- Example Input: ($a$: 18, $b$: 3, $c$: 4, $d$: 6, $e$: 10)

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

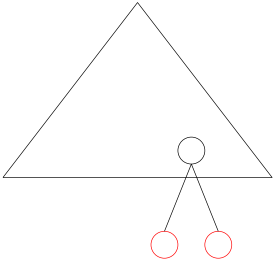
- Can we directly give a code for some letter?
- Hard to design a strategy; residual problem is complicated.

- 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.

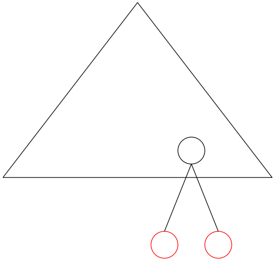
# Which Two Letters Can Be Safely Put Together As Brothers?

- Focus on the "structure" of the optimum encoding tree



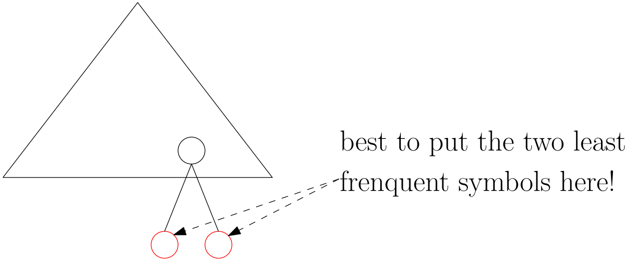
# Which Two Letters Can Be Safely Put Together As Brothers?

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



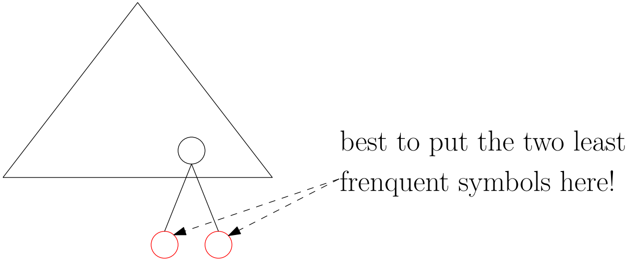
# Which Two Letters Can Be Safely Put Together As Brothers?

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



# 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?

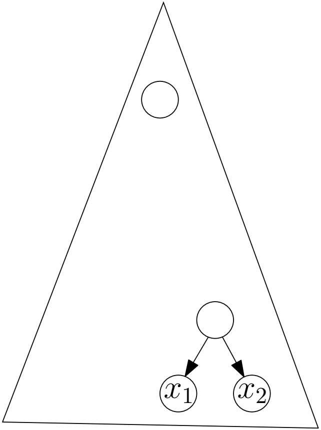
**Lemma** There is an optimum encoding tree, where the two least frequent letters are brothers.

- 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?

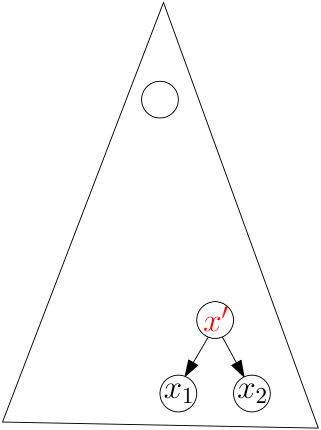
**A:** Yes, though it is not immediate to see why.

- $f_x$: 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.



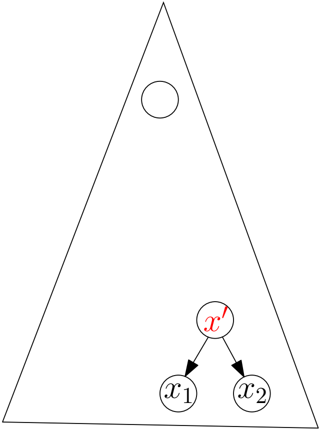

$$
\begin{aligned}
& \sum_{x \in S} f_x d_x \\
= & \sum_{x \in S \setminus \{x_1, x_2\}} f_x d_x + f_{x_1} d_{x_1} + f_{x_2} d_{x_2} \\
= & \sum_{x \in S \setminus \{x_1, x_2\}} f_x d_x + (f_{x_1} + f_{x_2}) d_{x_1}
\end{aligned}
$$

