Unit #4: Sorting

CPSC 221: Basic Algorithms and Data Structures

Anthony Estey, Ed Knorr, and Mehrdad Oveisi

2016W2

Annotated Slides from Ed's Class

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

operations

time → CPSC 221

- Computational complexity (a.k.a. runtime)
  - Worst case
  - Average case
  - Best case

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

# of disk I/O's
CPSC 404

e.g., sort by first within last name

- Stability: What happens to elements with identical keys? Why do we care?

ties

e.g., Sedin, Daniel
Sedin, Henrik

keep D → H order for ties

- Memory Usage: How much extra memory is used?

221 = memory-resident sorts
404 = disk-resident sorts

Insertion Sort: Running Time

$n = 100$

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.

Worst case: backwards input

[0]	[1]	[2]				...			[n-1]
n	$n-1$	$n-2$							1
100	99	98							

1, 2, 3 comparisons

$$T(n) = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} \Rightarrow \Theta(n^2)$$

Best case:

1	2	3						...	$n-1$

in each iteration: do 1 comparison but no swap $\Rightarrow \Theta(n)$

1) warning: not $\Omega(n)$ for worst-case

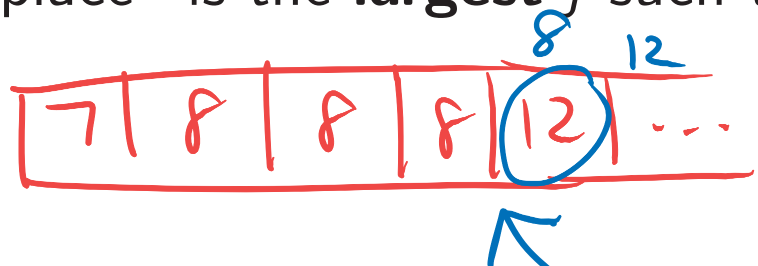
2) average case: $T(n) = \sum_{i=1}^{n-1} \frac{1}{2} i = \frac{1}{2} \sum_{i=1}^{n-1} i \Rightarrow \Theta(n^2)$

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.



insert another 8

Memory:

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

Heapsort

Robert Floyd (1964) — efficient alg. to build a heap $\Theta(n)$ time (steps)

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

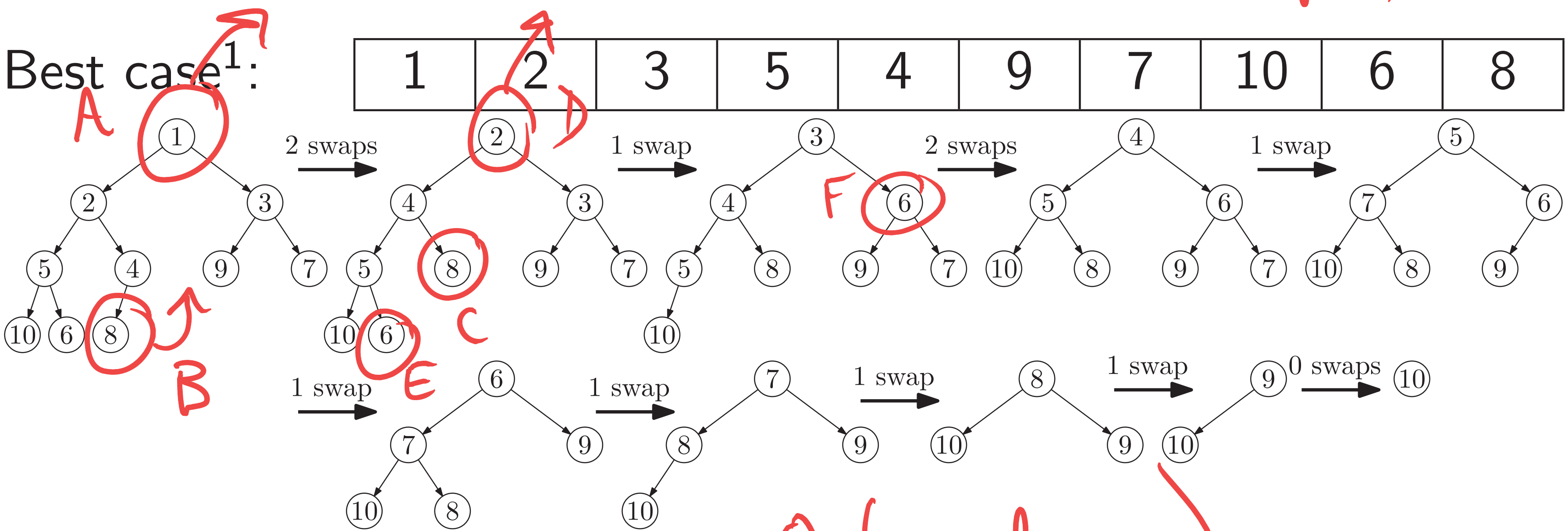
$n * \lg n = \sum_{i=1}^{n} \lfloor \lg i \rfloor$

Worst case:

$(1) \,\&\, (2) \Rightarrow \Theta(n) + \Theta(n \cdot \lg n) = \Theta(n \lg n)$

Best case[1]:

1	2	3	5	4	9	7	10	6	8



$\Theta(n \lg n)$

[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:

if you encounter ties while sorting, you can break the ties by using the smaller index of the array

Hack: Use the index in the input array to break comparison ties; but, this takes more space.

