<u>Warm up</u>

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"

limited memory

different kinds of memory

speed of memory – cache

memory = slow (?)

fast Memory = $$$

memory ≤ time

Today's Keywords

- Greedy Algorithms
- Choice Function
- Cache Replacement
- Hardware & Algorithms

CLRS Chapter 16

Announcements

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

REVIEW: 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

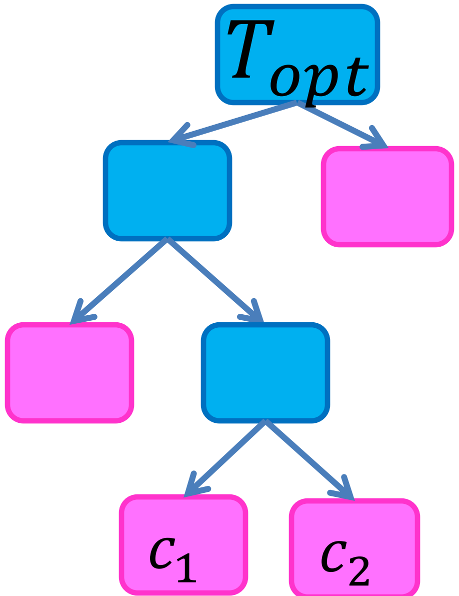
Greedy Choice Property

Optimal Substructure works

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

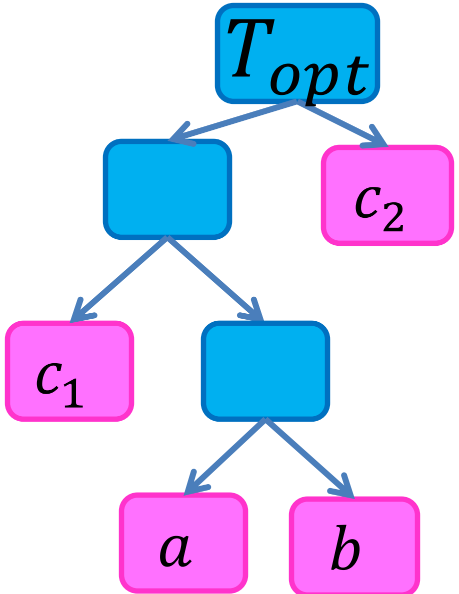
Case 1: Consider some optimal tree T_{opt}. If c_1, c_2 are siblings in this tree, then claim holds



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



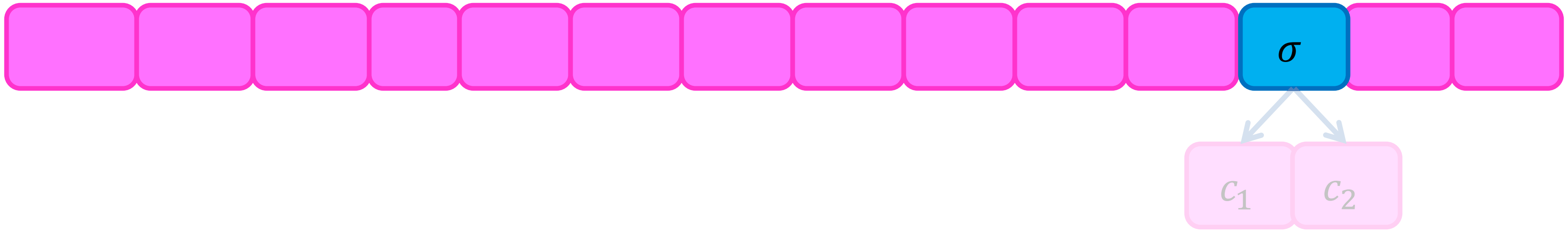
Let a, b be the two characters of lowest depth that are siblings
(Why must they exist?)

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 $f_{c2} \leq f_b$

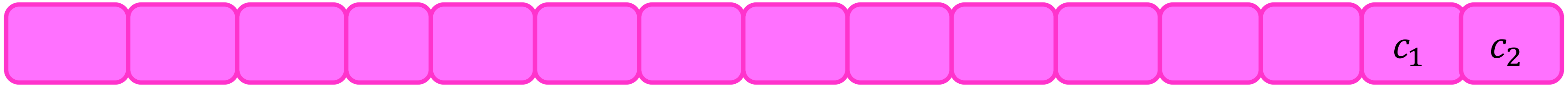
Finishing the Proof

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

Why does solving this smaller problem:

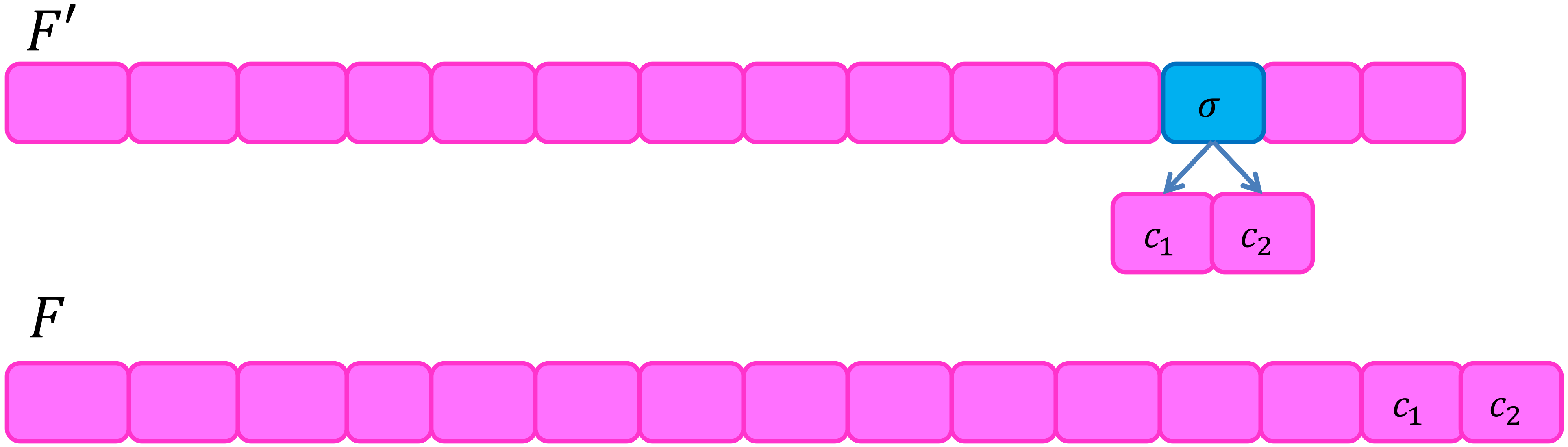


Give an optimal solution to this?:



