Chapter 8
Greedy Algorithms

The Greedy Approach

- As in the case of dynamic programming algorithms, greedy algorithms are usually designed to solve optimization problems in which a quantity is to be minimized or maximized.
- Unlike dynamic programming algorithms, greedy algorithms typically consist of an iterative procedure that tries to find a local optimal solution.
- In some instances, these local optimal solutions translate to global optimal solutions. In others, they fail to give optimal solutions.

The Standard Knapsack Problem

- Given \( n \) items of sizes \( s_1, s_2, ..., s_n \) and values \( v_1, v_2, ..., v_n \) and size \( C \), the knapsack capacity, the objective is to find integers \( x_1, x_2, ..., x_n \) in \( \{0, 1\} \) that maximize the sum
  \[
  \sum_{i=1}^{n} x_i v_i
  \]
  subject to the constraint
  \[
  \sum_{i=1}^{n} x_i s_i \leq C
  \]

The Fractional Knapsack Problem

- Given \( n \) items of sizes \( s_1, s_2, ..., s_n \) and values \( v_1, v_2, ..., v_n \) and size \( C \), the knapsack capacity, the objective is to find nonnegative real numbers \( x_1, x_2, ..., x_n \) between 0 and 1 that maximize the sum
  \[
  \sum_{i=1}^{n} x_i v_i
  \]
  subject to the constraint
  \[
  \sum_{i=1}^{n} x_i s_i \leq C
  \]

The Greedy Approach

- A greedy algorithm makes a correct guess on the basis of little calculation without worrying about the future. Thus, it builds a solution step by step. Each step increases the size of the partial solution and is based on local optimization.
- The choice make is that which produces the largest immediate gain while maintaining feasibility.
- Since each step consists of little work based on a small amount of information, the resulting algorithms are typically efficient.
The Standard Knapsack Problem

- Example:
  - Optimal Solution: \{3, 4\} has value 40.
- Greedy: repeatedly add item with maximum \( \frac{v_i}{w_i} \).
- Greedy Solution:
  - \{5, 2, 1\} achieves only value = 35 \(\Rightarrow\) greedy not optimal.

The Shortest Path Problem

- The set of vertices is partitioned into two sets \(X\) and \(Y\) so that \(X\) is the set of vertices whose distance from the source has already been determined, while \(Y\) contains the rest vertices. Thus, initially \(X = \{1\}\) and \(Y = \{2, 3, \ldots, n\}\).
- Associated with each vertex \(y\) in \(Y\) is a label \(\lambda[y]\), which is the length of a shortest path that passes only through vertices in \(X\). Thus, initially
  \[
  \lambda[1] = 0, \quad \lambda[i] = \begin{cases} 
  \text{length}(1, i) & \text{if } (1, i) \in E \\
  \infty & \text{if } (1, i) \notin E
  \end{cases}, \quad 2 \leq i \leq n
  \]
- At each step, we select a vertex \(y \in Y\) with minimum \(\lambda[y]\) and move it to \(X\), and \(\lambda\) of each vertex \(w \in Y\) that is adjacent to \(y\) is updated indicating that a shorter path to \(w\) via \(y\) has been discovered.
- Finally, \(\lambda\) of each vertex in \(X\) is the distance from the source vertex to this one.

The Shortest Path Problem

- Input: A weighted directed graph \(G = (V, E)\), where \(V = \{1, 2, \ldots, n\}\);
- Output: The distance from vertex 1 to every other vertex in \(G\);
- Example:
- Example:
- Example:
Dijkstra's Shortest Path Algorithm

Find shortest path from \( s = 1 \) to \( t = 8 \).