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).
4. Merge the sorted halves to produce one sorted result:
    - Consider the two halves to be queues.
    - 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.

$$T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + \underbrace{n}_{\text{merge}}$$

$$\approx 2T\left(\frac{n}{2}\right) + n$$

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

$$\left\{ \begin{array}{ll} T(n) = 1 & \text{if } n = 1 \text{ or } 0 \\ = 2T\left(\frac{n}{2}\right) + n & \text{if } n > 1 \end{array} \right.$$

Mergesort Example

unsorted

3	-4	7	5	9	6	2	1

Mergesort Example

3	-4	7	5	9	6	2	1

3	-4	7	5

9	6	2	1

Mergesort Example

3	-4	7	5	9	6	2	1

3	-4	7	5

9	6	2	1

3	-4

7	5

Mergesort Example

3	-4	7	5	9	6	2	1

3	-4	7	5

9	6	2	1

3	-4

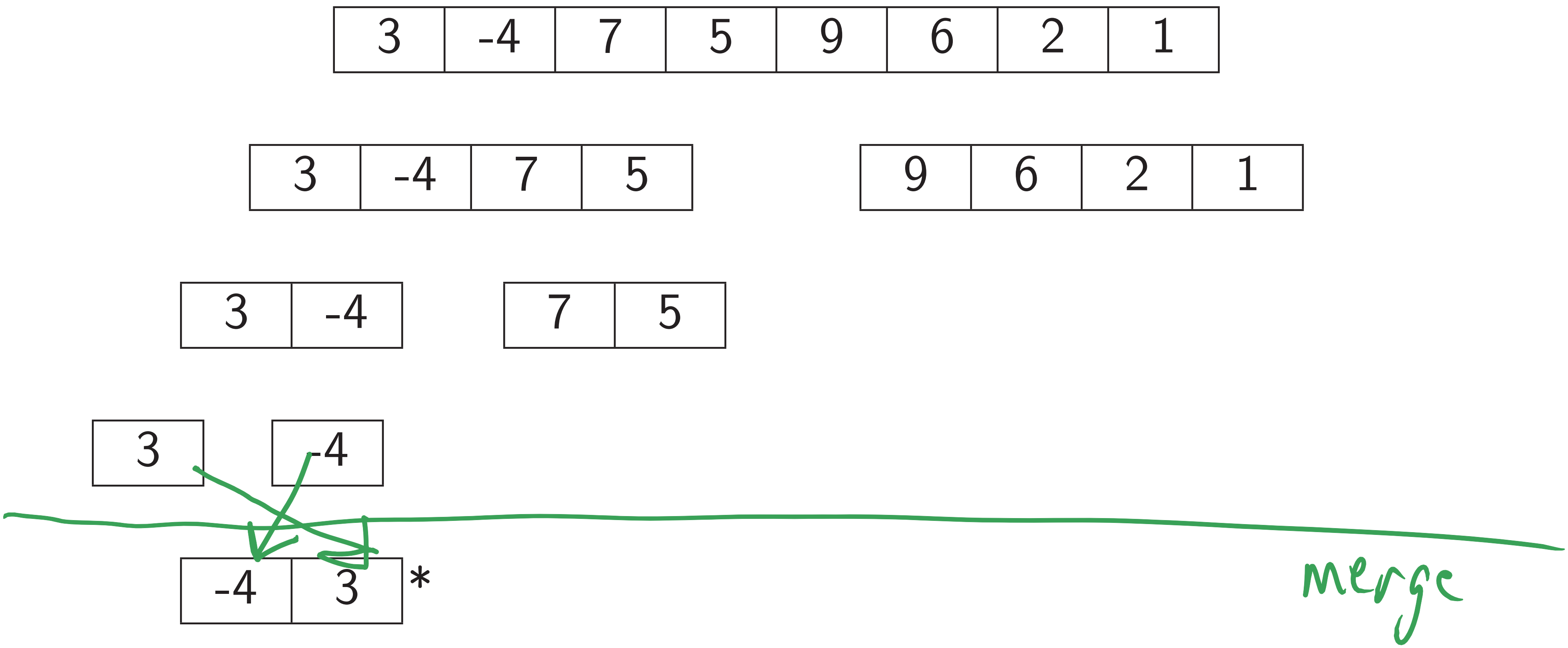
7	5

3

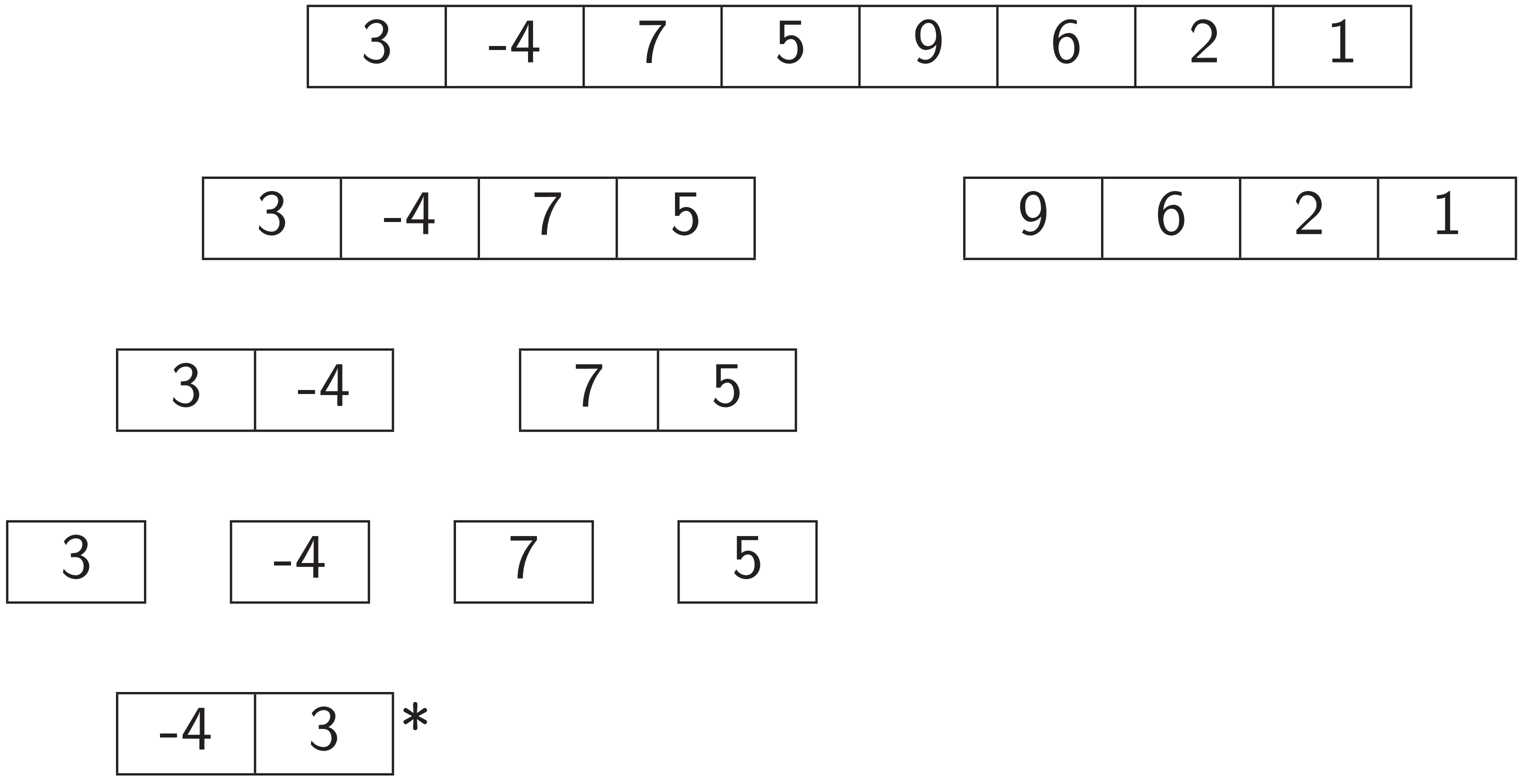
-4

each sublist is one element = trivially sorted (size 1)

Mergesort Example



Mergesort Example



Mergesort Example

3	-4	7	5	9	6	2	1

3	-4	7	5

9	6	2	1

3	-4

7	5

3

-4

7

5

-4	3

*

5	7

Mergesort Example

3	-4	7	5	9	6	2	1

3	-4	7	5

9	6	2	1

[3, -4] [7, 5]

[3] [-4] [7] [5]

