Exam 2 Review

CSE 101

Winter 2023

Exam Details

- In class
- Randomized assigned seats
- You may use 6 one-sided pages of notes
- No textbook or electronic aids
- No need to provide proofs unless asked for
- 3 Questions in 45 minutes
  - 1st straightforward implementation of algorithm
  - 2nd requires some thought
  - 3rd can be quite tricky

This Review

- Brief outline of topics that might show up on the exam
- To see anything in more depth use other review options.

Other Review Options

- Lecture podcasts / slides
- Textbook
- OH questions
- Old exams from problem archive

Topics

- Divide and Conquer
  - Basic paradigm
  - Master Theorem
  - Karatsuba Multiplication
  - MergeSort
  - Order statistics
  - Binary Search
  - Closest Pair of Points
- Greedy algorithms
  - Basic paradigm
  - Exchange arguments
  - Interval packing
  - Optimal Caching

Divide & Conquer (Ch 2)

- General Technique
- Master Theorem
- Karatsuba Multiplication
- Strassen’s Algorithm
- Merge Sort
- Order Statistics
- Binary Search
- Closest Pair of Points

Divide and Conquer

This is the first of our three major algorithmic techniques.

1. Break problem into pieces
2. Solve pieces recursively
3. Recombine pieces to get answer

Example: Integer Multiplication

Problem: Given two n-bit numbers find their product.

Naïve Algorithm: Schoolboy multiplication. The binary version of the technique that you probably learned in elementary school.

Runtime: $O(n^2)$

Schoolboy Multiplication

$$
\begin{array}{cccccccc}
 & & & a_1 & a_2 & \ldots & a_{n-1} & a_n \\
 & & \mathrm{x} & b_1 & b_2 & \ldots & b_{n-1} & b_n \\
\cline{4-8}
 & & a_1 b_n & a_2 b_n & a_3 b_n & \ldots & a_{n-1} b_n & a_n b_n \\
 & a_1 b_{n-1} & a_2 b_{n-1} & a_3 b_{n-1} & a_4 b_{n-1} & \ldots & a_n b_{n-1} & 0 \\
 & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\
+a_1 b_n & a_2 b_n & a_3 b_n & \ldots & a_n b_n & 0 & 0 & 0 \\
\hline
\end{array}
$$

ANSWER

Formally

Want to multiply N and M:

1. Let $X \approx \sqrt{(N+M)}$ be a power of 2.
2. Write $N = AX+B$, $M = CX+D$
   – This can be done by just taking the high and low bits.
3. $N \cdot M = AC \cdot X^2+(AD+BC)X+BD$

   $= AC \cdot X^2+[(A+B)(C+D)-AC-BD]X + BD$

   – The multiplications by X are just bit shifts.

Improved Multiplication

```
ImprovedMult(N,M)
  Let X be a power of 2⌊log(N+M)/2⌋
  Write N = AX + B, M = CX + D
  P1 ← Product(A,C)
  P2 ← Product(B,D)
  P3 ← Product(A+B,C+D)
  Return P1X² + [P3-P1-P2]X + P2
```

Karatsuba

KaratsubaMult(N,M)

If N+M<99, Return Product(N,M)

Let X be a power of $2^{\lfloor \log(N+M)/2 \rfloor}$

Write N = AX + B, M = CX + D

$P_1 \leftarrow$ Product(A,C)

$P_1 \leftarrow$ KaratsubaMult(A,C)

$P_2 \leftarrow$ Product(B,D)

$P_2 \leftarrow$ KaratsubaMult(B,D)

$P_3 \leftarrow$ Product(A+B,C+D)

$P_3 \leftarrow$ KaratsubaMult(A+B,C+D)

Return $P_1X^2 + [P_3-P_1-P_2]X + P_2$

Runtime Recurrence

Karatsuba multiplication on inputs of size n spends O(n) time, and then makes three recursive calls to problems of (approximately) half the size.

If T(n) is the runtime for n-bit inputs, we have the recursion:

$$T(n) = \begin{cases} O(1) & \text{if } n = O(1) \\ 3T(n/2 + O(1)) + O(n) & \text{otherwise} \end{cases}$$

How do we solve this recursion?