- $f_x$: 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.




$$\sum_{x \in S} f_x d_x$$
$$= \sum_{x \in S \setminus \{x_1, x_2\}} f_x d_x + f_{x_1} d_{x_1} + f_{x_2} d_{x_2}$$
$$= \sum_{x \in S \setminus \{x_1, x_2\}} f_x d_x + (f_{x_1} + f_{x_2}) d_{x_1}$$

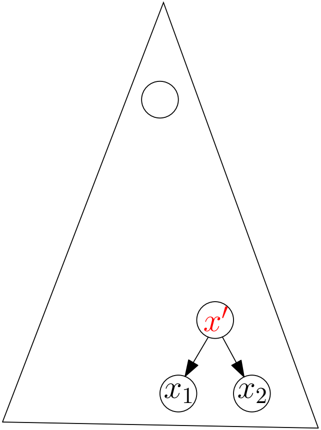
- $f_x$: the frequency of the letter $x$ in the support.
- $x_1$ and $x_2$: the two letters we decided to put together.
- $d_x$ the depth of letter $x$ in our output encoding tree.




$$\sum_{x \in S} f_x d_x$$
$$= \sum_{x \in S \setminus \{x_1, x_2\}} f_x d_x + f_{x_1} d_{x_1} + f_{x_2} d_{x_2}$$
$$= \sum_{x \in S \setminus \{x_1, x_2\}} f_x d_x + (f_{x_1} + f_{x_2}) d_{x_1}$$

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

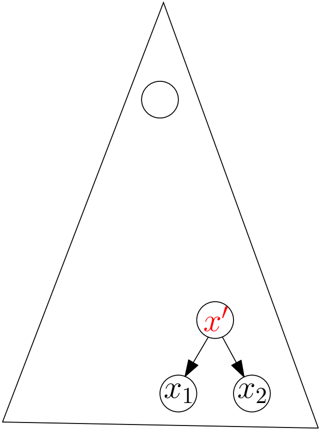
- $f_x$: the frequency of the letter $x$ in the support.
- $x_1$ and $x_2$: the two letters we decided to put together.
- $d_x$ the depth of letter $x$ in our output encoding tree.




$$
\begin{aligned}
&\sum_{x \in S} f_x d_x \\
&= \sum_{x \in S \setminus \{x_1, x_2\}} f_x d_x + f_{x_1} d_{x_1} + f_{x_2} d_{x_2} \\
&= \sum_{x \in S \setminus \{x_1, x_2\}} f_x d_x + (f_{x_1} + f_{x_2}) d_{x_1} \\
&= \sum_{x \in S \setminus \{x_1, x_2\}} f_x d_x + f_{x'} (d_{x'} + 1)
\end{aligned}
$$

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

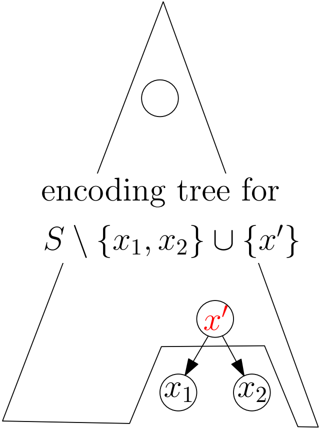
- $f_x$: the frequency of the letter $x$ in the support.
- $x_1$ and $x_2$: the two letters we decided to put together.
- $d_x$ the depth of letter $x$ in our output encoding tree.




$$\sum_{x \in S} f_x d_x$$
$$= \sum_{x \in S \setminus \{x_1, x_2\}} f_x d_x + f_{x_1} d_{x_1} + f_{x_2} d_{x_2}$$
$$= \sum_{x \in S \setminus \{x_1, x_2\}} f_x d_x + (f_{x_1} + f_{x_2}) d_{x_1}$$
$$= \sum_{x \in S \setminus \{x_1, x_2\}} f_x d_x + f_{x'} (d_{x'} + 1)$$
$$= \sum_{x \in S \setminus \{x_1, x_2\} \cup \{x'\}} f_x d_x + f_{x'}$$

