CS 3100
Data Structures and Algorithms 2
Lecture 14: Huffman Encoding

Co-instructors: Robbie Hott and Ray Pettit

Spring 2024

Readings in CLRS 4th edition:
- Chapter 16

Warm Up

Decode the line below into English

(hint: use Google or Wolfram Alpha)

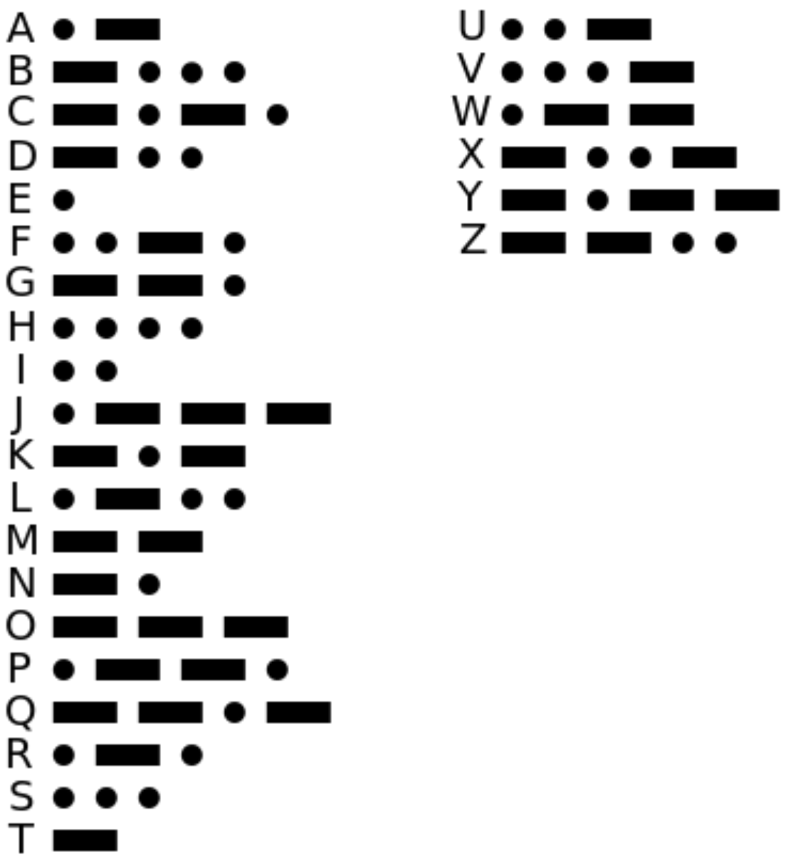
.. .-.. .. -.- . .- .-.. --. --- .-. .. - .... -- ...

Warm Up

Decode the line below into English

(hint: use Google or Wolfram Alpha)

..   .-.. .. -.- .   .- .-.. --. --- .-. .. - .... -- ...

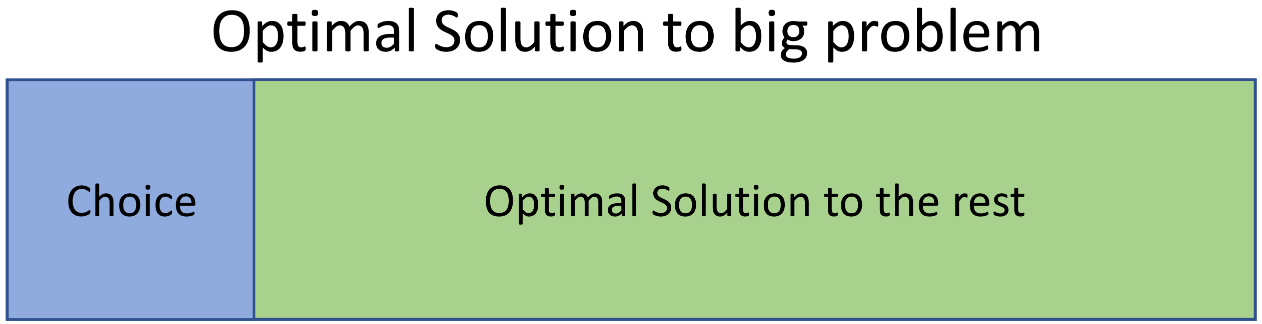


Announcements

- PS6 coming soon
- PA3 available!
- Grading update
  - PS0-2 grades returned, PS3 coming very soon
  - Regrade requests:
    - PS0-2 open through Sunday 3/17pm
    - PS3 and onward: 7 days after release
- Office hours (reminder)
  - Prof Hott Office Hours: Mondays 11a-12p, Fridays 10-11a and 2-3p
  - Prof Pettit Office Hours: Mondays and Fridays 2:30-4:00p
  - TA office hours posted on our website
  - Office hours are not for "checking solutions"

Reminders about Greedy Algorithms

Greedy Algorithms



Require two things:

- Optimal Substructure
- Greedy Choice Function

Optimal Substructure:

- If A is an optimal solution to a problem, then the components of A are optimal solutions to subproblems

Greedy Choice Function

- The rule for how to choose an item guaranteed be in the optimal solution

Greedy Algorithm Procedure:

- Apply the Greedy Choice Function to pick an item
- Identify your subproblem, then solve it

Prim's Algorithm Implementation

1. Start with an empty tree T and pick a start node and add it to T
2. Repeat $|V| - 1$ times:
   - Add the min-weight edge which connects a node in T with a node not in T

Implementation:

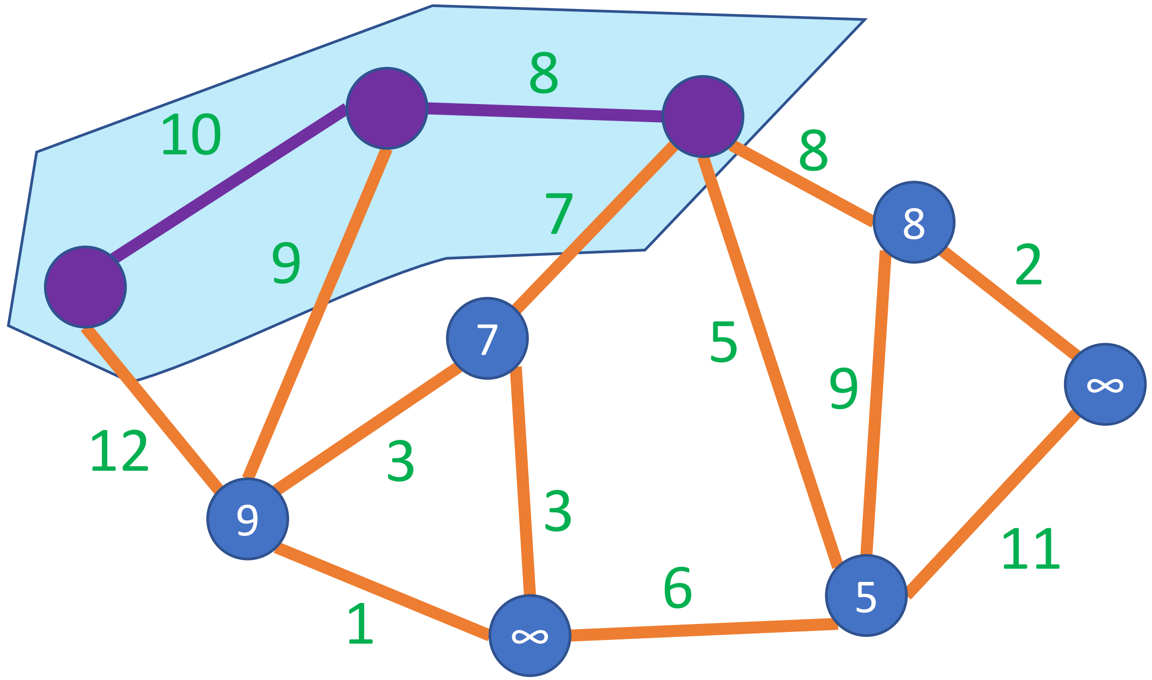
initialize $d_v = \infty$ for each node v
add all nodes $v \in V$ to the priority queue PQ, using d_v as the key
pick a starting node s and set $d_s = 0$
while PQ is not empty:
    $v = \mathrm{PQ.extractMin}()$
    for each $u \in V$ such that $(v, u) \in E$:
        if $u \in \mathrm{PQ}$ and $w(v, u) < d_u$:
            $\mathrm{PQ.decreaseKey}(u, w(v, u))$
            $u.\mathrm{parent} = v$

each node also maintains a parent, initially NULL

key: minimum cost to connect u to nodes in PQ

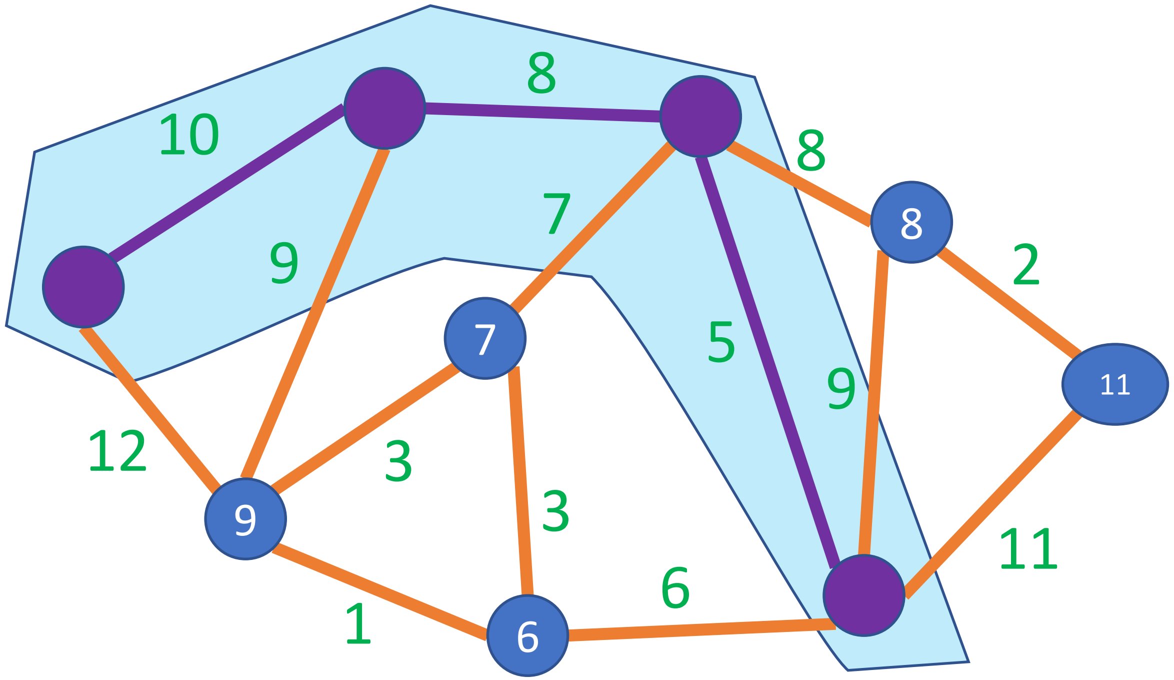
Prim's Algorithm

1. Start with an empty tree T and pick a start node and add it to T
2. Repeat $|V| - 1$ times:
   - Add the min-weight edge which connects a node in T with a node not in T



