Unit #4: Sorting

CPSC 221: Basic Algorithms and Data Structures

Anthony Estey, Ed Knorr, and Mehrdad Oveisi

2016W2

Unit Outline

- Comparing Sorting Algorithms
- Heapsort
- Mergesort
- Quicksort
- More Comparisons
- Complexity of Sorting

Learning Goals

- Describe, apply, and compare various sorting algorithms.
- Analyze the complexity of these sorting algorithms.
- Explain the difference between the complexity of a problem (sorting) and the complexity of a particular algorithm for solving that problem (e.g., Insertion Sort).

How to Measure Sorting Algorithms

- Computational complexity (a.k.a. runtime)
  - Worst case → worst possible ordering
  - Average case → expected
  - Best case

  How often is the input sorted, reverse sorted, or "almost" sorted (k swaps from sorted where $k \ll n$)?

What situation are we using our data for. How often are we adding new items / deleting / finding values.

- Stability: What happens to elements with identical keys? Why do we care? stable: preserve input (original) order when equal.
- Memory Usage: How much extra memory is used?

J. Smith, A. Smith, B. Smyth, T Smith

Sort by first name ⇒ A. Smith, B. Smyth, J. Smith, T. Smith

Sort by last name (and ties will ordered by ⇒ A. Smith, J. Smith, T. Smith, B. Smyth

Insertion Sort: Running Time

At the start of iteration i, the first i elements in the array are sorted, and we insert the $(i+1)$st element into its proper place.

$i=5$ items in index 0, 4

every time, we have shuffle all $i-1$ items over

Worst case:

n	n-1	n-2					3	2	1

run-time: $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} \in \Theta(n^2)$

$\Theta(n)$

Best case:

1	2	3						n-1	n

Expected/average → closer to worst (n^2) or the best (n)?

↳ on average, probably going to shuffle about half-way

$$\sum_{i=1}^{n-1} \frac{i}{2} \simeq \frac{n^2}{4} \in \Theta(n^2)$$

Insertion Sort: Stability & Memory

At the start of iteration i, the first i elements in the array are sorted, and we insert the $(i+1)$st element into its proper place.

Easily made stable:

The "proper place" is the **largest** j such that $A[j-1] \leq$ new element.

Memory:

Sorting is done **in-place**, meaning only a constant number of extra memory locations are used.

↳ don't need to create more memory/space as n grows

stable/stability: a sorting algorithm is said to be stable if two objects with equal keys appear in the same order in the sorted output array as they appeared in the input array that was originally to be sorted.

in-place: when an algorithm sorts input data without the need to use auxiliary data structures. It is okay to use a small amount of extra memory for variables.

Heapsort

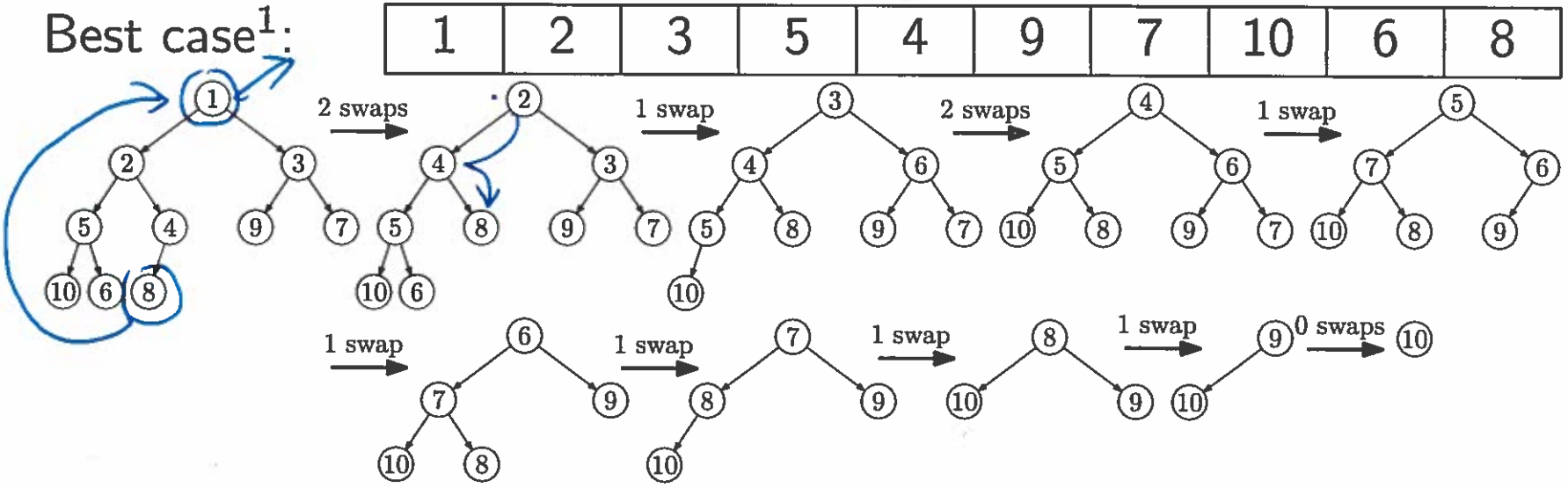
1. Run Heapify on the input array. — $\Theta(n)$
2. Repeat n times: Perform `deleteMin`.

Worst case: $\Theta(n) + \Theta(n \lg n) = \Theta(n \lg n)$

(1) (2.)

$\Theta(n \lg n)$

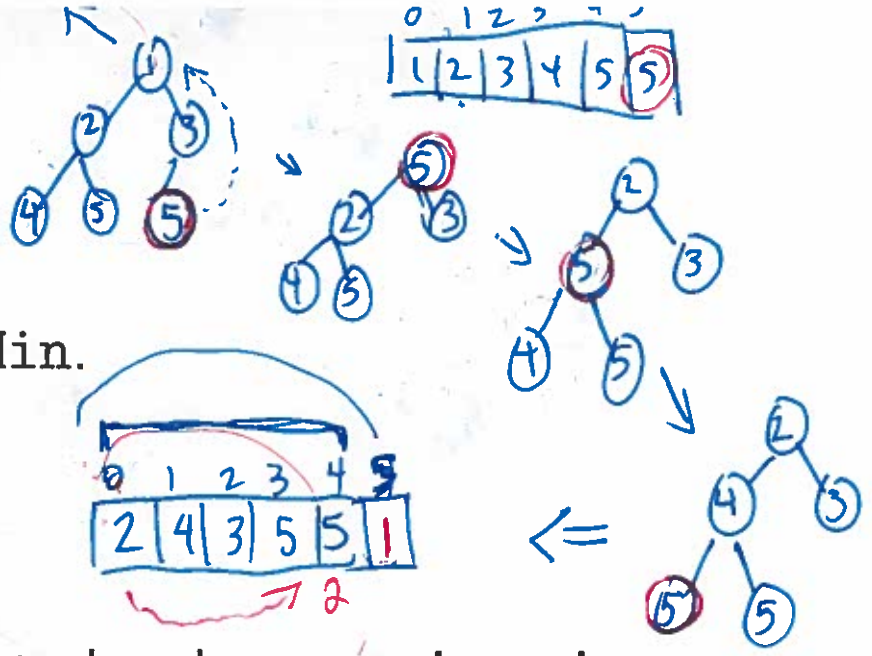
Best case[1]:



[1] Schaffer & Sedgewick, The Analysis of Heapsort, *J. Algorithms* **15** (1993), 76–100.

Heapsort: Stability & Memory

1. Run Heapify on the input array.
2. Repeat n times: Perform `deleteMin`.



Not stable:
Hack: Use the index in the input array to break comparison ties; but, this takes more space.
↳ add an extra extra field to represent the index in the original array

Memory:

- **in-place**. You can avoid using another array by storing the result of the ith `deleteMin` in heap location $n - i$, except the array is then sorted in reverse order, so use a Max-Heap (and `deleteMax`).
- Far-apart array accesses ruin cache performance.

Mergesort

Mergesort is a "divide and conquer" algorithm.

1. If the array has 0 or 1 elements, it's sorted. Stop. $T(1) = 1$
2. Split the array into two approximately equal-sized halves.
3. Sort each half recursively (using Mergesort). $T(n) = 2T(\frac{n}{2})$
4. Merge the sorted halves to produce one sorted result: $+ n$.
   - Consider the two halves to be queues. ⇒ Programming project 1.
   - Repeatedly dequeue the smaller of the two front elements (or dequeue the only front element if one queue is empty) and add it to the result.

