Properties of Encoding Tree

- Rooted binary tree
- Left edges labelled 0 and right edges labelled 1
- A leaf corresponds to a code for some letter
- If coding scheme is not wasteful: a non-leaf has exactly two children

Best Prefix Codes

**Input:** frequencies of letters in a message

**Output:** prefix coding scheme with the shortest encoding for the message
example

<table>
<thead>
<tr>
<th>letters</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>frequencies</td>
<td>18</td>
<td>3</td>
<td>4</td>
<td>6</td>
<td>10</td>
</tr>
</tbody>
</table>

- scheme 1: total = 89
- scheme 2: total = 87
- scheme 3: total = 84

scheme 1

scheme 2

scheme 3
### Example

<table>
<thead>
<tr>
<th>letters</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>frequencies</td>
<td>18</td>
<td>3</td>
<td>4</td>
<td>6</td>
<td>10</td>
</tr>
</tbody>
</table>

| scheme 1 length | 2 | 3 | 3 | 2 | 2 | total = 89 |
| scheme 2 length | 1 | 3 | 3 | 3 | 3 | total = 87 |
| scheme 3 length | 1 | 4 | 4 | 3 | 2 | total = 84 |

#### Scheme 1
```
        a
       / \   
      b   d   e
     /     / \
    c     c   c
```

#### Scheme 2
```
        a
       /   \   
      b   c   d
     /     /   
    e     e   e
```

#### Scheme 3
```
        a
       /   
      c   e
     /     
    d     b
   /     / \
  c     b   c
```
Example Input: \((a: 18, b: 3, c: 4, d: 6, e: 10)\)
Example Input: \((a: 18, b: 3, c: 4, d: 6, e: 10)\)

Q: What types of decisions should we make?
Example Input: 
(a: 18, b: 3, c: 4, d: 6, e: 10)

Q: What types of decisions should we make?

Can we directly give a code for some letter?
Example Input: \((a: 18, b: 3, c: 4, d: 6, e: 10)\)

**Q:** What types of decisions should we make?

- Can we directly give a code for some letter?
- Hard to design a strategy; residual problem is complicated.
Example Input: \((a: 18, b: 3, c: 4, d: 6, e: 10)\)

**Q:** What types of decisions should we make?

- Can we directly give a code for some letter?
- Hard to design a strategy; residual problem is complicated.
- Can we partition the letters into left and right sub-trees?
Example Input: \((a: 18, b: 3, c: 4, d: 6, e: 10)\)

**Q:** What types of decisions should we make?

- Can we directly give a code for some letter?
- Hard to design a strategy; residual problem is complicated.
- Can we partition the letters into left and right sub-trees?
- Not clear how to design the greedy algorithm
Example Input: \((a: 18, b: 3, c: 4, d: 6, e: 10)\)

Q: What types of decisions should we make?

- Can we directly give a code for some letter?
- Hard to design a strategy; residual problem is complicated.
- Can we partition the letters into left and right sub-trees?
- Not clear how to design the greedy algorithm

A: We can choose two letters and make them brothers in the tree.
Which Two Letters Can Be Safely Put Together As Brothers?

- Focus on the “structure” of the optimum encoding tree
Which Two Letters Can Be Safely Put Together As Brothers?

- Focus on the “structure” of the optimum encoding tree
- There are two deepest leaves that are brothers
Which Two Letters Can Be Safely Put Together As Brothers?

- Focus on the “structure” of the optimum encoding tree
- There are two deepest leaves that are brothers

![Diagram](image)

best to put the two least frequent symbols here!
Which Two Letters Can Be Safely Put Together As Brothers?

- Focus on the “structure” of the optimum encoding tree
- There are two deepest leaves that are brothers

**Lemma** It is safe to make the two least frequent letters brothers.
Lemma  There is an optimum encoding tree, where the two least frequent letters are brothers.
**Lemma**  There is an optimum encoding tree, where the two least frequent letters are brothers.

- So we can irrevocably decide to make the two least frequent letters brothers.
**Lemma** There is an optimum encoding tree, where the two least frequent letters are brothers.

