Overview

- One of the strategies used to solve *optimization problems*
  - Multiple solutions exist; pick one of low (or least) cost
- *Greedy strategy*: make a locally optimal choice, or simply, what appears best at the moment
Overview

One of the strategies used to solve optimization problems

- Multiple solutions exist; pick one of low (or least) cost

Greedy strategy: make a locally optimal choice, or simply, what appears best at the moment

Often, locally optimality $\not\Rightarrow$ global optimality

So, use with a great deal of care

- Always need to prove optimality
Overview

- One of the strategies used to solve *optimization problems*
  - Multiple solutions exist; pick one of low (or least) cost

- *Greedy strategy*: make a locally optimal choice, or simply, what appears best at the moment

- Often, *locally optimality $\not\Rightarrow$ global optimality*

- So, use with a great deal of care
  - *Always need to prove optimality*

- If it is unpredictable, why use it?
  - *It simplifies the task!*
Making change

Given coins of denominations 25¢, 10¢, 5¢ and 1¢, make change for x cents (0 < x < 100) using *minimum number of coins*.

Greedy solution

```python
makeChange(x)
if (x = 0) return
Let y be the largest denomination that satisfies y ≤ x
Issue ⌊x/y⌋ coins of denomination y
makeChange(x mod y)
```
Making change

Given coins of denominations 25¢, 10¢, 5¢ and 1¢, make change for \( x \) cents (\( 0 < x < 100 \)) using \textit{minimum number of coins}.

Greedy solution

\begin{verbatim}
makeChange(x)
    if (x = 0) return
    Let y be the largest denomination that satisfies \( y \leq x \)
    Issue \( \lfloor x/y \rfloor \) coins of denomination \( y \)
    makeChange(x mod y)
\end{verbatim}

- Show that it is optimal
Making change

Given coins of denominations 25¢, 10¢, 5¢ and 1¢, make change for \( x \) cents (\( 0 < x < 100 \)) using \textit{minimum number of coins}.

Greedy solution

\[
\text{makeChange}(x)
\]

\[
\text{if } (x = 0) \text{ return}
\]

Let \( y \) be the largest denomination that satisfies \( y \leq x \)

Issue \( \lfloor x/y \rfloor \) coins of denomination \( y \)

\[
\text{makeChange}(x \mod y)
\]

- Show that it is optimal
- Is it optimal for arbitrary denominations?
When does a Greedy algorithm work?

Greedy choice property

The greedy (i.e., locally optimal) choice is always consistent with some (globally) optimal solution.
When does a Greedy algorithm work?

Greedy choice property
The greedy (i.e., locally optimal) choice is always consistent with some (globally) optimal solution

What does this mean for the coin change problem?
When does a Greedy algorithm work?

Greedy choice property
The greedy (i.e., locally optimal) choice is always consistent with some (globally) optimal solution.

What does this mean for the coin change problem?

Optimal substructure
The optimal solution contains optimal solutions to subproblems.
When does a Greedy algorithm work?

Greedy choice property

The greedy (i.e., locally optimal) choice is always consistent with some (globally) optimal solution.

What does this mean for the coin change problem?

Optimal substructure

The optimal solution contains optimal solutions to subproblems.

Implies that a greedy algorithm can invoke itself recursively after making a greedy choice.
A sack that can hold a maximum of $x$ lbs

You have a choice of items you can pack in the sack

Maximize the combined “value” of items in the sack

<table>
<thead>
<tr>
<th>item</th>
<th>calories/lb</th>
<th>weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>bread</td>
<td>1100</td>
<td>5</td>
</tr>
<tr>
<td>butter</td>
<td>3300</td>
<td>1</td>
</tr>
<tr>
<td>tomato</td>
<td>80</td>
<td>1</td>
</tr>
<tr>
<td>cucumber</td>
<td>55</td>
<td>2</td>
</tr>
</tbody>
</table>
Knapsack Problem

- A sack that can hold a maximum of $x$ lbs
- You have a choice of items you can pack in the sack
- Maximize the combined “value” of items in the sack