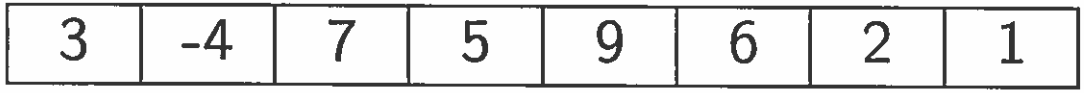
$$T(n) = \begin{cases} 1 & \text{if } n = 0 \text{ or } 1 \\ 2T(\frac{n}{2}) + n & \text{if } n \geq 1 \end{cases}$$

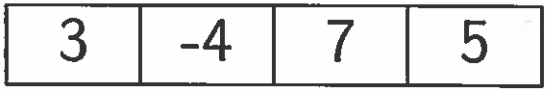
$$T(n) \in \Theta(n \lg n)$$

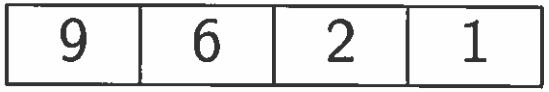
Mergesort Example

3	-4	7	5	9	6	2	1

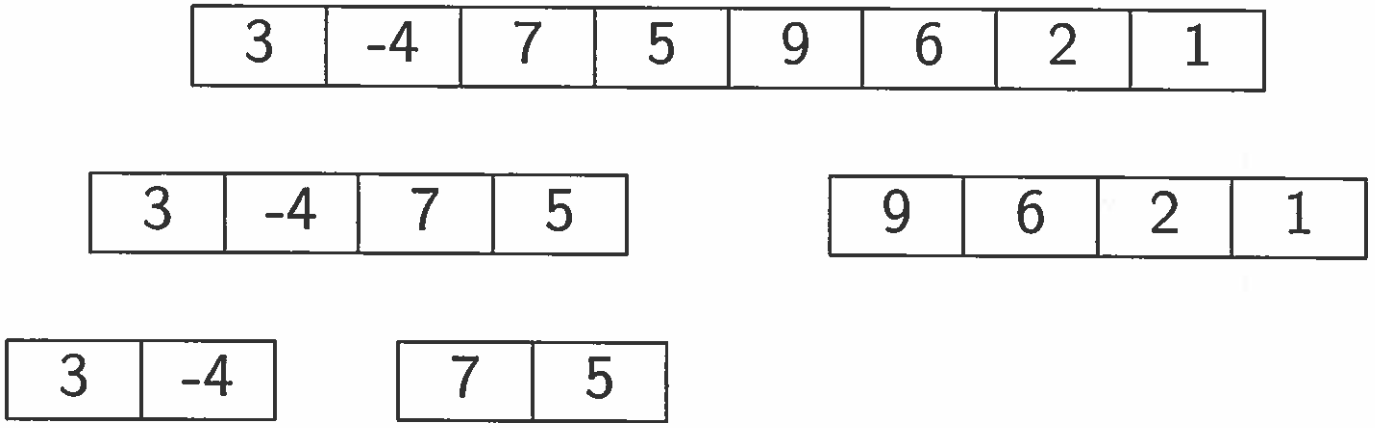
Mergesort Example



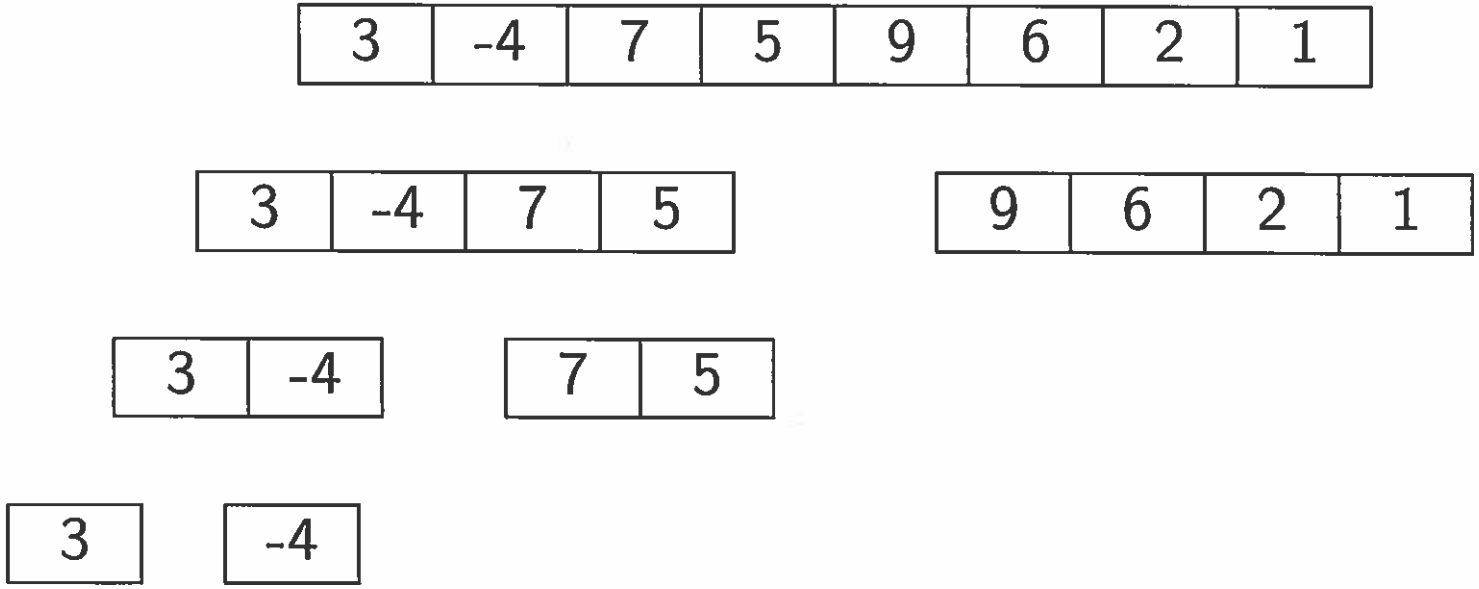




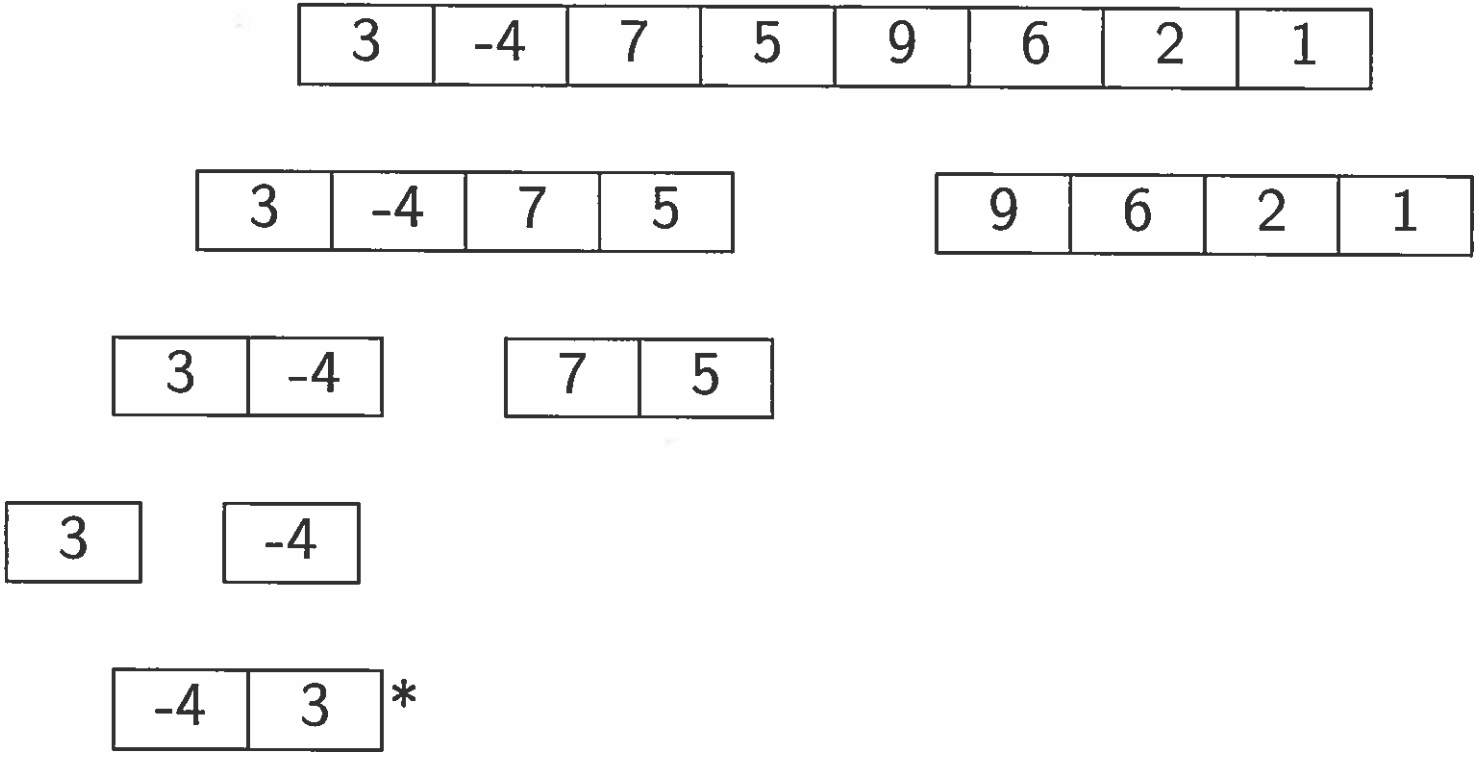
Mergesort Example



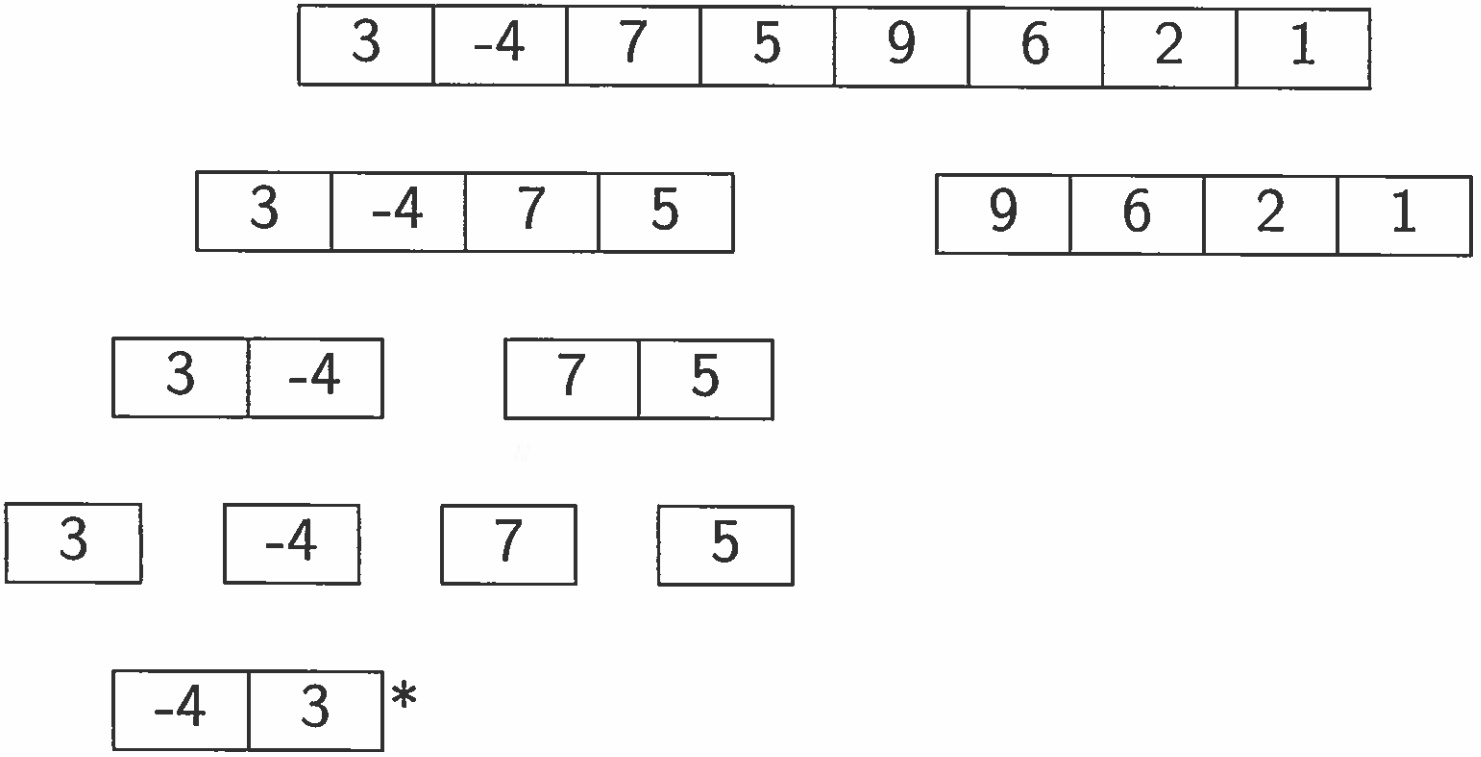
Mergesort Example



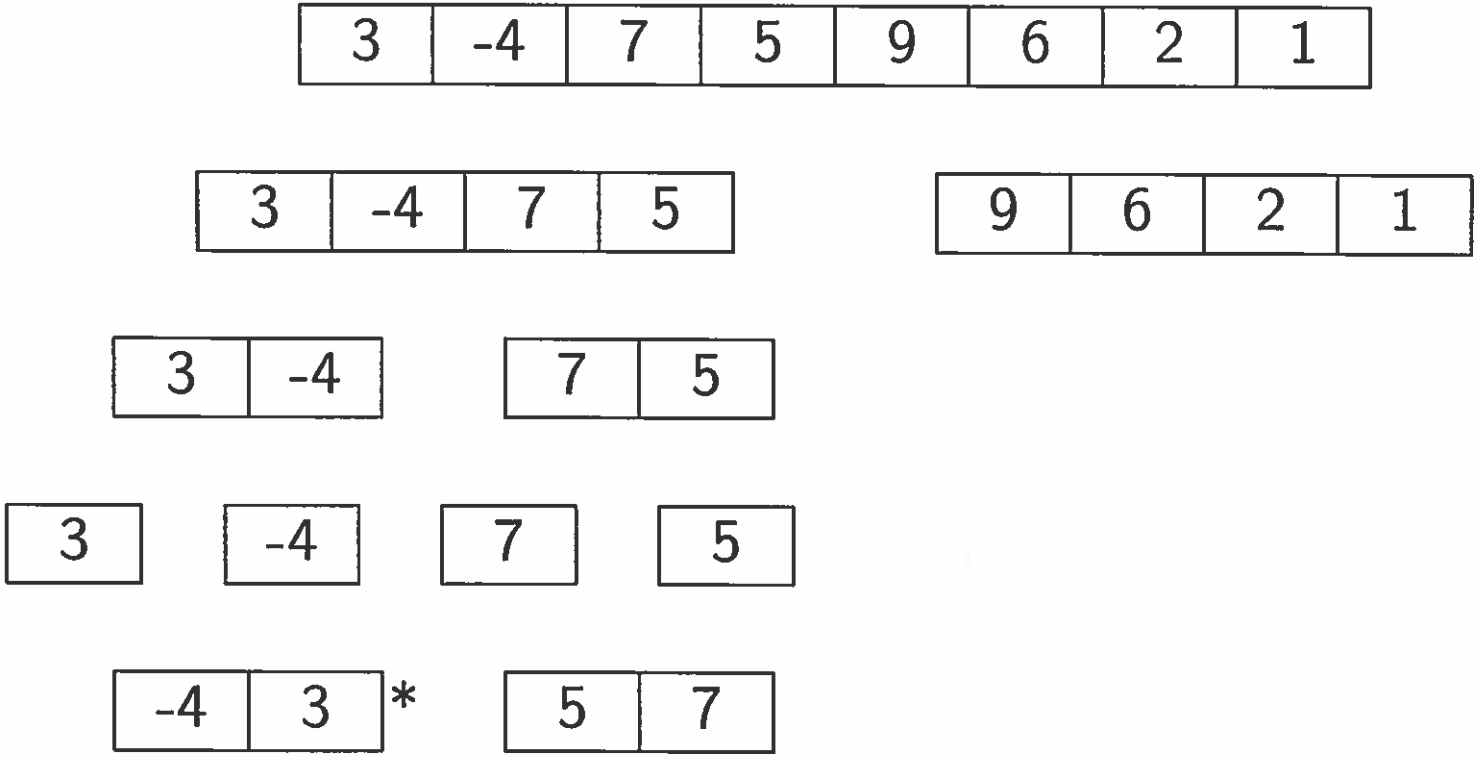
Mergesort Example



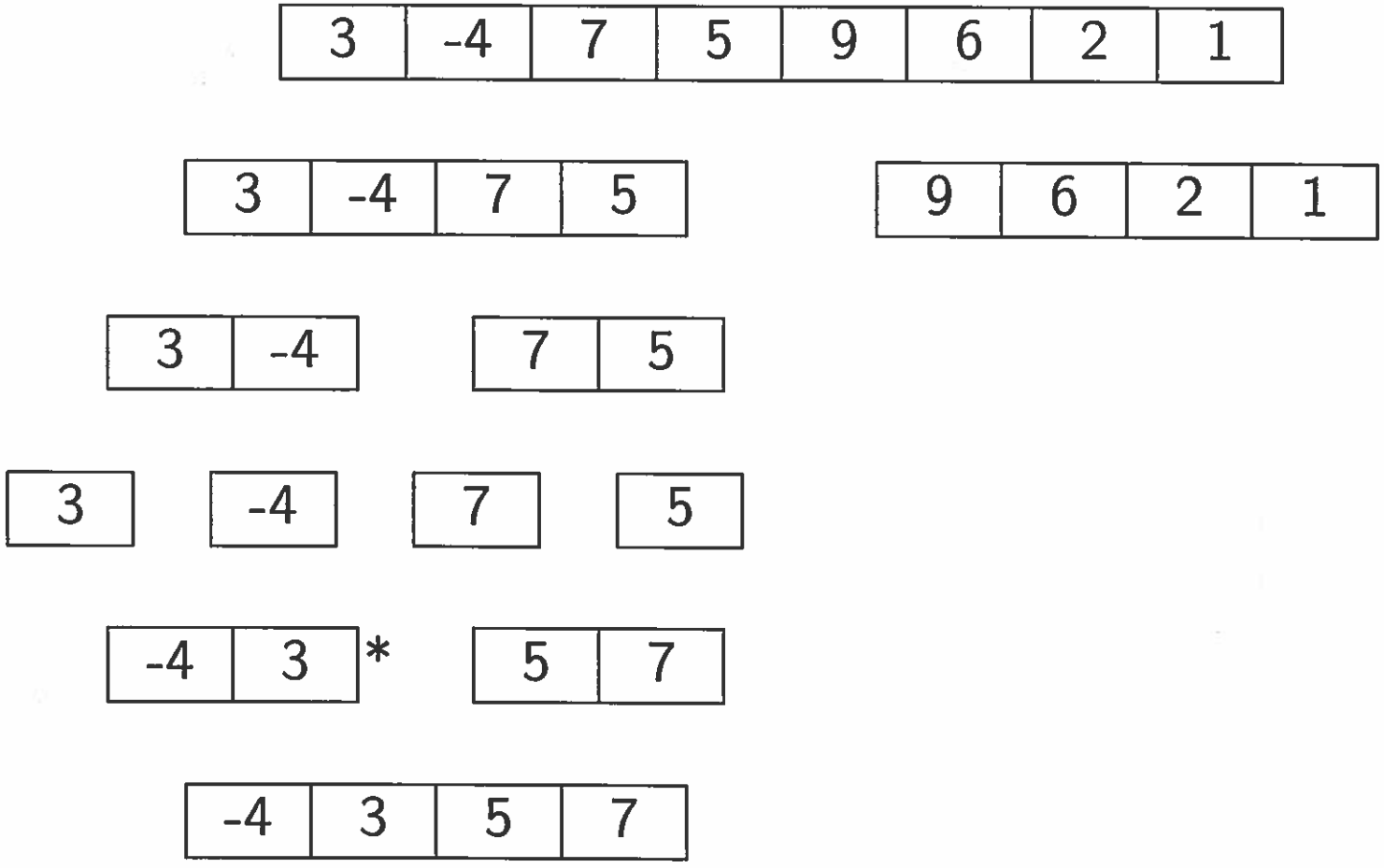
Mergesort Example