X = \{ s \}

Y = \{ 2, 3, 4, 5, 6, 7, t \}

distance label

X = \{ s, 2 \}

Y = \{ 3, 4, 5, 6, 7, t \}

distance label

distance label

distance label

distance label

X = \{ s, 2 \}

Y = \{ 3, 4, 5, 6, 7, t \}

distance label

distance label

distance label

distance label

X = \{ s, 2 \}

Y = \{ 3, 4, 5, 6, 7, t \}

distance label

distance label

distance label

distance label

X = \{ s, 2 \}

Y = \{ 3, 4, 5, 6, 7, t \}

distance label

distance label

distance label

distance label

X = \{ s, 2 \}

Y = \{ 3, 4, 5, 6, 7, t \}

distance label

distance label

distance label

distance label

X = \{ s, 2 \}

Y = \{ 3, 4, 5, 6, 7, t \}

distance label

distance label

distance label

distance label

X = \{ s, 2 \}

Y = \{ 3, 4, 5, 6, 7, t \}

distance label

distance label

distance label

distance label

X = \{ s, 2 \}

Y = \{ 3, 4, 5, 6, 7, t \}

distance label

distance label

distance label

distance label

X = \{ s, 2 \}

Y = \{ 3, 4, 5, 6, 7, t \}

distance label

distance label

distance label

distance label
Dijkstra’s Shortest Path Algorithm

X = {s, 2, 3, 4, 5, 6, 7, t}
Y = {t}

delmin
Dijkstra’s Algorithm: Proof of Correctness

Invariant: For each node \( u \in X \), \( \lambda(u) \) is the length of the shortest \( s-u \) path.

Pf. (by induction on \(|X|\))

Base case: \(|X| = 1\) is trivial: \( \lambda(s) = 0 \).

Inductive hypothesis: Assume true for \(|X| = k \geq 1\).

Inductive case:
- Let \( v \) be next node added to \( X \), and let \( u-v \) be the chosen edge.
- The shortest \( s-u \) path plus \( (u, v) \) is an \( s-v \) path of length \( \lambda(v) \).
- Consider any \( s-v \) path \( P \). We’ll see that it’s no shorter than \( \lambda(v) \).
- Let \( x-y \) be the first edge in \( P \) that leaves \( X \), and let \( P' \) be the subpath to \( x \).
- \( P \) is already too long as soon as \( P \) leaves \( X \).
- So \( s-u-v \) is the shortest path among all the \( s-v \) paths.

Dijkstra’s Algorithm: Implementation

For each unexplored node in \( Y \), explicitly maintain \( \lambda(v) \).

Next node to explore = node with minimum \( \lambda(v) \).

When exploring \( v \), for each incident edge \( e = (v, w) \), update \( \lambda(w) \).

Efficient implementation. Maintain a priority queue of unexplored nodes \( Y \), prioritized by \( \lambda(v) \).

**Priority Queue (PQ) Operations**

- **Insert**
- **ExtractMin**
- **ChangeKey**

**Binary heap**

- \( \log n \)
- \( \log n \)
- \( \log n \)

**Array**

- \( n \)
- \( n \)
- \( 1 \)

**Total**

- \( m \log n \)
- \( n^2 \)

Dijkstra’s Algorithm

For each unexplored node in \( Y \), explicitly maintain \( \lambda(v) \).
- Next node to explore: \( v \) with minimum \( \lambda(v) \).
- When exploring \( v \), for each incident edge \( e = (v, w) \), update \( \lambda(w) \).

Minimum Cost Spanning Trees

- Let \( G=(V, E) \) be a connected undirected graph with weights on its edges.
- A spanning tree \( (V, T) \) of \( G \) is a subgraph of \( G \) that is a tree containing all the vertices.
- If \( G \) is weighted and the sum of the weights of the edges in \( T \) is minimum, then \( (V, T) \) is called a minimum cost spanning tree or simply a minimum spanning tree.

Algorithm Characteristics

- Both Prim’s and Kruskal’s Algorithms work with undirected graphs.
- Both work with weighted and unweighted graphs but are more interesting when edges are weighted.
- Both are greedy algorithms that produce optimal solutions.

Prim’s Algorithm

- Similar to Dijkstra’s Algorithm except that \( \lambda(v) \) records edge weights, not path lengths.

**Input:** A weighted undirected graph \( G=(V, E) \), \( V = \{1, 2, ..., n\} \);

**Output:** The MST recorded in \( p[v] \);

1. \( X = \{1\}; \ Y \leftarrow \emptyset; \ \lambda(1) \leftarrow 0; \)
2. for \( y \) from 2 to \( n \)
3.   if \( y \) is adjacent to \( 1 \) \( \{ \lambda(y) \leftarrow \min(\lambda(y), \lambda(1)+\text{length}(1, y)); p(y) \leftarrow 1 \} \)
4.   else \( \lambda(y) \leftarrow \infty; \)
5. for \( j \) from 2 to \( n \)
6.   let \( y \in Y \) be such that \( \lambda(y) \) is minimum;
7.   \( X \leftarrow X \cup \{y\}; \ \text{add } y \text{ to } X \)
8.   \( Y \leftarrow Y \setminus \{y\}; \ \text{delete } y \text{ from } Y \)
9. for each edge \( (y, w) \)
10.   if \( y = 1 \) and \( \lambda(y)+\text{length}(y, w) < \lambda(w) \)
11.     \{ \lambda(w) \leftarrow \min(\lambda(w), \lambda(y)+\text{length}(y, w)); p[w] \leftarrow y; \} \)
Start with any node, say D