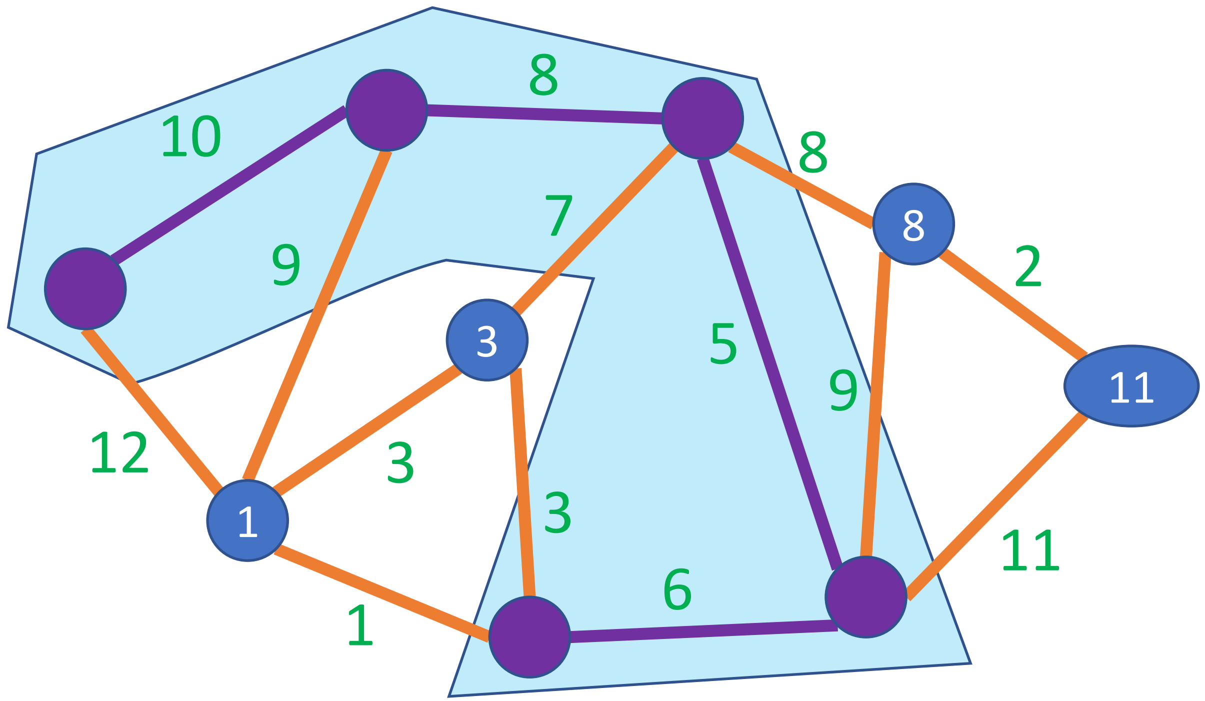
Prim's Algorithm

1. Start with an empty tree T and pick a start node and add it to T
2. Repeat $|V| - 1$ times:
   - Add the min-weight edge which connects a node in T with a node not in T



Prim's Algorithm

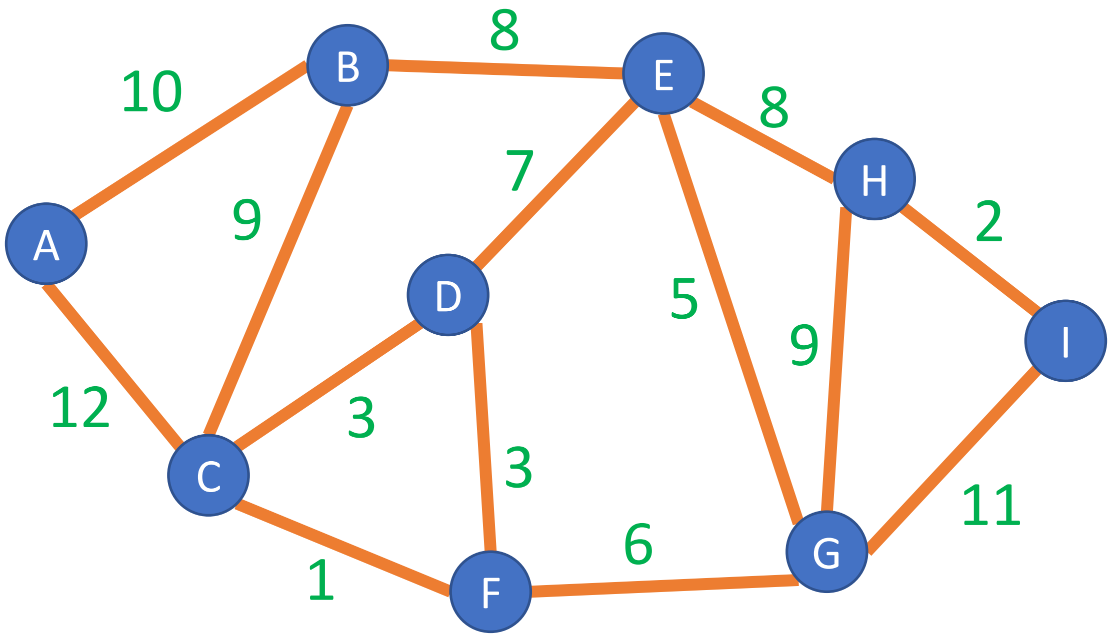
1. Start with an empty tree T and pick a start node and add it to T
2. Repeat $|V| - 1$ times:
   - Add the min-weight edge which connects a node in T with a node not in T



Kruskal's Algorithm

The *Greedy Choice* for Kruskal's

1. Start with an empty set of edges T
2. Repeatedly add to T the lowest-weight edge that does not create a cycle. (Stop when we've added $n - 1$ edges.)



Kruskal’s Algorithm

1. Start with an empty tree T
2. Repeatedly add to T the lowest-weight edge that does not create a cycle

Implementation: iterate over each of the edges in the graph (sorted by weight), and maintain nodes in a union-find (also called disjoint-set) data structure:

- Data structure that tracks elements partitioned into different sets
- **Union:** Merges two sets into one
- **Find:** Given an element, return the index of the set it belongs to
- Both “union” and “find” operations are very fast

Time complexity: $O(\alpha(n))$,
where α is the “inverse Ackermann function” (extremely slow-growing function)
for all “practical” n, $\alpha(n) < 5$ (e.g., for all $n < 2^{2^{2^{65536}}} - 3$)

Union/Find and Disjoint Sets

An Abstract Data Type (ADT) for a collection of sets of any kind of item, where an item can only belong to one of the sets

- We'll assume each item is identified by a unique integer value

Need to support the following operations

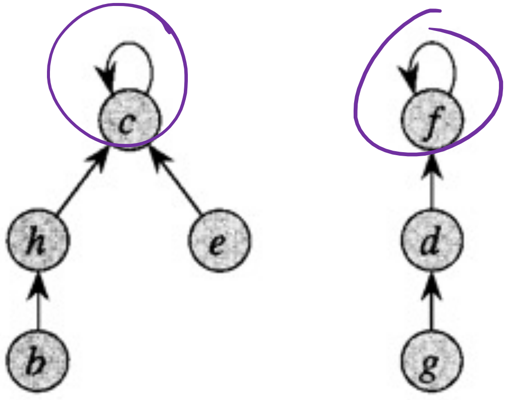
- void makeSet(int n) // construct n independent sets
- int findSet(int i) // given i, which set does i belong to?
- void union(int i, int j) // merge sets containing i and j

Union/Find and Disjoint Sets

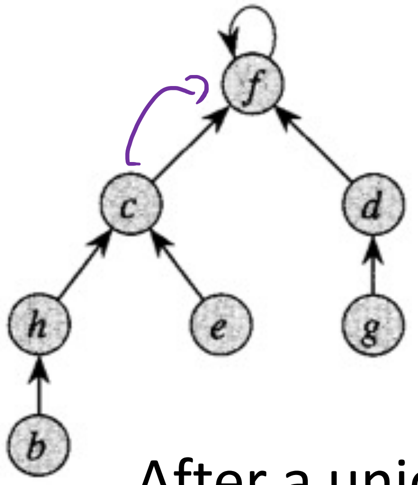
Represent Sets As Trees

item, "parent"

- Represent each set as a tree
- Identify set by its root node's ID (its "label")
  - findSet() means tracing up to root
  - union() makes one root child of the other root



Two sets



After a union

Union/Find and Disjoint Sets

Needs to support the following operations

- void makeSet(int n)   //construct n independent sets

Solution:

- Store as array of size n. Each location stores label for that set.

0	1	2	3	4	5	6	7	"parent"
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	values, items

Union/Find and Disjoint Sets

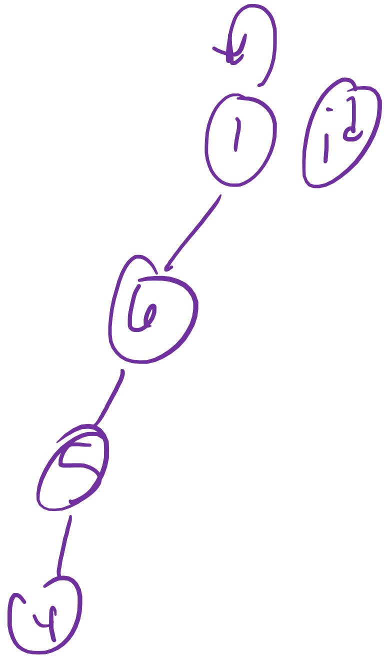
Needs to support the following operations

- int findSet(int i) //given i, which set does i belong to?

Solution: Trace around array until we find place where index and contents match

- Start at index i and repeat:
  - If a[i] == i then return i
  - Else set i = a[i]

0	1	3	3	5	6	1	7
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*



Union/Find and Disjoint Sets

Needs to support the following operations

- void union(int i, int j)   //merge sets i and j

Solution: find label for each set (call find() method), then set one label to point to other

- Label1 = find(i); Label2 = find(j)
- a[Label1] = Label2 //OR a[Label2] = Label1

union(2, 5)

3   1

0	1	3	~~3~~ 1	5	6	1	7
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*

Time Complexity: Kruskal's Algorithm

1. Start with an empty tree T
2. Repeatedly add to T the lowest-weight edge that does not create a cycle

Implementation: iterate over each of the edges in the graph (sorted by weight), and maintain nodes in a union-find (also called disjoint-set) data structure:

- Data structure that tracks elements partitioned into different sets
- **Union:** Merges two sets into one
- **Find:** Given an element, return the index of the set it belongs to
- Both "union" and "find" operations are very fast

$\log|V|^2$

- **Overall running time:** $O(|E| \log |E|) = O(|E| \log |V|)$

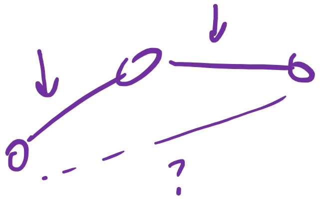
$|E| \leq |V|^2 \Rightarrow \log|E| = O(\log|V|)$

$E \to V^2$

More on Implementation for Kruskal’s

Let *EL* be the set of edges sorted ascending by weight

Consider each vertex to be in a tree of size 1



For each edge *e* in *EL*

T1 = tree ID for vertex *head(e)*

T2 = tree ID for vertex *tail(e)*

if (*T1* != *T2*)   *// the nodes are not in the same Tree*

Add *e* to the output set of edges *T* (which becomes the MST)

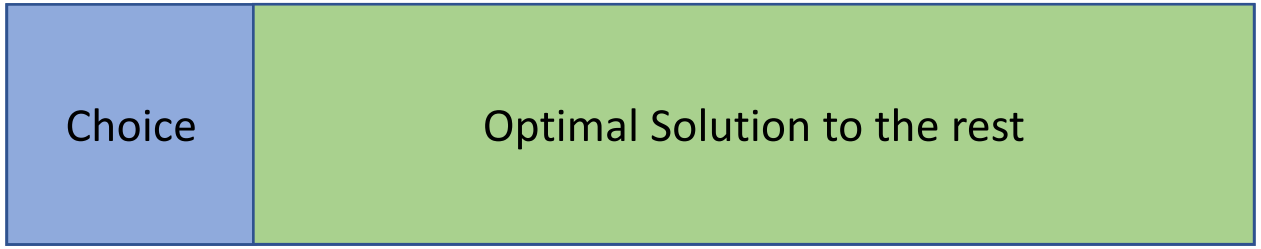
Combine trees *T1* and *T2*

Seems simple, no?