<table>
<thead>
<tr>
<th>item</th>
<th>calories/lb</th>
<th>weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>bread</td>
<td>1100</td>
<td>5</td>
</tr>
<tr>
<td>butter</td>
<td>3300</td>
<td>1</td>
</tr>
<tr>
<td>tomato</td>
<td>80</td>
<td>1</td>
</tr>
<tr>
<td>cucumber</td>
<td>55</td>
<td>2</td>
</tr>
</tbody>
</table>

0-1 knapsack: Take all of one item or none at all

Fractional knapsack: Fractional quantities acceptable
### Knapsack Problem

- A sack that can hold a maximum of $x$ lbs
- You have a choice of items you can pack in the sack
- Maximize the combined “value” of items in the sack

<table>
<thead>
<tr>
<th>item</th>
<th>calories/lb</th>
<th>weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>bread</td>
<td>1100</td>
<td>5</td>
</tr>
<tr>
<td>butter</td>
<td>3300</td>
<td>1</td>
</tr>
<tr>
<td>tomato</td>
<td>80</td>
<td>1</td>
</tr>
<tr>
<td>cucumber</td>
<td>55</td>
<td>2</td>
</tr>
</tbody>
</table>

**0-1 knapsack:** Take all of one item or none at all

**Fractional knapsack:** Fractional quantities acceptable

**Greedy choice:** pick item that maximizes calories/lb
Knapsack Problem

- A sack that can hold a maximum of $x$ lbs
- You have a choice of items you can pack in the sack
- Maximize the combined “value” of items in the sack

<table>
<thead>
<tr>
<th>item</th>
<th>calories/lb</th>
<th>weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>bread</td>
<td>1100</td>
<td>5</td>
</tr>
<tr>
<td>butter</td>
<td>3300</td>
<td>1</td>
</tr>
<tr>
<td>tomato</td>
<td>80</td>
<td>1</td>
</tr>
<tr>
<td>cucumber</td>
<td>55</td>
<td>2</td>
</tr>
</tbody>
</table>

0-1 knapsack: Take all of one item or none at all

Fractional knapsack: Fractional quantities acceptable

Greedy choice: pick item that maximizes calories/lb

Will a greedy algorithm work, with $x = 5$?
Fractional Knapsack

**Greedy choice property**

Proof by contradiction: Start with the assumption that there is an optimal solution that does not include the greedy choice, and show a contradiction.
Fractional Knapsack

Greedy choice property
Proof by contradiction: Start with the assumption that there is an optimal solution that does not include the greedy choice, and show a contradiction.

Optimal substructure
After taking as much of the item with \( j \)th maximal value/weight, suppose that the knapsack can hold \( y \) more lbs. Then the optimal solution for the problem includes the optimal choice of how to fill a knapsack of size \( y \) with the remaining items.
Fractional Knapsack

**Greedy choice property**
Proof by contradiction: Start with the assumption that there is an optimal solution that does not include the greedy choice, and show a contradiction.

**Optimal substructure**
After taking as much of the item with $j$th maximal value/weight, suppose that the knapsack can hold $y$ more lbs.
Then the optimal solution for the problem includes the optimal choice of how to fill a knapsack of size $y$ with the remaining items.

Does not work for 0-1 knapsack because greedy choice property does not hold.
0-1 knapsack is NP-hard, but a pseudo-polynomial algorithm is available.
Spanning Tree

A subgraph of a graph $G = (V, E)$ that includes:

- All the vertices $V$ in the graph
- A subset of $E$ such that these edges form a tree
Spanning Tree

A subgraph of a graph $G = (V, E)$ that includes:

- All the vertices $V$ in the graph
- A subset of $E$ such that these edges form a tree

We consider *connected undirected graphs*, where the second condition for MST can be replaced by
Spanning Tree

A subgraph of a graph $G = (V, E)$ that includes:

- All the vertices $V$ in the graph
- A subset of $E$ such that these edges form a tree

We consider *connected undirected graphs*, where the second condition for MST can be replaced by

- A maximal subset of $E$ such that the subgraph has no cycles
Spanning Tree

A subgraph of a graph $G = (V, E)$ that includes:

