CS4102 Algorithms

Spring 2022

<u>Warm up</u>

Decode the line below into English

(hint: use Google or Wolfram Alpha)

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

CS4102 Algorithms

Spring 2022

Warm up

Decode the line below into English

(hint: use Google or Wolfram Alpha)

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	— — • •

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

Today’s Keywords

- Greedy Algorithms
- Exchange Argument
- Choice Function
- Prefix-free code
- Compression
- Huffman Code

CLRS: Chapter 16

Announcements

- Unit B
  - Advanced due Friday, 11:30pm
  - Programming due Friday, 4/15, 11:30pm
- Unit C
  - Basic 1 released, due Friday, 4/15, 11:30pm
  - Basic 2 coming soon!
  - Advanced due Friday, 4/22
  - Programming due Friday 4/22 – Seam carving!

Greedy Algorithms

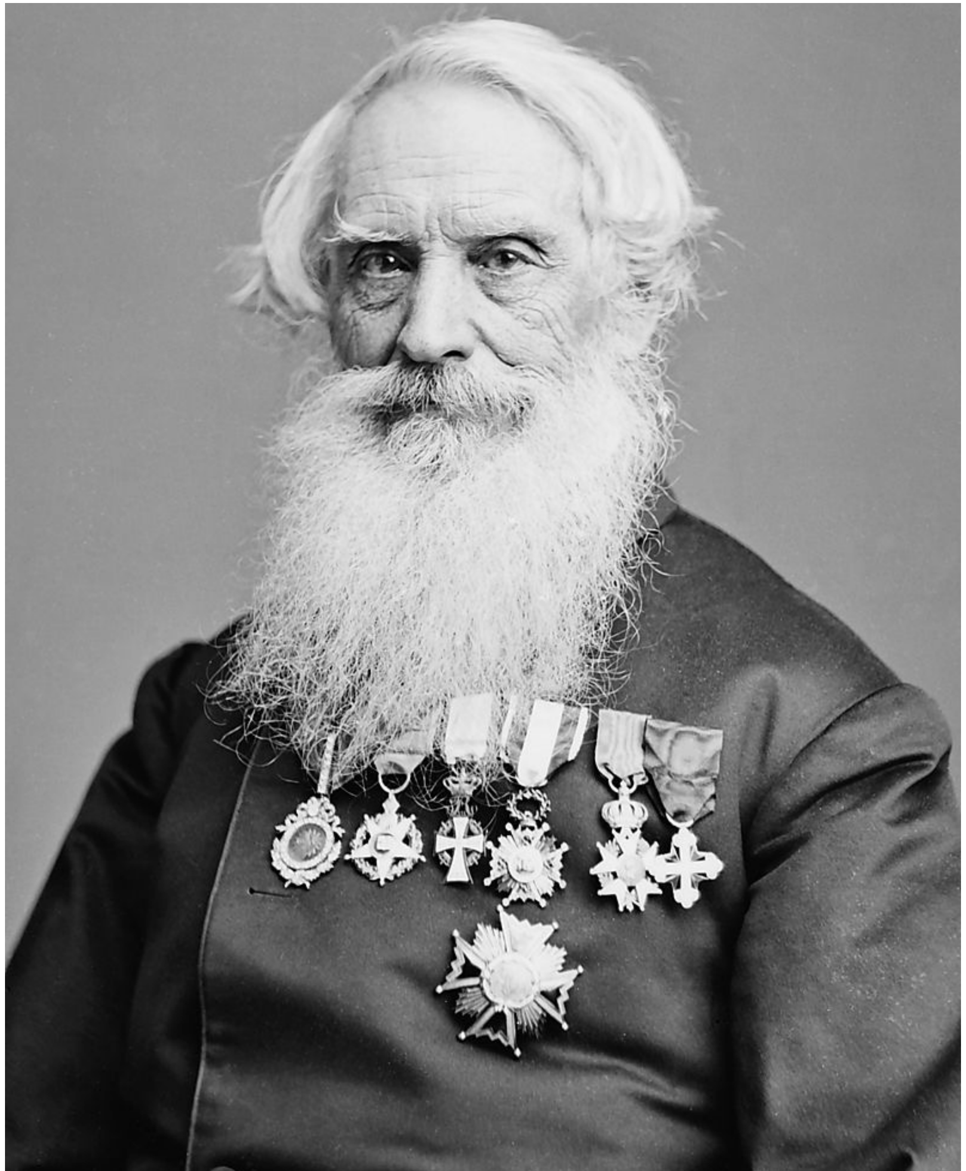
- Require Optimal Substructure
  - Solution to larger problem contains the solutions to smaller ones
    - Or: If S is an optimal solution to a problem, then the components of S are optimal solutions to sub-problems
  - Only one subproblem to consider!
- Idea:
  1. Identify a greedy choice property
     - How to make a choice guaranteed to be included in some optimal solution
  2. Repeatedly apply the choice property until no subproblems remain

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”

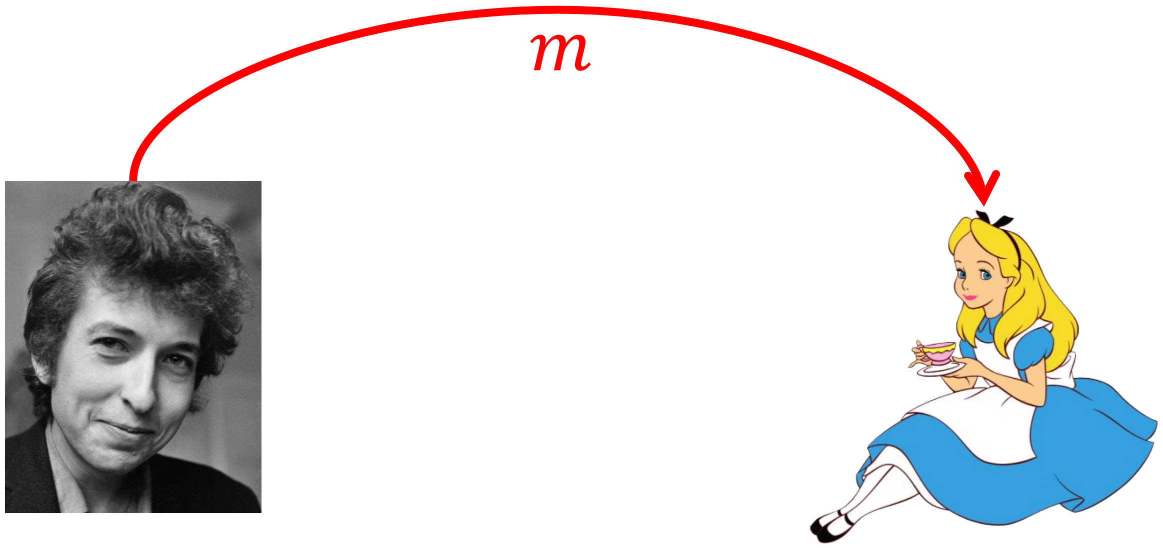
Sam Morse

- 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

$$\ell_c = 4$$

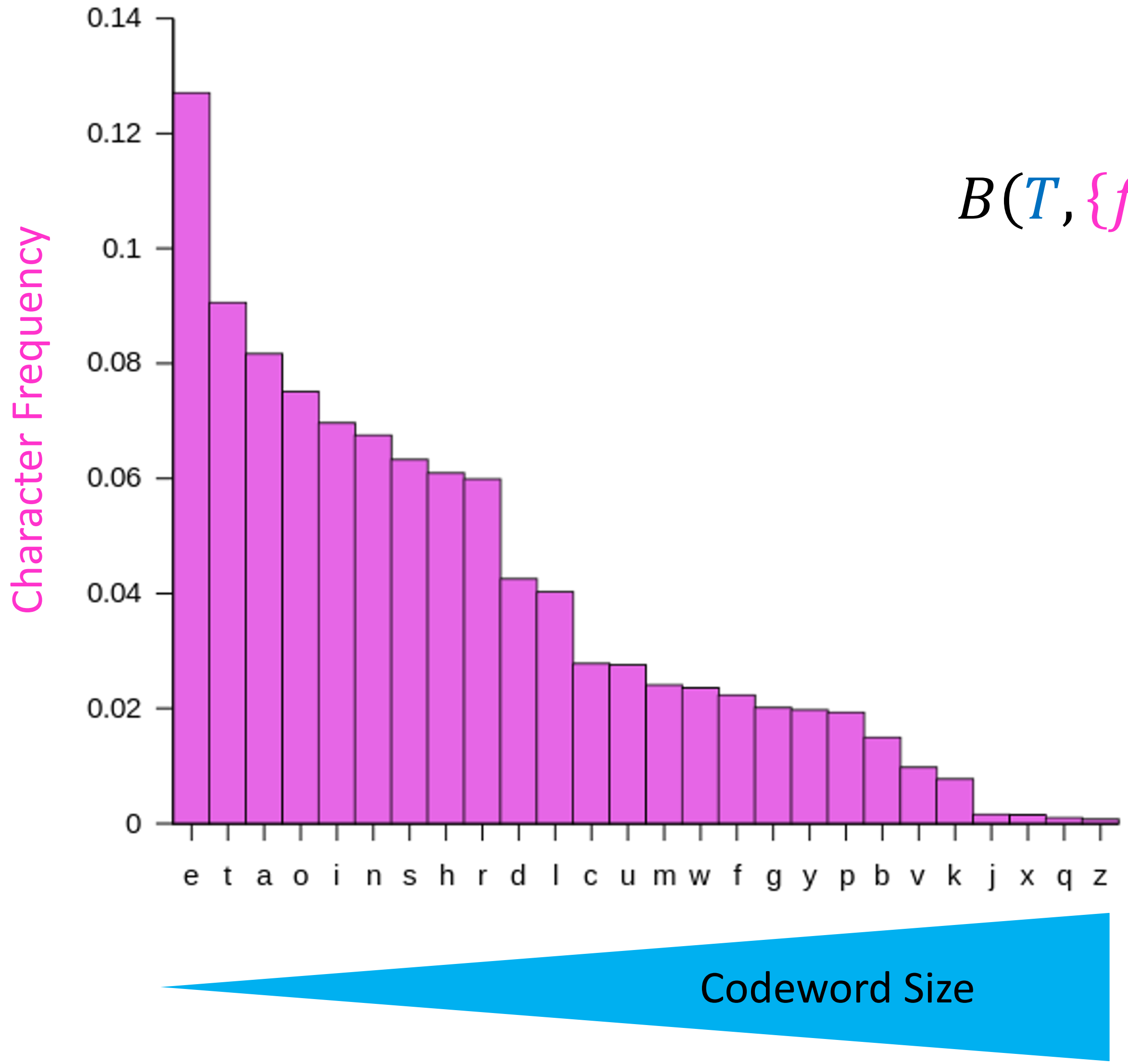
Cost of encoding:

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

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

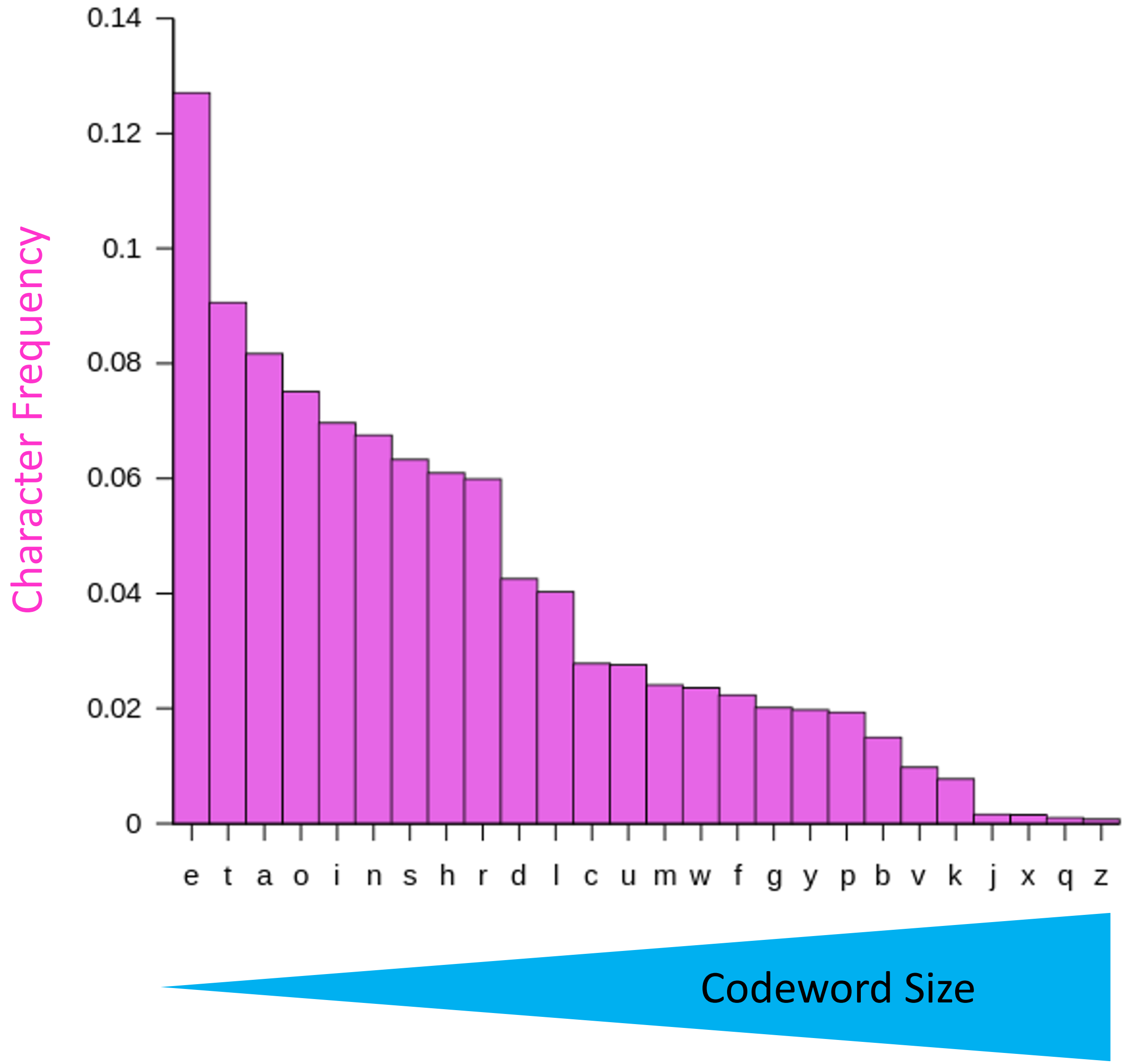


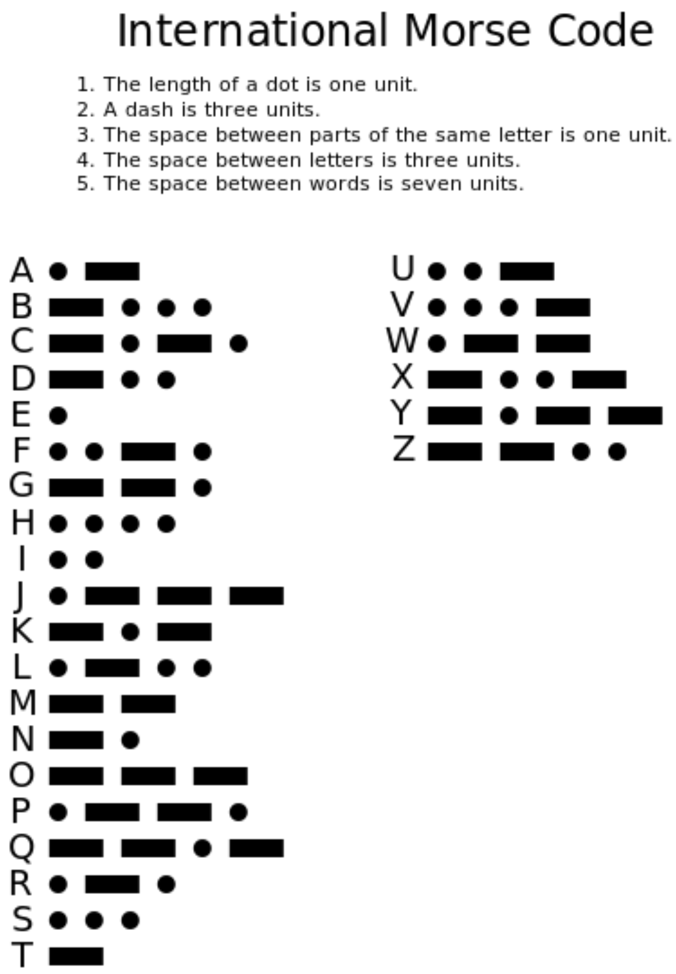
$$B(T, \{f_c\}) = \sum_{\text{character } c} \ell_c f_c$$

When this is big

Make this small

Morse Code

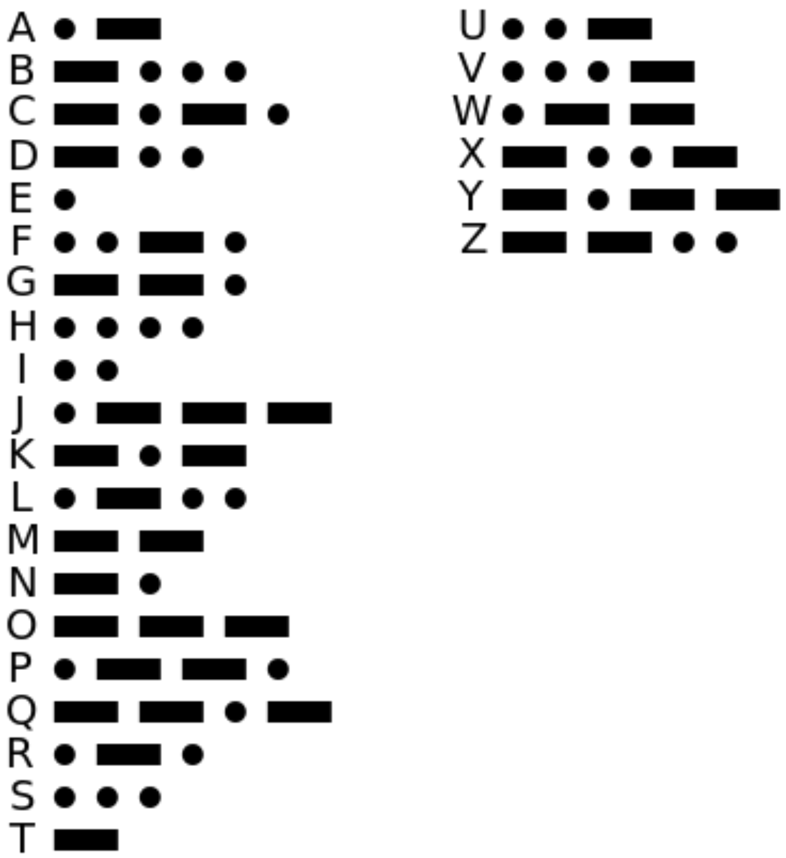

Character Frequency
0.14
0.12
0.1
0.08
0.06
0.04
0.02
0
e t a o i n s h r d l c u m w f g y p b v k j x q z
Codeword Size


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

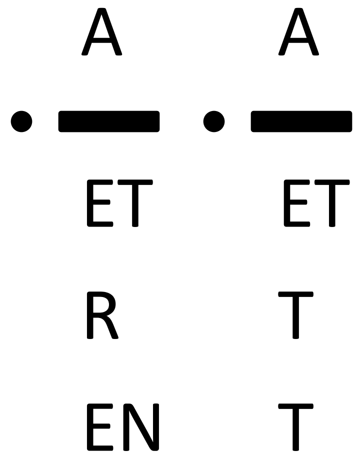
Problem with Morse Code

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.