Generalization

We will often get runtime recurrences with D&C looking something like this:

$T(n) = O(1)$ for $n = O(1)$

$T(n) = a\,T(n/b + O(1)) + O(n^d)$ otherwise.

Tracking Recursive Calls

We have:

- 1 recursive call of size n
- a recursive calls of size $n/b+O(1)$
- a^2 recursive calls of size $n/b^2+O(1)$
- …
- a^k recursive calls of size $n/b^k+O(1)$

Bottoms out when $k = \log_b(n)$.

Runtime

Combining the runtimes from each level of the recursion we get:

$$\text{Total Runtime} = \sum_{k=0}^{\log_b(n)} a^k O((n/b^k)^d)$$

$$= O(n^d) \sum_{k=0}^{\log_b(n)} (a/b^d)^k.$$

The asymptotics will depend on whether a/b^d is bigger than 1.

Master Theorem

<u>Theorem:</u> Let T(n) be given by the recurrence:

$$T(n) = \begin{cases} O(1) & \text{if } n = O(1) \\ aT(n/b + O(1)) + O(n^d) & \text{otherwise} \end{cases}$$

Then we have that

$$T(n) = \begin{cases} O(n^{\log_b(a)}) & \text{if } a > b^d \\ O(n^d \log(n)) & \text{if } a = b^d \\ O(n^d) & \text{if } a < b^d \end{cases}$$

Note

In divide and conquer, it is important that the recursive subcalls are a constant *fraction* of the size of the original.

Note on Proving Correctness

There's a general procedure for proving correctness of a D&C algorithm:

<u>Use Induction:</u> Prove correctness by induction on problem size.

<u>Base Case:</u> Your base case will be the non-recursive case of your algorithm (which your algorithm does need to have).

<u>Inductive Step:</u> Assuming that the (smaller) recursive calls are correct, show that algorithm works.

Sorting

<u>Problem:</u> Given a list of n numbers, return those numbers in ascending order.

<u>Example:</u>

Input: {0, 5, 2, 7, 4, 6, 3, 1}

Output: {0, 1, 2, 3, 4, 5, 6, 7}

Merge

Problem: Given two sorted lists, combine them into a single sorted list.

We want something that takes advantage of the individual lists being sorted.

Merge

```
Merge(A,B)
  C ← List of length Len(A)+Len(B)
  a ← 1, b ← 1
  For c = 1 to Len(C)
    If (b > Len(B))
      C[c] ← A[a], a++
    Else if (a > Len(A))
      C[c] ← B[b], b++
    Else if A[a] < B[b]
      C[c] ← A[a], a++
    Else
      C[c] ← B[b], b++
  Return C
```

Runtime: $O(|A|+|B|)$

MergeSort

```
MergeSort(L)
  If Len(L) = 1          \\ Base Case
    Return L
  Split L into equal L1 and L2      O(n)
  A ← MergeSort(L1)
  B ← MergeSort(L2)                 2T(n/2)
  Return Merge(A,B)                 O(n)
```

Runtime

<u>Master Theorem:</u>

$a = 2, b = 2, d = 1$

$a = b^d$

$O(n^d \log(n)) = O(n \log(n))$

Order Statistics

Problem: Given a list L of numbers and an integer k, find the kth largest element of L.

Naïve Algorithm: Sort L and return kth largest. O(n log(n)) time.

Divide Step

Select a *pivot* x ∈ L. Compare it to the other elements of L.

< x = x > x

Only recurse on one list

Which list is our answer in?

- Answer is > x if there are ≥ k elements bigger than x.
- Answer is x if there are < k elements bigger and ≥ k elements bigger than or equal to x.
- Otherwise answer is less than x.