- All the vertices $V$ in the graph
- A subset of $E$ such that these edges form a tree

We consider *connected undirected graphs*, where the second condition for MST can be replaced by

- A maximal subset of $E$ such that the subgraph has no cycles
- A subset of $E$ with $|V| - 1$ edges such that the subgraph is connected
Spanning Tree

A subgraph of a graph \( G = (V, E) \) that includes:

- All the vertices \( V \) in the graph
- A subset of \( E \) such that these edges form a tree

We consider \textit{connected undirected graphs}, where the second condition for MST can be replaced by

- A maximal subset of \( E \) such that the subgraph has no cycles
- A subset of \( E \) with \( |V| - 1 \) edges such that the subgraph is connected
- A subset of \( E \) such that there is a unique path between any two vertices in the subgraph
Minimal Spanning Tree (MST)

A spanning tree with \textit{minimal cost}. Formally:

\textbf{Input}: An undirected graph \( G = (V, E) \), a cost function \( w : E \rightarrow \mathbb{R} \).

\textbf{Output}: A tree \( T = (V, E') \) such that \( E' \subseteq E \) that minimizes \( \sum_{e \in E'} w(e) \)
Minimal Spanning Tree (MST)

A tree $T = (V, E)$, with $E \subseteq E$, that minimizes weight $\text{weight}(T) = \sum_{e \in E} w_e$.

In the preceding example, the minimum spanning tree has a cost of 16:

A B C D E F

1 4 2 5

However, this is not the only optimal solution. Can you spot another?

5.1.1 A greedy approach

Kruskal's minimum spanning tree algorithm starts with the empty graph and then selects edges from $E$ according to the following rule.

Repeatedly add the next lightest edge that doesn't produce a cycle.

In other words, it constructs the tree edge by edge and, apart from taking care to avoid cycles, simply picks whichever edge is cheapest at the moment. This is a greedy algorithm: every decision it makes is the one with the most obvious immediate advantage.

Figure 5.1 shows an example. We start with an empty graph and then attempt to add edges in increasing order of weight (ties are broken arbitrarily):

B − C, C − D, B − D, C − F, D − F, E − F, A − D, A − B, C − E, A − C.

The first two succeed, but the third, B − D, would produce a cycle if added. So we ignore it and move along. The final result is a tree with cost 14, the minimum possible.

The correctness of Kruskal's method follows from a certain cut property, which is general enough to also justify a whole slew of other minimum spanning tree algorithms.

Figure 5.1

The minimum spanning tree found by Kruskal's algorithm.

B A 6 5
3
42 FD
C E
5 41 24
B A FD C E

Chapter 5
Greedy algorithms

A game like chess can be won only by thinking ahead: a player who is focused entirely on immediate advantage is easy to defeat. But in many other games, such as Scrabble, it is possible to do quite well by simply making whichever move seems best at the moment and not worrying too much about future consequences.

This sort of myopic behavior is easy and convenient, making it an attractive algorithmic strategy. Greedy algorithms build up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit. Although such an approach can be disastrous for some computational tasks, there are many for which it is optimal. Our first example is that of minimum spanning trees.

5.1 Minimum spanning trees

Suppose you are asked to network a collection of computers by linking selected pairs of them. This translates into a graph problem in which nodes are computers, undirected edges are potential links, and the goal is to pick enough of these edges that the nodes are connected. But this is not all; each link also has a maintenance cost, reflected in that edge's weight. What is the cheapest possible network?

One immediate observation is that the optimal set of edges cannot contain a cycle, because removing an edge from this cycle would reduce the cost without compromising connectivity:

Property 1

Removing a cycle edge cannot disconnect a graph.

So the solution must be connected and acyclic: undirected graphs of this kind are called trees. The particular tree we want is the one with minimum total weight, known as the minimum spanning tree. Here is its formal definition.

Input: An undirected graph $G = (V, E)$; edge weights $w_e$. 
Kruskal’s algorithm

- Start with the empty set of edges
- Repeat: add lightest edge that doesn’t create a cycle

Adds edges $B\rightarrow C$, $C\rightarrow D$, $C\rightarrow F$, $A\rightarrow D$, $E\rightarrow F$

![Graph](image1.png)

Output: A tree $T = (V, E \subseteq E)$, with $E \subseteq E$, that minimizes weight($T$) = $\sum_{e \in E}$ $w(e)$.

