CS 3100
Data Structures and Algorithms 2
Lecture 15: Huffman Encoding &

Co-instructors: Robbie Hott and Ray Pettit

Spring 2024

Readings in CLRS 4th edition:

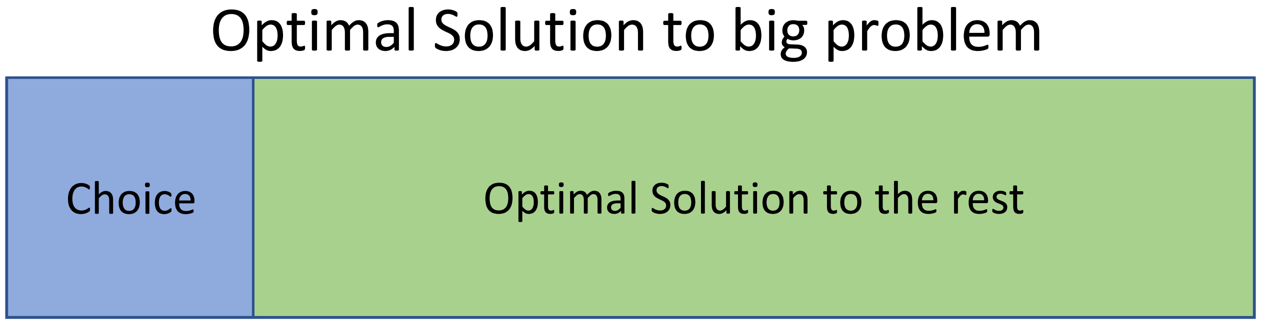
- Chapter 16

Announcements

- PS6 due tomorrow (3/20)
- PA3 due Friday (3/22)
- Grading update
  - Quiz 1 and PS3 have been returned
  - We are currently grading: Quiz 2, PS4, PS5
- Office hours (reminder)
  - Prof Hott Office Hours: Traveling this week
  - Prof Pettit Office Hours: Mondays and Fridays 2:30-4:00p
  - TA office hours posted on our website
  - Office hours are not for “pre-grading"

Reminders about Greedy Algorithms

Greedy Algorithms


Optimal Solution to big problem

Choice

Optimal Solution to the rest

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

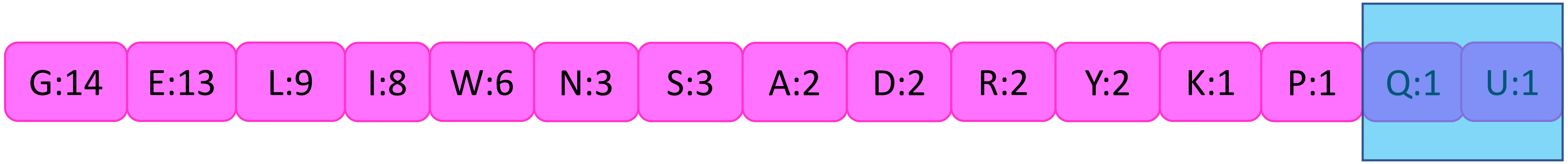
- The rule for how to choose an item guaranteed be in the optimal solution

Greedy Algorithm Procedure:

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

Huffman Algorithm

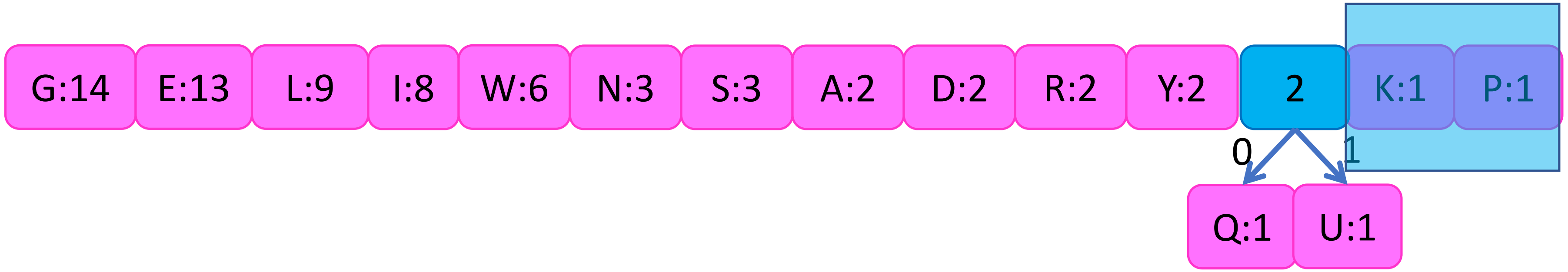
Choose the least frequent pair, combine into a subtree



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

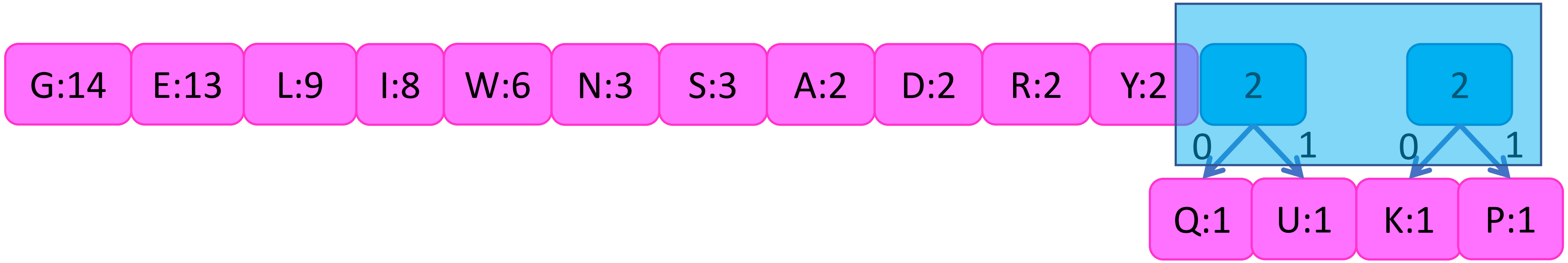
Choose the least frequent pair, combine into a subtree



Subproblem of size $n - 1$!