- So we can irrevocably decide to make the two least frequent letters brothers.

**Q:** Is the residual problem another instance of the best prefix codes problem?
**Lemma**  There is an optimum encoding tree, where the two least frequent letters are brothers.

- So we can irrevocably decide to make the two least frequent letters brothers.

**Q:** Is the residual problem another instance of the best prefix codes problem?

**A:** Yes, though it is not immediate to see why.
- $f_x$: the frequency of the letter $x$ in the support.
- $x_1$ and $x_2$: the two letters we decided to put together.
- $d_x$ the depth of letter $x$ in our output encoding tree.

$$
\sum_{x \in S} f_x d_x \\
= \sum_{x \in S \setminus \{x_1, x_2\}} f_x d_x + f_1 d_1 + f_2 d_2 \\
= \sum_{x \in S \setminus \{x_1, x_2\}} f_x d_x + (f_1 + f_2) d_1
$$
- $f_x$: the frequency of the letter $x$ in the support.
- $x_1$ and $x_2$: the two letters we decided to put together.
- $d_x$ the depth of letter $x$ in our output encoding tree.

\[
\sum_{x \in S} f_x d_x
\]
\[
= \sum_{x \in S \setminus \{x_1, x_2\}} f_x d_x + f_{x_1} d_{x_1} + f_{x_2} d_{x_2}
\]
\[
= \sum_{x \in S \setminus \{x_1, x_2\}} f_x d_x + (f_{x_1} + f_{x_2}) d_{x_1}
\]
- $f_x$: the frequency of the letter $x$ in the support.
- $x_1$ and $x_2$: the two letters we decided to put together.
- $d_x$ the depth of letter $x$ in our output encoding tree.

\[
\sum_{x \in S} f_x d_x \]

\[
= \sum_{x \in S \setminus \{x_1, x_2\}} f_x d_x + f_{x_1} d_{x_1} + f_{x_2} d_{x_2}
\]

Def: $f_{x'} = f_{x_1} + f_{x_2}$
- $f_x$: the frequency of the letter $x$ in the support.
- $x_1$ and $x_2$: the two letters we decided to put together.
- $d_x$ the depth of letter $x$ in our output encoding tree.

Def: $f_{x'} = f_{x_1} + f_{x_2}$

\[
\sum_{x \in S} f_x d_x
= \sum_{x \in S \setminus \{x_1, x_2\}} f_x d_x + f_{x_1} d_{x_1} + f_{x_2} d_{x_2}
= \sum_{x \in S \setminus \{x_1, x_2\}} f_x d_x + (f_{x_1} + f_{x_2}) d_{x_1}
= \sum_{x \in S \setminus \{x_1, x_2\}} f_x d_x + f_{x'} (d_{x'} + 1)
\]
- $f_x$: the frequency of the letter $x$ in the support.
- $x_1$ and $x_2$: the two letters we decided to put together.
- $d_x$ the depth of letter $x$ in our output encoding tree.

\begin{align*}
\text{Def: } f' &= f_x' = f_{x_1} + f_{x_2} \\
\sum_{x \in S} f_x d_x &= \sum_{x \in S \setminus \{x_1, x_2\}} f_x d_x + f_{x_1} d_{x_1} + f_{x_2} d_{x_2} \\
&= \sum_{x \in S \setminus \{x_1, x_2\}} f_x d_x + (f_{x_1} + f_{x_2}) d_{x_1} \\
&= \sum_{x \in S \setminus \{x_1, x_2\}} f_x d_x + f_{x'} (d_{x'} + 1) \\
&= \sum_{x \in S \setminus \{x_1, x_2\} \cup \{x'\}} f_x d_x + f_{x'}
\end{align*}
• \( f_x \): the frequency of the letter \( x \) in the support.
• \( x_1 \) and \( x_2 \): the two letters we decided to put together.
• \( d_x \) the depth of letter \( x \) in our output encoding tree.