Order Statistics

```
Select(L,k)
  Pick x ∈ L
  Sort L into L_{>x}, L_{<x}, L_{=x}
  If Len(L_{>x}) ≥ k
    Return Select(L_{>x},k)
  Else if Len(L_{>x})+Len(L_{=x}) ≥ k
    Return x
  Return
   Select(L_{<x},k-Len(L_{>x})-Len(L_{=x}))
```

Runtime

Runtime recurrence

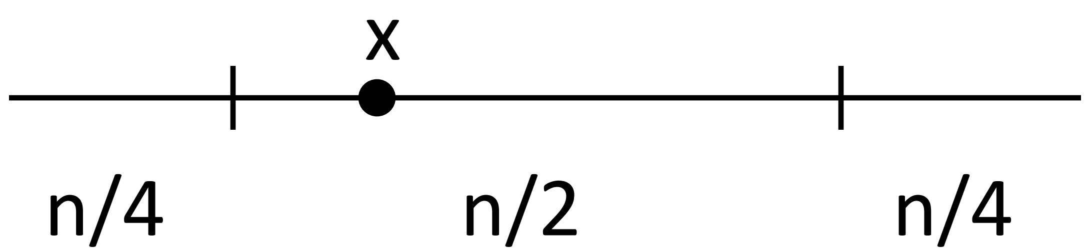
$$T(n) = O(n) + T(\text{sublist size})$$

<u>Problem:</u> The sublist we recurse on could have size as big as n-1. If so, runtime is $O(n^2)$.

Need to ensure this doesn't happen.

Randomization

__Fix:__ Pick a *random* pivot.



- There's a 50% chance that x is selected in the middle half.
- If so, no matter where the answer is, recursive call of size at most 3n/4.
- On average need two tries to reduce call.

Runtime

Master Theorem:

$a = 1$, $b = 4/3$, $d = 1$

$a < b^d$

$O(n^d) = O(n)$

Search

<u>Problem:</u> Given a sorted list L and a number x, find the location of x in L.

Divide

Split L into two lists.

- Could search for x in each
  $T(n) = 2T(n/2)+O(1)$ → Too slow
- Use sorting to figure out which list to check.

If L[i] > x, location must be before i.

If L[i] < x, location must be after i.

If L[i] = x, we found it.

Binary Search

```
BinarySearch(L,i,j,x)
\\Search between L[i] and L[j]
  If j < i, Return 'error'
  k ← [(i+j)/2]
  If L[k] = x, Return k
  If L[k] > x
    Return BinarySearch(L,i,k-1,x)
  If L[k] < x
    Return BinarySearch(L,k+1,j,x)
```

Runtime

<u>Master Theorem:</u>

$a = 1, b = 2, d = 0$

$a = b^d$

$O(n^d \log(n)) = O(\log(n))$

Binary Search Puzzles

You have 27 coins one of which is heavier than the others, and a balance. Determine the heavy coin in 3 weightings.

Lots of puzzles have binary search-like answers. As long as you can spend constant time to divide your search space in half (or thirds). You can use binary search in O(log(n)) time.

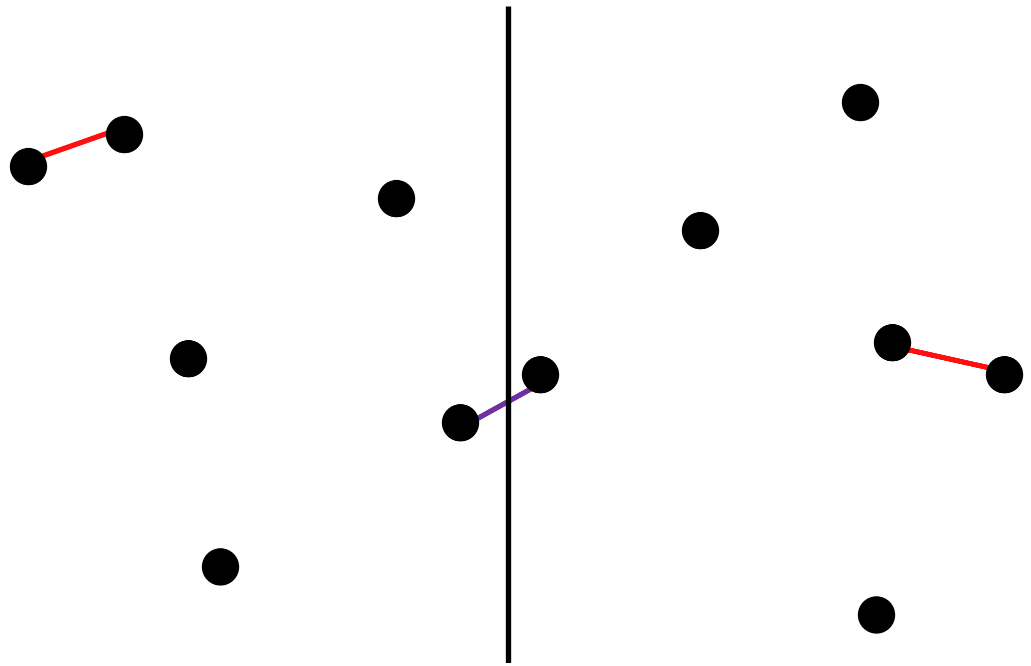
Closest Pair of Points (Ex 2.32)

