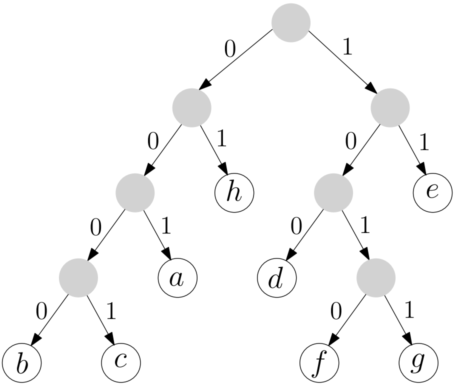


Properties of Encoding Tree

- 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

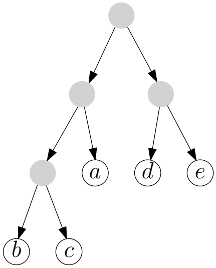
Best Prefix Codes

Input: frequencies of letters in a message

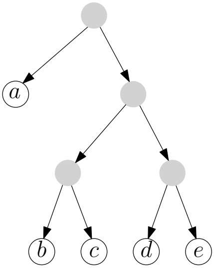
Output: prefix coding scheme with the shortest encoding for the message

example

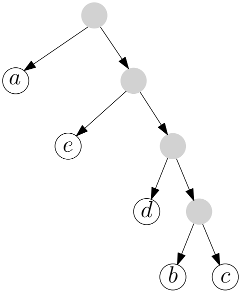
letters	a	b	c	d	e	
frequencies	18	3	4	6	10	



scheme 1



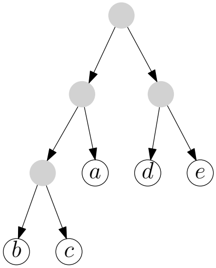
scheme 2



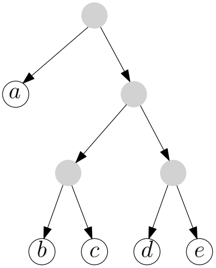
scheme 3

example

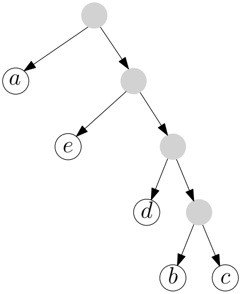
letters	a	b	c	d	e	
frequencies	18	3	4	6	10	
scheme 1 length	2	3	3	2	2	total = 89
scheme 2 length	1	3	3	3	3	total = 87
scheme 3 length	1	4	4	3	2	total = 84



scheme 1



scheme 2



scheme 3

- 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?

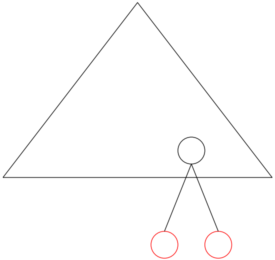
- Can we directly give a code for some letter?
- Hard to design a strategy; residual problem is complicated.

- 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.

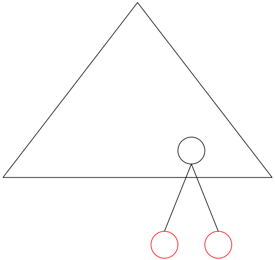
Which Two Letters Can Be Safely Put Together As Brothers?

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



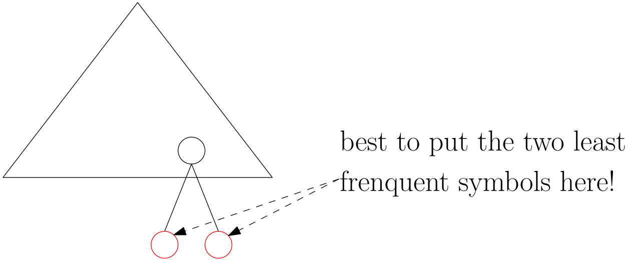
Which Two Letters Can Be Safely Put Together As Brothers?