In the preceding example, the minimum spanning tree has a cost of 16:

![Graph](image2.png)

However, this is not the only optimal solution. Can you spot another?

5.1.1 A greedy approach

Kruskal’s minimum spanning tree algorithm starts with the empty graph and then selects edges from $E$ according to the following rule.

Repeatedly add the next lightest edge that doesn’t produce a cycle.

In other words, it constructs the tree edge by edge and, apart from taking care to avoid cycles, simply picks whichever edge is cheapest at the moment. This is a greedy algorithm: every decision it makes is the one with the most obvious immediate advantage.

Figure 5.1 shows an example. We start with an empty graph and then attempt to add edges in increasing order of weight (ties are broken arbitrarily):

$B\rightarrow C$, $C\rightarrow D$, $B\rightarrow D$, $C\rightarrow F$, $D\rightarrow F$, $E\rightarrow F$, $A\rightarrow D$, $A\rightarrow B$, $C\rightarrow E$, $A\rightarrow C$.

The first two succeed, but the third, $B\rightarrow D$, would produce a cycle if added. So we ignore it and move along. The final result is a tree with cost 14, the minimum possible.

The correctness of Kruskal’s method follows from a certain cut property, which is general enough to also justify a whole slew of other minimum spanning tree algorithms.
Kruskal’s algorithm

\[ MST(V, E, w) \]

\[
X = \emptyset \\
Q = priorityQueue(E) \quad \text{// from min to max weight} \\
\textbf{while} \ Q \text{ is nonempty} \\
\quad e = deleteMin(Q) \\
\quad \textbf{if} \ e \text{ connects two disconnected components in } (V, X) \quad X = X \cup \{e\}
\]
Kruskal’s: Correctness (by induction)

*Induction Hypothesis:* The first $i$ edges selected by Kruskal’s are included in some MST $T$. 

In the base case, the empty set of edges is always included in any MST. In the induction step, for $i+1$ edges, we show that the $i+1$-th edge chosen by Kruskal’s is included in the MST $T$. Let $e = (v, w)$ be the edge chosen at the $i+1$-th step of Kruskal’s.

**Case 1:** $e \in T$. The induction step is done.

**Case 2:** $e \notin T$. $T$ is a spanning tree, so there is a unique path from $v$ to $w$. At least one edge $e'\in T$ on this path is not in $X$, the set of edges chosen in the first $i$ steps by Kruskal’s. Otherwise, $v$ and $w$ would be connected in $X$, so Kruskal’s won’t choose $e$. Since neither $e$ nor $e'$ are in $X$, and Kruskal’s chose $e$, we have $w(e') \geq w(e)$. Replace $e'$ by $e$ in $T$ to get another spanning tree $T'$.

Either $w(T') < w(T)$, a contradiction to the assumption $T$ is minimal; or $w(T') = w(T)$, and we have another MST $T'$ consistent with $X \cup \{e\}$. In both cases, we have completed the induction step.
Kruskal’s: Correctness (by induction)

*Induction Hypothesis:* The first $i$ edges selected by Kruskal’s are included in some MST $T$

*Base case:* trivial — the empty set of edges is always in any MST.
Kruskal’s: Correctness (by induction)

*Induction Hypothesis:* The first $i$ edges selected by Kruskal’s are included in some MST $T$

*Base case:* trivial — the empty set of edges is always in any MST.

*Induction step:* Show that $i+1$th edge chosen by Kruskal’s is in the MST $T$

*Proof:* Let $e = (v, w)$ be the edge chosen at $i + 1$th step of Kruskal’s.

- **Case 1:** $e \in T$: The induction step is done.
Kruskal’s: Correctness (by induction)

Induction Hypothesis: The first $i$ edges selected by Kruskal’s are included in some MST $T$

Base case: trivial — the empty set of edges is always in any MST.

