APPLIED ALGORITHMS

Lecture 8: Some catchup, then starting Section 2

ADMIN

- Assignment 2 is over!
  - It was a hard one
  - Mini-midterm will be much much easier I think
- Mini-Midterm, code pushed this afternoon

TODAY

- Finish Assignment 1 discussion
- Loop unrolling
- Midterm
- Intro to probability, hashing

NEXT WEEK

- Assignment 2 solutions/discussion
- More probability!

- I need to leave early next Tuesday (may end class ~20 minutes early; or may have guest mini-lecture)

PROBLEM 3: BALANCED TOWERS

The bottom two blocks must be at most one apart.

Again, break it down:

- The best (smaller) balanced tower must consist of some blocks from the first half, and some blocks from the second half. AND the largest block not included in this tower must be within 1 of the largest included.

PROBLEM 3: BALANCED TOWERS

Why won't the following work?

- Iterate through each first half and find the second half closest to the target. If the second half is balanced with the first, check if it is a better solution. Otherwise, throw it out.

- Underspecified. (What does "throw it out" mean? Do we keep the same first half, or move on to the next one?)
- Might not find the best/might take a long time to find the best.

PROBLEM 3: BALANCED TOWERS

"Algorithmic" strategy:

- The smaller tower either has the largest block in it; or it doesn't have the largest block in it.
- Can we solve these two cases?
- Assume without loss of generality that the largest block is in the first half of the input (can just reverse the input if not true)

PROBLEM 3: BALANCED TOWERS

Case 1 (smaller tower does not have largest block):

- There needs to be at least one block in the smaller tower within 1 of largest
- Build two tables on the second half of input: Table1 only contains subsets which have at least one block within 1 of largest. Table 2 contains all subsets.
- For each first-half subset:
  - If the subset contains no block within 1 of largest, search in Table 1
  - If it does have a block within 1 of largest, search in Table 2

PROBLEM 3: BALANCED TOWERS

Case 2 (smaller tower does have largest block):

- There needs to be at least one block in the larger tower within 1 of largest (must be *excluded* from our small-tower solution)
- Build two tables: Table1 only contains subsets from 2nd half which are *missing* at least one block within 1 of largest. Table2 contains all subsets.
- For each first-half subset:
  - If the subset contains all possible blocks within 1 of largest, search in Table 1
  - If it is missing a block within 1 of largest, search in Table 2

PROBLEM 3: BALANCED TOWERS (RUNNING TIME)

Case 1 (smaller tower does not have largest block):

- There needs to be at least one block in the smaller tower within 1 of largest
- Build two tables: Table1 only contains subsets which have at least one block within 1 of largest. Table2 contains all subsets.
- For each first-half subset:
  - If the subset contains no block within 1 of largest, search in Table 1
  - If it does have a block within 1 of largest, search in Table 2

$O(n\ 2^{n/2})$ (just like before!)

PROBLEM 3: BALANCED TOWERS (RUNNING TIME)

Case 1 (smaller tower does not have largest block):

- There needs to be at least one block in the smaller tower within 1 of largest
- Build two tables: Table1 only contains subsets which have at least one block within 1 of largest. Table2 contains all subsets.
- For each first-half subset:
  - If the subset contains no block within 1 of largest, search in Table 1
  - If it does have a block within 1 of largest, search in Table 2

Can check in $O(n)$ time per subset

PROBLEM 3: BALANCED TOWERS (RUNNING TIME)

Case 2 (smaller tower does have largest block):

- There needs to be at least one block in the larger tower within 1 of largest (must be *excluded* from our small-tower solution)
- Build two tables: Table1 only contains subsets which are *missing* at least one block within 1 of largest. Table2 contains all subsets.
- For each first-half subset:
  - If the subset contains all possible blocks within 1 of largest, search in Table 1
  - If it is missing a block within 1 of largest, search in Table 2

$O(n\ 2^{n/2})$ (just like before!)

PROBLEM 3: BALANCED TOWERS (RUNNING TIME)

Case 2 (smaller tower does have largest block):

- There needs to be at least one block in the larger tower within 1 of largest (must be *excluded* from our small-tower solution)
- Build two tables: Table1 only contains subsets which are *missing* at least one block within 1 of largest. Table2 contains all subsets.
- For each first-half subset:
  - If the subset contains all possible blocks within 1 of largest, search in Table 1
  - If it is missing a block within 1 of largest, search in Table 2

Can check in $O(n)$ time per subset

PROBLEM 3: BALANCED TOWERS

Final algorithm:

- Split in two cases: either the smallest tower has the largest block, or it does not. We will run the algorithm for both cases and take the better solution
- (Case 1 is as described above; Case 2 is as described above)

- Running time: $O(n\ 2^{n/2})$

PROBLEM 3: BALANCED TOWERS

Whoa…do I really need all that?

PROBLEM ANSWER EXPECTATIONS

- Make sure your algorithm is clear (I should be able to implement it with your description)
  - "Store the best k elements in a sorted array. When considering a new element, insert it into the sorted array, shifting remaining elements down" – OK
  - "Store the best k elements in a (k+1)-sized max heap. Remove the smallest element before inserting a new element" – also OK (I know what a heap is)
  - "Keep track of the best when going through each subset. If it's not balanced, throw it out" – not enough detail!
- I went very slowly through the problem
  - Want to give an example of algorithmic thinking
  - Want to give an example of a VERY clear description
- There existed correct 3-4 line solutions to both of these problems

3-SUM

The problem on the midterm

THE PROBLEM

- Given 3 arrays A, B, and C
- Each consists of n integers
- Problem: give i, j, k such that A[i] + B[j] = C[k]

Can someone give me a simple algorithm to solve this problem?

IS THIS ACTUALLY WORTH SOLVING?

- Yes, surprisingly!

- Important subroutine for:
  - Finding 3 collinear points (important for ruling out corner cases in computational geometry)
  - Problems in graphs (finding 0-sum triangles)
  - Pattern matching (problems involving dictionaries of large strings

BETTER ALGORITHM

- Can solve it in $O(n^2)$
- Another "walk from both sides" algorithm
- Idea: sort A and B. (can also sort C if you want)
- Fix a k
- Can find in $O(n)$ time if there is an i, j such that A[i] + B[j] = C[k]

- Invariant: if pointing at i' and j', then the correct i and j satisfy i >= i', and j <= j'