\[
\begin{align*}
\sum_{x \in S} f_x d_x \\
= \sum_{x \in S \setminus \{x_1, x_2\}} f_x d_x + f_{x_1} d_{x_1} + f_{x_2} d_{x_2} \\
= \sum_{x \in S \setminus \{x_1, x_2\}} f_x d_x + (f_{x_1} + f_{x_2}) d_{x_1} \\
= \sum_{x \in S \setminus \{x_1, x_2\}} f_x d_x + f' (d_x' + 1) \\
= \sum_{x \in S \setminus \{x_1, x_2\} \cup \{x'\}} f_x d_x + f_{x'}
\end{align*}
\]

Def: \( f_{x'} = f_{x_1} + f_{x_2} \)
In order to minimize
\[
\sum_{x \in S} f_x d_x,
\]
we need to minimize
\[
\sum_{x \in S \setminus \{x_1, x_2\} \cup \{x'\}} f_x d_x,
\]
subject to that \( d \) is the depth function for an encoding tree of \( S \setminus \{x_1, x_2\} \).

- This is exactly the best prefix codes problem, with letters \( S \setminus \{x_1, x_2\} \cup \{x'\} \) and frequency vector \( f \).
Example
Example
Example

A 27
B 15
C 11
D 9
E 8
F 5
Example

```
  28
   /\  \\
 20 /  \20
 / \  /  \\
11 /  \ 11
  /    /    \\
 9     8     8
 /    /    /    \\
5     5     5     5
```

```
Example
Example