Induction step: Show that $i+1$th edge chosen by Kruskal’s is in the MST $T$

Proof: Let $e = (v, w)$ be the edge chosen at $i+1$th step of Kruskal’s.

- **Case 1**: $e \in T$: The induction step is done.
- **Case 2**: $e \notin T$: $T$ is a spanning tree: must include a unique path from $v$ to $w$
Kruskal’s: Correctness (by induction)

**Induction Hypothesis:** The first $i$ edges selected by Kruskal’s are included in some MST $T$

**Base case:** trivial — the empty set of edges is always in any MST.

**Induction step:** Show that $i+1$th edge chosen by Kruskal’s is in the MST $T$

**Proof:** Let $e = (v, w)$ be the edge chosen at $i + 1$th step of Kruskal’s.

- **Case 1:** $e \in T$: The induction step is done.

- **Case 2:** $e \notin T$: $T$ is a spanning tree: must include a unique path from $v$ to $w$
  
  - At least one edge $e'$ on this path is not in $X$, the set of edges chosen in the first $i$ steps by Kruskal’s. (Otherwise, $v$ and $w$ will be connected in $X$, so Kruskal’s won’t chose $e$.)
Kruskal’s: Correctness (by induction)

*Induction Hypothesis:* The first $i$ edges selected by Kruskal’s are included in some MST $T$

*Base case:* trivial — the empty set of edges is always in any MST.

*Induction step:* Show that $i+1$th edge chosen by Kruskal’s is in the MST $T$

**Proof:** Let $e = (v, w)$ be the edge chosen at $i + 1$th step of Kruskal’s.

- **Case 1:** $e \in T$: The induction step is done.

- **Case 2:** $e \notin T$: $T$ is a spanning tree: must include a unique path from $v$ to $w$
  
  - At least one edge $e'$ on this path is not in $X$, the set of edges chosen in the first $i$ steps by Kruskal’s. (Otherwise, $v$ and $w$ will be connected in $X$, so Kruskal’s won’t chose $e$.)
  
  - Since neither $e$ nor $e'$ are in $X$, and Kruskal’s chose $e$, $w(e') \geq w(e)$. 


**Kruskal’s: Correctness (by induction)**

*Induction Hypothesis:* The first \( i \) edges selected by Kruskal’s are included in some MST \( T \)

*Base case:* trivial — the empty set of edges is always in any MST.

*Induction step:* Show that \( i+1 \)th edge chosen by Kruskal’s is in the MST \( T \)

*Proof:* Let \( e = (v, w) \) be the edge chosen at \( i + 1 \)th step of Kruskal’s.

- **Case 1:** \( e \in T \): The induction step is done.

- **Case 2:** \( e \notin T \): \( T \) is a spanning tree: must include a unique path from \( v \) to \( w \)
  - At least one edge \( e' \) on this path is not in \( X \), the set of edges chosen in the first \( i \) steps by Kruskal’s. (Otherwise, \( v \) and \( w \) will be connected in \( X \), so Kruskal’s won’t chose \( e \).)
  - Since neither \( e \) nor \( e' \) are in \( X \), and Kruskal’s chose \( e \), \( w(e') \geq w(e) \).
  - Replace \( e' \) by \( e \) in \( T \) to get another spanning tree \( T' \). Either \( w(T') < w(T) \), a contradiction to the assumption \( T \) is minimal; or \( w(T') = w(T) \), and we have another MST \( T' \) consistent with \( X \cup \{e\} \). In both cases, we have completed the induction step.
Kruskal’s: Runtime complexity

\[ \text{MST}(V, E, w) \]

\[
X = \emptyset \\
Q = \text{priorityQueue}(E, w) \text{ // from min to max weight} \\
\textbf{while} \ Q \text{ is nonempty} \\
\hspace{1em} e = \text{deleteMin}(Q) \\
\hspace{1em} \textbf{if} \ e \text{ connects two disconnected components in } (V, X) \hspace{1em} X = X \cup \{e\}
\]

- Priority queue: \( O(\log |E|) = O(\log V) \) per operation
- Connectivity test: \( O(\log V) \) per check using a disjoint set data structure