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



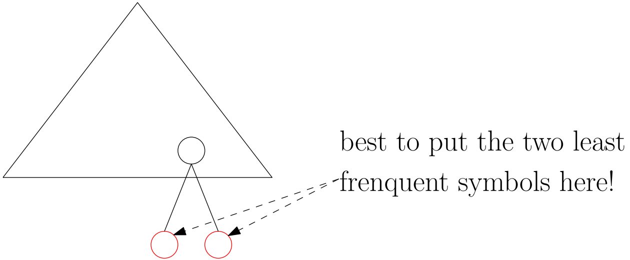
Which Two Letters Can Be Safely Put Together As Brothers?

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



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?

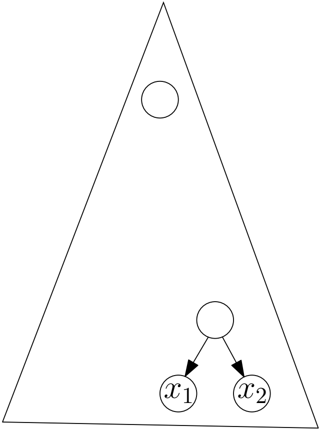
Lemma There is an optimum encoding tree, where the two least frequent letters are brothers.

- 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?

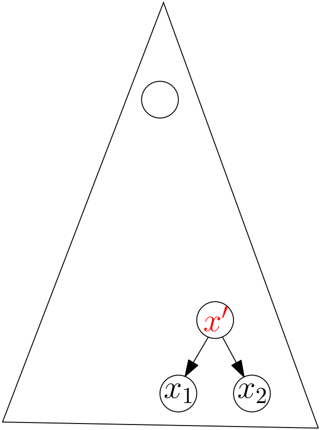
A: Yes, though it is not immediate to see why.

- 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.



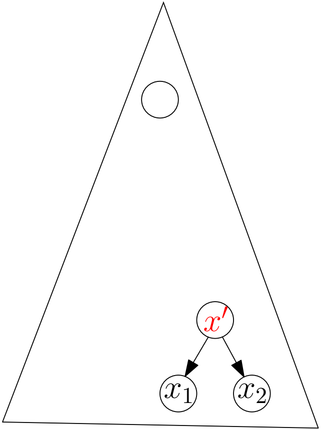
$$\sum_{x \in S} f_x d_x$$
$$= \sum_{x \in S \setminus \{x_1, x_2\}} f_x d_x + f_{x_1} d_{x_1} + f_{x_2} d_{x_2}$$
$$= \sum_{x \in S \setminus \{x_1, x_2\}} f_x d_x + (f_{x_1} + f_{x_2}) d_{x_1}$$

- 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.



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

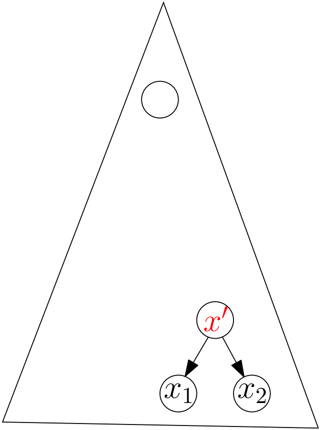
- f_x: the frequency of the letter x in the support.
- x_1 and x_2: the two letters we decided to put together.
- d_x the depth of letter x in our output encoding tree.



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

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.



$$\sum_{x \in S} f_x d_x$$

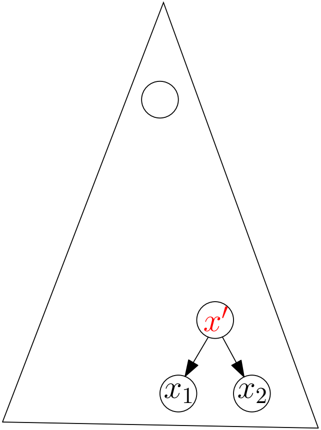
$$= \sum_{x \in S \setminus \{x_1, x_2\}} f_x d_x + f_{x_1} d_{x_1} + f_{x_2} d_{x_2}$$