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

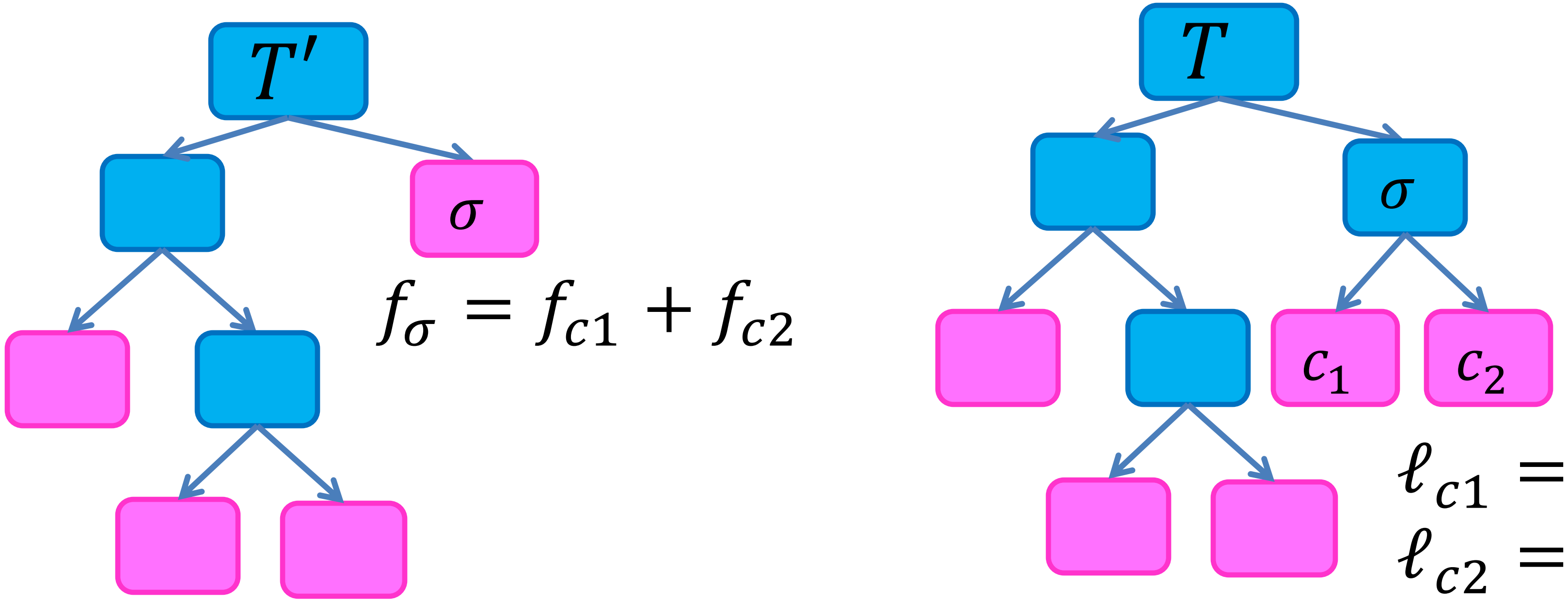


Optimal Substructure

- 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

Then this is optimal



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

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

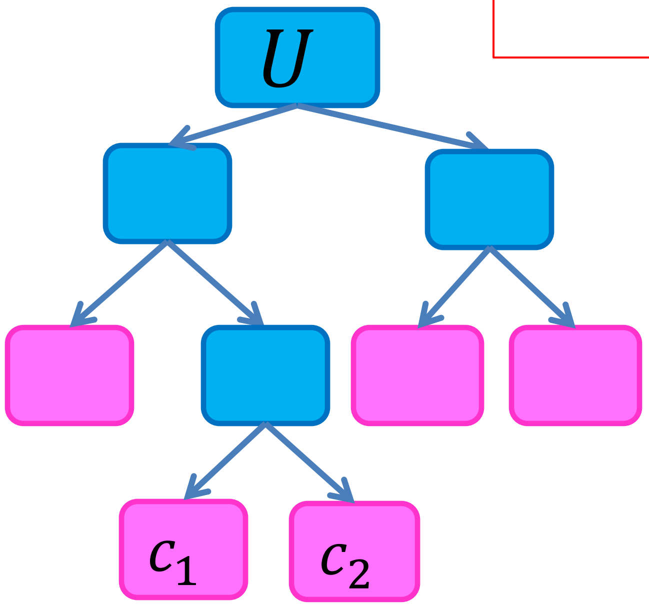
$$B(T') = B(T) - f_{c1} - f_{c2}$$

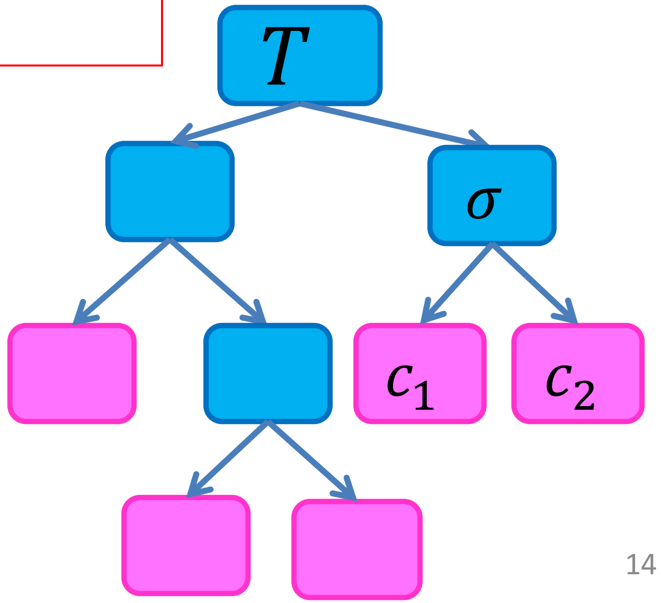
Optimal Substructure

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

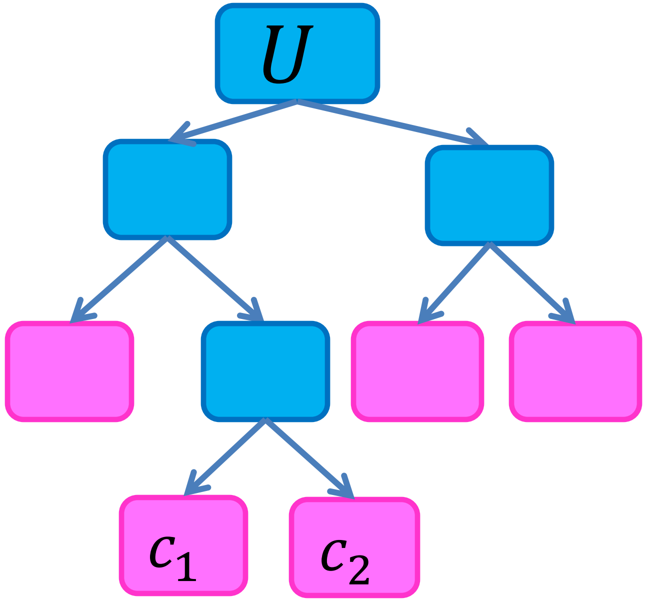




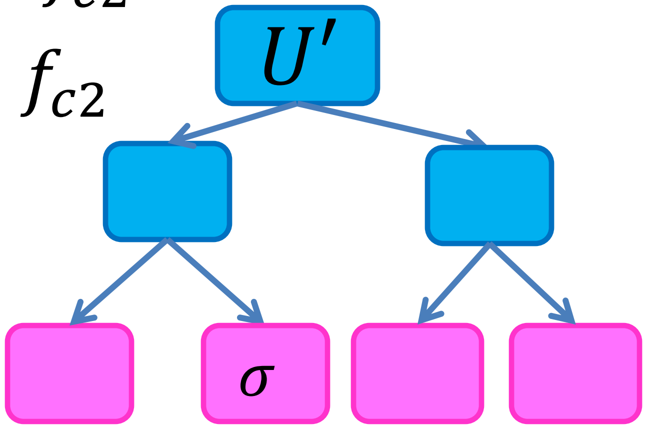
Optimal Substructure

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

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



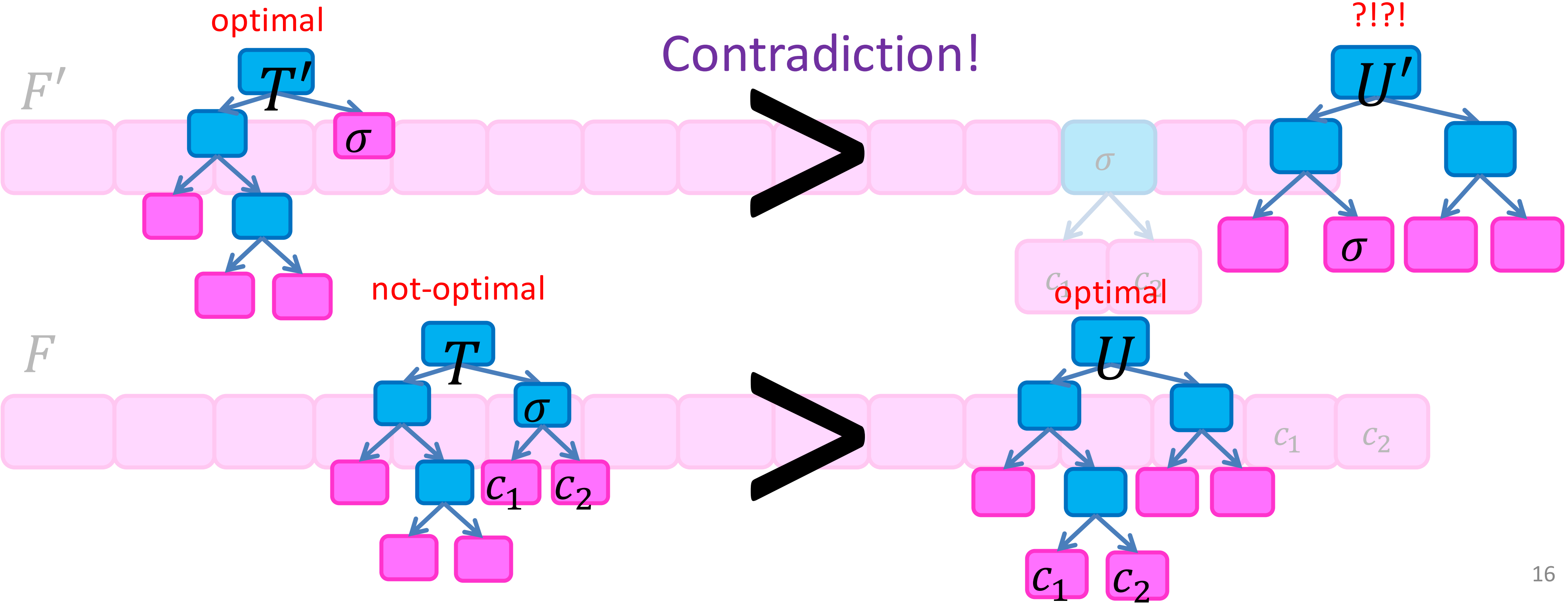
$$\begin{aligned} B(U') &= B(U) - f_{c1} - f_{c2} \\ &< B(T) - f_{c1} - f_{c2} \\ &= B(T') \end{aligned}$$



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



Caching Problem

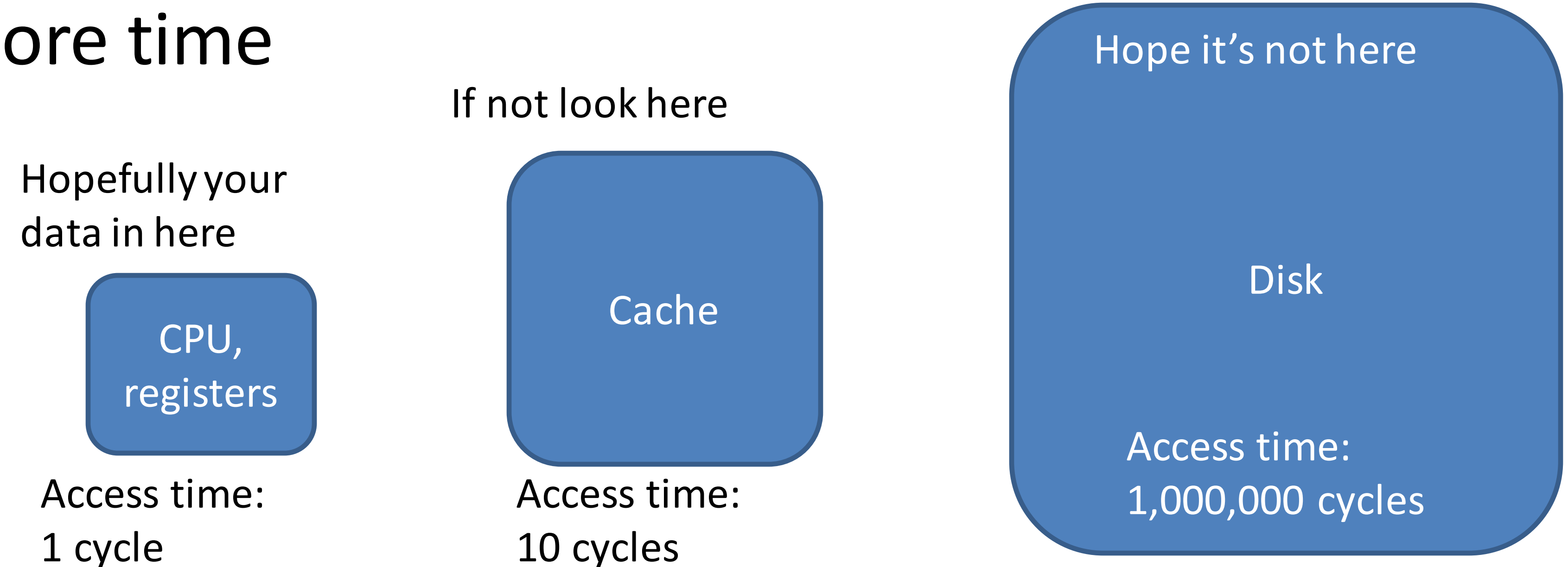
- Why is using too much memory a bad thing?

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

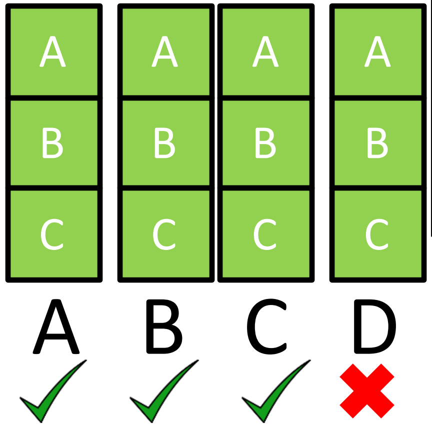
- Input:
  - k = size of the cache
  - $M = [m_1, m_2, \ldots m_n]$ = memory access pattern
- Output:
  - "schedule" for the cache (list of items in the cache at each time) which minimizes cache fetches

Example

Example

Example

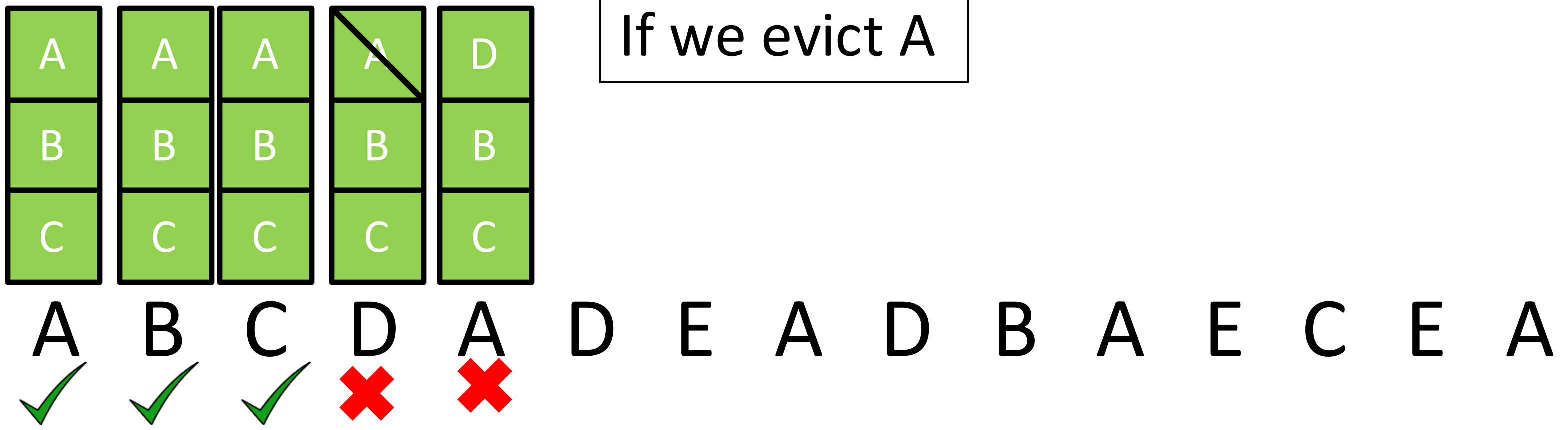
Example



We must evict something to make room for D

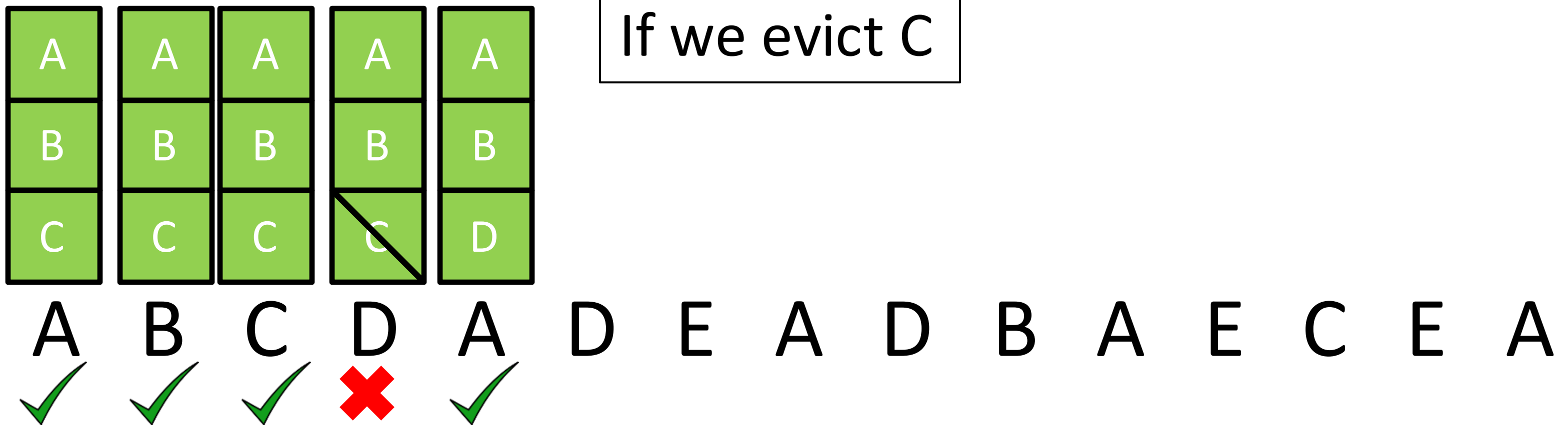
A B C D A D E A D B A E C E A

Example



If we evict A

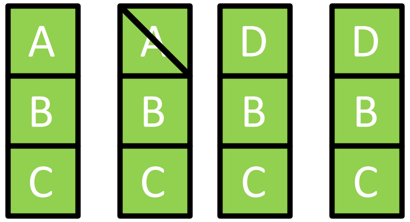
Example



If we evict C

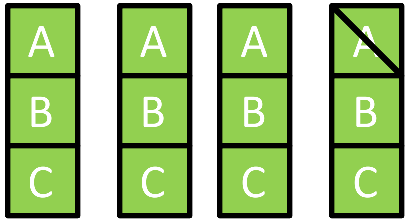
Our Problem vs Reality

- 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

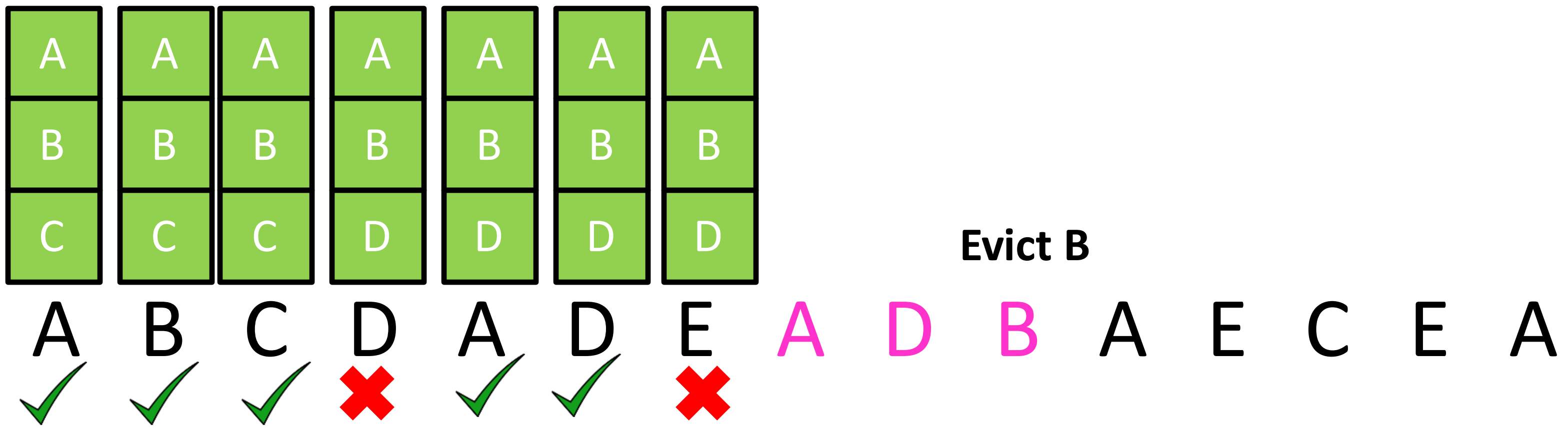
- Require Optimal Substructure
  - Solution to larger problem contains the solution to a smaller one
  - Only one subproblem to consider!
- Idea:
  1. Identify a greedy choice property
     - How to make a choice guaranteed to be included in some optimal solution
  2. Repeatedly apply the choice property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