Select node with minimum distance

Update distances of adjacent, unselected nodes

Select node with minimum distance

Update distances of adjacent, unselected nodes

Select node with minimum distance

Update distances of adjacent, unselected nodes

An Example
Select node with minimum distance

<table>
<thead>
<tr>
<th>Node</th>
<th>d</th>
<th>( p_d )</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>10</td>
<td>F</td>
</tr>
<tr>
<td>B</td>
<td>4</td>
<td>C</td>
</tr>
<tr>
<td>C</td>
<td>3</td>
<td>D</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>E</td>
<td>7</td>
<td>G</td>
</tr>
<tr>
<td>F</td>
<td>3</td>
<td>C</td>
</tr>
<tr>
<td>G</td>
<td>2</td>
<td>D</td>
</tr>
<tr>
<td>H</td>
<td>3</td>
<td>G</td>
</tr>
</tbody>
</table>

Table entries unchanged

Update distances of adjacent, unselected nodes

<table>
<thead>
<tr>
<th>Node</th>
<th>d</th>
<th>( p_d )</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>10</td>
<td>F</td>
</tr>
<tr>
<td>B</td>
<td>4</td>
<td>C</td>
</tr>
<tr>
<td>C</td>
<td>3</td>
<td>D</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>E</td>
<td>7</td>
<td>G</td>
</tr>
<tr>
<td>F</td>
<td>3</td>
<td>C</td>
</tr>
<tr>
<td>G</td>
<td>2</td>
<td>D</td>
</tr>
<tr>
<td>H</td>
<td>3</td>
<td>G</td>
</tr>
</tbody>
</table>
Minimum Cost Spanning Trees (Kruskal’s Algorithm)

- Kruskal’s algorithm works by maintaining a forest consisting of several spanning trees that are gradually merged until finally the forest consists of exactly one tree.

- The algorithm starts by sorting the edges in nondecreasing order by weight.

Minimum Cost Spanning Trees (Kruskal’s Algorithm)

- Next, starting from the forest \((V, T)\) consisting of the vertices of the graph and none of its edges, the following step is repeated until \((V, T)\) is transformed into a tree: Let \((V, T)\) be the forest constructed so far, and let \(e \in E - T\) be the current edge being considered. If adding \(e\) to \(T\) does not create a cycle, then include \(e\) in \(T\); otherwise discard \(e\).

- This process will terminate after adding exactly \(n - 1\) edges.
Kruskal’s Algorithm

Work with edges, rather than nodes
Two steps:
  – Sort edges by increasing edge weight
  – Select the first |V| – 1 edges that do not generate a cycle

Walk-Through

Consider an undirected, weight graph

Sort the edges by increasing edge weight

Select first |V|–1 edges which do not generate a cycle

Accepting edge (E,G) would create a cycle
Select first \(|V| - 1\) edges which do not generate a cycle.

```text
\begin{array}{|c|c|c|}
\hline
\text{edge} & d & v \\
\hline
(D,E) & 1 & \checkmark \\
(D,G) & 2 & \checkmark \\
(E,G) & 3 & \checkmark \\
(C,D) & 3 & \checkmark \\
(G,H) & 3 & \checkmark \\
(C,F) & 3 & \checkmark \\
(B,C) & 4 & \checkmark \\
\hline
\end{array}
```

Select first \(|V| - 1\) edges which do not generate a cycle.

```text
\begin{array}{|c|c|c|}
\hline
\text{edge} & d & v \\
\hline
(D,E) & 1 & \checkmark \\
(D,G) & 2 & \checkmark \\
(E,G) & 3 & \checkmark \\
(C,D) & 3 & \checkmark \\
(G,H) & 3 & \checkmark \\
(C,F) & 3 & \checkmark \\
(B,C) & 4 & \checkmark \\
\hline
\end{array}
```