Problem: Given n points in the plane $(x_1,y_1)\ldots(x_n,y_n)$ find the pair (x_i,y_i) and (x_j,y_j) whose Euclidean distance is as small as possible.

Naïve Algorithm: Try every pair of points. $O(n^2)$ time.

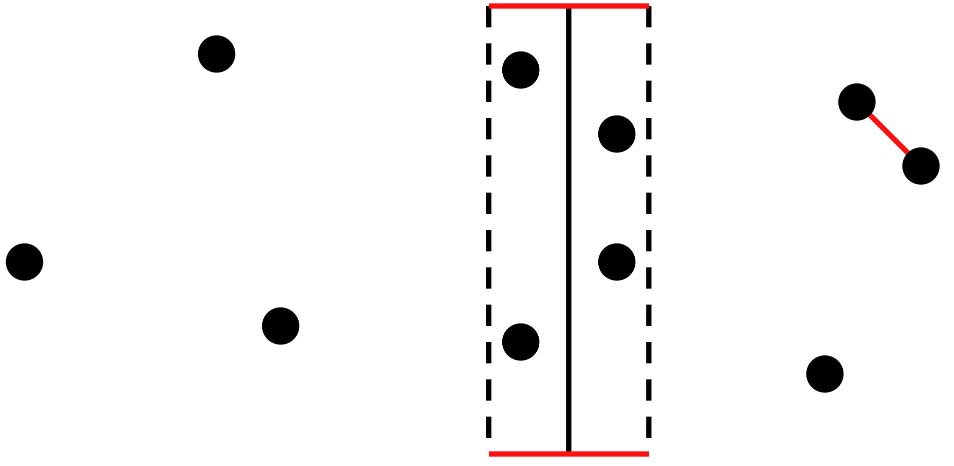
Divide and Conquer Outline

- Divide points into two sets by drawing a line.
- Compute closest pair on each side.
- What about pairs that cross the divide?



Observation

- Suppose closest pair on either side at distance δ.
- Only need to care about points within δ of dividing line.
- Need to know if some pair closer than δ.