Thus, for \(|E|\) iterations, we have a runtime of \( O(|E| \log |V|) \)
MST: Applications

**Network design:** Communication networks, transportation networks, electrical grid, oil/water pipelines, ...

**Clustering:** Application of minimum spanning forest (stop when $|X| = |V| - k$ to get $k$ clusters)

**Broadcasting:** Spanning tree protocol in Ethernets
Information Theory and Coding

**Information content**

For an event $e$ that occurs with probability $p$, its information content is given by

$$I(e) = - \log p$$
For an event $e$ that occurs with probability $p$, its information content is given by

$$I(e) = - \log p$$

- “surprise factor” — low probability event conveys more information; an event that is almost always likely ($p \approx 1$) conveys no information.
- Information content adds up: for two events $e_1$ and $e_2$, their combined information content is $-(\log p_1 + \log p_2)$
Information theory: Entropy

Information entropy

For a discrete random variable $X$ that can take a value $x_i$ with probability $p_i$, its entropy is defined as the expectation (“weighted average”) over the information content of $x_i$:

$$H(X) = E[I(X)] = -\sum_{i=1}^{n} p_i \log p_i$$

Entropy is a measure of uncertainty and plays a fundamental role in many areas, including coding theory and machine learning.
Information theory: Entropy

Information entropy

For a discrete random variable $X$ that can take a value $x_i$ with probability $p_i$, its entropy is defined as the expectation (“weighted average”) over the information content of $x_i$:

$$H(X) = E[I(X)] = - \sum_{i=1}^{n} p_i \log p_i$$

- Entropy is a measure of uncertainty
- Plays a fundamental role in many areas, including coding theory and machine learning.
Shannon’s source coding theorem

A random variable $X$ denoting chars in an alphabet $\Sigma = \{x_1, \ldots, x_n\}$

- cannot be encoded in fewer than $H(X)$ bits.
- can be encoded using at most $H(X) + 1$ bits
Optimal code length

Shannon’s source coding theorem

A random variable $X$ denoting chars in an alphabet $\Sigma = \{x_1, \ldots, x_n\}$

- cannot be encoded in fewer than $H(X)$ bits.
- can be encoded using at most $H(X) + 1$ bits

The first part of this theorem sets a lower bound, regardless of how clever the encoding is.
Optimal code length

Shannon’s source coding theorem

A random variable $X$ denoting chars in an alphabet $\Sigma = \{x_1, \ldots, x_n\}$

- cannot be encoded in fewer than $H(X)$ bits.
- can be encoded using at most $H(X) + 1$ bits

The first part of this theorem sets a lower bound, regardless of how clever the encoding is.

Surprisingly simple proof for such a fundamental theorem! (See Wikipedia.)
A random variable $X$ denoting chars in an alphabet $\Sigma = \{x_1, \ldots, x_n\}$

- cannot be encoded in fewer than $H(X)$ bits.
- can be encoded using at most $H(X) + 1$ bits

The first part of this theorem sets a lower bound, regardless of how clever the encoding is.

Surprisingly simple proof for such a fundamental theorem! (See Wikipedia.)

Huffman coding: an algorithm that achieves this bound
Variable-length encoding

Let $\Sigma = \{A, B, C, D\}$ with probabilities 0.55, 0.02, 0.15, 0.28.

- If we use a fixed-length code, each character will use 2-bits.
- Alternatively, use a variable length code
Variable-length encoding

Let $\Sigma = \{A, B, C, D\}$ with probabilities 0.55, 0.02, 0.15, 0.28.

- If we use a fixed-length code, each character will use 2-bits.

- Alternatively, use a variable length code
  
  - Let us use as many bits as the *information content* of a character
  - $A$ uses 1 bit, $B$ uses 6 bits, $C$ uses 3 bits, and $D$ uses 2 bits.
  - You get an average saving of 15%

$$0.55 \times 1 + 0.02 \times 6 + 0.15 \times 3 + 0.28 \times 2 = 1.68 \text{ bits}$$
Variable-length encoding

Let $\Sigma = \{A, B, C, D\}$ with probabilities 0.55, 0.02, 0.15, 0.28.