Greedy choice property

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

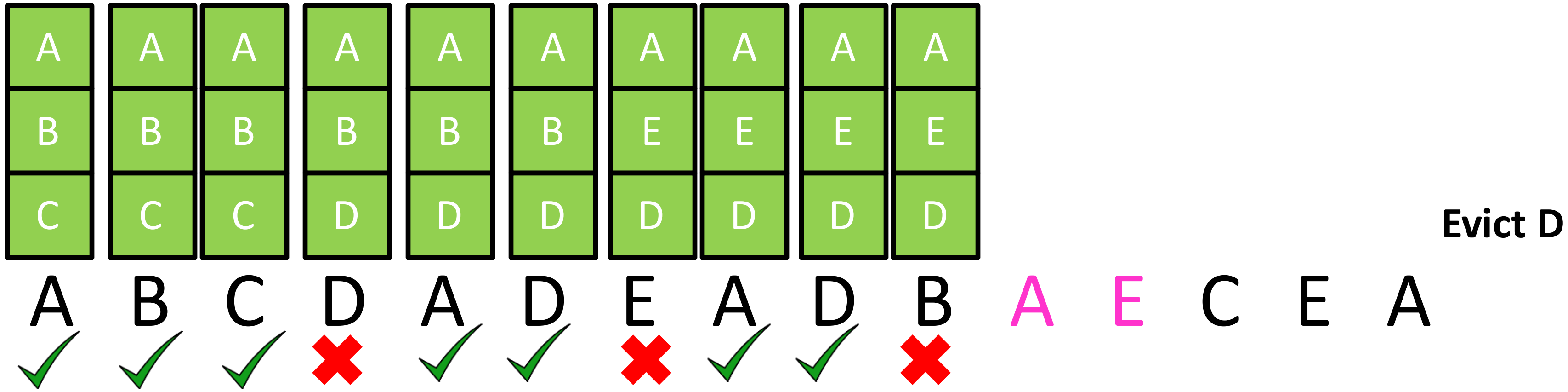
Greedy choice property

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



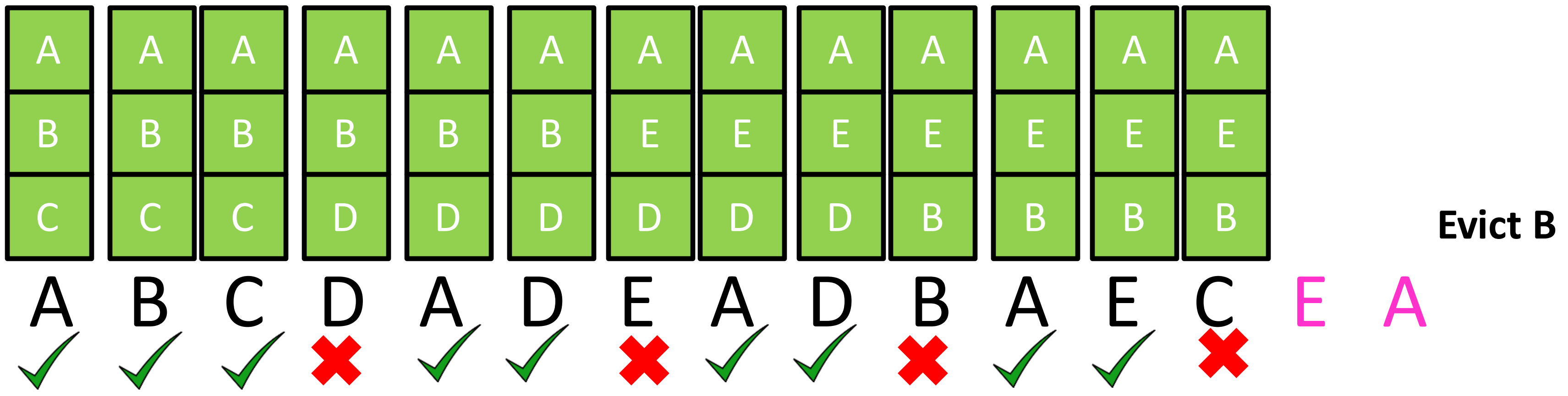
Greedy choice property

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



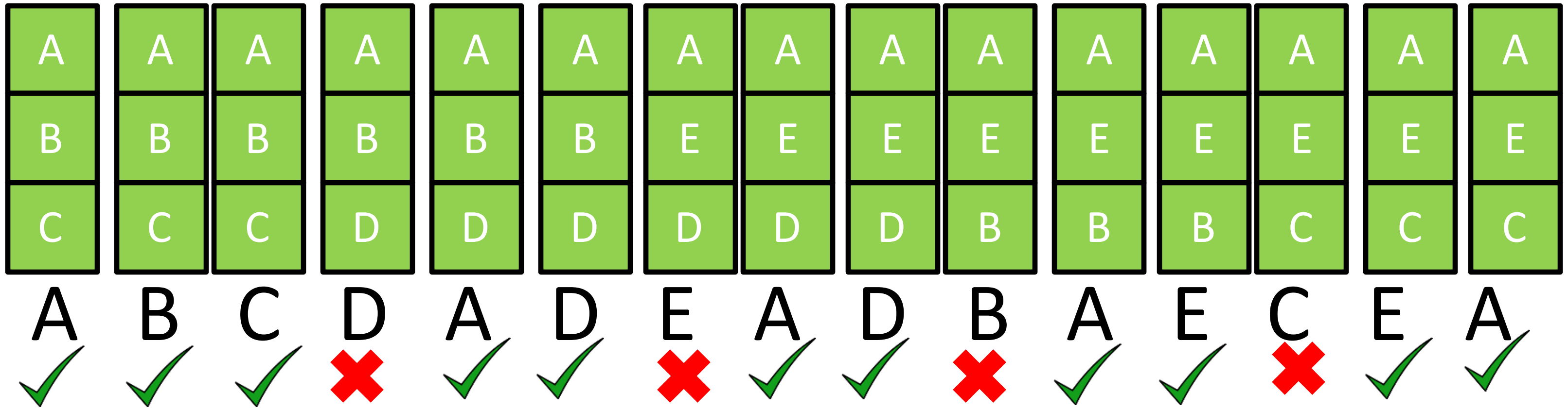
Greedy choice property

- 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



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

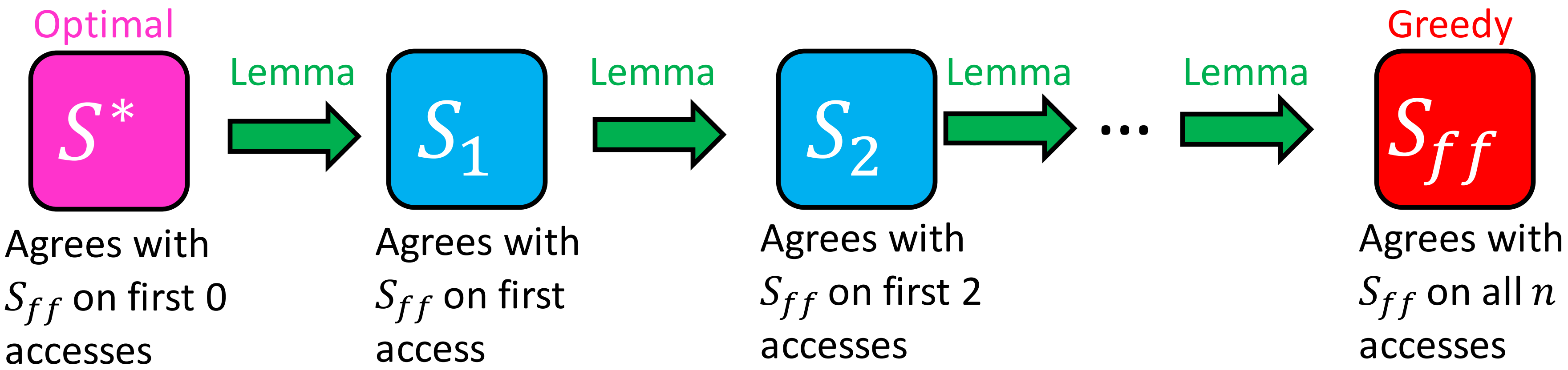
- Shows correctness of a greedy algorithm
- Idea:
  - Show exchanging an item from an arbitrary optimal solution with your greedy choice makes the new solution no worse
  - How to show my sandwich is at least as good as yours:
    - Show: “I can remove any item from your sandwich, and it would be no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

Belady Exchange Lemma

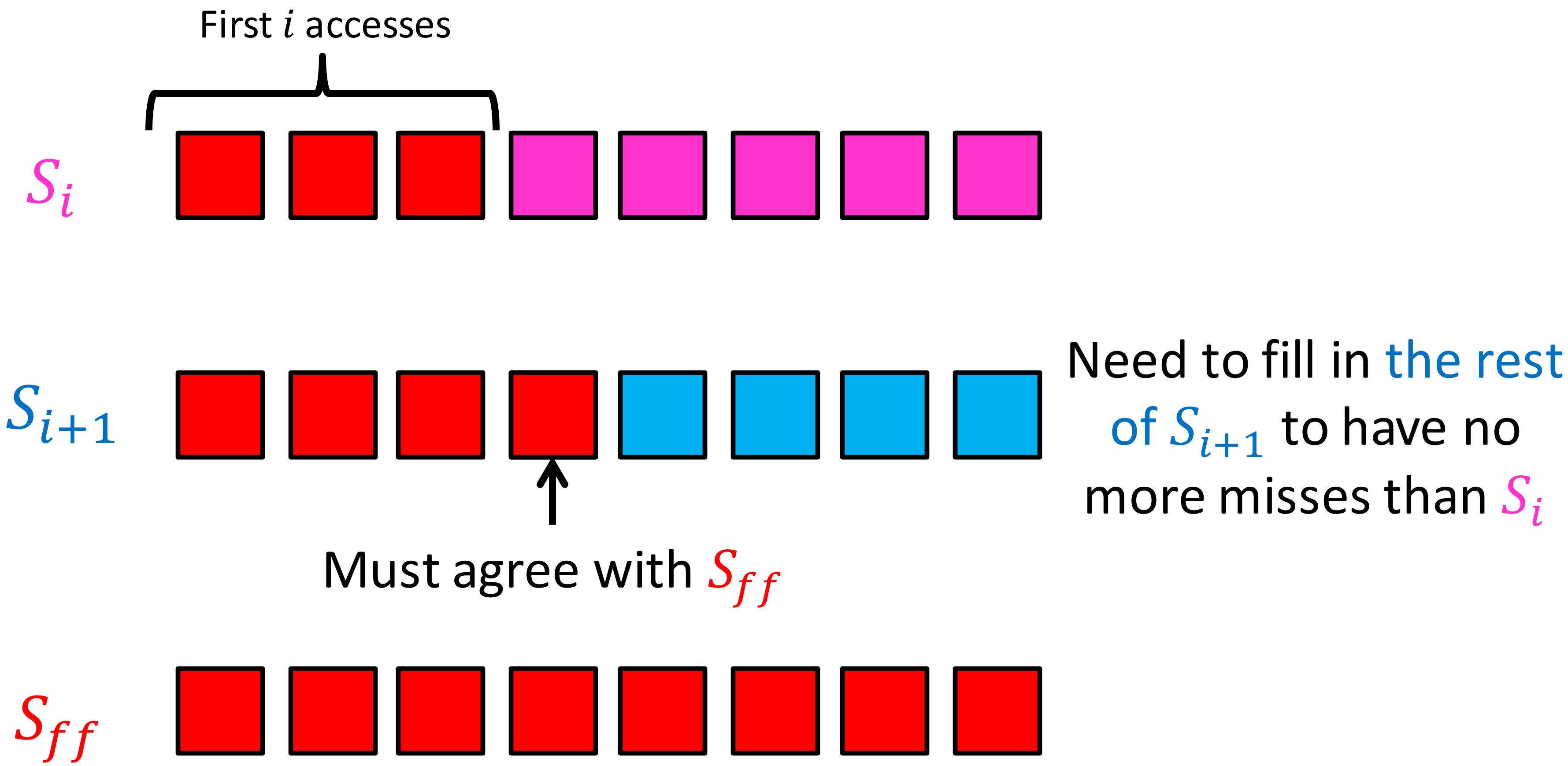
Let S_{ff} be the schedule chosen by our greedy algorithm

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



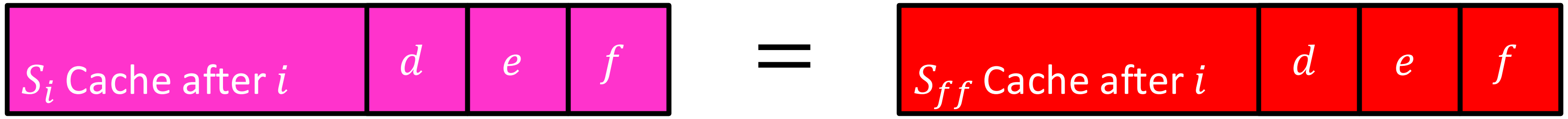
Belady Exchange Proof Idea



Proof of Lemma

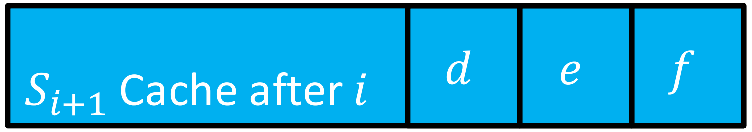
Goal: find S_{i+1} s.t. $misses(S_{i+1}) \leq misses(S_i)$