```
A 27
B 15
C 11
D 9
E 8
F 5
```

```
A
|-- B
|   |-- C
|   |   |-- D
|   |   |   |-- E
|   |   |           |-- F
```

Node values:
- A: 27
- B: 15
- C: 11
- D: 9
- E: 8
- F: 5
- Root: 75
- Node B: 47
- Node C: 28
- Node D: 20
Example
Example

A: 00
B: 10
C: 010
D: 011
E: 110
F: 111

A

B

C

D

E

F

27
15
11
9
8
5

20
9
8
13
11
10

Def. The codes given the greedy algorithm is called the **Huffman codes**.
**Def.** The codes given the greedy algorithm is called the **Huffman codes**.

**Huffman**($S, f$)

1: **while** $|S| > 1$ **do**
2: let $x_1, x_2$ be the two letters with the smallest $f$ values
3: introduce a new letter $x'$ and let $f_{x'} = f_{x_1} + f_{x_2}$
4: let $x_1$ and $x_2$ be the two children of $x'$
5: $S \leftarrow S \setminus \{x_1, x_2\} \cup \{x'\}$
6: **return** the tree constructed
# Huffman Algorithm

The Huffman algorithm is an efficient method for constructing a prefix code with the minimum expected code length for a given set of source symbols with associated probabilities. It is often used in compression algorithms, particularly in data compression like in Huffman coding.

## Algorithm Using Priority Queue

The algorithm uses a priority queue to efficiently select the two least frequent symbols at each step. The priority queue is updated with the sum of the frequencies of the two symbols. The process continues until only one symbol remains, forming the final tree.

### Algorithm: `Huffman(S, f)`

1. Let $Q \leftarrow \text{build-priority-queue}(S)$
2. While $Q$.size $> 1$
   3. Extract the min $x_1$ from $Q$.
   4. Extract the min $x_2$ from $Q$.
   5. Introduce a new letter $x'$ and let $f_{x'} = f_{x_1} + f_{x_2}$
   6. Let $x_1$ and $x_2$ be the two children of $x'$
   7. Insert $(x', f_{x'})$ into the priority queue $Q$
3. Return the tree constructed.
Outline

1. Toy Example: Box Packing
2. Interval Scheduling
   - Interval Partitioning
3. Offline Caching
   - Heap: Concrete Data Structure for Priority Queue
4. Data Compression and Huffman Code
5. Summary
6. Exercise Problems
Greedy Algorithm

- Build up the solutions in steps
- At each step, make an **irrevocable** decision using a “reasonable” strategy
### Summary for Greedy Algorithms

**Greedy Algorithm**

- Build up the solutions in steps
- At each step, make an **irrevocable** decision using a “reasonable” strategy

- Interval scheduling problem: schedule the job $j^*$ with the earliest deadline
Summary for Greedy Algorithms

**Greedy Algorithm**

- Build up the solutions in steps
- At each step, make an **irrevocable** decision using a “reasonable” strategy

- Interval scheduling problem: schedule the job \( j^* \) with the earliest deadline
- Offline Caching: evict the page that is used furthest in the future
# Summary for Greedy Algorithms

## Greedy Algorithm

- Build up the solutions in steps
- At each step, make an **irrevocable** decision using a “reasonable” strategy

<table>
<thead>
<tr>
<th>Interval scheduling problem: schedule the job $j^*$ with the earliest deadline</th>
</tr>
</thead>
<tbody>
<tr>
<td>Offline Caching: evict the page that is used furthest in the future</td>
</tr>
<tr>
<td>Huffman codes: make the two least frequent letters brothers</td>
</tr>
</tbody>
</table>
### Analysis of Greedy Algorithm

- **Safety:** Prove that the reasonable strategy is "safe" (key)
- **Self-reduce:** Show that the remaining task after applying the strategy is to solve a (many) smaller instance(s) of the same problem *(usually easy)*
Analysis of Greedy Algorithm

- **Safety**: Prove that the reasonable strategy is “safe” (key)
- **Self-reduce**: Show that the remaining task after applying the strategy is to solve a (many) smaller instance(s) of the same problem (usually easy)

**Def.** A strategy is “safe” if there is always an optimum solution that “agrees with” the decision made according to the strategy.
Proving a Strategy is Safe

- Take an arbitrary optimum solution $S$
Proving a Strategy is Safe

- Take an arbitrary optimum solution $S$
- If $S$ agrees with the decision made according to the strategy, done
Proving a Strategy is Safe

- Take an arbitrary optimum solution $S$
- If $S$ agrees with the decision made according to the strategy, done
- So assume $S$ does not agree with decision
Proving a Strategy is Safe

- Take an arbitrary optimum solution $S$
- If $S$ agrees with the decision made according to the strategy, done
- So assume $S$ does not agree with decision
- Change $S$ slightly to another optimum solution $S'$ that agrees with the decision
Proving a Strategy is Safe

- Take an arbitrary optimum solution $S$
- If $S$ agrees with the decision made according to the strategy, done
- So assume $S$ does not agree with decision
- Change $S$ slightly to another optimum solution $S'$ that agrees with the decision
- Interval scheduling problem: exchange $j^*$ with the first job in an optimal solution
Proving a Strategy is Safe

- Take an arbitrary optimum solution $S$
- If $S$ agrees with the decision made according to the strategy, done