Decode:



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
…	…

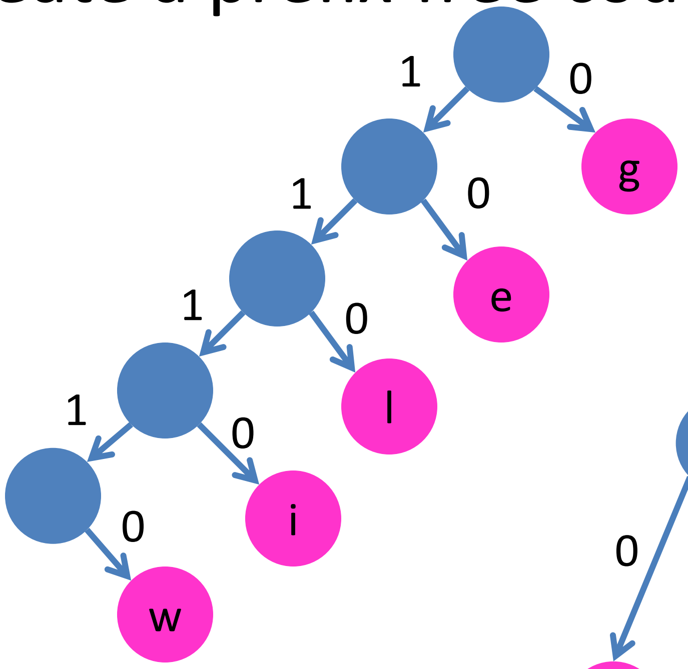
11110111000011010

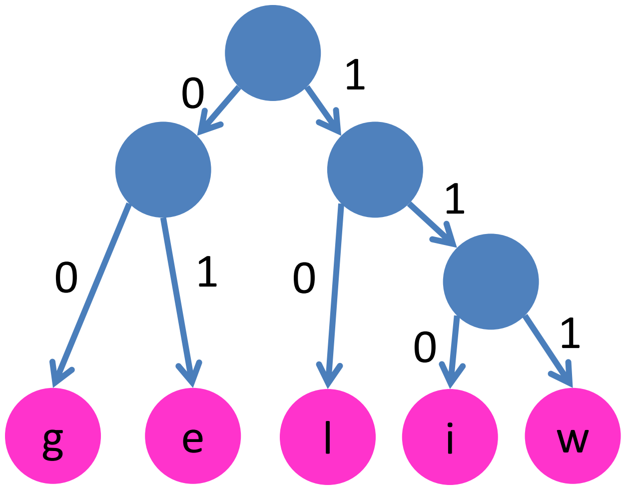
w i gg l e

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

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

Huffman Coding!!

Greedy Algorithms

- Require Optimal Substructure
  - Solution to larger problem contains the solution to a smaller one
  - Only one subproblem to consider!
- Idea:
  1. Identify a greedy choice property
     - How to make a choice guaranteed to be included in some optimal solution
  2. Repeatedly apply the choice property until no subproblems remain

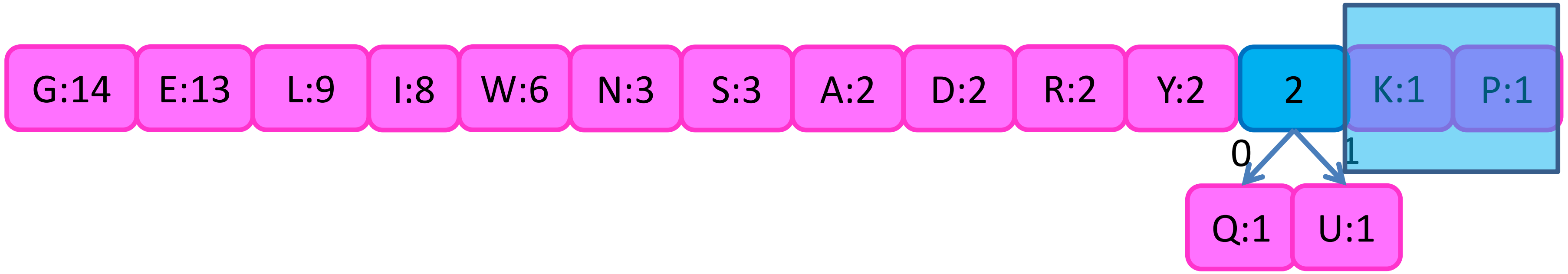
Huffman Algorithm

- 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

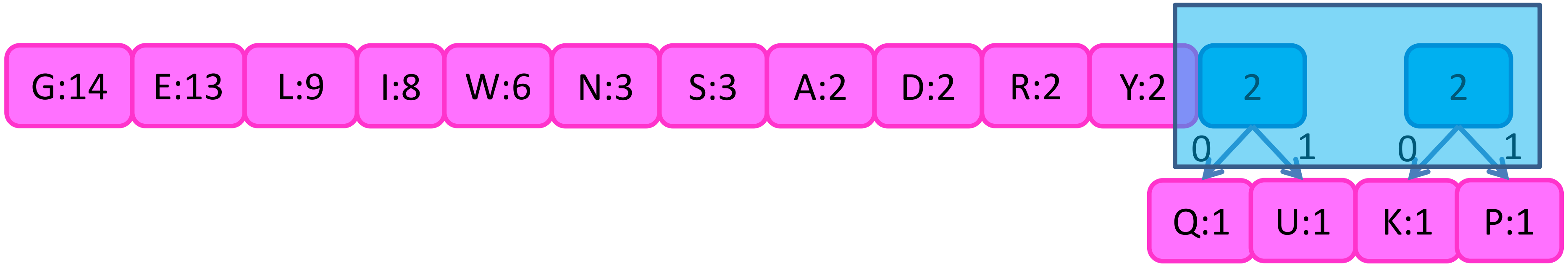
- Choose the least frequent pair, combine into a subtree



Subproblem of size $n - 1$!