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 $m_{i+1} = d$

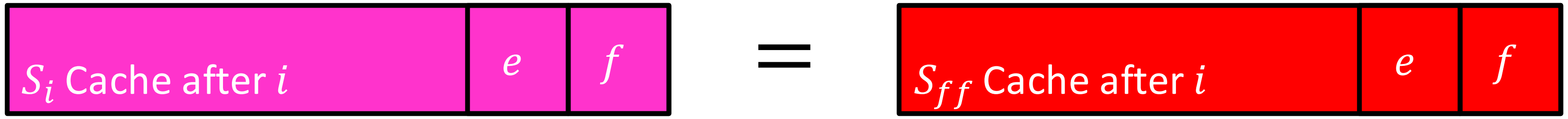
Case 1: if d is in the cache, then neither S_i nor S_{ff} evict from the cache, use the same cache for S_{i+1}



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

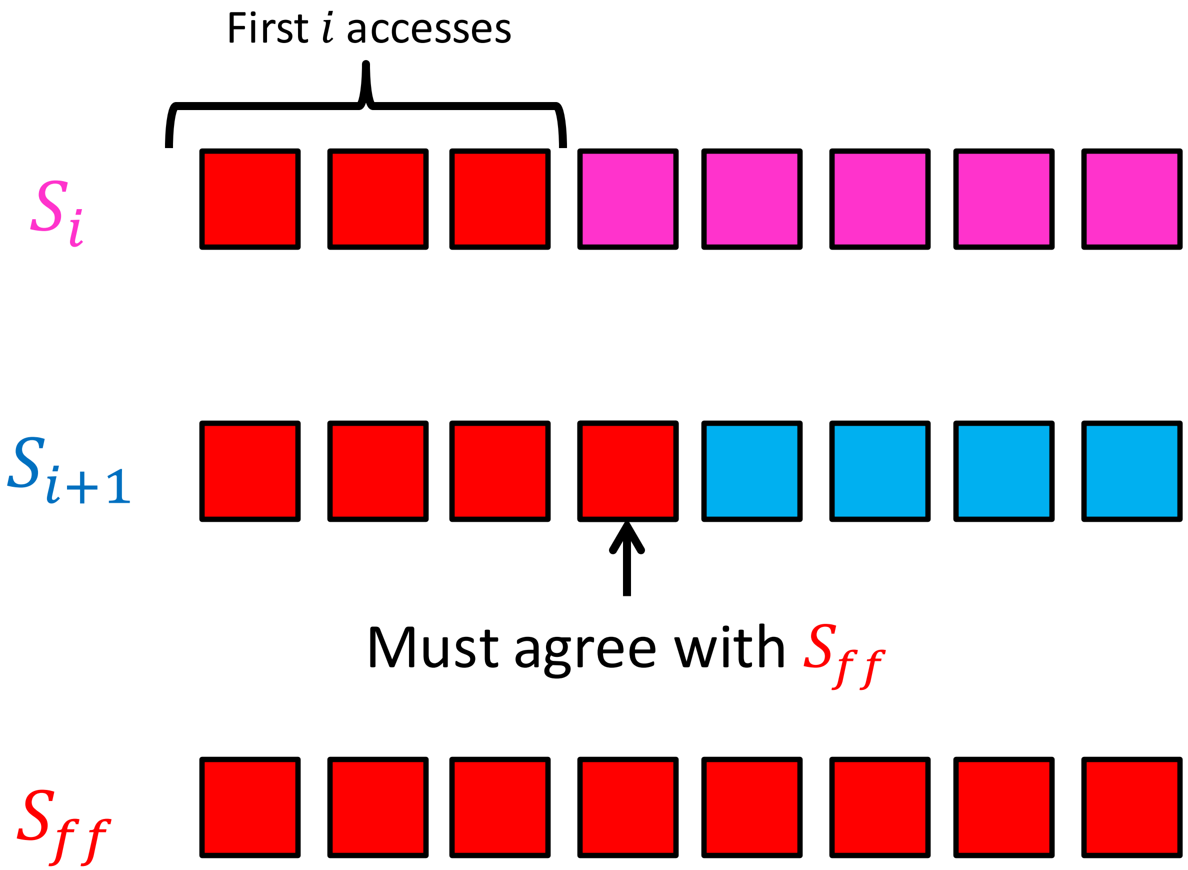
S_i Cache after i	e	f	$=$	S_{ff} Cache after i	e	f

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

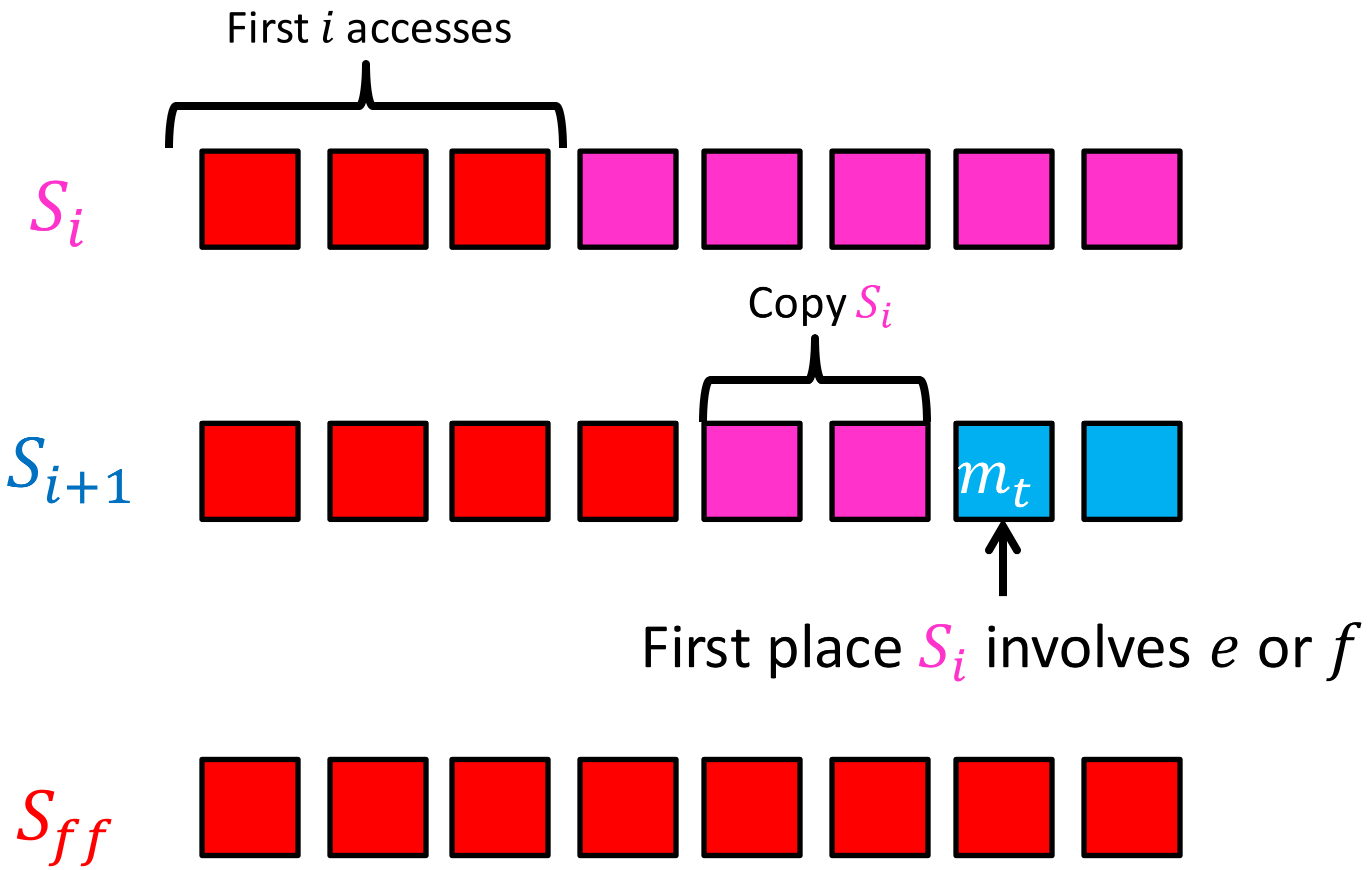
S_i Cache after $i+1$	d	f	$\neq$	S_{ff} Cache after $i+1$	e	d

