

CSC148 – Introduction to Computer Science

Lecture 10: Introduction to Sorting

Sean Henderson

Sorting

- Given an array $A[0, \dots, n-1]$
- Permute the elements of A such that $A[0] \leq A[1] \leq \dots \leq A[n-1]$
- Comparison based sorting
 - Don't know anything about what we are sorting
 - Have access to a “black box” which lets us compare, two elements ($A[i] < A[j]$, $A[i] == A[j]$, $A[i] > A[j]$)

Classic $O(n^2)$ Sorting Algorithms

- Bubble Sort
- Insertion Sort
- Selection Sort
- Of course, we've already seen better sorting algorithms...

Bubble Sort

- Idea: find two adjacent elements which are out of order ($A[i] > A[i+1]$) and swap them.
- Repeat this until no adjacent elements are out of order
- So $A[0] \leq A[1]$, and $A[1] \leq A[2]$, and $A[2] \leq A[3]$...
- Implies $A[0] \leq A[1] \leq A[2] \leq \dots$
- The list is sorted

Bubble Sort

- How can we prove this is correct?
- Define an *inversion* to be a pair (i,j) with $i < j$ such that $A[i] > A[j]$
- How many inversions can there be in a list with n elements?
- If there are no inversions, the list is sorted
- Each swap in bubble sort decreases the number of inversions by 1

Insertion Sort

- Suppose our list $A[0..i-1]$ is sorted, and we want to insert $A[i]$ into
- Repeatedly swap $A[i]$ with its predecessor until $A[0..i]$ is sorted
- Similar to the lab with inserting into a sorted linked list (actually, exactly this) and inserting into a heap (insertion sort on the path from root to leaf)

Selection Sort

- Similar to insertion sort
- We will maintain the invariant that after iteration i of the for loop that $A[0..i]$ is sorted and will never get updated
- Then find the smallest element of $A[i+1..(n-1)]$ and move it to position $A[i+1]$
- Each time we “select” the next element

Better Sorting Algorithms

- Heapsort
- Mergesort
- Quicksort
- Introsort

Heapsort

- Insert everything into a heap, and `extract_min` repeatedly
- Basically selection sort, but using a heap instead of a linear search
- Problem: this version requires making a copy of our list (internally in our heap implementation)

Heapsort

- We would like to modify heapsort so that it is *in-place*
- In-place: rearrange the ordering of the elements in the list we are given instead of creating a new list
- Use a backwards insertion sort
 - First block of the list contains our heap
 - The back of the list is sorted (stores what we removed)

Mergesort

- Recursive sorting algorithm
- Idea: we can efficiently merge two sorted lists into one sorted list

Merge

- Given two sorted lists, $A[0:n]$ and $B[0:m]$
 - $A[0] \leq A[1] \leq \dots \leq A[n-1]$
 - $B[0] \leq B[1] \leq \dots \leq B[m-1]$
- Want to create a new list $C[0:n+m]$, a sorted list containing all the elements of A and B combined
- How can we create C ?

Merge

- Bad Idea: Copy all the elements of A and B into C, and then run some sorting algorithm (say, heapsort) on C.
- Better idea: given A and B, what value do we want to put in C[0]?
- What about C[1]?