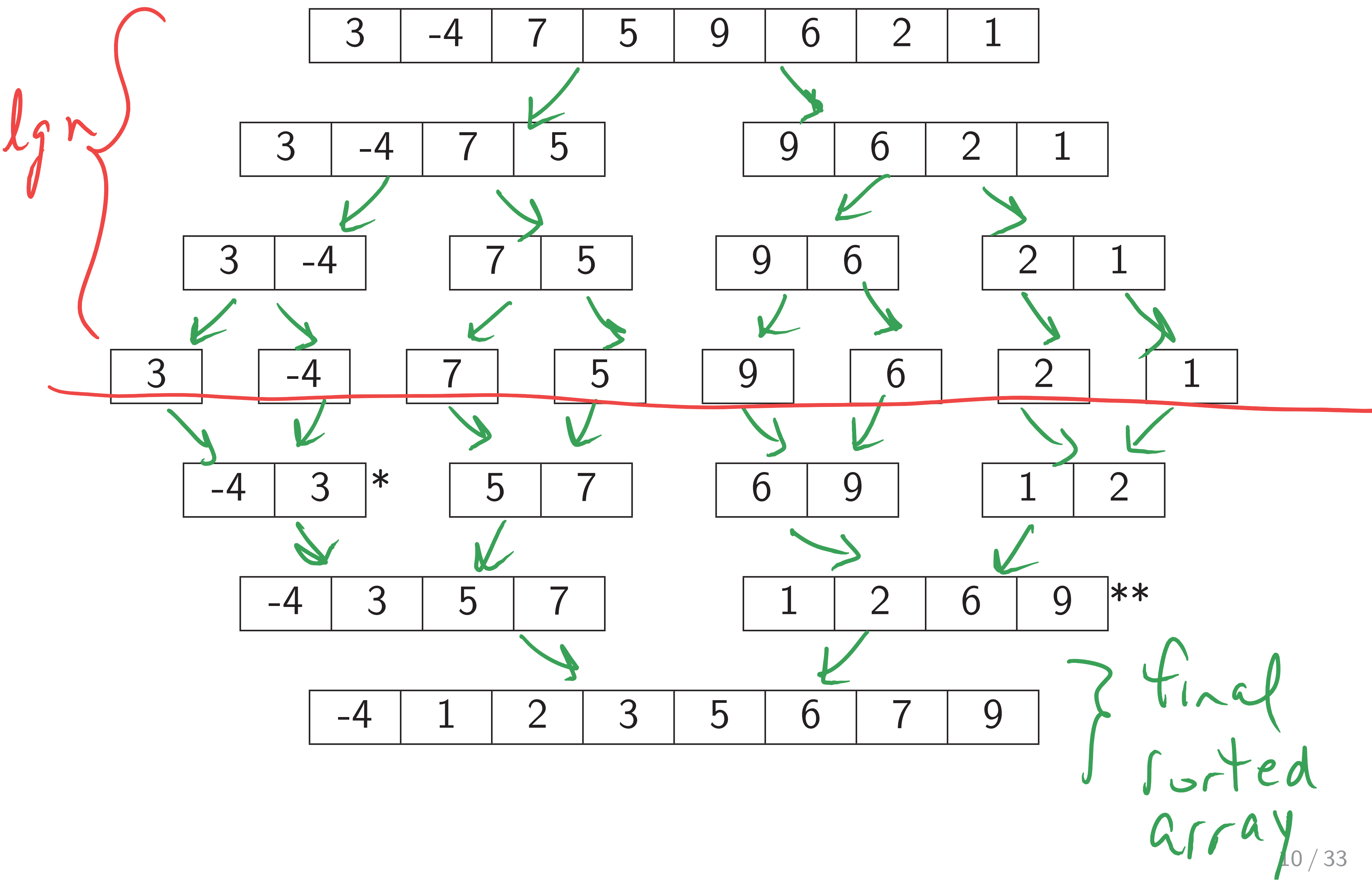
[-4, 3]* [5, 7]

-4	3	5	7

subarray of size 4 is sorted } 1/2

do the same for the right subarray of size 4

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);
}
```

find midpoint; split there

array to be sorted

int x[] is the same as int * x

1

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

work space — allocated once for all of mergesort and then passed by reference

deletes work array

call by reference

lower index

upper index

purpose: storage used for copying during merge

Merge Code

sublists are within array "x"

```
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];
}
```

for stability

tmp[lo..k-1] is sorted

& we're now starting on iteration k

Sample Merge Steps

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

work space

perm-anent array

x :	3	-4	7	5	9	6	2	1
tmp :	-4	3	?	?	?	?	?	?
x :	-4	3	7	5	9	6	2	1

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

5+1 7

x :	-4	3	5	7	6	9	1	2
tmp :	?	?	?	?	1	2	6	9
