Unit #4: Sorting

CPSC 221: Algorithms and Data Structures

Will Evans and Jan Manuch

2016W1

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.

How to Measure Sorting Algorithms

- 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 $k \ll n$)?
- Stability: What happens to elements with identical keys?

  Why do we care? → Sort by secondary key first (e.g. first name)

  → then sort by primary key (e.g. last name)
- Memory Usage: How much extra memory is used?

we get this if sorting alg. is stable

A. Lo
B. Li
C. Li
D. Li

$\xrightarrow[\text{by last name}]{\text{stable sort}}$

B. Li
C. Li
D. Li
A. Lo

original order preserved for elements with equal keys

Insertion Sort: Running Time

Loop invariant:

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.

$0 \ldots i-1$ sorted $\quad A[i]$

Worst case:

0	1	2						n−2	n−1
n	$n-1$	$n-2$	$n-3$					2	1

$n-2 \quad n-1 \quad n$

$A[3]$ inserting $A[3]$ .. 3 comparisons

insert $A[i]$ .. i comparisons

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

Best case:

1	2	3						$n-1$	n

insert $A[i]$..
1 comparison

$$\sum_{i=1}^{n-1} 1 = n-1 \in \Theta(n)$$

Average case:

inserting $A[i]$ .. between $1 \ldots i$ comparisons
on average $\frac{1+i}{2}$

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

... half of the worst case

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.

i

| 1 | 2 | 3 | 2 | $\rightarrow$ | 1 | 2 | 2 | 3 |

Easily made stable:

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

...original Insertion Sort from Unit #03 is stable.

Memory:

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

Heapsort (1964 Williams) (1964 Floyd)

1'. (Repeat n times: Perform insert.) ... $\Theta(n \log n)$

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

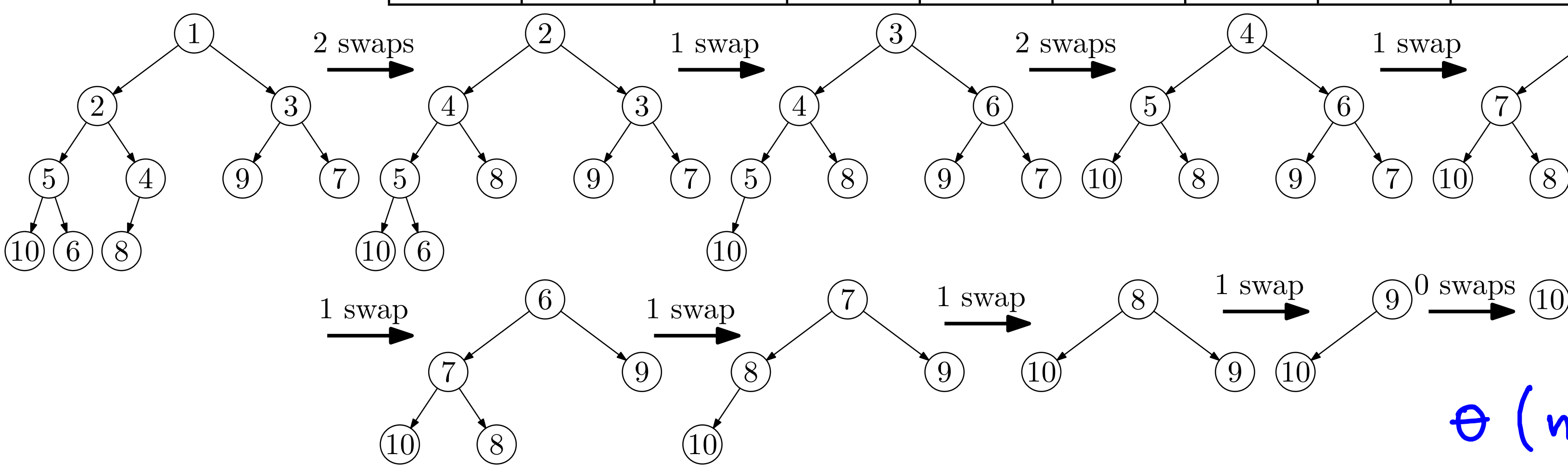
$$T(n) \le \lg(n) + \lg(n-1) + \ldots + \lg(2) + \lg(1)$$
$$= \lg(n!) \in \Theta(n \log n)$$

Worst case: $O(n \log n)$

Total time: $\Theta(n) + O(n \log n)$
$= O(n \log n)$.