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.




$$\begin{aligned}
&\sum_{x \in S} f_x d_x \\
&= \sum_{x \in S \setminus \{x_1, x_2\}} f_x d_x + f_{x_1} d_{x_1} + f_{x_2} d_{x_2} \\
&= \sum_{x \in S \setminus \{x_1, x_2\}} f_x d_x + (f_{x_1} + f_{x_2}) d_{x_1} \\
&= \sum_{x \in S \setminus \{x_1, x_2\}} f_x d_x + f_{x'} (d_{x'} + 1) \\
&= \sum_{x \in S \setminus \{x_1, x_2\} \cup \{x'\}} f_x d_x + f_{x'}
\end{aligned}$$

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

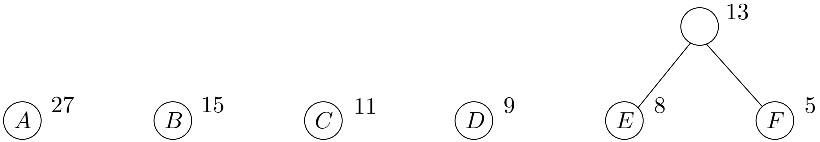
$$\sum_{x \in S \setminus \{x_1, x_2\} \cup \{x'\}} f_x d_x,$$

subject to that $d$ is the depth function for an encoding tree of $S \setminus \{x_1, x_2\}$.

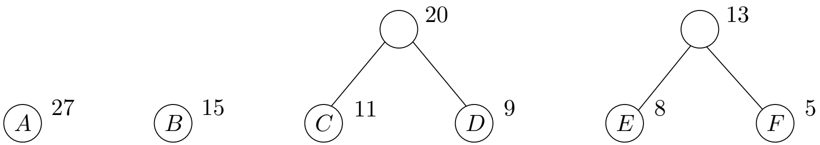
- This is exactly the best prefix codes problem, with letters $S \setminus \{x_1, x_2\} \cup \{x'\}$ and frequency vector $f$!