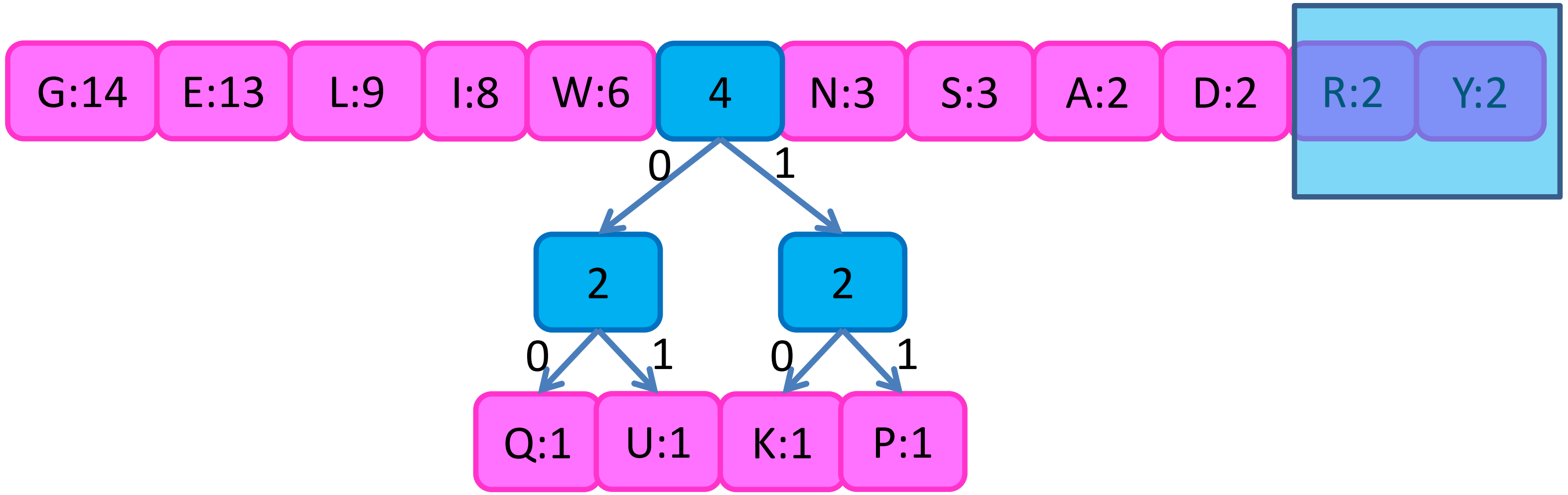
Huffman Algorithm

- Choose the least frequent pair, combine into a subtree



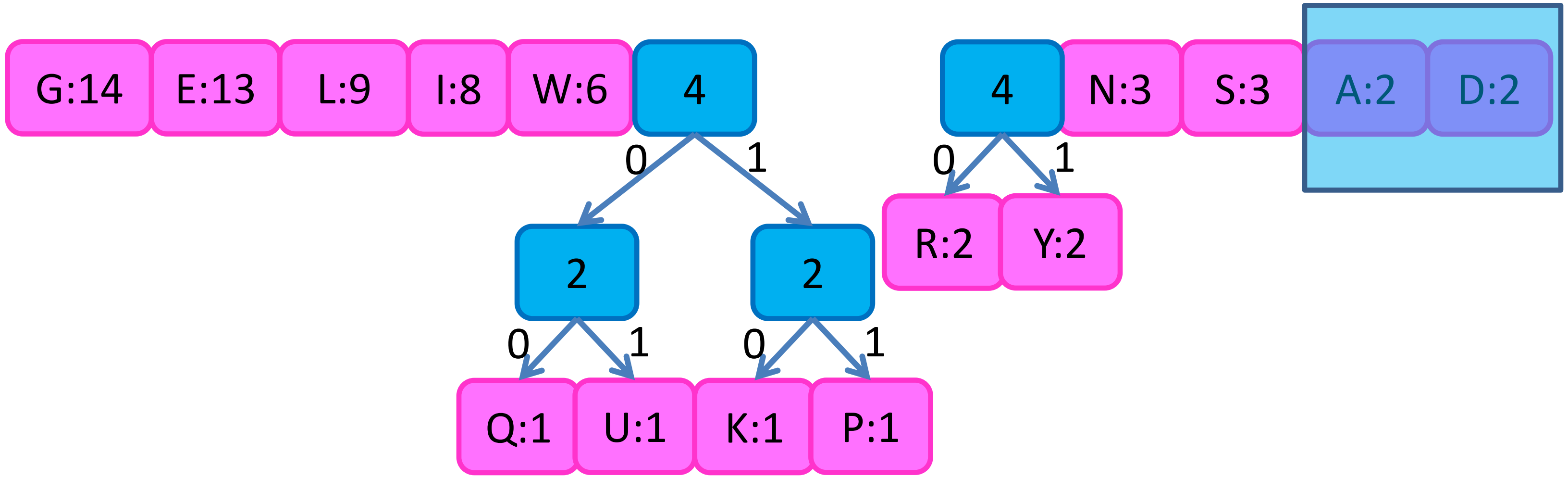
Huffman Algorithm

- Choose the least frequent pair, combine into a subtree



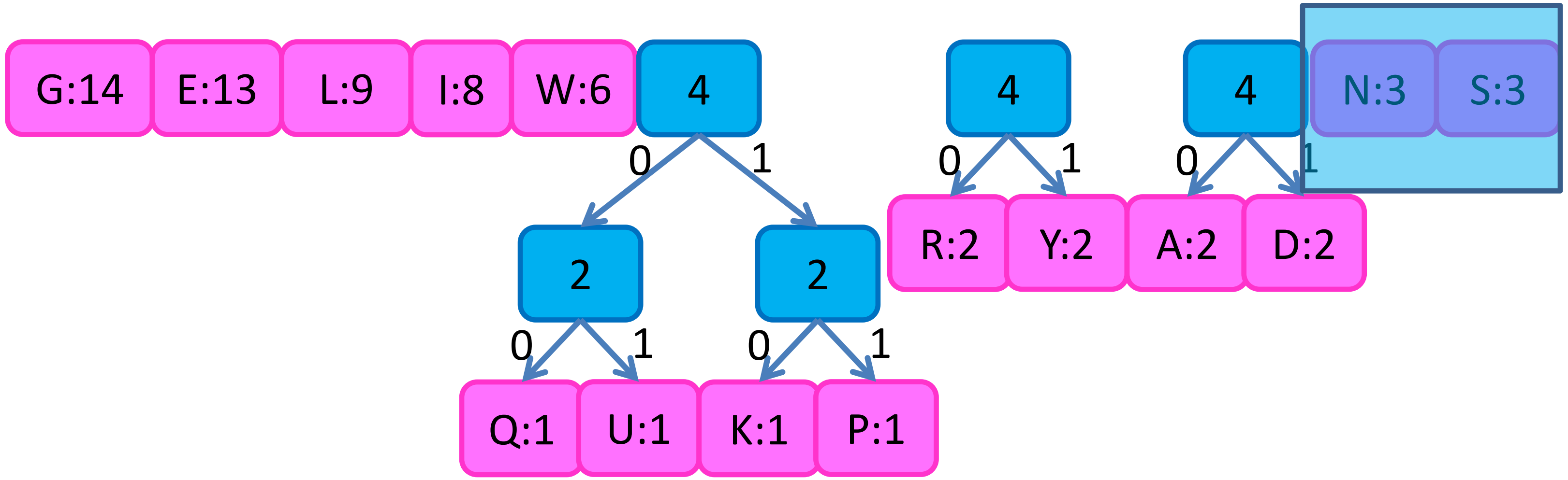
Huffman Algorithm

- Choose the least frequent pair, combine into a subtree



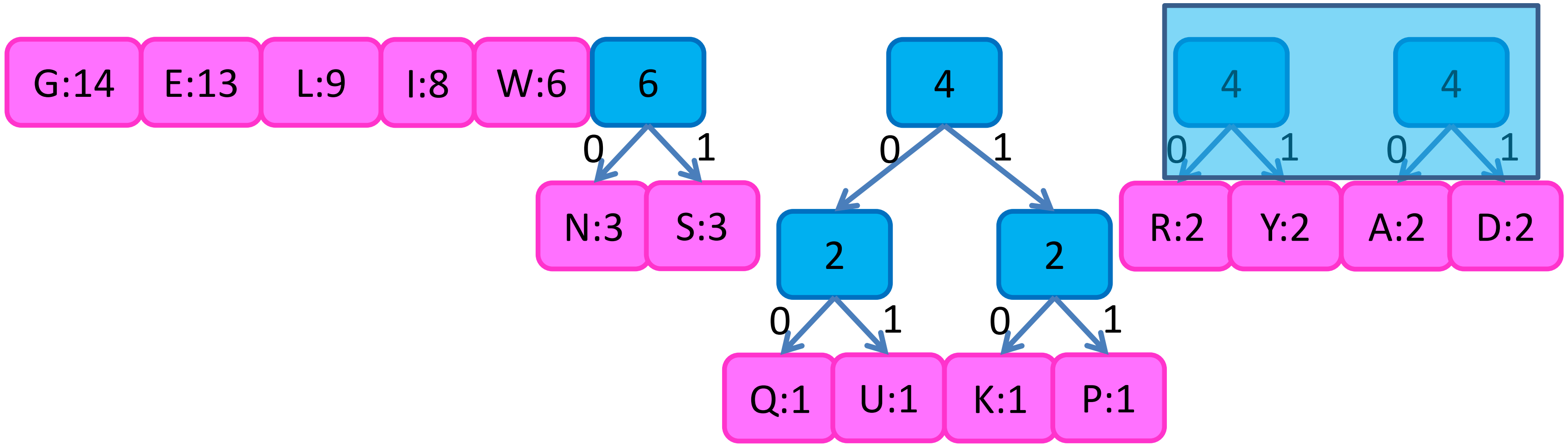
Huffman Algorithm

- Choose the least frequent pair, combine into a subtree



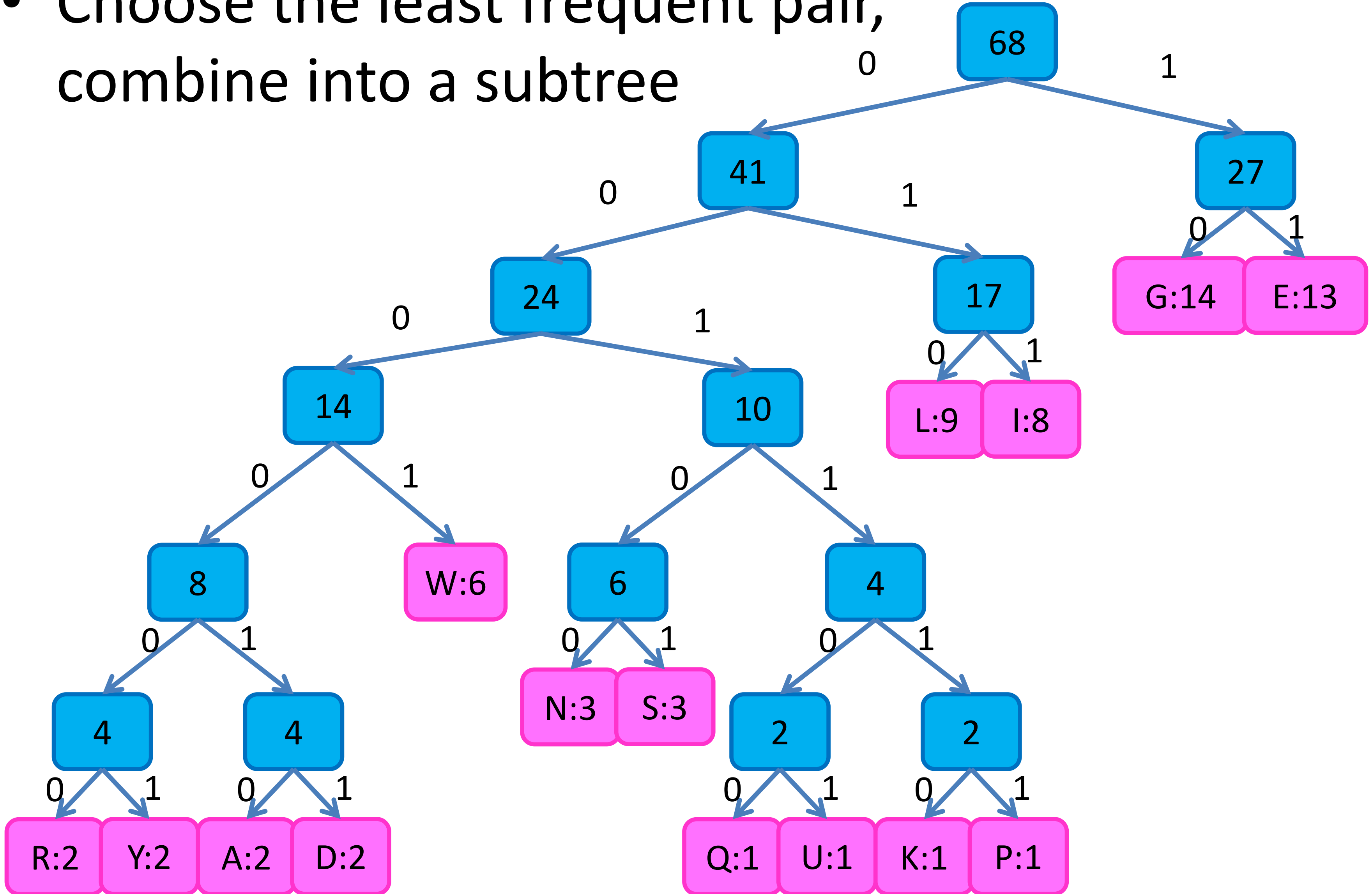
Huffman Algorithm

- Choose the least frequent pair, combine into a subtree



Huffman Algorithm

- Choose the least frequent pair, combine into a subtree



Exchange argument

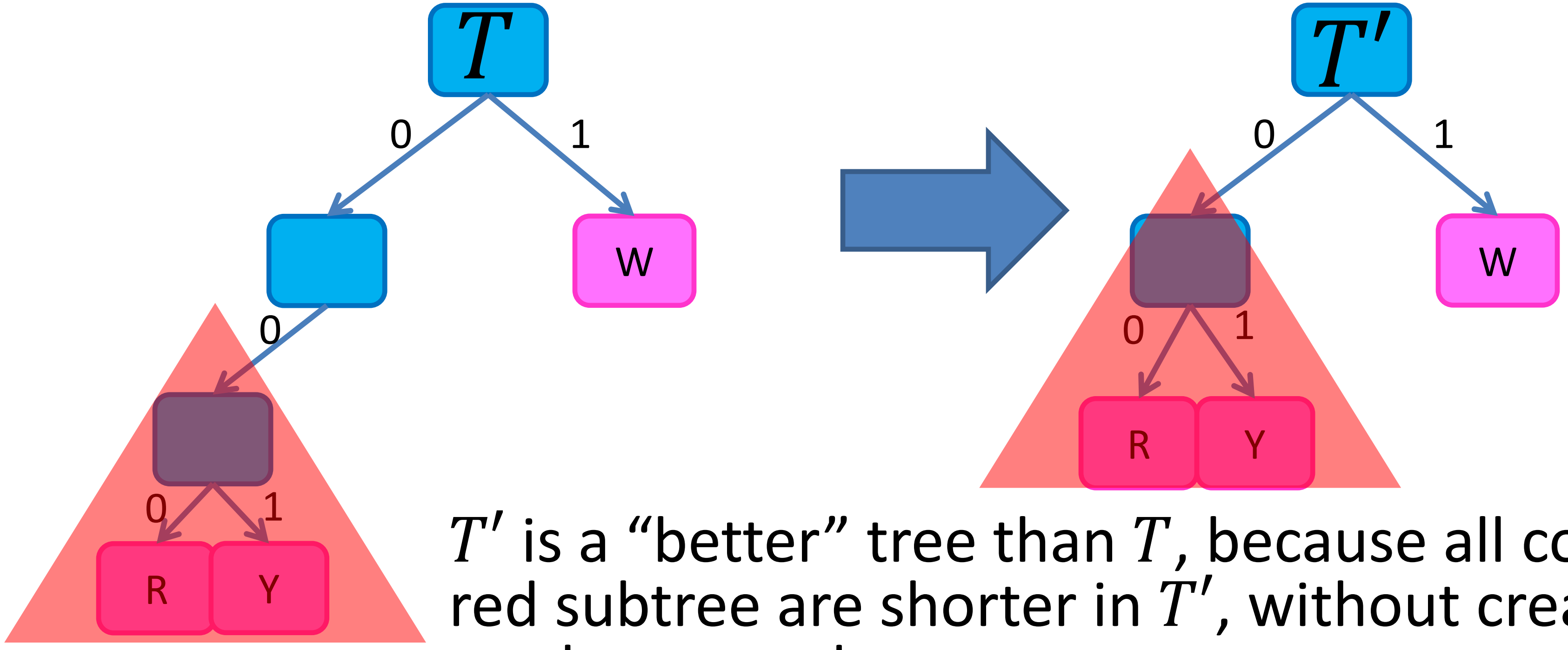
- Shows correctness of a greedy algorithm
- Idea:
  - Show exchanging an item from an arbitrary optimal solution with your greedy choice makes the new solution no worse
  - How to show my sandwich is at least as good as yours:
    - Show: “I can remove any item from your sandwich, and it would be no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

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 <u>results in</u> **an** optimal solution
    - Proof by contradiction

Showing Huffman is Optimal

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

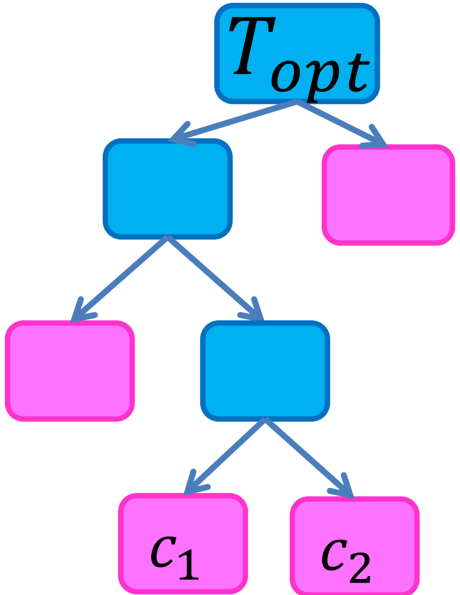


T' is a "better" tree than T, because all codes in red subtree are shorter in T', without creating any longer codes

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

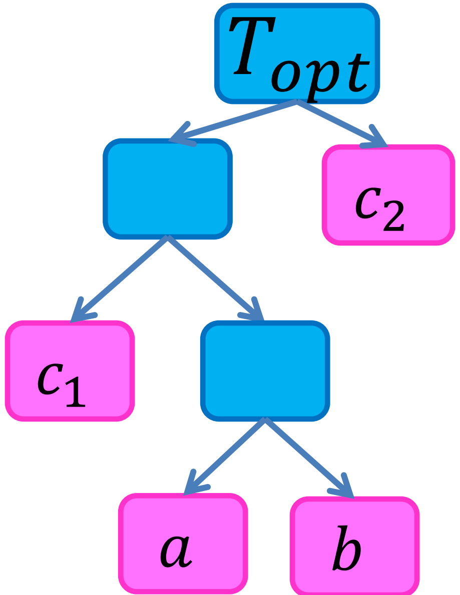
Case 1: Consider some optimal tree T_{opt}. If c_1, c_2 are siblings in this tree, then claim holds



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}

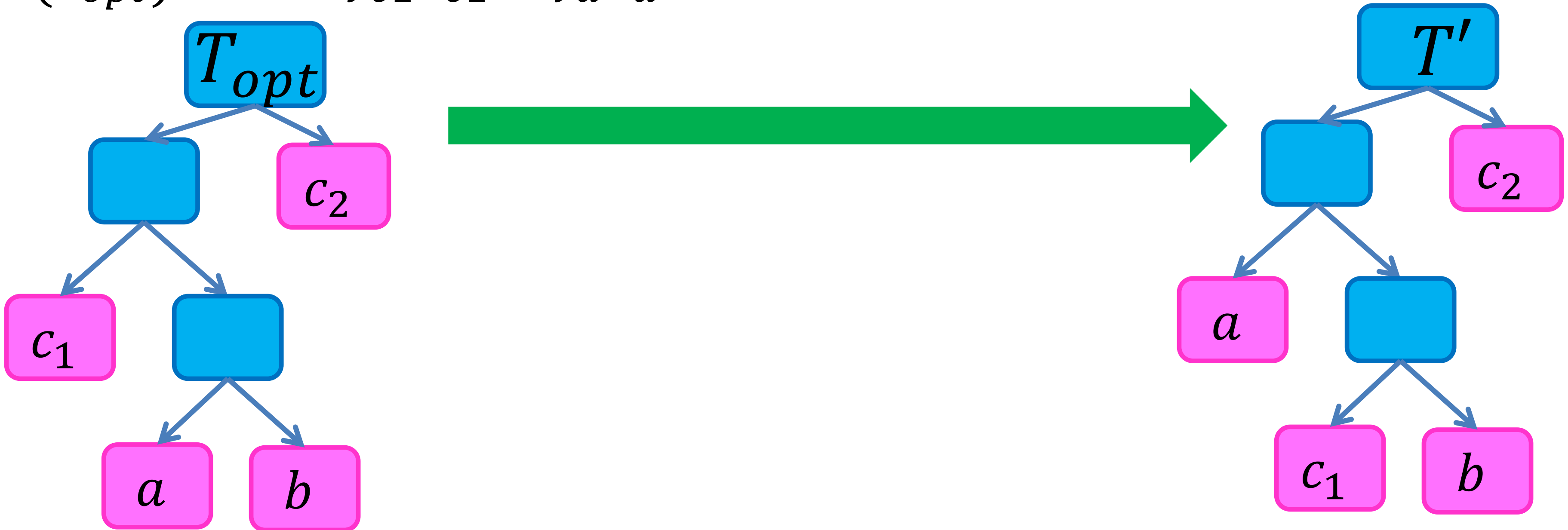
- Claim: the least-frequent characters (c_1, c_2), are siblings in some optimal tree

a, b = lowest-depth siblings

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

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

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



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

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

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

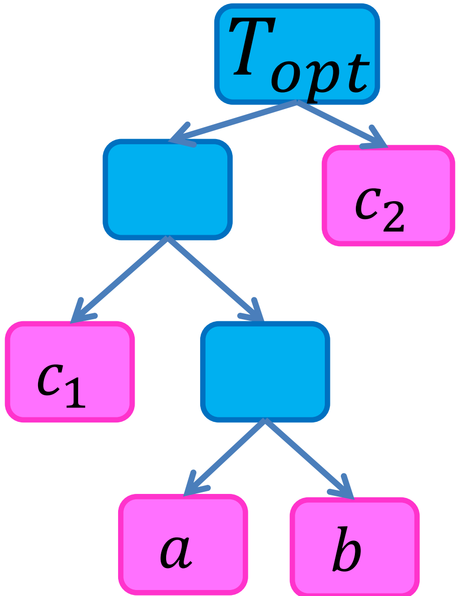
- Claim: the least-frequent characters (c_1, c_2), are siblings in some optimal tree

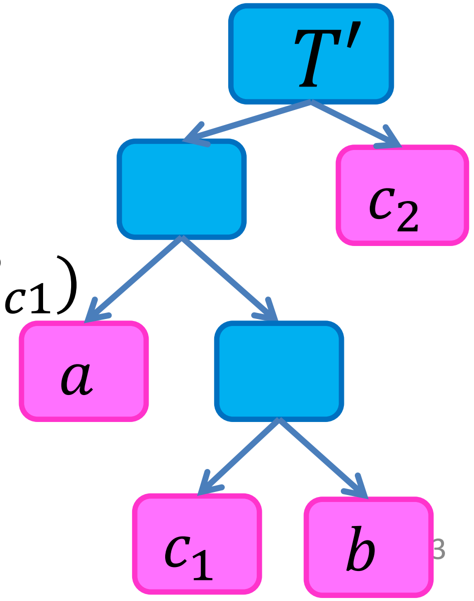
a, b = lowest-depth siblings

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

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

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





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

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

T' is also optimal!

Case 2:Repeat to swap c_2, b!

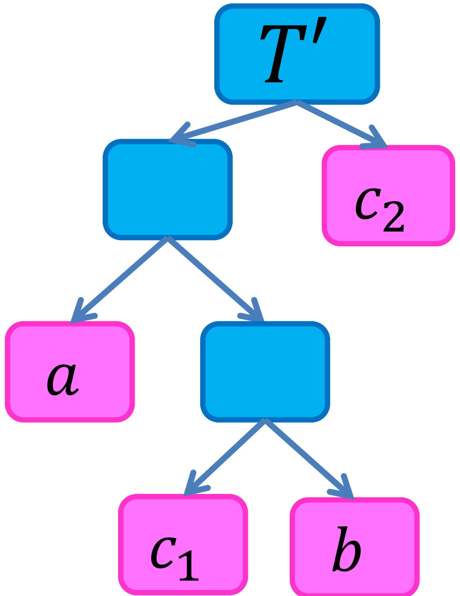
- Claim: the least-frequent characters (c_1, c_2), are siblings in some optimal tree

$a, b =$ lowest-depth siblings

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

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

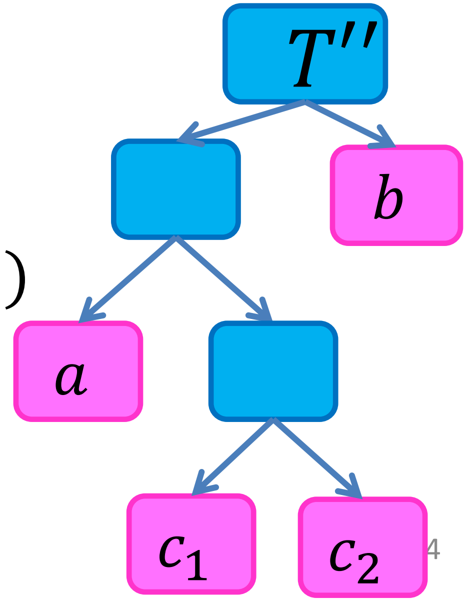
$$B(T'') = C + f_{c2}\ell_b + f_b\ell_{c2}$$



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

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

T'' is also optimal! Claim holds!



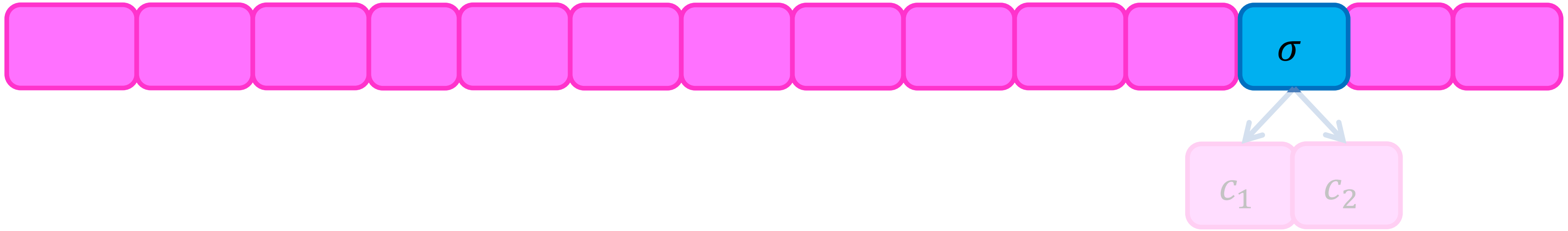
Showing Huffman is Optimal

- 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
    - Proof by contradiction

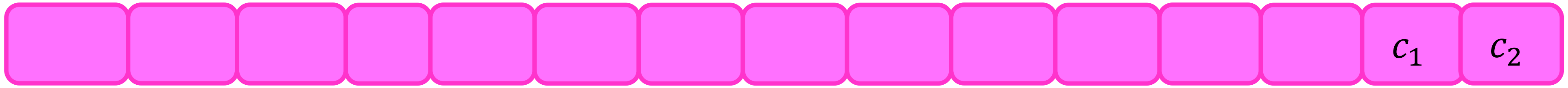
Finishing the Proof

- Show Optimal Substructure
  - Show treating c_1, c_2 as a new "combined" character gives optimal solution

Why does solving this smaller problem:

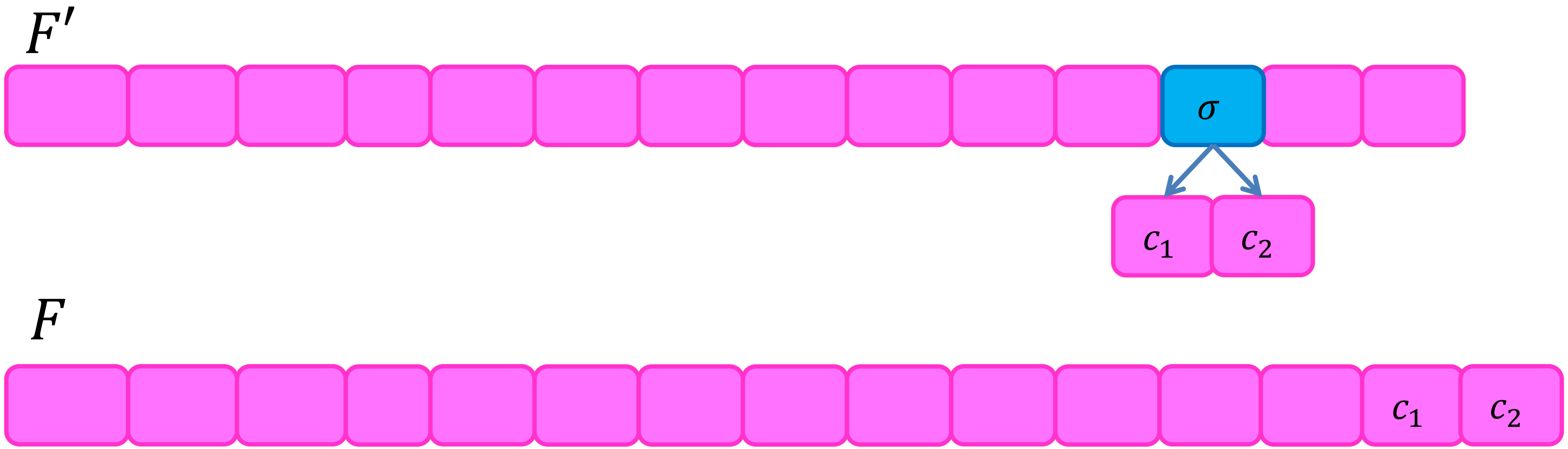


Give an optimal solution to this?:



Optimal Substructure

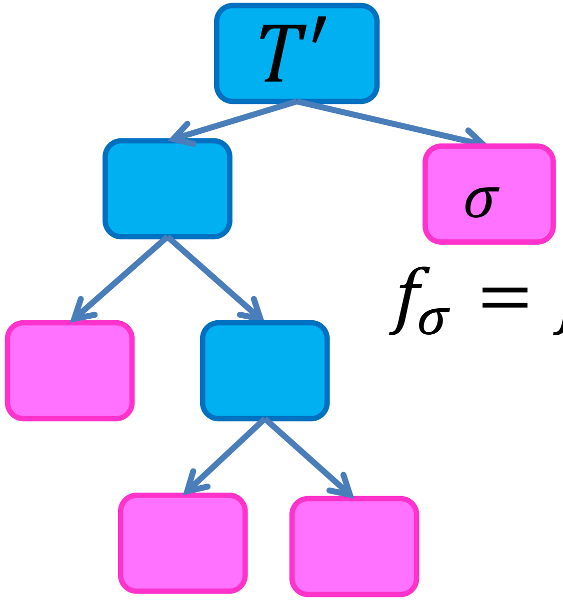
- Claim: An optimal solution for F involves finding an optimal solution for F', then adding c_1, c_2 as children to σ



Optimal Substructure

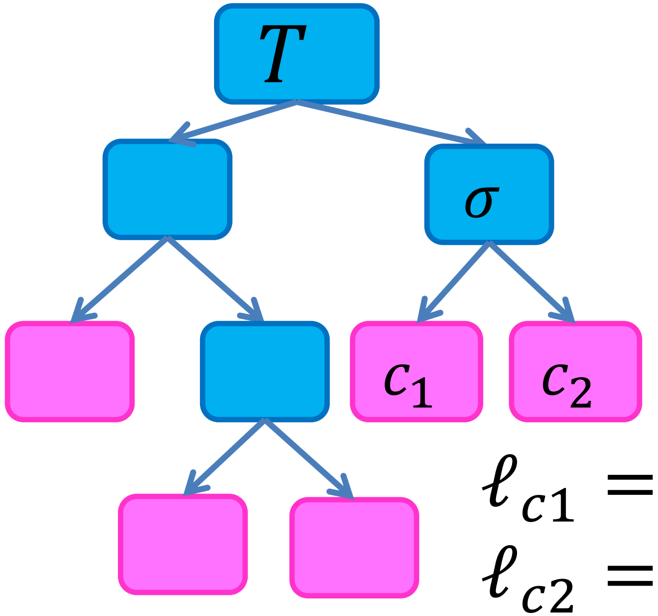
- Claim: An optimal solution for F involves finding an optimal solution for F', then adding c_1, c_2 as children to σ

If this is optimal



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

Then this is optimal



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

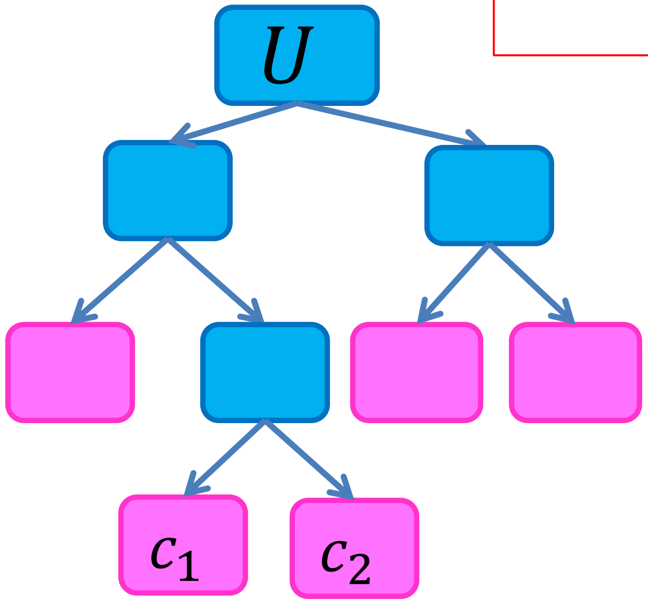
$$B(T') = B(T) - f_{c1} - f_{c2}$$

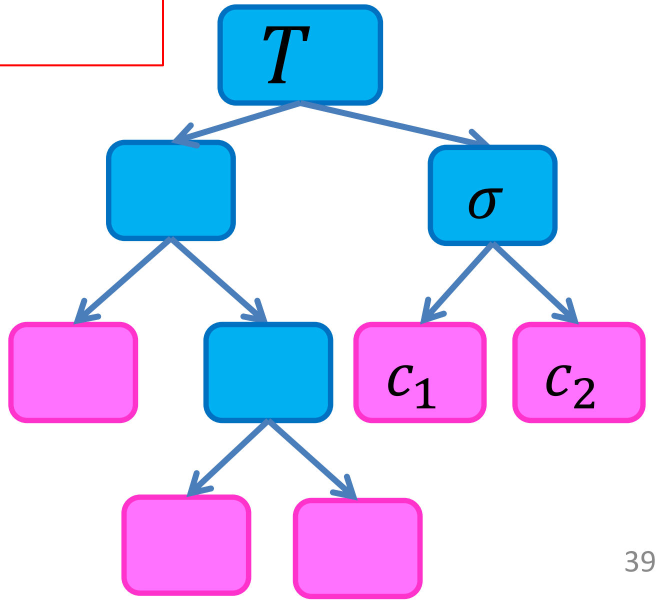
Optimal Substructure

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

Toward contradiction

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



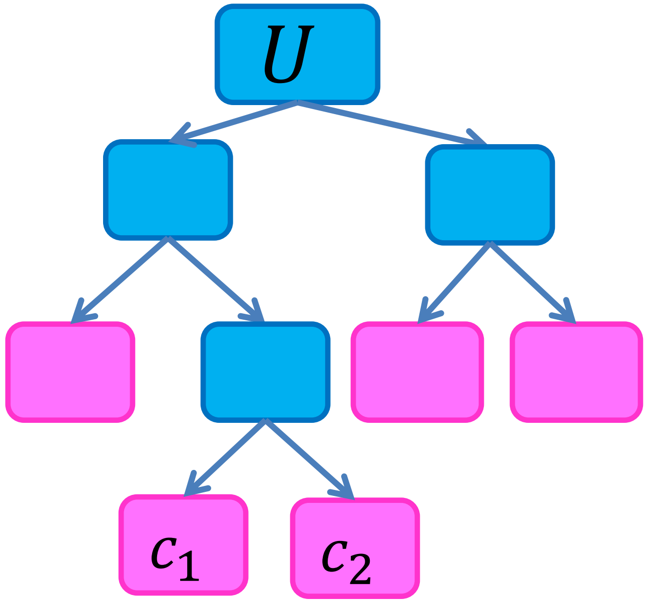


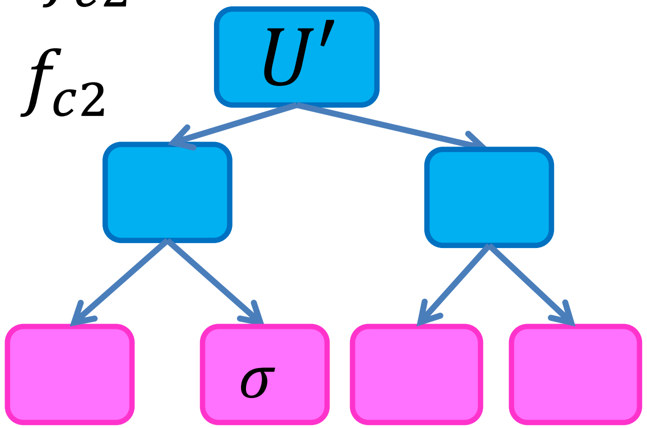
Optimal Substructure

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

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

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

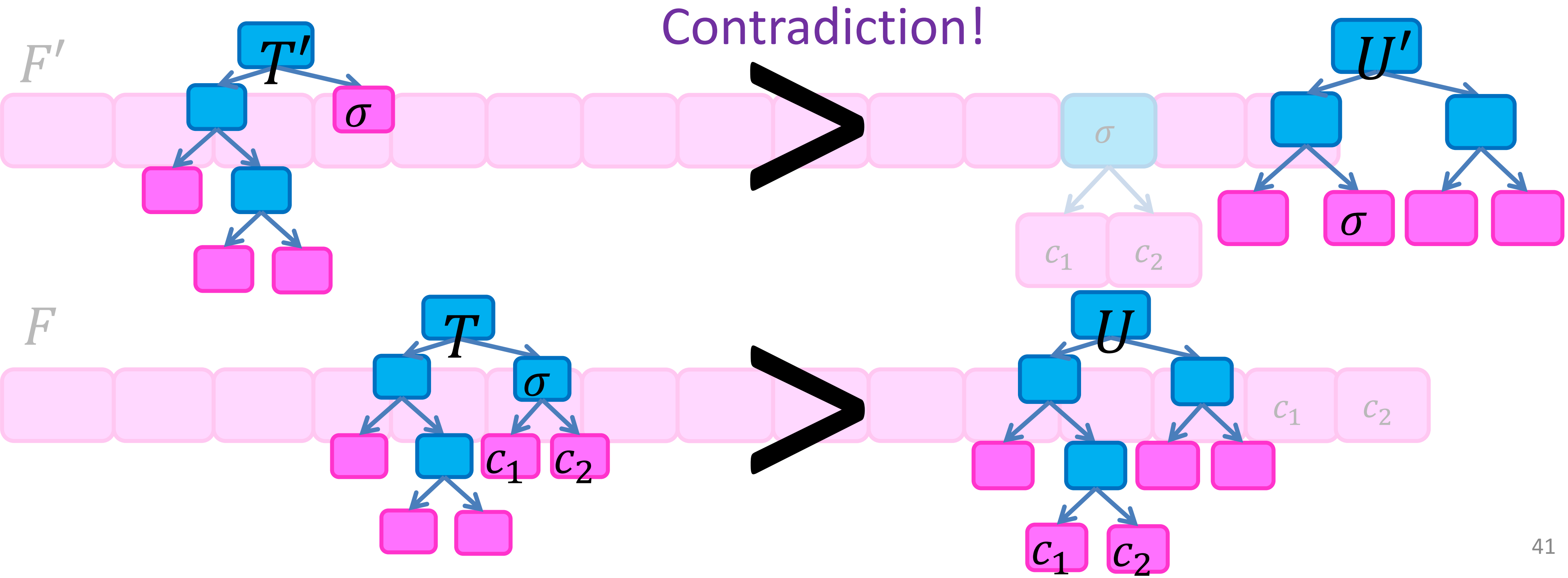




Contradicts optimality of T', so T is optimal!

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



Mental Stretch

Why is an algorithm's space complexity (how much memory it uses) important?

Why might a memory-intensive algorithm be a "bad" one?

Why lots of memory is “bad”

Caching Problem

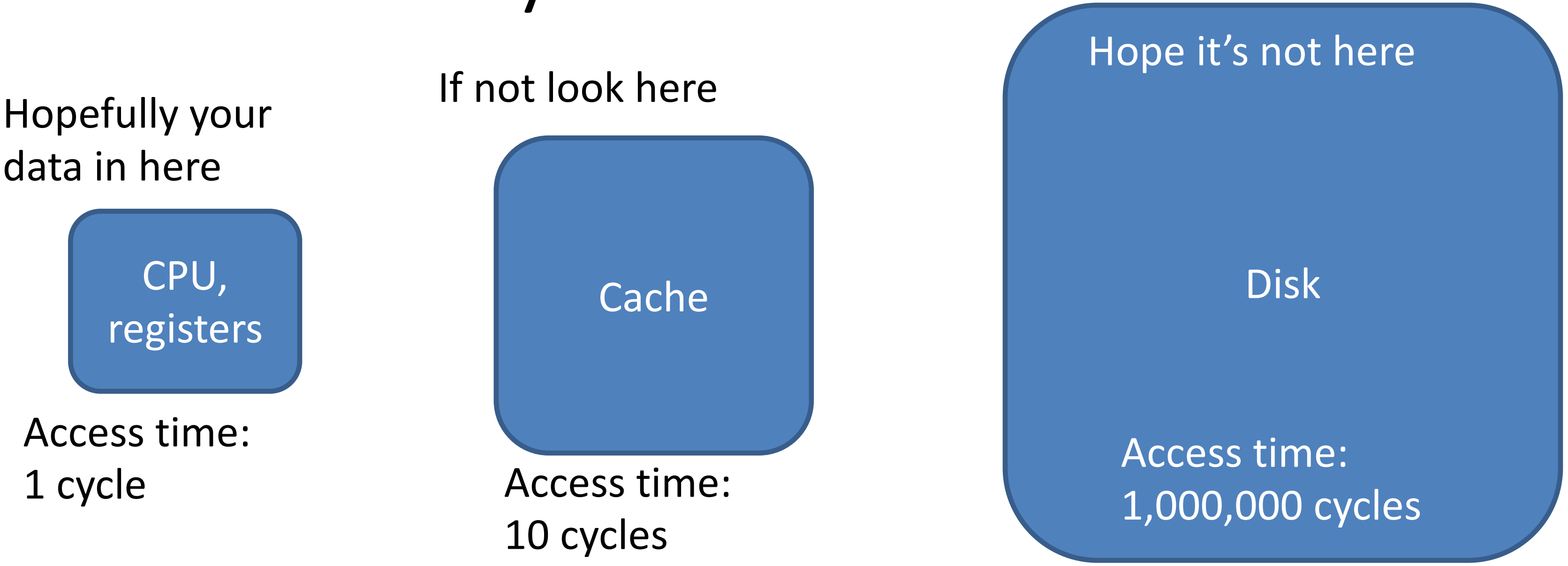
- Why is using too much memory a bad thing?

Von Neumann Bottleneck

- Named for John von Neumann
- Inventor of modern computer architecture
- Other notable influences include:
  - Mathematics
  - Physics
  - Economics
  - Computer Science

Von Neumann Bottleneck

- Reading from memory is VERY slow
- Big memory = slow memory
- Solution: hierarchical memory
- Takeaway for Algorithms: Memory is time, more memory is a lot more time



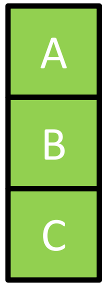
Caching Problem

- Cache misses are very expensive
- When we load something new into cache, we must eliminate something already there
- We want the best cache “schedule” to minimize the number of misses

Caching Problem Definition

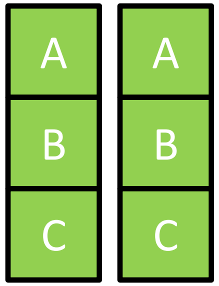
- Input:
  - k = size of the cache
  - $M = [m_1, m_2, \dots m_n]$ = memory access pattern
- Output:
  - "schedule" for the cache (list of items in the cache at each time) which minimizes cache fetches

Example



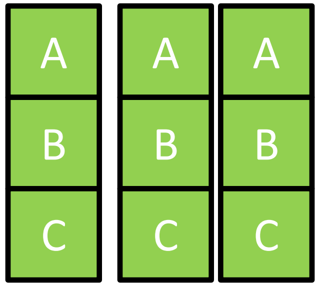
A B C D A D E A D B A E C E A

Example



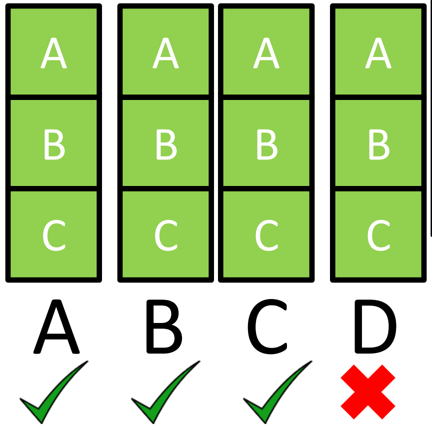
A B C D A D E A D B A E C E A

Example



A B C D A D E A D B A E C E A

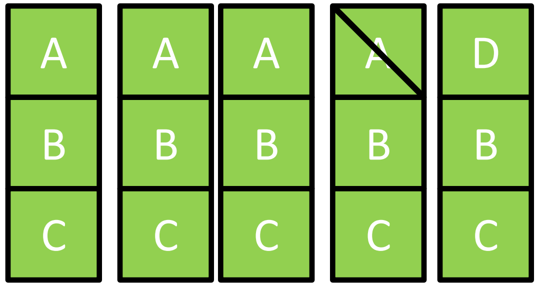
Example



We must evict something to make room for D

A B C D A D E A D B A E C E A

Example

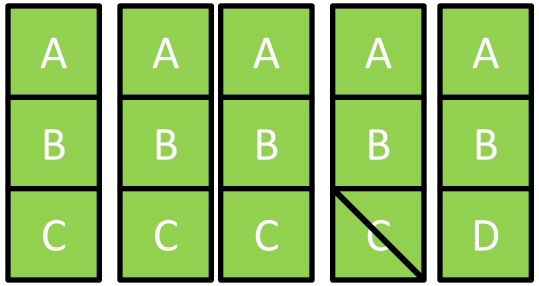


If we evict A

A B C D A D E A D B A E C E A

Example

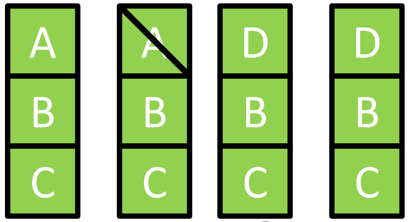
If we evict C



A B C D A D E A D B A E C E A

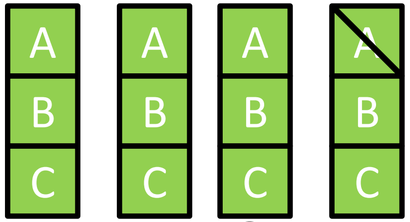
Our Problem vs Reality

- Assuming we know the entire access pattern
- Cache is Fully Associative
- Counting # of fetches (not necessarily misses)
- “Reduced” Schedule: Address only loaded on the cycle it’s required
  - Reduced == Unreduced (by number of misses)