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

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

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.



$$\sum_{x \in S} f_x d_x$$

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

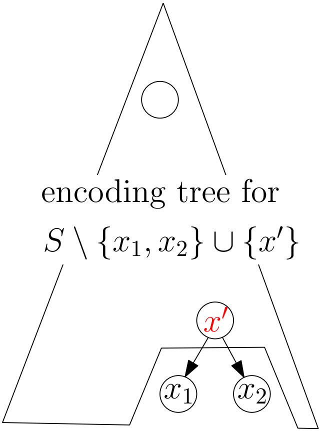
$$= \sum_{x \in S \setminus \{x_1, x_2\}} f_x d_x + (f_{x_1} + f_{x_2}) d_{x_1}$$

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

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

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

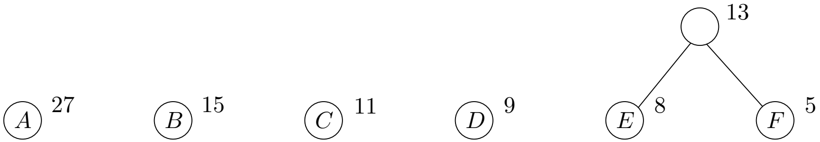
$$\sum_{x \in S \setminus \{x_1, x_2\} \cup \{x'\}} f_x d_x,$$

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

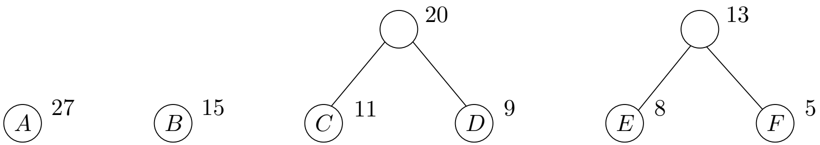
- This is exactly the best prefix codes problem, with letters $S \setminus \{x_1, x_2\} \cup \{x'\}$ and frequency vector f!