x :	-4	3	5	7	1	2	6	9
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

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

Mergesort: Stability & Memory

Stable:

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

- make sense because that's the order of input (original array)

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 ($\Omega(\log n)$).

so if 1 GB of data is being sorted...

Quicksort (C.A.R. Hoare 1961)

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

1. Pick a **pivot**

2	-4	6	1	5	-3	3	7

e.g., first element, last element, random element, etc.

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

-4	1	-3	2	6	5	3	7

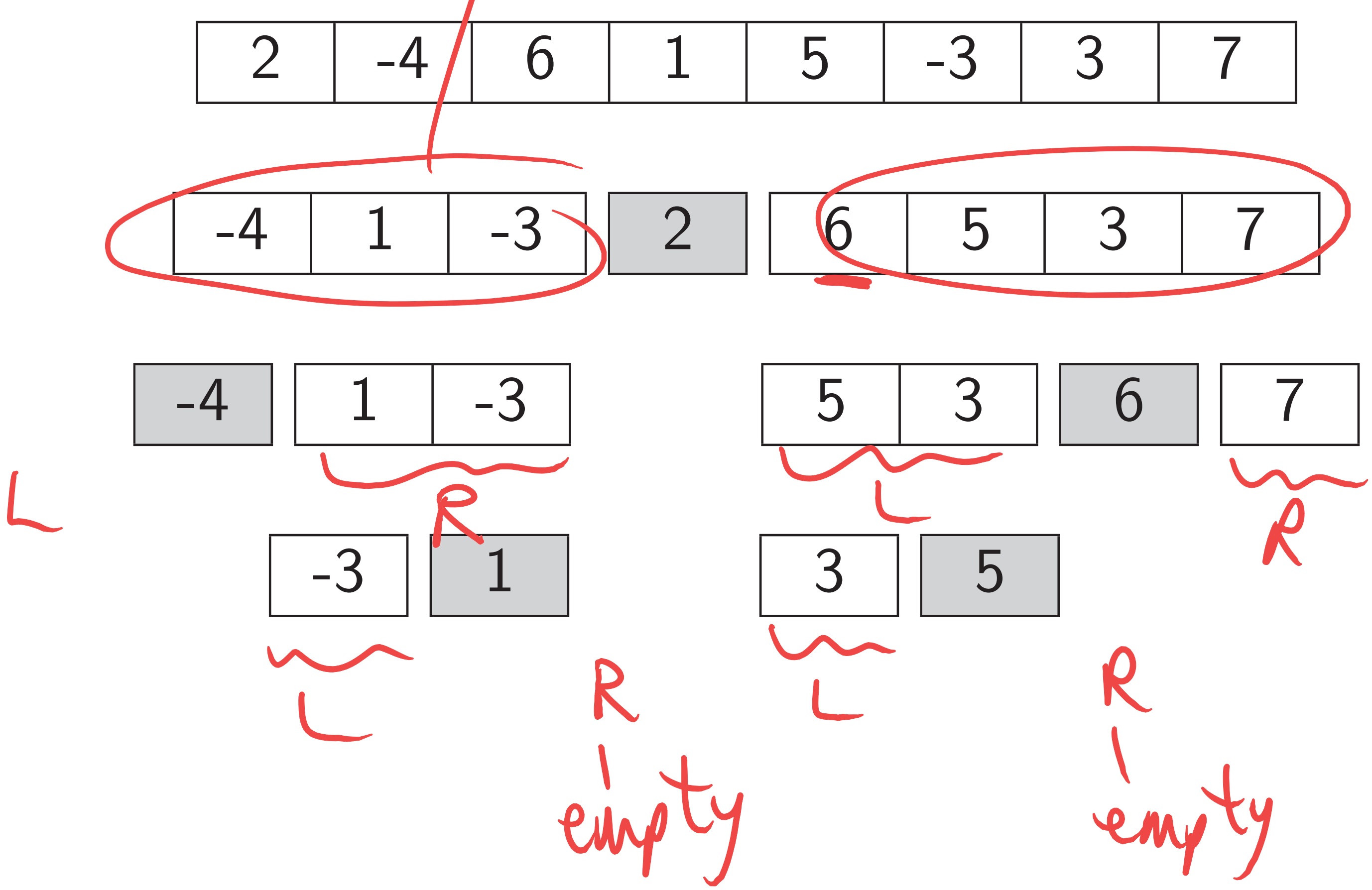
left partition — subproblem; pivot — final resting place for the pivot; right partition — subproblem

3. Recursively sort each partition.

What's the base case? $n = 1$ (or 0)

Quicksort: Visually

can be in any order
but < pivot's value



Quicksort by Jon Bentley

trace through this with some data

helper function

3