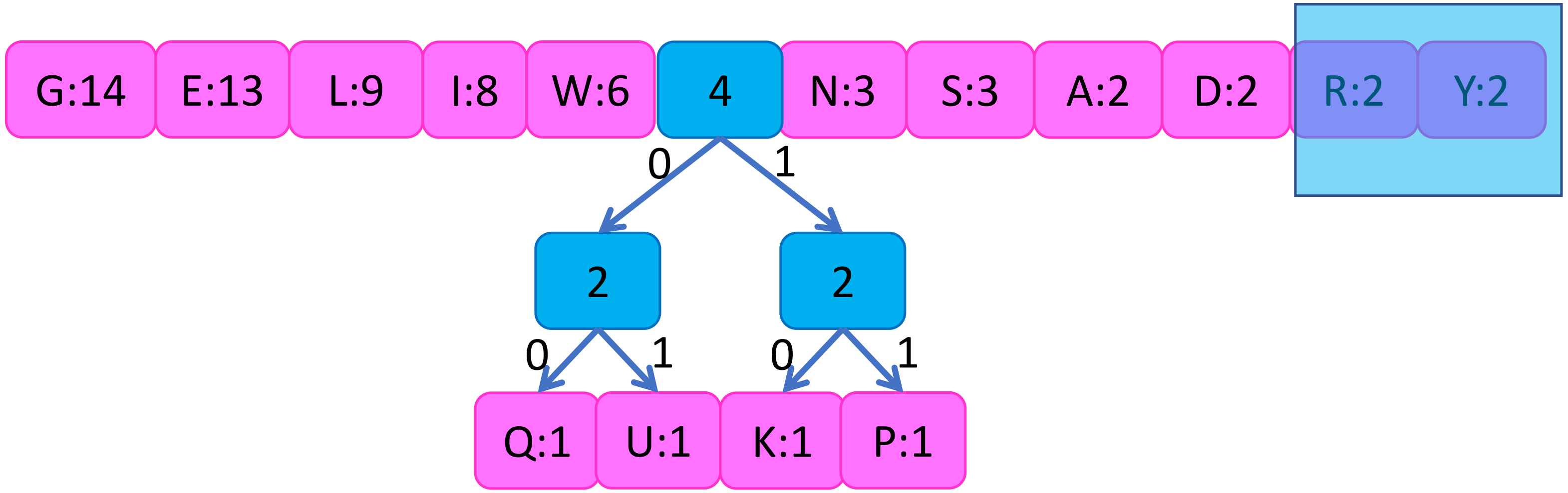
Huffman Algorithm

Choose the least frequent pair, combine into a subtree



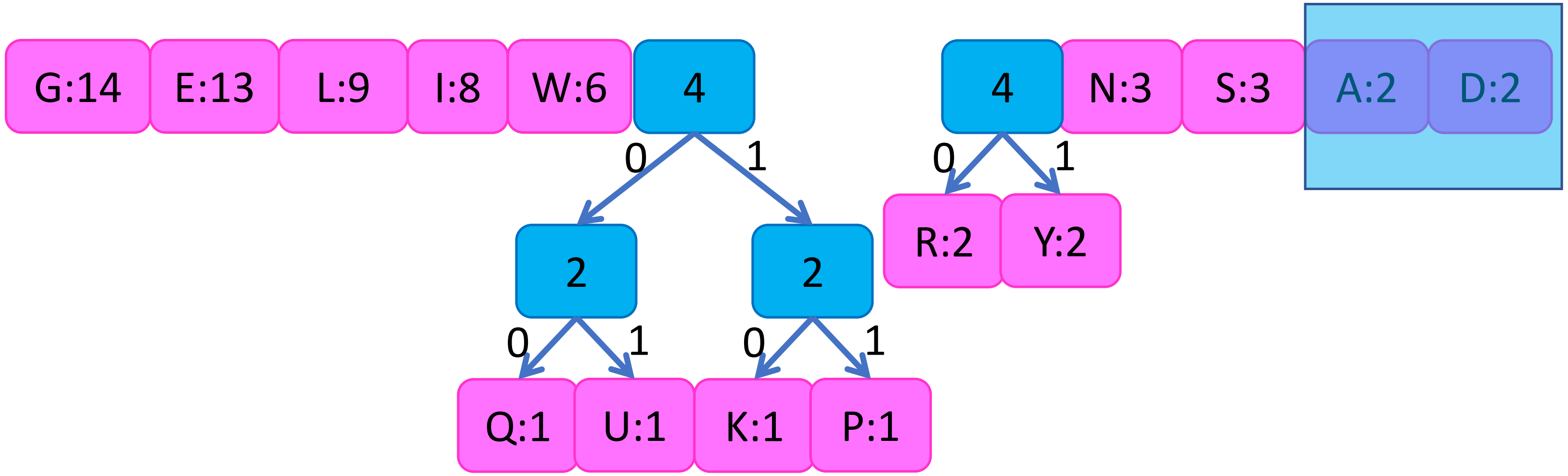
Huffman Algorithm

Choose the least frequent pair, combine into a subtree



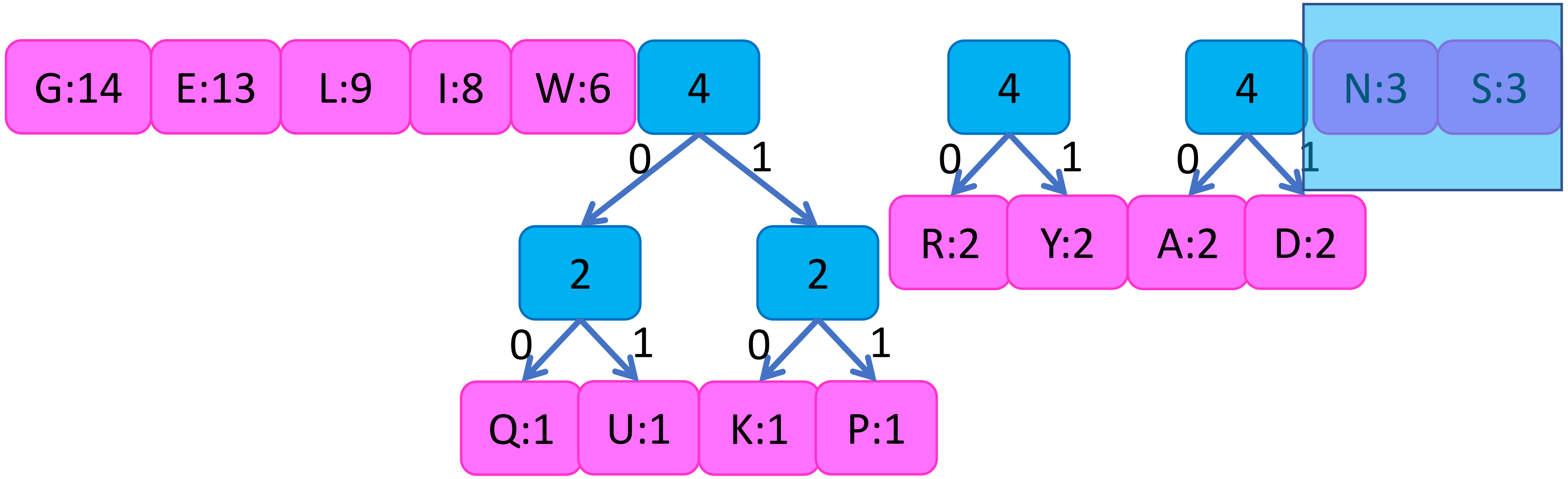
Huffman Algorithm

Choose the least frequent pair, combine into a subtree



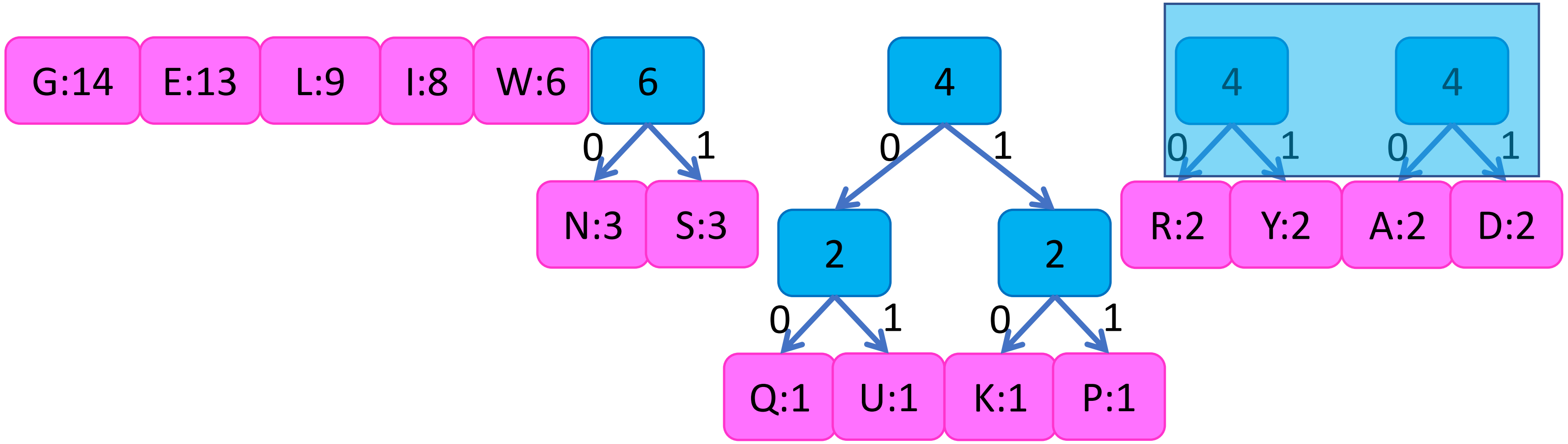
Huffman Algorithm

Choose the least frequent pair, combine into a subtree



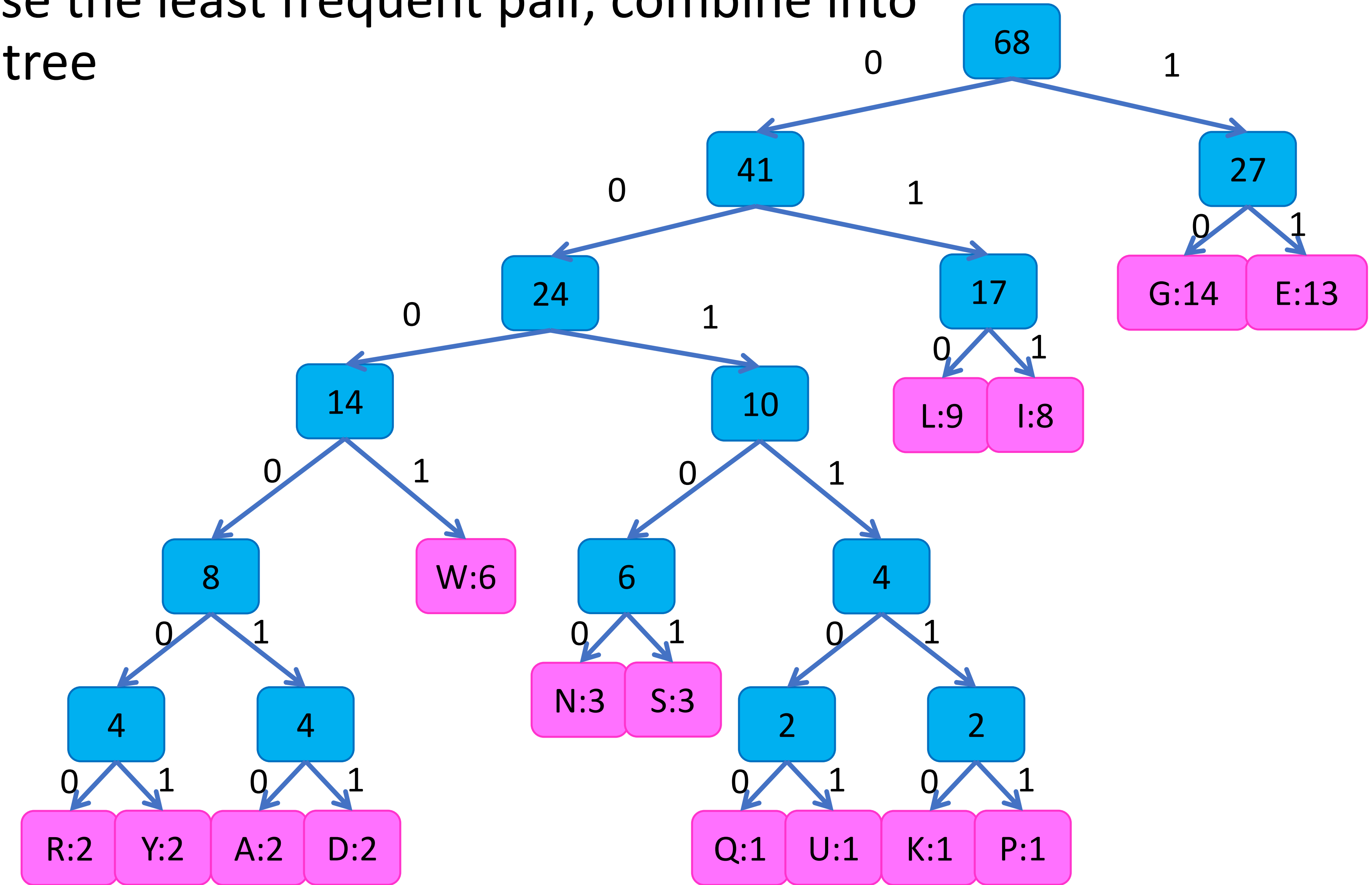
Huffman Algorithm

Choose the least frequent pair, combine into a subtree



Huffman Algorithm

Choose the least frequent pair, combine into a subtree



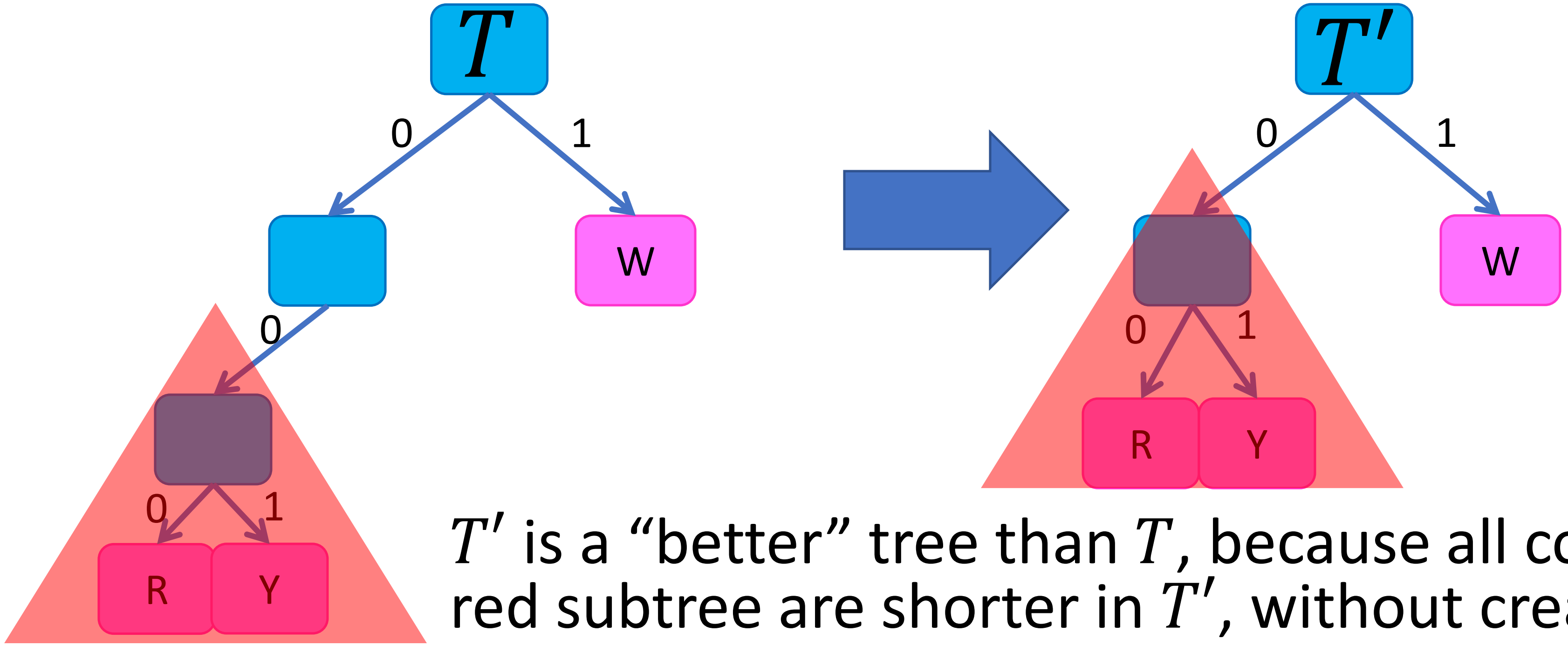
Showing Huffman is Optimal

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)



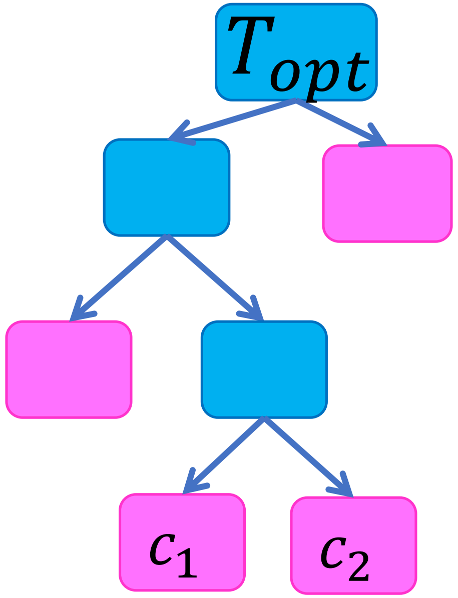
T' is a “better” tree than T, because all codes in red subtree are shorter in T', without creating any longer codes

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

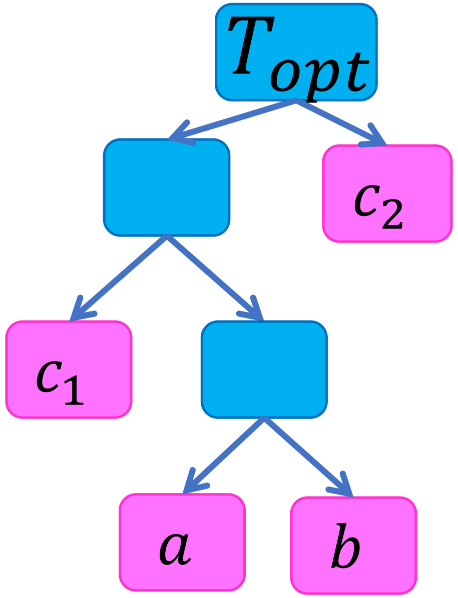


Huffman Exchange Argument

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: $f_{c1} \leq f_a$ and $f_{c2} \leq f_b$