Case 3



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

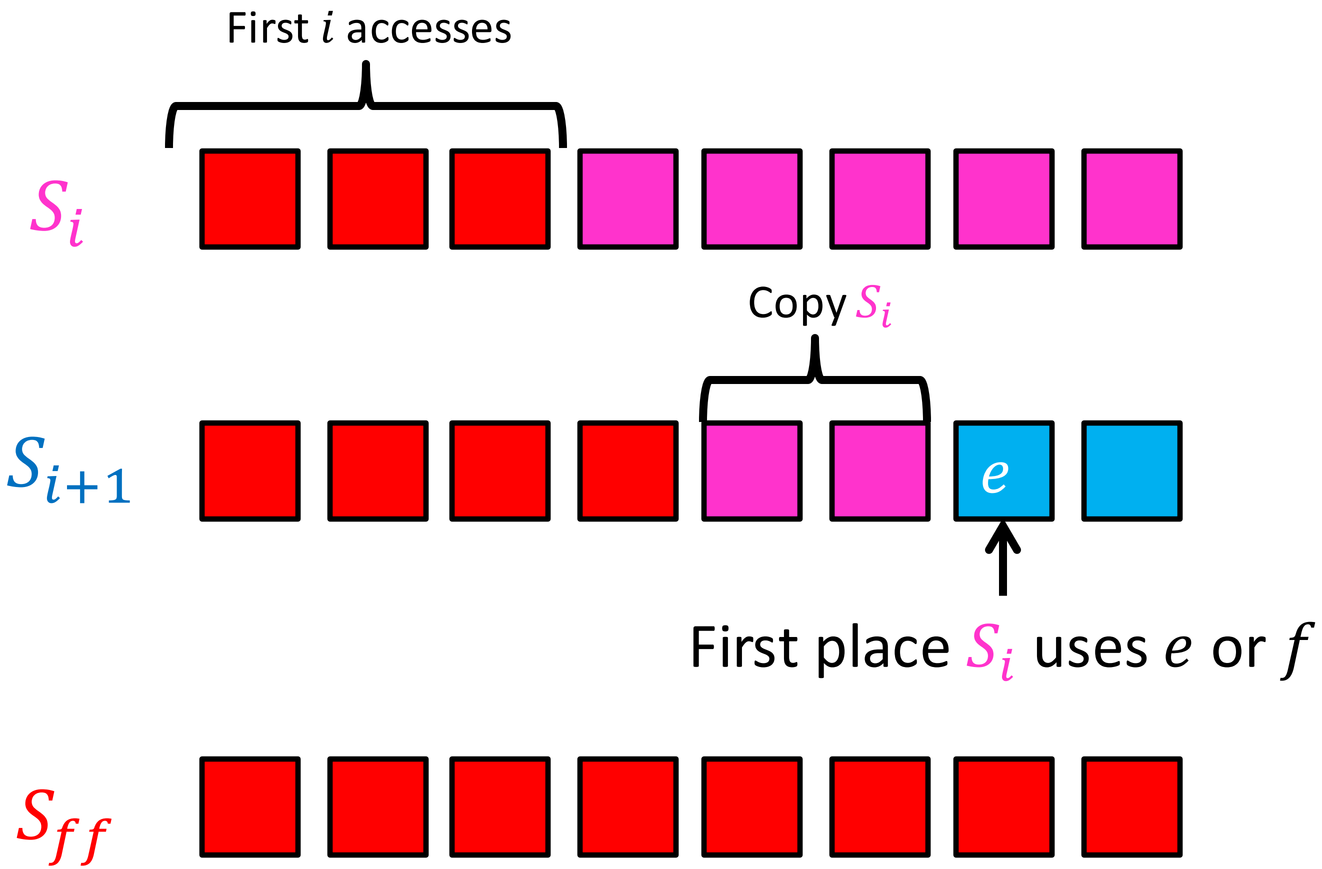
Case 3



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, $m_t = e$

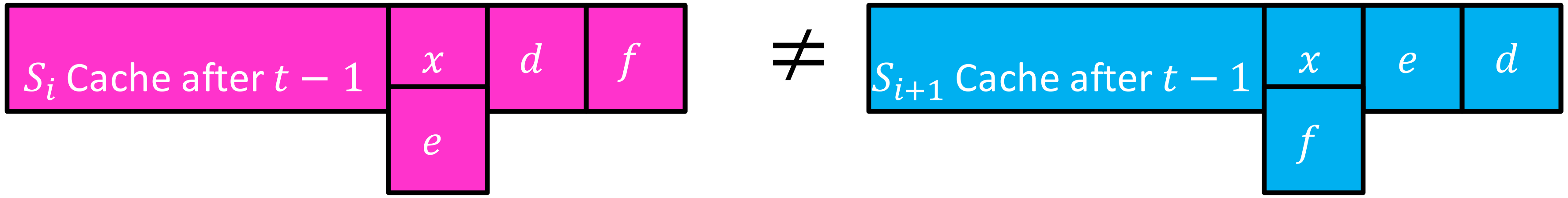


m_t = the first access after $i+1$ in which S_i deals with e or f

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

Case 3, $m_t = e$

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



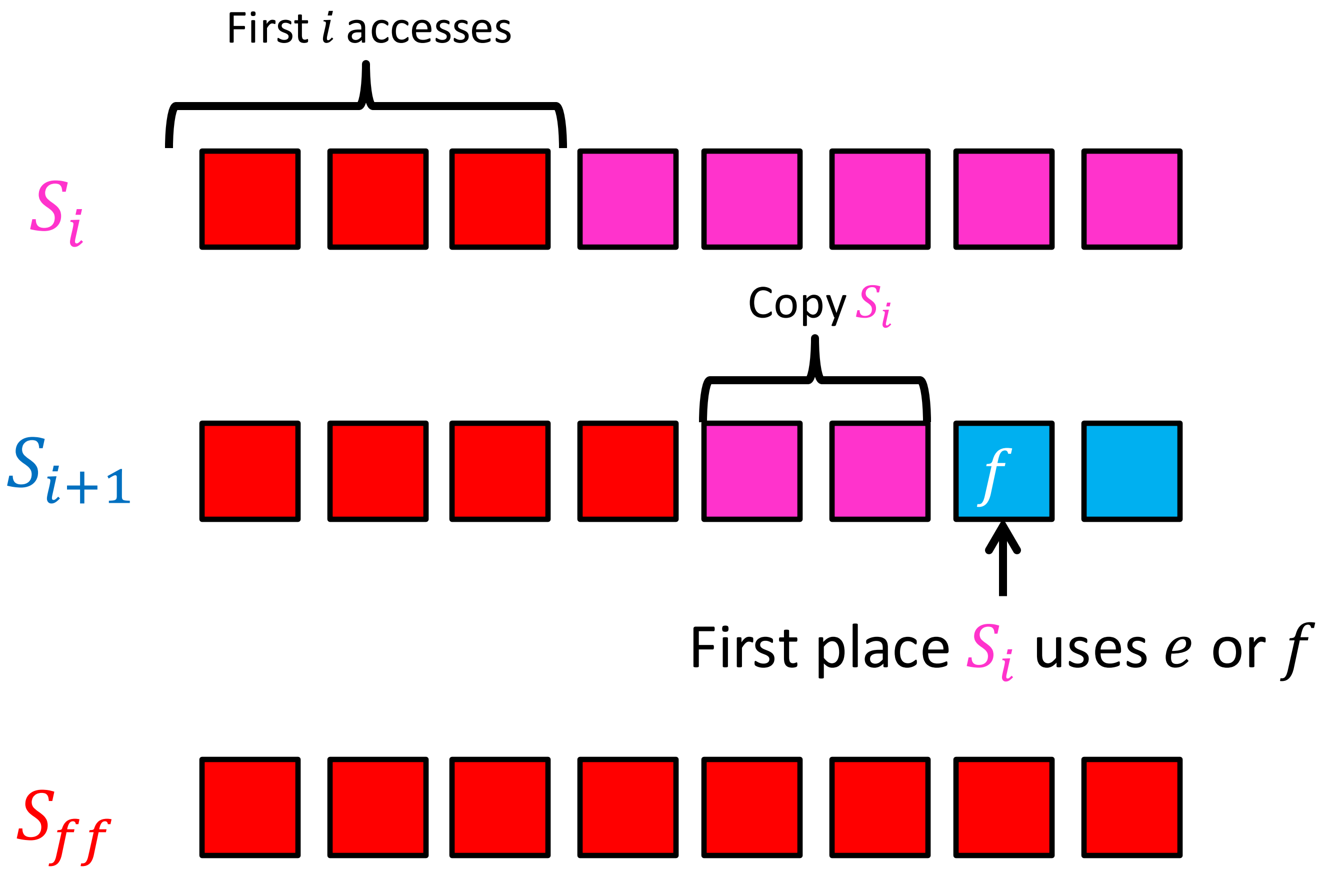
S_i must load e into the cache, assume it evicts x

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, $m_t = f$

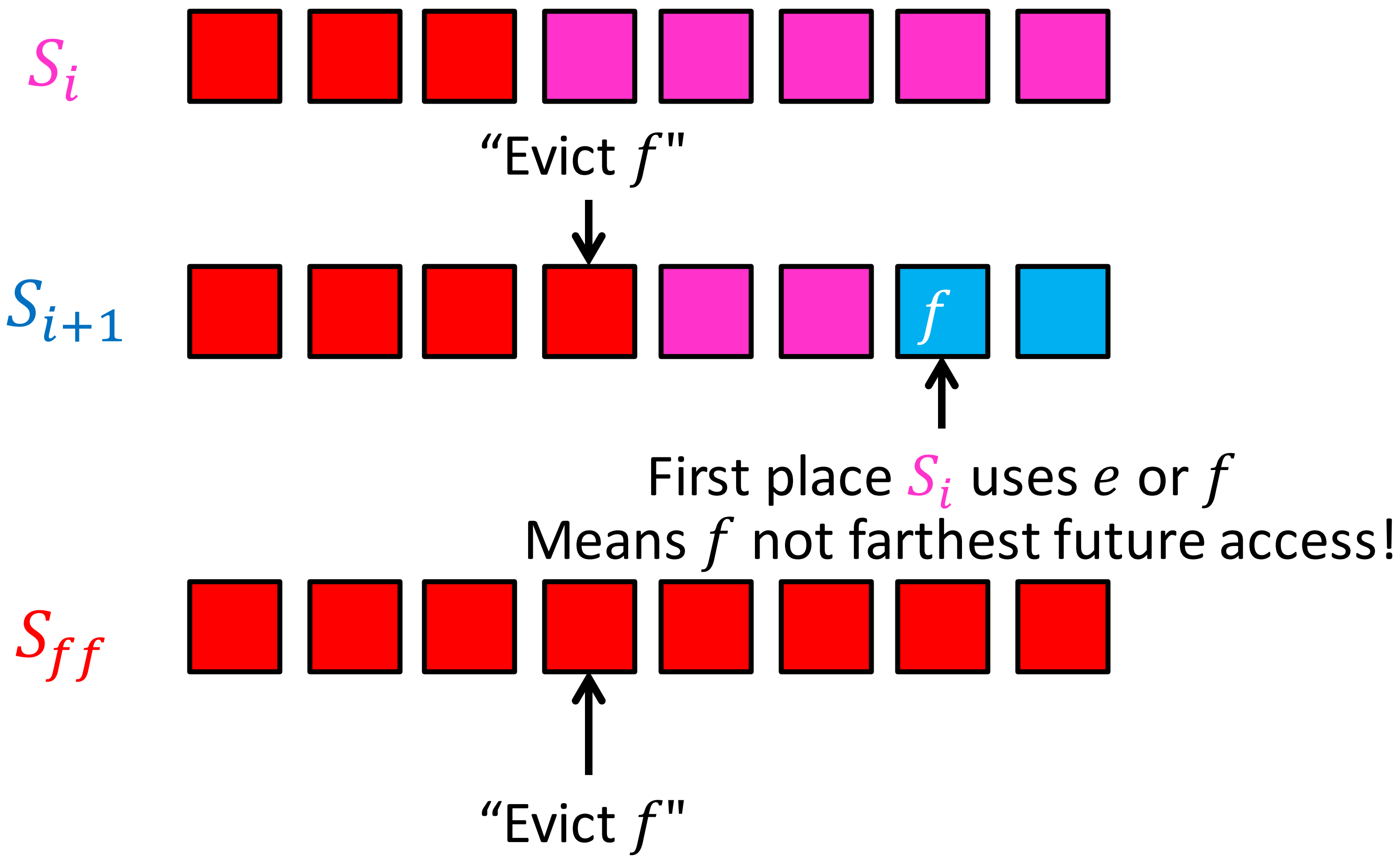


m_t = the first access after $i + 1$ in which S_i deals with e or f

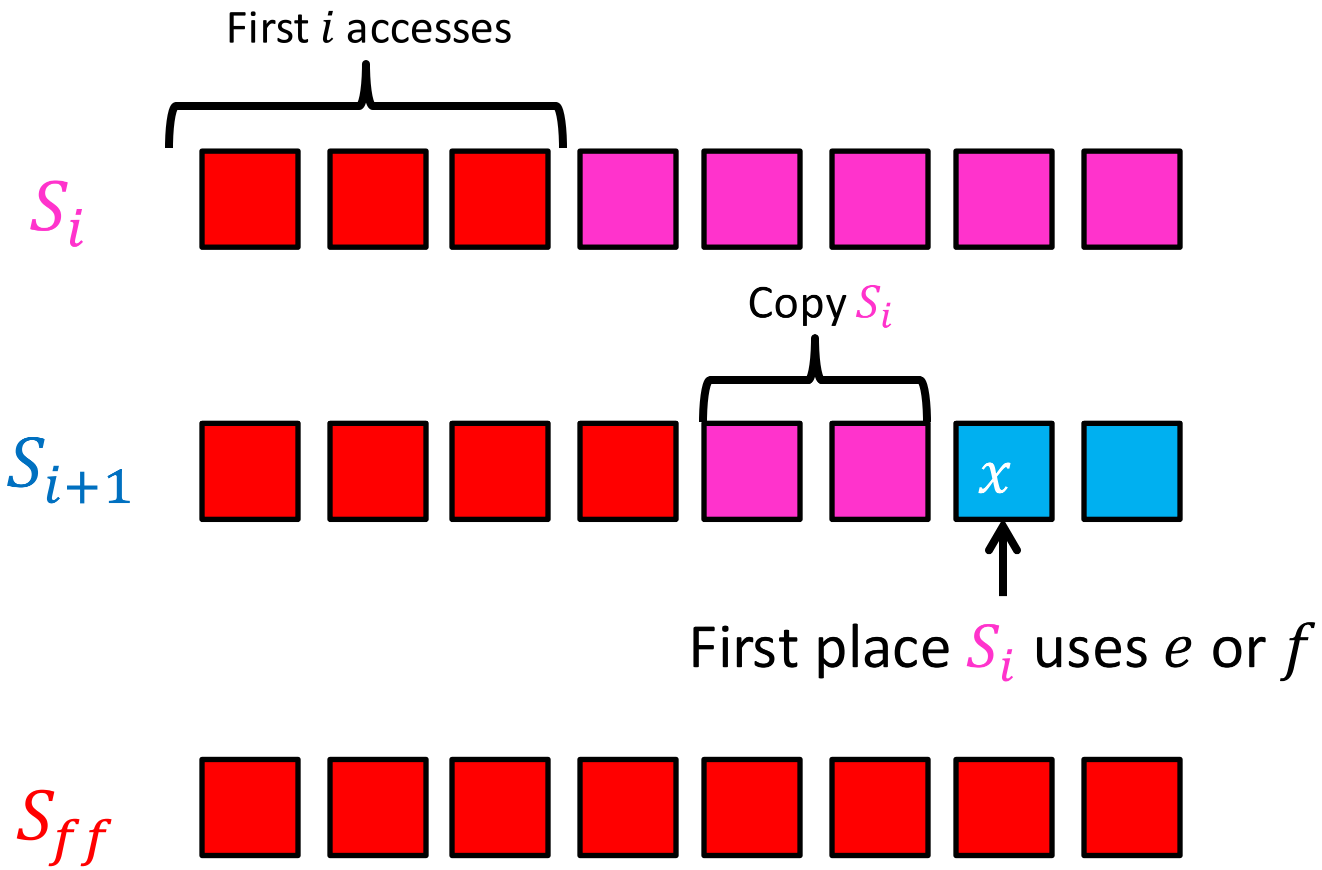
3 options: $m_t = e$ or $\boldsymbol{m_t = f}$ or $m_t = x \neq e, f$

Case 3, $m_t = f$

Cannot Happen!



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

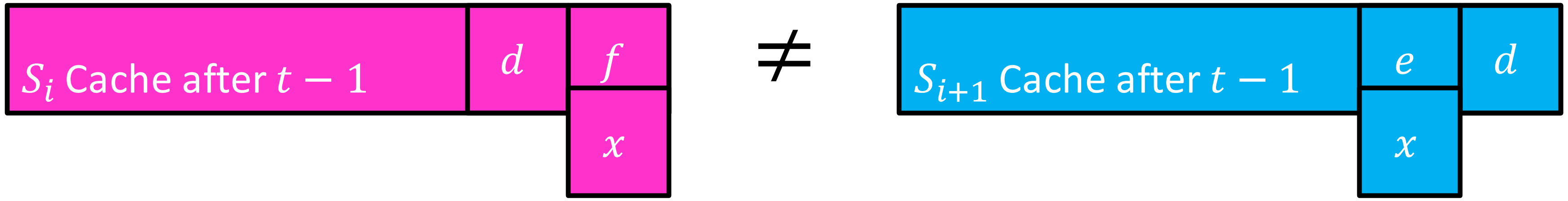


m_t = the first access after $i+1$ in which S_i deals with e or f

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



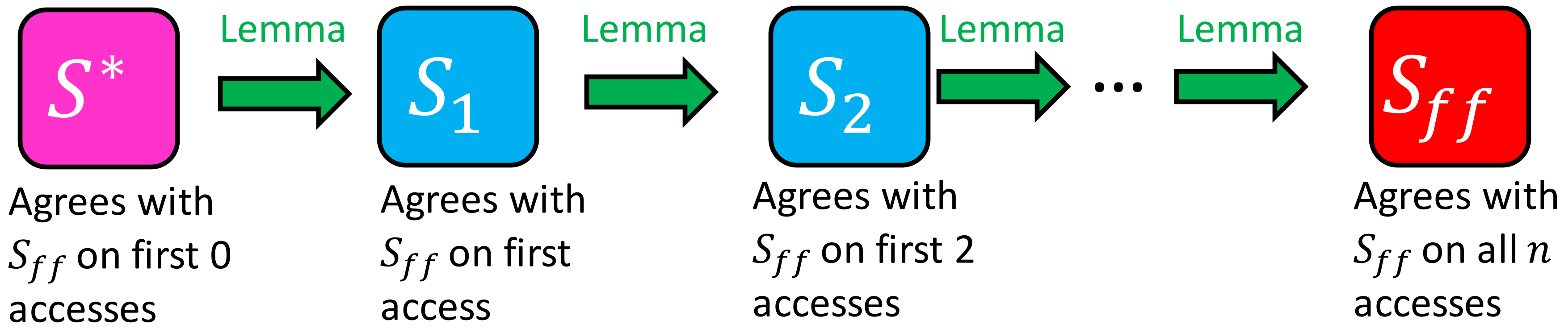
S_i loads x into the cache, it must be evicting f