```
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);
}
```

means increment p by 1 first before executing the rest

1 initial call : quicksort(myArray, size)

2

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

Quicksort Example (using Bentley's Algorithm)

lo	hi	p	i
0	7	0	1
		1	2
		2	3
			4

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

i=1

[0]	[1]	[2]					[7]
lo							hi
2	-4	6	1	5	-3	3	7
p	i						

i=2 pivot is currently in x[lo]

2	-4	6	1	5	-3	3	7
	p	i					

i=3

2	-4	6	1	5	-3	3	7
	p		i				

i=4

2	-4	1	6	5	-3	3	7
		p		i			
		[2]					

Quicksort Example (using Bentley's Algorithm)

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

lo							hi
				[4]	[5]		
2	-4	1	6	5	-3	3	7
		p			i		

still the pivot value

2	-4	1	-3	5	6	3	7
			p			i	

2	-4	1	-3	5	6	3	7
			p				i

2	-4	1	-3	5	6	3	7
			p				i

lo	hi	p	i
0	7	2	4
		3	5
			6
			7
			8

Quicksort Example (using Bentley's Algorithm)

[0] [3]

lo							hi
2	-4	1	-3	5	6	3	7
			p				i

our pivot

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

final resting place for "2"

lo							hi
-3	-4	1	$\boxed{2}$	5	6	3	7
			p				i

left   right partition