Select first \(|V| - 1\) edges which do not generate a cycle.

```text
\begin{array}{|c|c|c|}
\hline
\text{edge} & d & v \\
\hline
(D,E) & 1 & \checkmark \\
(D,G) & 2 & \checkmark \\
(E,G) & 3 & \checkmark \\
(C,D) & 3 & \checkmark \\
(G,H) & 3 & \checkmark \\
(C,F) & 3 & \checkmark \\
(B,C) & 4 & \checkmark \\
\hline
\end{array}
```

Select first \(|V| - 1\) edges which do not generate a cycle.

```text
\begin{array}{|c|c|c|}
\hline
\text{edge} & d & v \\
\hline
(D,E) & 1 & \checkmark \\
(D,G) & 2 & \checkmark \\
(E,G) & 3 & \checkmark \\
(C,D) & 3 & \checkmark \\
(G,H) & 3 & \checkmark \\
(C,F) & 3 & \checkmark \\
(B,C) & 4 & \checkmark \\
\hline
\end{array}
```

Select first \(|V| - 1\) edges which do not generate a cycle.

```text
\begin{array}{|c|c|c|}
\hline
\text{edge} & d & v \\
\hline
(D,E) & 1 & \checkmark \\
(D,G) & 2 & \checkmark \\
(E,G) & 3 & \checkmark \\
(C,D) & 3 & \checkmark \\
(G,H) & 3 & \checkmark \\
(C,F) & 3 & \checkmark \\
(B,C) & 4 & \checkmark \\
\hline
\end{array}
```
Select first $|V| - 1$ edges which do not generate a cycle

Kruskal’s Algorithm

Two steps:
- Sort edges by increasing edge weight
- Select the first $|V| - 1$ edges that do not generate a cycle

How to show the algorithm is correct?

File Compression

Motivation

The motivations for data compression are obvious:

- reducing the space required to store files on disk or tape
- reducing the time to transmit large files.

Huffman savings are between 20% - 90%
**Basic Idea:**

Let the set of characters in the file be $C = \{c_1, c_2, \ldots, c_n\}$. Let also $f(c_i)$, $1 \leq i \leq n$, be the frequency of character $c_i$ in the file, i.e., the number of times $c_i$ appears in the file.

It uses a **variable-length** code table for encoding a source symbol (such as a character in a file) where the variable-length code table has been derived in a particular way based on the frequency of occurrence for each possible value of the source symbol.

---

**Example:**

Suppose you have a file with 100K characters.

For simplicity assume that there are only 6 distinct characters in the file from $a$ through $f$, with frequencies as indicated below.

We represent the file using a unique binary string for each character.

<table>
<thead>
<tr>
<th>Frequency (in 100s)</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed-length codeword</td>
<td>45</td>
<td>13</td>
<td>12</td>
<td>16</td>
<td>9</td>
<td>5</td>
</tr>
</tbody>
</table>

Space = $(45 \times 3 + 13 \times 3 + 12 \times 3 + 16 \times 3 + 9 \times 3 + 5 \times 3) \times 1000$

= **300K bits**

Can we do better ??

**YES !!**

By using **variable-length** codes instead of fixed-length codes.

Idea: Giving frequent characters **short** codewords, and infrequent characters long codewords.

i.e. The length of the encoded character is inversely proportional to that character’s frequency.

<table>
<thead>
<tr>
<th>Character</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency</td>
<td>45</td>
<td>13</td>
<td>12</td>
<td>16</td>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td>Fixed-length codeword</td>
<td>000</td>
<td>001</td>
<td>010</td>
<td>011</td>
<td>100</td>
<td>101</td>
</tr>
<tr>
<td>Variable-length codeword</td>
<td>0</td>
<td>101</td>
<td>100</td>
<td>111</td>
<td>1101</td>
<td>1100</td>
</tr>
</tbody>
</table>

Space = $(45 \times 1 + 13 \times 3 + 12 \times 3 + 16 \times 3 + 9 \times 4 + 5 \times 4) \times 1000$

= **224K bits** (Savings = 25%)

**PREFIX CODES:**

Codes in which no codeword is also a **prefix** of some other codeword.

(*“prefix-free codes” would have been a more appropriate name)*

No **Ambiguity** !!

It is possible to show (although we won’t do so here) that the optimal data compression achievable by a character code can always be achieved with a prefix code, so there is no loss of generality in restricting attention to prefix codes.