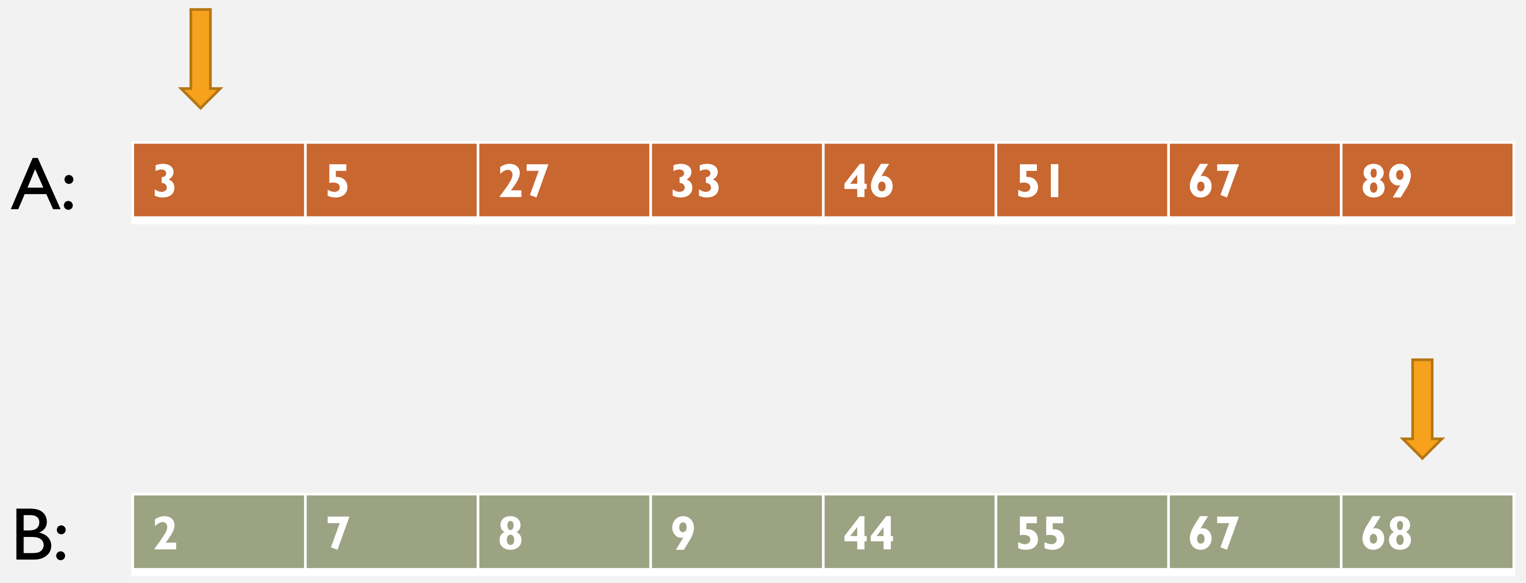
WALK FROM BOTH SIDES

A:	3	5	27	33	46	51	67	89
B:	2	7	8	9	44	55	67	68

Target: C[k] = 53

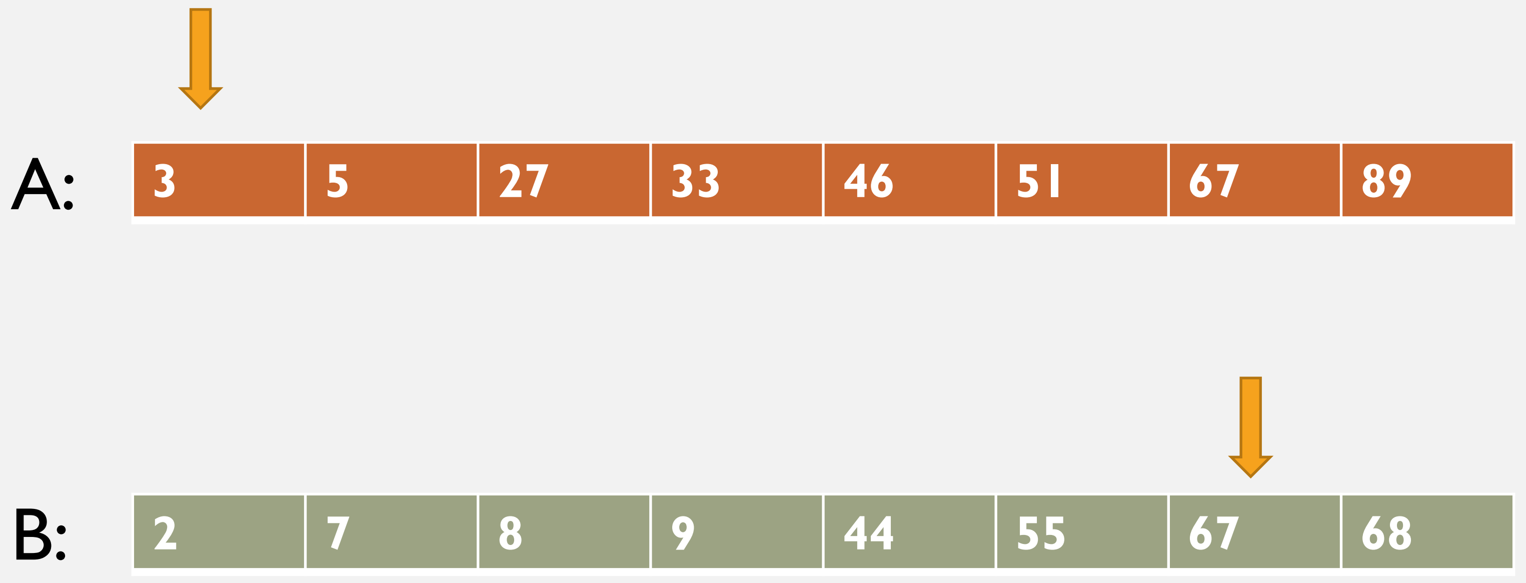
WALK FROM BOTH SIDES

68 + 3 = 71 > 53
So we decrement B’s pointer



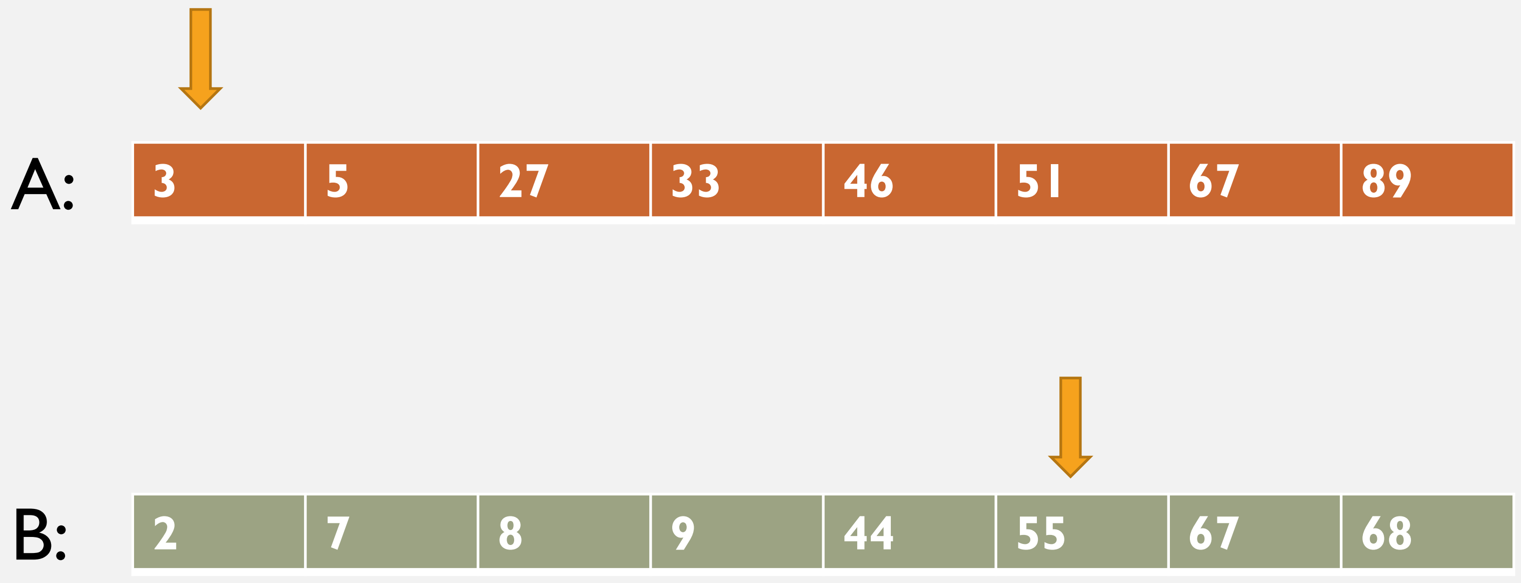