- But, how do you keep track of what tree a vertex is in?
- Trees are sets of vertices. Need to findset(v) and “union” two sets

Greedy Algorithms

Optimal Solution to big problem



Require two things:

- Optimal Substructure
- Greedy Choice Function

Optimal Substructure:

- If A is an optimal solution to a problem, then the components of A are optimal solutions to subproblems

Greedy Choice Function

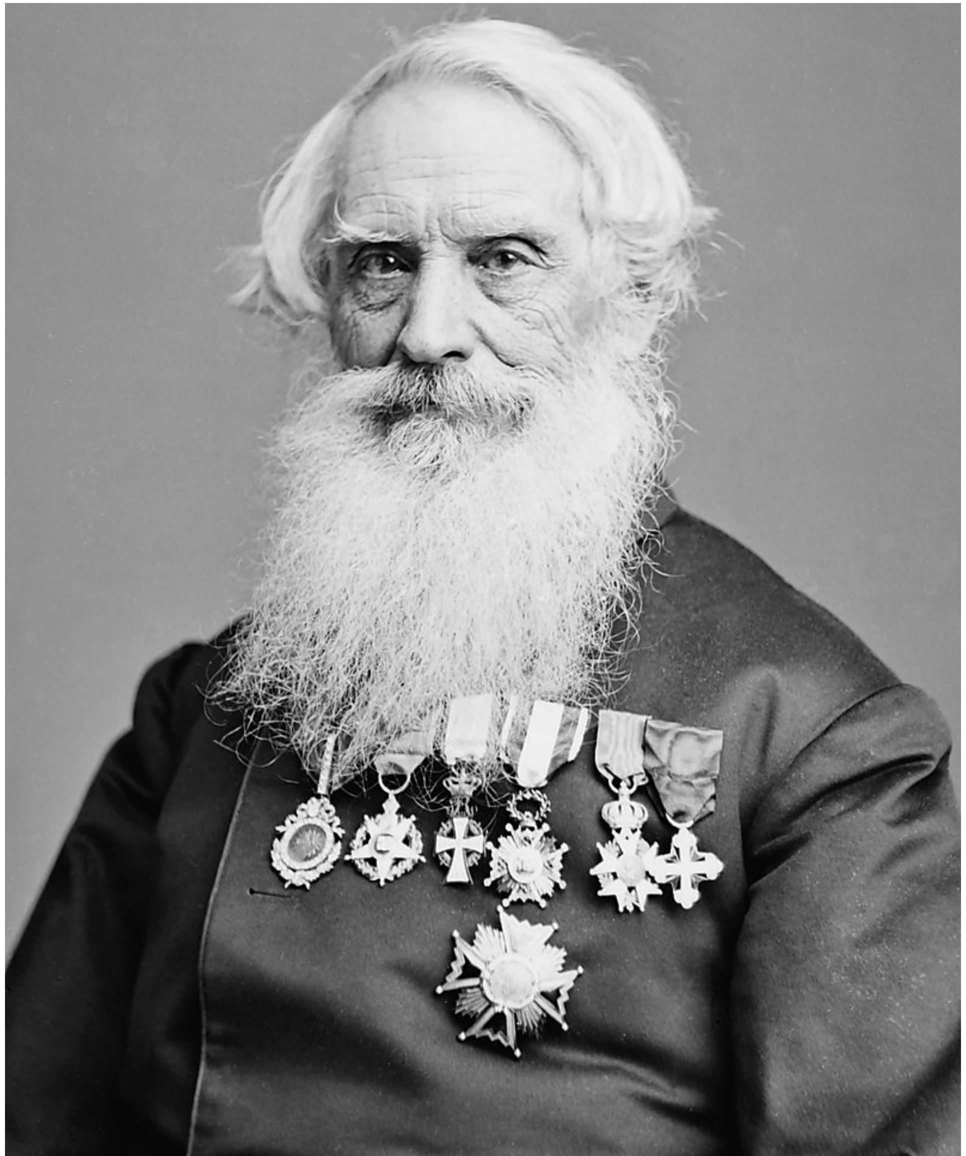
- The rule for how to choose an item guaranteed be in the optimal solution

Greedy Algorithm Procedure:

- Apply the Greedy Choice Function to pick an item
- Identify your subproblem, then solve it

Sam Morse

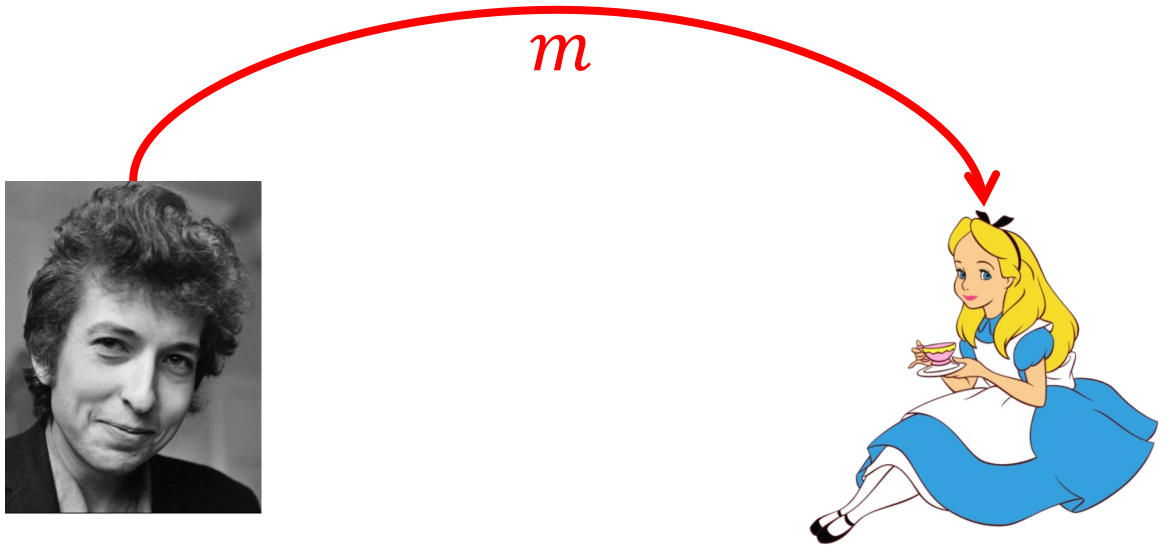
Engineer
and artist



Message Encoding

Problem: need to electronically send a message to two people at a distance.

Channel for message is binary (either on or off)



How can we do it?

wiggle, wiggle, wiggle like a gypsy queen
wiggle, wiggle, wiggle all dressed in green

Take the message, send it over character-by-character with an encoding

Character Frequency	Encoding
a: 2	0000
d: 2	0001
e: 13	0010
g: 14	0011
i: 8	0100
k: 1	0101
l: 9	0110
n: 3	0111
p: 1	1000
q: 1	1001
r: 2	1010
s: 3	1011
u: 1	1100
w: 6	1101
y: 2	1110

How efficient is this?

wiggle wiggle wiggle like a gypsy queen
wiggle wiggle wiggle all dressed in green

Each character requires 4 bits

$$\ell_c = 4$$

Cost of encoding:

codeword length

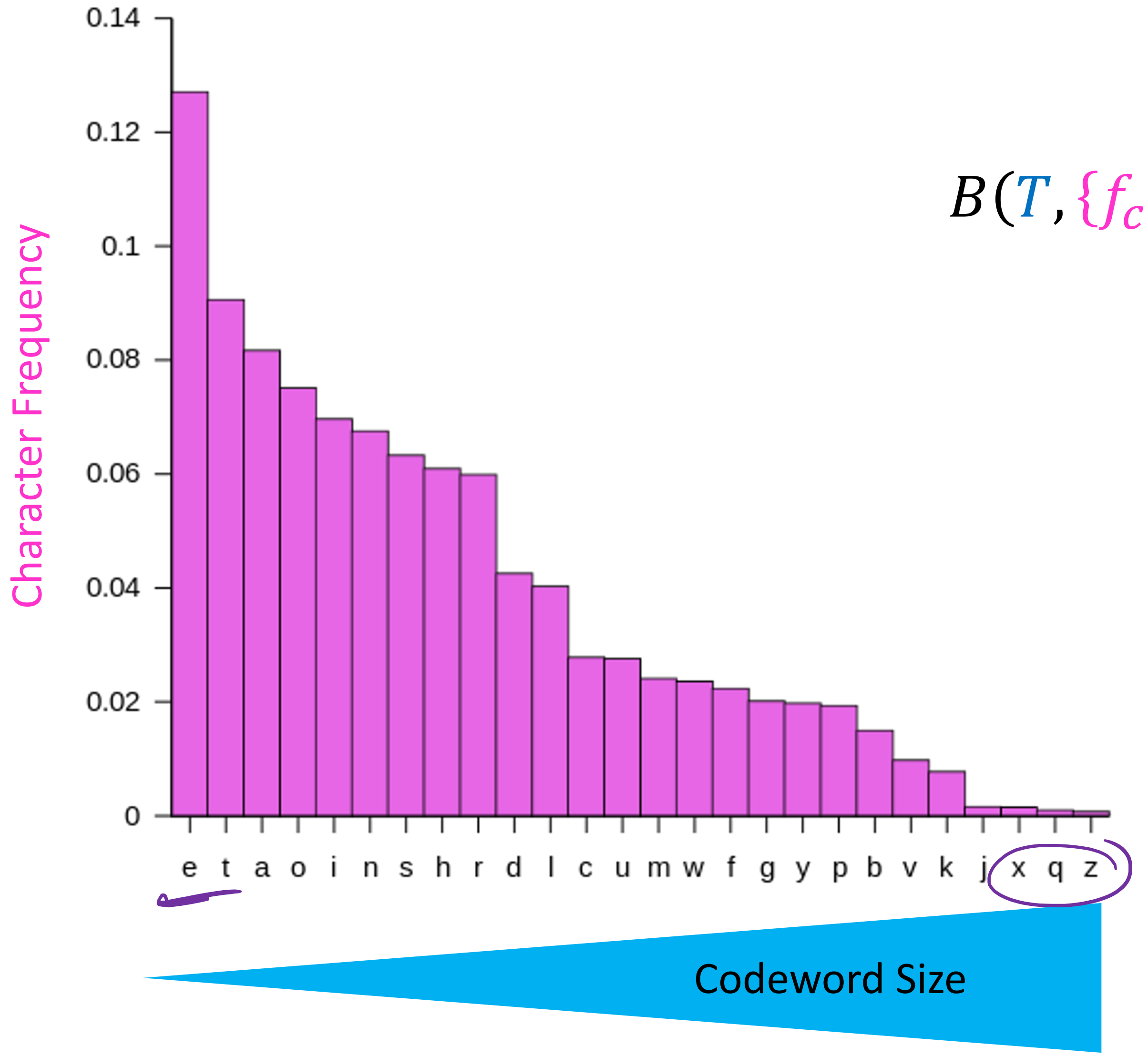
$$B(T, \{f_c\}) = \sum_{\text{character } c} \ell_c f_c = 68 \cdot 4 = 272$$

frequency

Better Solution: Allow for different characters to have different-size encodings (high frequency → short code)

Character Frequency	Encoding
a: 2	0000
d: 2	0001
e: 13	0010
g: 14	0011
i: 8	0100
k: 1	0101
l: 9	0110
n: 3	0111
p: 1	1000
q: 1	1001
r: 2	1010
s: 3	1011
u: 1	1100
w: 6	1101
y: 2	1110

