Lecture 10: Huffman Coding
Another Greedy Algorithm

Version of February 28, 2019

Encoding

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

Encoding: Replace characters by corresponding codewords.

Example: Encode the word faded

Fixed-length Code: 101000011100011

Variable-length Code: 11000111101111

Encoding

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

Encoding: Replace characters by corresponding codewords.

Q: How can one design a code minimizing length of encoded message?

Ex: For a file with 100,000 characters that appear with the frequencies given in the table,

The fixed-length code requires

$$3 \cdot 100{,}000 = 300{,}000 \text{ bits}$$

The variable-length code requires

$$(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1000 = 224{,}000 \text{ bits}$$

Decoding

Decoding: Replace codewords by corresponding characters.

$$C_1 = \{\texttt{a = 00, b = 01, c = 10, d = 11}\}.$$
$$C_2 = \{\texttt{a = 0, b = 110, c = 10, d = 111}\}.$$
$$C_3 = \{\texttt{a = 1, b = 110, c = 10, d = 111}\}$$

A message is uniquely decodable if it can only be decoded in one way.

Ex:

- Relative to C_1, **`010011`** is uniquely decodable to **`bad`**.
- Relative to C_2, **`1100111`** is uniquely decodable to **`bad`**.
- But, relative to C_3, **`1101111`** is not uniquely decipherable since it could have encoded to either **`bad`** or **`acad`**.

In fact, one can show that every message encoded using C_1 or C_2 is uniquely decodable.

- C_1: Because it is a fixed-length code.
- C_2: Because it is a prefix-free code.

Prefix Codes

Def: A code is called a prefix (free) code if no codeword is a prefix of another one.

Theorem: Every message encoded by a prefix free code is uniquely decodable.

Pf: Since no codeword is a prefix of any other, we can always find the first codeword in a message, peel it off, and continue decoding.

Ex: code: {`a = 0, b = 110, c = 10, d = 111`}.

`01101100 = 0 110 110 0 = abba`

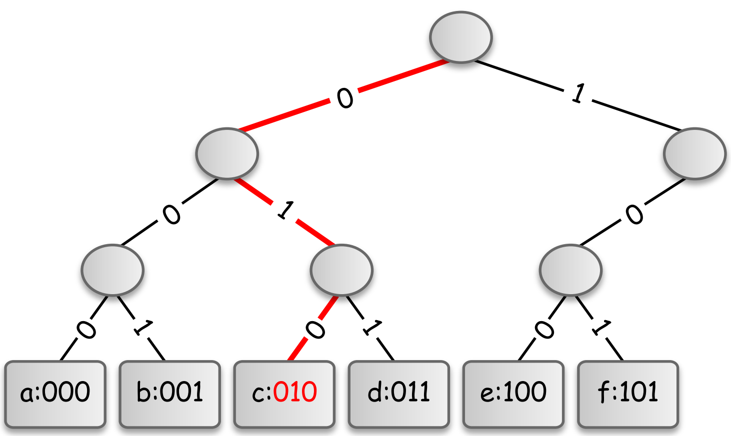
Note: There are other kinds of codes that are also uniquely decodable.

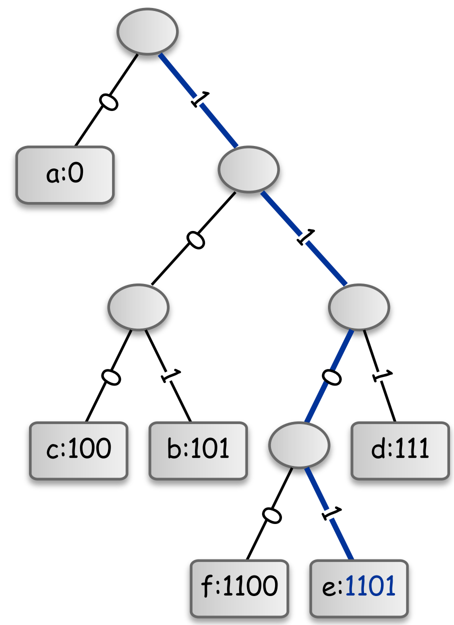
Theorem (proof omitted): The best prefix code can achieve the optimal data compression among any code that is uniquely decodable.

Problem: For a given input file, find the (a) prefix code that results in the smallest encoded message. (Compression)

Correspondence between Binary Trees and Prefix Codes

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100





Left edge labeled 0; right edge is labeled 1.