Example

1	1	2	3	5	8
---	---	---	---	---	---

A

--	--	--	--	--	--	--	--	--	--	--	--

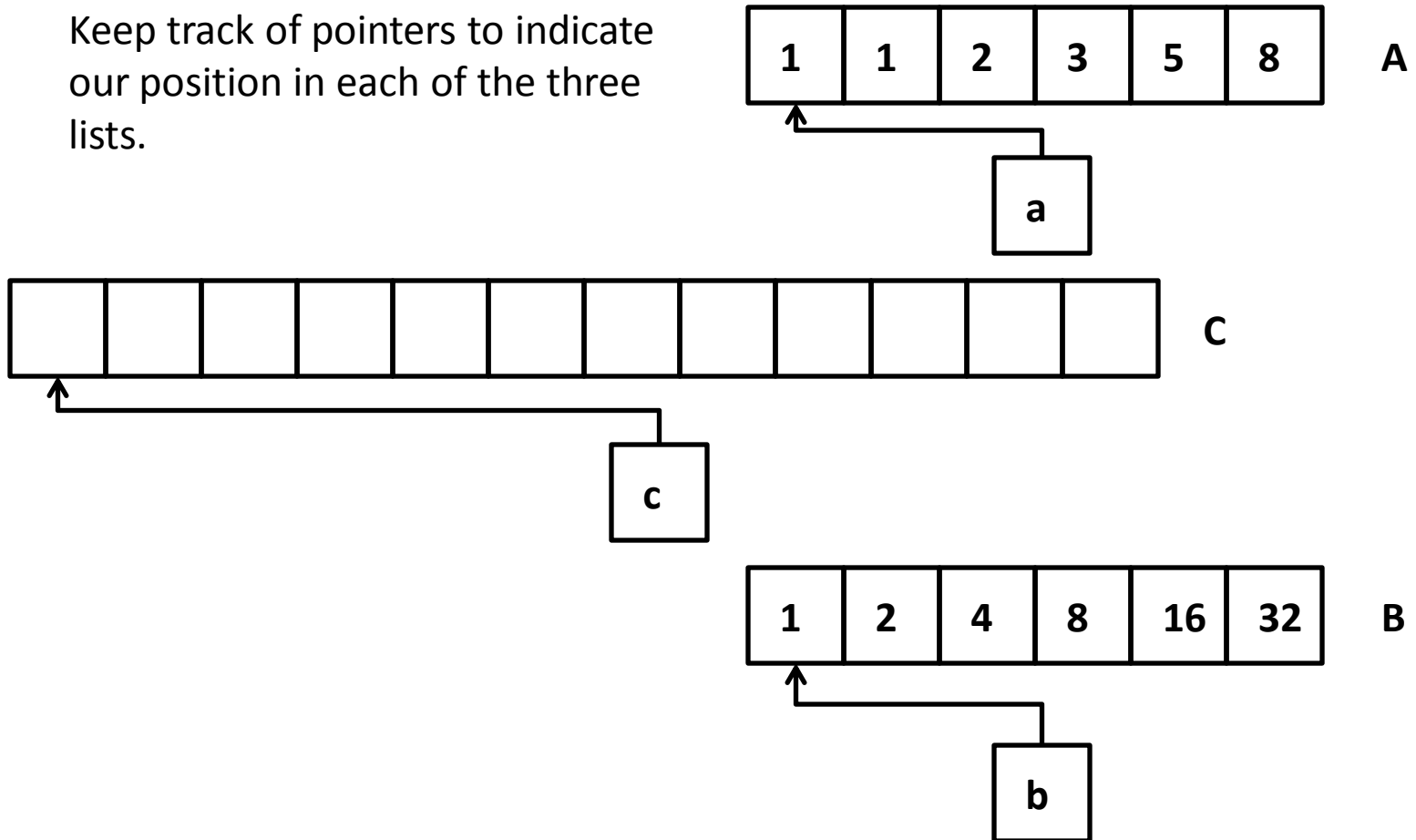
C

1	2	4	8	16	32
---	---	---	---	----	----

B

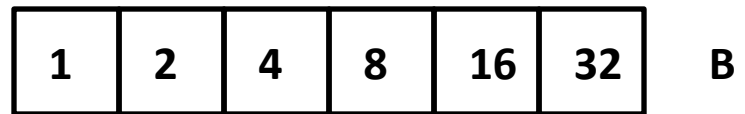
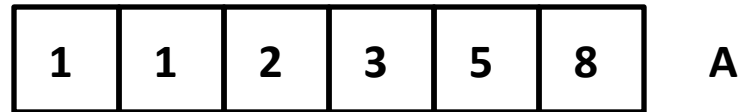
Example

Keep track of pointers to indicate our position in each of the three lists.