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: $f_{c1} \leq f_a$

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

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

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

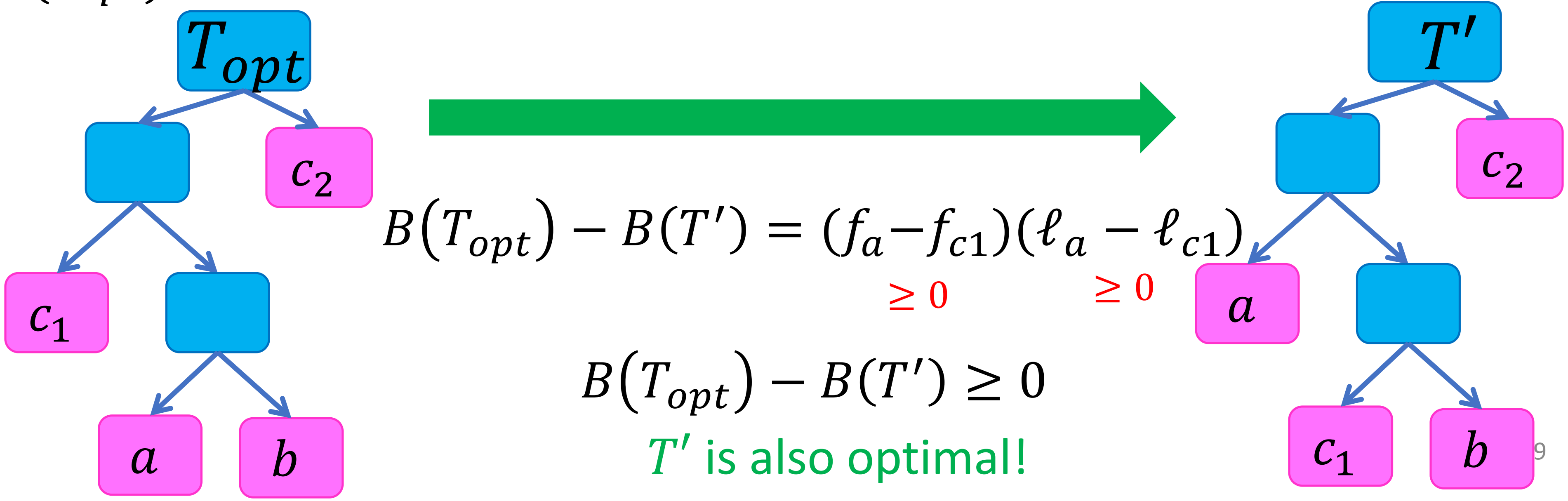
$a, b =$ lowest-depth siblings

Idea: show that swapping c_1 with a does not increase cost of the tree.

Assume: $f_{c1} \leq f_a$

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

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



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

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

T' is also optimal!

Case 2:Repeat to swap c_2, b!

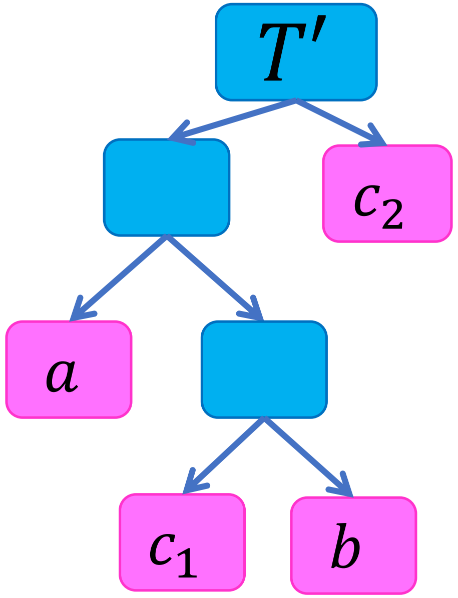
- Claim: the least-frequent characters (c_1, c_2), are siblings in some optimal tree

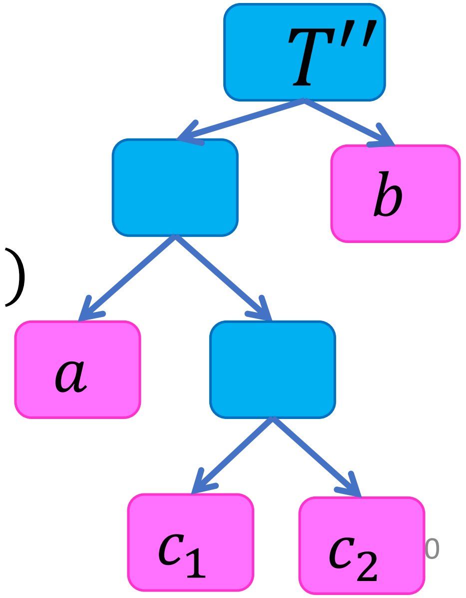
$a, b =$ lowest-depth siblings

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

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

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





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

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

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:

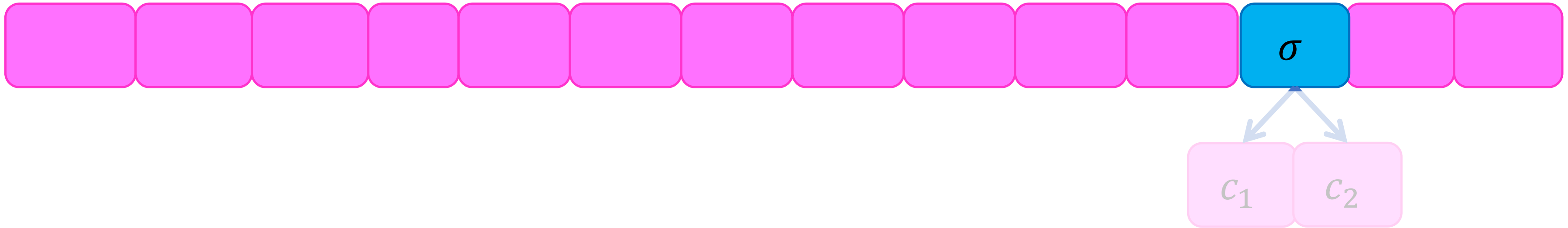
- Assume that c_1, c_2 are siblings in at least one optimal solution
- Assume that solving the subproblem with this new character, then adding in c_1, c_2 is not optimal
- Show that removing c_1, c_2 from a better overall solution must produce a better solution to the subproblem

Finishing the Proof

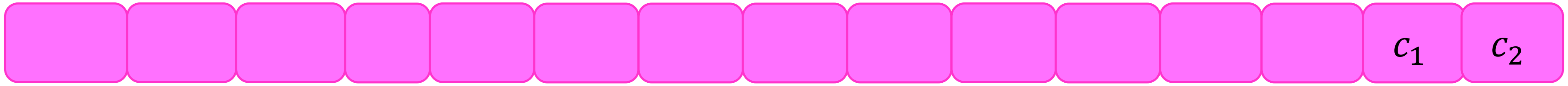
Show Recursive Substructure

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

Why does solving this smaller problem:

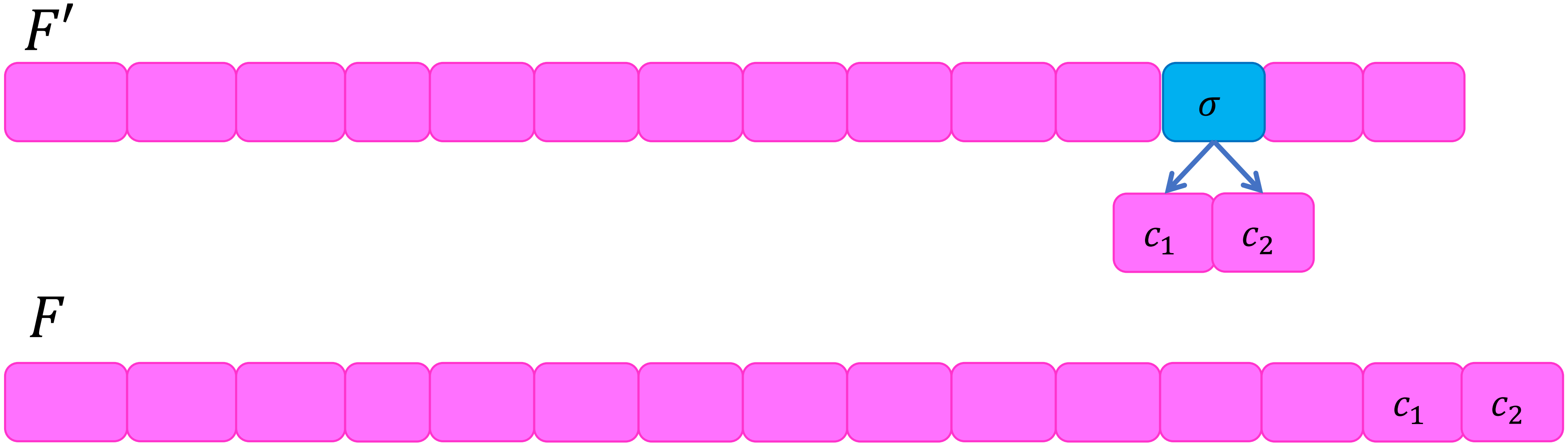


Give an optimal solution to this?:



Substructure

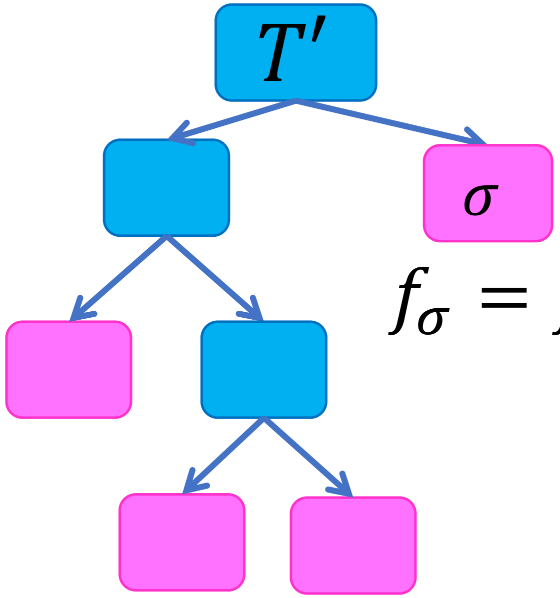
Claim: An optimal solution for F involves finding an optimal solution for F', then adding c_1, c_2 as children to σ



Substructure

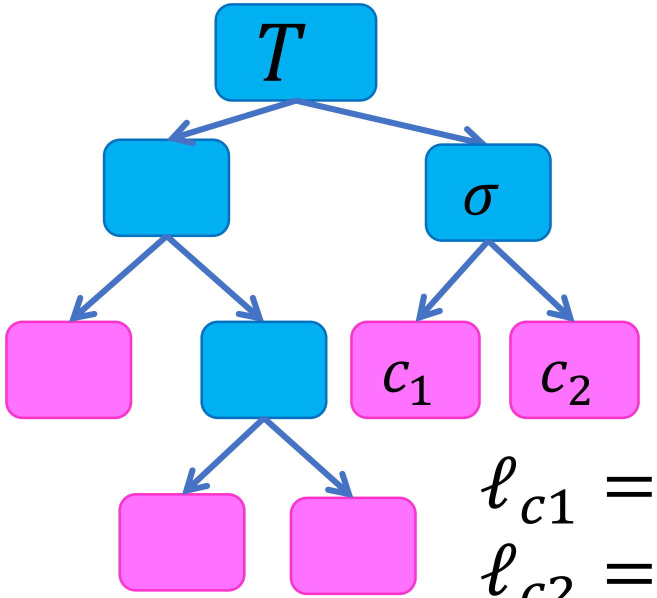
Claim: An optimal solution for F involves finding an optimal solution for F', then adding c_1, c_2 as children to σ