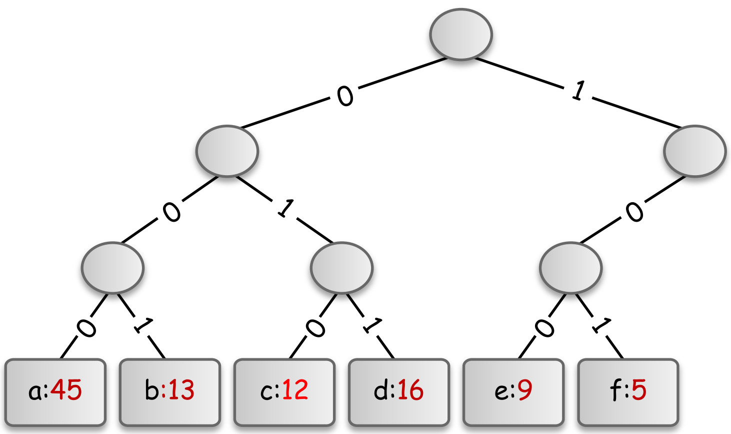
Binary string read off on path from root to a leaf
is the codeword associated with the character at that leaf.

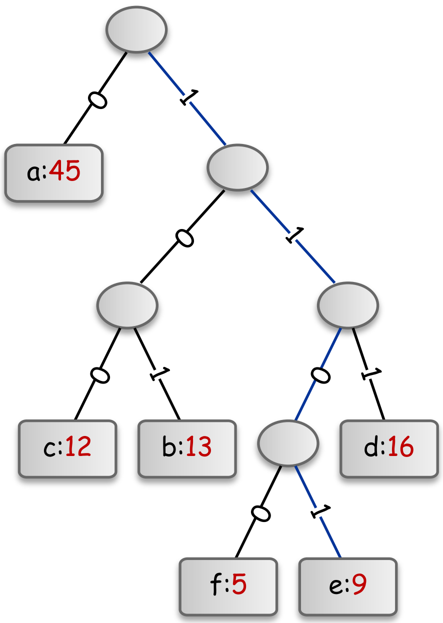
Depth of a leaf is equal to the length of the codeword.

Weighted External Path Length

Given a tree with n leaves labeled $a_1, \ldots, a_n$ and associated leaf weights $f(a_1), \ldots, f(a_n)$, the *weighted external path length* of the tree is

$$B(T) = \sum_{i=1}^{n} f(a_i) d(a_i)$$



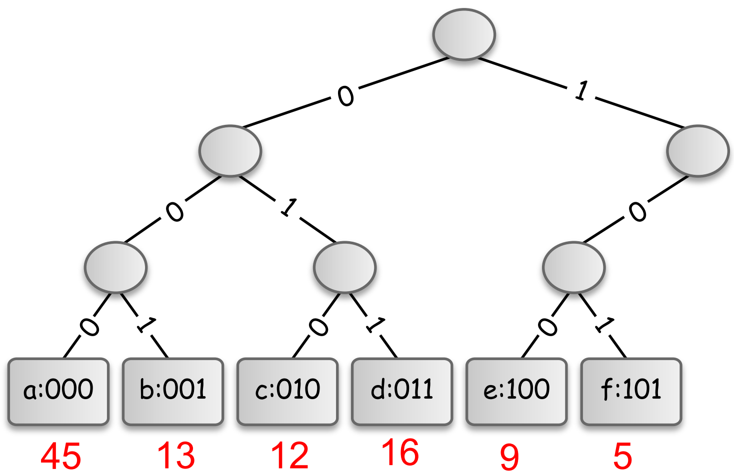


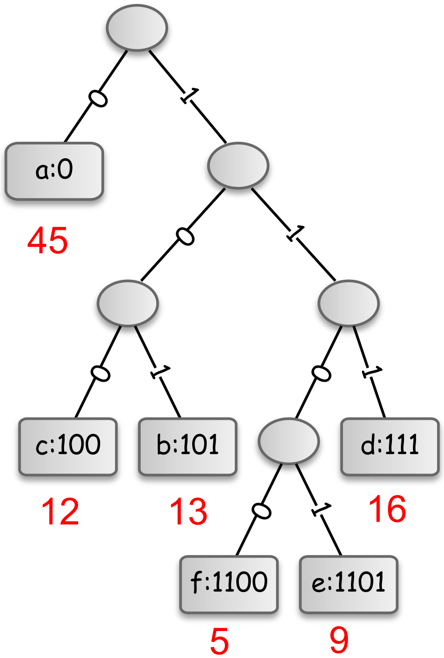
$(45 + 13 + 12 + 16 + 9 + 5) * 3 = 300$

$45 * 1 + (12 + 13 + 16) * 3 + (9 + 5) * 4 = 224$

Correspondence between Binary Trees and Prefix Codes

	a	b	c	d	e	f	Total Cost
Frequency (in thousands)	45	13	12	16	9	5	
Fixed-length codeword	000	001	010	011	100	101	300
Variable-length codeword	0	101	100	111	1101	1100	224





Set weight of leaf to be frequency of associated code word

External Weighted Path Length (cost) of tree is EXACTLY total cost of code

=> Finding min cost code is same as finding min-cost tree!