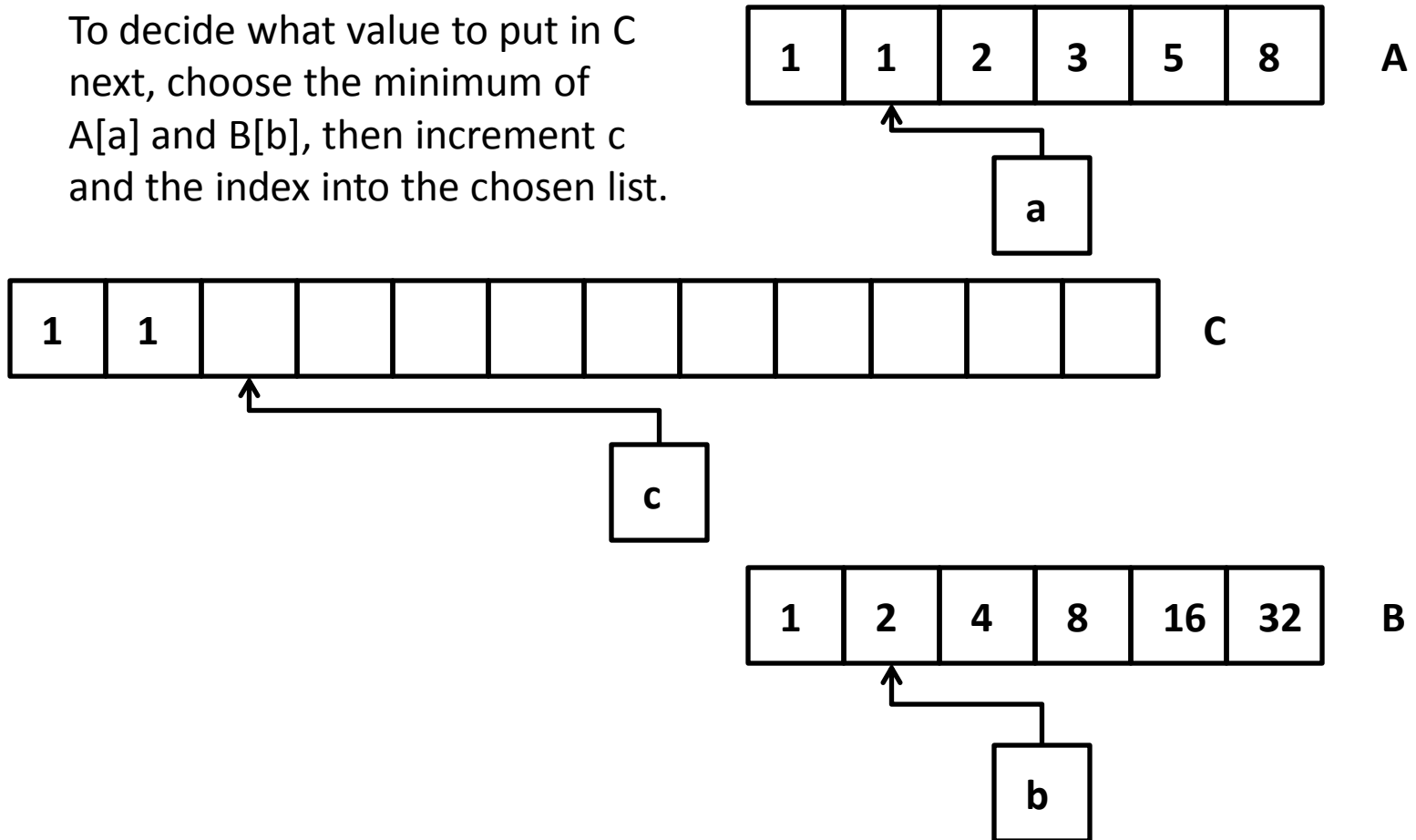
Example

To decide what value to put in C next, choose the minimum of $A[a]$ and $B[b]$, then increment c and the index into the chosen list.



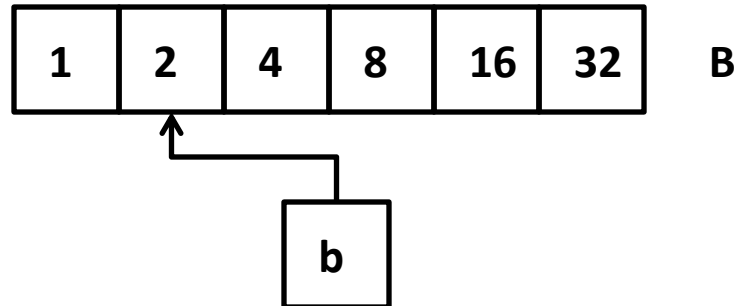
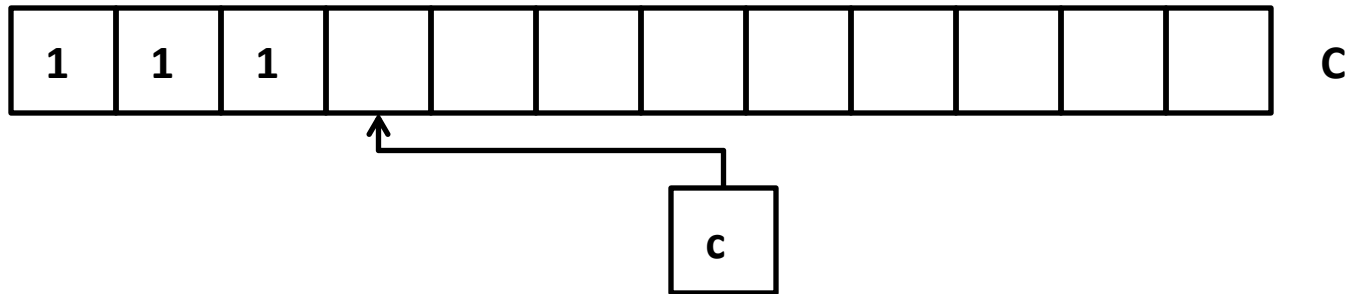
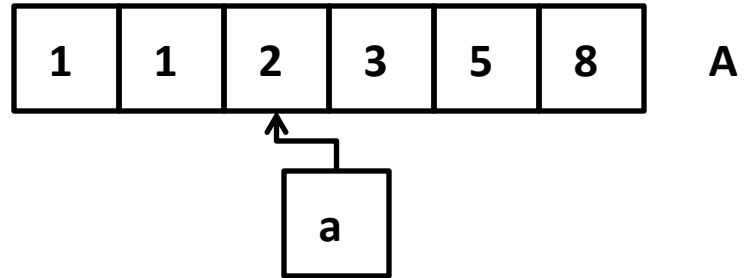
Example

To decide what value to put in C next, choose the minimum of $A[a]$ and $B[b]$, then increment c and the index into the chosen list.