- If we use a fixed-length code, each character will use 2-bits.
- Alternatively, use a variable length code
  - Let us use as many bits as the information content of a character
  - $A$ uses 1 bit, $B$ uses 6 bits, $C$ uses 3 bits, and $D$ uses 2 bits.
  - You get an average saving of 15%
  \[
  0.55 \times 1 + 0.02 \times 6 + 0.15 \times 3 + 0.28 \times 2 = 1.68 \text{ bits}
  \]
  - Lower bound (entropy)
  \[
  -(0.5 \log_2 0.5 + 0.02 \log_2 0.02 + 0.14 \log_2 0.14 + 0.27 \log_2 0.27) = 1.51 \text{ bits}
  \]
Variable-length encoding

Let \( \Sigma = \{A, B, C, D\} \) with probabilities 0.55, 0.02, 0.15, 0.28.

- Let us try fixing the codes, not just their lengths:
  
  \[
  A = 0, \quad D = 11, \quad C = 101, \quad B = 100.
  \]

- Note: enough to assign 3 bits to \( B \), not 6. So, average coding size reduces to 1.62.
Variable-length encoding

Let $\Sigma = \{A, B, C, D\}$ with probabilities 0.55, 0.02, 0.15, 0.28.

- Let us try fixing the codes, not just their lengths:
  
  $A = 0, D = 11, C = 101, B = 100$.

- Note: enough to assign 3 bits to $B$, not 6. So, average coding size reduces to 1.62.

Prefix encoding

- No code is a prefix of another.

- Necessary property to enable decoding.

- Every such encoding can be represented using a full binary tree (either 0 or 2 children for every node)
Huffman encoding

- Build the prefix tree bottom-up
- Start with a node whose children are codewords $c_1$ and $c_2$ that occur least often
- Remove $c_1$ and $c_2$ from alphabet, replace with $c'$ that occurs with frequency $f_1 + f_2$
- Recurse

Procedure Huffman

\begin{verbatim}
procedure Huffman(f)
    Input: An array $f[1 \cdot \cdot \cdot n]$ of frequencies
    Output: An encoding tree with $n$ leaves
    let $H$ be a priority queue of integers, ordered by $f$
    for $i = 1$ to $n$:
        insert($H$, $i$)
    for $k = n + 1$ to $2n - 1$:
        $i = \text{deletemin}(H)$,
        $j = \text{deletemin}(H)$
        create a node numbered $k$ with children $i$, $j$
        if $[k] = f[i] + f[j]$
            insert($H$, $k$)
\end{verbatim}

Returning to our toy example: can you tell if the tree of Figure 5.10 is optimal?

How to make this algorithm fast?

What is its complexity?
Huffman encoding

- Build the prefix tree bottom-up
- Start with a node whose children are codewords $c_1$ and $c_2$ that occur least often
- Remove $c_1$ and $c_2$ from alphabet, replace with $c'$ that occurs with frequency $f_1 + f_2$
- Recurse

How to make this algorithm fast?

What is its complexity?
After 19 merges, all 20 characters have been merged together. The record of merges gives us our code tree. The algorithm makes a number of arbitrary choices; as a result, there are actually several different Huffman codes. One such code is shown below. For example, the code for A is 110000, and the code for S is 00.

This sentence contains three a’s, three c’s, two d’s, twenty-six e’s, five f’s, three g’s, eight h’s, thirteen i’s, two l’s, sixteen n’s, nine o’s, six r’s, twenty-seven s’s, twenty-two t’s, two u’s, five v’s, eight w’s, four x’s, five y’s, and only one z. Images from Jeff Erickson’s “Algorithms”

Uses about 650 bits, vs 850 for fixed-length (5-bit) code.
Huffman encoding: Optimality

- Crux of the proof: *Greedy choice property*
- Familiar exchange argument
Huffman encoding: Optimality

- **Crux of the proof:** *Greedy choice property*
- **Familiar exchange argument**
  - Suppose the optimal prefix tree does not use longest path for two least frequent codewords $c_1$ and $c_2$
  - Show that by exchanging $c_1$ with the codeword using the longest path in the optimal tree, you can reduce the cost of the “optimal code” — a contradiction
Huffman encoding: Optimality