Problem Restated

Problem definition: Given an alphabet A of n characters $a_1, \dots, a_n$ with weights $f(a_1), \dots, f(a_n)$, find a binary tree T with n leaves labeled $a_1, \dots, a_n$ such that

$$B(T) = \sum_{i=1}^{n} f(a_i) d(a_i)$$

is minimized, where $d(a_i)$ is the depth of a_i.

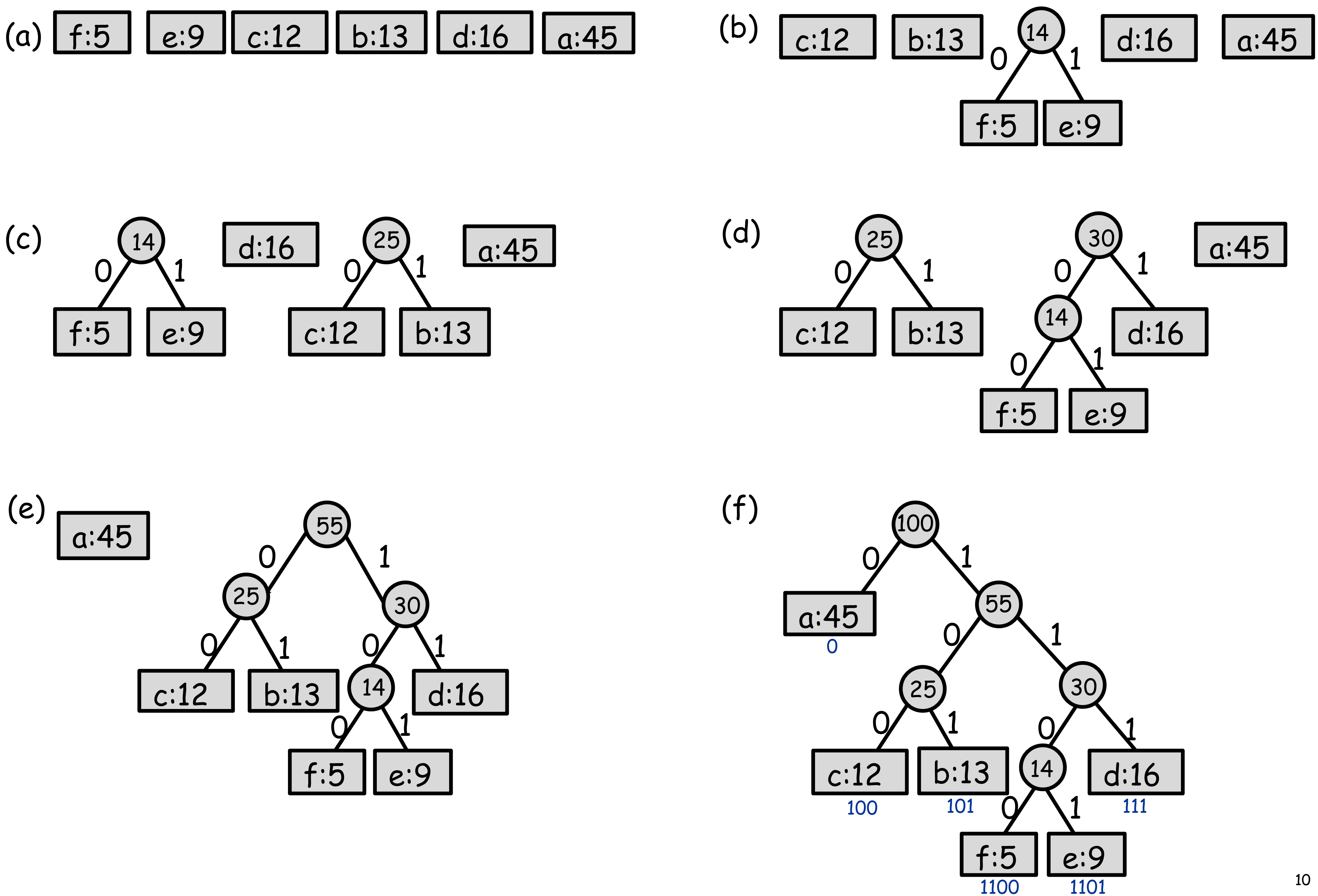
Greedy idea:

- Pick two characters x, y from A with the smallest weights
- Create a subtree that has these two characters as leaves.
- Label the root of this subtree as z.
- Set frequency $f(z) \leftarrow f(x) + f(y)$.