Benefits of using **Prefix Codes**:

Example:

<table>
<thead>
<tr>
<th>Character</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable-length codeword</td>
<td>0</td>
<td>101</td>
<td>100</td>
<td>111</td>
<td>1101</td>
<td>1100</td>
</tr>
</tbody>
</table>

FACE = 1100 0 100 1101

To decode, we have to decide where each code begins and ends, since they are no longer all the same length. But this is easy, since, no codes share a prefix. This means we need only scan the input string from left to right, and as soon as we recognize a code, we can print the corresponding character and start looking for the next code.

In the above case, the only code that begins with "1100.." or a prefix is "f", so we can print "f" and start decoding "0100...", get 'a', etc.

Benefits of using **Prefix Codes**:

Example:

To see why the no-common prefix property is essential, suppose that we encoded "e" with the shorter code "10".

<table>
<thead>
<tr>
<th>Character</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable-length codeword</td>
<td>0</td>
<td>101</td>
<td>100</td>
<td>111</td>
<td>1101</td>
<td>1110</td>
</tr>
</tbody>
</table>

FACE = 11000100110

When we try to decode "1100", we could not tell whether 1100 = "f" or 1100 = 110 + 0 = "ea"
**Representation:**

The Huffman algorithm is represented as:

- **binary tree**
- each edge represents either 0 or 1
  
  - 0 means "go to the left child"
  
  - 1 means "go to the right child."

- each leaf corresponds to the sequence of 0s and 1s traversed from the root to reach it, i.e. a particular code.

Since no prefix is shared, all legal codes are at the leaves, and decoding a string means following edges, according to the sequence of 0s and 1s in the string, until a leaf is reached.

**Optimal Code**

An optimal code for a file is always represented by a **full binary tree**, in which every non-leaf node has two children.

The fixed-length code in our example is not optimal since its tree, is not a full binary tree: there are codewords beginning 10 . . . , but none beginning 11 . . . .

Since we can now restrict our attention to full binary trees, we can say that if C is the alphabet from which the characters are drawn, then the tree for an optimal prefix code has exactly |C| leaves, one for each letter of the alphabet, and exactly |C| - 1 internal nodes.

**Constructing a Huffman code**

Huffman invented a greedy algorithm that constructs an optimal prefix code called a **Huffman code**. The algorithm builds the tree T corresponding to the optimal code in a bottom-up manner.

It begins with a set of |C| leaves and performs a sequence of |C| - 1 "merging" operations to create the final tree.

**Greedy Choice?**

The two smallest nodes are chosen at each step, and this local decision results in a globally optimal encoding tree.

In general, greedy algorithms use local minimal/maximal choices to produce a global minimum/maximum.

**Given a tree T corresponding to a prefix code, it is a simple matter to compute the number of bits required to encode a file.**

For each character c in the alphabet C,

- \( f(c) \) denote the frequency of c in the file
- \( d(c) \) denote the depth of c’s leaf in the tree.

\( d(c) \) is also the length of the codeword for character c.

The number of bits required to encode a file is thus

\[
R(T) = \sum_{c \in C} f(c) \cdot d(c)
\]

which we define as the cost of the tree.
The steps of Huffman’s algorithm

Running Time Analysis

Assumes that Q is implemented as a binary min-heap.

- For a set C of n characters, the initialization of Q in line 2 can be performed in \( O(n) \) time using the BUILD-MIN-HEAP procedure.
- The for loop in lines 3-8 is executed exactly \( n - 1 \) times. Each heap operation requires time \( O(\log n) \). The loop contributes \( (n - 1) \times O(\log n) \) = \( O(n\log n) \)

Thus, the total running time of HUFFMAN on a set of \( n \) characters = \( O(n) + O(n\log n) \) = \( O(n \log n) \)

Correctness of Huffman’s algorithm

To prove that the greedy algorithm HUFFMAN is correct, we show that the problem of determining an optimal prefix code exhibits the greedy-choice and optimal-substructure properties.

The Greedy-Choice Property

Lemma 1:

Let \( C \) be an alphabet in which each character \( c \) in \( C \) has frequency \( f(c) \). Let \( x \) and \( y \) be two characters in \( C \) having the lowest frequencies. Then 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 Idea of Lemma 1:
The idea of the proof is to take the tree \( T \) representing an arbitrary optimal prefix code and modify it to make a tree representing another optimal prefix code such that the characters \( x \) and \( y \) appear as sibling leaves of maximum depth in the new tree. If we can do this, then their code words will have the same length and differ only in the last bit.