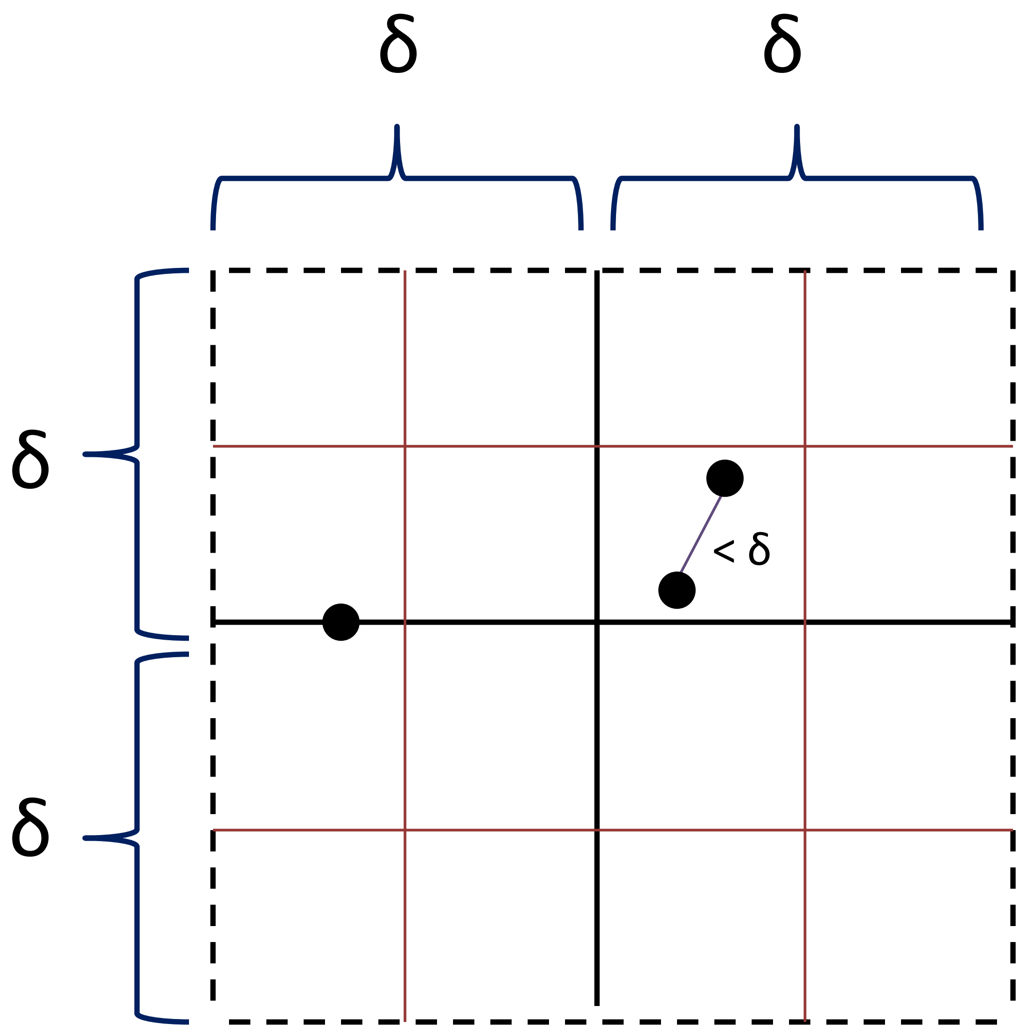


Main Idea

Proposition: Take the points within δ of the dividing line and sort them by y-coordinate. Any one of these points can only be within δ of the 8 closest points on either side of it.

Proof

- Nearby points must have y-coordinate within δ.
- Subdivide region into δ/2-sided squares.
- At most one point in each square.



Algorithm

```
CPP(S)
  If |S| ≤ 3
    Return closest distance
  Find line L evenly dividing points
  Sort S into S_left, S_right
  δ ← min(CPP(S_left),CPP(S_right))
  Let T be points within δ of L
  Sort T by y-coordinate
  Compare each element of T to 8 closest
    on either side. Let min dist be δ'.
  Return min(δ,δ')
```

Runtime

With a more careful analysis you can obtain a runtime of $O(n \log(n))$.

Greedy Algorithms (Ch 5)

- Basics
- Change making
- Interval scheduling
- Exchange arguments
- Optimal caching

Greedy Algorithms

General Algorithmic Technique:

1. Find decision criterion
2. Make best choice according to criterion
3. Repeat until done

Things to Keep in Mind about Greedy Algorithms

- Algorithms are very natural and easy to write down.
- However, not all greedy algorithms work.
- Proving correctness is important.

Interval Scheduling

Problem: Given a collection C of intervals, find a subset $S \subseteq C$ so that:

1. No two intervals in S overlap.
2. Subject to (1), $|S|$ is as large as possible.

Algorithm

```
IntervalScheduling(C)
  S ← {}
  While(some interval in C
       doesn’t overlap any in S)
    Let J be the non-overlapping
      interval with smallest max
    Add J to S
  Return S
```

Proof of Correctness

- Algorithm produces J_1, J_2,…,J_s with $J_i = [x_i,y_i]$.
- Consider some other solution K_1, K_2,…,K_t with $K_i = [w_i,z_i]$.

Claim: For each $m \le t$, $y_m \le z_m$.

In particular, $s \ge t$.