- Remove x, y from A and add z to A.

- Repeat the above procedure (called a "merge"), until only one character is left.

Example



(a)
f:5
e:9
c:12
b:13
d:16
a:45
(b)
c:12
b:13
14
0
1
f:5
e:9
d:16
a:45
(c)
14
0
1
f:5
e:9
d:16
25
0
1
c:12
b:13
a:45
(d)
25
0
1
c:12
b:13
30
0
1
14
0
1
f:5
e:9
d:16
a:45
(e)
a:45
55
0
1
25
0
1
c:12
b:13
30
0
1
14
0
1
f:5
e:9
d:16
(f)
100
0
1
a:45
0
55
0
1
25
0
1
c:12
100
b:13
101
30
0
1
14
0
1
f:5
1100
e:9
1101
d:16
111

The Algorithm

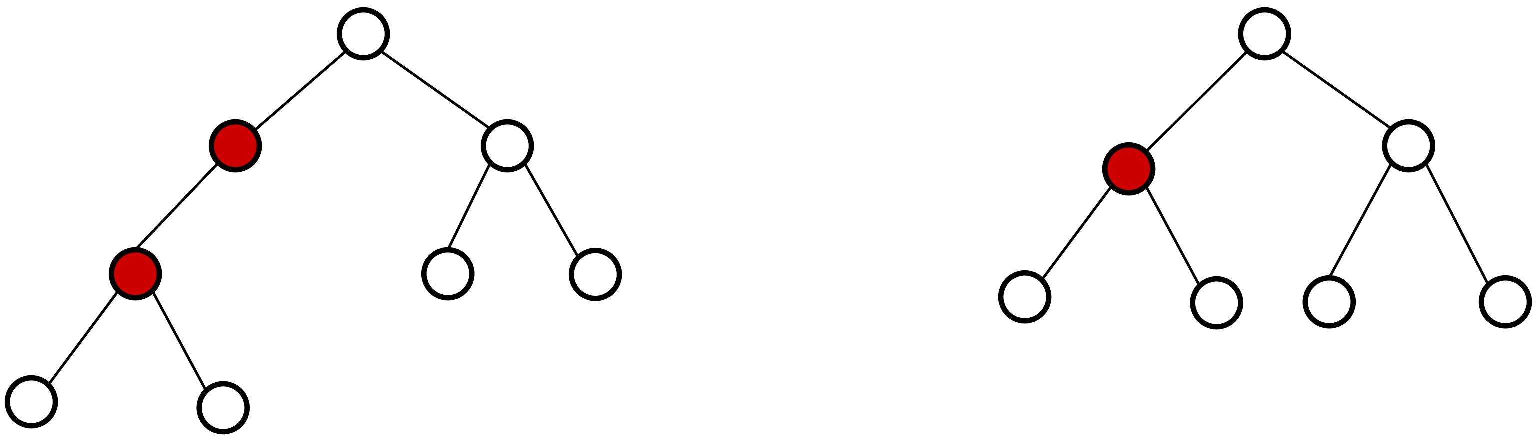
```
Huffman(A):
create a min-priority queue Q on A, with weight as key
for i ← 1 to n − 1
    allocate a new node z
    x ← Extract-Min(Q)
    y ← Extract-Min(Q)
    z.left ← x
    z.right ← y
    z.weight ← x.weight + y.weight
    Insert(Q,z)
return Extract-Min(Q) // return the root of the tree
```

Running time: $O(n \log n)$

Huffman Coding: Correctness

Lemma 1: An optimal prefix code tree must be "full", i.e., every internal node has exactly two children.