# Example

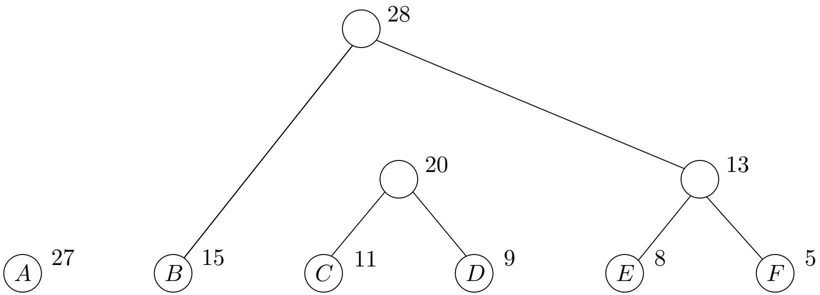
# Example



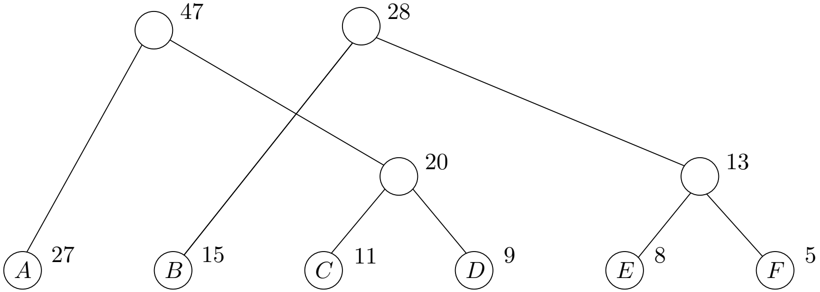
# Example



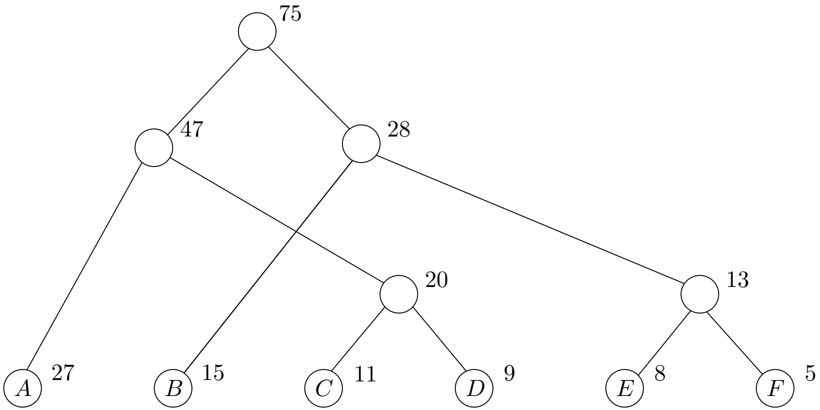
# Example



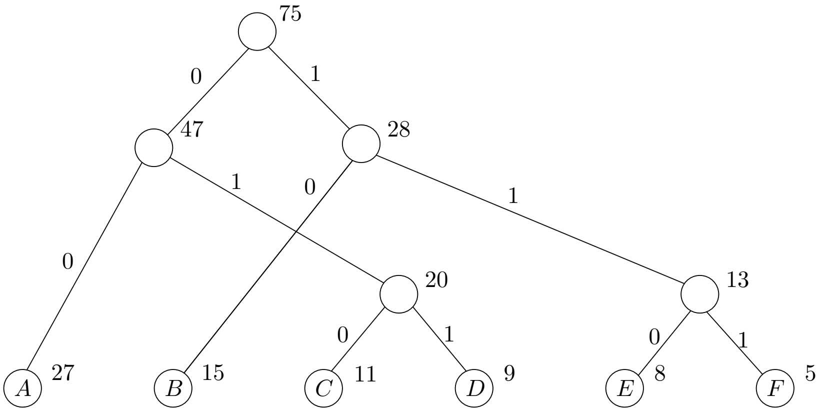
# Example



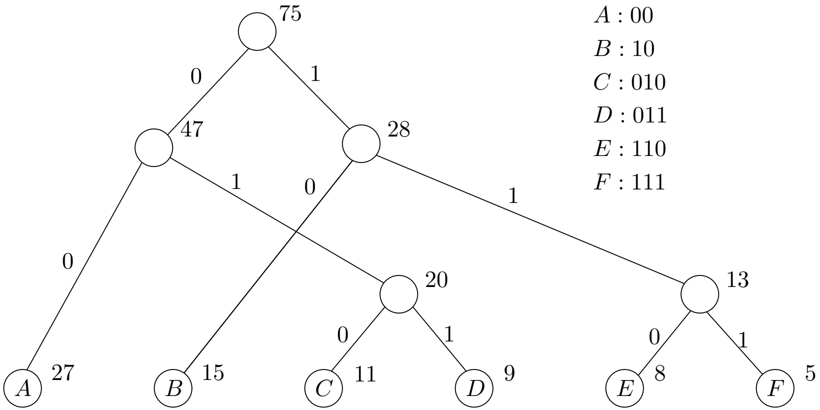
# Example



# Example



# Example




$A$ : 00

$B$ : 10

$C$ : 010

$D$ : 011

$E$ : 110

$F$ : 111

**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$(S, f)$

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

# Algorithm using Priority Queue

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

$Q \leftarrow$ build-priority-queue($S$)
**while** $Q.\text{size} > 1$ **do**
    $x_1 \leftarrow Q.\text{extract-min}()$
    $x_2 \leftarrow Q.\text{extract-min}()$
    introduce a new letter $x'$ and let $f_{x'} = f_{x_1} + f_{x_2}$
    let $x_1$ and $x_2$ be the two children of $x'$
    $Q.\text{insert}(x', f_{x'})$
**return** the tree constructed

# Outline

# Summary for Greedy Algorithms

## 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

- Interval scheduling problem: schedule the job $j^*$ with the earliest deadline
- Offline Caching: evict the page that is used furthest in the future
- Huffman codes: make the two least frequent letters brothers

# 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)

# 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)

**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.

# 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)

# 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

# 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

# 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