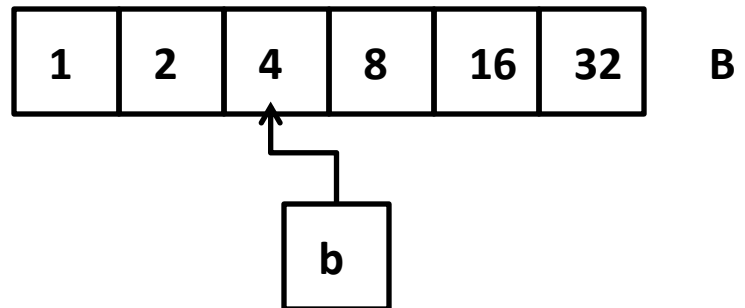
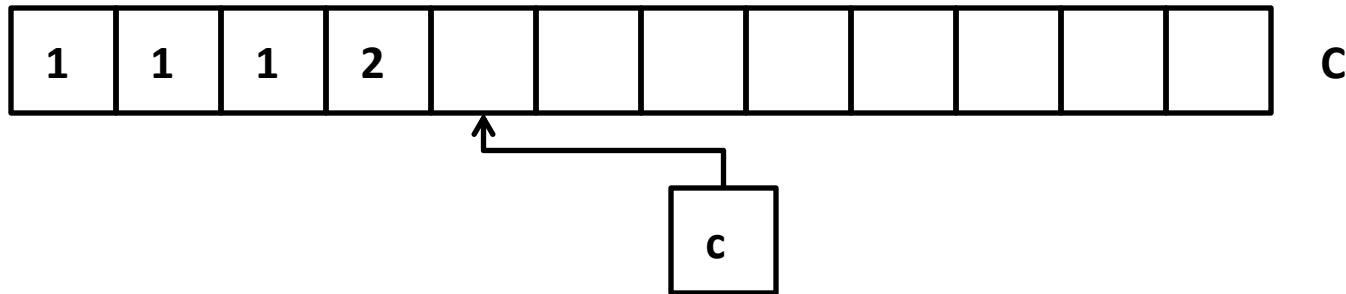
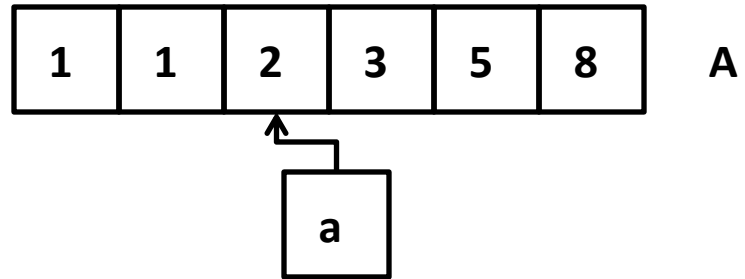
Example

To decide what value to put in C next, choose the minimum of $A[a]$ and $B[b]$, then increment c and the index into the chosen list.



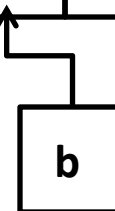
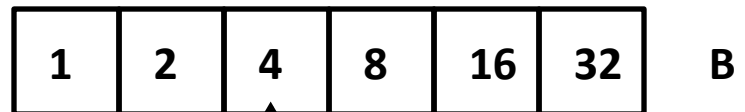
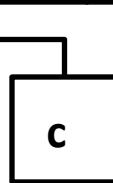
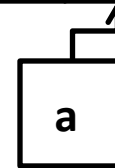
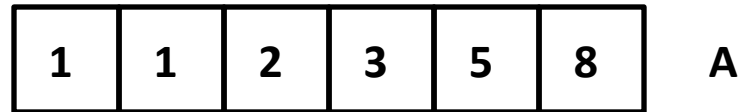
Example

To decide what value to put in C next, choose the minimum of $A[a]$ and $B[b]$, then increment c and the index into the chosen list.