Pf: If some internal node had only one child,



then we could simply get rid of this node and replace it with its child. This would decrease the total cost of the encoding

(because no leaf increases depth and some leaves(s) decrease depth)

Huffman Coding: Correctness

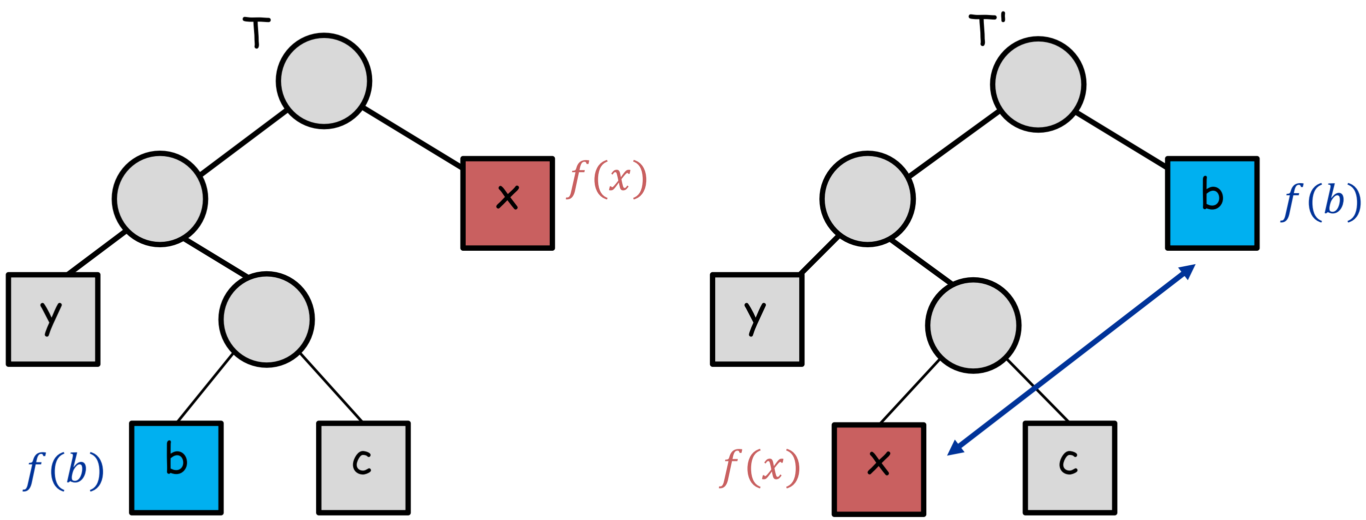
Observation: Moving a small-frequency character downward in T doesn't increase tree cost.

Lemma 2: Let T be prefix code tree and T' be another obtained from T by swapping two leaf nodes x and b. If,

$$f(x) \le f(b), \qquad d(x) \le d(b)$$

then,

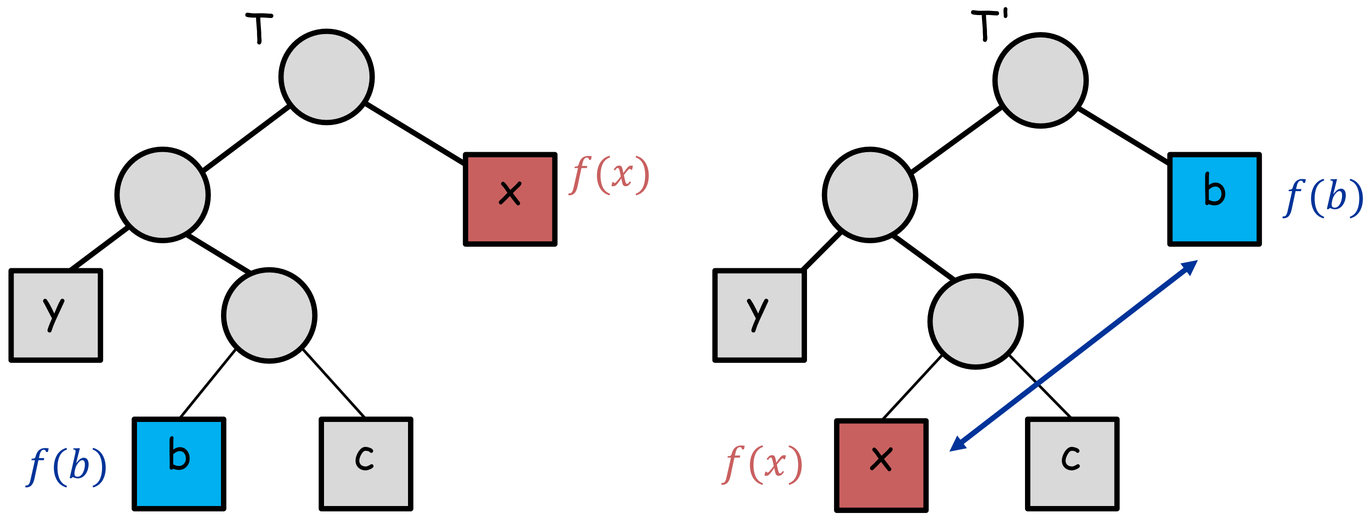
$$B(T') \le B(T).$$



Huffman Coding: Correctness

Pf:

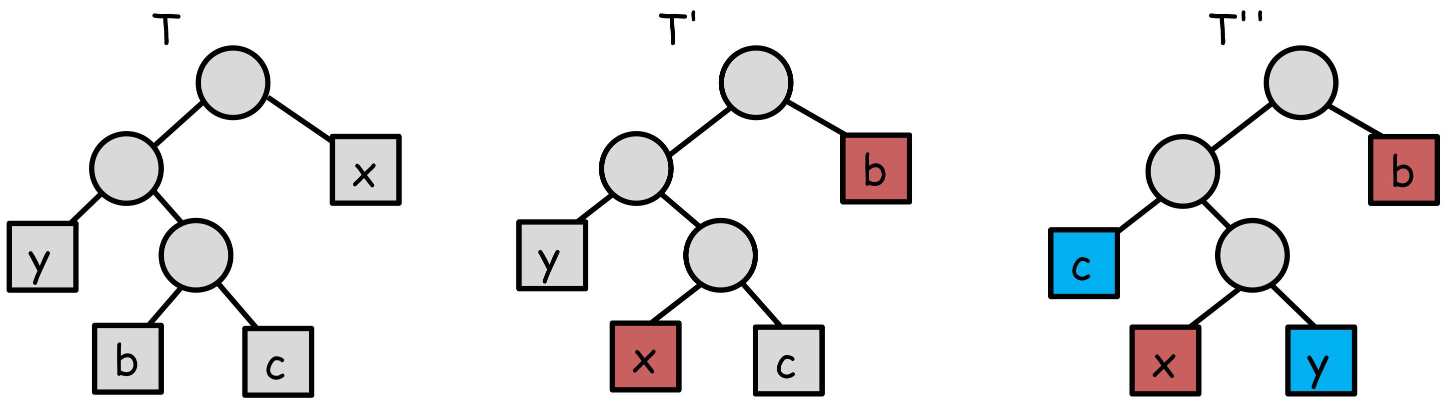
$$
\begin{aligned}
B(T') &= B(T) - f(x)d(x) - f(b)d(b) + f(x)d(b) + f(b)d(x) \\
&= B(T) + \underbrace{(f(x) - f(b))}_{\leq 0}\underbrace{(d(b) - d(x))}_{\geq 0} \\
&\leq B(T).
\end{aligned}
$$



Huffman Coding: Correctness

Lemma 3: Consider the two characters x and y with the smallest frequencies. There is an optimal code tree in which these two letters are sibling leaves at the deepest level of the tree.

Pf: Let T be any optimal prefix code tree, b and c be two siblings at the deepest level of the tree (must exist because T is full).



Assume without loss of generality that $f(x) \le f(b)$ and $f(y) \le f(c)$

- (If necessary) swap x with b and swap y with c.
- Proof follows from Lemma 2.

  (Lemma 2 => cost can't increase => since old tree optimal, new one is also)

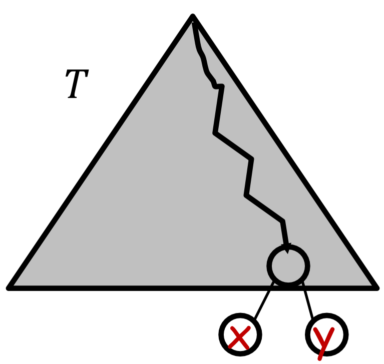
Lemma 4: Let T be a prefix code tree in which x and y are two *sibling* leaves. Let T' be obtained from T by removing x and y, naming their parent z, and setting $f(z) = f(x) + f(y)$. Then