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

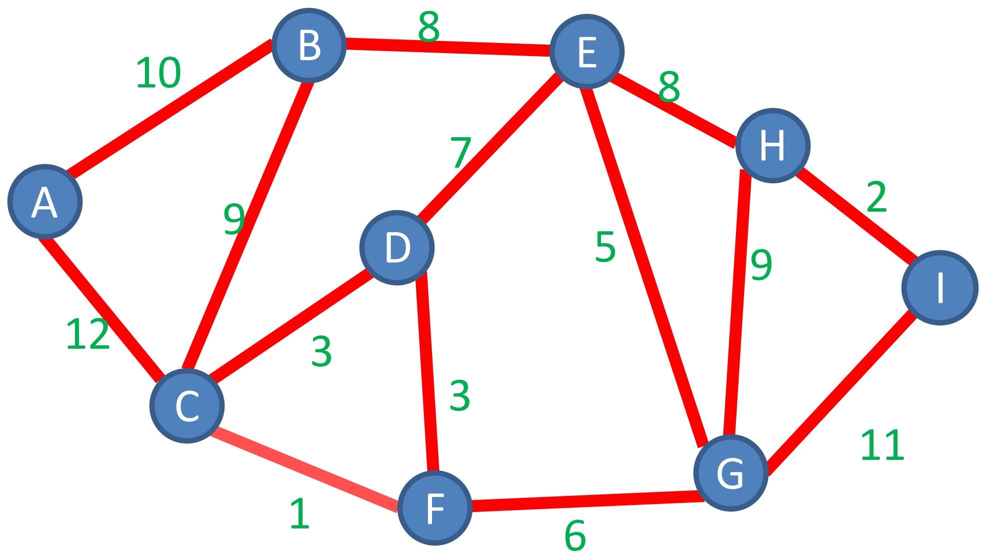
Use Lemma to show Optimality



Kruskal's Algorithm

Start with an empty tree A

Add to A the lowest-weight edge that does not create a cycle



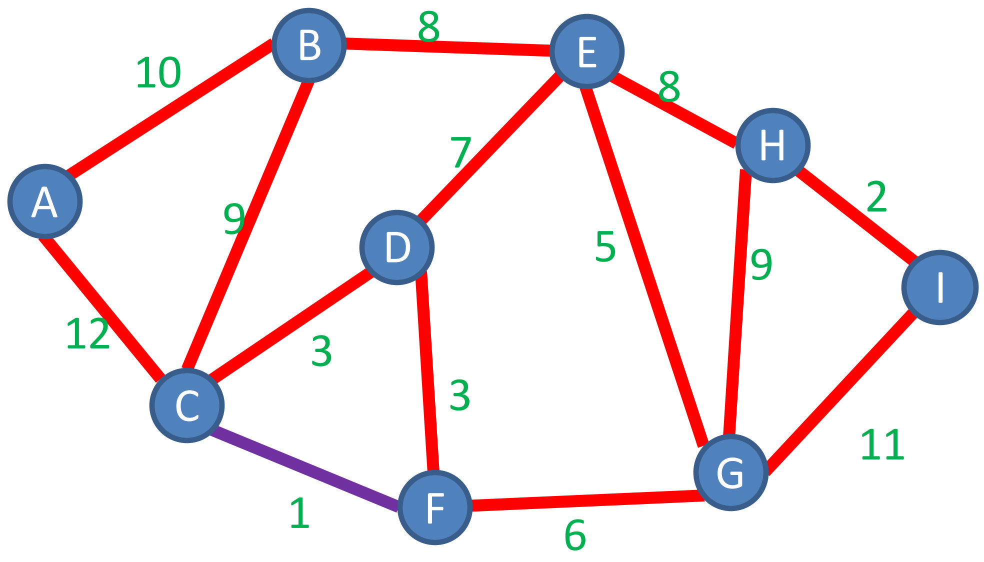
Greedy Algorithms

- 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

Kruskal's Algorithm

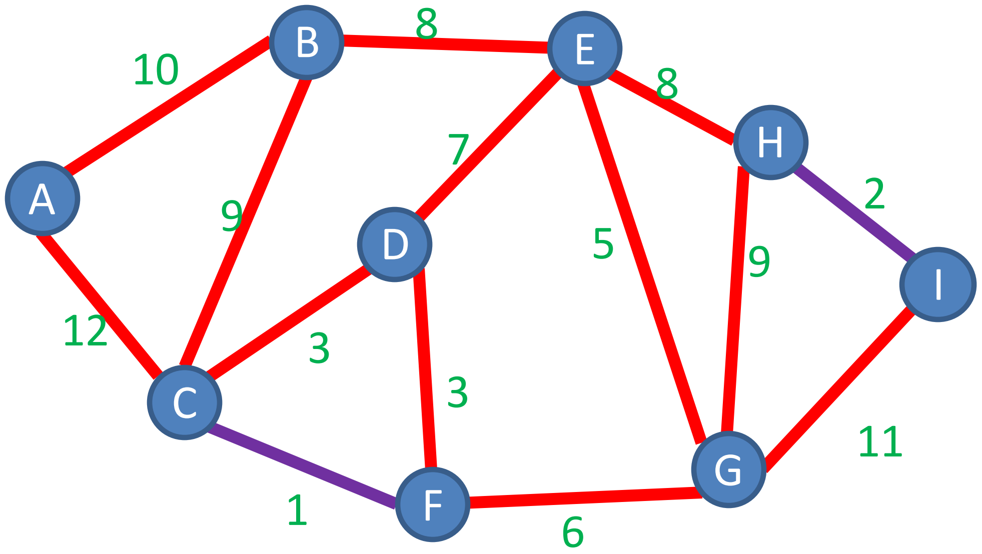
Start with an empty tree A

Add to A the lowest-weight edge that does not create a cycle



Kruskal's Algorithm

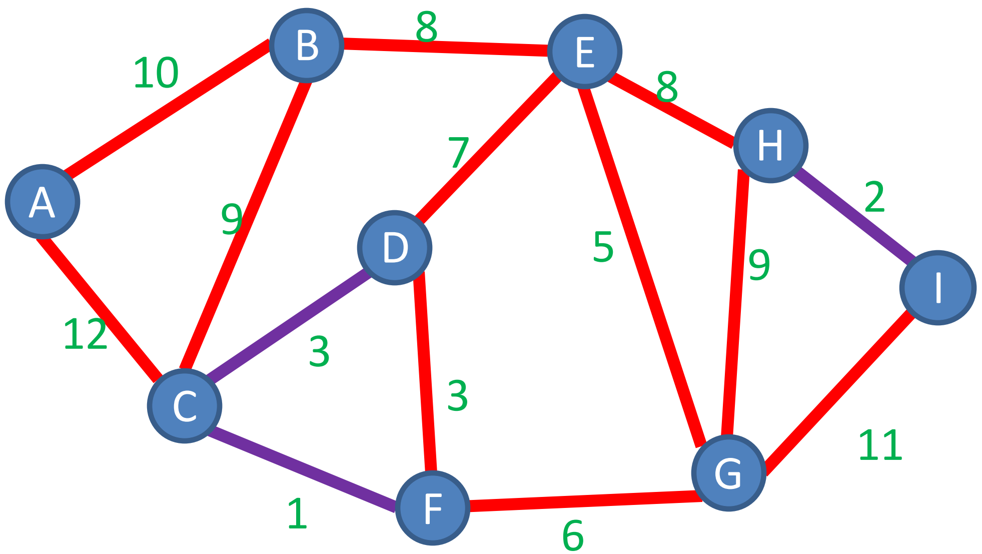
Start with an empty tree A
Add to A the lowest-weight edge that does not create a cycle



Kruskal's Algorithm

Start with an empty tree A

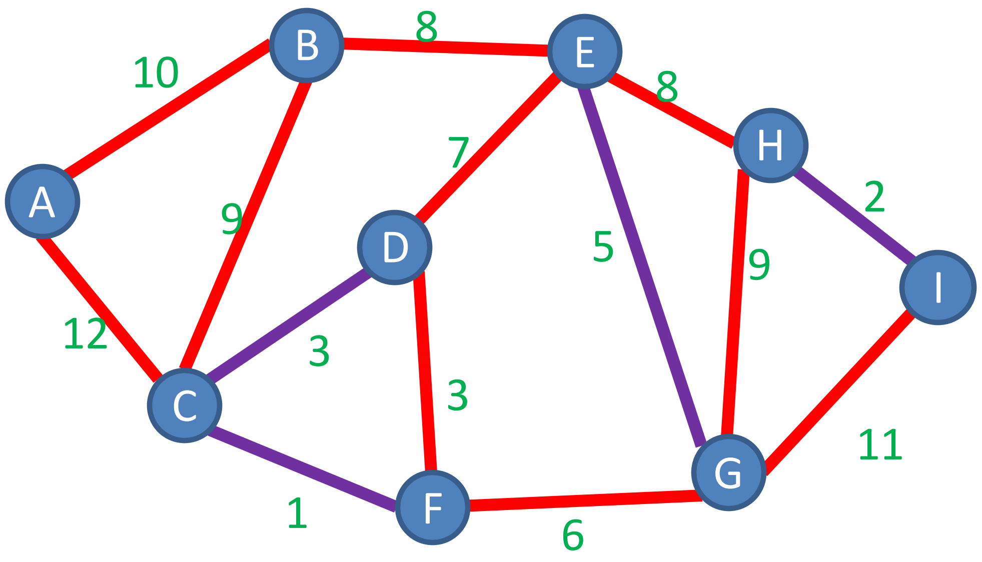
Add to A the lowest-weight edge that does not create a cycle



Kruskal's Algorithm

Start with an empty tree A

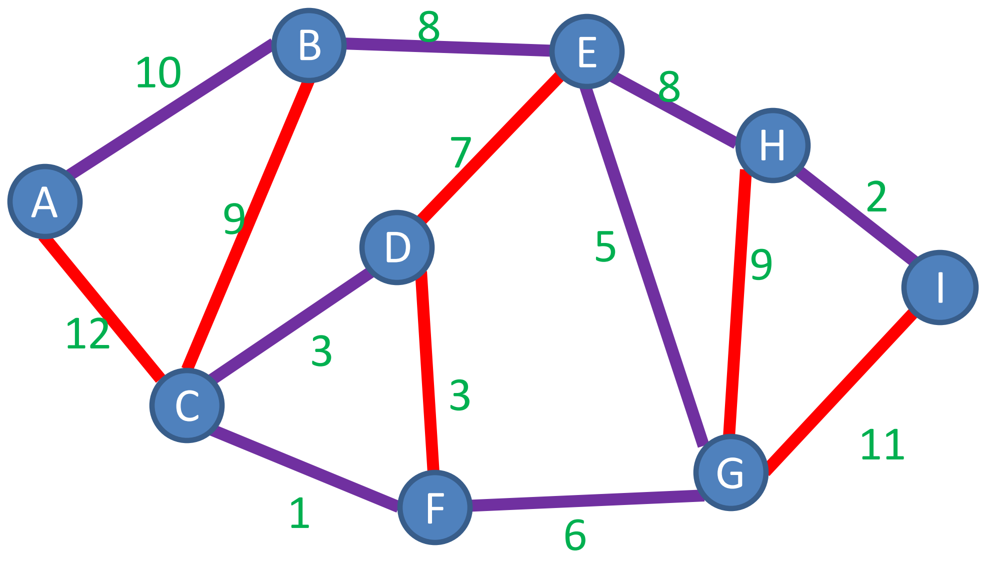
Add to A the lowest-weight edge that does not create a cycle



Kruskal's Algorithm

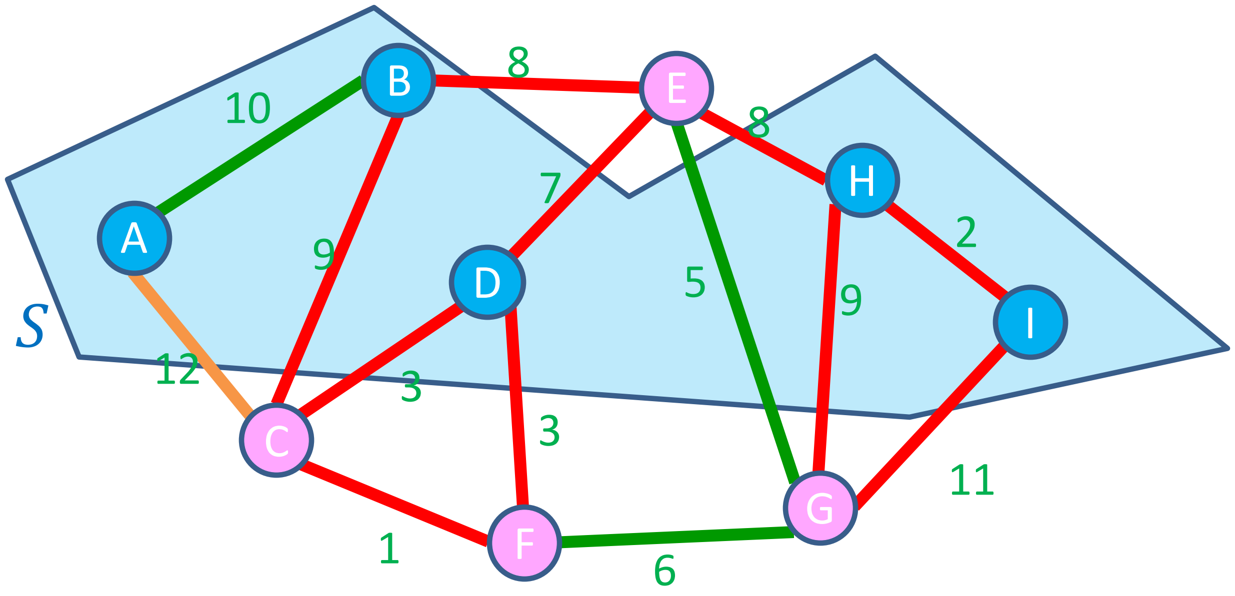
Start with an empty tree A

Add to A the lowest-weight edge that does not create a cycle



Definition: Cut

A Cut of graph $G = (V, E)$ is a partition of the nodes into two sets, S and $V - S$



Edge $(v_1, v_2) \in E$ crosses a cut if $v_1 \in S$ and $v_2 \in V - S$ (or opposite), e.g. (A, C)

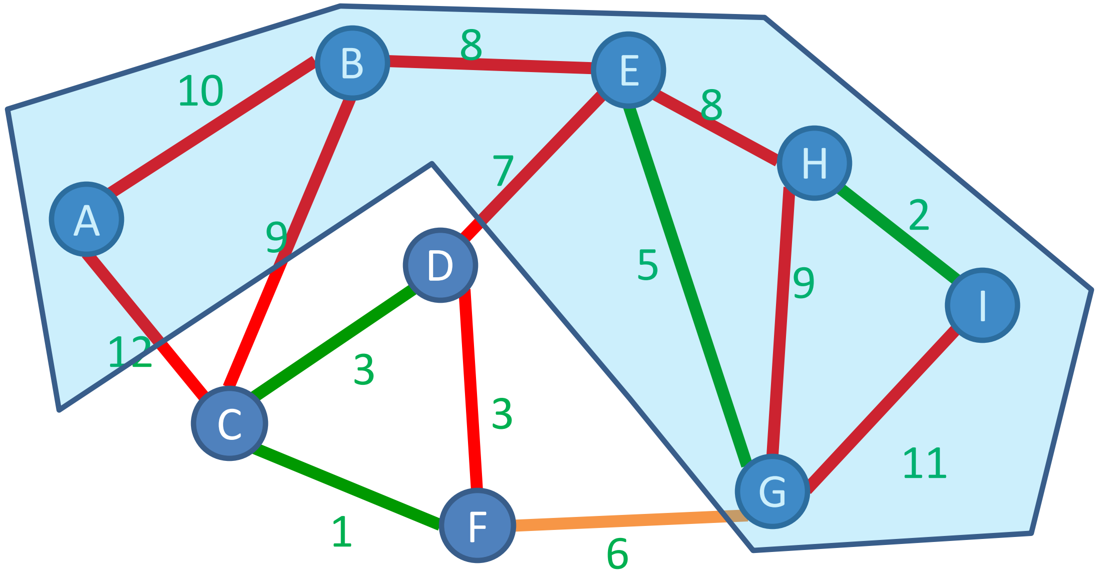
A set of edges R Respects a cut if no edges cross the cut e.g. $R = \{(A, B), (E, G), (F, G)\}$