Exchange Argument

- Greedy algorithm makes a sequence of decisions D_1, D_2, D_3,…,D_n eventually reaching solution G.
- Need to show that for arbitrary solutions A that G ≥ A.
- Find sequence of solutions
  $A=A_0, A_1, A_2,\ldots,A_n = G$
  so that:
  - $A_i \leq A_{i+1}$
  - A_i agrees with $D_1,D_2,\ldots,D_i$

Exchange Argument

In particular, we need to show that given any A_i consistent with $D_1,\ldots,D_i$ we can find an A_{i+1} so that:

- A_{i+1} is consistent with $D_1,\ldots,D_{i+1}$
- $A_{i+1} \geq A_i$

Then we inductively construct sequence

$A=A_0 \leq A_1 \leq A_2 \leq \ldots \leq A_n = G$

Thus, $G \geq A$ for any A. So G is optimal.

Optimal Caching Model

- k words in cache at a time.
- CPU asks for memory access.
- If in cache already, easy.
- Otherwise, need to load into cache replacing something else, slow.

Optimal Caching

<u>Problem:</u> Given sequence of memory accesses and cache size, find a cache schedule that involves fewest possible number of swaps with disk.

Observation

- No need to get new entries in cache ahead of time.
- Only make replacements when new value is called for.
- Only question algorithm needs to answer is which memory cells to overwrite.

Furthest In The Future (FITF)

- For each cell consider the next time that memory location will be called for.
- Replace cell whose next call is the furthest in the future.

Proof of Optimality

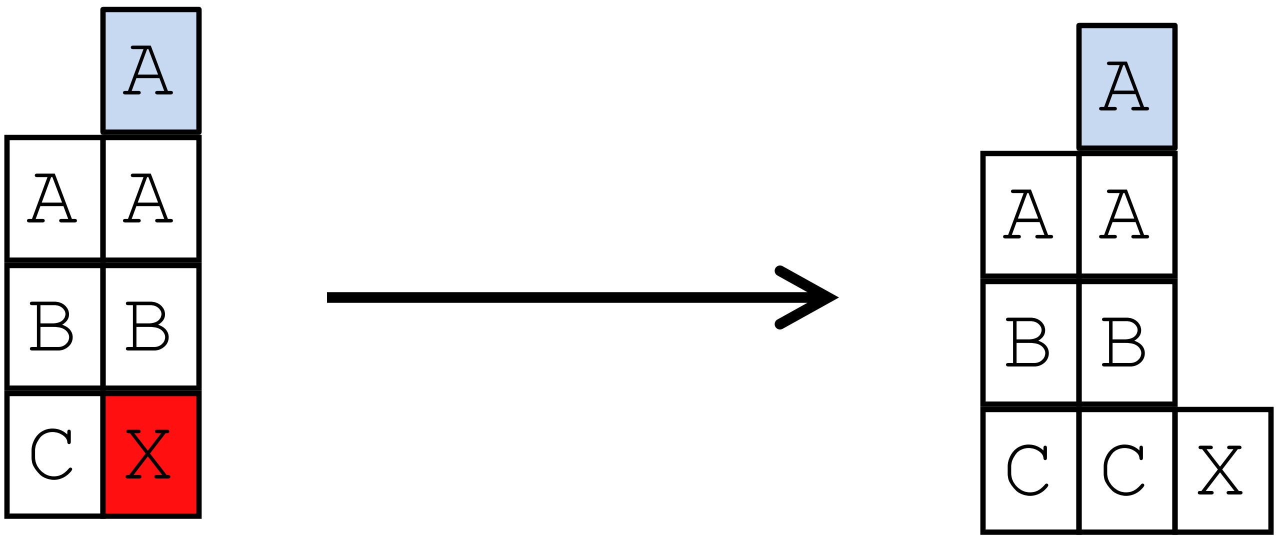
- Exchange argument
- n^{th} decision: What to do at n^{th} time step.
- Given schedule S that agrees with FITF for first n time steps, create schedule S' that agrees for n+1 and has no more cache misses.

Case 1: S agrees with FITF on step n+1

Nothing to do. S' = S.

Case 2: S Makes Unnecessary Replacement

If S replaces some element of memory that was not immediately called for, put it off.

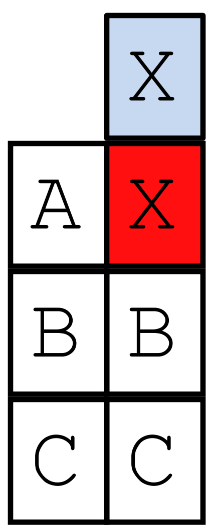


Can assume that S only replaces elements if there's a cache miss.