More efficient coding

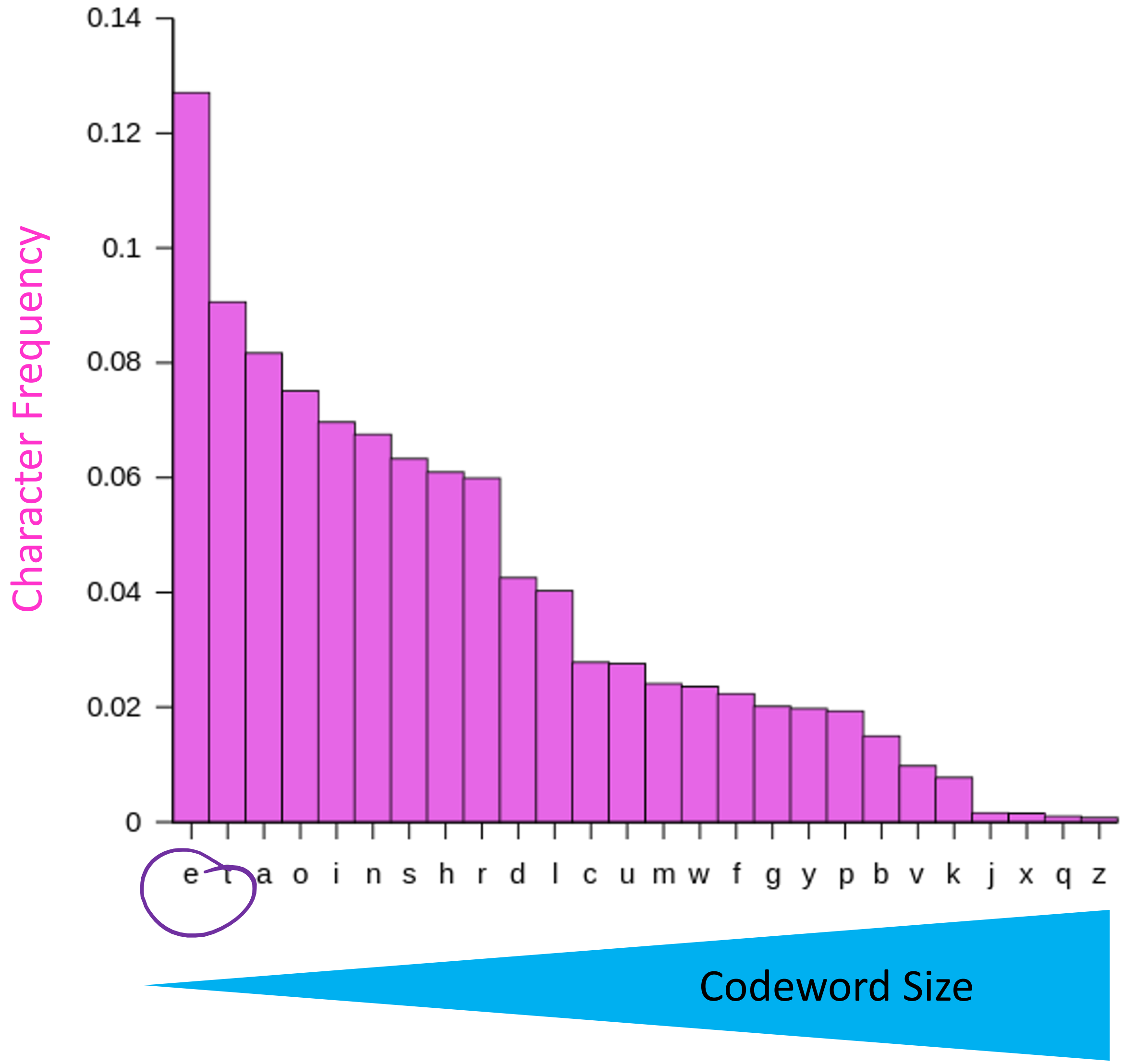


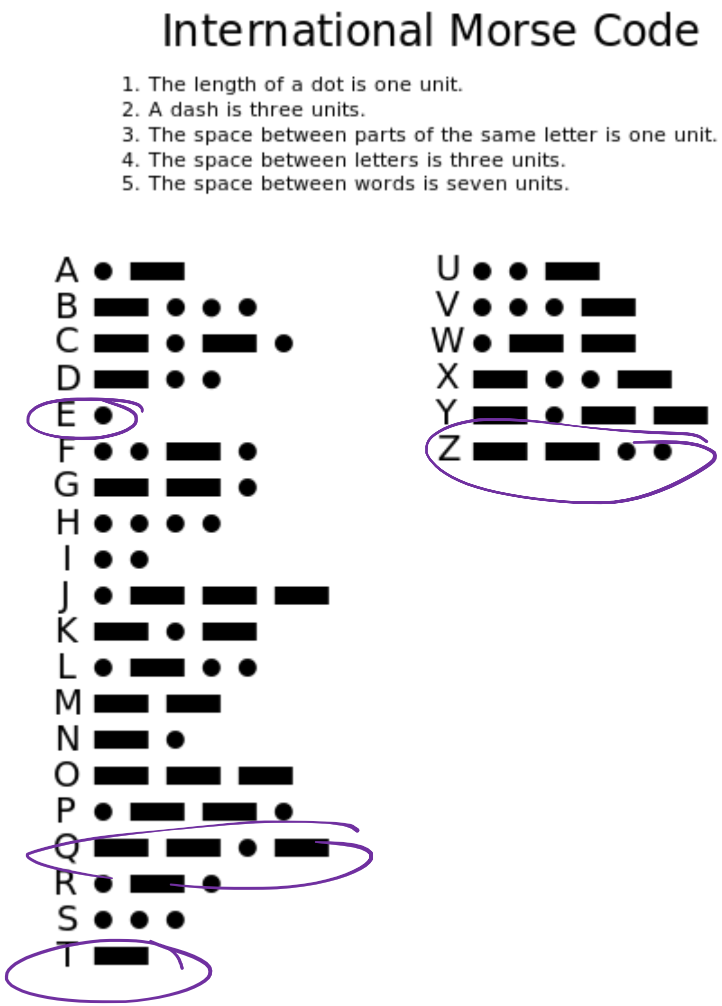
When this is big

$$B(T, \{f_c\}) = \sum_{character\ c} \ell_c f_c$$

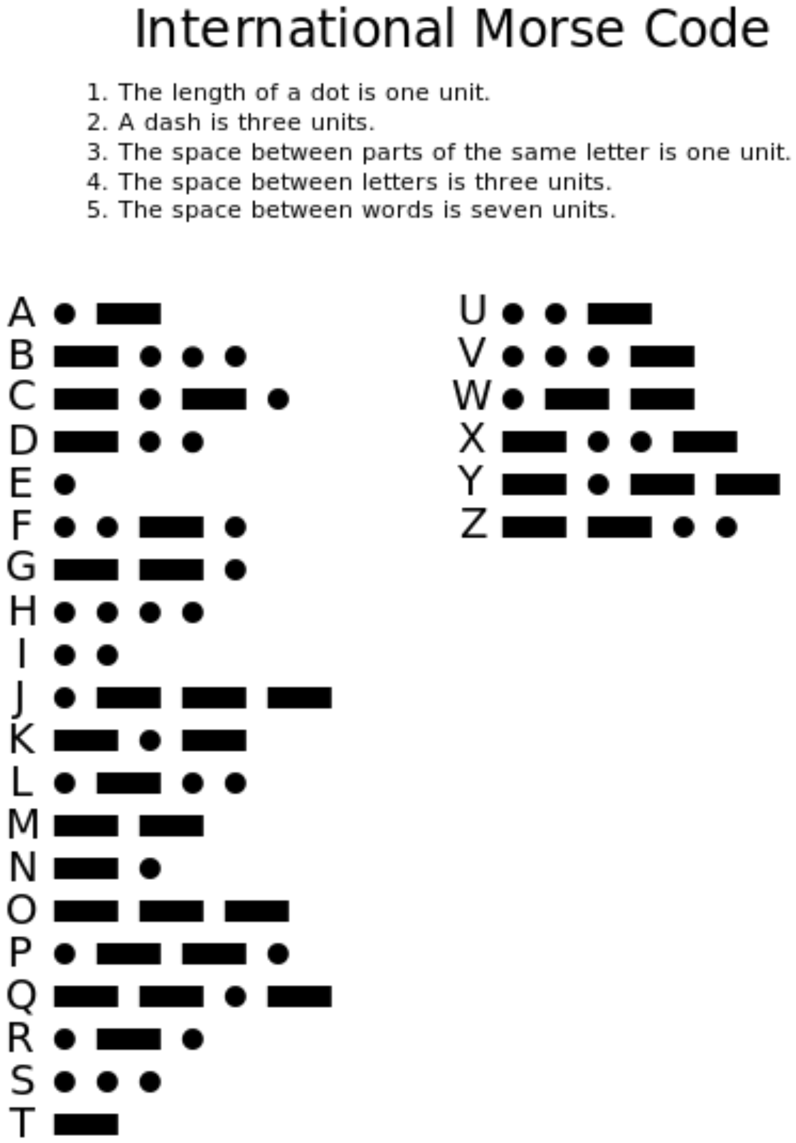
Make this small

Morse Code


Character Frequency
0.14
0.12
0.1
0.08
0.06
0.04
0.02
0
e t a o i n s h r d l c u m w f g y p b v k j x q z
Codeword Size


International Morse Code
1. The length of a dot is one unit.
2. A dash is three units.
3. The space between parts of the same letter is one unit.
4. The space between letters is three units.
5. The space between words is seven units.
A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P
Q
R
S
T
U
V
W
X
Y
Z

Problem with Morse Code



	A	A
Decode:	• ▬	• ▬
	ET	ET
	R	T
	EN	T

Ambiguous Decoding

Prefix-Free Code

A prefix-free code is codeword table T such that for any two characters c_1, c_2, if $c_1 \neq c_2$ then $code(c_1)$ is not a prefix of $code(c_2)$

g	0
e	10
l	110
i	1110
w	11110
…	…

11110111000011010

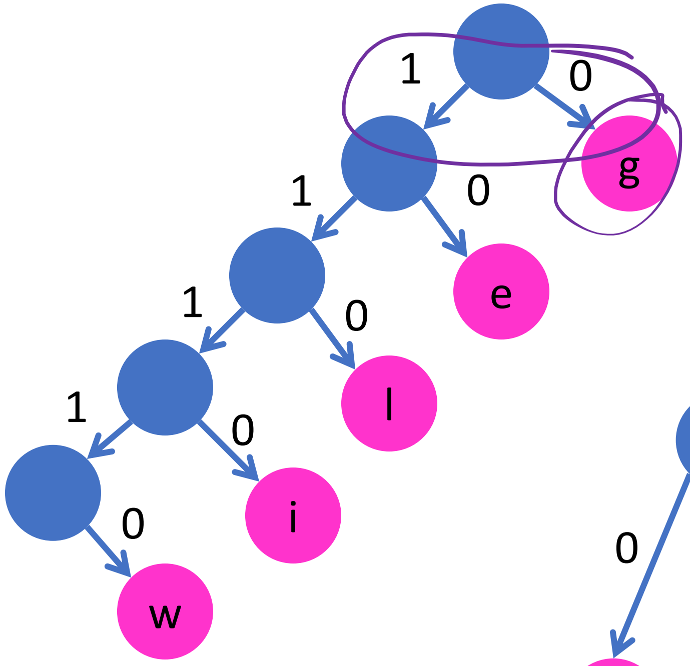
w i gg l e

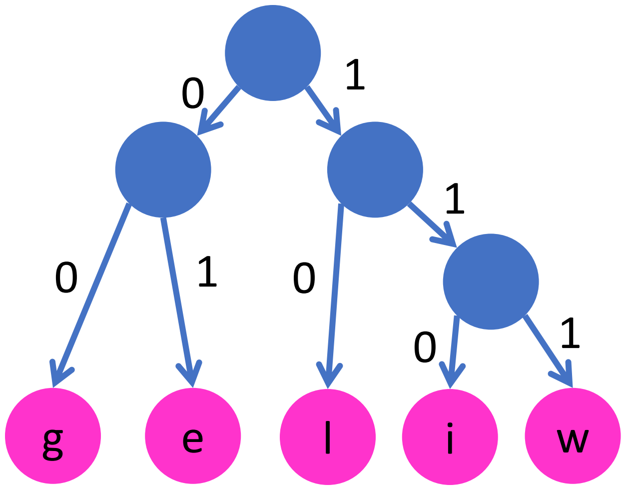
Binary Trees = Prefix-free Codes

I can represent any prefix-free code as a binary tree

I can create a prefix-free code from any binary tree

g	0
e	10
l	110
i	1110
w	11110
…	…





g	00
e	01
l	10
i	110
w	111
…	…

Goal: Shortest Prefix-Free Encoding

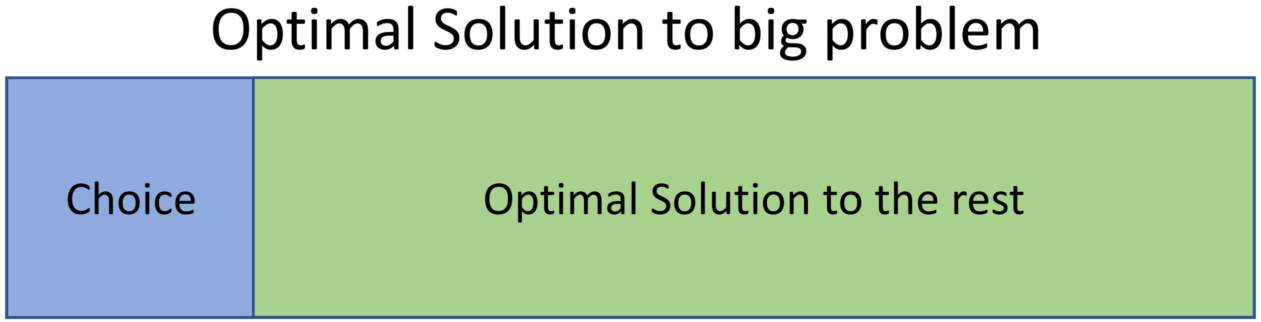
Input: A set of character frequencies $\{f_c\}$

Output: A prefix-free code T which minimizes

$$B(T, \{f_c\}) = \sum_{\text{character } c} \ell_c f_c$$

Huffman Coding!!