```
qsort(x, lo, p-1); run on left partition
qsort(x, p+1, hi); run on right partition
```

-4	-3	1	2	3	5	6	7

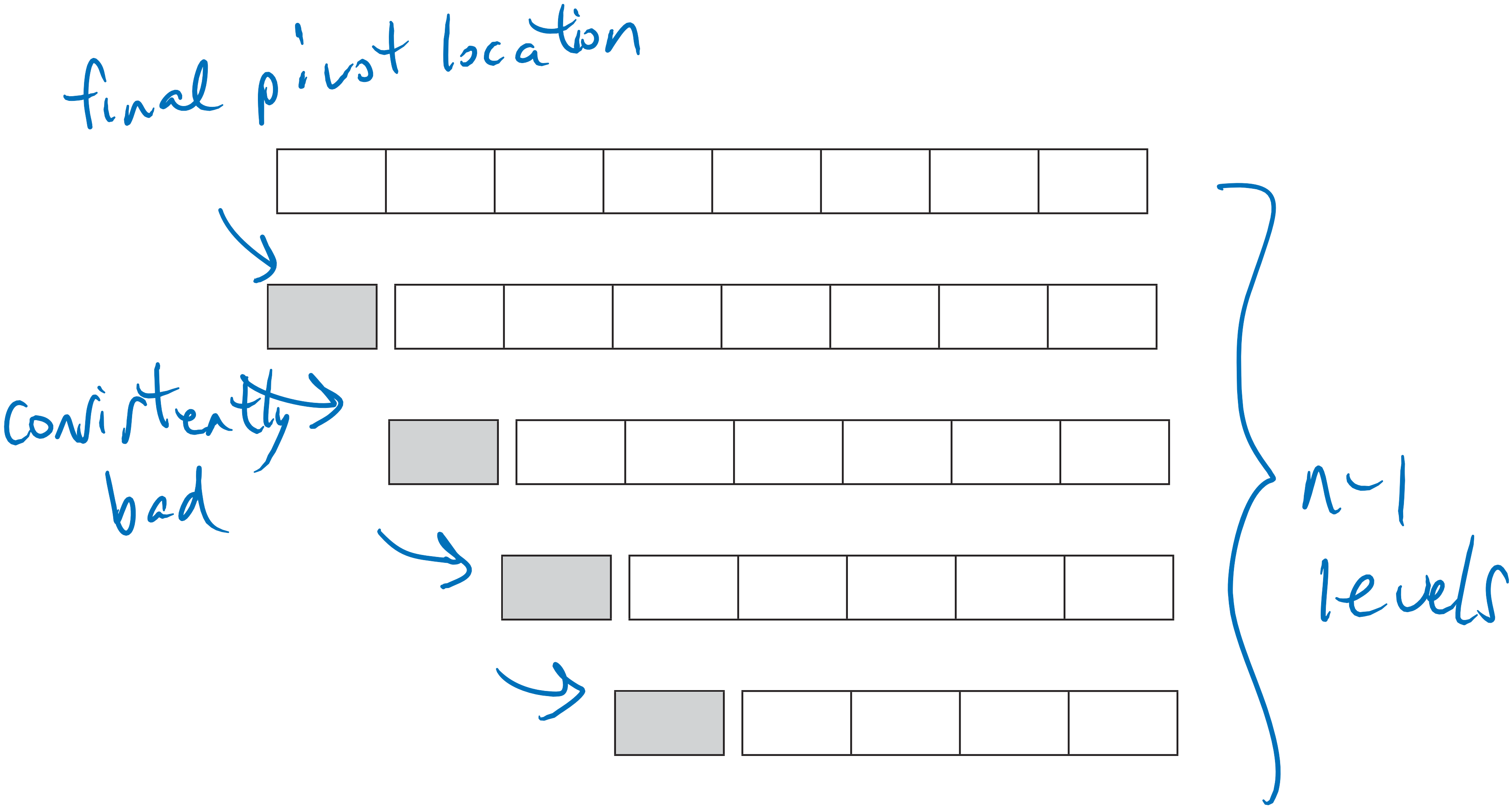
Quicksort: Running Time

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

1. Pick a pivot. - first element, last, random
   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?

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 $n-1$:

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

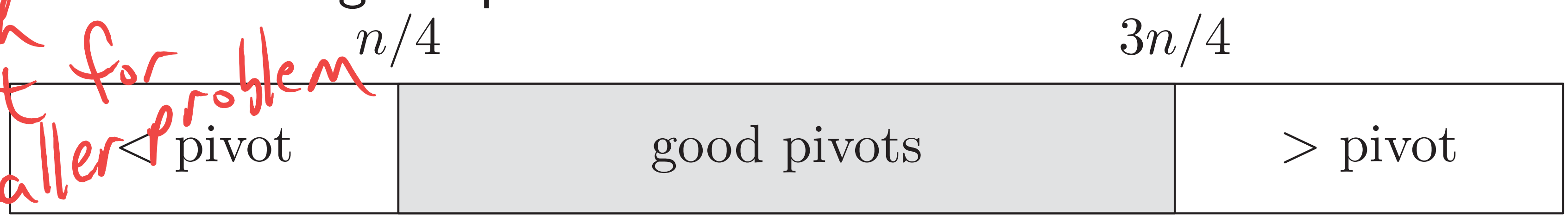
$\Theta(n^2)$

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.

left with $\frac{1}{2}$ list for our smaller problem



- 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.

$\log_2 n$

e.g., binary search: splits the list in half; so, we continue with the next iteration of $\frac{1}{2}$ list