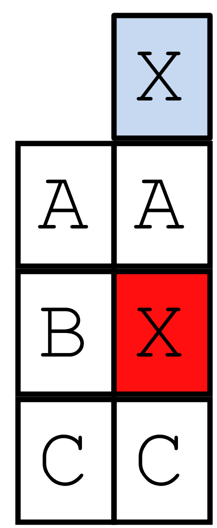
Case 3

The remaining case is that there is a cache miss at step n+1 and that S replaces the *wrong* thing.

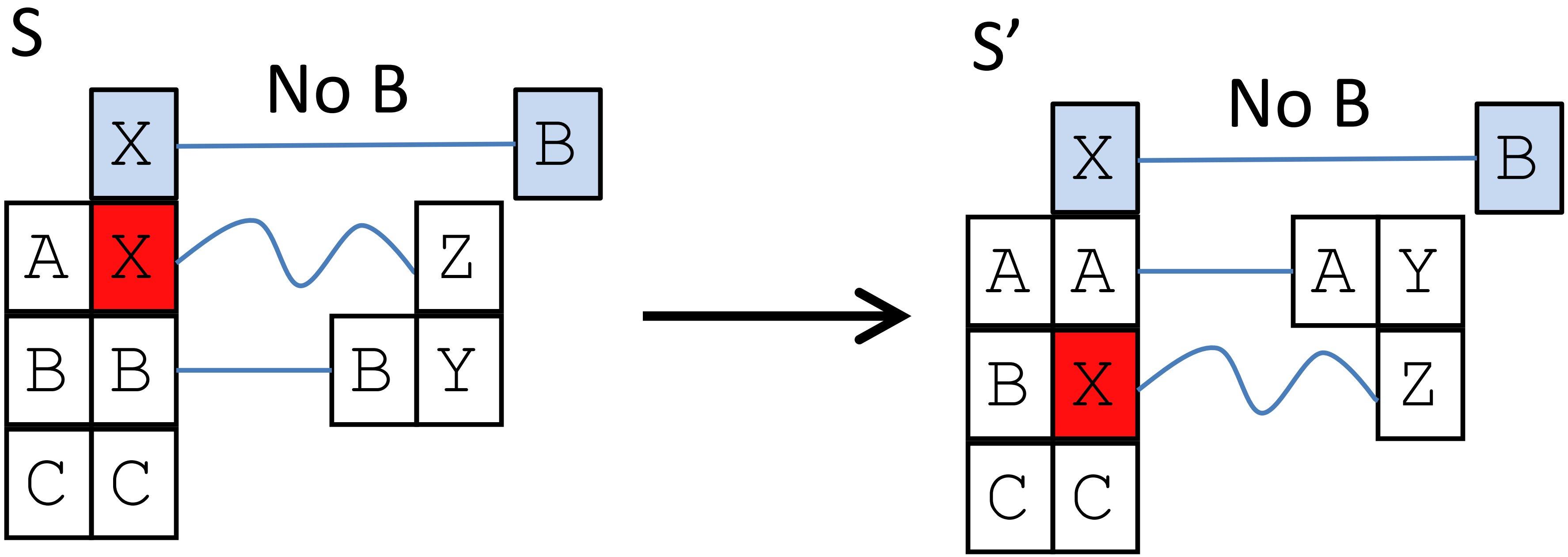
S



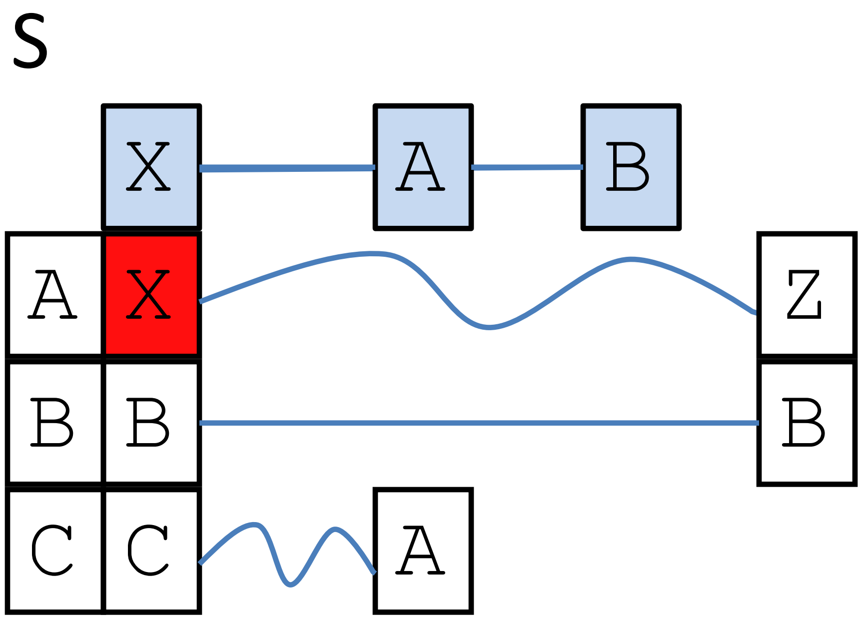
FITF



Case 3a: S throws out B before using it