Greedy Algorithms



Require two things:

- Optimal Substructure
- Greedy Choice Function

Optimal Substructure:

- If A is an optimal solution to a problem, then the components of A are optimal solutions to subproblems

Greedy Choice Function

- The rule for how to choose an item guaranteed be in the optimal solution

Greedy Algorithm Procedure:

- Apply the Greedy Choice Function to pick an item
- Identify your subproblem, then solve it

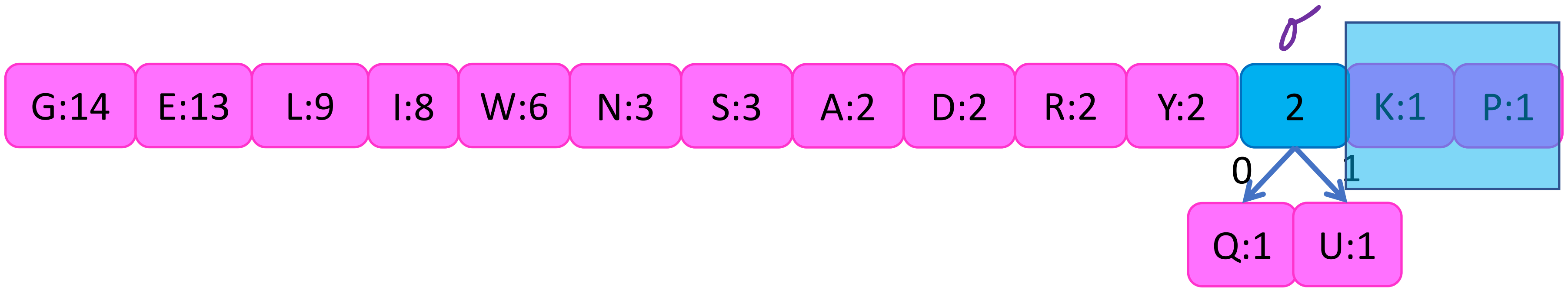
Huffman Algorithm

Choose the least frequent pair, combine into a subtree

G:14 E:13 L:9 I:8 W:6 N:3 S:3 A:2 D:2 R:2 Y:2 K:1 P:1 Q:1 U:1

Huffman Algorithm

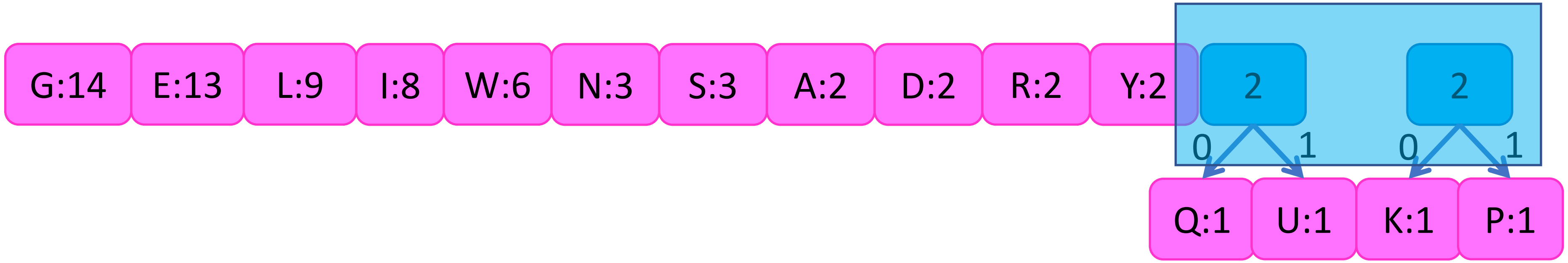
Choose the least frequent pair, combine into a subtree



Subproblem of size $n-1$!