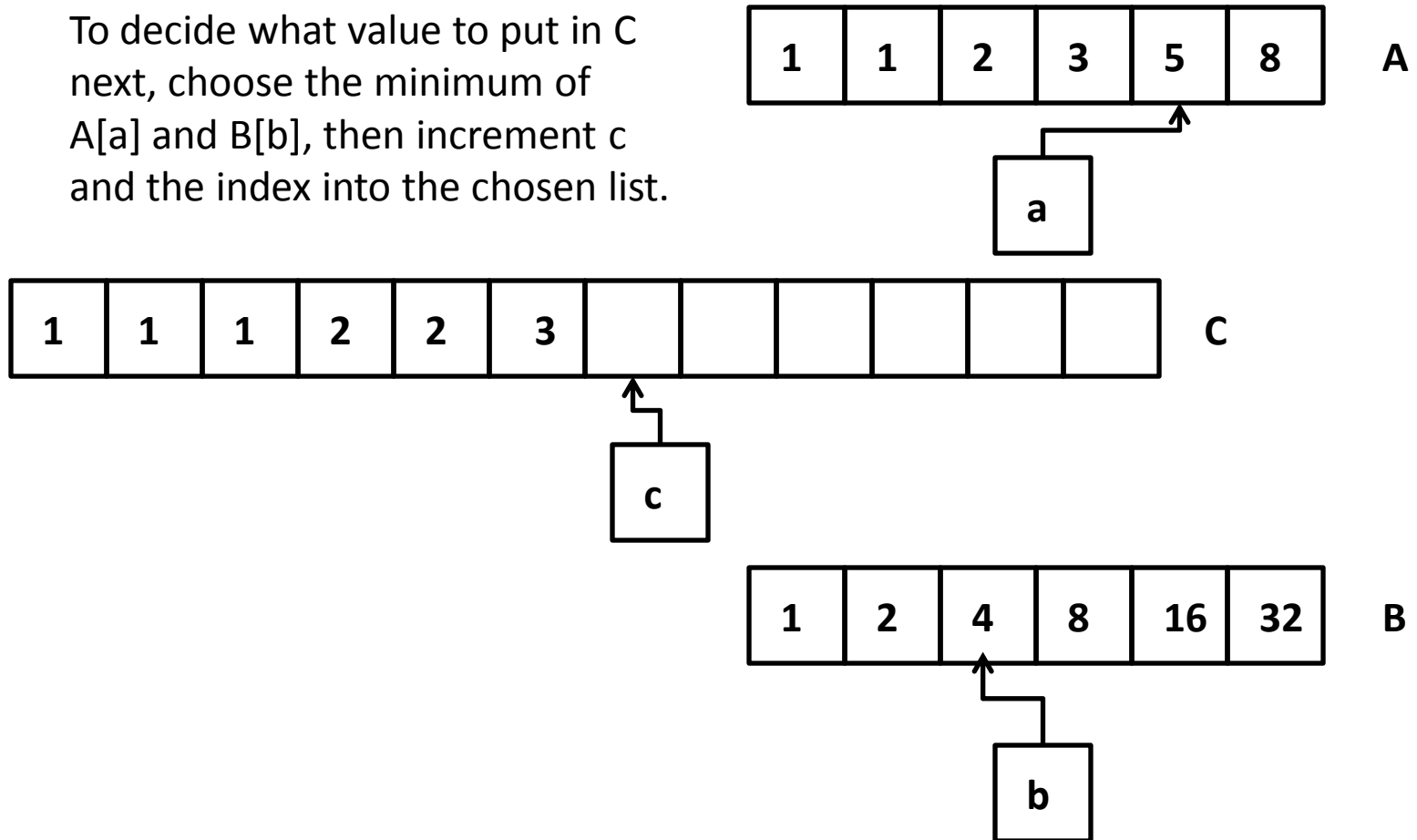
Example

To decide what value to put in C next, choose the minimum of $A[a]$ and $B[b]$, then increment c and the index into the chosen list.