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”

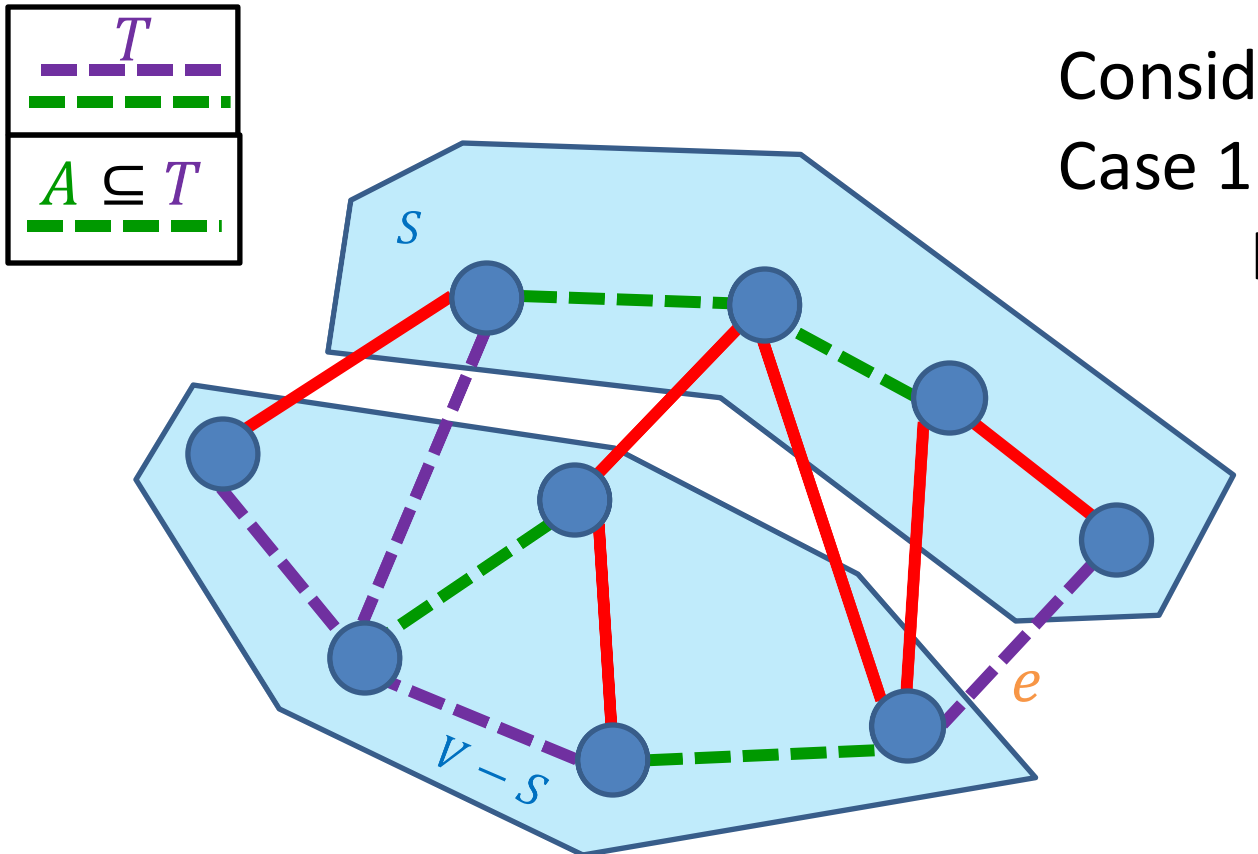
Cut Theorem

If a set of edges A is a subset of a minimum spanning tree T, let $(S, V - S)$ be any cut which A respects. Let e be the least-weight edge which crosses $(S, V - S)$. $A \cup \{e\}$ is also a subset of a minimum spanning tree.



Proof of Cut Theorem

Claim: If A is a subset of a MST T, and e is the least-weight edge which crosses cut $(S, V - S)$ (which A respects) then $A \cup \{e\}$ is also a subset of a MST.



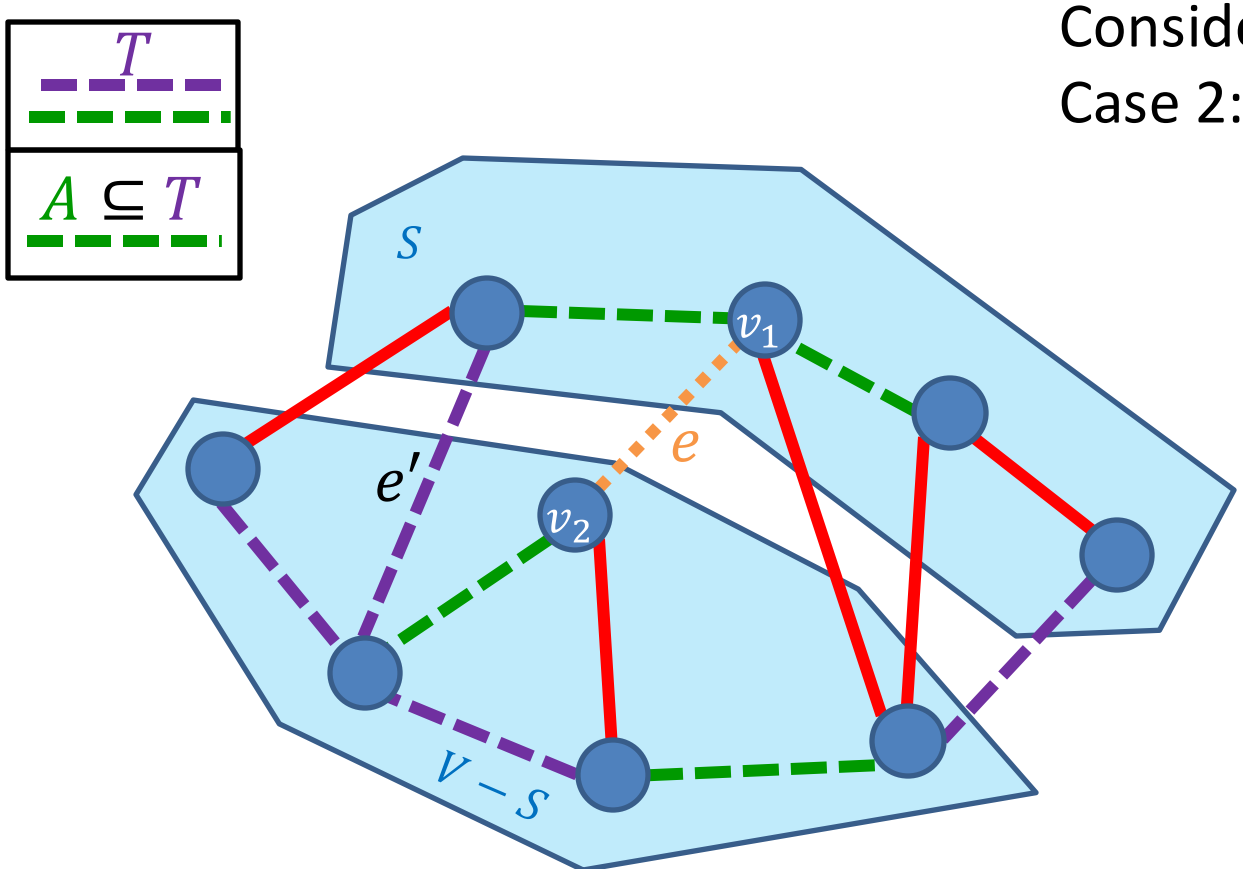
Consider some MST T,

Case 1: (the easy case)

If $e \in T$ Then claim holds

Proof of Cut Theorem

Claim: If A is a subset of a MST T, and e is the least-weight edge which crosses cut $(S, V-S)$ (which A respects) then $A \cup \{e\}$ is also a subset of a MST.



Consider some MST T,

Case 2:

Consider if $e = (v_1, v_2) \notin T$

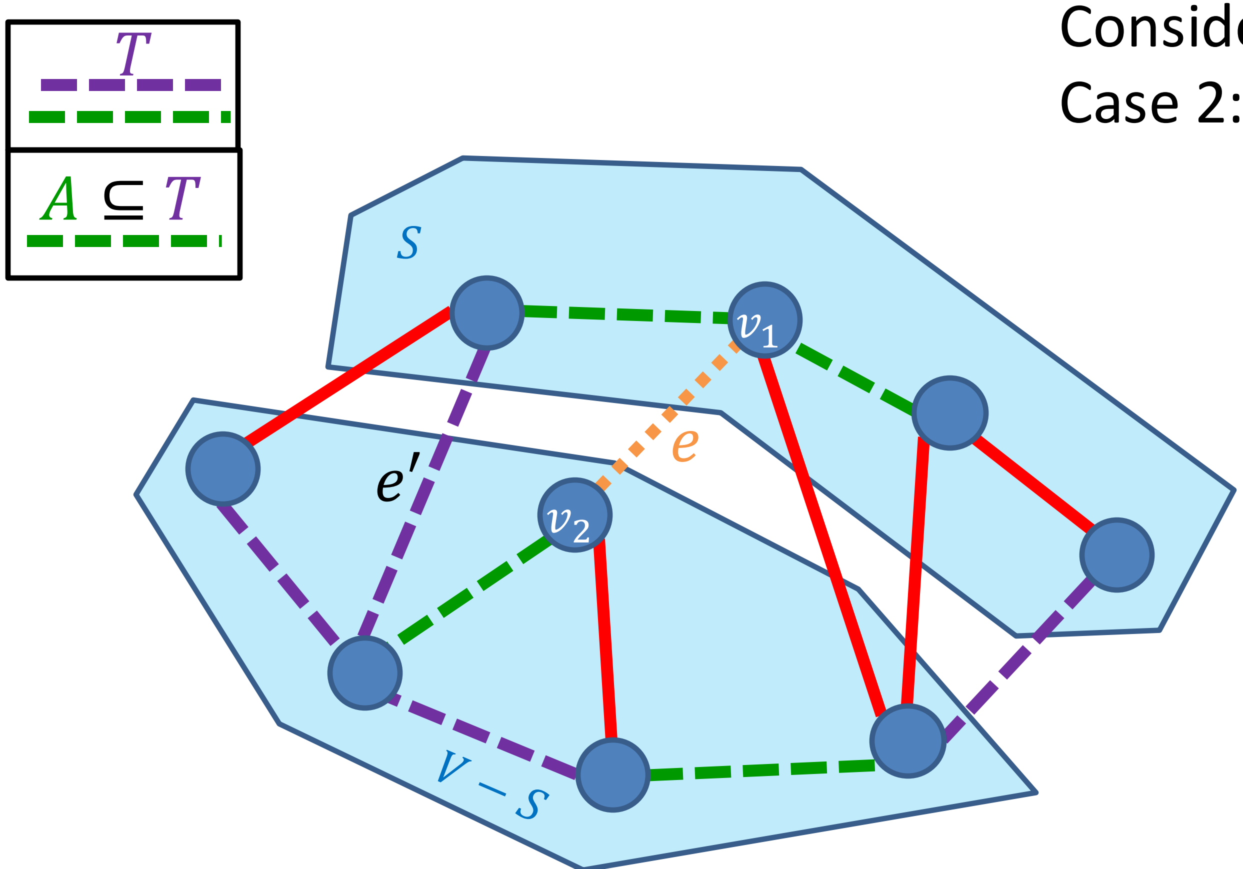
Since T is a MST, there is some path from v_1 to v_2.

Let e' be the first edge on this path which crosses the cut

Build tree T' by exchanging e' for e

Proof of Cut Theorem

Claim: If A is a subset of a MST T, and e is the least-weight edge which crosses cut $(S, V-S)$ (which A respects) then $A \cup \{e\}$ is also a subset of a MST.



Consider some MST T,

Case 2:

Consider if $e = (v_1, v_2) \notin T$

$T' = T$ with edge e instead of e'

We assumed $w(e) \leq w(e')$

$w(T') = w(T) - w(e') + w(e)$

$w(T') \leq w(T)$

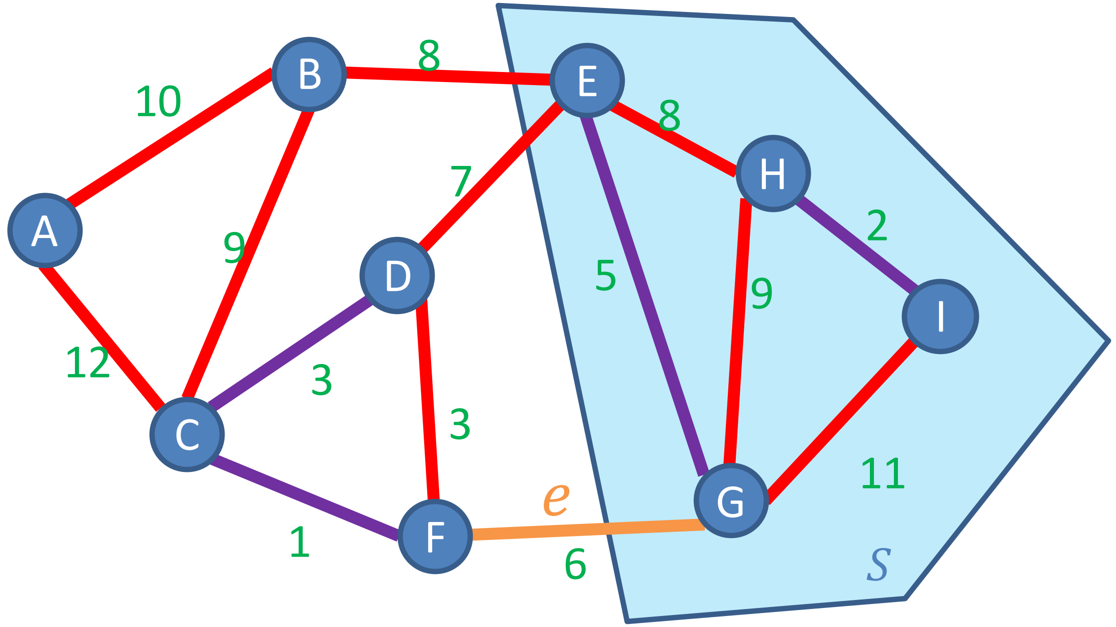
So T' is also a MST!