Target: C[k] = 53

WALK FROM BOTH SIDES



Target: C[k] = 53

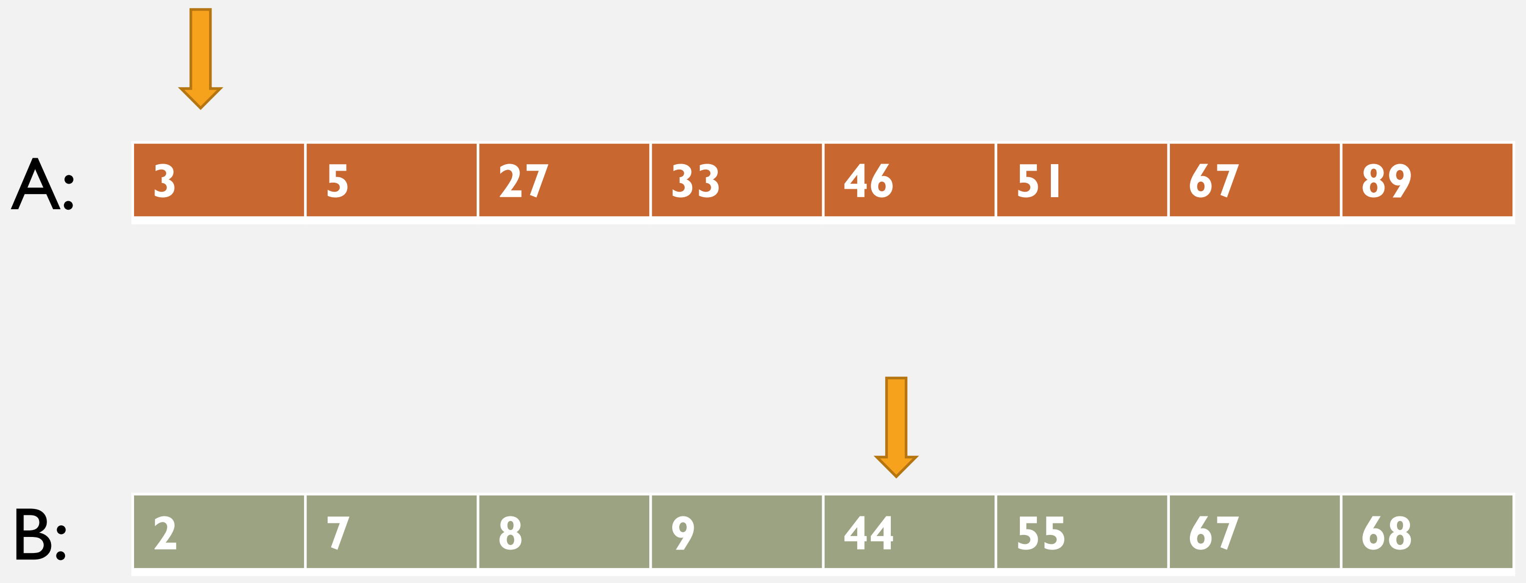
WALK FROM BOTH SIDES



Target: C[k] = 53

WALK FROM BOTH SIDES

44 + 3 = 47 < 53
So we increment A’s pointer



A:

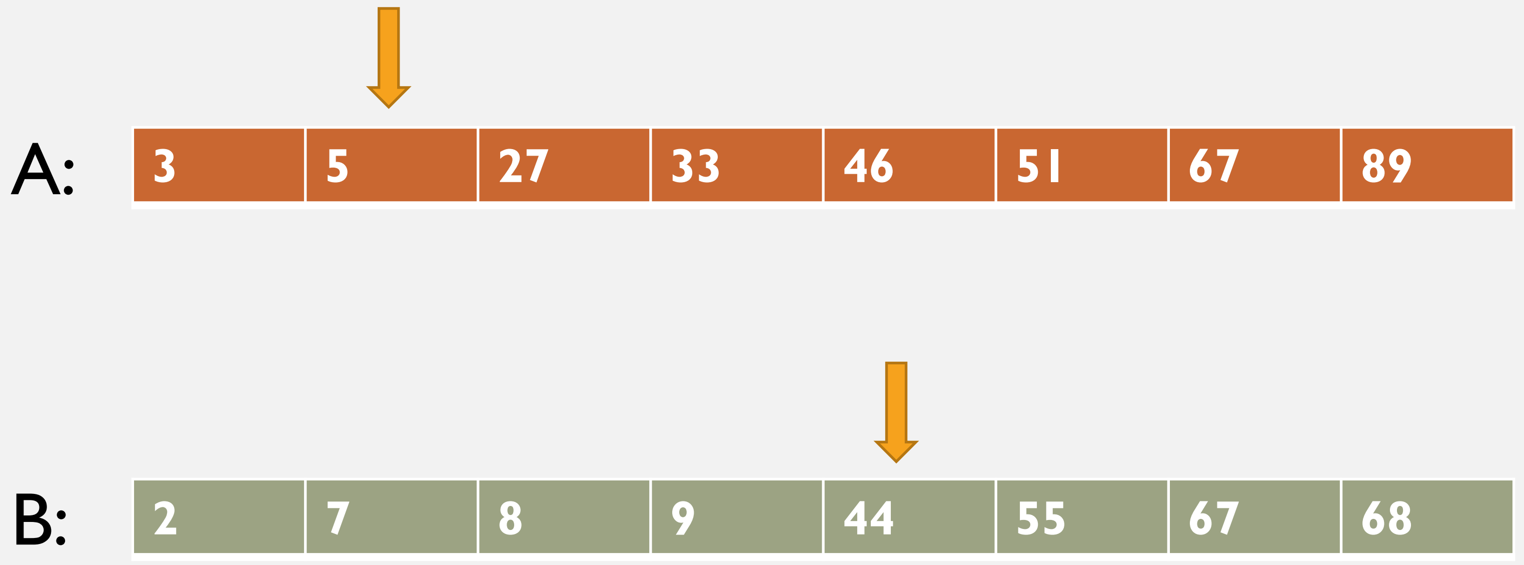
3	5	27	33	46	51	67	89

B:

2	7	8	9	44	55	67	68

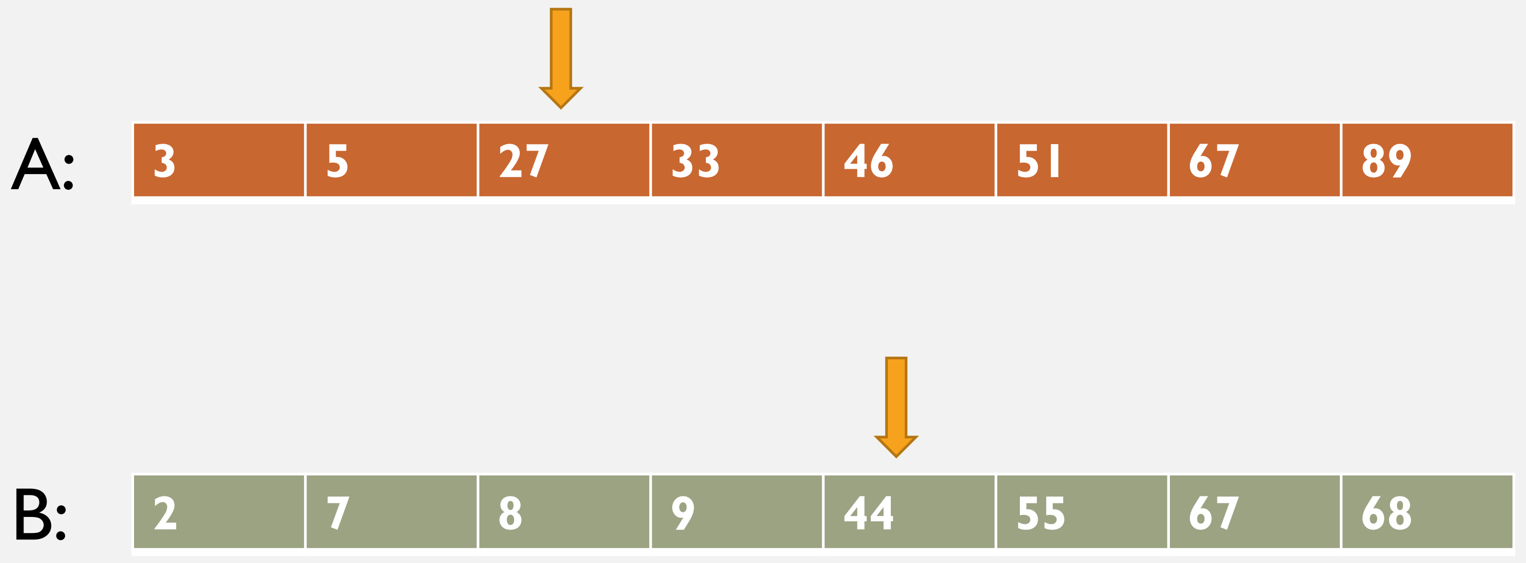
Target: C[k] = 53

WALK FROM BOTH SIDES



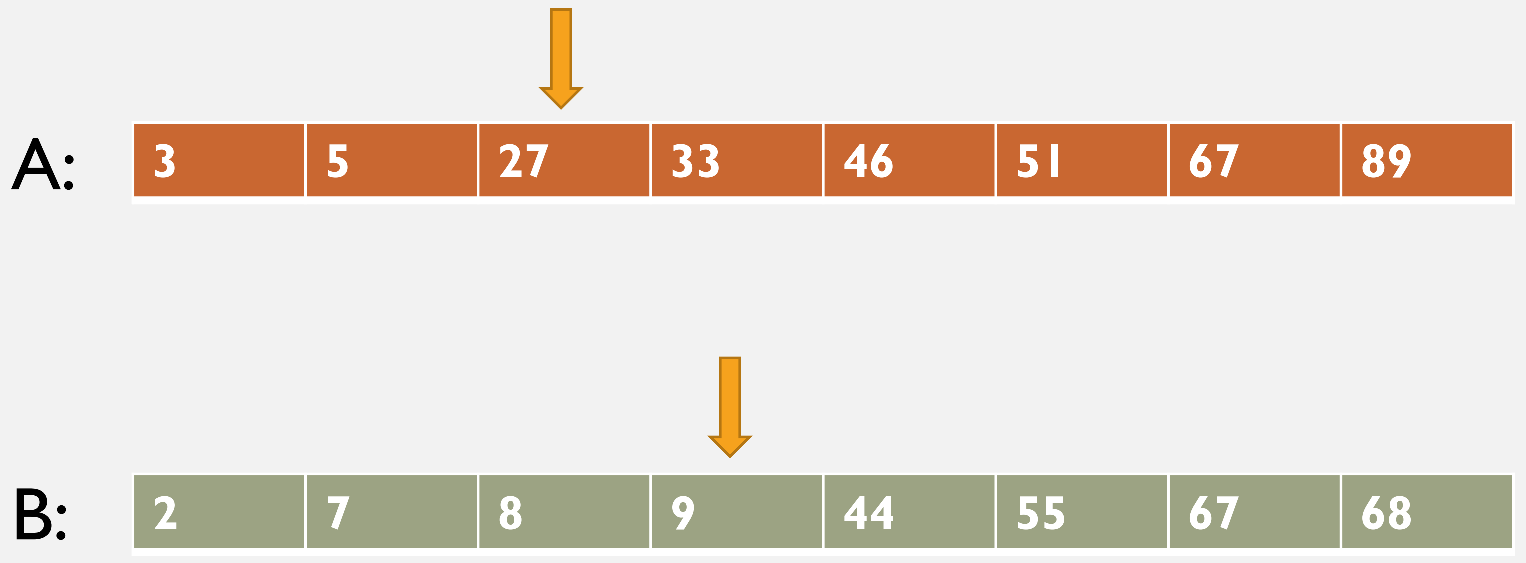
Target: C[k] = 53

WALK FROM BOTH SIDES