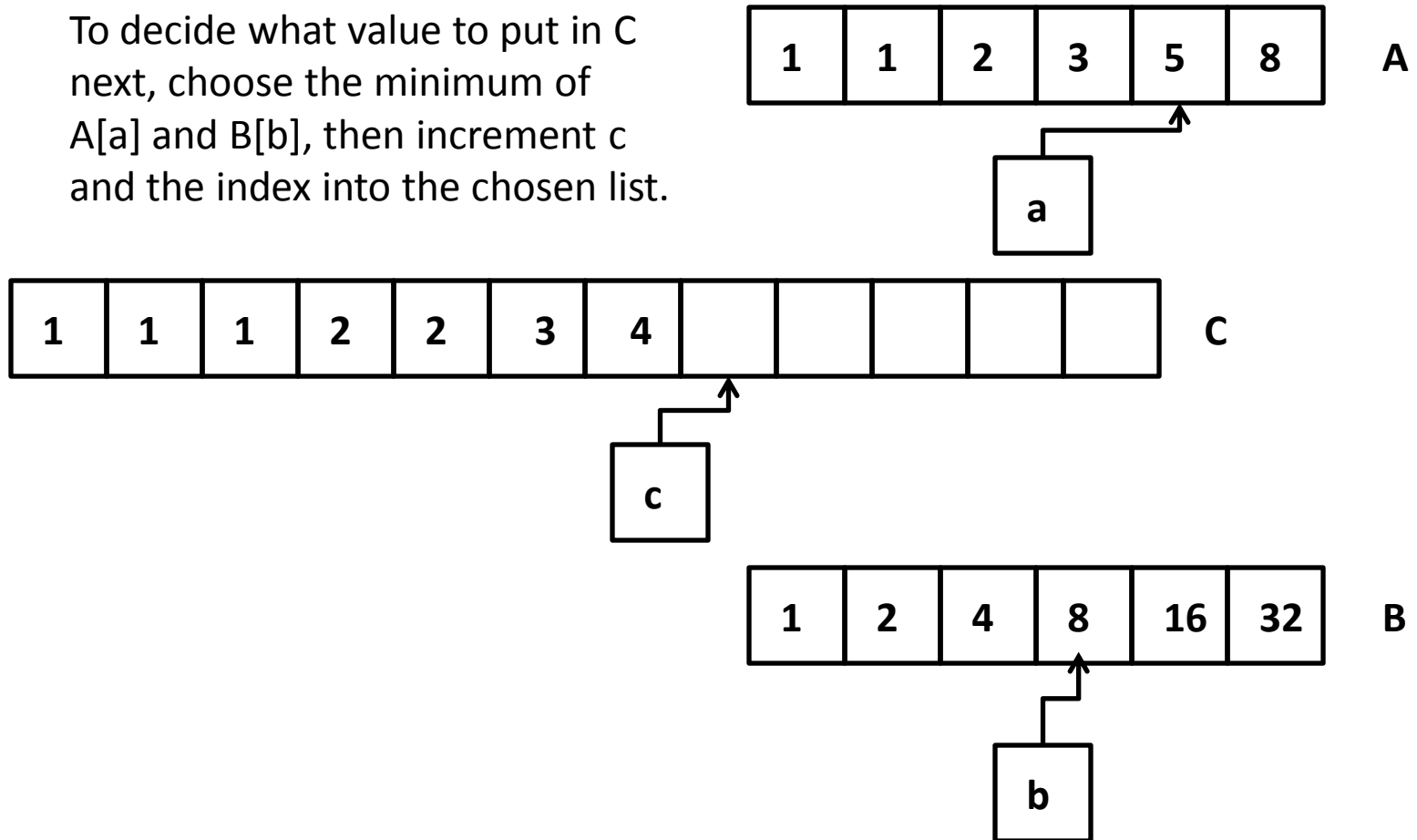
Example

To decide what value to put in C next, choose the minimum of $A[a]$ and $B[b]$, then increment c and the index into the chosen list.



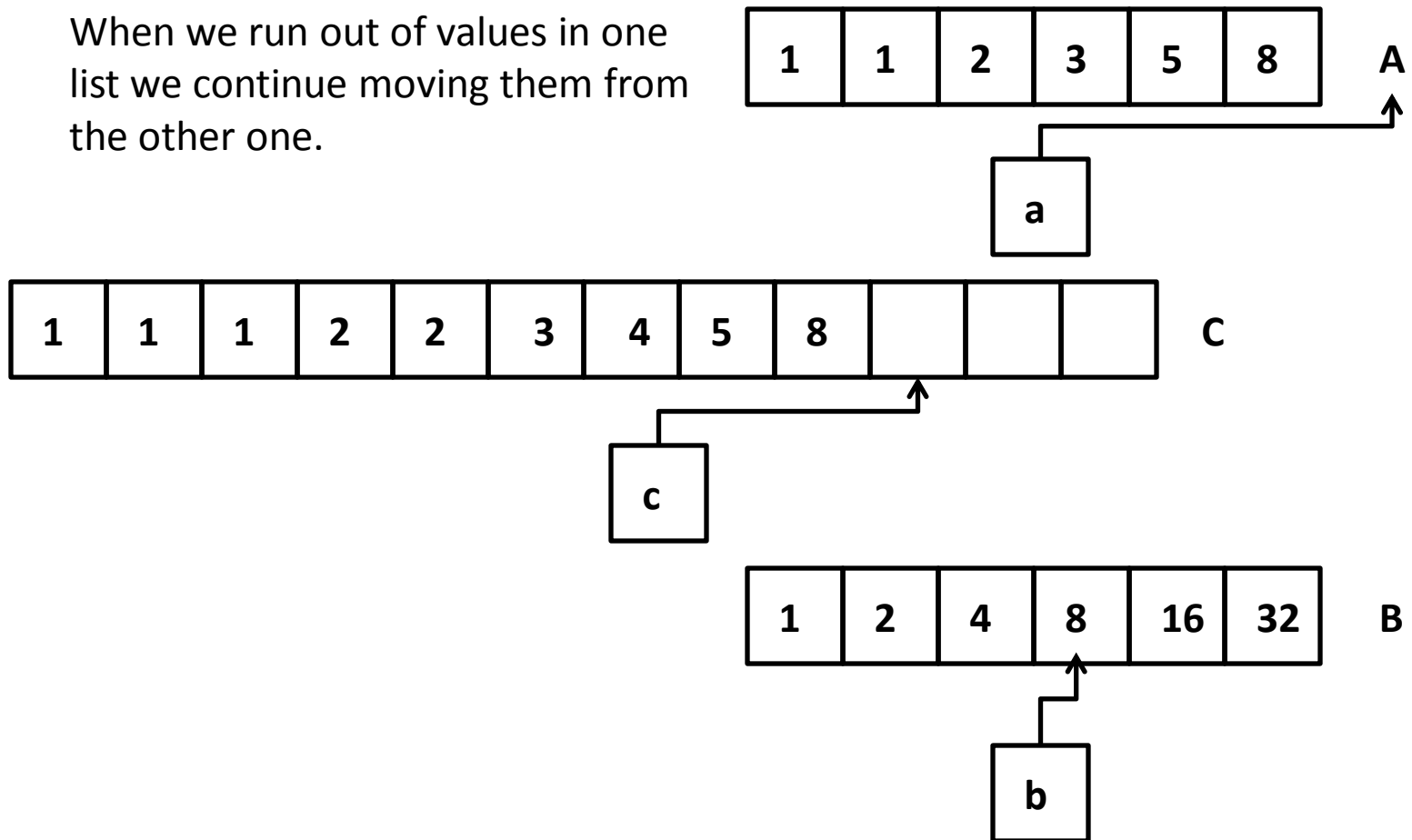
Example

To decide what value to put in C next, choose the minimum of $A[a]$ and $B[b]$, then increment c and the index into the chosen list.