Mergesort Example



Mergesort Example



Mergesort Code

```
(2) void msort(int x[], int lo, int hi, int tmp[]) {
      if (lo >= hi) return;
      int mid = (lo+hi)/2;
      msort(x, lo, mid, tmp);
      msort(x, mid+1, hi, tmp);
      merge(x, lo, mid, hi, tmp);
    }

(1) void mergesort(int x[], int n) {
      int *tmp = new int[n];
      msort(x, 0, n-1, tmp);
      delete[] tmp;
    }
```

array to sort

# of elements in the array

temporary storage, to hold the sorted values during merge

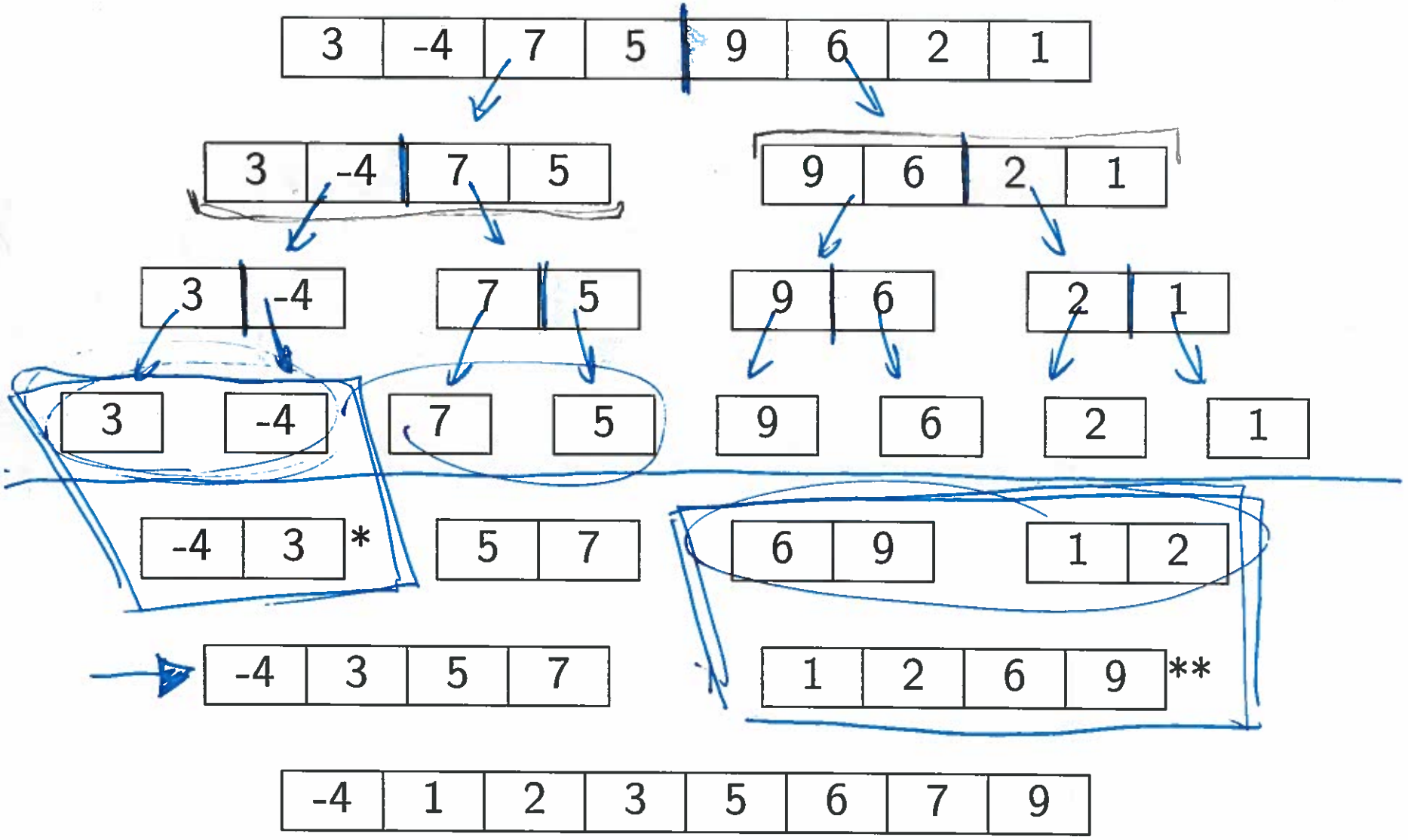
lower bound

upper bound.

Merge Code

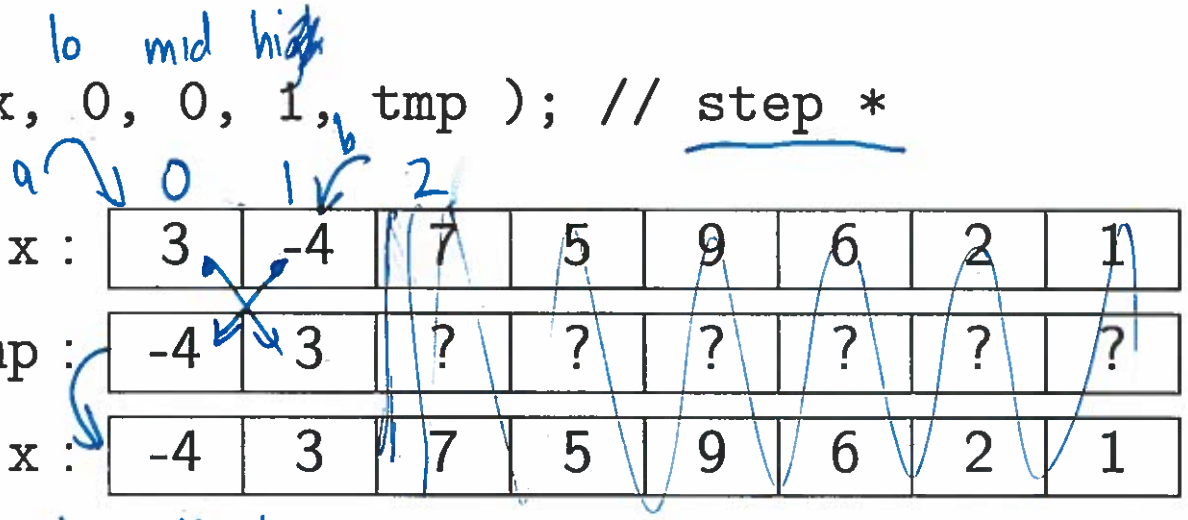
```
void merge(int x[],int lo,int mid,int hi,int tmp[]) {