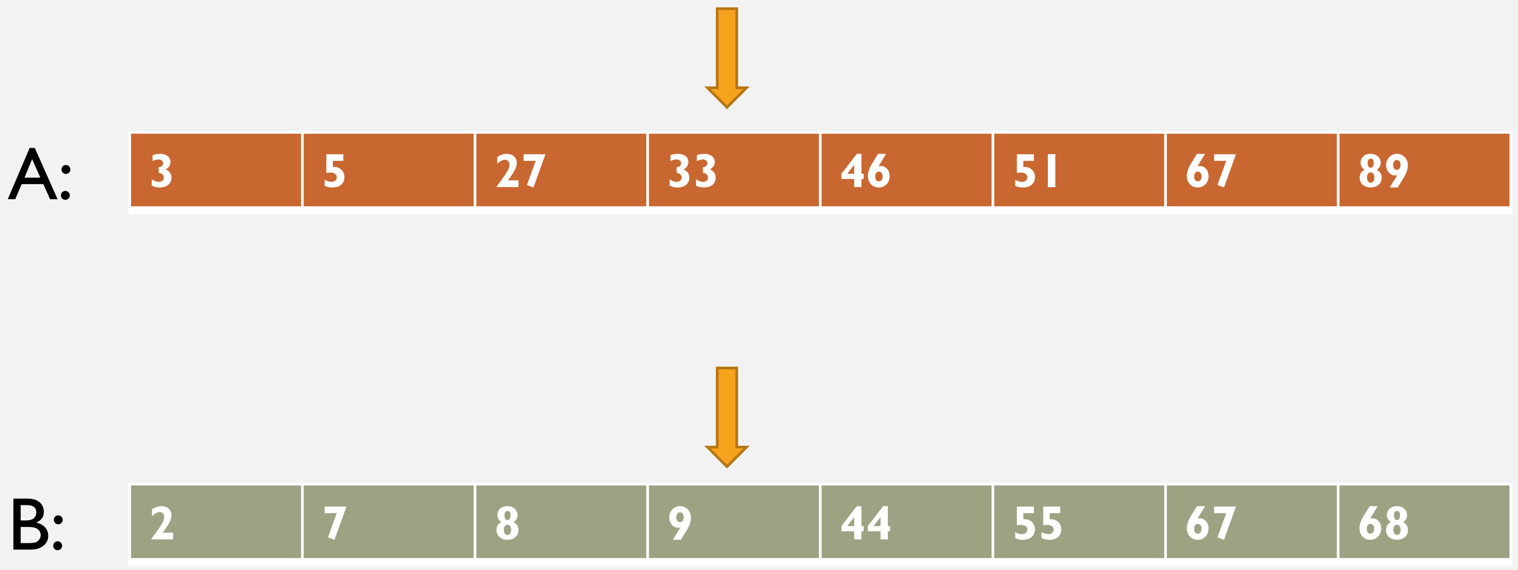
Target: C[k] = 53

WALK FROM BOTH SIDES



Target: C[k] = 53

WALK FROM BOTH SIDES



A:

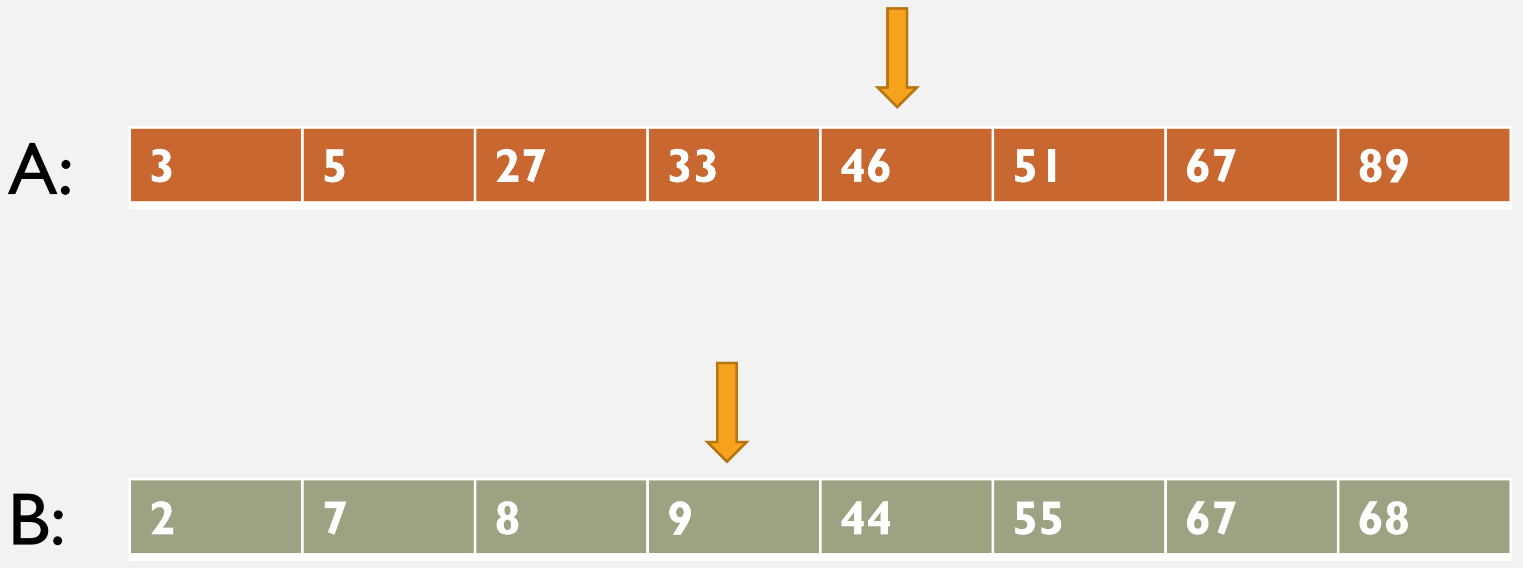
3	5	27	33	46	51	67	89

B:

2	7	8	9	44	55	67	68

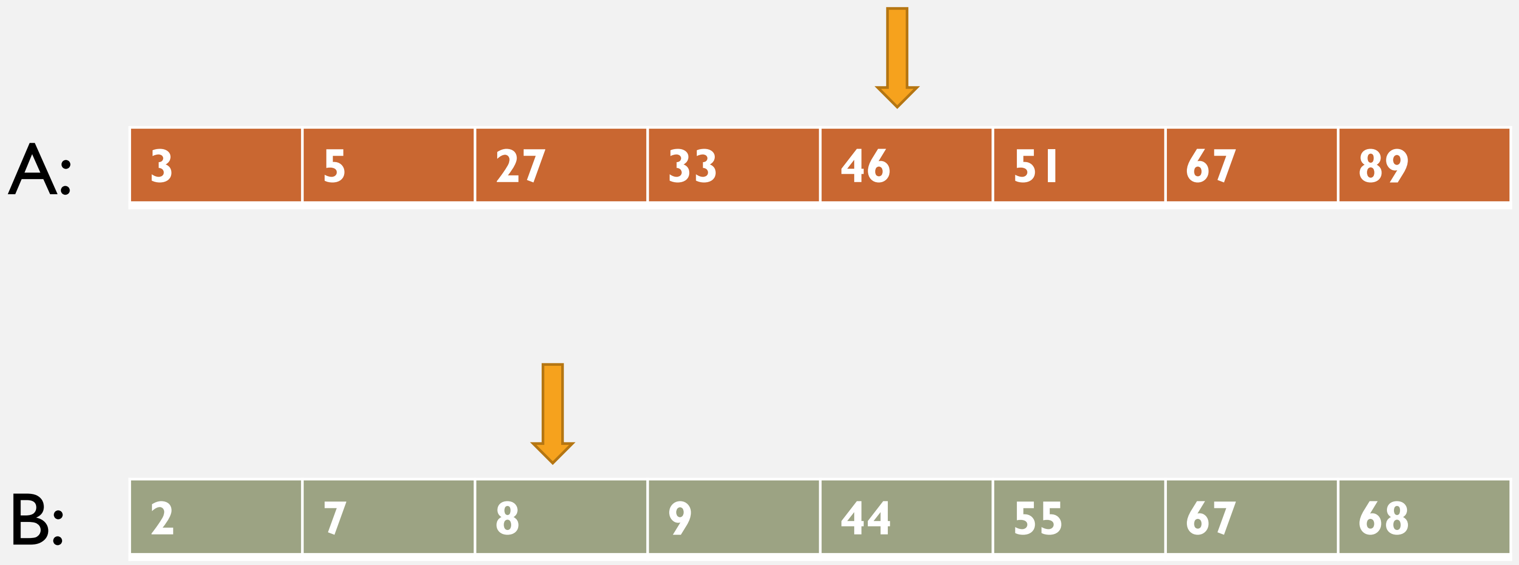
Target: C[k] = 53

WALK FROM BOTH SIDES



Target: C[k] = 53

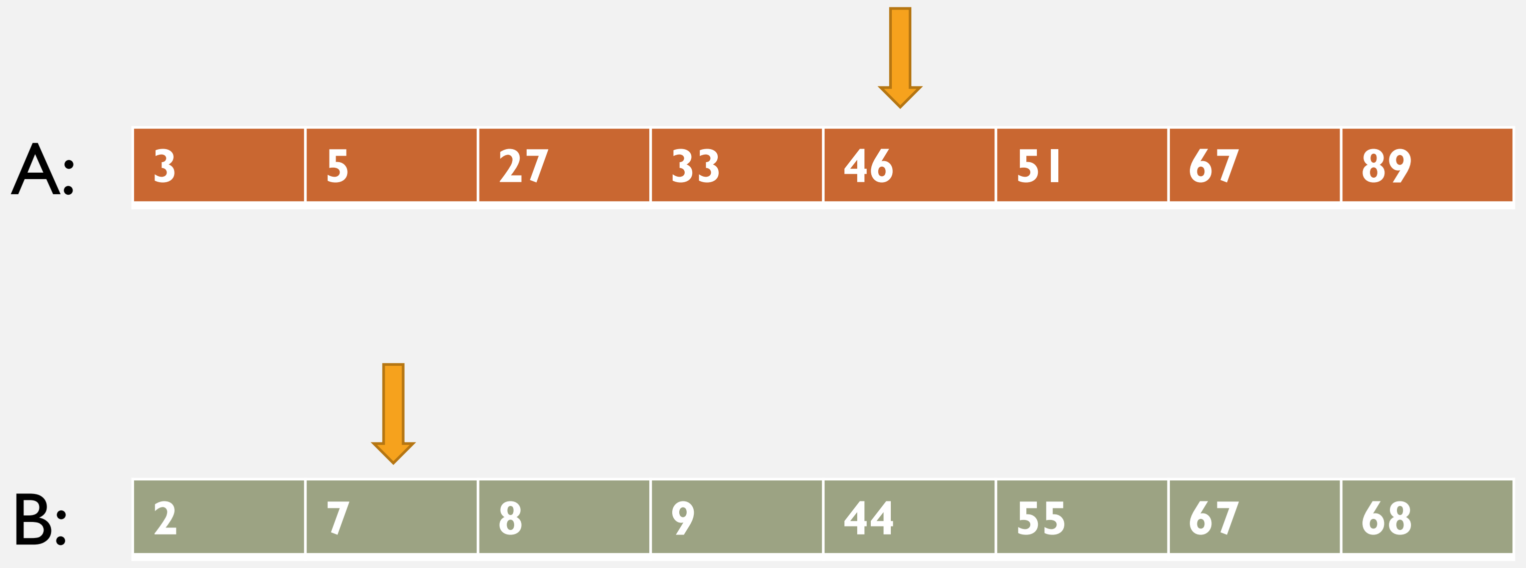
WALK FROM BOTH SIDES



Target: C[k] = 53

WALK FROM BOTH SIDES

Done!



A:	3	5	27	33	46	51	67	89
B:	2	7	8	9	44	55	67	68

Target: C[k] = 53

RUNNING TIME

- How long does all this take?
- Time to walk?
- How many values of C do we need to iterate over?

TAKING 3SUM FURTHER

- That was a cool algorithm! But it's a bit simple to implement
- We're implementing a version of 3-SUM that uses *tiling*. It has much better efficiency in terms of cache misses.

- (An aside: I believe this tiled version of 3-SUM will not be much faster, if it's faster at all. This midterm is about what you learned: taking a new algorithm, and turning it into efficient code.)

MAGIC HASH FUNCTION

```
uint64_t hash3(uint64_t value){
  return (value * 0x765a3cc864bd9779) >> (64 - SHIFT)
}
```

- Why is this magic?
- If X + Y = Z, then either:
  - `hash3(X) + hash3(Y) = hash3(Z)`
  - `hash3(X) + hash3(Y) = hash3(Z) + 1`

MAGIC HASH FUNCTION: EXPLANATION

```
uint64_t hash3(uint64_t value){
  return (value * 0x765a3cc864bd9779) >> (64 - SHIFT)
}
```

This number is not magic (any large odd number will work)

- You don’t need to know why this works. (Short version: modular arithmetic is linear)
- You DO need to know: how many values can this hash output?
  - Answer: `1 << SHIFT`

FINAL ALGORITHM

- Create `1 << SHIFT` hash buckets for A, called BucketA
- For each item x in A, store x in bucket `BucketA[hash3(x)]`
- Create `1 << SHIFT` hash buckets for B, called BucketB
- For each item x in B, store x in bucket `BucketB[hash3(x)]`
- Create `1 << SHIFT` hash buckets for C, called BucketC
- For each item x in C, store x in bucket `BucketC[hash3(x)]`

FINAL ALGORITHM

```
For a = 1 to (1 << SHIFT)
    For b = 1 to (1 << SHIFT)
        Call the simple 3SUM algorithm with lists: BucketA[a], BucketB[b], BucketC[(a + b) (modulo 1 << SHIFT)]
        Call the simple 3SUM algorithm with lists: BucketA[a], BucketB[b], BucketC[(a + b + 1) (modulo 1 << SHIFT)]
```

QUICK COMMENTS

- How to store hash buckets?
  - You don't know the size ahead of time
  - But, want to be cache-efficient within each bucket
- Need to find *original* (unsorted) value
- Which of the operations in this algorithm are expensive? Can they be avoided, or turned into cheaper operations?
- This midterm is not graded on benchmarks. Instead, I want you to implement and discuss the above.
- Rule of thumb: ~2 good optimizations

LAST EFFICIENCY TOPIC: LOOP UNROLLING

LOOP UNROLLING

- Last time we saw some basic downsides of loops:
  - Conditionals can be costly
  - Can have costly (non-obvious) dependencies
- Loop unrolling is a common technique to improve performance
- Idea: do multiple iterations of the loop within the loop body. This saves a conditional check! It can also allow for better performance (out of order execution, etc.)