If this is optimal



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

Then this is optimal



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

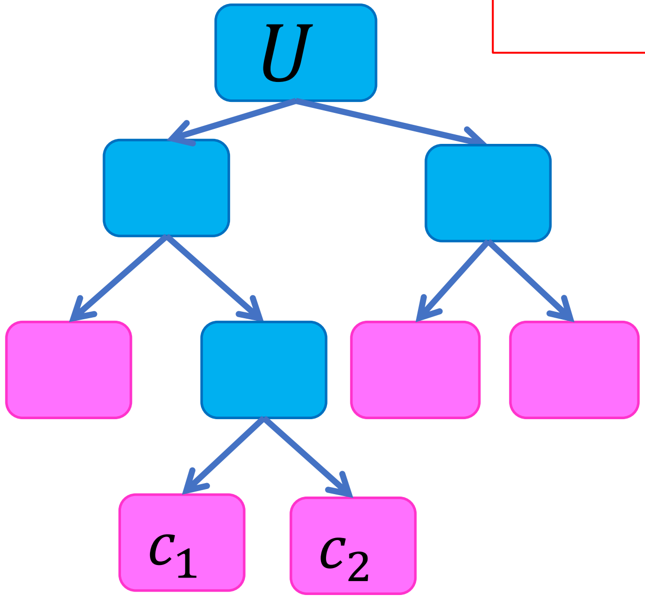
$$B(T') = B(T) - f_{c1} - f_{c2}$$

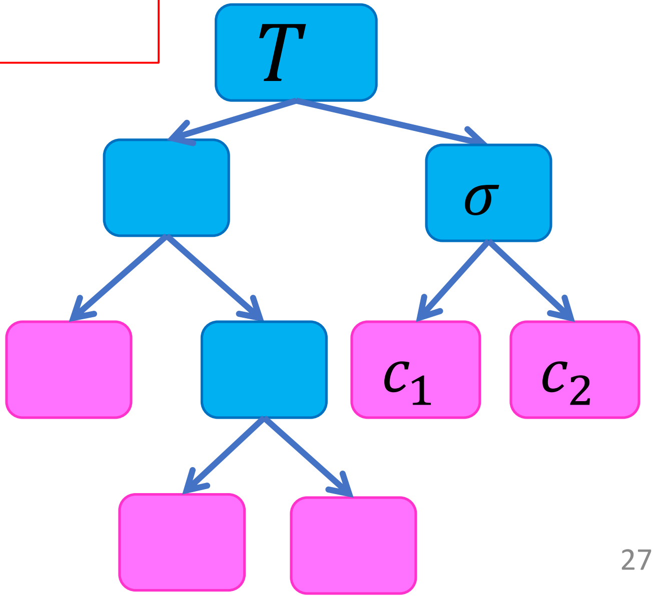
Substructure

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
$B(U) < B(T)$



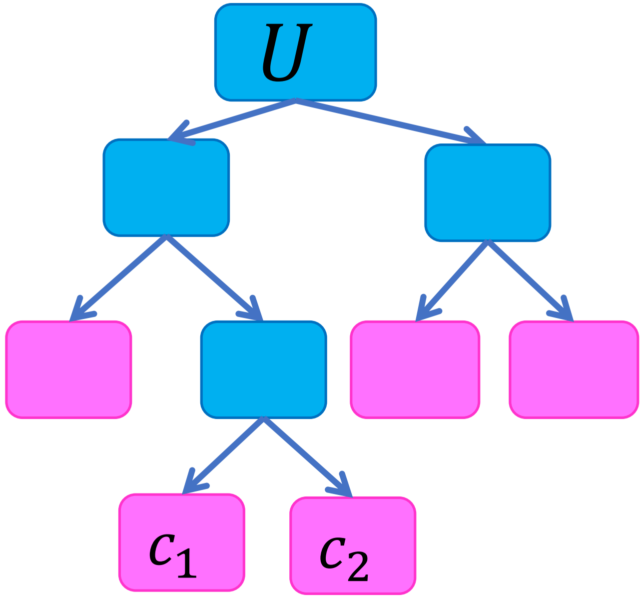


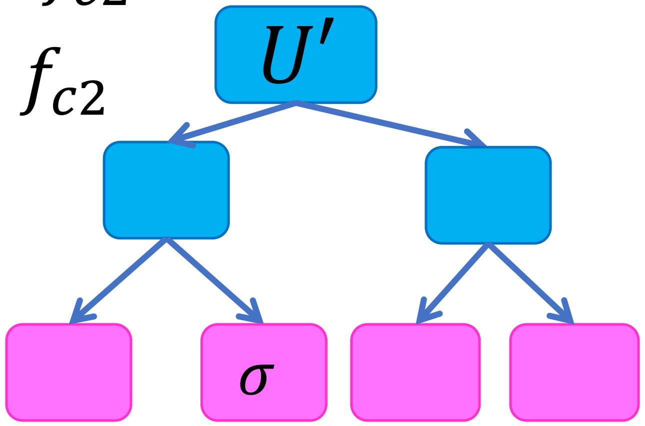
Substructure

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

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

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

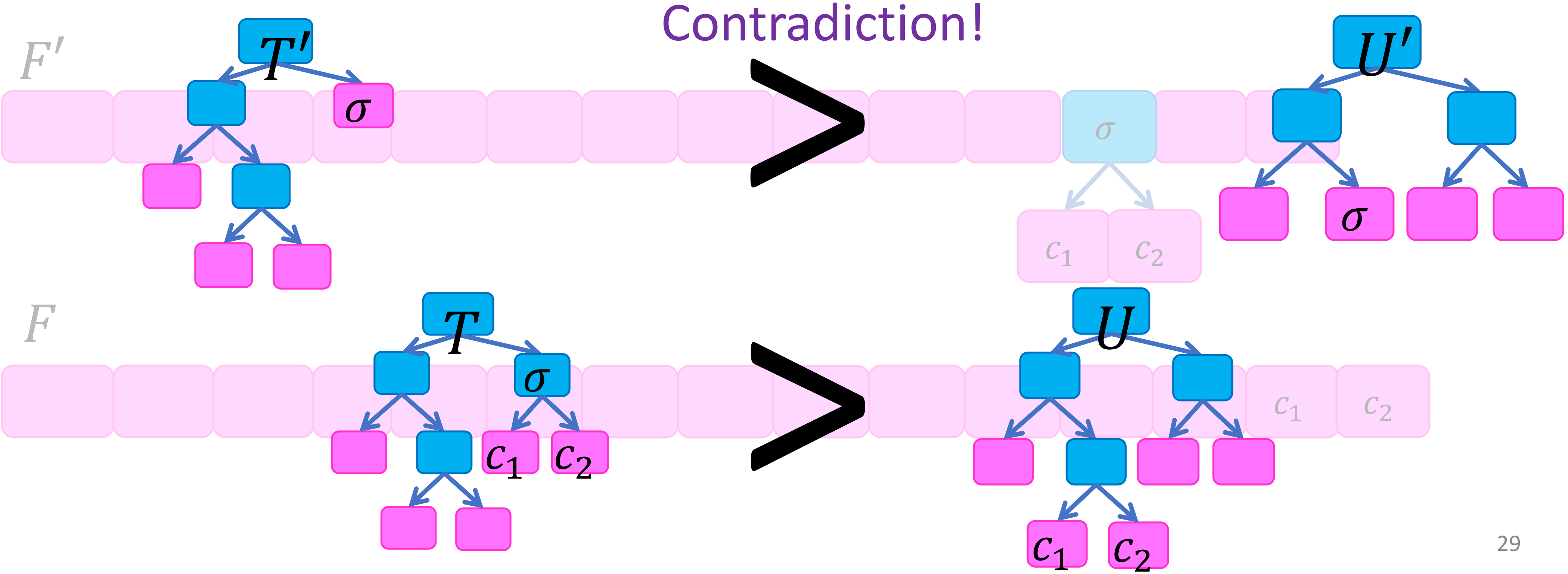




Contradicts optimality of T', so T is optimal!

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



Let’s Talk About Memory

Why using lots of memory is “bad”

Using too much memory forces you to use slow memory

Memory == $$

May have too little memory for the algorithm to even run

Lots of memory => not parallelizable

Contention for the memory

Memory <= time

Von Neumann bottleneck

Cache coherency

Fast memory is expensive

Von Neumann Bottleneck

Named for John von Neumann

Inventor of modern computer architecture

Other notable influences include:

- Mathematics
- Physics
- Economics
- Computer Science

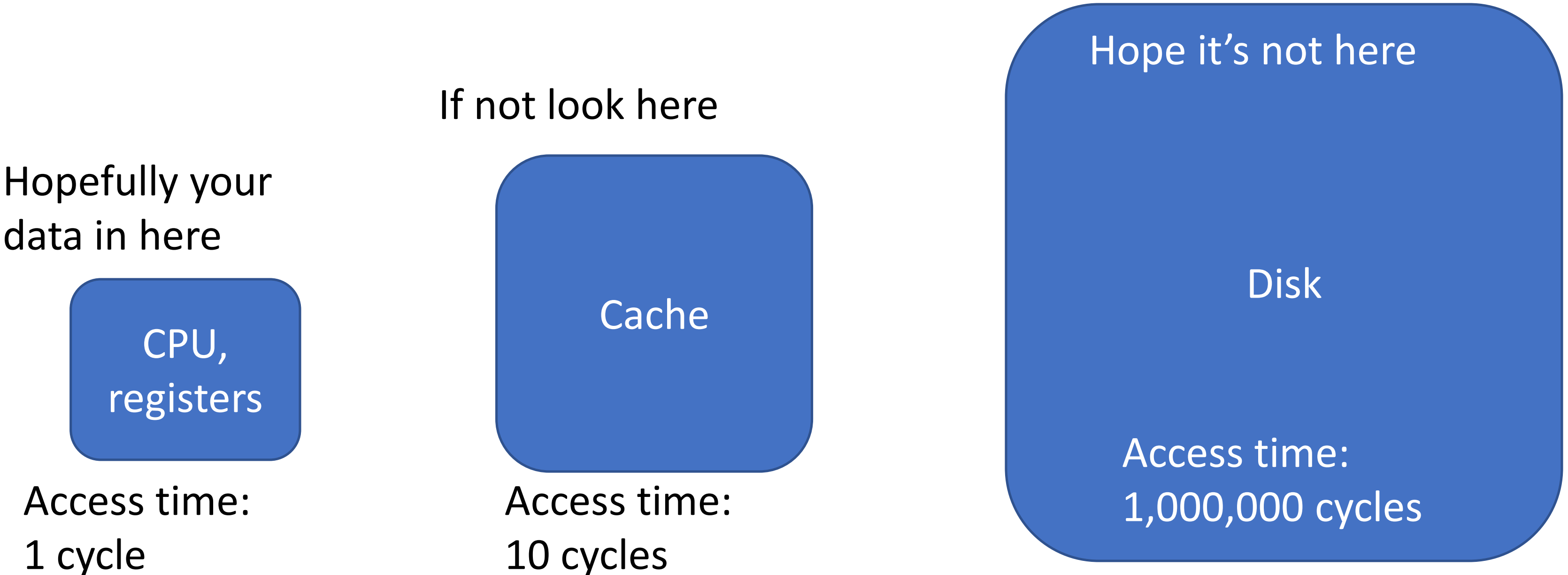
Von Neumann Bottleneck

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



Hopefully your data in here

CPU, registers

Access time: 1 cycle

If not look here

Cache

Access time: 10 cycles

Hope it’s not here

Disk

Access time: 1,000,000 cycles

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
- $M = [m_1, m_2, \ldots m_n] =$ 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