Unreduced

A B C D A D E A D B A E C E A



Reduced

Leaving A in longer does not save fetches

A B C D A D E A D B A E C E A

Greedy Algorithms

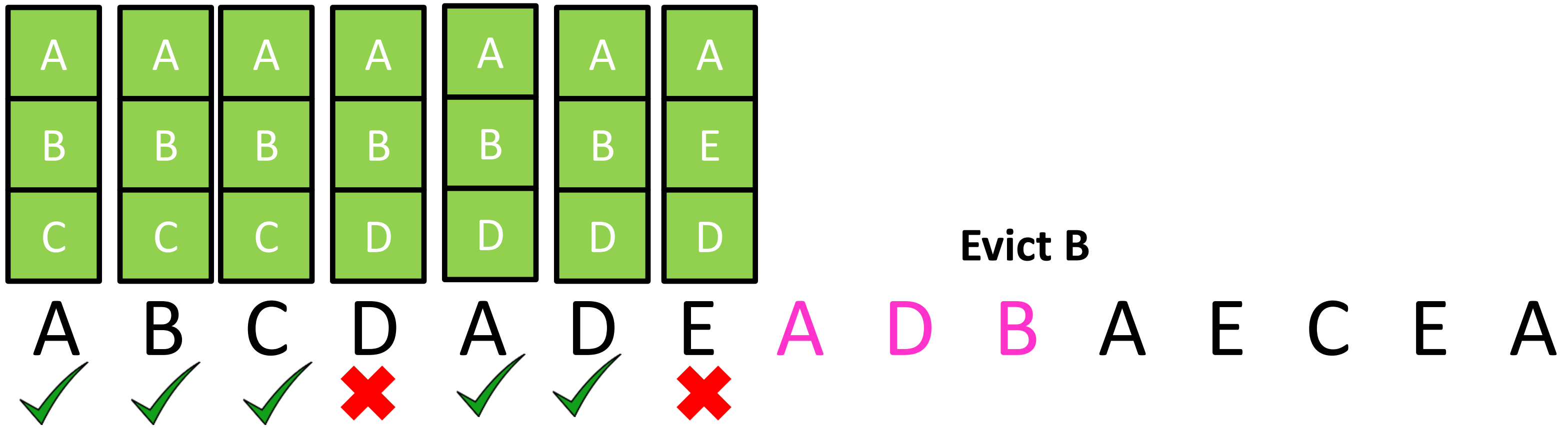
- Require Optimal Substructure
  - Solution to larger problem contains the solution to a smaller one
  - Only one subproblem to consider!
- Idea:
  1. Identify a greedy choice property
     - How to make a choice guaranteed to be included in some optimal solution
  2. Repeatedly apply the choice property until no subproblems remain

Greedy choice property

- Belady evict rule:
  - Evict the item accessed farthest in the future

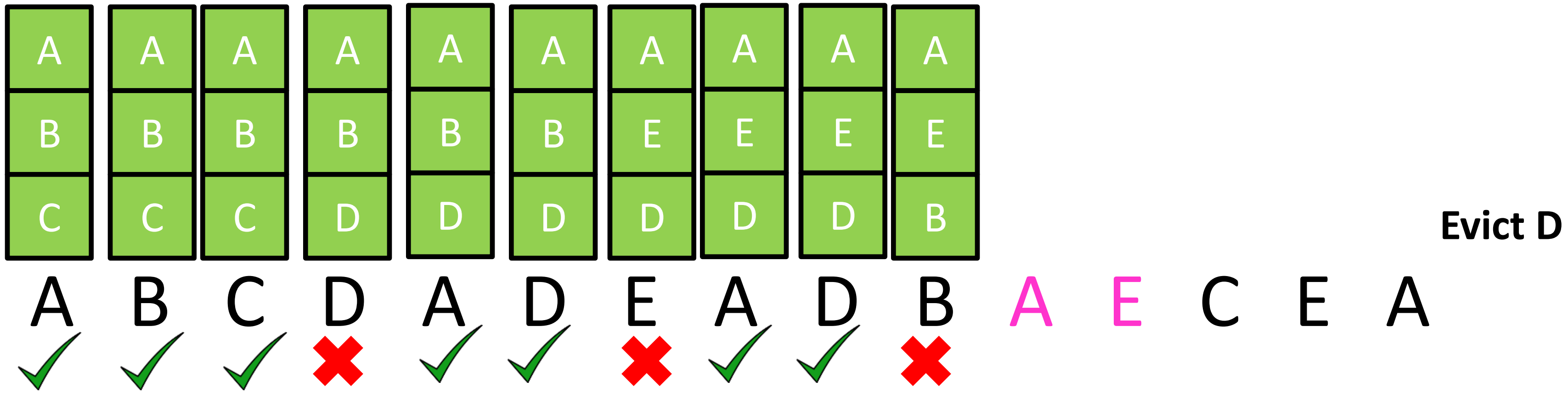
Greedy choice property

- Belady evict rule:
  - Evict the item accessed farthest in the future



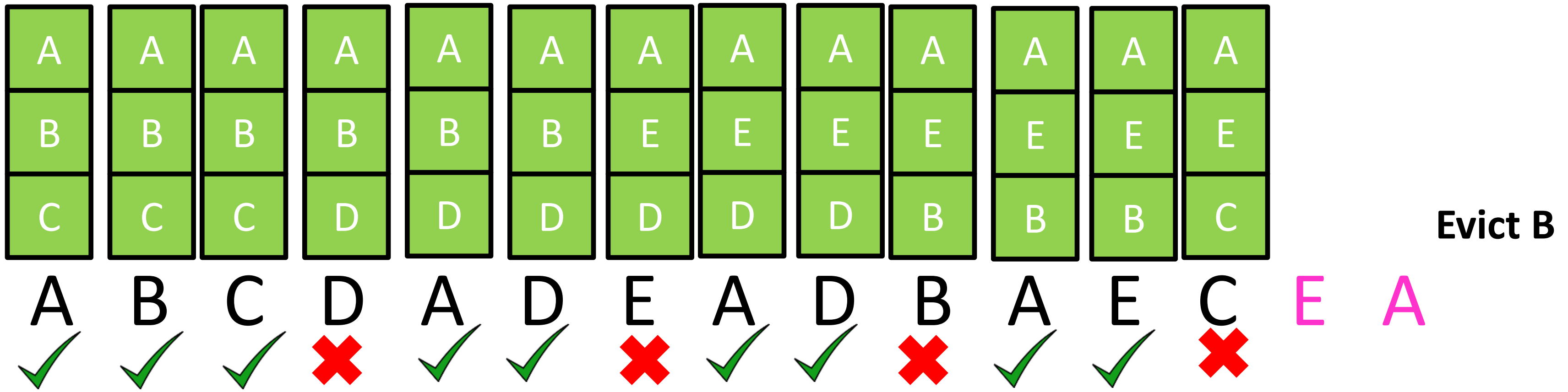
Greedy choice property

- Belady evict rule:
  - Evict the item accessed farthest in the future



Greedy choice property

- Belady evict rule:
  - Evict the item accessed farthest in the future



Greedy choice property

- Belady evict rule:
  - – Evict the item accessed farthest in the future

A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
B	B	B	B	B	B	E	E	E	E	E	E	E	E	E
C	C	C	D	D	D	D	D	D	B	B	B	C	C	C
A	B	C	D	A	D	E	A	D	B	A	E	C	E	A
✓	✓	✓	✗	✓	✓	✗	✓	✓	✗	✓	✓	✗	✓	✓

4 Cache Misses

Greedy Algorithms

- Require Optimal Substructure
  – Solution to larger problem contains the solution to a smaller one
  – Only one subproblem to consider!
- Idea:
  1. Identify a greedy choice property
     - How to make a choice guaranteed to be included in some optimal solution
  2. Repeatedly apply the choice property until no subproblems remain

Caching Greedy Algorithm

Initialize $cache$= first k accesses $O(k)$

For each $m_i \in M$: n times

if $m_i \in cache$: $O(k)$

print $cache$ $O(k)$

else:

$m =$ furthest-in-future from cache $O(kn)$

evict m, load m_i $O(1)$

print $cache$ $O(k)$

$O(kn^2)$

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”

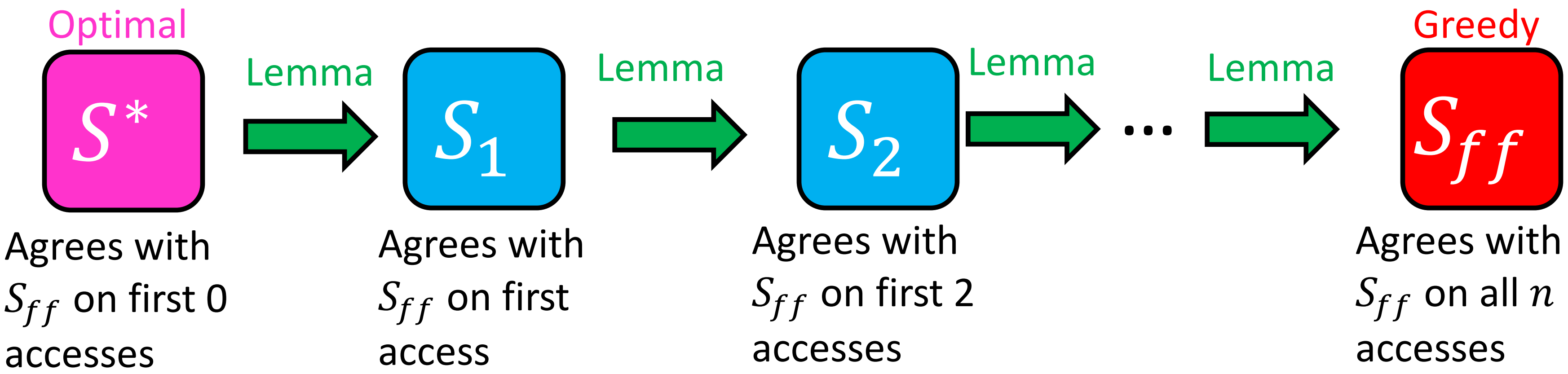
Belady Exchange Lemma

Let S_{ff} be the schedule chosen by our greedy algorithm

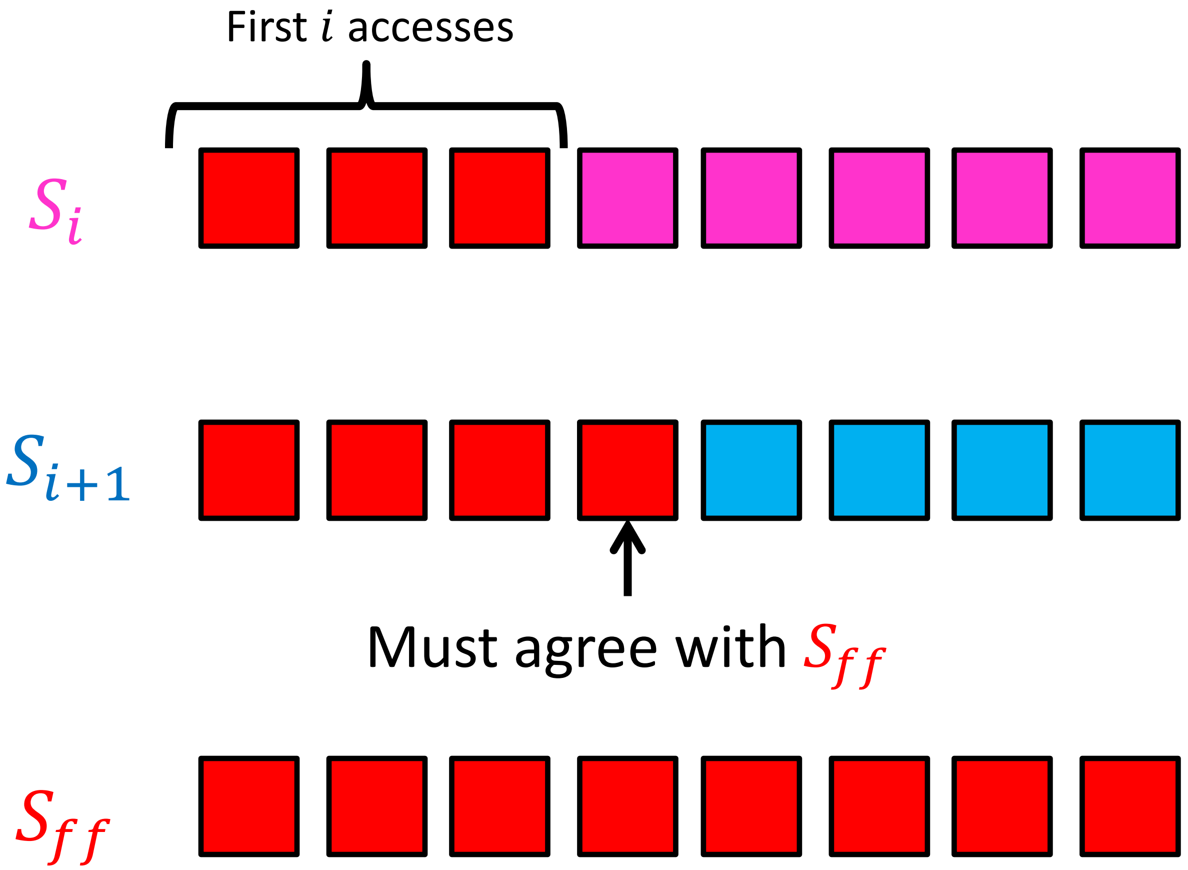
Let S_i be a schedule which agrees with S_{ff} for the first i memory accesses.

We will show: there is a schedule S_{i+1} which agrees with S_{ff} for the first $i + 1$ memory accesses, and has no more misses than S_i

(i.e. $misses(S_{i+1}) \leq misses(S_i)$)



Belady Exchange Proof Idea



Need to fill in the rest of S_{i+1} to have no more misses than S_i

Proof of Lemma

Goal: find S_{i+1} s.t. $misses(S_{i+1}) \leq misses(S_i)$

Since S_i agrees with S_{ff} for the first i accesses, the state of the cache at access $i+1$ will be the same

S_i Cache after i	e	f

$=$

S_{ff} Cache after i	e	f

Consider access $m_{i+1} = d$

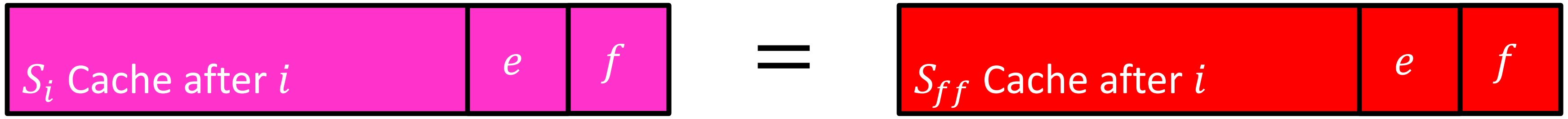
Case 1: if d is in the cache, then neither S_i nor S_{ff} evict from the cache, use the same cache for S_{i+1}

S_{i+1} Cache after i	e	f

Proof of Lemma

Goal: find S_{i+1} s.t. $misses(S_{i+1}) \leq misses(S_i)$

Since S_i agrees with S_{ff} for the first i accesses, the state of the cache at access $i+1$ will be the same



Consider access $m_{i+1} = d$

Case 2: if d isn't in the cache, and both S_i and S_{ff} evict f from the cache, evict f for d in S_{i+1}

Proof of Lemma

Goal: find S_{i+1} s.t. $misses(S_{i+1}) \le misses(S_i)$

Since S_i agrees with S_{ff} for the first i accesses, the state of the cache at access $i+1$ will be the same

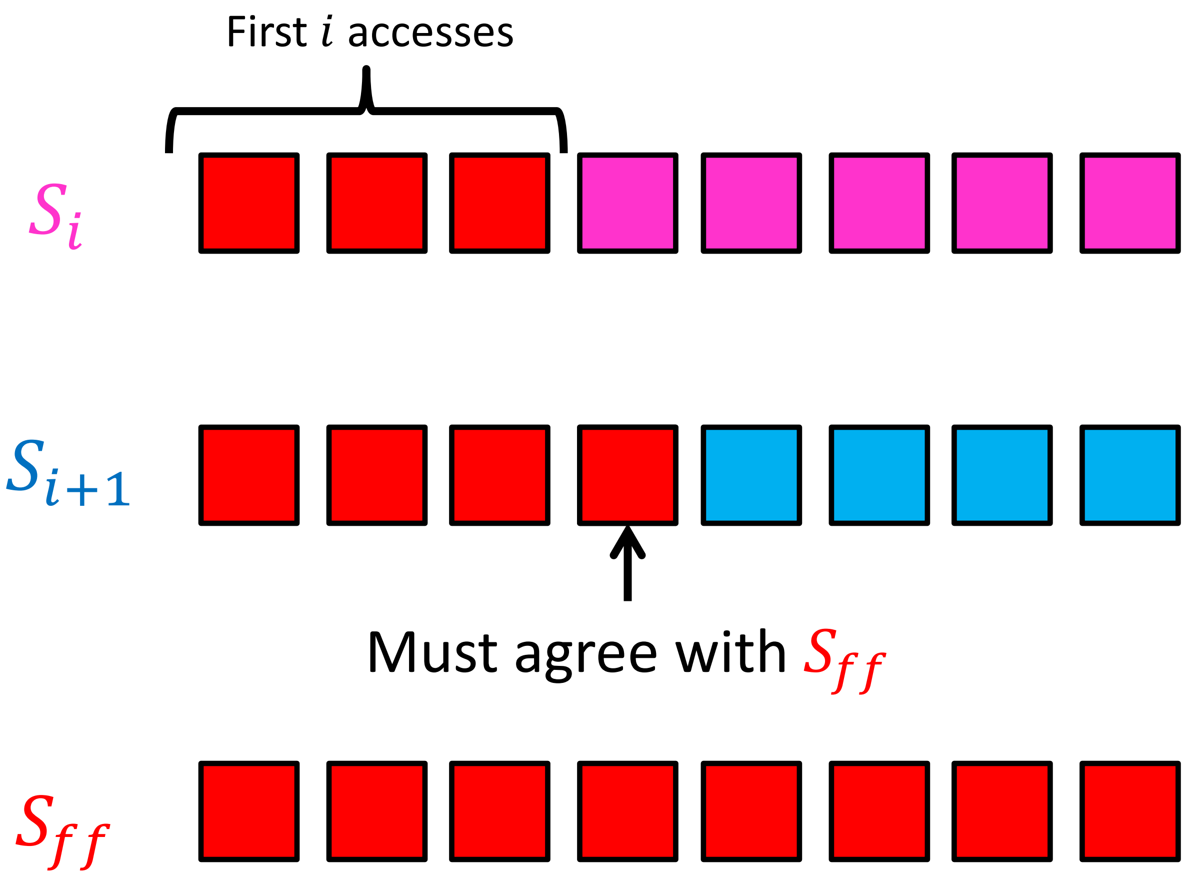
S_i Cache after i	e	f	$=$	S_{ff} Cache after i	e	f