  int a = lo, b = mid+1;
  for( int k = lo; k <= hi; k++ ) {
    // What's the loop invariant, at this point?
    if( a <= mid && (b > hi || x[a] < x[b]) )
      tmp[k] = x[a++];
    else tmp[k] = x[b++];
  }
  for( int k = lo; k <= hi; k++ )
    x[k] = tmp[k];
}
```

Mergesort Example

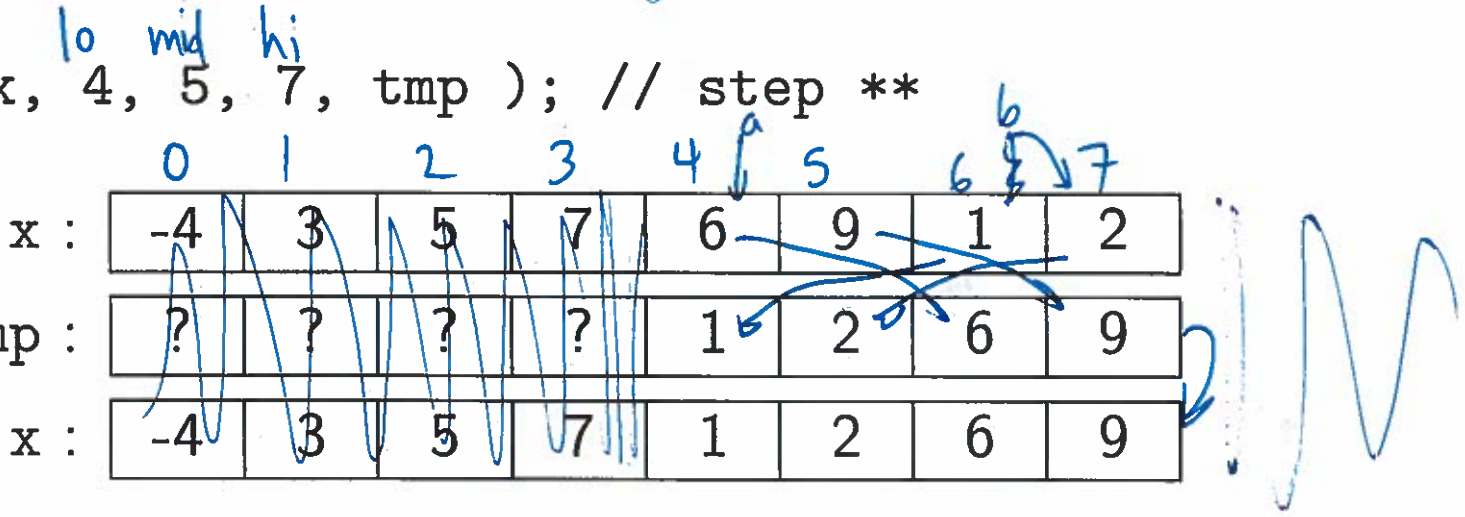


Sample Merge Steps

```
merge( x, 0, 0, 1, tmp ); // step *
```




```
merge( x, 4, 5, 7, tmp ); // step **
```




```
merge( x, 0, 3, 7, tmp ); // is the final step
```

Mergesort: Stability & Memory

Best case : $\Theta(n \lg n)$

case case : $\Theta(n \lg n)$

Stable:

Dequeue from the left queue if the two front elements are equal.