- **Crux of the proof:** *Greedy choice property*

- **Familiar exchange argument**
  - Suppose the optimal prefix tree does not use longest path for two least frequent codewords $c_1$ and $c_2$
  - Show that by exchanging $c_1$ with the codeword using the longest path in the optimal tree, you can reduce the cost of the “optimal code” — a contradiction
  - Same argument holds for $c_2$
Huffman Coding: Applications

- Document compression
- Signal encoding
- As part of other compression algorithms (MP3, gzip, PKZIP, JPEG, ...)

Overview  Kruskal  Huffman  Compression
Lossless Compression

- How much compression can we get using Huffman?
Lossless Compression

- How much compression can we get using Huffman?
  - It depends on what we mean by a codeword!
Lossless Compression

How much compression can we get using Huffman?

- It depends on what we mean by a codeword!
  - If they are English characters, effect is relatively small
  - if they are English words, or better, sentences, then much higher compression is possible
Lossless Compression

- How much compression can we get using Huffman?
  - It depends on what we mean by a codeword!
    - If they are English characters, effect is relatively small
    - If they are English words, or better, sentences, then much higher compression is possible

- To use words/sentences as codewords, we probably need to construct document-specific codebook
  - Larger alphabet size implies larger codebooks!
  - Need to consider the combined size of codebook plus the encoded document
How much compression can we get using Huffman?

- It depends on what we mean by a codeword!
  - If they are English characters, effect is relatively small
  - If they are English words, or better, sentences, then much higher compression is possible

To use words/sentences as codewords, we probably need to construct document-specific codebook

- Larger alphabet size implies larger codebooks!
- Need to consider the combined size of codebook plus the encoded document

Can the codebook be constructed on-the-fly?

- Lempel-Ziv compression algorithms (gzip)
gzip Algorithm [Lempel-Ziv 1977]

**Key Idea:** Use preceding $W$-bytes as the codebook ("sliding window", up to 32KB in gzip)
gzip Algorithm [Lempel-Ziv 1977]

Key Idea: Use preceding $W$-bytes as the codebook (“sliding window”, up to 32KB in gzip)

Encoding:
- Strings previously seen in the window are replaced by the pair $(\text{offset}, \text{length})$
  - Need to find the longest match for the current string
  - Matches should have a minimum length, or else they will be emitted as literals
  - Encode offset and length using Huffman encoding

Decoding: Interpret $(\text{offset}, \text{length})$ using the same window of $W$-bytes of preceding text. (Much faster than encoding.)
gzip Algorithm [Lempel-Ziv 1977]

Key Idea: Use preceding $W$-bytes as the codebook (“sliding window”, up to 32KB in gzip)

Encoding:
- Strings previously seen in the window are replaced by the pair ($offset$, $length$)
  - Need to find the longest match for the current string
  - Matches should have a minimum length, or else they will be emitted as literals
  - Encode offset and length using Huffman encoding

Decoding: Interpret ($offset$, $length$) using the same window of $W$-bytes of preceding text. (Much faster than encoding.)
Greedy Algorithms: Summary

- One of the strategies used to solve *optimization problems*
- Frequently, locally optimal choices are *NOT* globally optimal, so use with a great deal of care.
  - *Always need to prove optimality.* Proof typically relies on *greedy choice property*, usually established by an “exchange” argument, and *optimal substructure.*
Greedy Algorithms: Summary

- One of the strategies used to solve optimization problems
- Frequently, locally optimal choices are NOT globally optimal, so use with a great deal of care.
  - *Always need to prove optimality.* Proof typically relies on greedy choice property, usually established by an “exchange” argument, and optimal substructure.
Greedy Algorithms: Summary

- One of the strategies used to solve optimization problems

- Frequently, locally optimal choices are NOT globally optimal, so use with a great deal of care.
  - *Always need to prove optimality*. Proof typically relies on greedy choice property, usually established by an “exchange” argument, and optimal substructure.

- **Examples**
  - MST and clustering
  - Shortest path
  - Huffman encoding