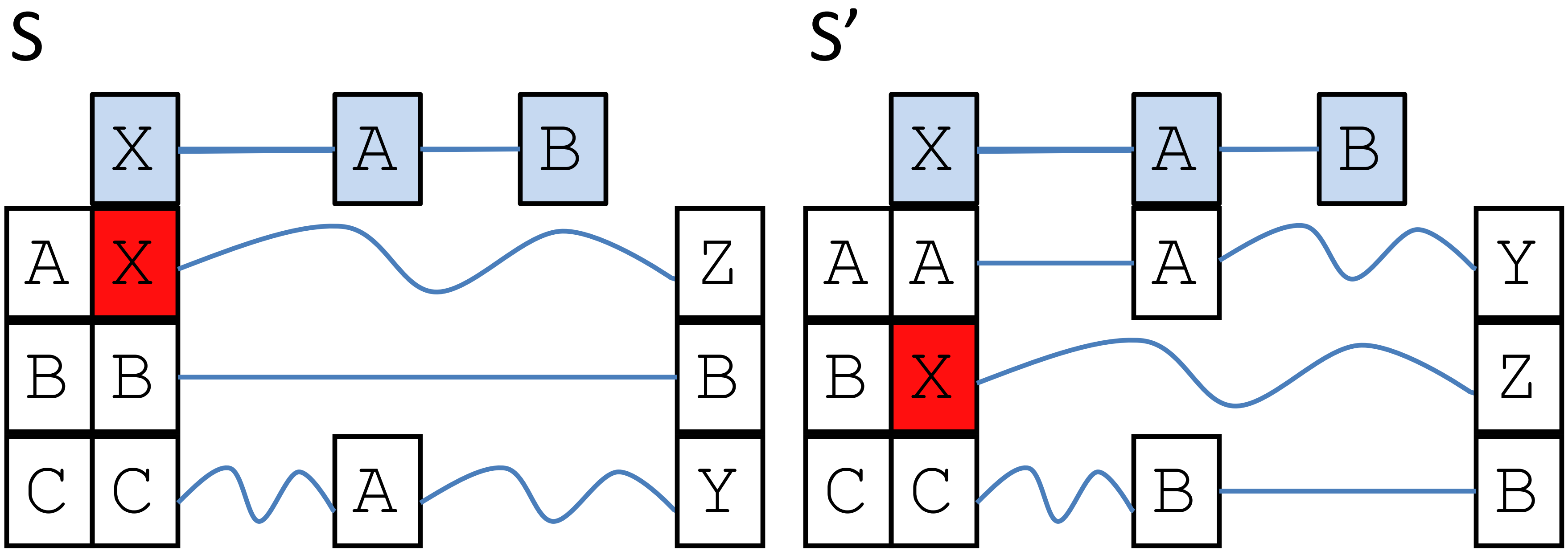


Case 3b: S keeps B until it is used



- B is FITF
- A is used sometime before B.
- A must be loaded into memory somewhere else.

Case 3b: S keeps B until it is used



Instead of replacing A and then bringing it back, we can replace B and then bring it back.