Memory:

This is not easy to implement without using $\Omega(n)$ extra space; so, it is not viewed as an in-place sort. Plus there's the cost of the call stack ($\Omega(\log n)$).

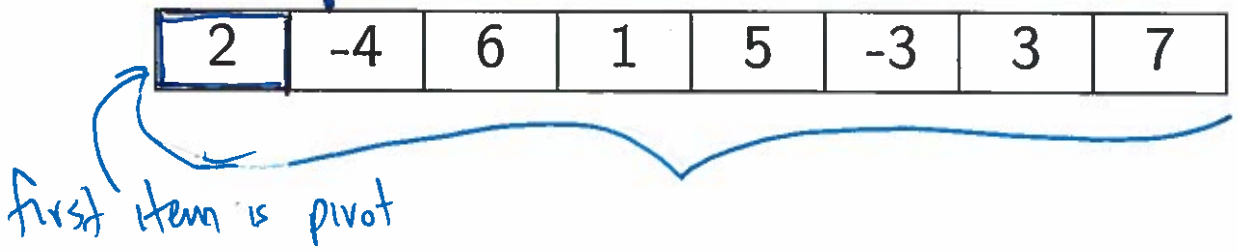
tmp array of size n

this algorithm, msort was not tail recursive

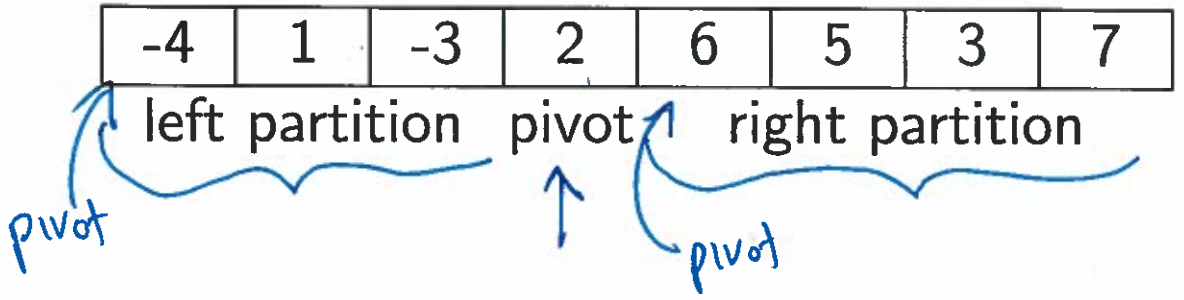
Quicksort (C.A.R. Hoare 1961)

In practice, this is one of the fastest sorting algorithms.

1. Pick a **pivot** → first item, it could be the last item or randomly selected



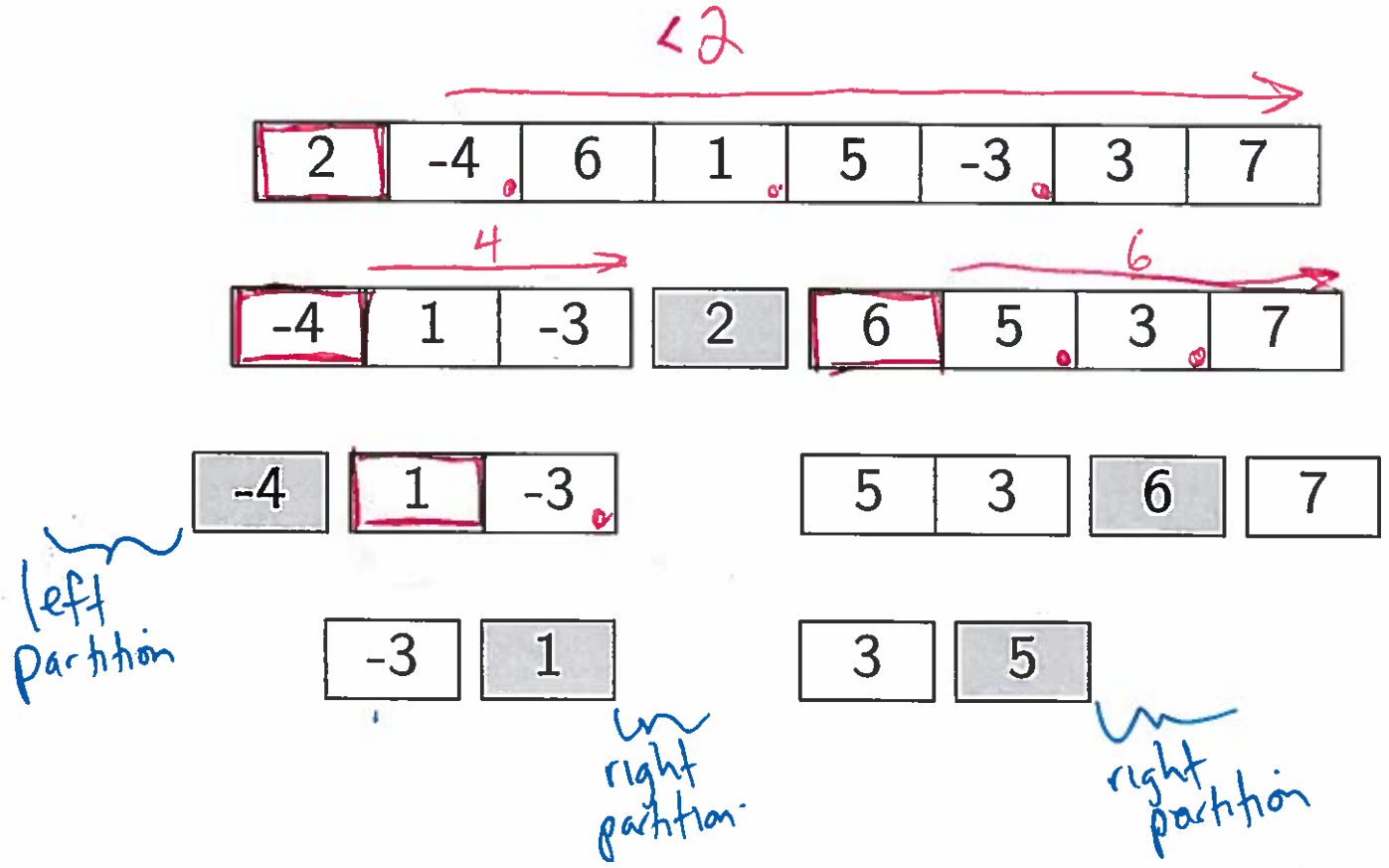
2. Reorder the array such that all elements $<$ pivot are to its left, and all elements $\geq$ pivot are to its right.



3. Recursively sort each partition.

What's the base case? partition size is 0 or 1.

Quicksort: Visually



Quicksort by Jon Bentley

(2)
```
void qsort(int x[], int lo, int hi) {
  int i, p;
  if (lo >= hi) return;
  p = lo;
  for( i=lo+1; i <= hi; i++ )
    if( x[i] < x[lo] ) swap(x[++p], x[i]);
  swap(x[lo], x[p]);
  qsort(x, lo, p-1);
  qsort(x, p+1, hi);
}
```