LOOP UNROLLING: EXAMPLE

LOOP UNROLLING: DISCUSSION

- Compiler will do this for you (if it can and it thinks it might help) if you set flag -funroll-loops
  - I can't get it to help with the example
- Can be downsides: code becomes larger (might cause worse instruction cache performance)
  - Space-speed tradeoff
- Could hinder other compiler optimizations

PROBABILITY AND HASHING

NEXT SECTION OF COURSE

- Randomized algorithms
- Focus on small-space algorithms, usually involving hashing
- Why?
  - As you may know, worst-case compression is impossible
  - Randomness lets us get worst-case (algorithmic) bounds, while still getting really good performance in terms of space
- Cool part: these algorithms are pretty recent
  - Many within your lifetime
  - Most within mine

QUICK ASIDE

- Your midterm does have a hash
- BUT it’s not random
- Should not be treated like a hashing problem
- Don’t store your items using chaining or linear probing!
    - Make buckets like we discussed

SIMPLE EXAMPLE: DICTIONARY

- You’ve all seen in 136 and/or 256
- Idea: want to store n key/value pairs
- Query: given a key, get the associated value in the dictionary

- How fast can we do this?
  - $O(1)$ expected time, if using $O(n)$ space

HASH ASSUMPTIONS

For now: let's assume I have access to a fully-random hash function h

Is this a good assumption?

- We'll discuss later: tradeoff between randomness and evaluation time

DICTIONARY

To store n items:

- Allocate a table (perhaps an array) of size cn for some c
- Store item x at position $h(x)$ % cn

To find an item q:

- Look it position $h(q)$ % cn
- Always finds it if it exists!

HANDLING COLLISIONS

- What happens if several items hash to the same location?
- Chaining
- Linear Probing
- Double hashing

CHAINING

- Set $c = 1$
- Each entry in your table is the head of a singly linked list
- Follow the linked list to find your item
- Advantages?
  - Not too much extra space (just need pointers for linked list)
  - Simple
  - Slight advantage in worst-case behavior
- Disadvantages?
  - Cache-inefficient

LINEAR PROBING

- Set $c > 1$ (often have $c = 1.5$ or $c = 2$ in practice)
- Table is just an array of stored items
- If the slot is full, keep moving down the table to find the next empty slot
- When querying, need to keep checking until you find the item or an empty slot
- Advantages?
  - Pretty space-efficient, reasonable runtime
  - Cache-efficient
- Disadvantages?
  - Not that space- or time-efficient
  - Can have incredibly bad lookup if table fills up

“DOUBLE HASHING”

- Similar to linear probing
- Now, instead of moving to next slot, have a second (random) hash h_2
- If slot $h(x)$ is full, check $(h(x) + h_2(x))$ % cn, then $(h(x) + 2h_2(x))$ % cn, and so on
- Advantages?
  - Finds empty slot faster than linear probing
- Disadvantages?
  - Terrible cache efficiency again!

ANALYSIS

- All three check $O(1)$ items and answer a query in $O(1)$ time in expectation
- Linear probing has $O(\log n)$ worst case, chaining has $O(\log n\ /\ \log\ n)$

If your hash function is random, then for any set of items and every query, the average query time (taking the average over the choice of your hash function) is $O(1)$

ANALYSIS

- All three check $O(1)$ items and answer a query in $O(1)$ time in expectation
- Linear probing has $O(\log n)$ worst case, chaining has $O(\log n \,/\, \log\log n)$

This isn't QUITE worst case, but the probability of being worse than this is at most $1/n^2$

WHAT DO PEOPLE USE?

- Depends on the use case
- But, oftentimes, linear probing
    - Chaining can good when you have a cache miss anyway due to the pointer, and/or the items are very large

- One more strategy that does get used sometimes: cuckoo hashing

CUCKOO HASHING

- Pagh, Rodler 2005
- Lookup time is $O(1)$ in the *worst case!*
- Insert/delete is $O(1)$ in expectation
  - $O(\log n)$ worst case (same caveat as before)

CUCKOO HASHING INVARIANT

- Have two hash functions h_1, h_2
- Table of size cn with $c = 2$
- Invariant: item x is stored either at slot $h_1(x)\%\ cn$, or at slot $h_2(x)\%\ cn$
- I'll talk about inserts in a second
- How do we query?
- How much time does that take?

CUCKOO HASHING INSERTS

- Let's put a new item x into our hash table. How?
- Easy case: if slot $h_1(x)\%\ cn$ or $h_2(x)\%\ cn$ is free, can just store x
- What if they're both full?
- Answer: pick one of the slots. Kick the item stored there out; store it using its other hash
  - Hence "cuckoo"

Cuckoos kick other birds' eggs out of the nest, replacing them with their own

CUCKOO HASHING EXAMPLE

Table:

	5		93		51		89

CUCKOO HASHING EXAMPLE

Table:

	5		93		51		89

Insert new item: 33. $h_1(33) = 0, h_2(33) = 3$

CUCKOO HASHING EXAMPLE

Table:

33	5		93		51		89

CUCKOO HASHING EXAMPLE

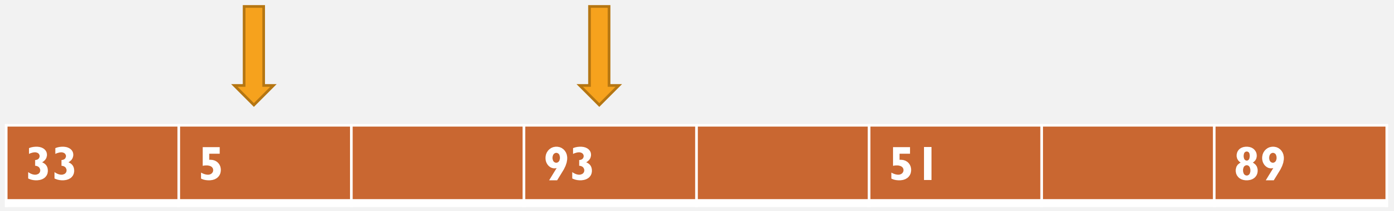
Table:

33	5		93		51		89

Insert new item: **49.** $h_1(49) = 3, h_2(49) = 1$

CUCKOO HASHING EXAMPLE

Table:



Insert new item: **49.** $h_1(49) = 3, h_2(49) = 1$

CUCKOO HASHING EXAMPLE

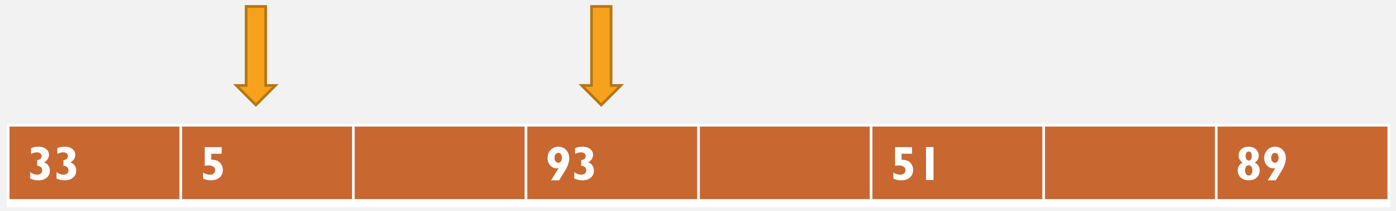


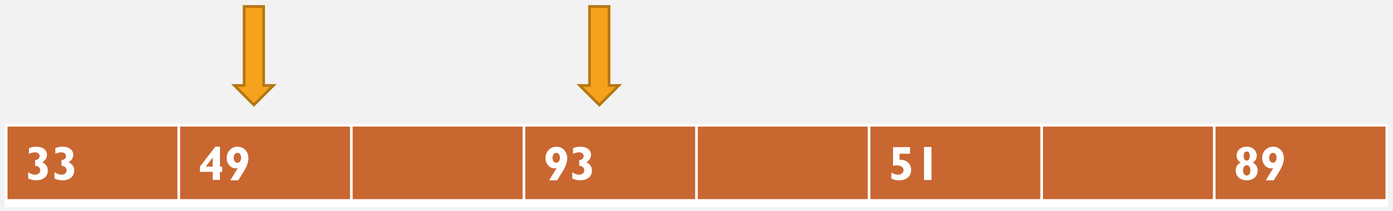
Table:

33	5		93		51		89

Let's choose to kick out 5, replacing it with 49

CUCKOO HASHING EXAMPLE

Table:

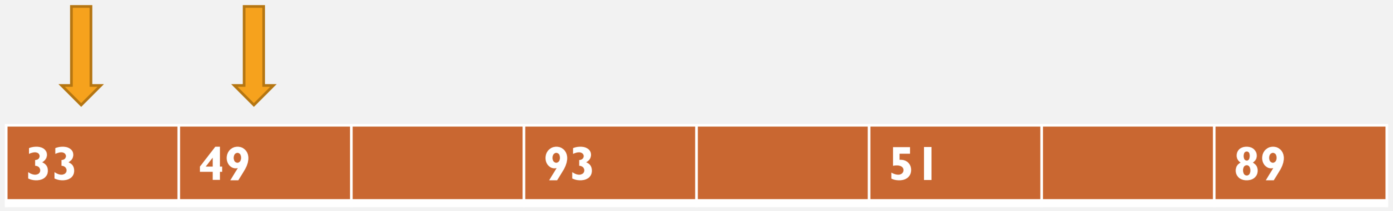


33	49		93		51		89

Now we need to find a place for 5

CUCKOO HASHING EXAMPLE

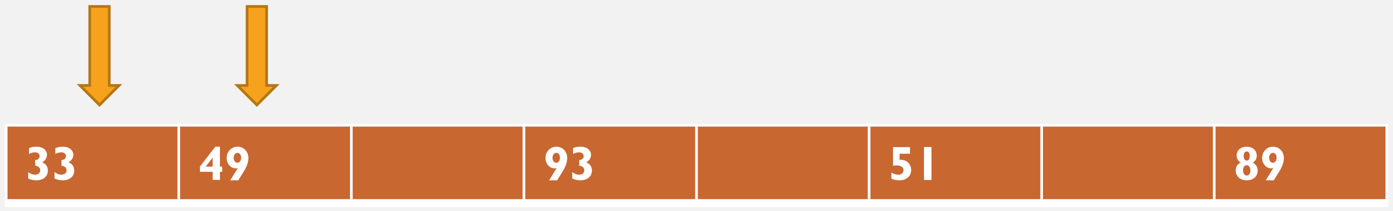
Table:



Reinsert item: 5. $h_1(5) = 0, h_2(5) = 1$

CUCKOO HASHING EXAMPLE

Table:



We have to kick out 33. (Don't want to loop back)

CUCKOO HASHING EXAMPLE

Table:

5	49		93		51		89

Reinsert item: **33**. $h_1(33) = 0, h_2(33) = 2$

CUCKOO HASHING EXAMPLE

Table:

5	49	33	93		51		89

Reinsert item: **33**. $h_1(33) = 0, h_2(33) = 2$

CUCKOO HASHING EXAMPLE

Table:

5	49	33	93		51		89

Done!

CUCKOO HASHING ANALYSIS

- Insert: expected number of swaps is $O(1)$
- Largest number of swaps is $O(\log n)$

- Wait a minute…does this always work?
  - No.

CUCKOO HASHING FAILURE

Table:

33	5						

$$h_1(33) = 0, h_2(33) = 1$$
$$h_1(5) = 0, h_2(5) = 1$$
$$h_1(17) = 0, h_2(17) = 1$$

We can't maintain the invariant!!!

CUCKOO HASHING FAILURE

- [PR'05]: the probability of failure is only $O(1/n)$
- What do we do if we fail???
  - Pick new hash functions and start from scratch
  - (Ouch)

CUCKOO HASHING

- Advantages?
  - Great worst-case performance on queries
  - Only two cache misses on queries
  - Fairly simple
- Disadvantages?
  - Rebuilds are a huge issue!
  - Two cache misses can be much worse tha near probing
  - On inserts, every swap of an ele ent is another cache miss
  - Space usage is not great

You can avoid this by storing a constant number of elements in each slot (say 4)

HASH FUNCTION DISCUSSION

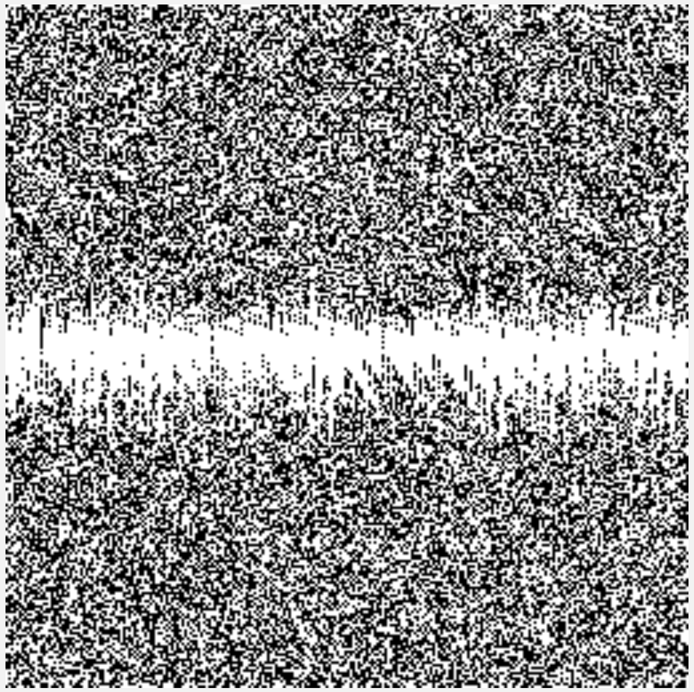
- Fully-random hash functions are very unreasonable
  - Take forever to evaluate, and/or take tons of space
- Practice often uses linear feedback registers
- Pseudorandom number generator that does some shifts, adds, etc. to the number each time
- Looks pretty random!
- Super fast

LINEAR FEEDBACK REGISTERS DOWNSIDES

- Can have tricky correlations between elements
  - The bounds I mentioned may not hold!
  - Unlikely in practice. (But very possible! Especially if you really want small constants)

LINEAR FEEDBACK REGISTERS DOWNSIDES

- Need to make sure the stored numbers don’t get too small



CRYPTOGRAPHY

- We know how to encode stuff pretty well
- It’s basically impossible to decode---information about the output tells us nothing about the input whatsoever
- Doesn’t that mean it’s about as “random” as it can get?
  - Yep.
  - So why don’t we use it?

CRYPTOGRAPHIC HASH FUNCTIONS

- Easily obtainable, work very well
- Important for security
- Downside: generally high cost

NEXT CLASS

- Some examples of these in practice
- Bloom filters/quotient filters!