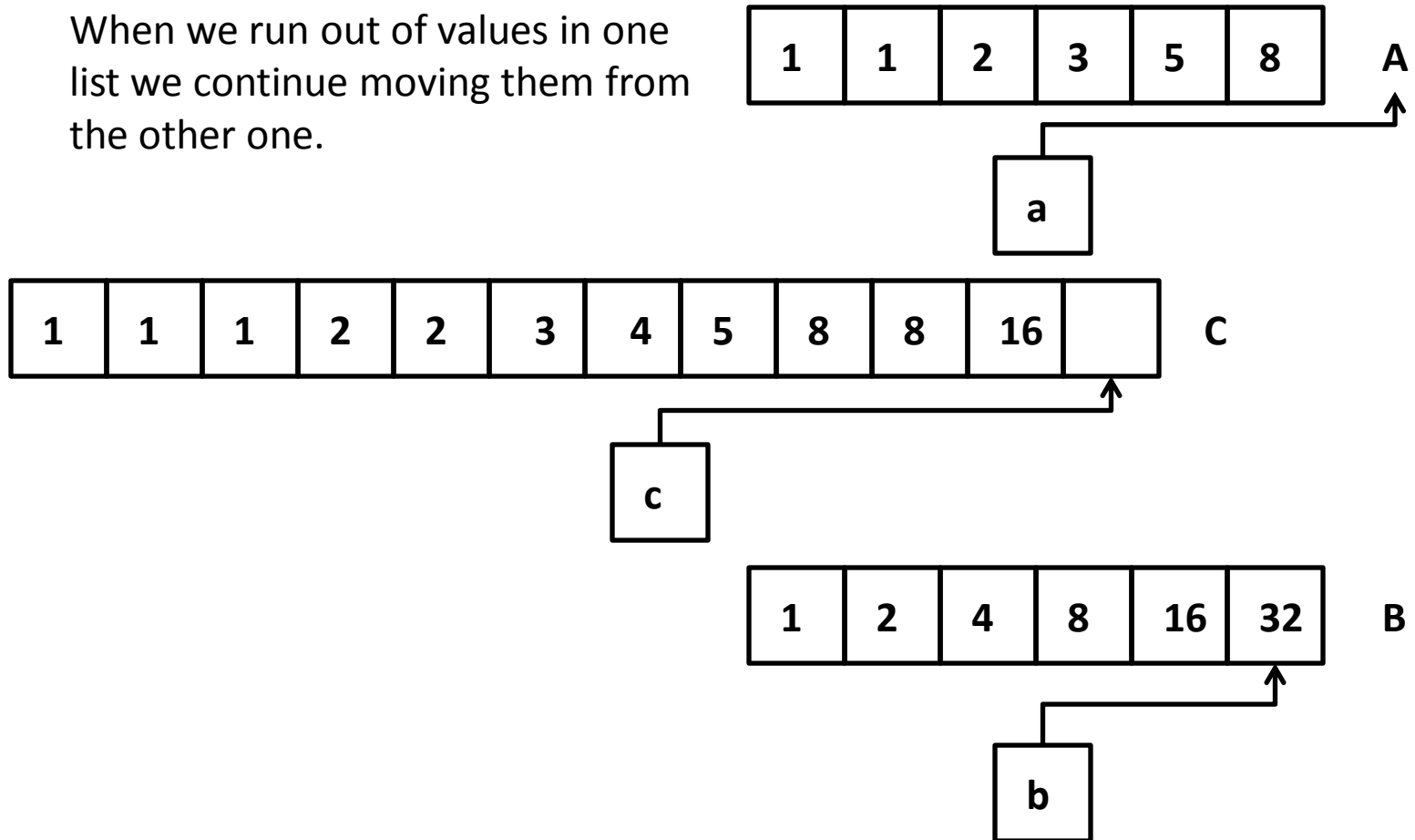
Example

When we run out of values in one list we continue moving them from the other one.