keep recursively with smaller partition sizes until base case is reached

move p up by one, and swap

at the end, swap pivot into the place of p

recursively solve left and right partitions

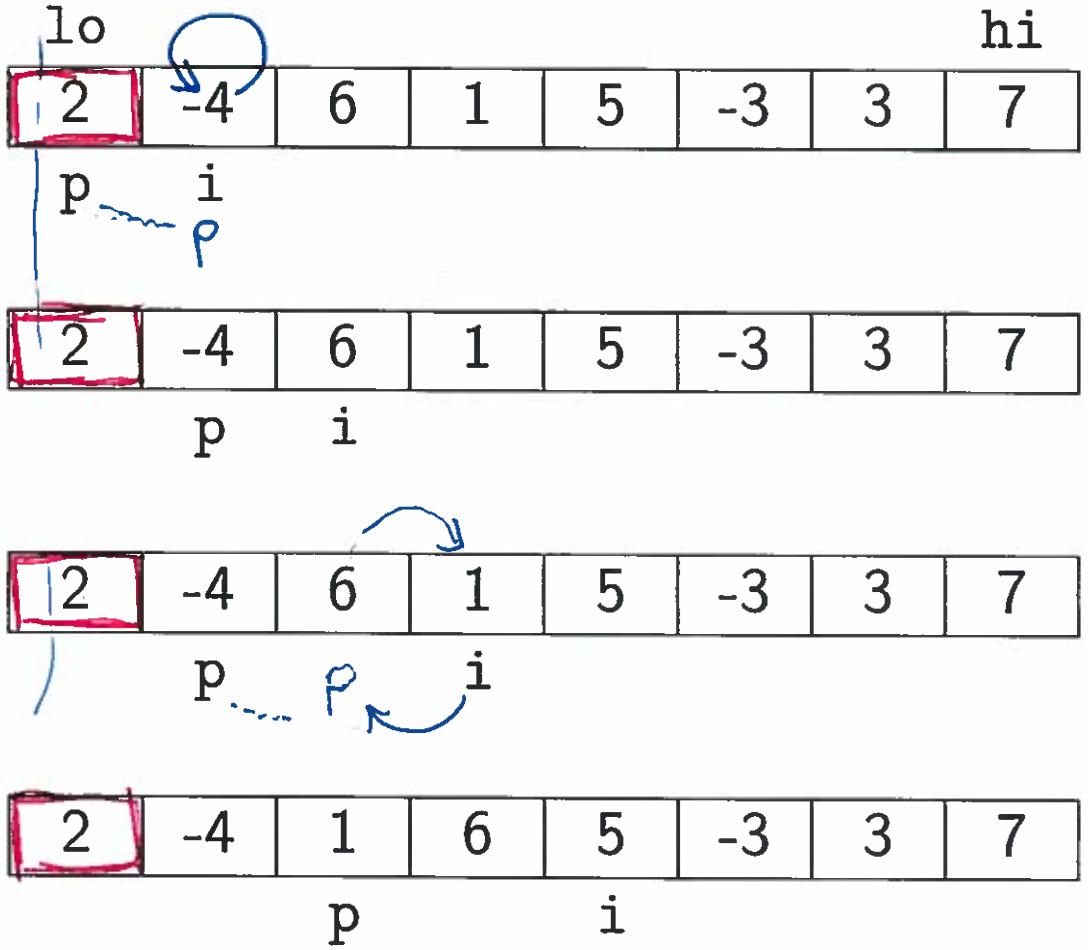
(1)
```
void quicksort(int x[], int n) {
  qsort(x, 0, n-1);
}
```

array to sort

range of valid indexes within array to sort.

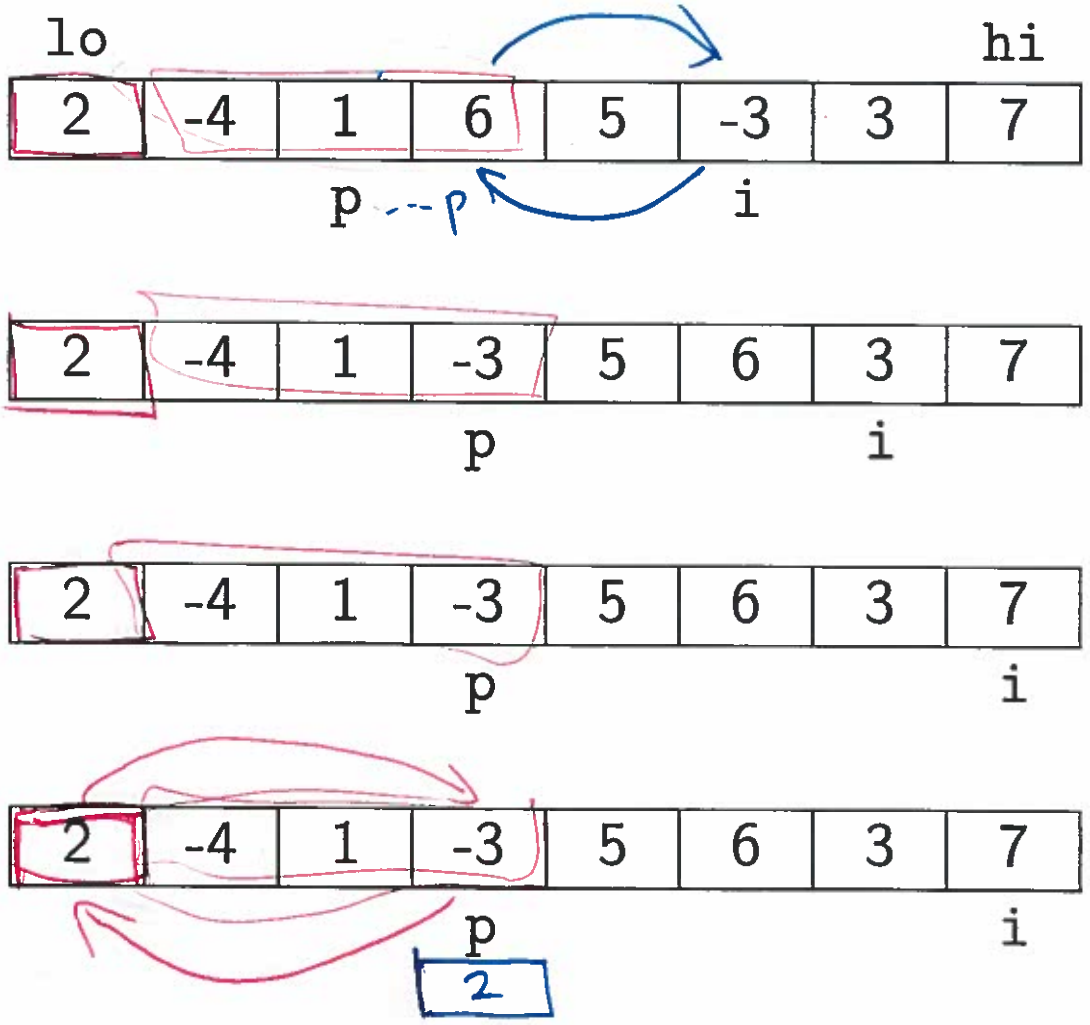
Quicksort Example (using Bentley's Algorithm)

```
if( x[i] < x[lo] ) swap(x[++p], x[i]);
```

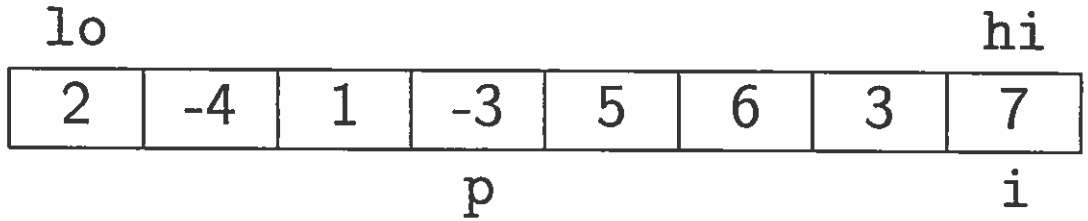


Quicksort Example (using Bentley's Algorithm)

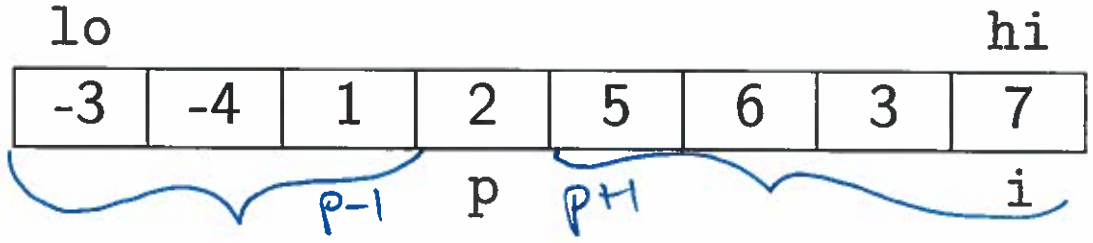
```
if( x[i] < x[lo] ) swap(x[++p], x[i]);
```



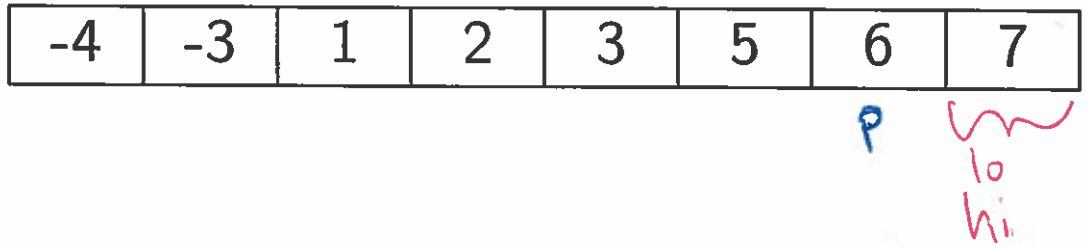
Quicksort Example (using Bentley's Algorithm)




```
swap(x[lo], x[p]);
```




```
qsort(x, lo, p-1);
qsort(x, p+1, hi);
```