Proof of Lemma 1:

Let \( a \) and \( b \) be two characters that are are sibling leaves of maximum depth in \( T \), and \( x \) and \( y \) are the two characters of the minimum frequency. Without loss in generality, assume that \( f(x) < f(y) < f(a) < f(b) \). Then we must have \( d_r(x) = d_r(y) = d_r(a) = d_r(b) \).

Proof: Exchange the positions of \( a \) and \( x \) in \( T \), to produce \( T' \). The difference in cost between \( T \) and \( T' \) is

\[
B(T) - B(T') = \sum f(c) d_r(c) - \sum f(c) d_r(c) = (f(x) d_r(x) + f(a) d_r(a) - f(x) d_r(a) - f(a) d_r(x)) = ((f_a - f_x)) d_r(x) > 0 \]  

// if \( f(x) \) is min and \( d_r(a) \) is max

So \( B(T) > B(T') \) and \( d_r(a) = d_r(x) \) because \( B(T) \) is optimal (minimal).
Similarly exchanging the positions of $b$ and $y$ in $T'$, to produce $T''$ does not increase the cost,

$$B(T') - B(T'') = 0.$$  
Since $T$ is optimal, so is $T'$ and $T''$.  
Thus, $T''$ is an optimal tree in which $x$ & $y$ appear as sibling leaves of maximum depth from which Lemma 1 follows.

**Lemma 2:** Let $C$ be a given alphabet with frequency $f[c]$ defined for each character $c \in C$. Let $x$ and $y$ be two characters in $C$ with minimum frequency. Let $C'$ be the alphabet $C$ with characters $x$, $y$ removed and a new character $z$ added, so that $C' = C - \{x, y\} \cup \{z\}$; 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:**
We first express $B(T)$ in terms of $B(T')$.

For $c \in C - \{x, y\}$ we have $d_T(c) = d_{T'}(c)$, and hence

$$f[c]d_T(c) = f[c]d_{T'}(c).$$

From which we conclude that

$$B(T) = B(T') + f[z]d_T(z) = B(T') + (f[x] + f[y]).$$

**Proof of Claim by contradiction**
Suppose that $T$ does not represent an optimal prefix code for $C$. Then there exists a tree Opt such that $B(\text{Opt}) < B(T)$.

Without loss in generality (by Lemma 1) Opt has $x$ & $y$ as siblings. Let $T''$ be the tree Opt with the common parent of $x$ & $y$ replaced by a leaf $z$ with frequency $f[z] = f[x] + f[y]$.

Then, $B(T'') = B(\text{Opt}) - f[z]d_T(z) = B(\text{Opt}) - (f[x] + f[y]) < B(T) - (f[x] + f[y]) = B(T) - (f[x] + f[y]) = B(T')$.

Yielding a contradiction to the assumption that $T'$ represents an optimal prefix code for $C'$. Thus, $T$ must represent an optimal prefix code for the alphabet $C$.

**Drawbacks**
The main disadvantage of Huffman’s method is that it makes two passes over the data:

- one pass to collect frequency counts of the letters in the message, followed by the construction of a Huffman tree and transmission of the tree to the receiver; and
- a second pass to encode and transmit the letters themselves, based on the static tree structure.

This causes delay when used for network communication, and in file compression applications the extra disk accesses can slow down the algorithm.

We need one-pass methods, in which letters are encoded “on the fly”.

**File Compression**

- **An Example:**
  
  $C = \{a, b, c, d, e\}$
  
  $f[\text{a}] = 20$
  
  $f[\text{b}] = 7$
  
  $f[\text{c}] = 10$
  
  $f[\text{d}] = 4$
  
  $f[\text{e}] = 18$
How to Show a Greedy Method is Optimal?

In general, a greedy method is simple to describe, efficient to run, but difficult to prove.

To show a greedy method is not optimal, we need to find a counterexample.

To show a greedy method is indeed optimal, we use the following proof strategy:

Suppose $S$ is the solution found by the greedy method and $Opt$ is an optimal solution that differs from $S$ minimally. If $S = Opt$, we are done. If not, we "modify" $Opt$ to obtain another optimal solution $Opt'$, such that $Opt'$ has less difference than $Opt$ comparing to $S$. That's a contradiction to the assumption that $Opt$ differs from $S$ minimally.