Thus the claim holds

Kruskal's Algorithm

Start with an empty tree A
Repeat $V-1$ times:
Add the min-weight edge that doesn't cause a cycle

Keep edges in a Disjoint-set data structure (very fancy)
$O(E \log V)$



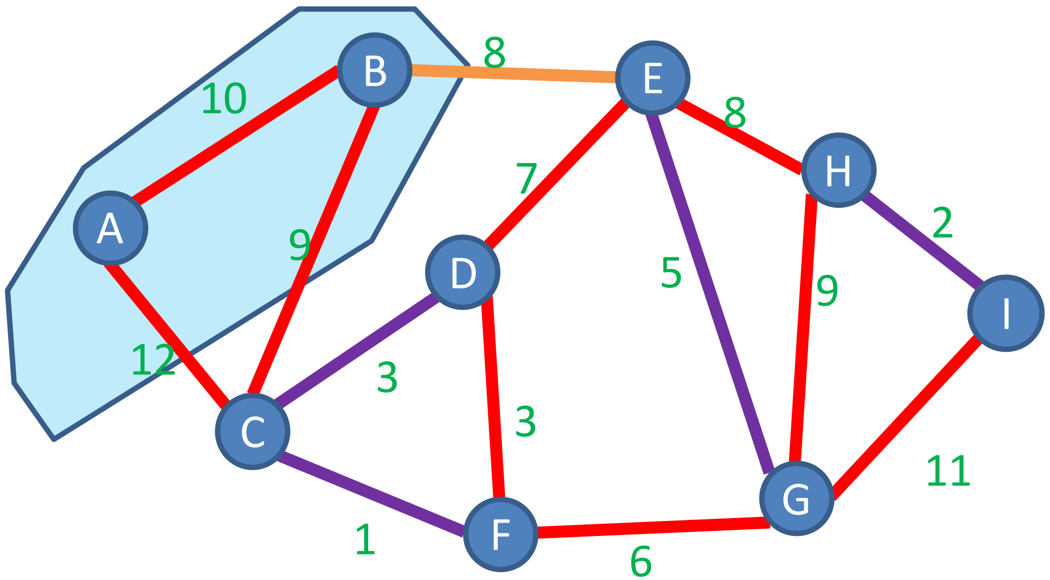
General MST Algorithm

Start with an empty tree A

Repeat $V - 1$ times:

Pick a cut $(S, V - S)$ which A respects

Add the min-weight edge which crosses $(S, V - S)$



Prim's Algorithm

Start with an empty tree A

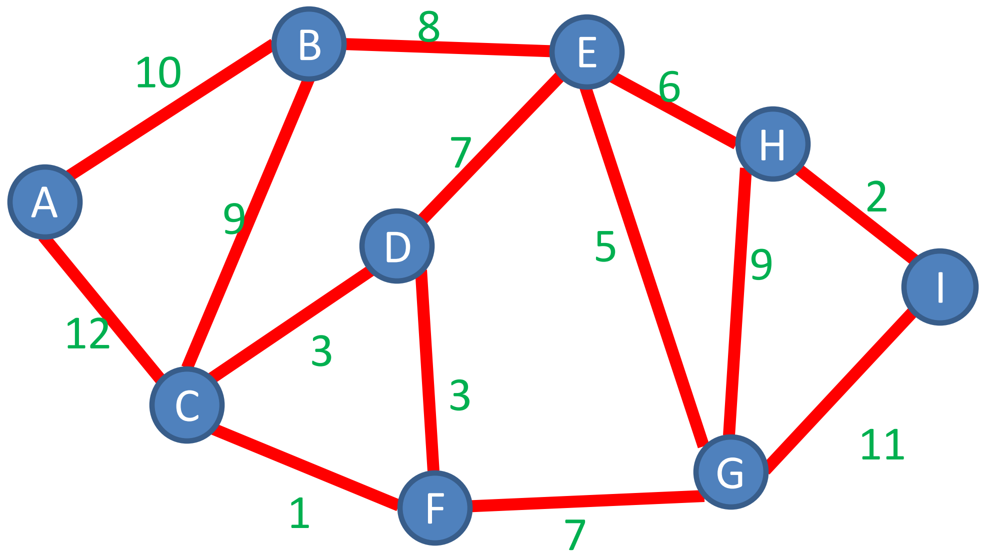
Repeat $V - 1$ times:

Pick a cut $(S, V - S)$ which A respects

Add the min-weight edge which crosses $(S, V - S)$

S is all endpoint of edges in A

e is the min-weight edge that grows the tree



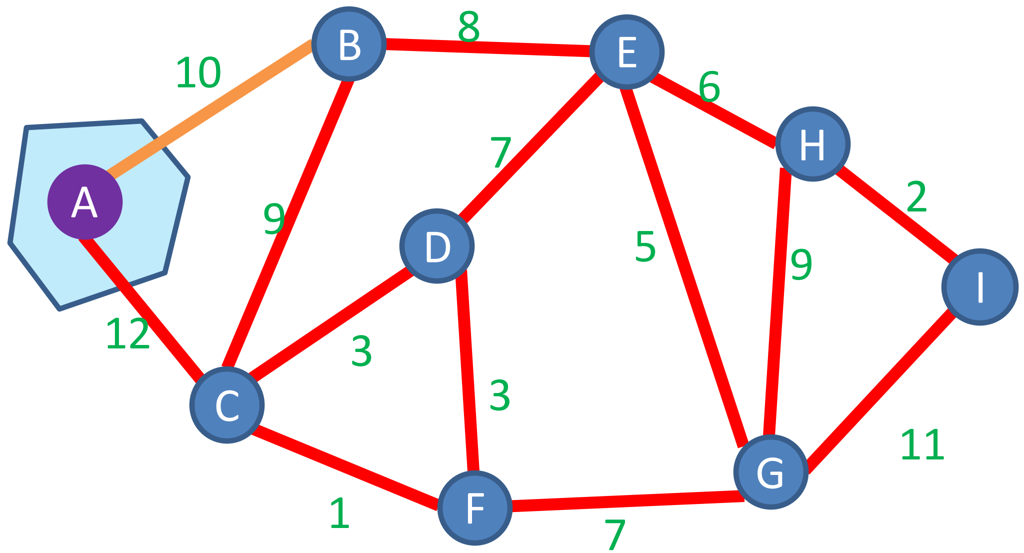
Prim’s Algorithm

Start with an empty tree A

Pick a start node

Repeat $V - 1$ times:

Add the min-weight edge which connects to node in A with a node not in A



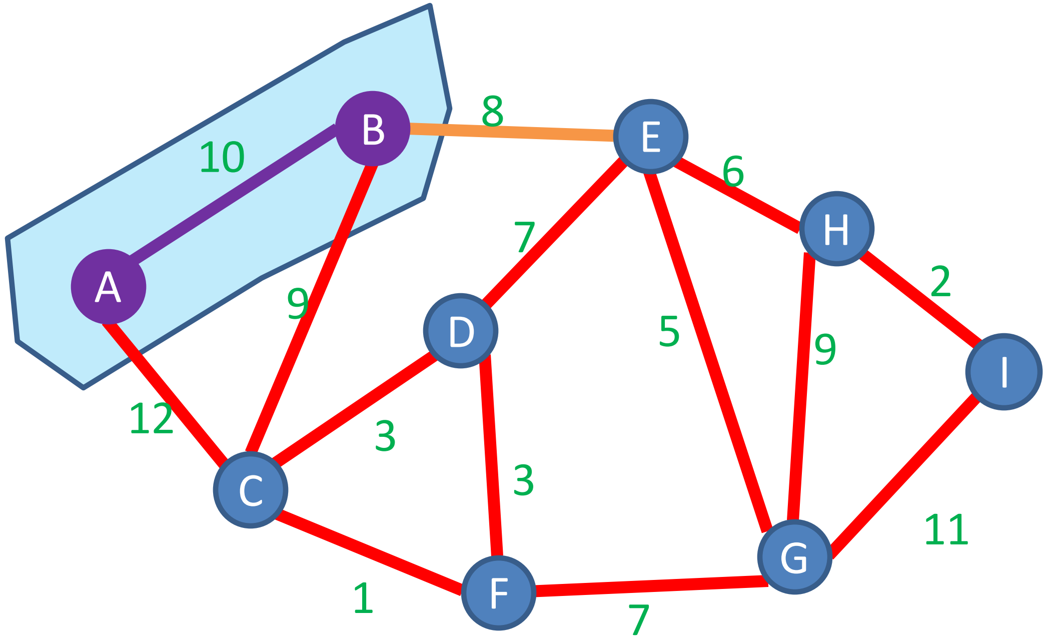
Prim's Algorithm

Start with an empty tree A

Pick a start node

Repeat $V - 1$ times:

Add the min-weight edge which connects to node in A with a node not in A



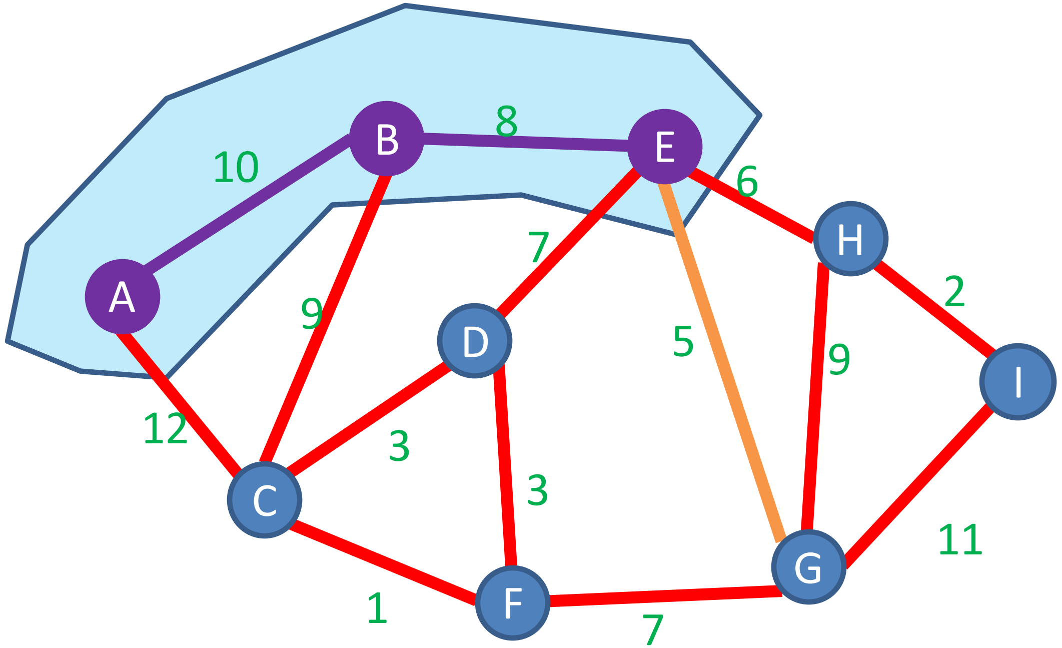
Prim's Algorithm

Start with an empty tree A

Pick a start node

Repeat $V - 1$ times:

Add the min-weight edge which connects to node in A with a node not in A



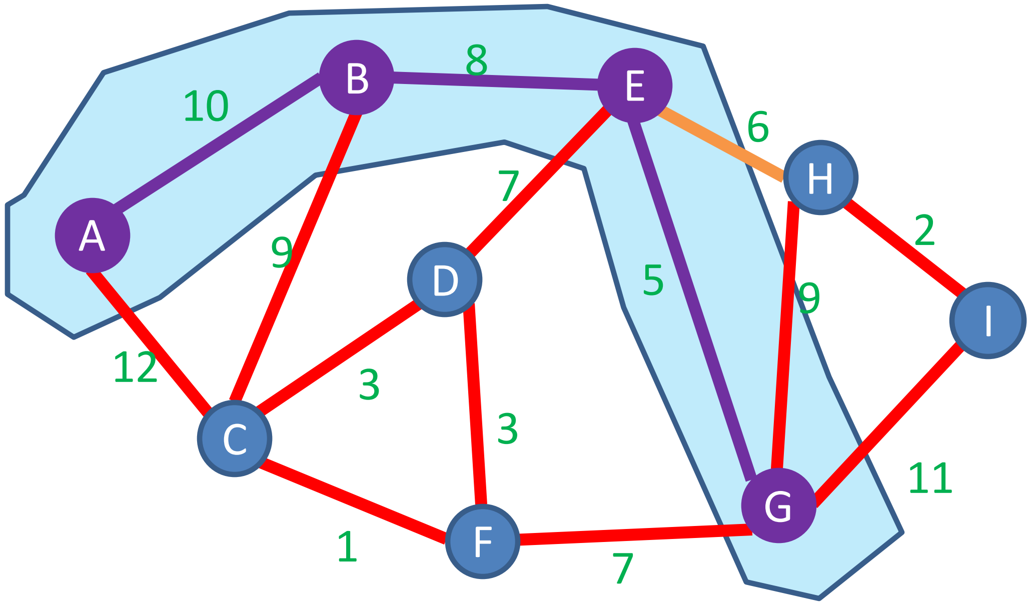
Prim's Algorithm

Start with an empty tree A

Pick a start node

Repeat $V - 1$ times:

Add the min-weight edge which connects to node in A with a node not in A



Prim's Algorithm

Start with an empty tree A
Pick a start node
Repeat $V - 1$ times:
Add the min-weight edge which connects to node in A with a node not in A

Keep edges in a Heap
$O(E \log V)$