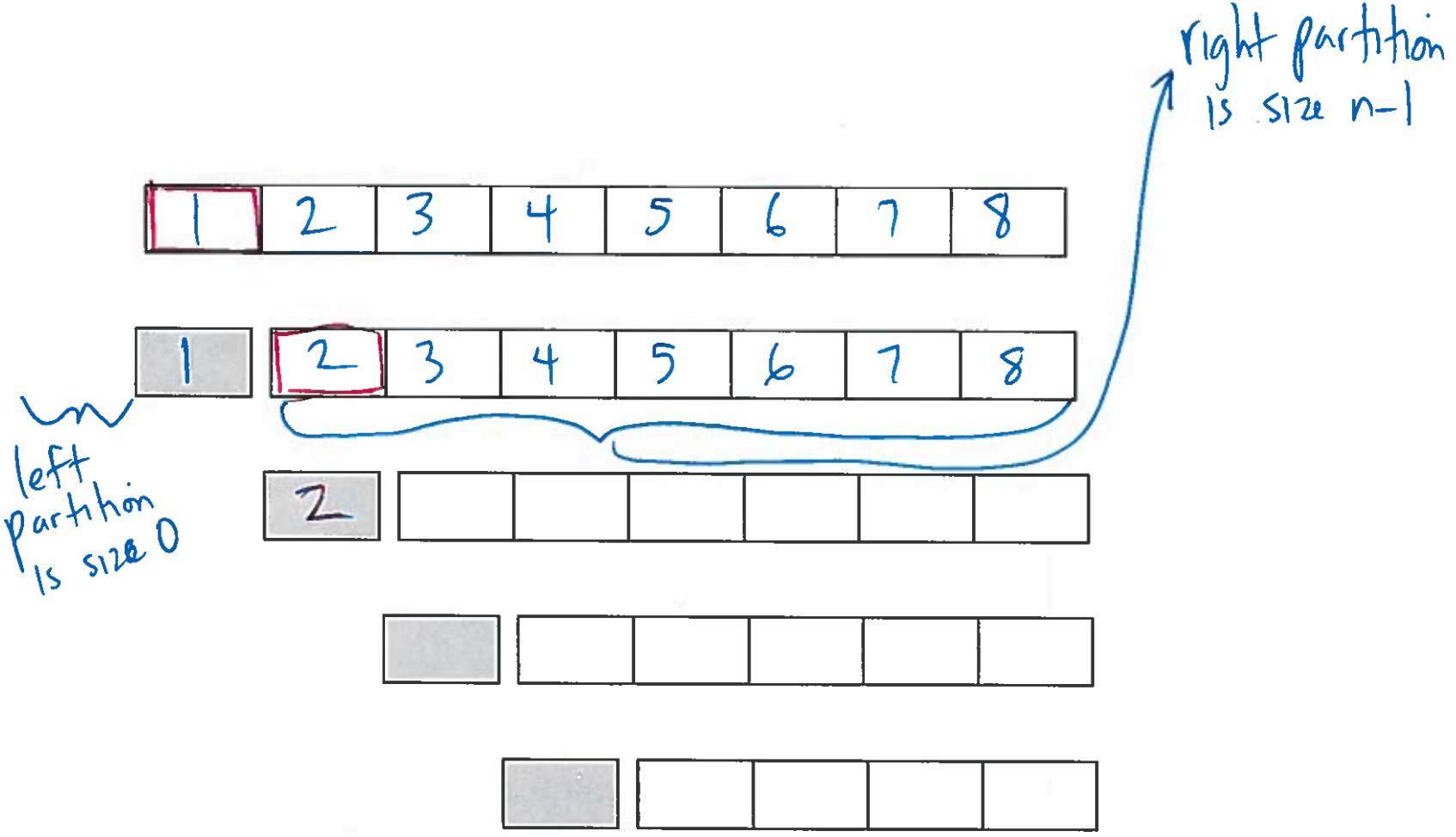
Quicksort: Running Time

The running time is proportional to number of comparisons; so, let's count comparisons.

1. Pick a pivot.
   Zero comparisons
2. Reorder (partition) the array around the pivot value.
   Quicksort compares each element to the pivot.
   $n - 1$ comparisons
3. Recursively sort each partition.
   The number of comparisons depends on the size of the partitions.

- If the partitions have size $n/2$ (or any constant fraction of n), the runtime is $\Theta(n \log n)$ (like Mergesort).
- In the worst case, however, we might create partitions with sizes 0 and $n - 1$. When might this occur? ↳ pivot is the highest or the lowest value ↘ $\Theta(n^2)$

Quicksort: Visually – the Worst Case



Quicksort: Worst Case

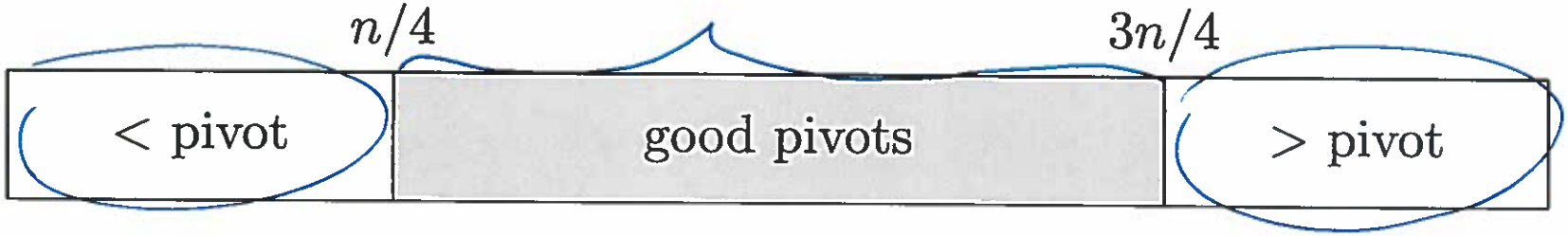
If this happens at every partition...
Quicksort makes $n-1$ comparisons in the first partition and recurses on a problem of size 0 and size $n-1$:

$$\begin{aligned} T(n) &= (n-1) + T(0) + T(n-1) = (n-1) + T(n-1) \\ &= (n-1) + (n-2) + T(n-2) \\ &\vdots \\ &= \sum_{i=1}^{n-1} i = (n-1)(n-2)/2 \end{aligned}$$

This is $\Theta(n^2)$ comparisons.

Quicksort: Average Case (Intuition)

- On an average input (i.e., random order of n items), our chosen pivot is equally likely to be the ith smallest for any $i = 1, 2, \ldots, n$.
- With probability $1/2$, our pivot will be from the middle $n/2$ elements – a good pivot.



- Any good pivot creates two partitions of size at most $3n/4$.
- We expect to pick one good pivot every two tries.
- Expected number of splits is at most $2 \log_{4/3} n \in O(\log n)$.
- $O(n \log n)$ total comparisons. True, but this intuition is not a proof.

Quicksort: Stability & Memory

Stable:

Quicksort can be made stable – most easily by using more memory.

↳ Depends on how the pivot is chosen, and how the swaps are done

Memory:

In-place sort

↳ swaps within the input array.
No additional memory is needed.

Comparison of Running Times for 100 Samples

n	Insertion		Heap		Merge		Quick	
	avg	max	avg	max	avg	max	avg	max
100,000	11.20s	16.37s	0.04s	0.08s	0.03s	0.04s	0.02s	0.04s
200,000	36.97s	60.01s	0.08s	0.16s	0.06s	0.11s	0.06s	0.16s
400,000	172.36s	505.38s	0.56s	1.74s	0.54s	0.91s	0.46s	0.69s
800000			0.37s	0.83s	0.21s	0.35s	0.19s	0.32s
1600000			0.93s	1.77s	0.52s	1.12s	0.44s	0.78s
3200000			2.07s	3.04s	1.01s	1.95s	0.91s	1.44s
6400000			4.76s	7.54s	2.18s	3.88s	1.97s	3.45s
12800000			10.65s	12.38s	4.56s	7.01s	4.13s	5.94s

The code is from the lecture notes and labs, but it is not optimized.

A Comparison of Quicksort, Mergesort, Heapsort, and Insertion Sort

Running Time:

	$\Theta(n)$	$\Theta(n \log n)$	$\Theta(n^2)$
Best case:	Insert	Quick, Merge, Heap	
Average case:		Quick, Merge, Heap	Insert
Worst case:		Merge, Heap	Quick, Insert
"Real" data:		Quick $<$ Merge $<$ Heap $<$ Insert	

Some Quicksort/Mergesort implementations use Insertion Sort on small arrays (base cases).

Some results depend on the implementation. For example, an initial check whether the last element of the left subarray is less than the first of the right can make Mergesort's best case linear.

A Comparions of Quicksort, Mergesort, Heapsort, and Insertion Sort (cont.)

Stability:

Stable (easy):	Insert, Merge (we prefer the left of the two sorted subarrays when encountering ties)
Stable (with effort):	Quick
Unstable:	Heap

Memory Use:

- Insert, Heap, Quick $<$ Merge

Complexity of the Sorting Problem

The **complexity** of a problem is the complexity of the best algorithm for that problem.