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

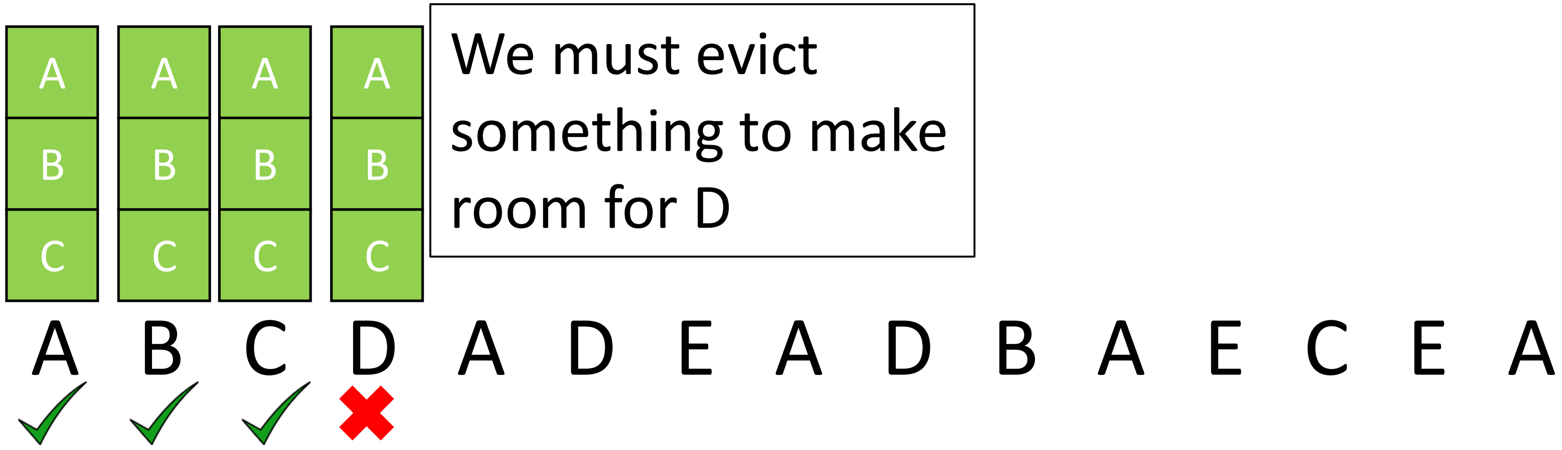
Example

A	A	A
B	B	B
C	C	C

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

✓ ✓ ✓

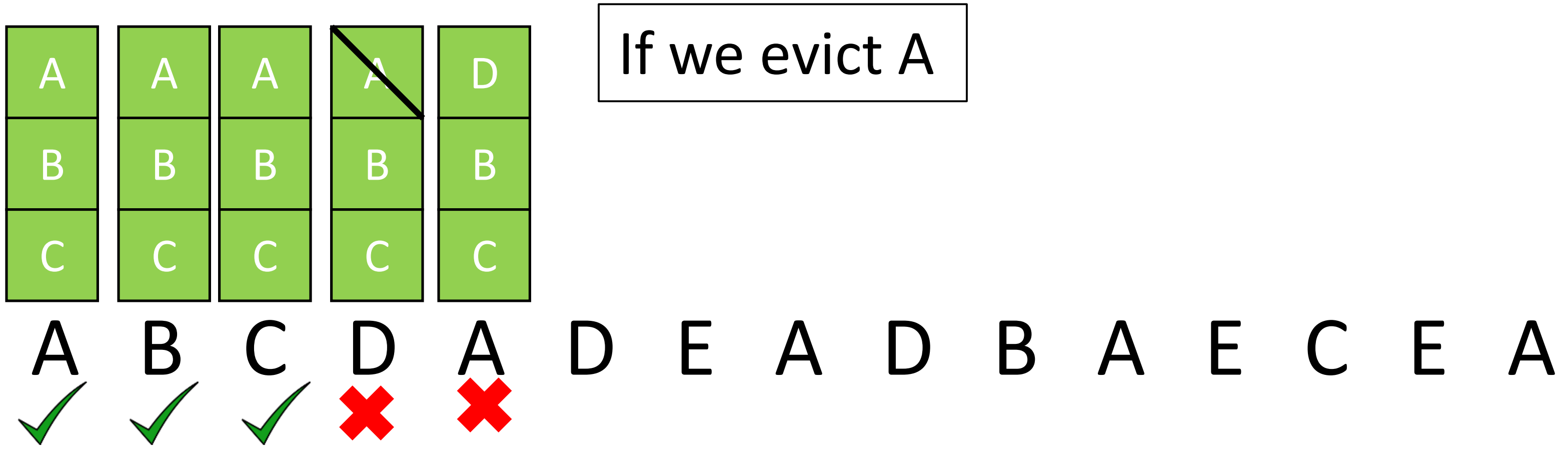
Example



A	B	C	D	A	D	E	A	D	B	A	E	C	E	A
A	A	A	A											
B	B	B	B											
C	C	C	C											
✓	✓	✓	✖											

We must evict something to make room for D

Example



A	A	A	~~A~~	D
B	B	B	B	B
C	C	C	C	C

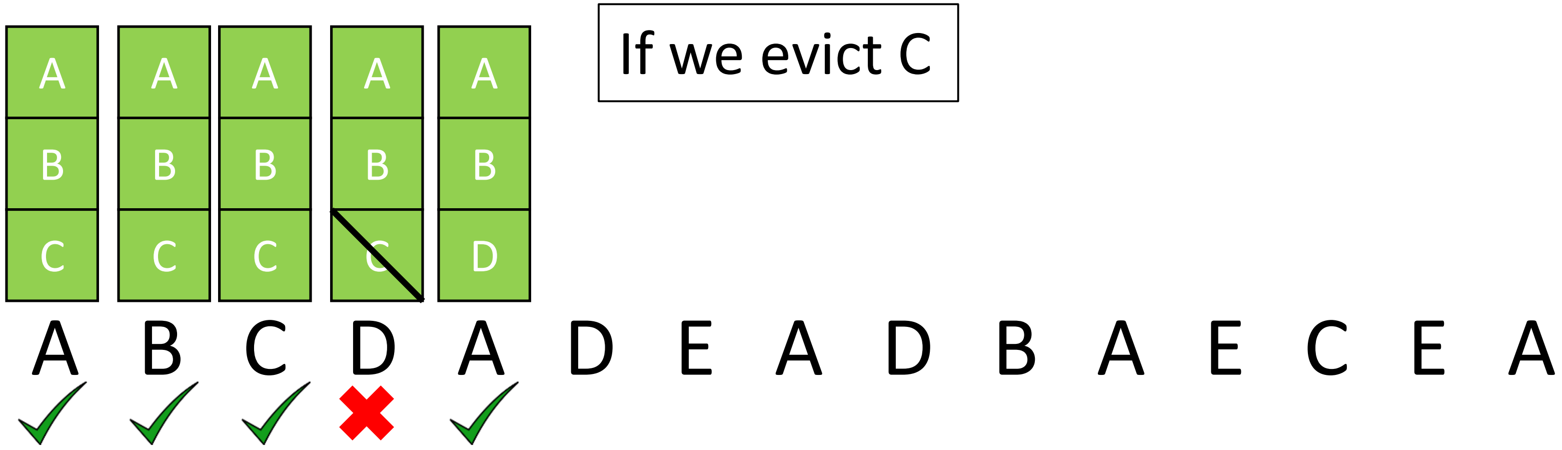
A B C D A D E A D B A E C E A

✓ ✓ ✓ ✗ ✗

If we evict A

Example

If we evict C



A	B	C	D	A	D	E	A	D	B	A	E	C	E	A
A	A	A	A	A										
B	B	B	B	B										
C	C	C	~~C~~	D										
✓	✓	✓	✖	✓										

Our Problem vs Reality

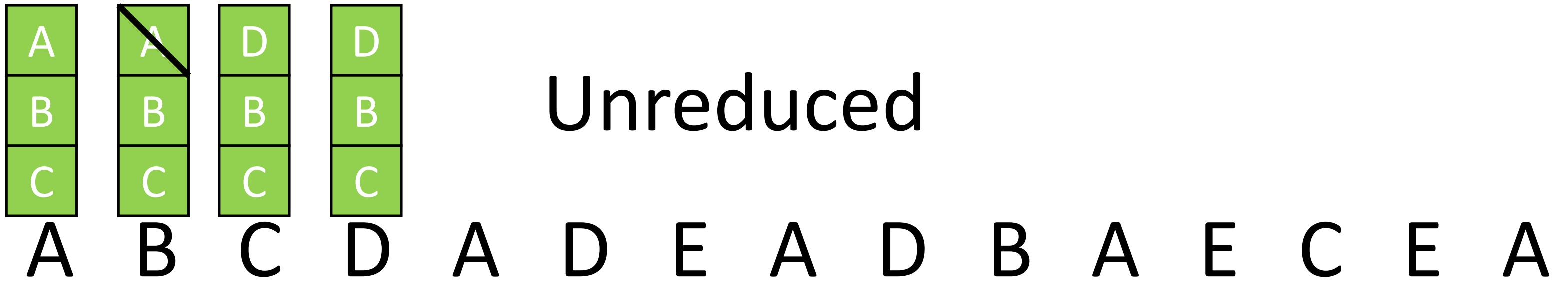
Assuming we know the entire access pattern

Cache is Fully Associative

Counting # of fetches (not necessarily misses)

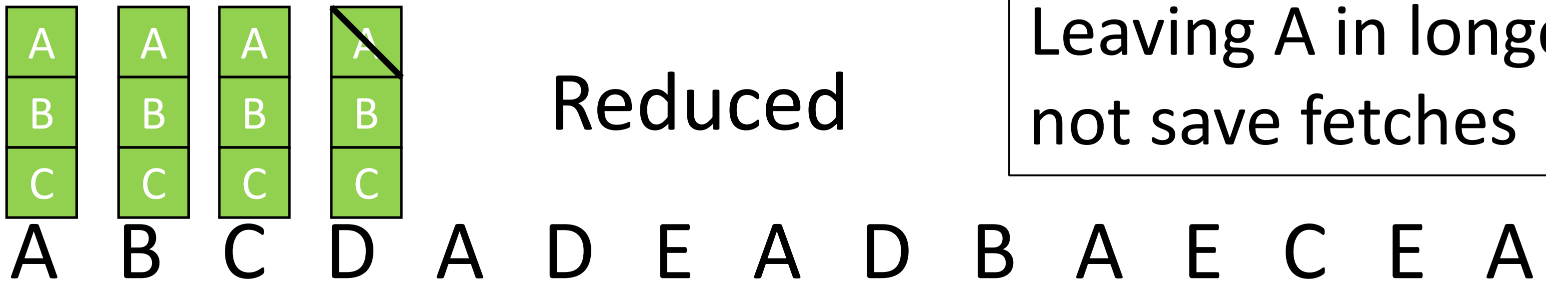
“Reduced” Schedule: Address only loaded on the cycle it’s required

- Reduced == Unreduced (by number of fetches)



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

Evict C

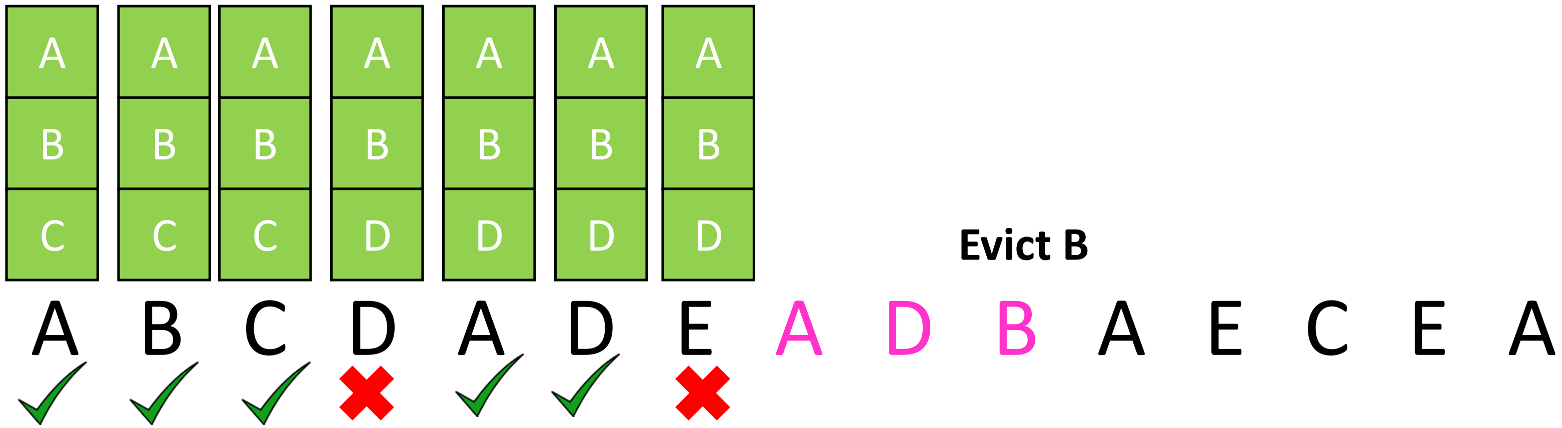
A B C D A D E A D B A E C E A

✓ ✓ ✓ ✗

Greedy choice property

Belady evict rule:

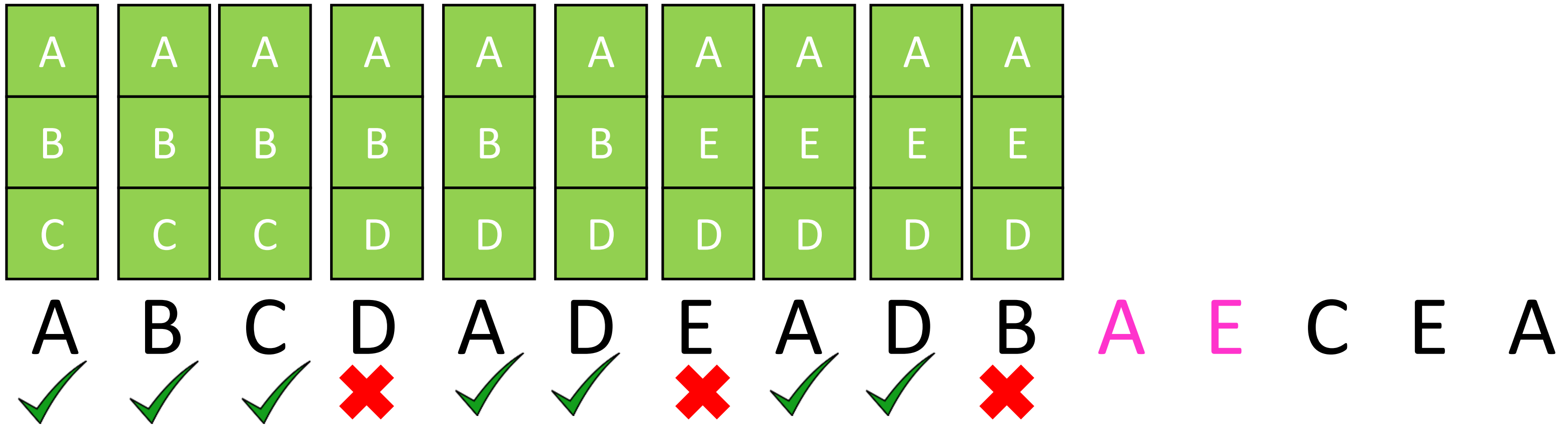
- Evict the item accessed farthest in the future



Greedy choice property

Belady evict rule:

- Evict the item accessed farthest in the future



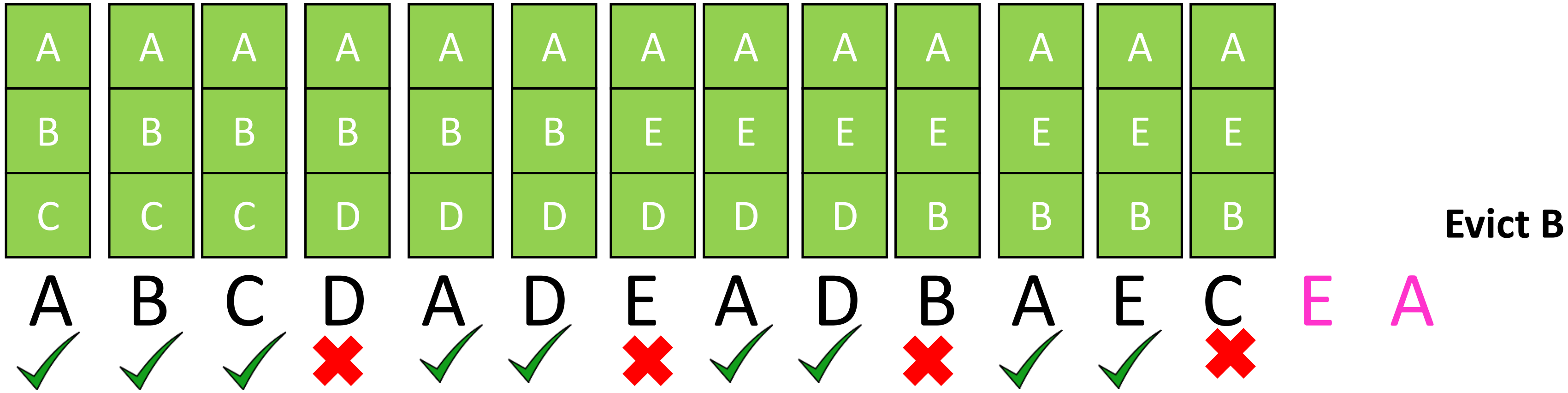
A	A	A	A	A	A	A	A	A	A					
B	B	B	B	B	B	E	E	E	E					
C	C	C	D	D	D	D	D	D	D					
A	B	C	D	A	D	E	A	D	B	A	E	C	E	A
✓	✓	✓	✗	✓	✓	✗	✓	✓	✗					

Evict D

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		
B	B	B	B	B	B	E	E	E	E	E	E	E		
C	C	C	D	D	D	D	D	D	B	B	B	B		
A	B	C	D	A	D	E	A	D	B	A	E	C	E	A
✓	✓	✓	✖	✓	✓	✖	✓	✓	✖	✓	✓	✖		

Evict B

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”

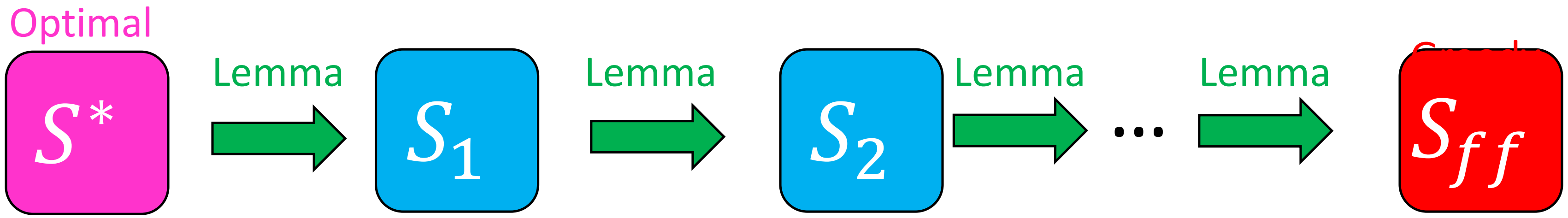
Belady Exchange Lemma

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



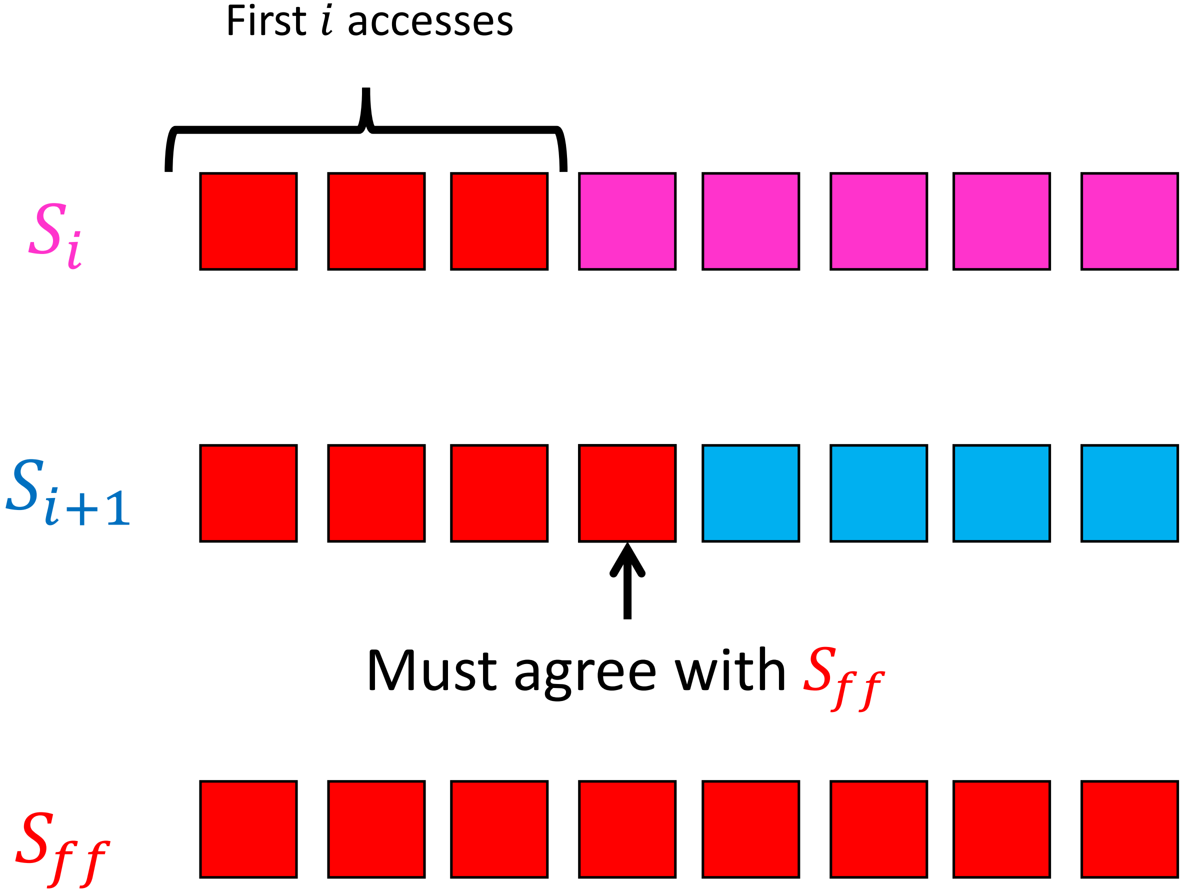
Agrees with S_{ff} on first 0 accesses

Agrees with S_{ff} on first access

Agrees with S_{ff} on first 2 accesses

Agrees with S_{ff} on all n accesses

Belady Exchange Proof Idea



First i accesses

S_i

S_{i+1}

Must agree with S_{ff}

S_{ff}

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	d	e	f

$=$

S_{ff} Cache after i	d	e	f

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

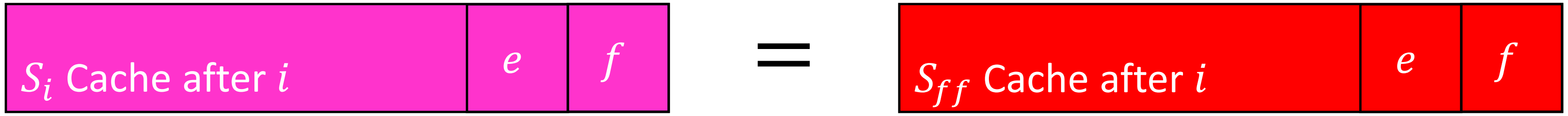
Case 1: if d is in the cache, then neither S_i nor S_{ff} evict from the cache, use the same cache for S_{i+1}

S_{i+1} Cache after i	d	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



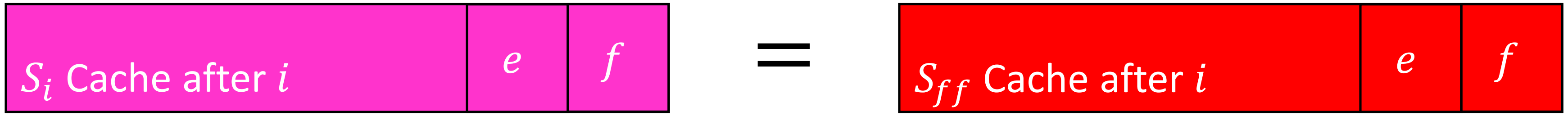
Consider access $m_{i+1} = d$

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

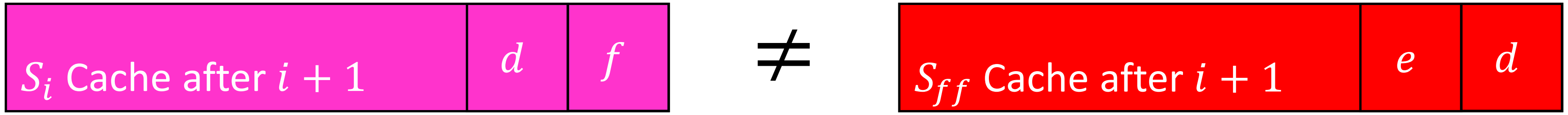
Since S_i agrees with S_{ff} for the first i accesses, the state of the cache at access $i+1$ will be the same