Consider access $m_{i+1} = d$

Case 3: if d isn't in the cache, S_i evicts e and S_{ff} evicts f from the cache

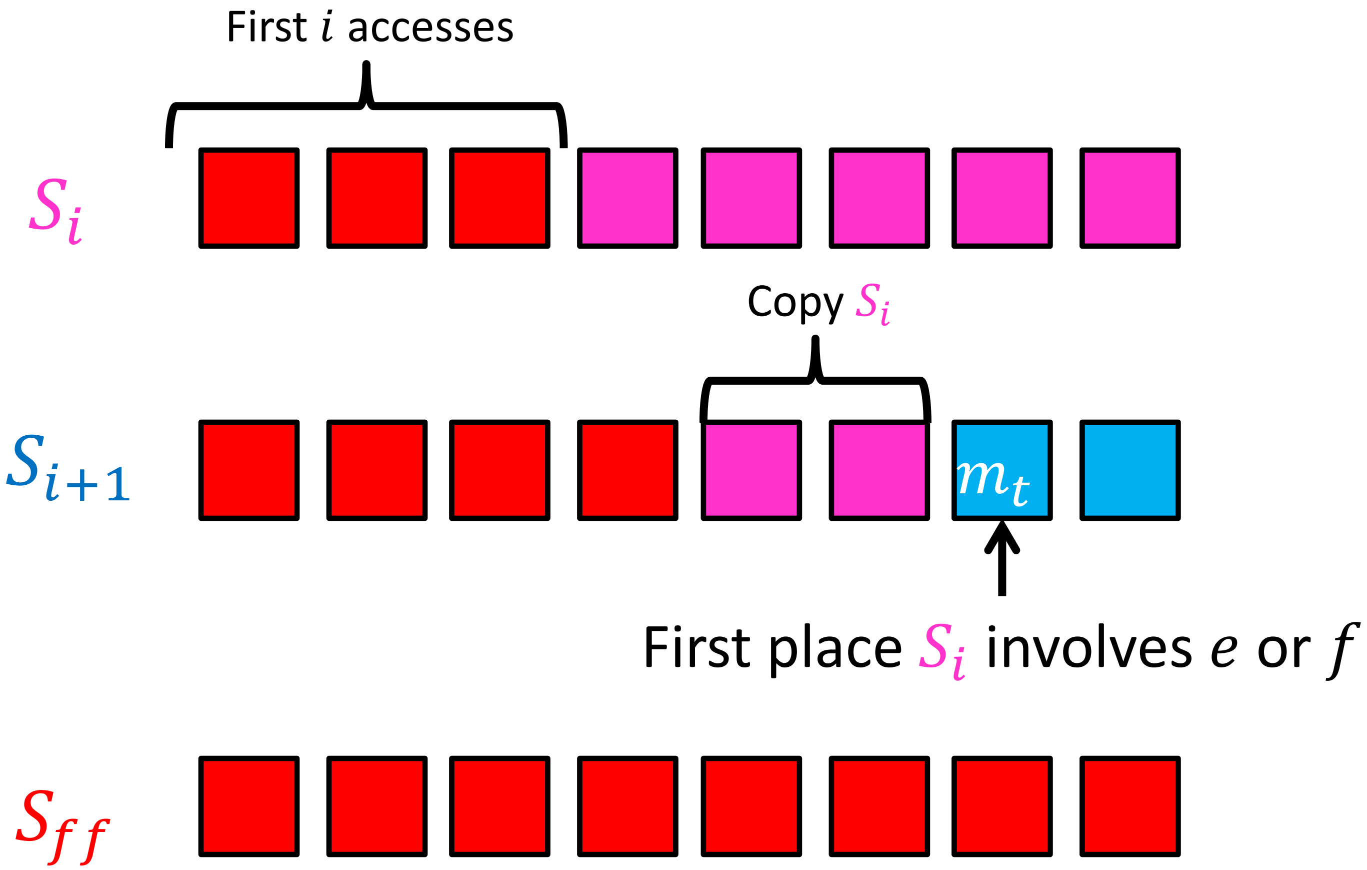
S_i Cache after $i+1$	d	f	$\neq$	S_{ff} Cache after $i+1$	e	d

Case 3



Need to fill in the rest of S_{i+1} to have no more misses than S_i

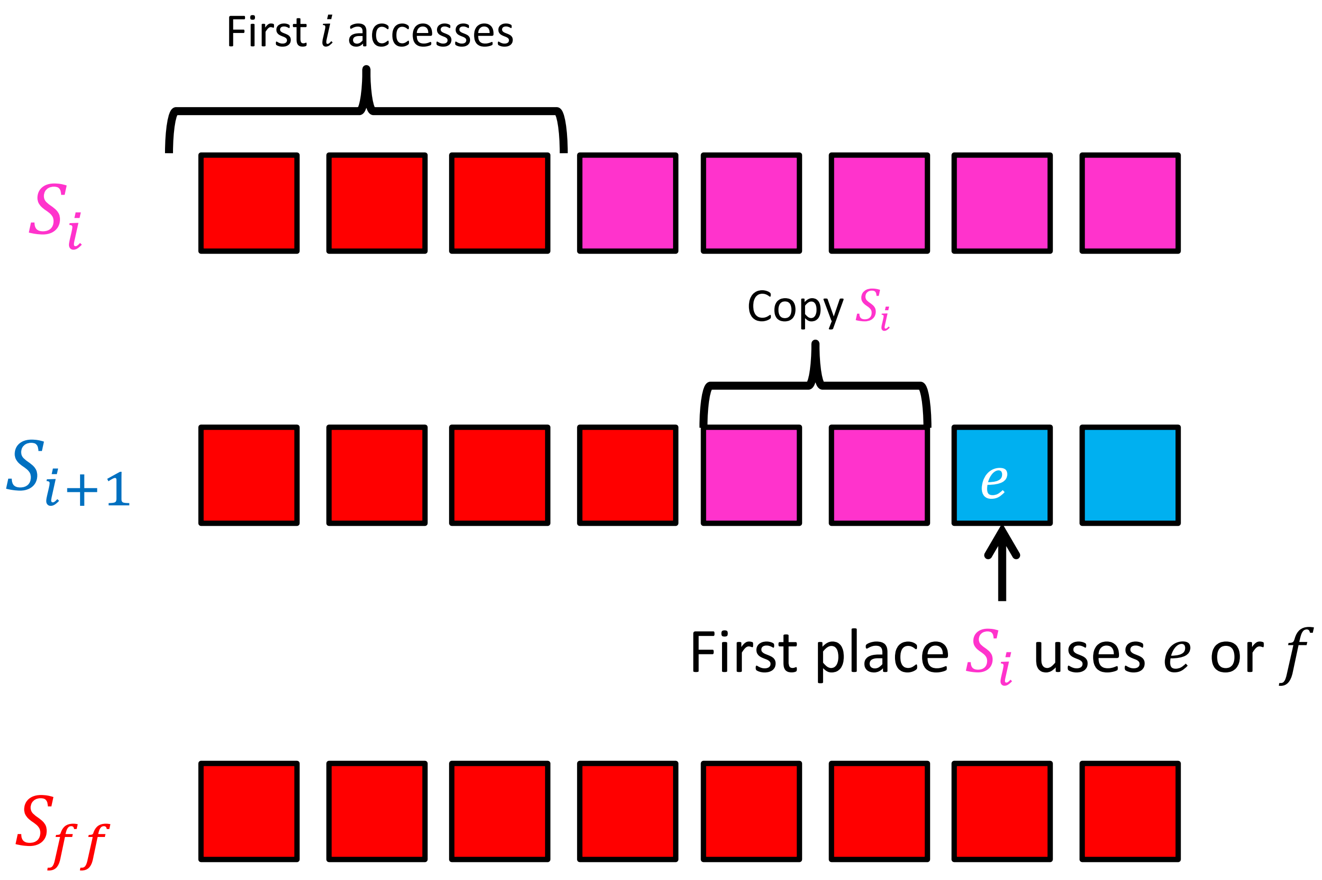
Case 3



m_t = the first access after $i+1$ in which S_i involveswith e or f

$\boldsymbol{m_t = e}$ or $\boldsymbol{m_t = f}$ or $\boldsymbol{m_t = x \neq e, f}$

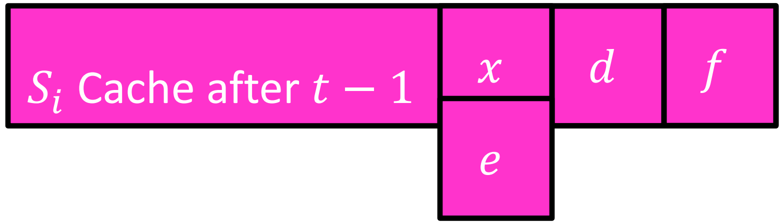
Case 3, $m_t = e$



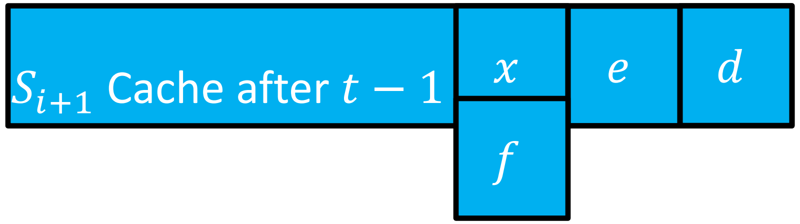
m_t = the first access after $i + 1$ in which S_i deals with e or f

Case 3, $m_t = e$

Goal: find S_{i+1} s.t. $misses(S_{i+1}) \leq misses(S_i)$



$\neq$



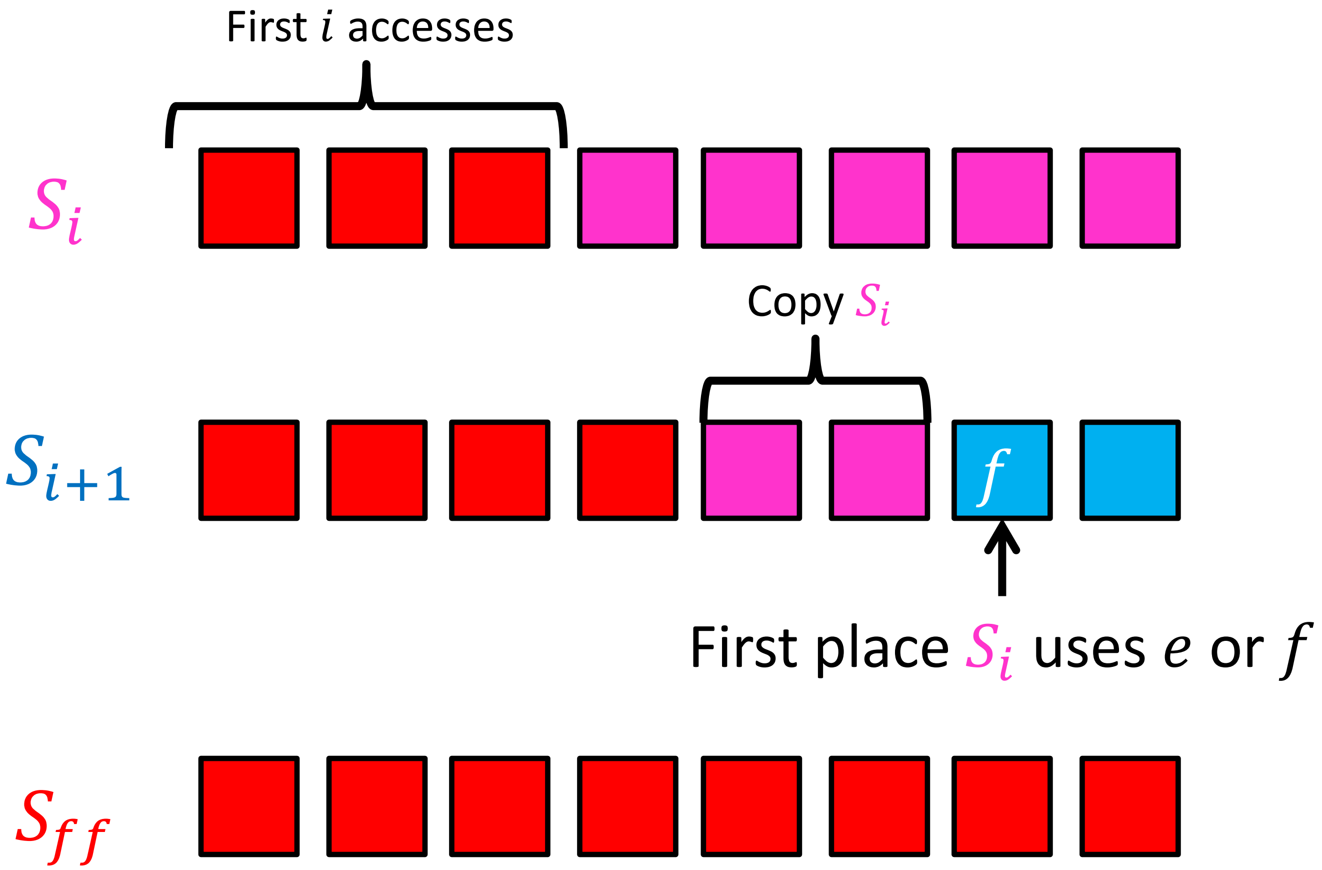
S_i must load e into the cache, assume it evicts x

S_{i+1} will load f into the cache, evicting x

The caches now match!

S_{i+1} behaved exactly the same as S_i between i and t, and has the same cache after t, therefore $misses(S_{i+1}) = misses(S_i)$

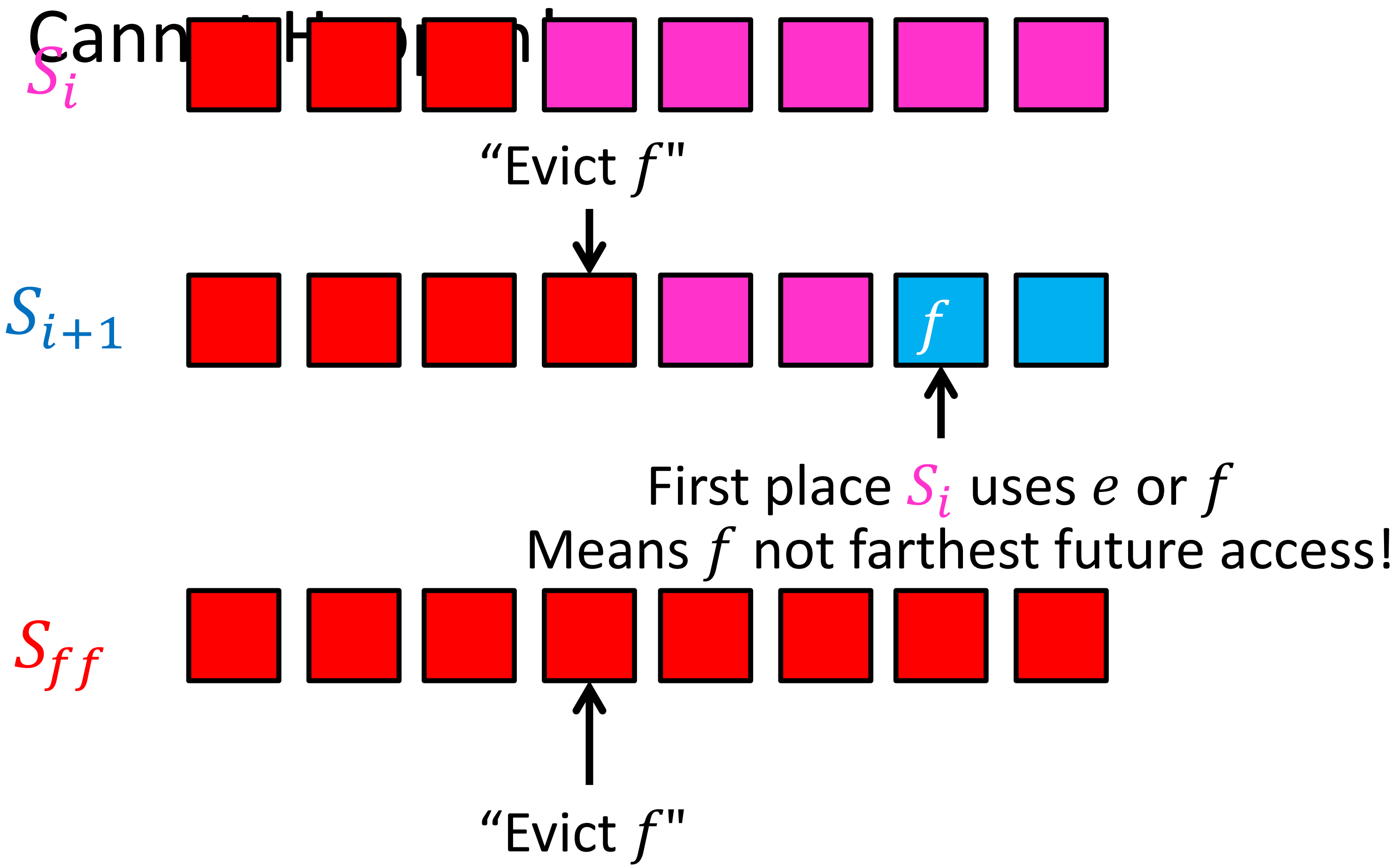
Case 3, $m_t = f$



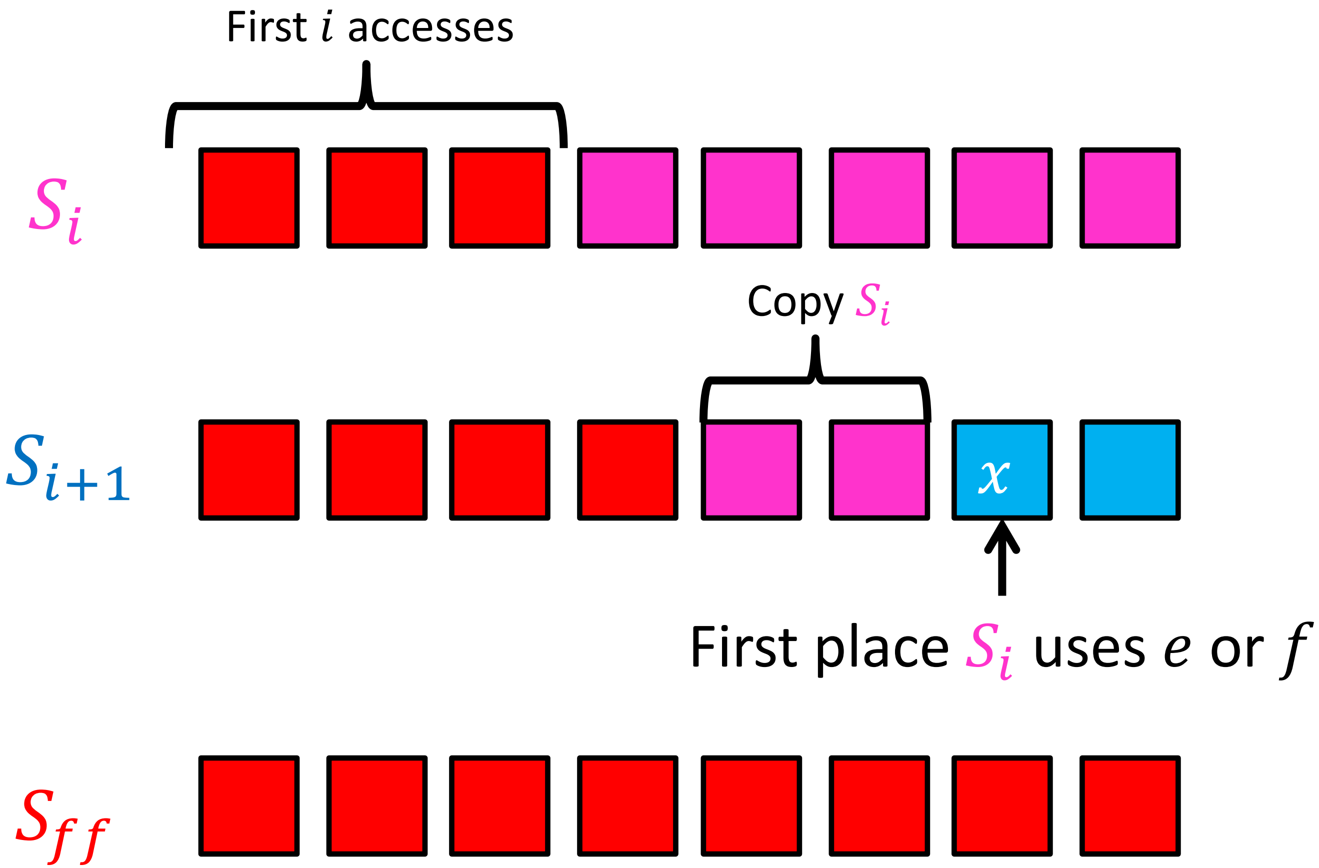
m_t = the first access after $i+1$ in which S_i deals with e or f

$\boldsymbol{m_t = e}$ or $\boldsymbol{m_t = f}$ or $\boldsymbol{m_t = x \neq e, f}$

Case 3, $m_t = f$


S_i
"Evict f"
S_{i+1}
f
First place S_i uses e or f
Means f not farthest future access!
S_{ff}
"Evict f"

Case 3, $m_t = x \neq e, f$

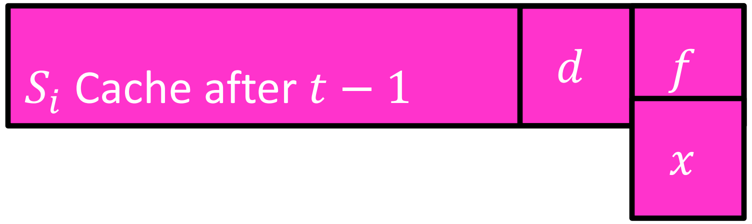


m_t = the first access after $i+1$ in which S_i deals with e or f

$m_t = e$ or $m_t = f$ or $m_t = x \neq e, f$

Case 3, $m_t = x \neq e, f$

Goal: find S_{i+1} s.t. $misses(S_{i+1}) \leq misses(S_i)$



$\neq$

S_i loads x into the cache, it must be evicting f

S_{i+1} will load x into the cache, evicting e

The caches now match!