Best case[1]:

1	2	3	5	4	9	7	10	6	8



$\Theta(n \log n)$

they proved

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

Heapsort: Stability & Memory

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

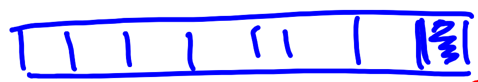
Not stable:

Hack: Use index in input array to break comparison ties.
(but this takes more space.)

would fix any unstable sorting alg.
but adds complexity

Memory:

free after deleteMin



place min here

- **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.
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.

$\rightarrow$ n steps $\quad T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + n$

Recurrence: $\quad T(n) = 2T(n/2) + n$

$\rightarrow \Theta(n \log n)$

Mergesort Example

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

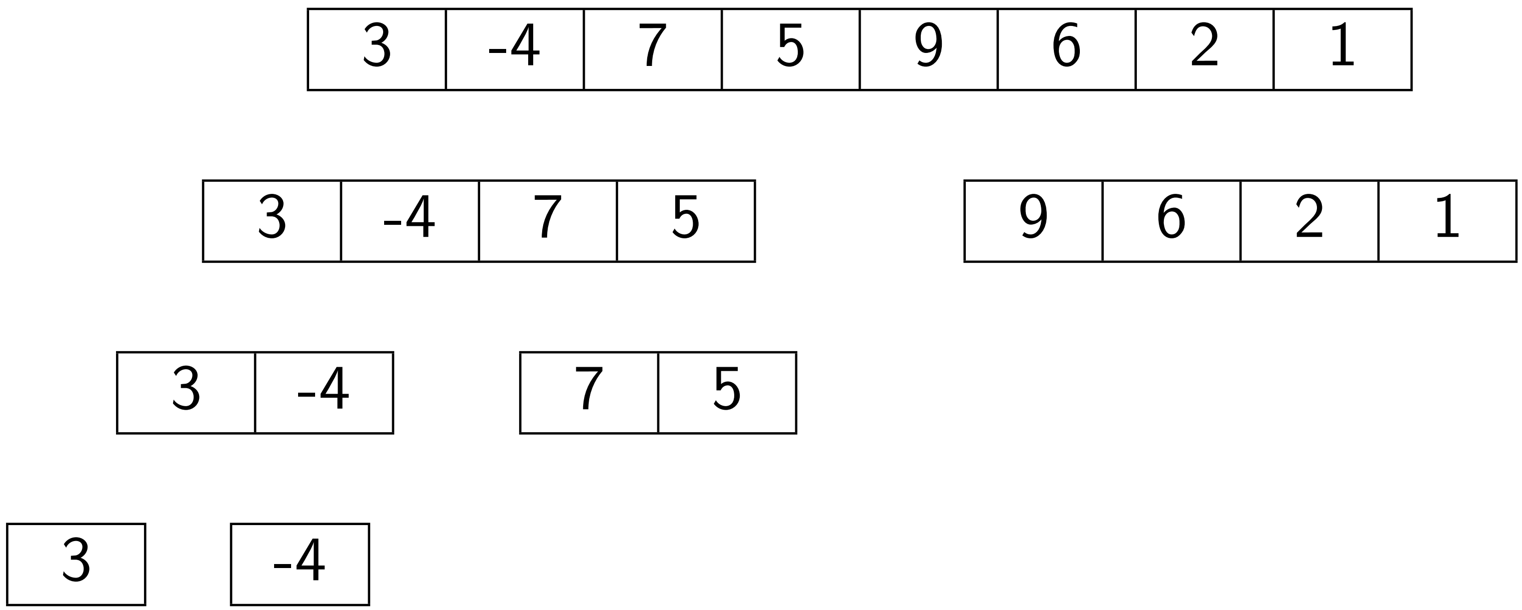
3	-4	7	5

9	6	2	1

3	-4

7	5

Mergesort Example



Mergesort Example

3	-4	7	5	9	6	2	1

3	-4	7	5