- So assume $S$ does not agree with decision
- Change $S$ slightly to another optimum solution $S'$ that agrees with the decision
- Interval scheduling problem: exchange $j^*$ with the first job in an optimal solution
- Offline caching: a complicated “copying” algorithm
Proving a Strategy is Safe

- Take an arbitrary optimum solution $S$
- If $S$ agrees with the decision made according to the strategy, done
- So assume $S$ does not agree with decision
- Change $S$ slightly to another optimum solution $S'$ that agrees with the decision
  - Interval scheduling problem: exchange $j^*$ with the first job in an optimal solution
  - Offline caching: a complicated “copying” algorithm
  - Huffman codes: move the two least frequent letters to the deepest leaves.
## Analysis of Greedy Algorithm

- **Safety:** Prove that the reasonable strategy is “safe” (key)
- **Self-reduce:** Show that the remaining task after applying the strategy is to solve a (many) smaller instance(s) of the same problem (usually easy)
Summary for Greedy Algorithms

Analysis of Greedy Algorithm

- **Safety**: Prove that the reasonable strategy is "safe" (key)
- **Self-reduce**: Show that the remaining task after applying the strategy is to solve a (many) smaller instance(s) of the same problem (usually easy)

- Interval scheduling problem: remove $j^*$ and the jobs it conflicts with
## Analysis of Greedy Algorithm

- **Safety**: Prove that the reasonable strategy is “safe” (key)
- **Self-reduce**: Show that the remaining task after applying the strategy is to solve a (many) smaller instance(s) of the same problem (usually easy)

- Interval scheduling problem: remove $j^*$ and the jobs it conflicts with
- Offline caching: trivial
Summary for Greedy Algorithms

Analysis of Greedy Algorithm

- Safety: Prove that the reasonable strategy is “safe” (key)
- Self-reduce: Show that the remaining task after applying the strategy is to solve a (many) smaller instance(s) of the same problem (usually easy)

- Interval scheduling problem: remove $j^*$ and the jobs it conflicts with
- Offline caching: trivial
- Huffman codes: merge two letters into one
Outline

1. Toy Example: Box Packing
2. Interval Scheduling
   - Interval Partitioning
3. Offline Caching
   - Heap: Concrete Data Structure for Priority Queue
4. Data Compression and Huffman Code
5. Summary
6. Exercise Problems
Exercise: Fractional Knapsack Problem

**Fractional Knapsack**

**Input:** A knapsack of bounded capacity $W$; $n$ items, each of weight $\{w_1, w_2, \ldots, w_n\}$ and each item also has a value $\{v_1, v_2, \ldots, v_n\}$.

**Output:** Select a set of fractions $\{p_1, p_2, \ldots, p_n\}$ $(0 \leq p_i \leq 1)$ for all items to maximize the total value $p_1 v_1 + p_2 v_2 + \ldots + p_n v_n$ while $\sum_{i \in [n]} w_i p_i \leq W$. 

Example: Given are a knapsack with capacity $W = 20$ and 5 items with the following weights and values:

<table>
<thead>
<tr>
<th>Item</th>
<th>Weight</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>15</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>12</td>
<td>10</td>
</tr>
</tbody>
</table>
Fractional Knapsack

**Input:** A knapsack of bounded capacity $W$; $n$ items, each of weight $\{w_1, w_2, \ldots, w_n\}$ and each item also has a value $\{v_1, v_2, \ldots, v_n\}$.

**Output:** Select a set of fractions $\{p_1, p_2, \ldots, p_n\}$ ($0 \leq p_i \leq 1$) for all items to maximize the total value $p_1v_1 + p_2v_2 + \ldots + p_nv_n$ while $\sum_{i \in [n]} wi p_i \leq W$.

**Example:** Given are a knapsack with capacity $W = 20$ and 5 items with the following weights and values:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>weight</td>
<td>10</td>
<td>6</td>
<td>5</td>
<td>8</td>
<td>12</td>
</tr>
<tr>
<td>value</td>
<td>15</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>
Exercise: Scheduling Problem with Min Weighted Completion Time

Scheduling Problem

**Input:** Given are $n$ jobs each $i \in [n]$ has a weight (or the importance) $w_i$ and the length (or the time required) $l_j$. We define the completion time $c_j$ of job $j$ to be the sum of the lengths of jobs in the ordering up to and including $l_j$.

**Output:** An ordering of jobs that minimizes the weighted sum of completion times $\sum_{i \in [n]} w_i c_i$. 
Exercise: Scheduling Problem with Min Weighted Completion Time

Scheduling Problem

**Input:** Given are $n$ jobs each $i \in [n]$ has a weight (or the importance) $w_i$ and the length (or the time required) $l_j$. We define the completion time $c_j$ of job $j$ to be the sum of the lengths of jobs in the ordering up to and including $l_j$.

**Output:** An ordering of jobs that minimizes the weighted sum of completion times $\sum_{i \in [n]} w_i c_i$.

- Example: Given are 5 jobs with the following weights and lengths:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>weight</td>
<td>2</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>length</td>
<td>5</td>
<td>4</td>
<td>10</td>
<td>8</td>
<td>3</td>
</tr>
</tbody>
</table>