S_{i+1} behaved exactly the same as S_i between i and t, and has the same cache after t, therefore $misses(S_{i+1}) = misses(S_i)$

Entire Huffman Derivation Follows

- Not covered in class, just for your review

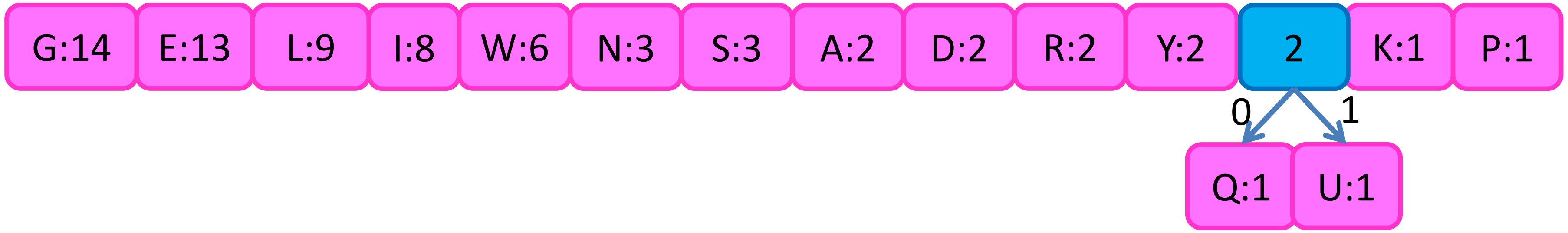
Huffman Algorithm

- 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

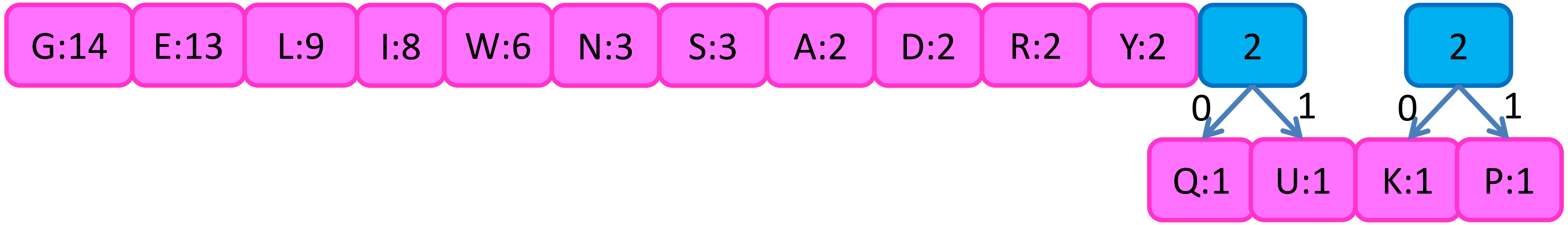
Huffman Algorithm

- Choose the least frequent pair, combine into a subtree



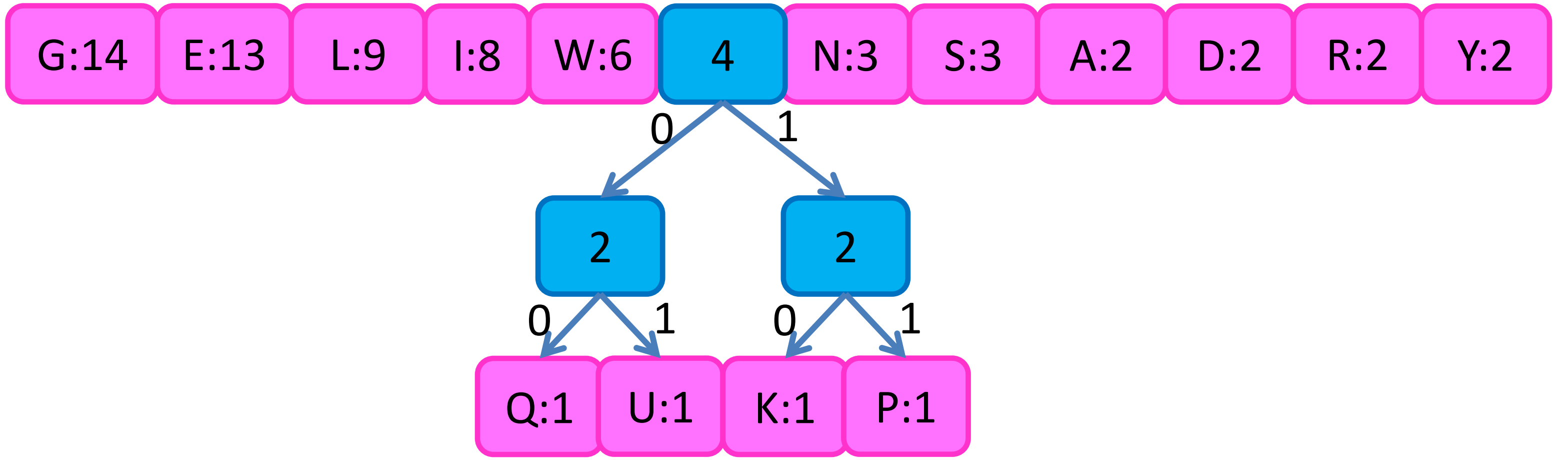
Huffman Algorithm

- Choose the least frequent pair, combine into a subtree



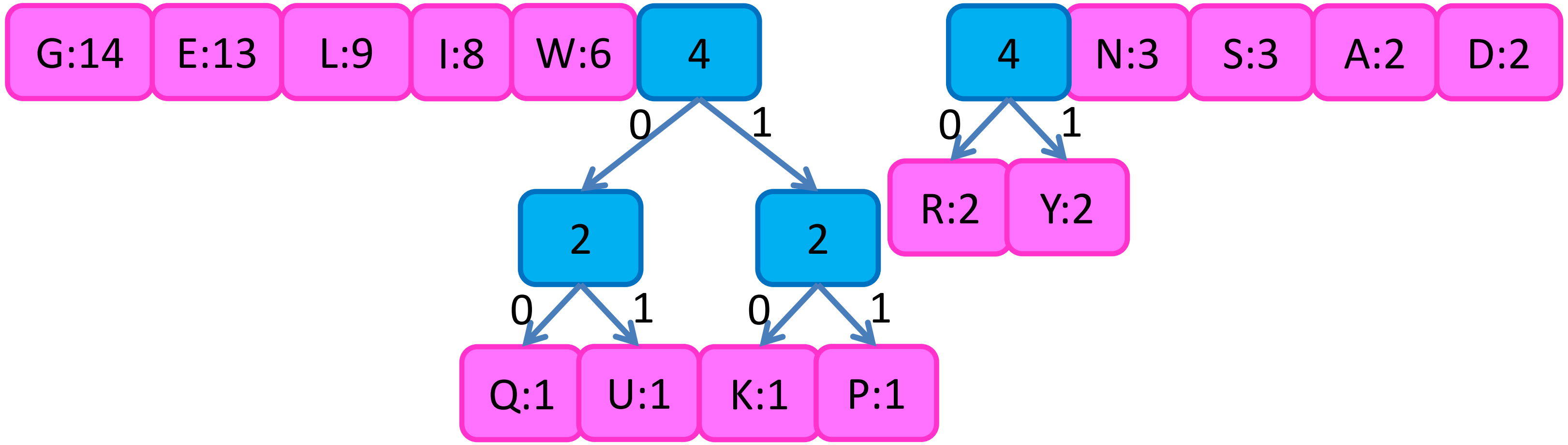
Huffman Algorithm

- Choose the least frequent pair, combine into a subtree



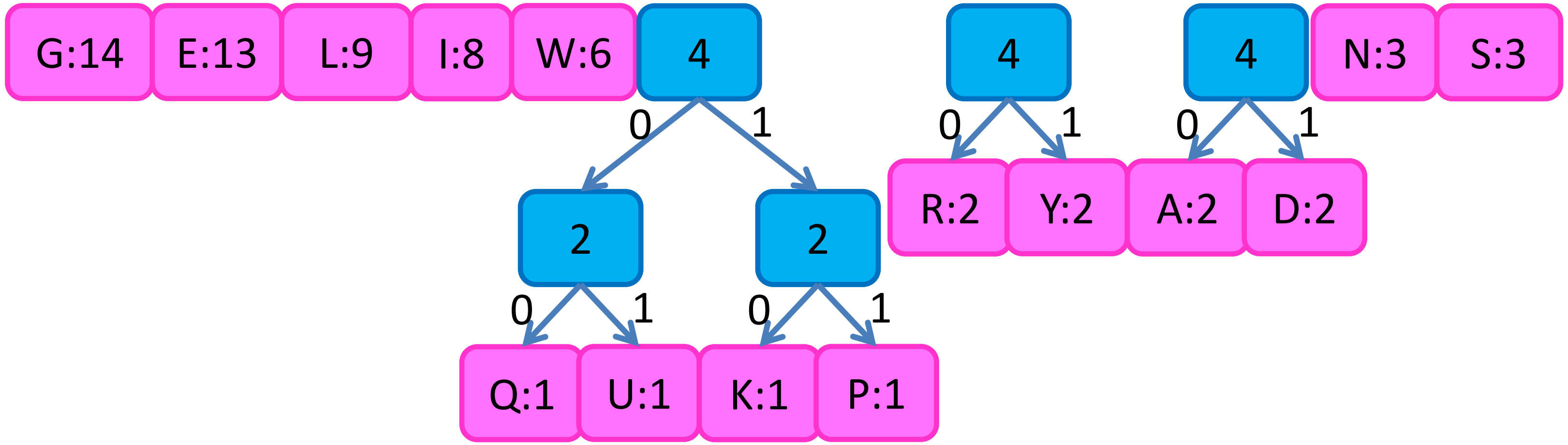
Huffman Algorithm

- Choose the least frequent pair, combine into a subtree



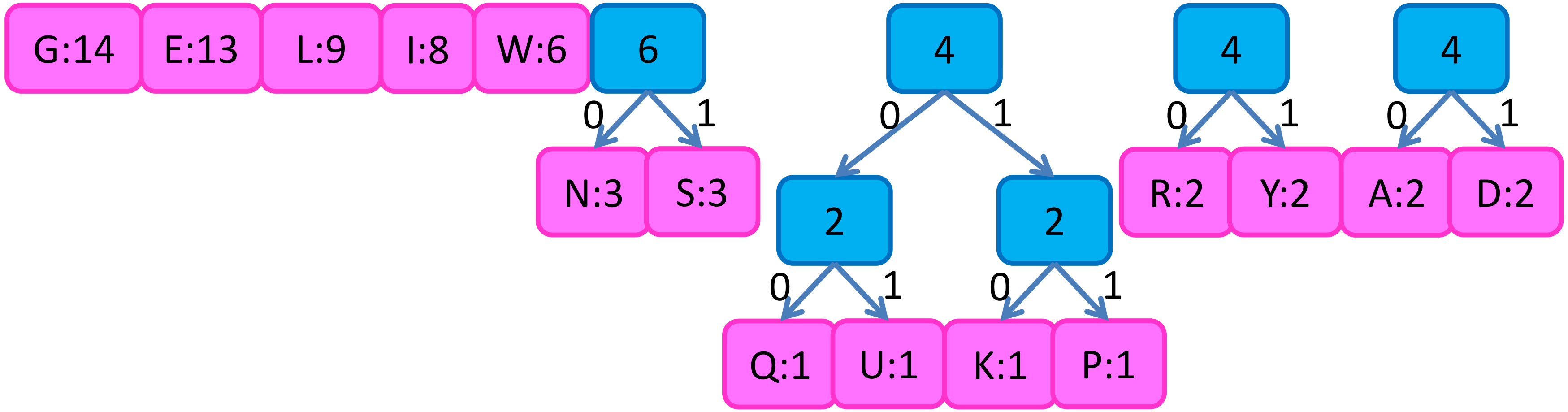
Huffman Algorithm

- Choose the least frequent pair, combine into a subtree



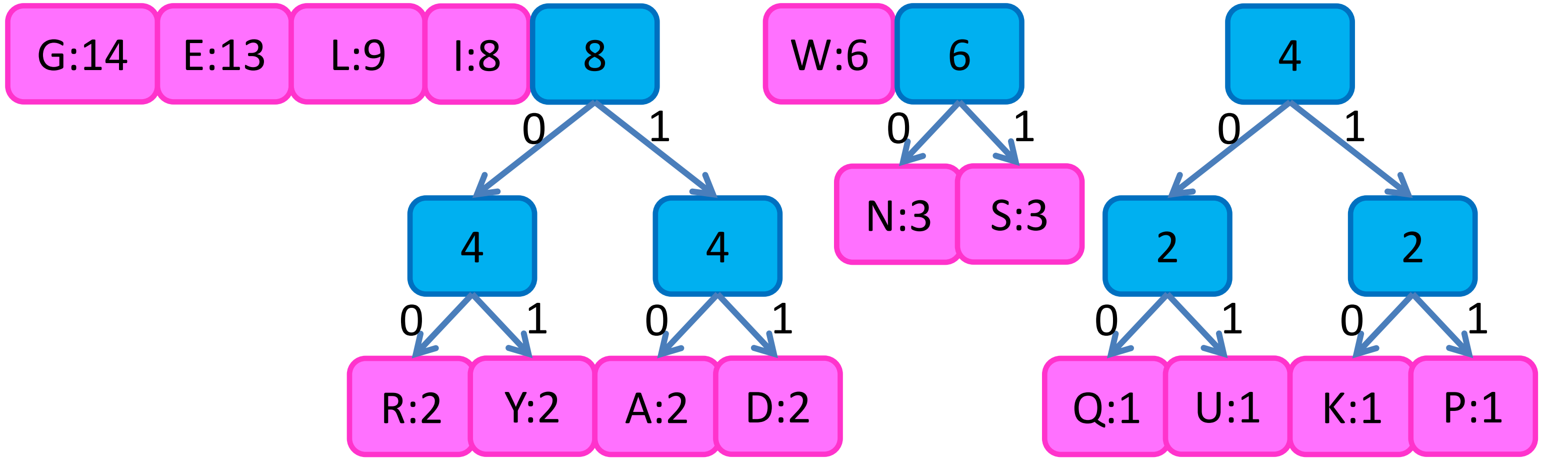