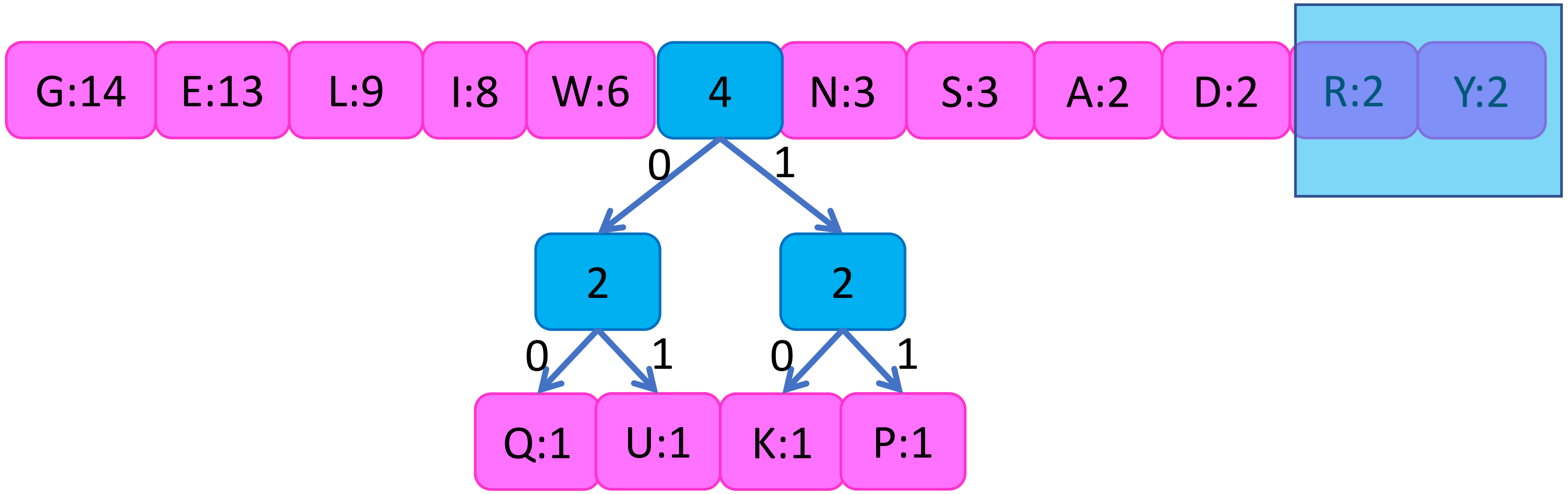
Huffman Algorithm

Choose the least frequent pair, combine into a subtree



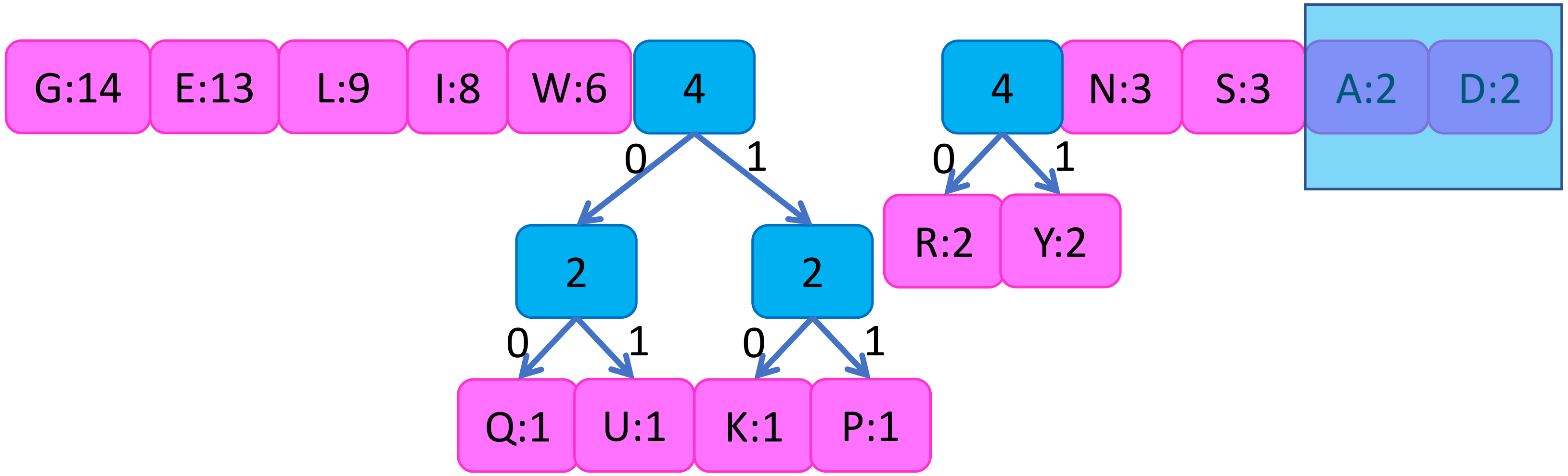
Huffman Algorithm

Choose the least frequent pair, combine into a subtree



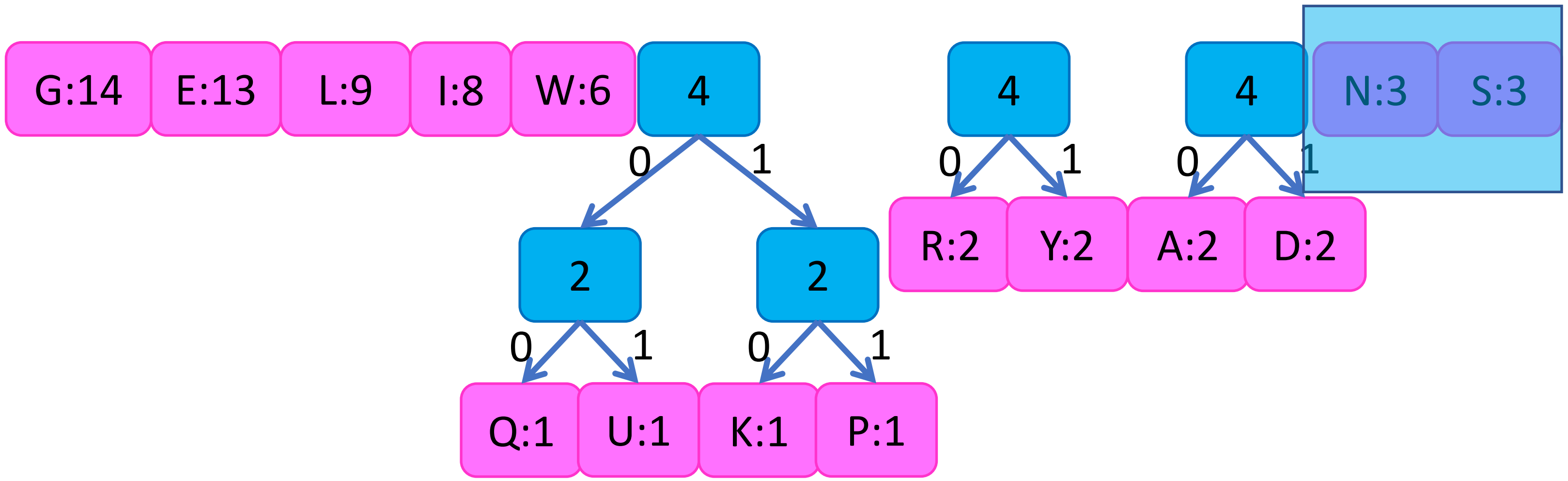
Huffman Algorithm

Choose the least frequent pair, combine into a subtree



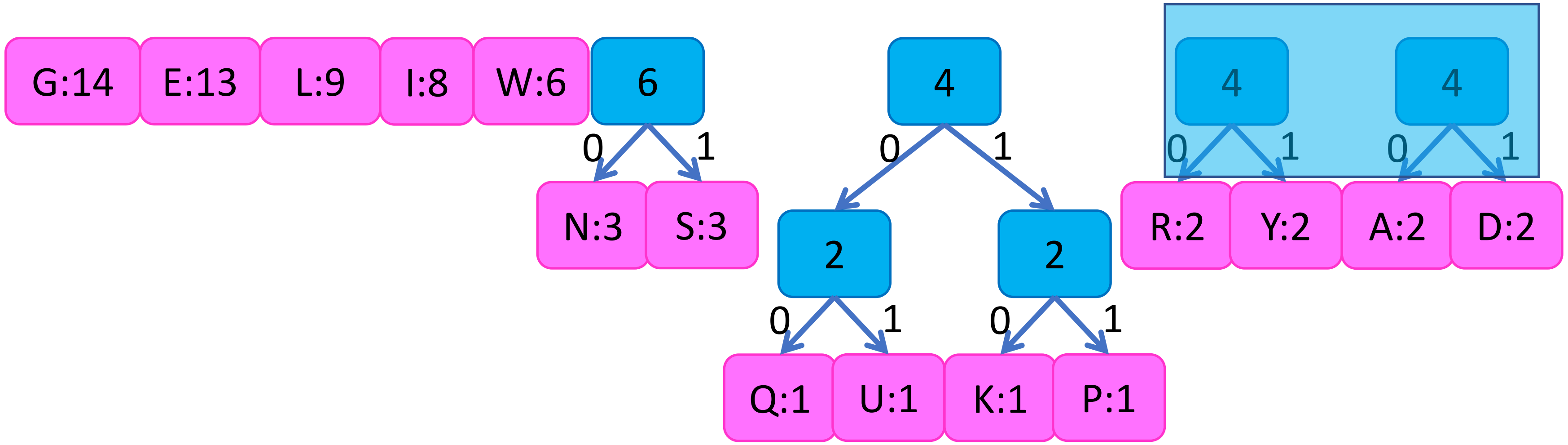
Huffman Algorithm

Choose the least frequent pair, combine into a subtree



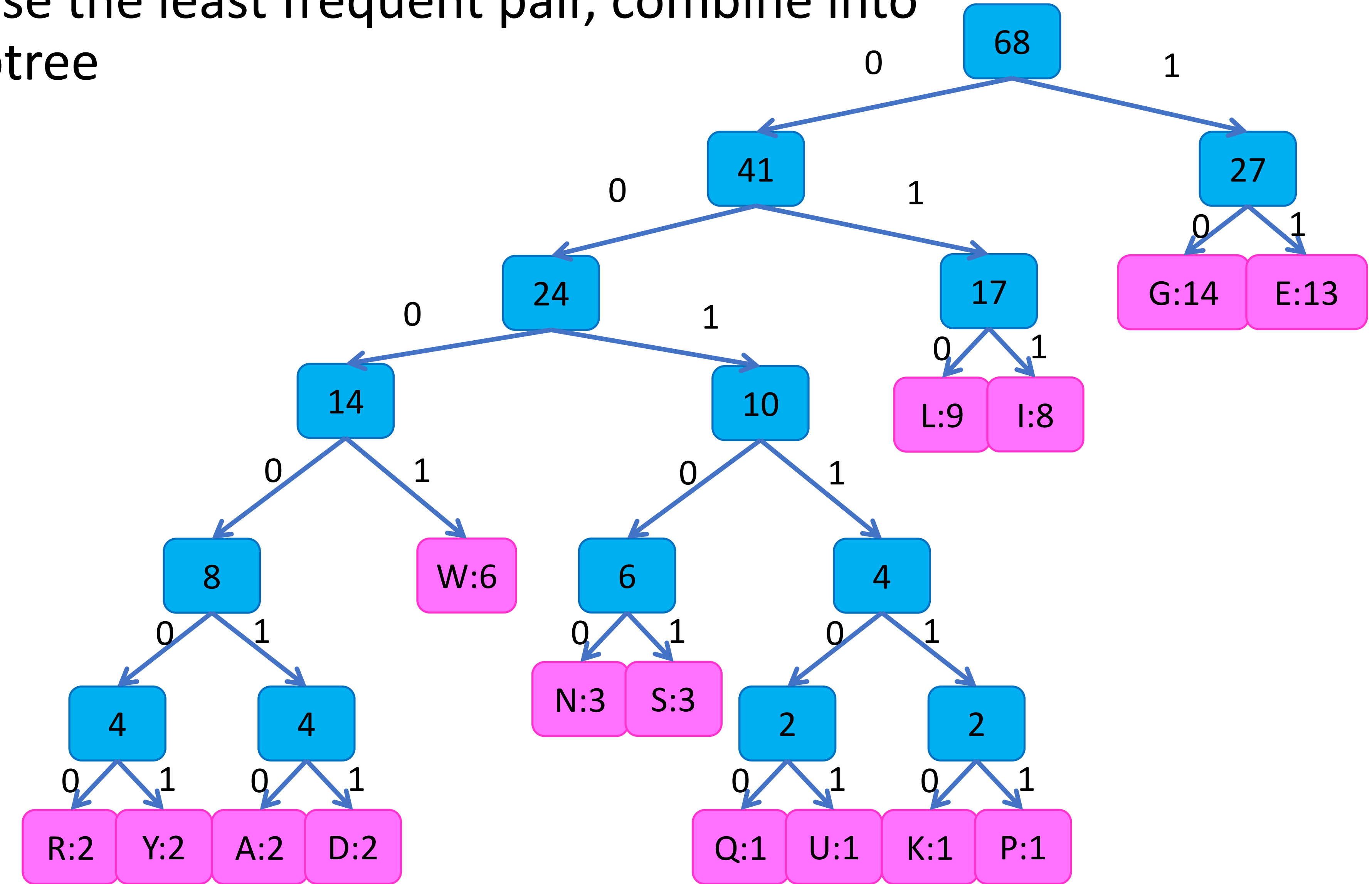
Huffman Algorithm

Choose the least frequent pair, combine into a subtree



Huffman Algorithm

Choose the least frequent pair, combine into a subtree



Exchange argument

Shows correctness of a greedy algorithm

Idea:

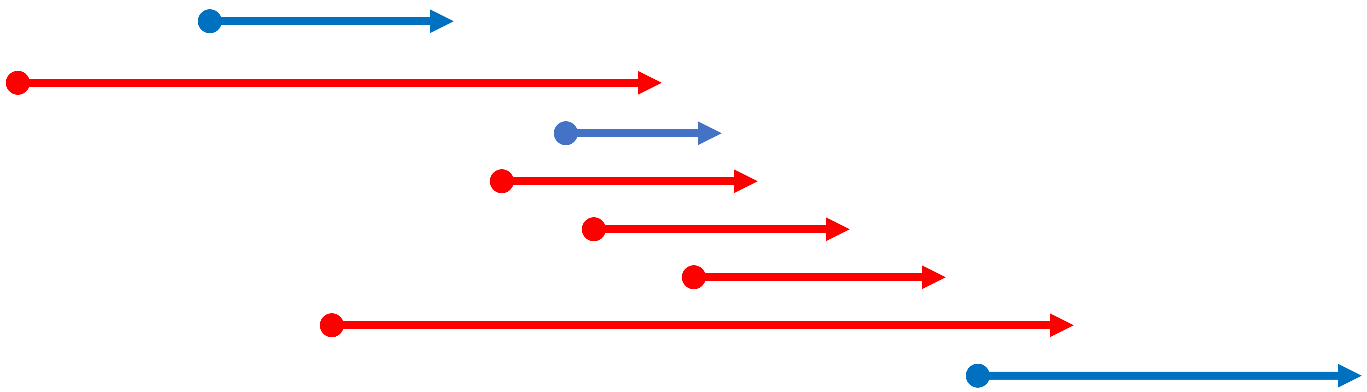
- Show exchanging an item from an arbitrary optimal solution with your greedy choice makes the new solution no worse
- How to show my sandwich is at least as good as yours:
  - Show: “I can remove any item from your sandwich, and it would be no worse by replacing it with the same item from my sandwich”

Remember: Interval Scheduling Algorithm

Find event ending earliest, add to solution,

Remove it and all conflicting events,

Repeat until all events removed, return solution



Remember: Exchange Argument

Claim: earliest ending interval is always part of some optimal solution

Let $OPT_{i,j}$ be an optimal solution for time range $[i,j]$

Let a^* be the first interval in $[i,j]$ to finish overall (greedy choice)

If $a^* \in OPT_{i,j}$ then claim holds

Else if $a^* \notin OPT_{i,j}$, let a be the first interval to end in $OPT_{i,j}$

- By definition a^* ends before a, and therefore does not conflict with any other events in $OPT_{i,j}$
- Therefore $OPT_{i,j} - \{a\} + \{a^*\}$ is also an optimal solution (same number events)
- Thus claim holds

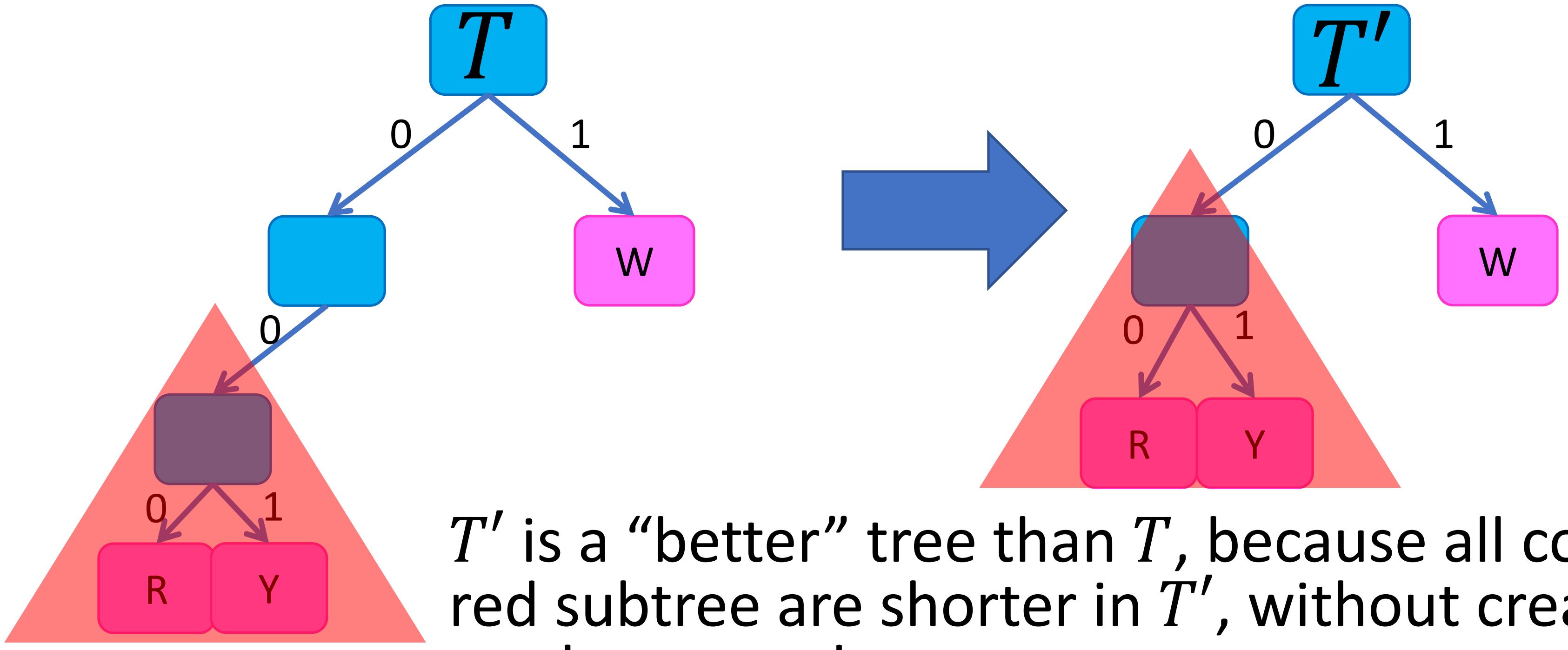
Showing Huffman is Optimal

Overview:

- Show that there is **an** optimal tree in which the least frequent characters are siblings
  - Exchange argument
- Show that making them siblings and solving the new smaller sub-problem results in **an** optimal solution
  - Optimal Substructure argument

Showing Huffman is Optimal

First Step: Show any optimal tree is “full” (each node has either 0 or 2 children)



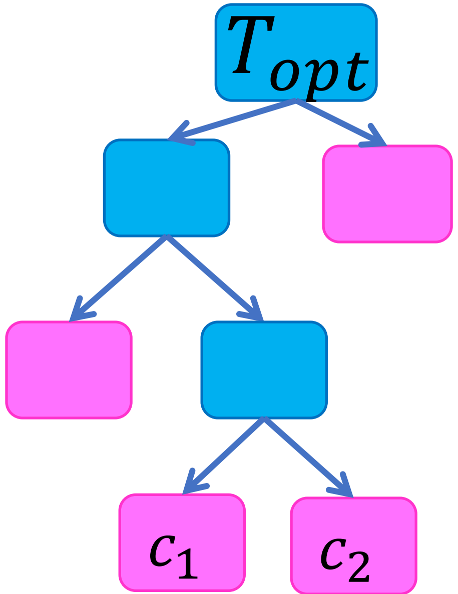
T' is a “better” tree than T, because all codes in red subtree are shorter in T', without creating any longer codes

Huffman Exchange Argument

Claim: if c_1, c_2 are the least-frequent characters, then there is an optimal prefix-free code s.t. c_1, c_2 are siblings

- i.e. codes for c_1, c_2 are the same length and differ only by their last bit

Case 1: Consider some optimal tree T_{opt}. If c_1, c_2 are siblings in this tree, then claim holds

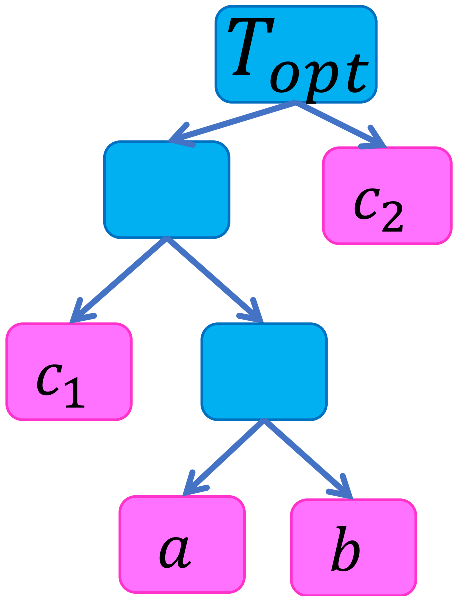


Huffman Exchange Argument

Claim: if c_1, c_2 are the least-frequent characters, then there is an optimal prefix-free code s.t. c_1, c_2 are siblings

- i.e. codes for c_1, c_2 are the same length and differ only by their last bit

Case 2: Consider some optimal tree T_{opt}, in which c_1, c_2 are not siblings



Let a, b be the two characters of lowest depth that are siblings
(Why must they exist?)