Quicksort: Stability & Memory

Stable:

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

Memory:

In-place sort

Comparison of Running Times for 100 Samples

avg $O(n \lg n)$

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

$c\, n \lg n = 4.76$ sec; at $n \lg n \Rightarrow n = 3.00$

double? $(2n)\lg(2n)$

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

$= 2(3.0)\underbrace{\lg(2(3.0))}_{2.585} \approx 15.50$

$c = ?$

rough approx.

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		

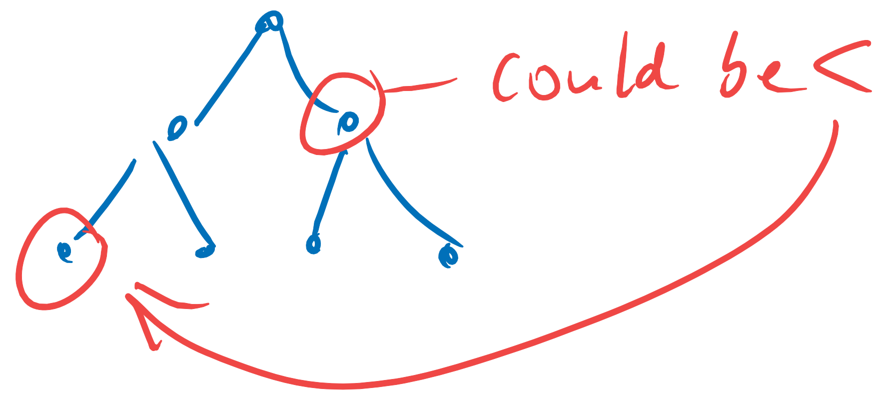
Some Quicksort/Mergesort implementations use Insertion Sort on small arrays (base cases).

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?

general sorts are comparison-based