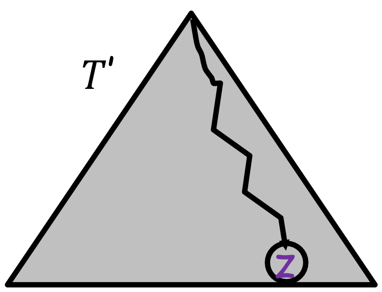
$$B(T) = B(T') + f(x) + f(y).$$

Pf: $B(T) = B(T') - f(z)d(z) + f(x)(d(z) + 1) + f(y)(d(z) + 1)$

$$= B(T') - f(z)d(z) + \big(f(x) + f(y)\big)d(z) + \big(f(x) + f(y)\big)$$

$$= B(T') + f(x) + \mathrm{f}(\mathrm{y}).$$



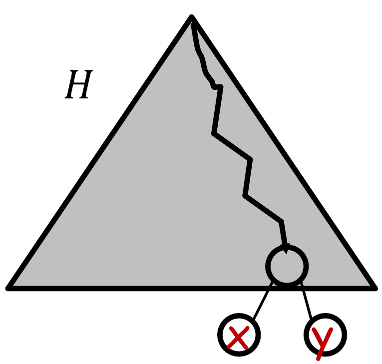


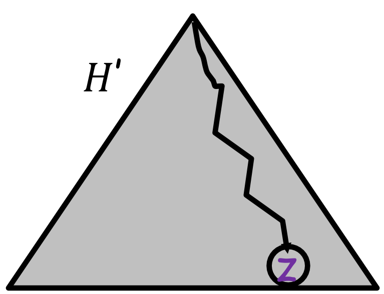
Huffman Coding: Correctness

Observation:

- Let H be the tree produced by Huffman's algorithm for alphabet A.
- Let x and y be the first two items merged together by the algorithm. Note that these are siblings in H.
- Let z be a new character with $f(z) = f(x) + f(y)$. Set $A' = A \cup \{z\} - \{x, y\}$
- Let H' be the tree obtained from H by removing x and y, naming their parent z, and setting $f(z) = f(x) + f(y)$.

Then H' is exactly the tree constructed by the Huffman algorithm on A'.





Huffman Coding: Correctness

Theorem: The Huffman tree is optimal.

Pf: (By induction on n, the number of characters)

- Base case $n = 2$: Tree with two leaves. Obviously optimal.

Huffman Coding: Correctness

Theorem: The Huffman tree is optimal.

Pf: (By induction on n, the number of characters)

- Induction hypothesis: Huffman's algorithm produces optimal tree for all inputs case of $n-1$ characters.

- Induction step: Consider input of n characters:
  - Let H be the tree produced by Huffman's algorithm.
  - Need to show: H is optimal.

- From operation of Huffman's algorithm:
  - There exist two characters x and y with two smallest frequencies that are sibling leaves in H.
- Let H' be obtained from H by (i) removing x and y, (ii) naming their parent z, and (iii) setting $f(z) = f(x) + f(y)$

- Alphabet for H: A; Alphabet for H' : $A' = A - \{x, y\} \cup \{z\}$
- By Lemma 4, $B(H) = B(H') + f(x) + f(y)$.

Huffman Coding: Correctness

- H is the tree produced by Huffman's algorithm for A (with x,y)
- H' is the tree produced by Huffman's algorithm for A' (with z, without x,y)
- **By the induction hypothesis, H' is optimal for A'.**

- By Lemma 3, there exists some optimal tree T for which x and y are sibling leaves.

- Let T' be obtained from T by (i) removing x, y, (ii) naming the parent z, and (iii) setting $f(z) = f(x) + f(y)$.
- T' is a prefix code tree for alphabet A'.

- By Lemma 4, $B(T) = B(T') + f(x) + f(y)$.
- Hence

$$\begin{aligned} B(H) &= B(H') + f(x) + f(y) \\ &\leq B(T') + f(x) + f(y) \qquad (H' \text{ is optimal for } A') \\ &= B(T). \end{aligned}$$