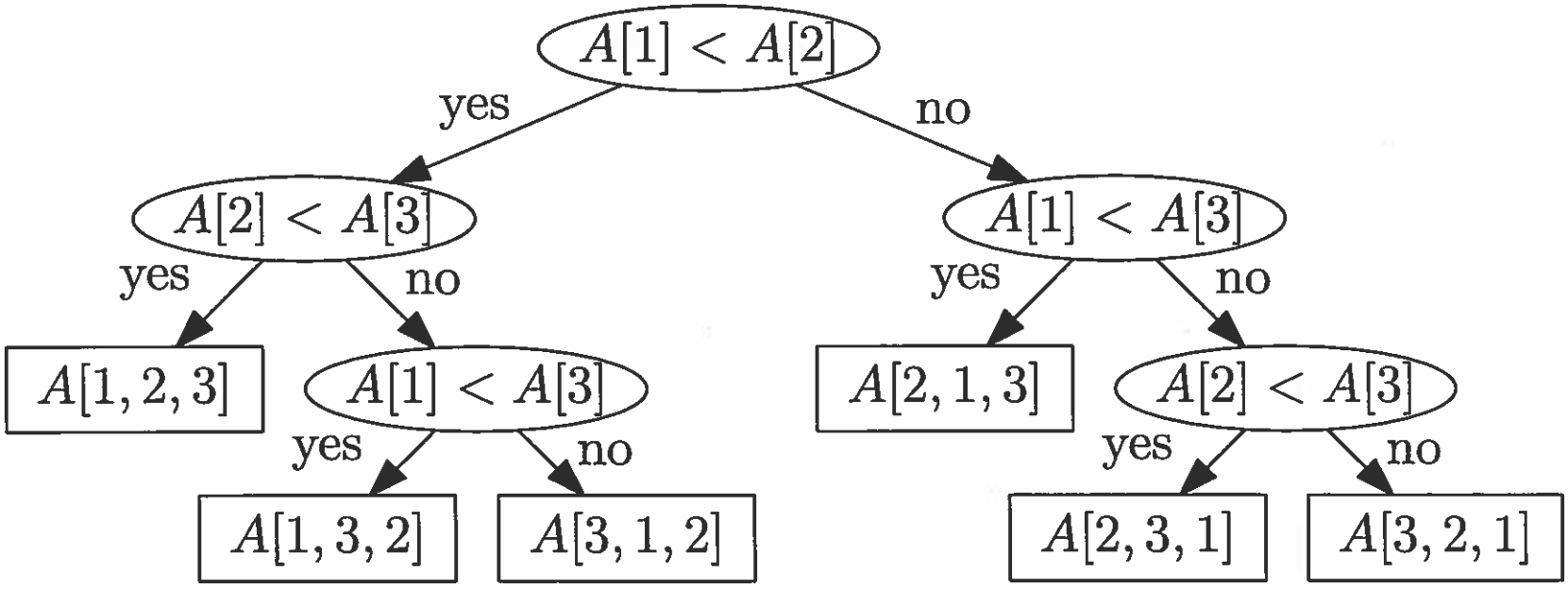
How powerful is our computer?

We'll only consider **comparison-based** algorithms.
They can compare two array elements in constant time.
They cannot manipulate array elements in any other way.
For example, they cannot assume that the elements are numbers and perform arithmetic operations (like division) on them.

Insertion Sort, Heapsort, Mergesort, and Quicksort are comparison-based.
Radix sort is not.

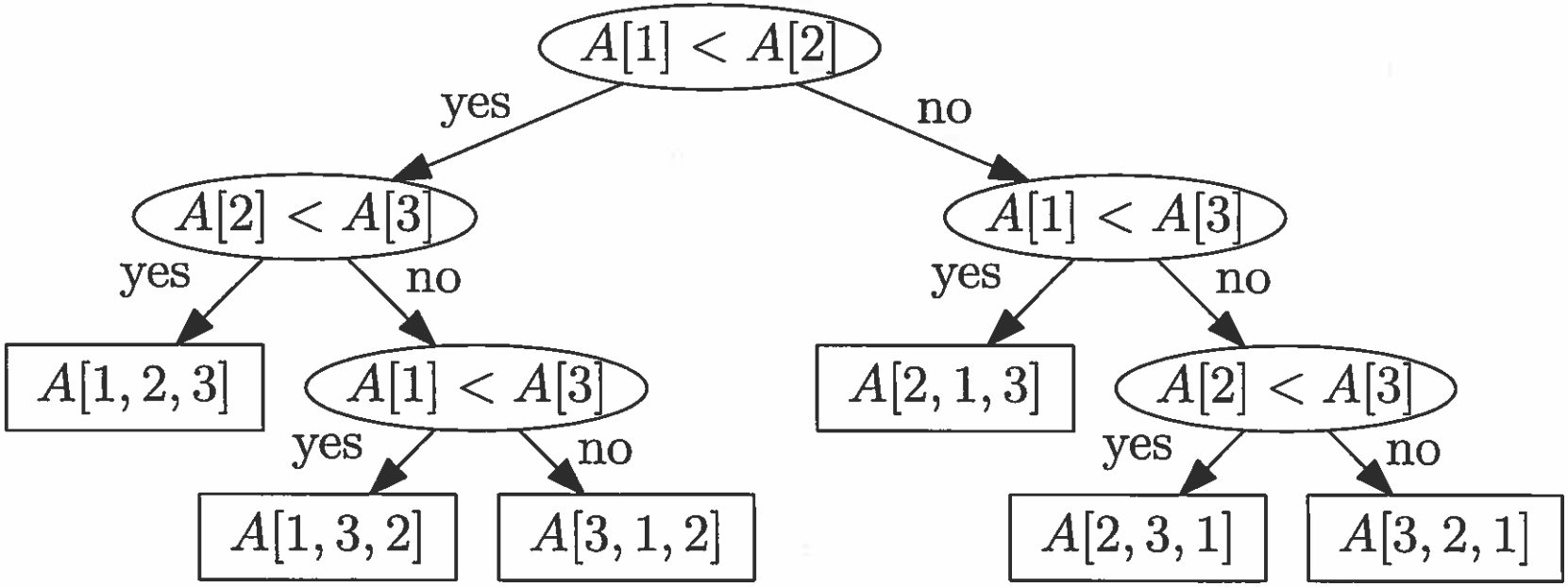
Comparison-Based Algorithms Using a Decision Tree Model

Each comparison is a "choice point" in the algorithm: the algorithm can do one thing if the comparison is true and another if false. So, the algorithm is like a binary tree...



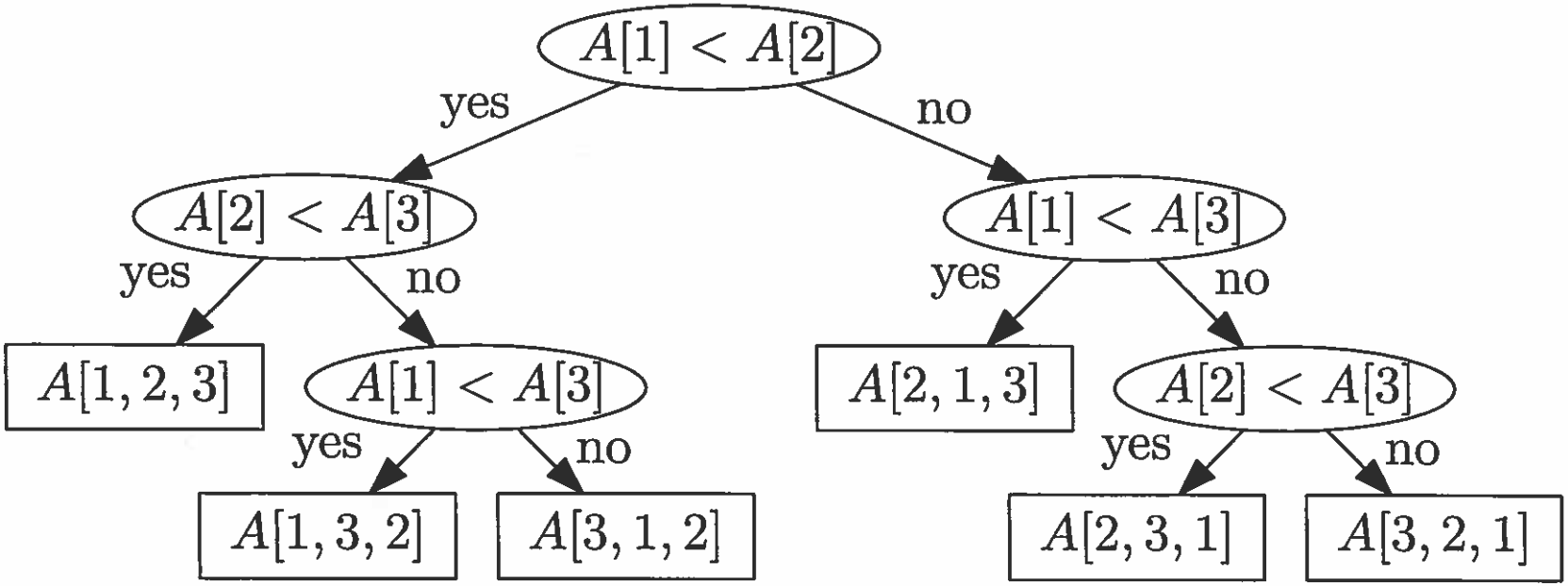
Complexity of the Sorting Problem

- This is the decision tree representation of Insertion Sort on inputs of size $n = 3$.
- Each leaf outputs the input array in some particular order. For example, $A[3, 1, 2]$ means output $A[3]$, $A[1]$, $A[2]$.



Complexity of the Sorting Problem

- There are $n!$ possible output orderings of an input array of size n.
- There must be a leaf for each one; otherwise, the algorithm fails to sort.
  - For example, if leaf $A[2, 3, 1]$ doesn't exist then the algorithm cannot sort [cat, ant, bee].



Complexity of the Sorting Problem

- The number of leaves is at least $n!$.
- The height of the decision tree is at least $\lceil \lg(n!) \rceil$.
- The number of comparisons made *in the worst case* is at least $\lceil \lg(n!) \rceil$.
- This is true for **any comparison-based sorting algorithm**; therefore, the complexity of the sorting problem is $\Omega(n \log n)$.