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.

179 28 64 38

64 28 38 179

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...

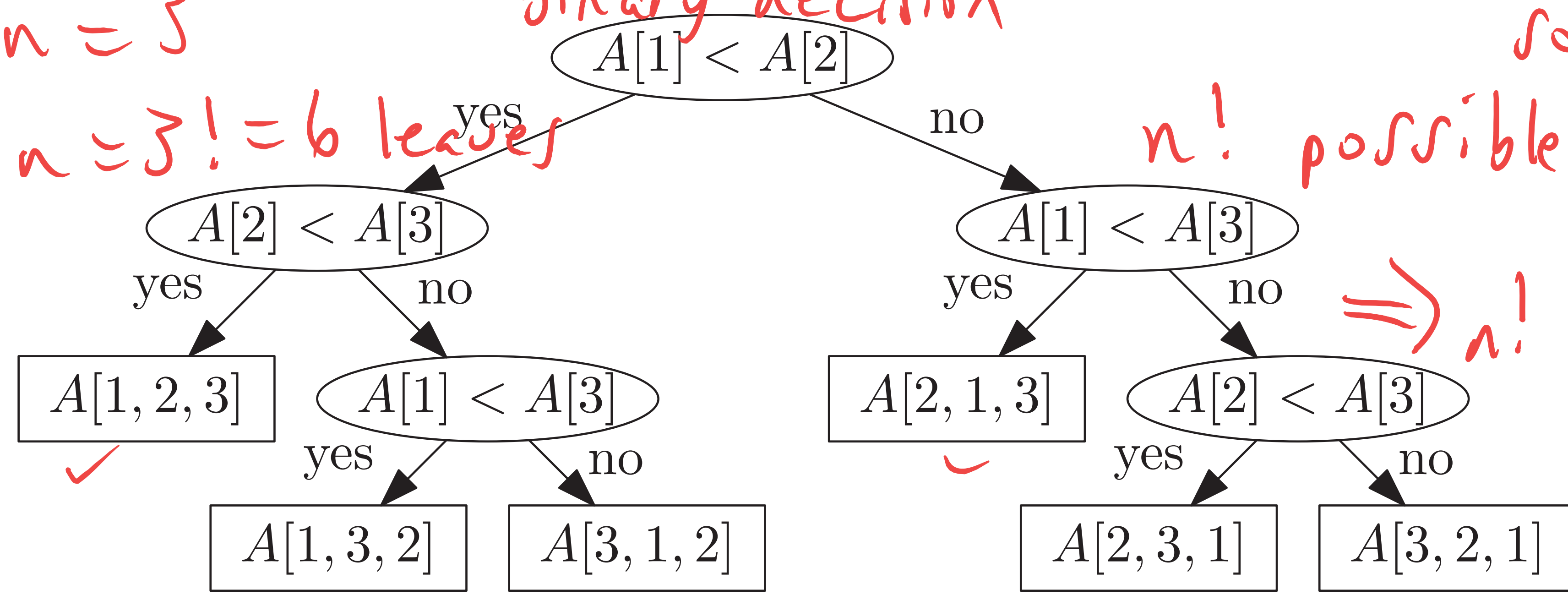
n items to sort

binary decision

$n = 3$

$\therefore n = 3! = 6$ leaves

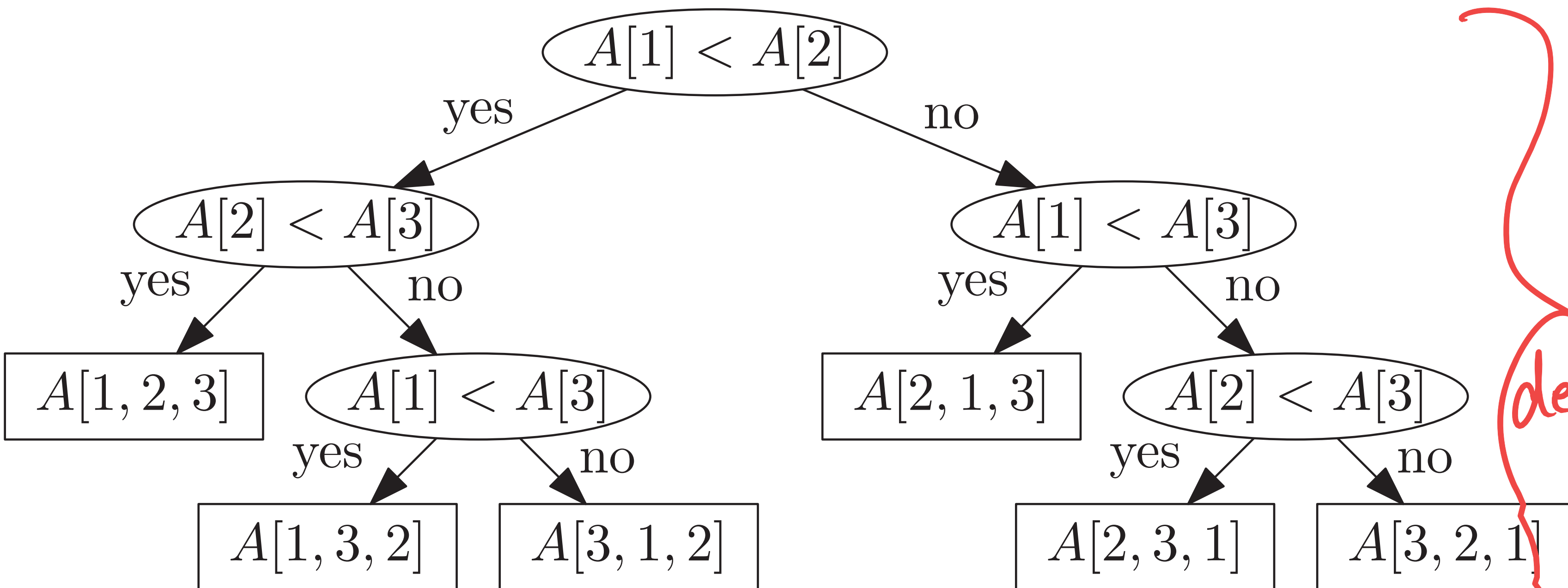
$n!$ possible outcomes $\Rightarrow n!$ leaves



$A[1] < A[3] < A[2]$

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 $A[3]$, $A[1]$, $A[2]$.



comparing 2 at a time

$O(\lg n!)$

decisions

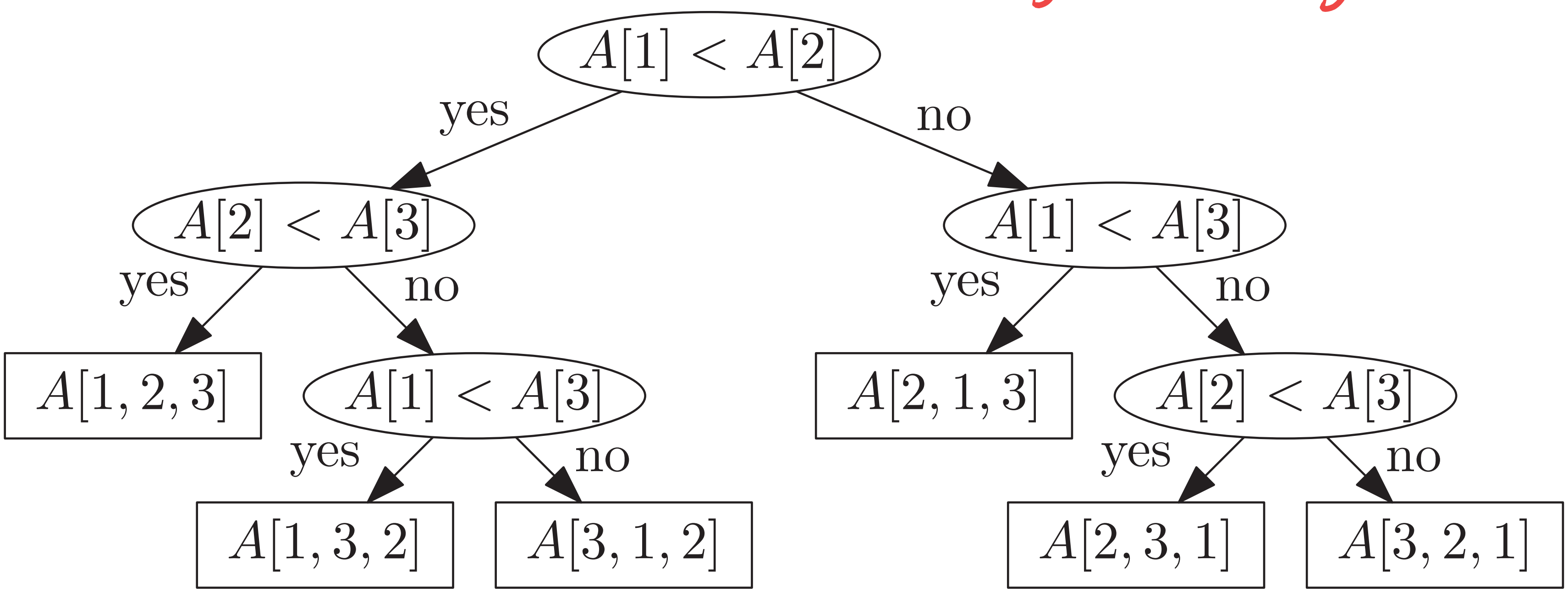
left off the ! in class

n! leaves

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 $[\texttt{cat}, \texttt{ant}, \texttt{bee}]$.

$A[2] < A[3] < A[1]$



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 $\Omega(n \log n)$.

from a previous unit