- **Therefore, H must be optimal!**

Diagram of the proof

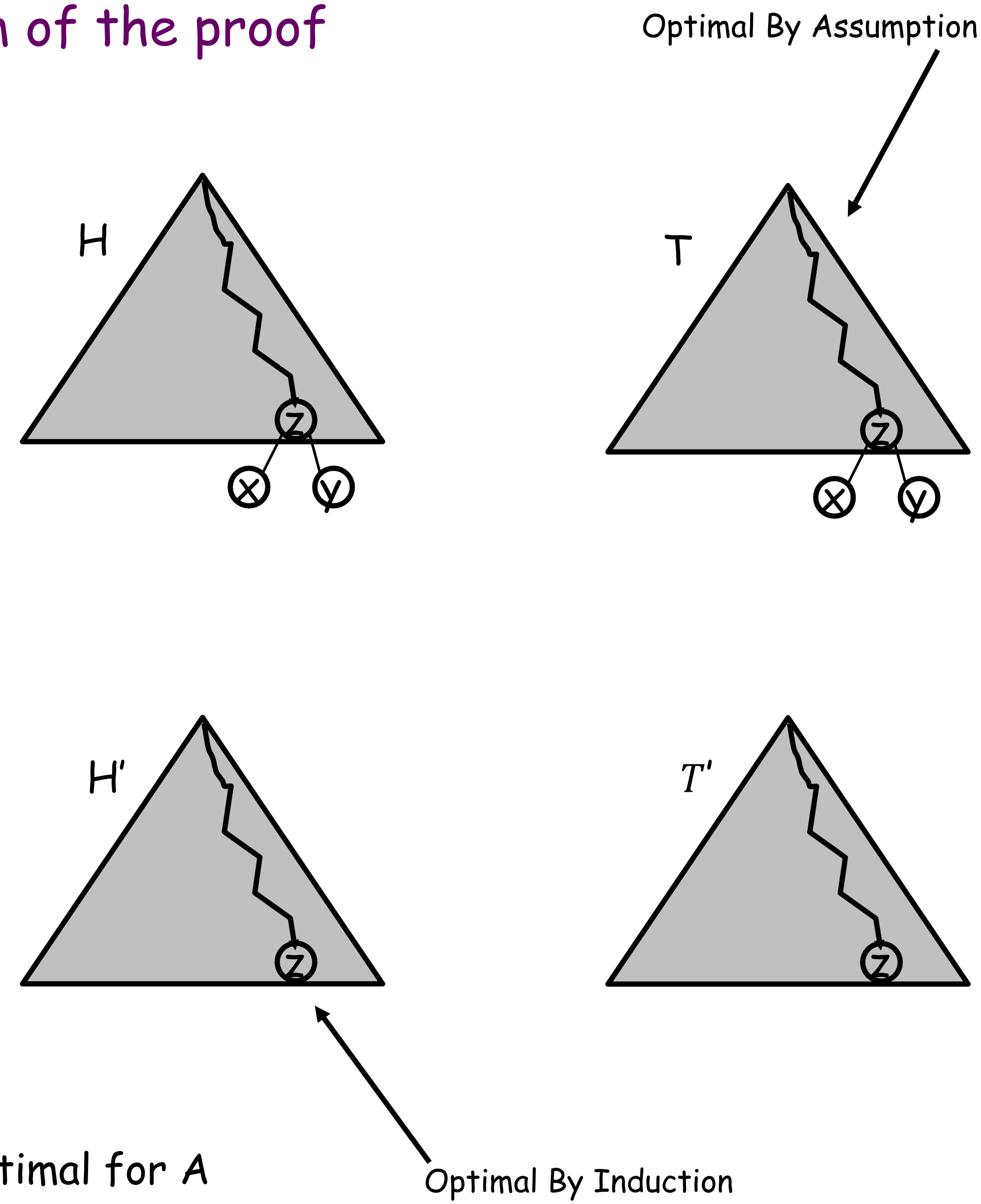
A is original character set
$A' = A \cup \{z\} - \{x, y\}$

H built by Huffman Alg on A
=> H' built by Huff Alg on A'
$\Rightarrow$ By induction H' optimal for A'

$$B(H) = B(H') + f(x) + f(y)$$

T chosen as Optimal tree for A
T' built from T as tree on A'

$$B(T) = B(T') + f(x) + f(y)$$



$$\begin{aligned} B(H) &= B(H') + f(x) + f(y) \\ &\le B(T') + f(x) + f(y) \\ &= B(T). \end{aligned}$$

=> H is optimal for A

The End of Greedy

This is the end of the section on Greedy Algorithms.

We learned 4 greedy algorithms (with more in the tutorials).

It's worthwhile looking back and reviewing the takeaways.

1. Greedy algorithms are often the simplest possible algorithms to design
   - They build solutions, one step at a time.
   - Each step makes a greedy, irrevocable, decision (solution can't be modified later)

2. Proving that the greedy solution is optimal is usually the hard part
   - Greedy is not always optimal (often isn't)
   - There is no one way to prove optimality
     - We saw three standard approaches
     - Note that not every method can work for every problem.

Three Different Proof Techniques

1. Modifying an Optimal Solution into Greedy
   - Did this for **Interval Scheduling** & **Fractional Knapsack**
   - Start (conceptually) with different **G**(reedy) and **O**(optimal) solutions
     - Define "distance" between **G** and **O**
     - Show how **O** can be modified to **O'** that is still optimal but closer to **G**
     - Repeat, until have created optimal solution that is exactly **G**

2. Lower Bound Technique
   - Did this for **Interval Partitioning**
   - For problems trying to find **minimum** solution (can be modified for max)
     - Define value **L** that is a lower bound on ANY feasible solution
     - Show that Greedy solution has value **L**

3. Inductive Proof
   - Did this for **Huffman Coding**
     - Prove inductively that if Greedy is correct for all problems of size n
       => it is correct for all problems of size n+1.