Idea: show that swapping c_1 with a does not increase cost of the tree.
Similar for c_2 and b
Assume: $f_{c1} \le f_a$ and $f_{c2} \le f_b$

Case 2: c_1, c_2 are not siblings in T_{opt}

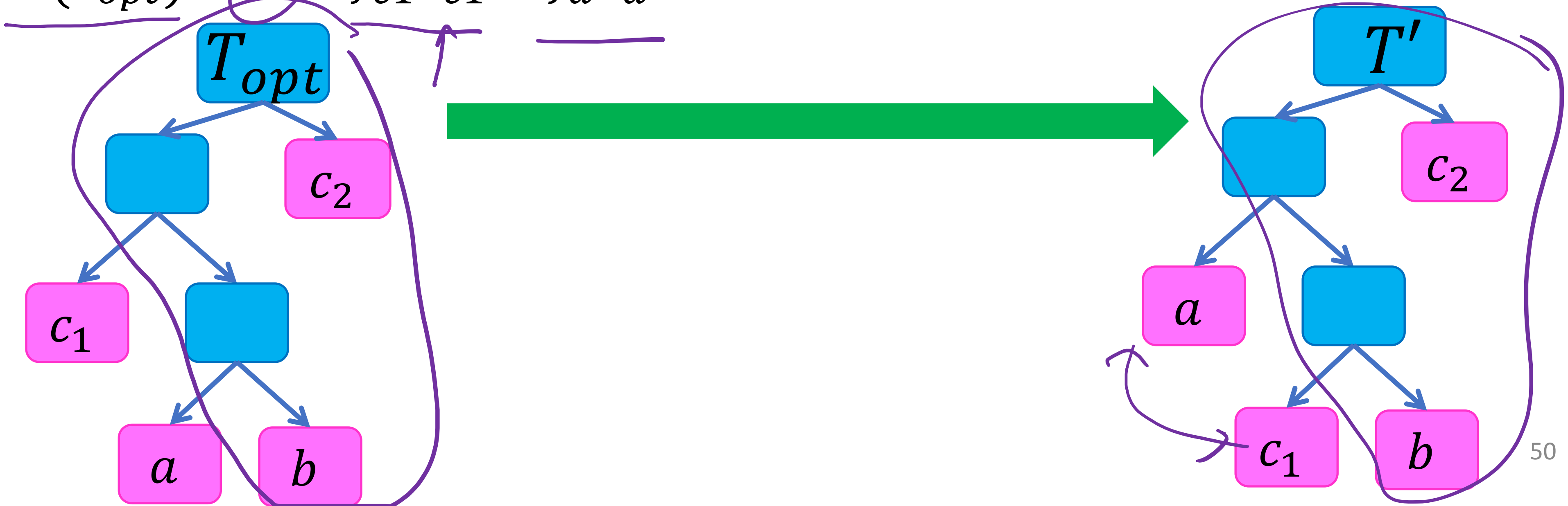
- Claim: the least-frequent characters (c_1, c_2), are siblings in some optimal tree

a, b = lowest-depth siblings

Idea: show that swapping c_1 with a does not increase cost of the tree.
Assume: $f_{c1} \le f_a$

$B(T_{opt}) = C + f_{c1}\ell_{c1} + f_a\ell_a$

$B(T') = C + f_{c1}\overset{c_1}{\ell_a} + f_a\overset{a}{\ell_{c1}}$



Case 2: c_1, c_2 are not siblings in T_{opt}

- Claim: the least-frequent characters (c_1, c_2), are siblings in some optimal tree

a, b = lowest-depth siblings

Idea: show that swapping c_1 with a does not increase cost of the tree.
Assume: $f_{c1} \leq f_a$

$$B(T_{opt}) = C + f_{c1}\ell_{c1} + f_a\ell_a$$

$$B(T') = C + f_{c1}\ell_a + f_a\ell_{c1}$$

$\geq 0 \Rightarrow T'$ optimal

$$B(T_{opt}) - B(T') = \cancel{C} + f_{c1}\ell_{c1} + f_a\ell_a - (\cancel{C} + f_{c1}\ell_a + f_a\ell_{c1})$$

$$= f_{c1}\ell_{c1} + f_a\ell_a - f_{c1}\ell_a - f_a\ell_{c1}$$

$$= f_{c1}(\ell_{c1} - \ell_a) + f_a(\ell_a - \ell_{c1})$$

$$= (f_a - f_{c1})(\ell_a - \ell_{c1})$$

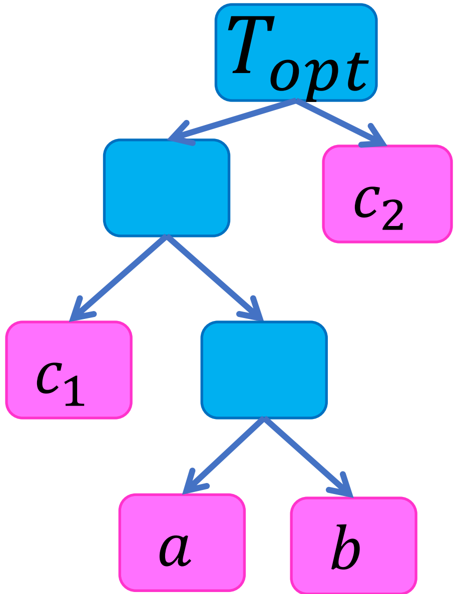
Case 2: c_1, c_2 are not siblings in T_{opt}

- Claim: the least-frequent characters (c_1, c_2), are siblings in some optimal tree

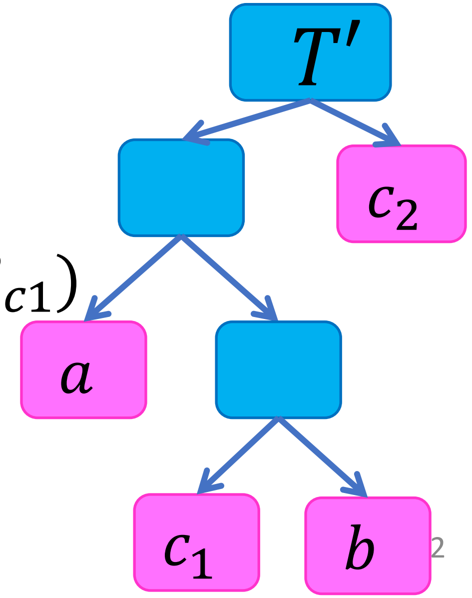
a, b = lowest-depth siblings

Idea: show that swapping c_1 with a does not increase cost of the tree.
Assume: $f_{c1} \leq f_a$

$$B(T_{opt}) = C + f_{c1}\ell_{c1} + f_a\ell_a$$



$$B(T') = C + f_{c1}\ell_a + f_a\ell_{c1}$$



$$B(T_{opt}) - B(T') = \underbrace{(f_a - f_{c1})}_{\geq 0}\underbrace{(\ell_a - \ell_{c1})}_{\geq 0}$$

$$B(T_{opt}) - B(T') \geq 0$$

T' is also optimal!

Case 2: Repeat to swap c_2, b!

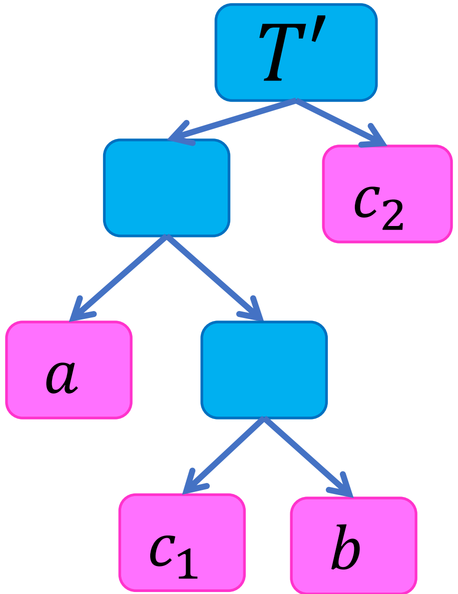
- Claim: the least-frequent characters (c_1, c_2), are siblings in some optimal tree

a, b = lowest-depth siblings

Idea: show that swapping c_2 with b does not increase cost of the tree.
Assume: $f_{c2} \leq f_b$

$$B(T') = C + f_{c2}\ell_{c2} + f_b\ell_b$$

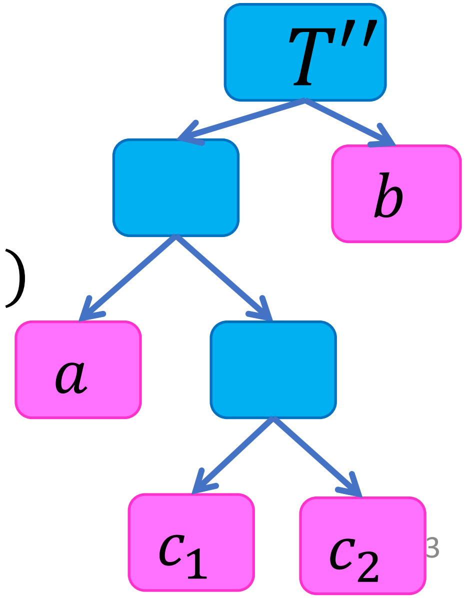
$$B(T'') = C + f_{c2}\ell_b + f_b\ell_{c2}$$



$$B(T') - B(T'') = (\underbrace{f_b - f_{c2}}_{\geq 0})(\underbrace{\ell_b - \ell_{c2}}_{\geq 0})$$

$$B(T') - B(T'') \geq 0$$

T'' is also optimal! Claim holds!



Showing Huffman is Optimal

Overview:

- Show that there is **an** optimal tree in which the least frequent characters are siblings
  - Exchange argument
- Show that making them siblings and solving the new smaller sub-problem results in **an** optimal solution
  - Optimal Substructure argument

Proving Optimal Substructure

Goal: show that if x is in an optimal solution, then the rest of the solution is an optimal solution to the subproblem.

Usually by Contradiction:

- Assume that x must be an element of my optimal solution
- Assume that solving the subproblem induced from choice x, then adding in x is not optimal
- Show that removing x from a better overall solution must produce a better solution to the subproblem

Huffman Optimal Substructure

Goal: show that if c_1, c_2 are siblings in an optimal solution, then an optimal prefix free code can be found by using a new character with frequency $f_{c_1} + f_{c_2}$ and then making c_1, c_2 its children.

