Back to Step 1 until there’s only one element in the Queue.
Constructing a Huffman Code(4)
Constructing a Huffman Code(2)

$C$: a set of $n$ characters
$f(i)$: the frequency of a character
$Q$: a priority queue, keyed on $f(i)$

for $i \leftarrow 1$ to $n - 1$
  do $z \leftarrow$ anode()
      $x \leftarrow$ left$(z) \leftarrow$ ExtractMin$(Q)$
      $y \leftarrow$ right$(z) \leftarrow$ ExtractMin$(Q)$
      $f(z) \leftarrow f(x) + f(y)$
      insert$(Q, z)$
  return ExtractMin$(Q)$
Correctness of Huffman’s Algorithm

- **Greedy Choice**
  - In each step, we select and extract two minimum elements in the queue, merge them into one node and insert the node in the queue again.
  - We can prove that this greedy choice yields globally optimal solution.

- **Optimal Substructure**
  - After greedy choice, the sub-solution must be optimal solution.
  - In this case, sub-tree \( T' \) of \( T \), \( T' = T - \{x, y\} \), represents an optimal prefix code for the alphabet \( C' = C - \{x, y\} \cup \{z\} \). (Let \( z \) be the parent of \( x \) and \( y \), and \( f(z) = f(x) + f(y) \))
  - Huffman’s Algorithm produces an optimal prefix code because it satisfies above two properties.
Proof of Greedy Choice Property(1)

Lemma
- There exists an optimal prefix code for $C$ in which the code-words for $x$ and $y$ have the same length and differ only in the last bit.

Proof
- $b, c$: Any characters with $f(b) \leq f(c)$.
- $x, y$: characters with $f(x) \leq f(y) \leq f(b)$.
- Let’s suppose 3 cases:
  - An optimal tree $T$, in which $b, c$ are in the deepest leaf.
  - A tree $T'$, in which $b$ and $x$ are exchanged their position in $T$.
  - A tree $T''$, in which $y$ and $c$ are exchanged their position in $T'$. 
Proof of Greedy Choice
Property(2)

- $B(T) - B(T') \geq 0 \& B(T) - B(T'') \geq 0$, and $B(T) - B(T'') \geq 0$.
- $B(T) - B(T') = f(x)d_T(x) + f(b)d_T(b) - f(x)d_{T'}(x) - f(b)d_{T'}(b)$
- $= f(x)d_T(x) + f(b)d_T(b) - f(x)d_T(b) - f(b)d_T(x)$
- $= (f(b) - f(x))(d_T(b) - d_T(x)) \geq 0$

And $B(T') - B(T'')$ can be calculated easily in this way.

- But $B(T)$ is the minimum cost tree so, $B(T) = B(T'')$

- So there must be a optimal prefix code tree which has the two least frequent elements in the deepest node.

- This lemma guarantees that the selection of the two least frequent elements at one step must be in the optimal solutions.
Lemma 16.3 - Optimal Substructure

Let C be a given alphabet with frequency f[c] defined for each character c ∈ C. Let x and y be two characters in C with minimum frequency. Let C’ be the alphabet C with characters x, y removed and (new) character z added, so that C’ = C - {x, y}∪ {z}; define f for C’ as for C, except that f[z] = f[x] + f[y]. Let T’ be any tree representing an optimal prefix code for the alphabet C’. Then the tree T, obtained from T’ by replacing the leaf node for z with an internal node having x and y as children, represents an optimal prefix code for the alphabet C.
Proof of Optimal Substructure (1)

$T$: full binary tree representing an optimal prefix code over an alphabet $C$.

$f(z) = f(x) + f(y)$, $T' = T - \{x, y\}$, $C' = C - \{x, y\} \cup \{z\}$

$B(T)$ is the cost of $T$ and $B(T')$ is the cost of $T'$

For each $c \in C - \{x, y\}$, $d_T'(c) = d_T(c)$

\[
B(T) - B(T') = f(x)d_T'(x) + f(y)d_T'(y) - f(z)d_T(z) \\
= f(x)d_T'(x) + f(y)d_T'(y) - (f(x) + f(y))d_T(z) \\
= f(x)d_T'(x) + f(y)d_T'(y) - (f(x) + f(y))(d_T(x) - 1) \\
= f(x) + f(y)
\]
Proof of Optimal Substructure(2)

Let’s suppose that $T'$ is the optimal prefix code for $C$, not $T$! That is $B(T') < B(T)$. Add $x$ and $y$ as the children of the $z$ in $T'$. Make a new tree $T''$ for character set $C$. Then, $B(T'') = B(T') + f(x) + f(y) < B(T)$

→ This contradicts the optimality of $T$.

So $T'$ must be optimal for the alphabet $C$!
Proof of optimality of the Huffman

Then we can get the optimal prefix code for \( C \) using *Huffman*.

- **The greedy choice property**
  - There must be an optimal code for two least frequent elements that have the same length and differ only in the last 1 bit.
  - So we build one node using the two least frequent elements, and instead of the two elements, insert the new node with the frequency that's the sum of the two elements.

- **The optimal substructure property**
  - A tree that's constructed using the remaining elements, must be optimal, too.
  - So if we do this step repeatedly, we can get the optimal prefix code.