Huffman Algorithm

- Choose the least frequent pair, combine into a subtree



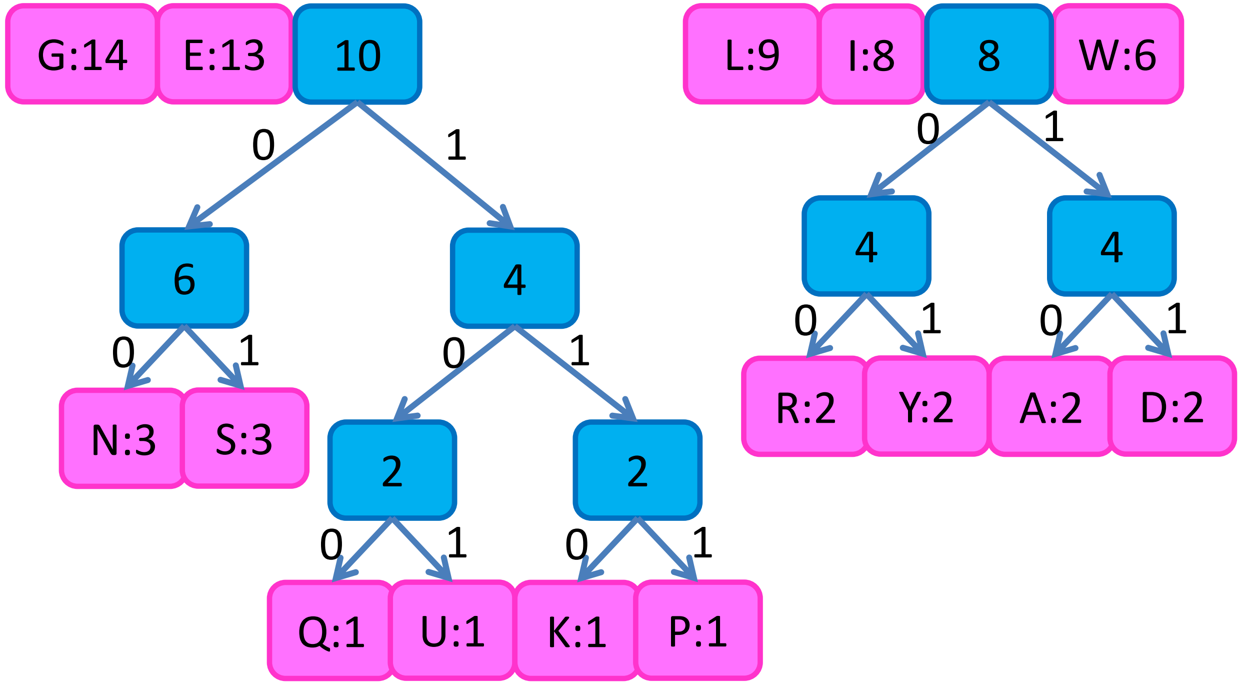
Huffman Algorithm

- Choose the least frequent pair, combine into a subtree



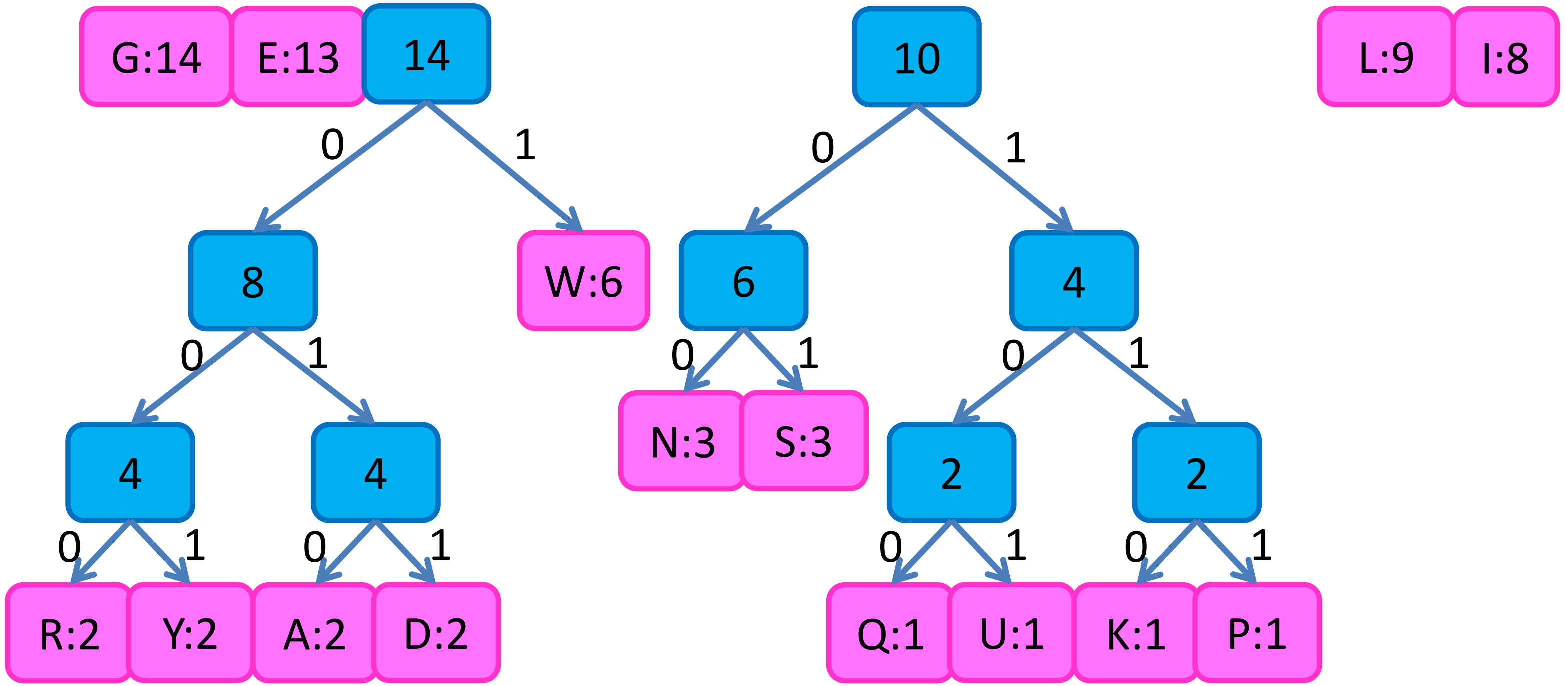
Huffman Algorithm

- Choose the least frequent pair, combine into a subtree



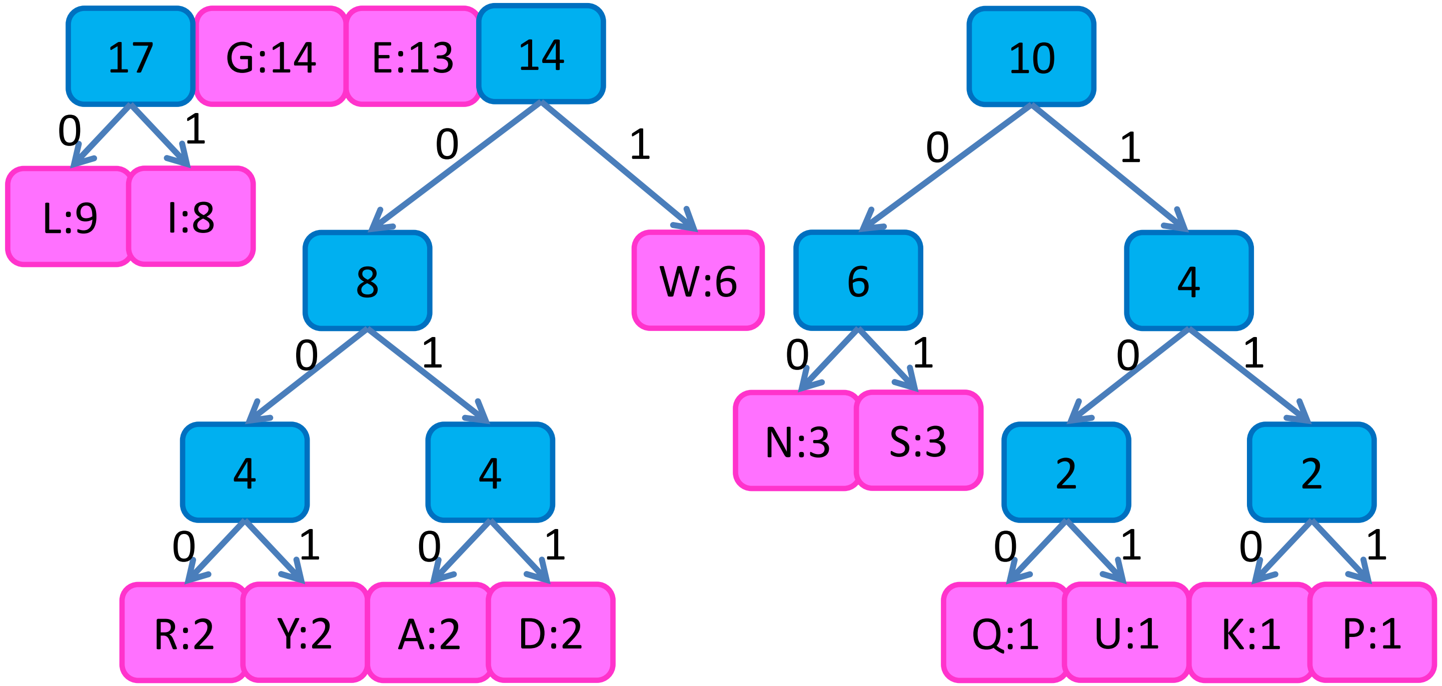
Huffman Algorithm

- Choose the least frequent pair, combine into a subtree



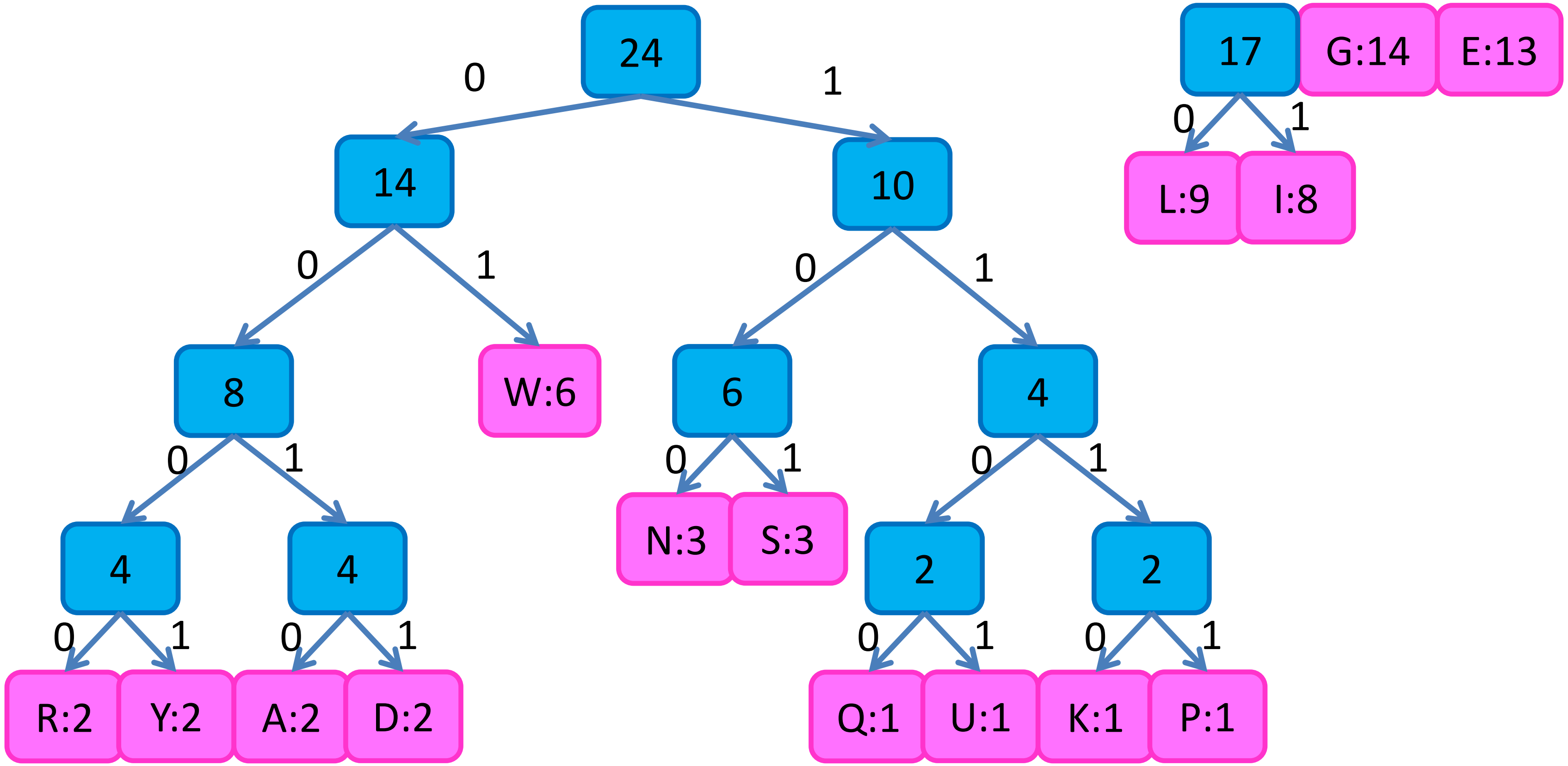
Huffman Algorithm

- Choose the least frequent pair, combine into a subtree



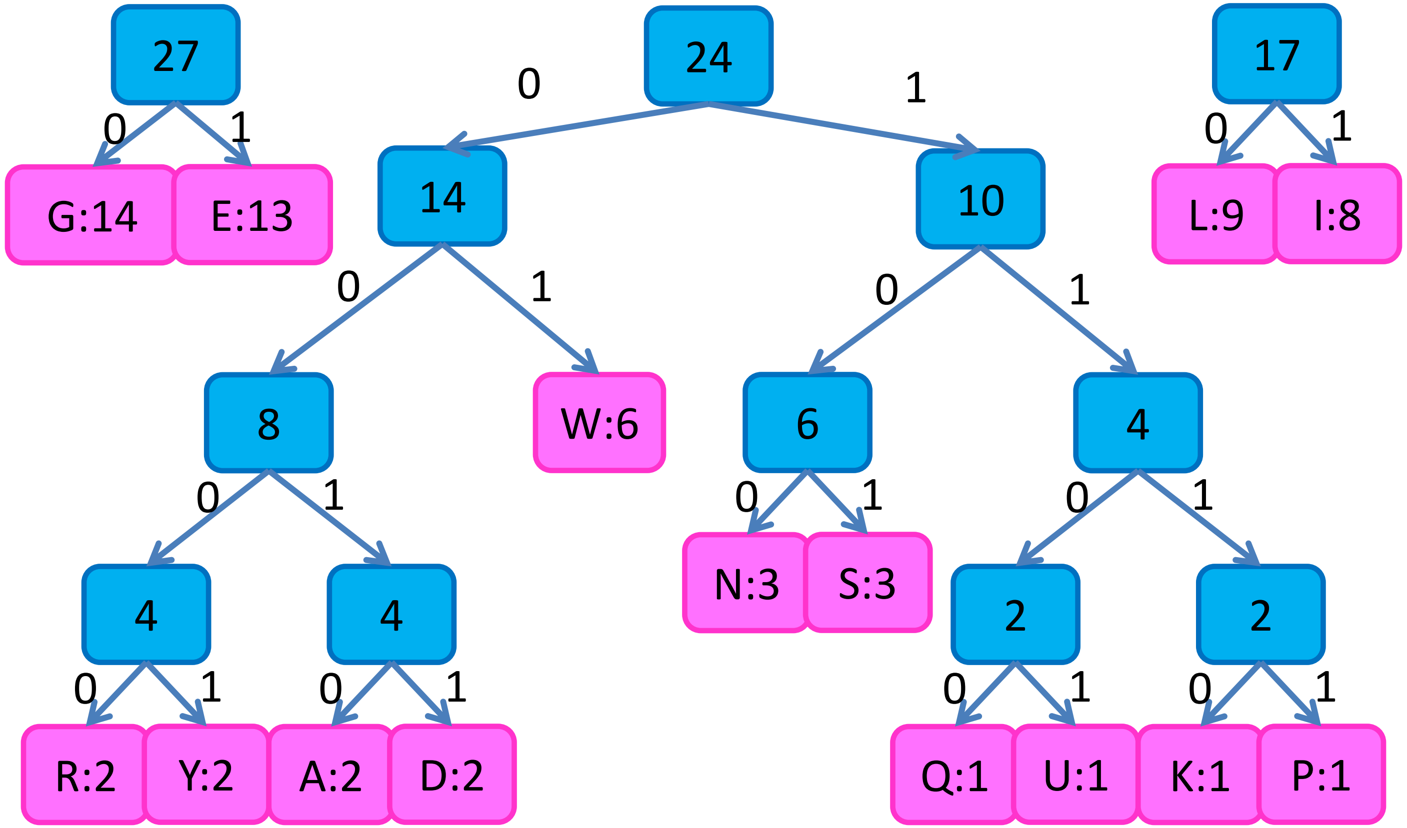
Huffman Algorithm

- Choose the least frequent pair, combine into a subtree



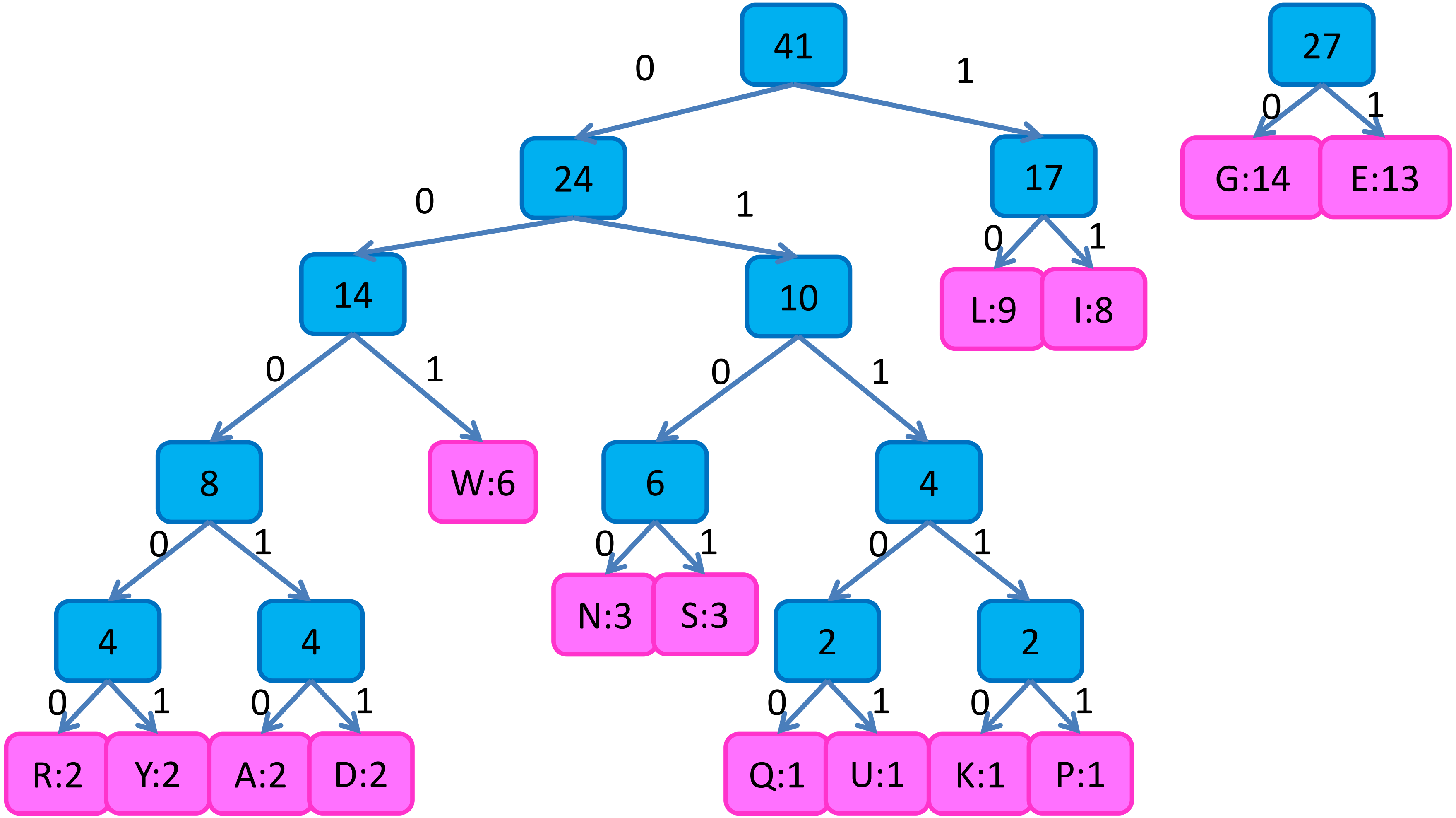
Huffman Algorithm

- Choose the least frequent pair, combine into a subtree



Huffman Algorithm

- Choose the least frequent pair, combine into a subtree



Huffman Algorithm

- Choose the least frequent pair, combine into a subtree