Example

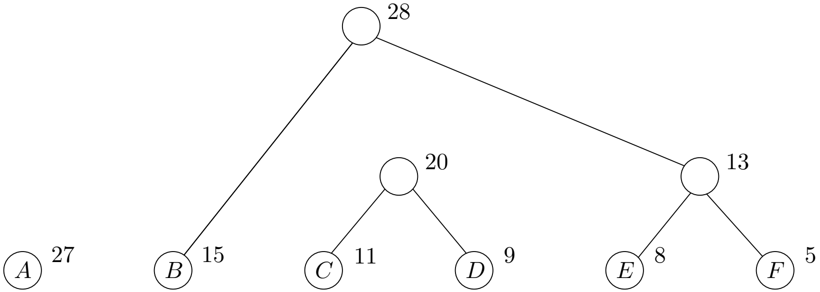
Example



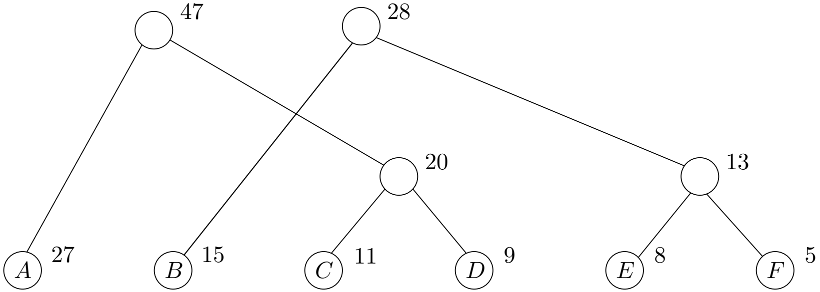
Example



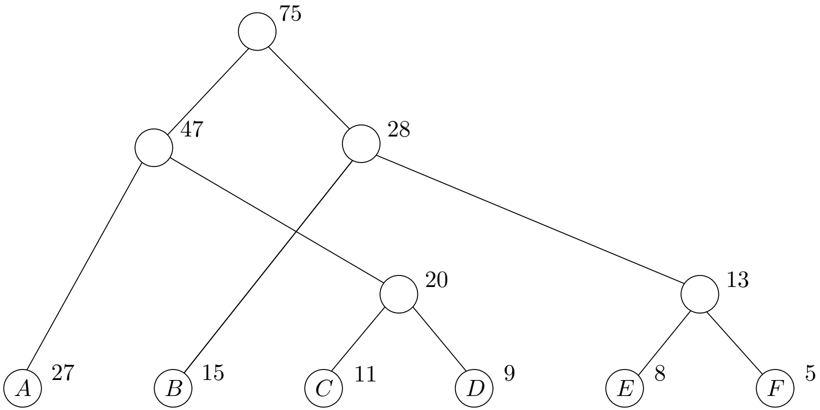
Example



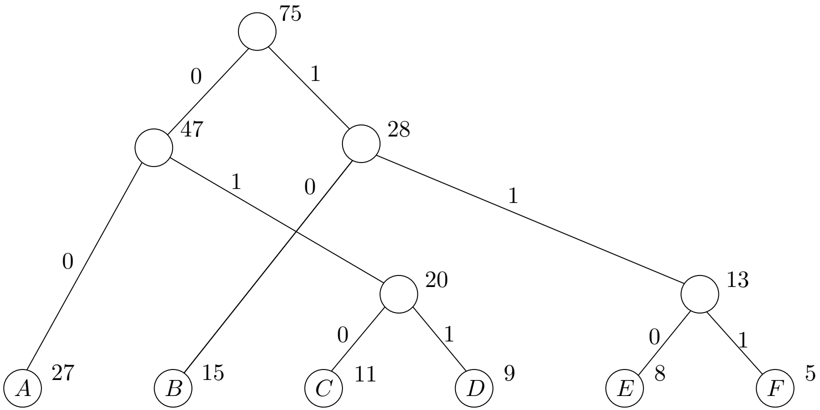
Example



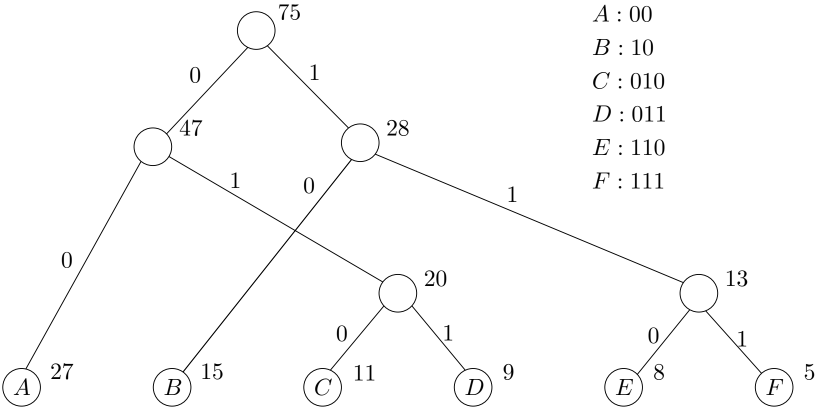
Example



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 ← S \ {x_1, x_2} ∪ {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

Outline

Exercise: Fractional Knapsack Problem

Fractional Knapsack

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

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

Exercise: Fractional Knapsack Problem

Fractional Knapsack

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

Output: Select a set of fractions $\{p_1, p_2, ..., p_n\}$ $(0 \leq p_i \leq 1)$ for all items to maximize the total value $p_1v_1 + p_2v_2 + ... + p_nv_n$ 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:

	1	2	3	4	5
weight	10	6	5	8	12
value	15	10	10	10	10

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:

	1	2	3	4	5
weight	2	6	5	4	2
length	5	4	10	8	3