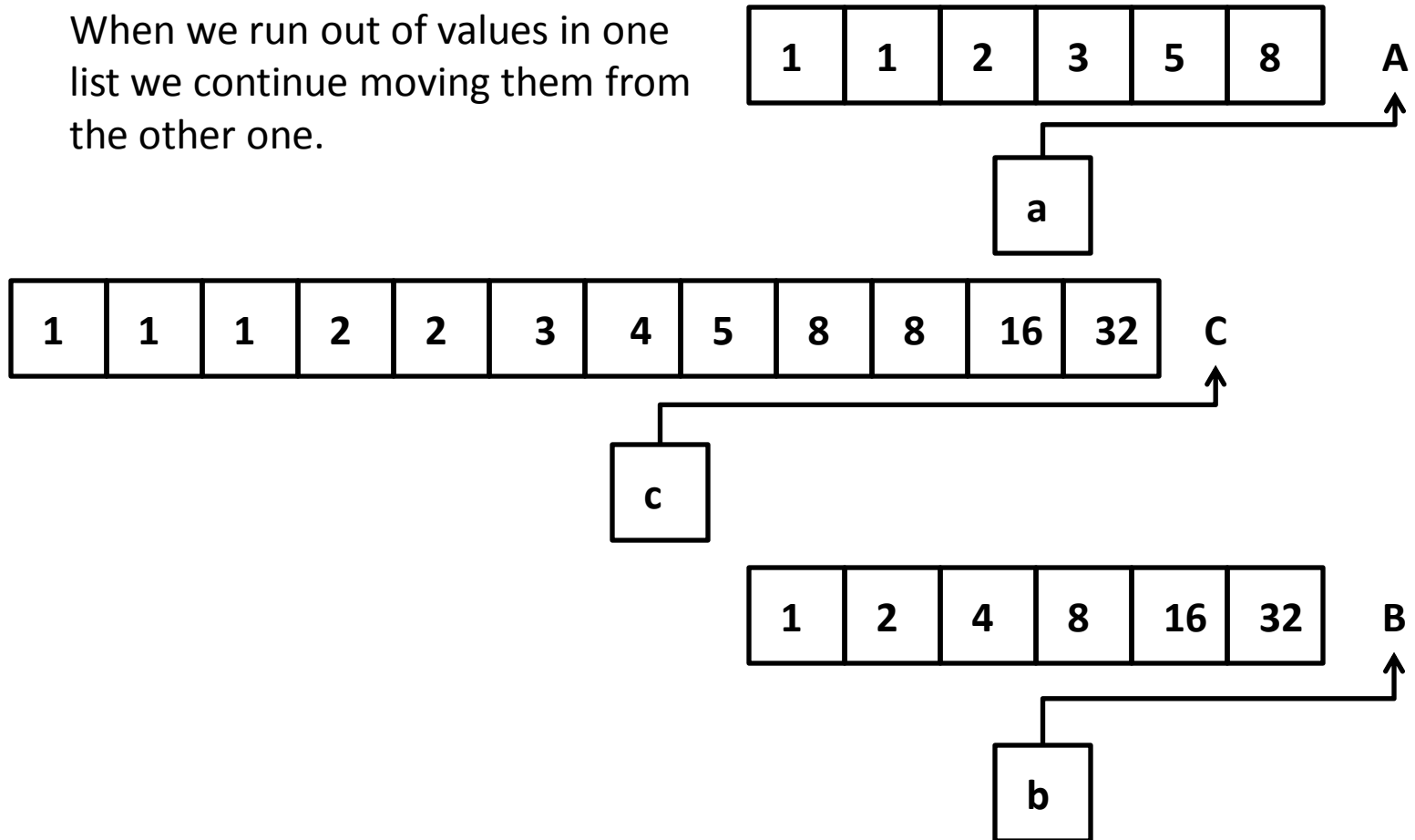
Example

When we run out of values in one list we continue moving them from the other one.



Example

When we run out of values in one list we continue moving them from the other one.



Runtime?

- What was the runtime?
- a gets incremented from 0 to n
- b gets incremented from 0 to m
- c gets incremented from 0 to n+m
- Plus a constant amount of work for each step (comparisons)
- Total runtime: $O(n + m)$

Mergesort

- Uses the merge procedure just described
- For array A , split it in half
 - $n = \text{len}(A)$
 - $A[:n/2]$, $A[n/2:]$
- Sort the left and right halves recursively
- Get back sorted arrays X and Y , both of length $n/2$ (approximately...)
- Use $\text{merge}(X, Y)$ to put them back together

Mergesort

- Base cases?
- When do we stop our recursion?
- List of length 0 and 1 are sorted by definition
- So we will use those as base cases

Runtime

- Let $T(n)$ be the runtime of our algorithm on a list of size n
- What does our algorithm do?
- Calls itself twice on lists of size $n/2$
- This contributes $2 * T(n/2)$ to the runtime
- Then merges those two lists; which takes roughly n steps

Runtime

- This gives $T(n) = 2T(n/2) + n$ (within a constant factor, which is all we care about)
- We assume $n = 2^k$ for some k (because it makes our analysis easier; don't have to deal with floors or ceilings for odd n)
- Claim: $T(2^k) \leq k 2^k$

Runtime

- Claim: $T(2^k) \leq k 2^k$
- Use induction
- Base case: $k = 1$ ($n = 2$)
- We'll say the runtime in this case is 2 for simplicity (since it is going to be a constant we are free to mess with)
- $T(2) = 2 \leq 1 * 2^1 = 2$
- Assume our claim holds for all $x < k$

Runtime

- Now need to show $T(2^k) \leq k 2^k$
- $T(2^k) = 2T(2^{k-1}) + 2^k$
- $T(2^k) \leq 2(k-1)2^{k-1} + 2^k$ (inductive hypothesis)
- $T(2^k) \leq (k-1)2^k + 2^k$
- $T(2^k) \leq k2^k$
- QED

Runtime

- This tells us the runtime is $O(n \log n)$
 - k was $\log n$
- We'll use this fact again when we look at another algorithm that has a similar runtime