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

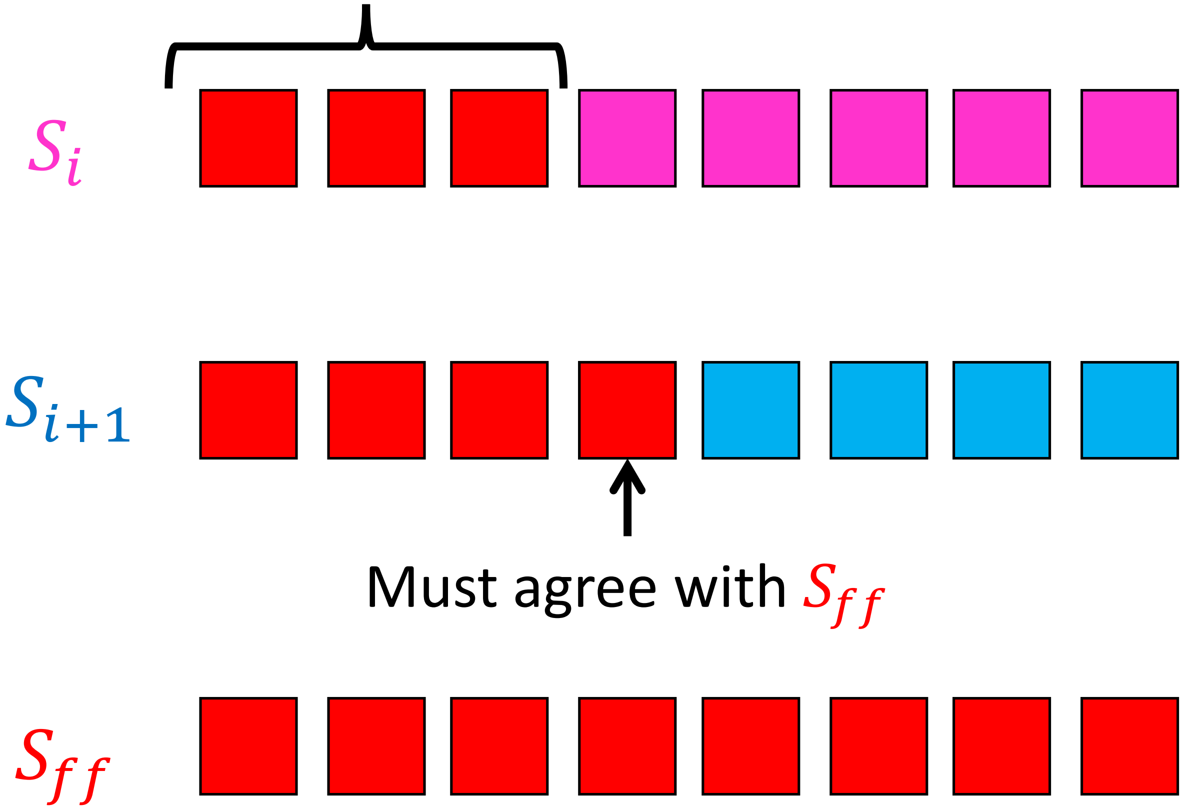
Consider access $m_{i+1} = d$

Case 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



First i accesses

S_i

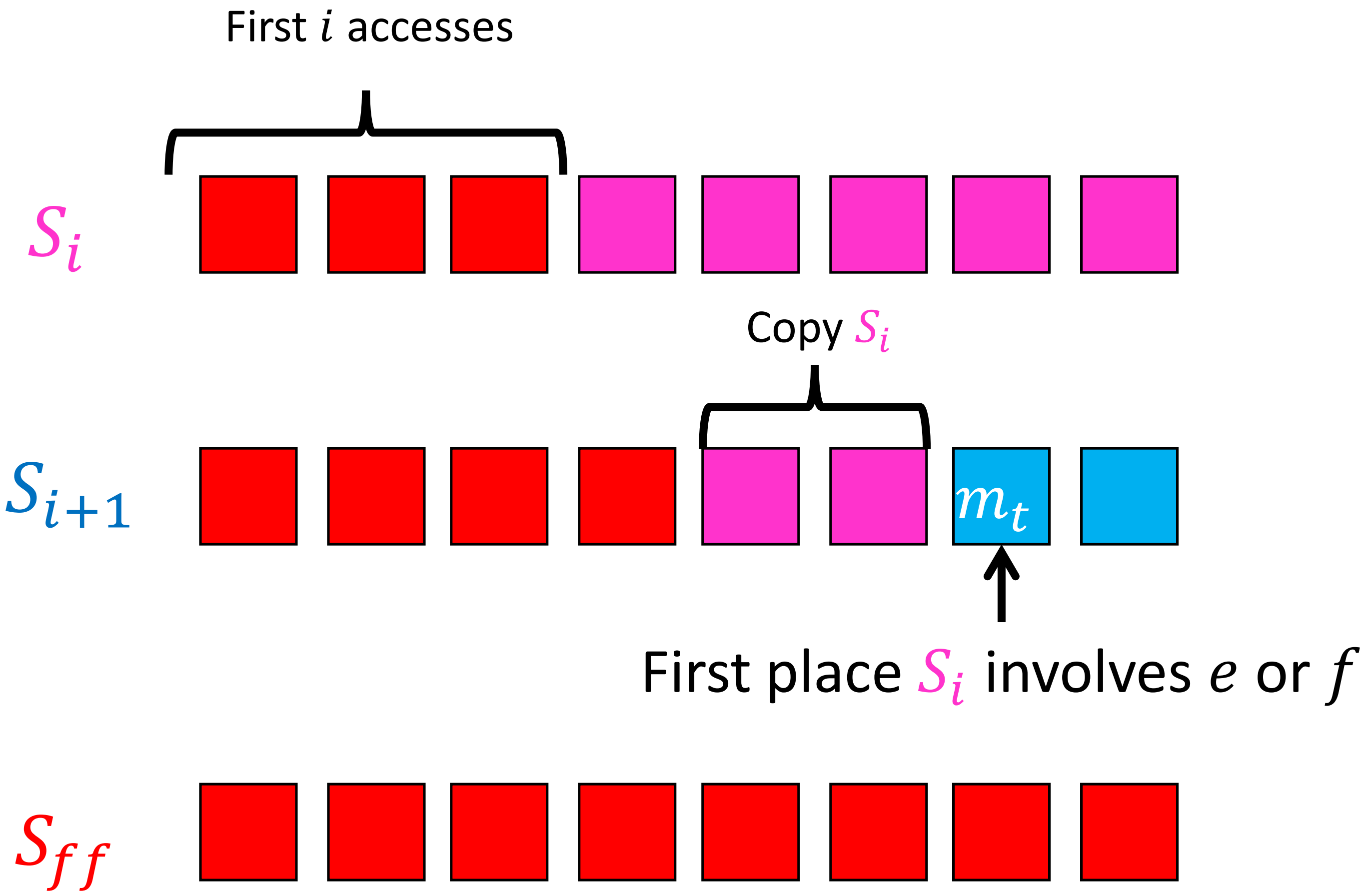
S_{i+1}

Must agree with S_{ff}

S_{ff}

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

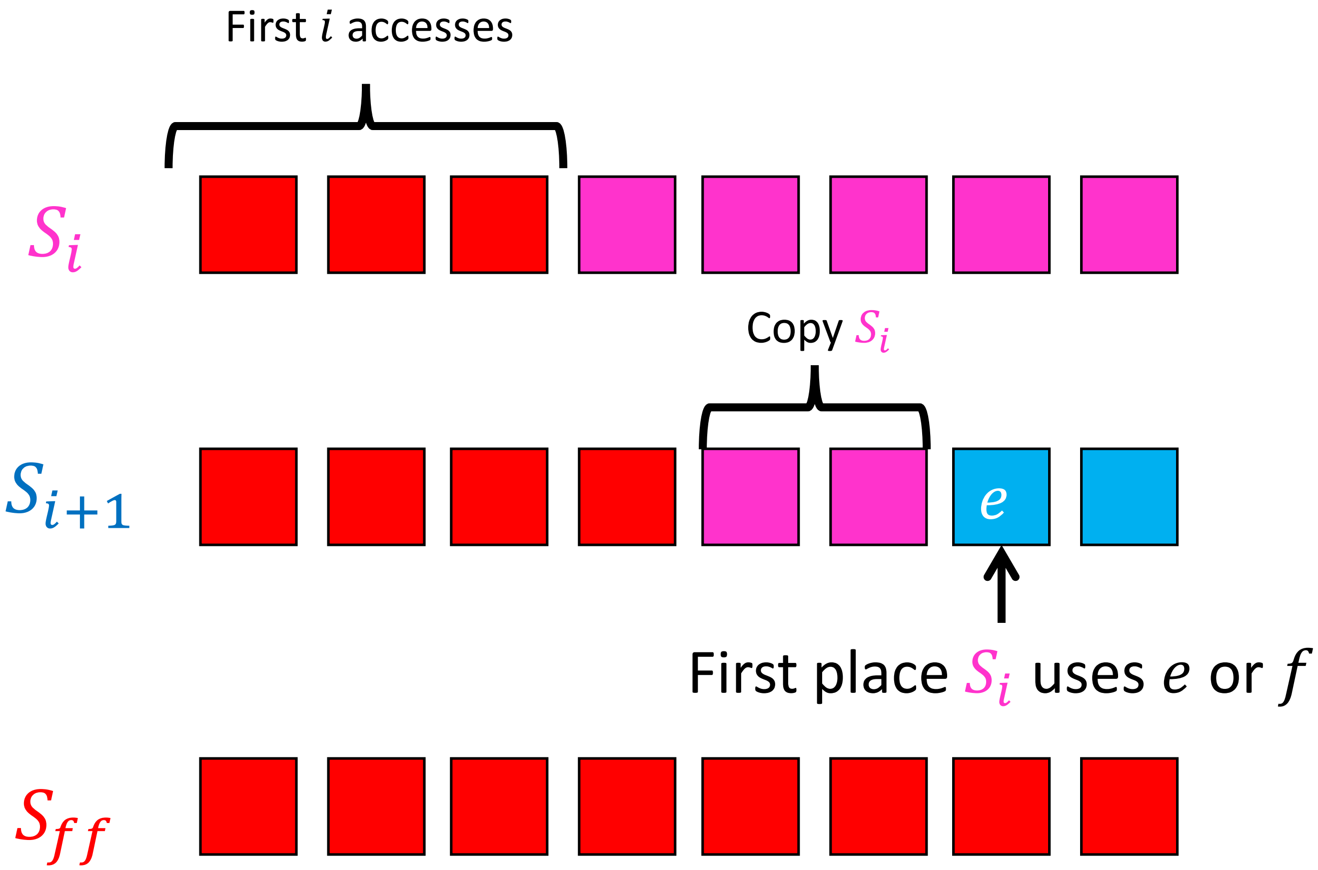
Case 3



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

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

Case 3, $m_t = e$

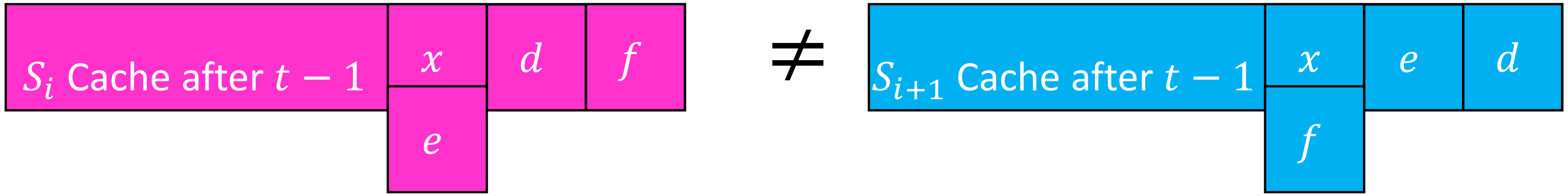


$m_t =$ the first access after $i + 1$ in which S_i deals with e or f

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

Case 3, $m_t = e$

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



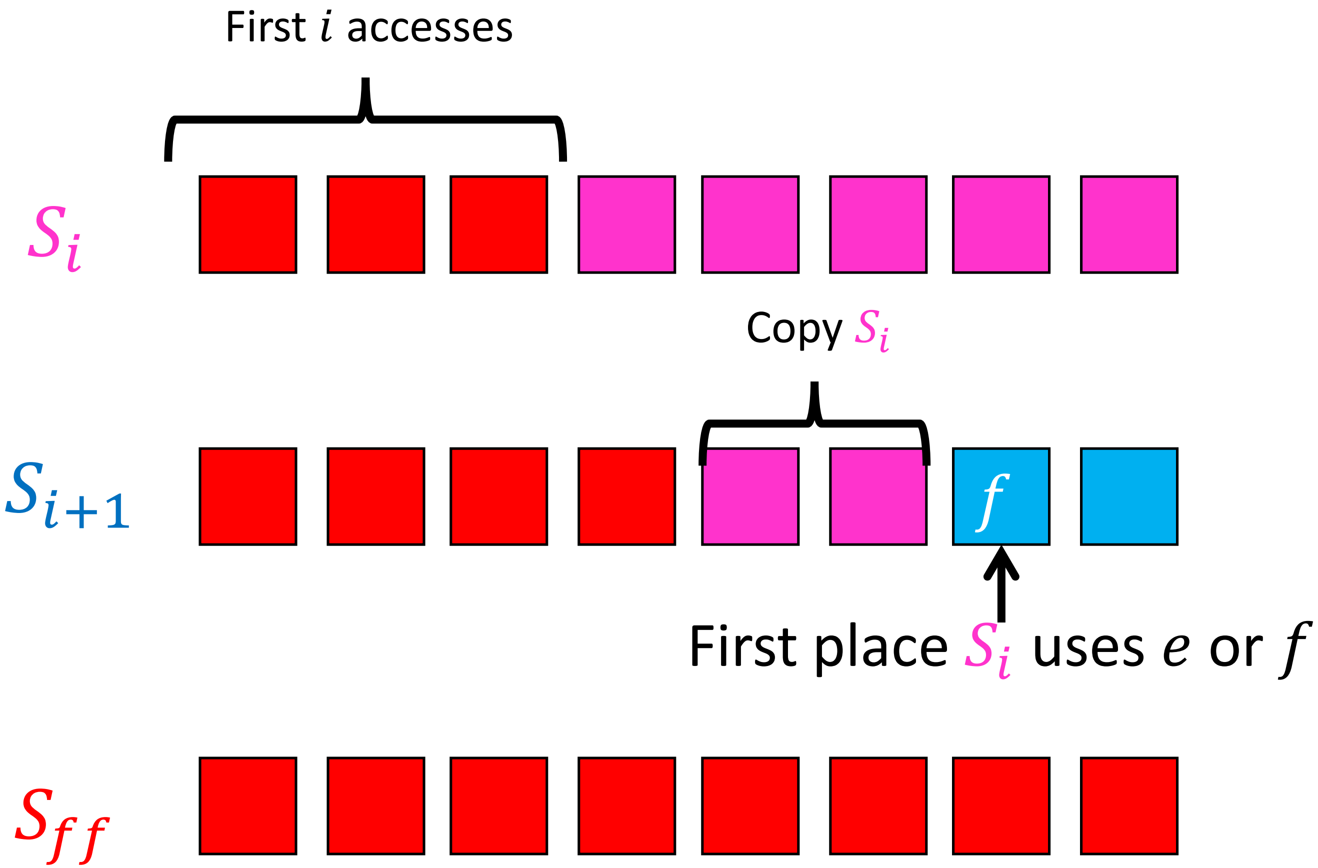
S_i must load e into the cache, assume it evicts x

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, $m_t = f$



First i accesses

S_i

Copy S_i

S_{i+1}

f

First place S_i uses e or f

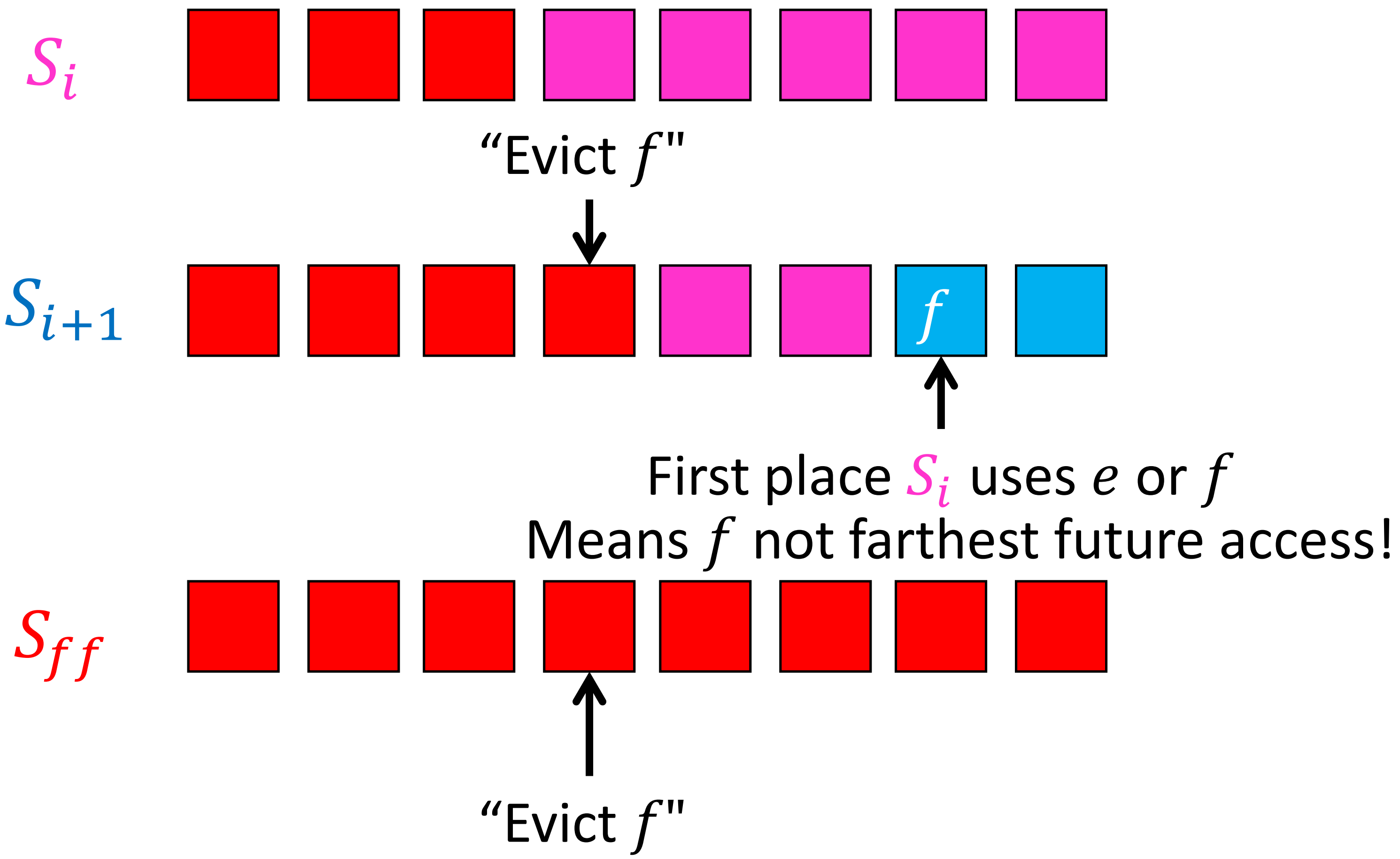