By Contradiction:

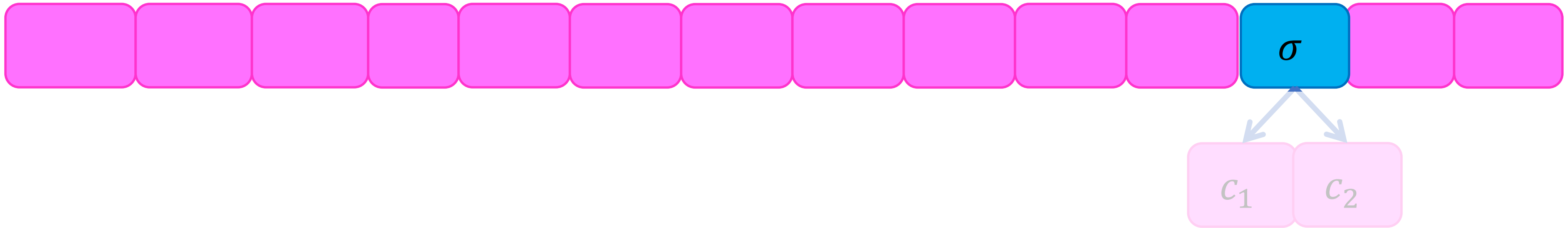
- Assume that c_1, c_2 are siblings in at least one optimal solution
- Assume that solving the subproblem with this new character, then adding in c_1, c_2 is not optimal
- Show that removing c_1, c_2 from a better overall solution must produce a better solution to the subproblem

Finishing the Proof

Show Recursive Substructure

- Show treating c_1, c_2 as a new “combined” character gives optimal solution

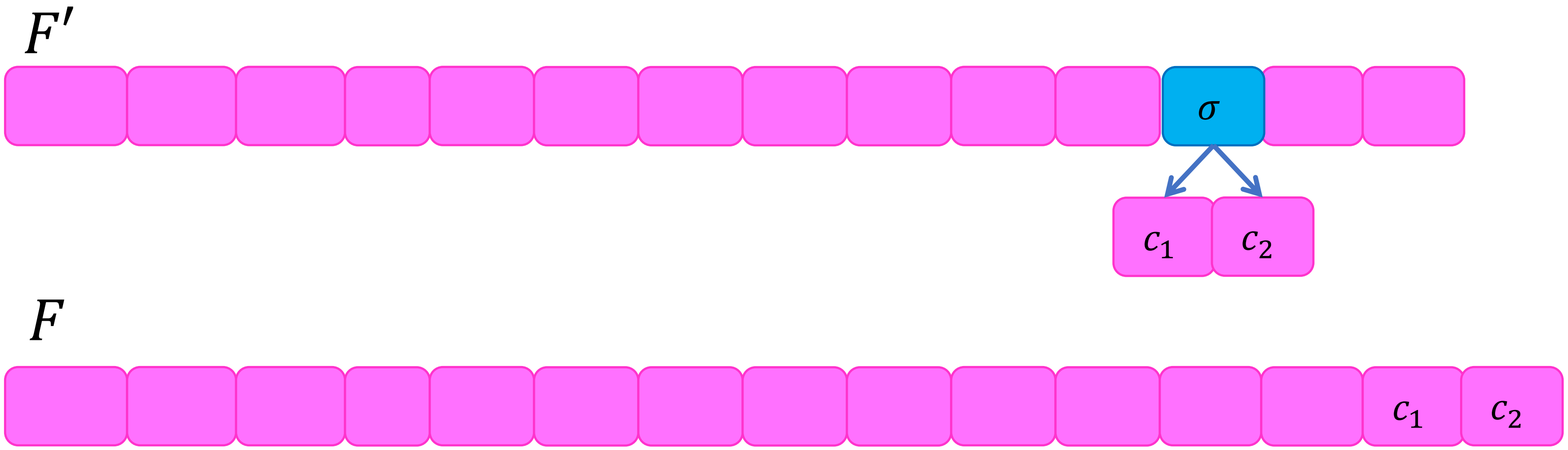
Why does solving this smaller problem:



Give an optimal solution to this?:

Substructure

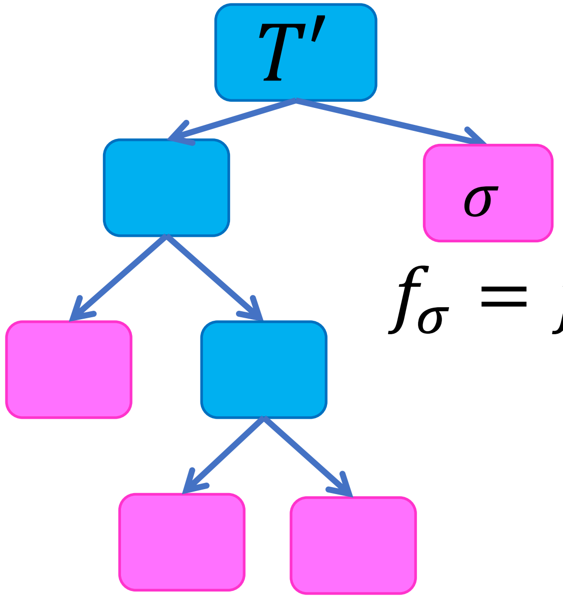
Claim: An optimal solution for F involves finding an optimal solution for F', then adding c_1, c_2 as children to σ



Substructure

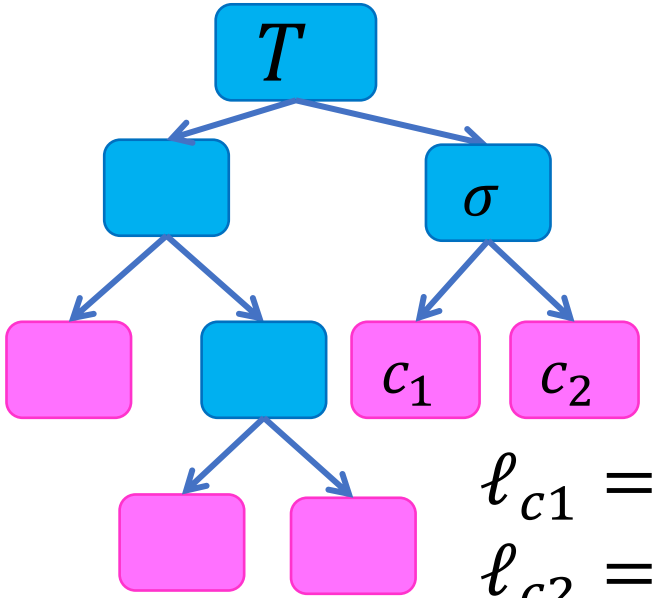
Claim: An optimal solution for F involves finding an optimal solution for F', then adding c_1, c_2 as children to σ

If this is optimal



$$f_\sigma = f_{c1} + f_{c2}$$

Then this is optimal



$$\ell_{c1} = \ell_\sigma + 1$$
$$\ell_{c2} = \ell_\sigma + 1$$

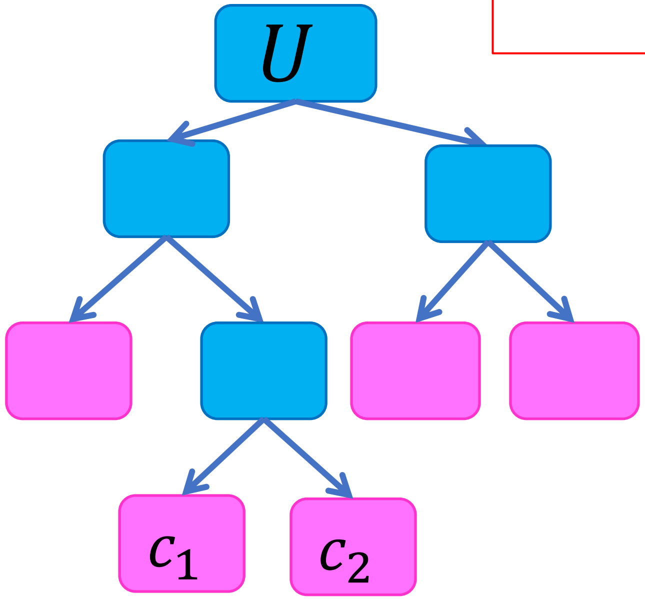
$$B(T') = B(T) - f_{c1} - f_{c2}$$

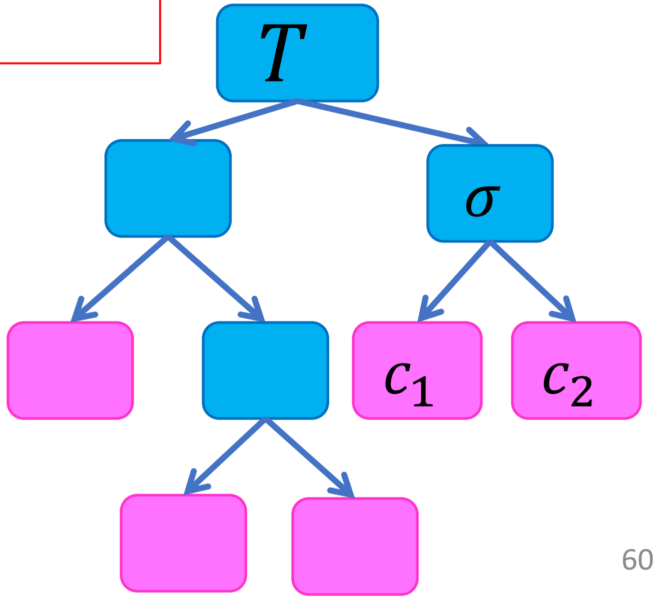
Substructure

Claim: An optimal solution for F involves finding an optimal solution for F', then adding c_1, c_2 as children to σ

Toward contradiction

Suppose T is not optimal
Let U be a lower-cost tree
$B(U) < B(T)$



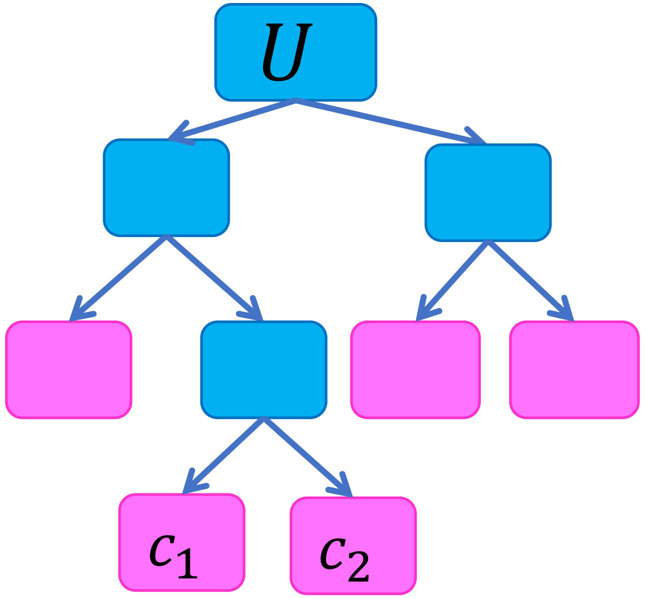


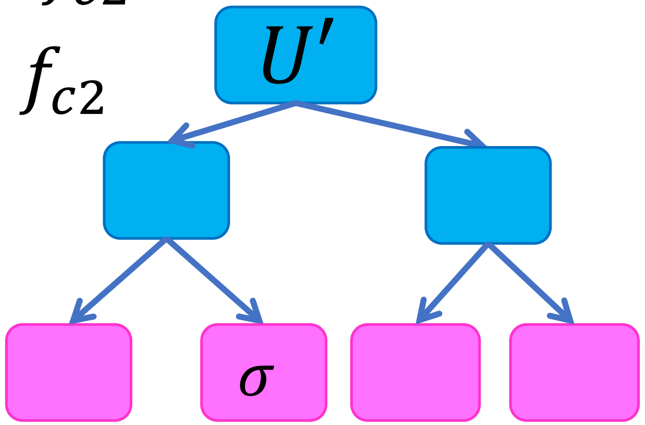
Substructure

Claim: An optimal solution for F involves finding an optimal solution for F', then adding c_1, c_2 as children to σ

$$B(U) < B(T)$$

$$\begin{aligned} B(U') &= B(U) - f_{c1} - f_{c2} \\ &< B(T) - f_{c1} - f_{c2} \\ &= B(T') \end{aligned}$$





Contradicts optimality of T', so T is optimal!

Optimal Substructure

Claim: An optimal solution for F involves finding an optimal solution for F', then adding c_1, c_2 as children to σ