S_{ff}

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

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

Case 3, $m_t = f$

Cannot Happen!



S_i

"Evict f"

S_{i+1}

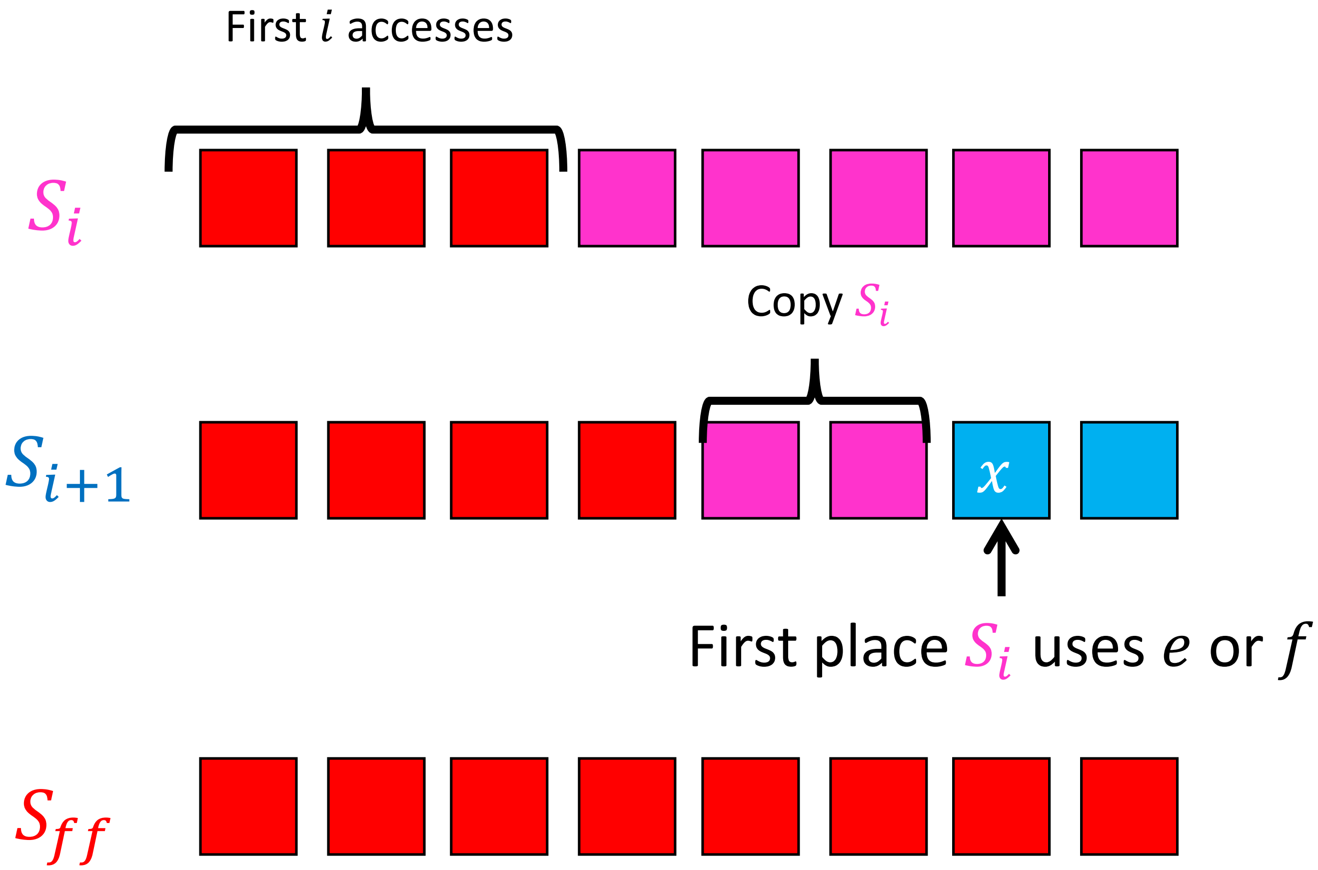
f

First place S_i uses e or f
Means f not farthest future access!

S_{ff}

"Evict f"

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

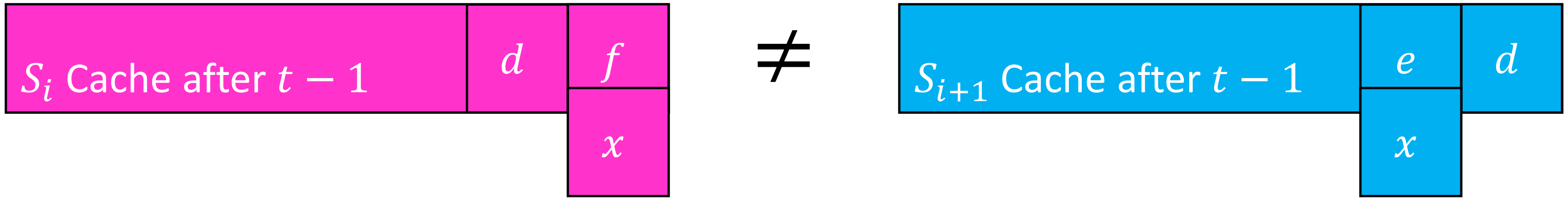


m_t = the first access after $i + 1$ in which S_i deals with e or f

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

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

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



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 $misses(S_{i+1}) = misses(S_i)$

Use Lemma to show Optimality

S^* $\xrightarrow{\text{Lemma}}$ S_1 $\xrightarrow{\text{Lemma}}$ S_2 $\xrightarrow{\text{Lemma}}$ … $\xrightarrow{\text{Lemma}}$ S_{ff}

Agrees with S_{ff} on first 0 accesses

Agrees with S_{ff} on first access

Agrees with S_{ff} on first 2 accesses

Agrees with S_{ff} on all n accesses