9	6	2	1

3	-4

7	5

3

-4

-4	3

* merge

Mergesort Example

3	-4	7	5	9	6	2	1

3	-4	7	5

9	6	2	1

3	-4

7	5

3

-4

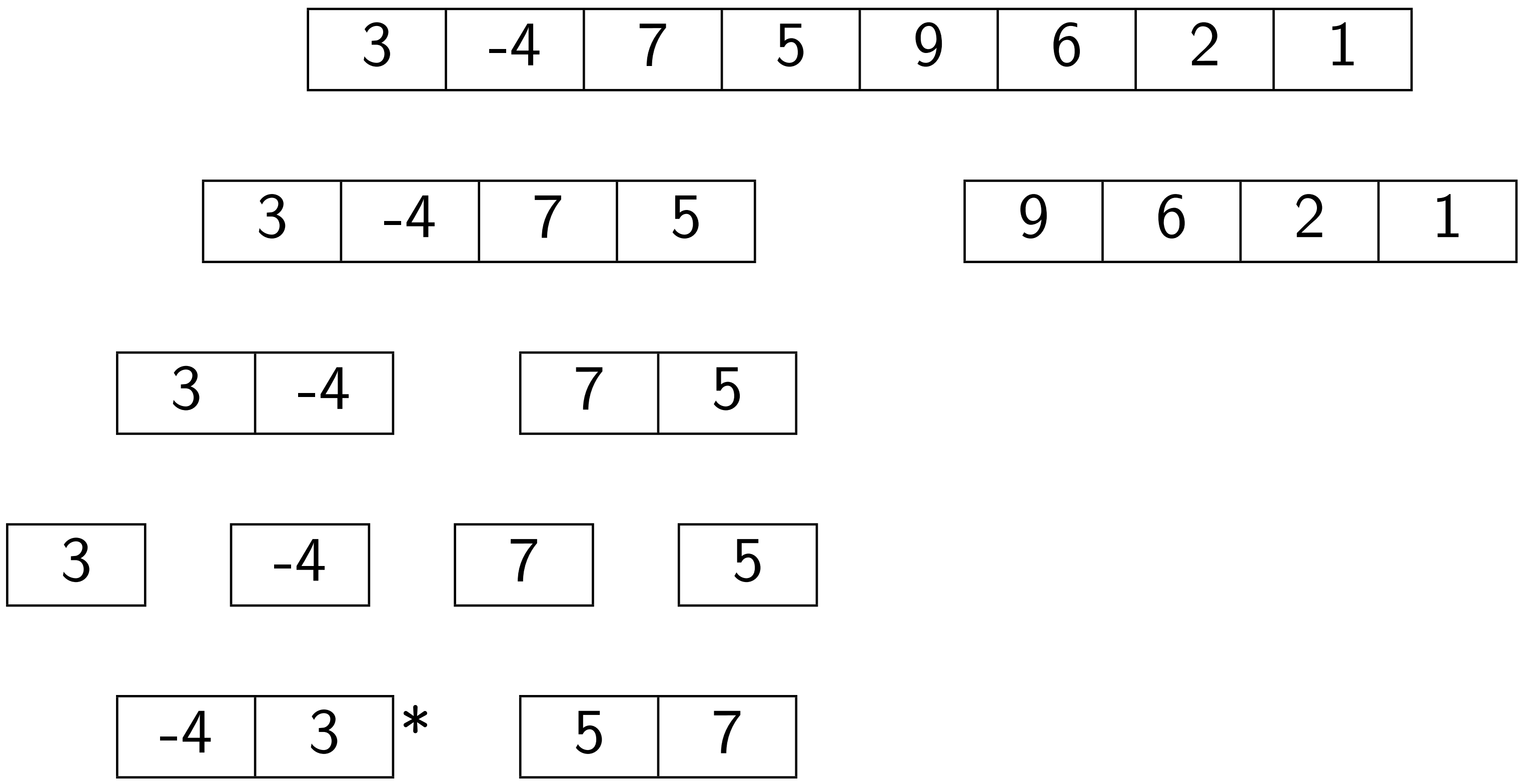
7

5

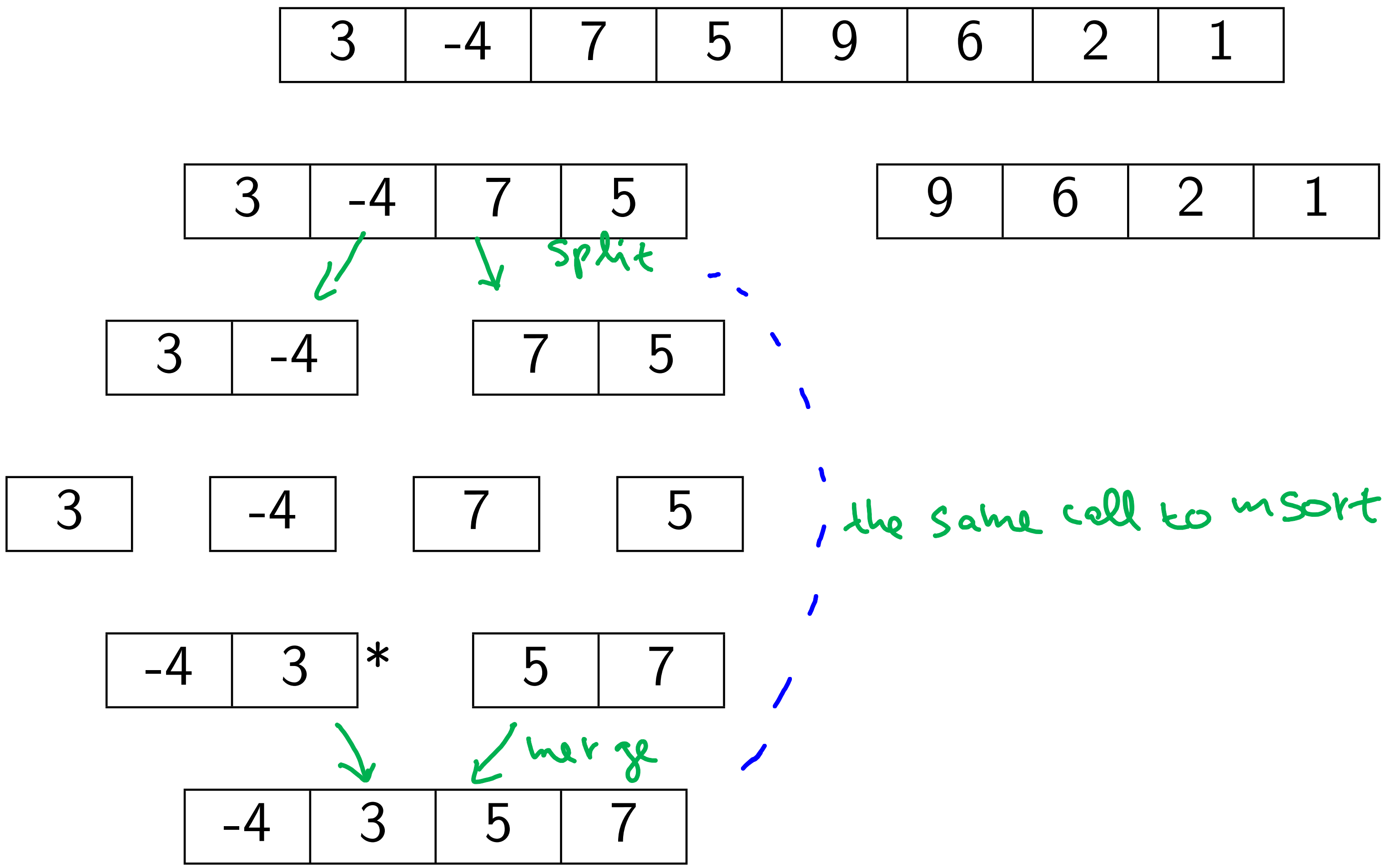
-4	3

*

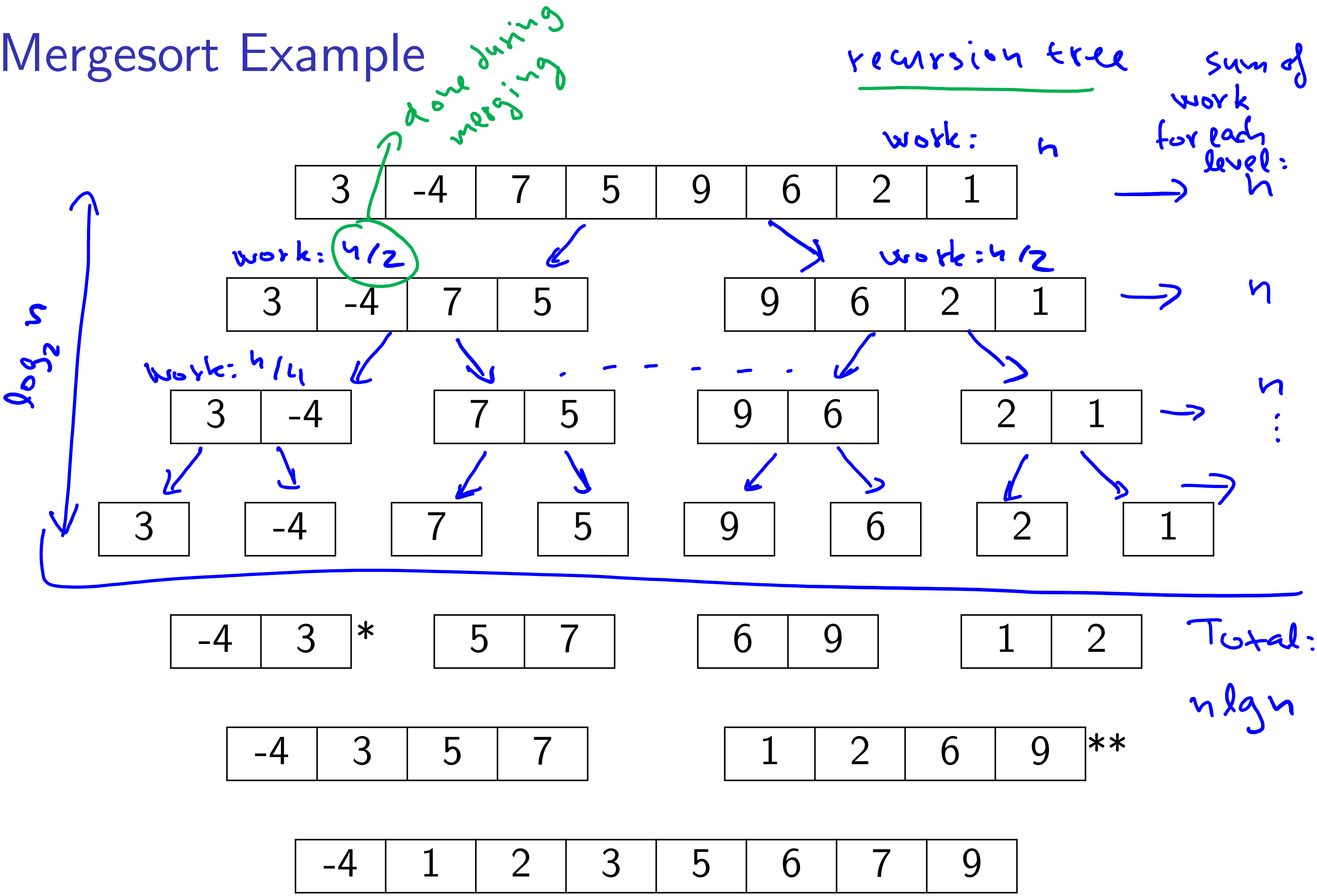
Mergesort Example



Mergesort Example



Mergesort Example



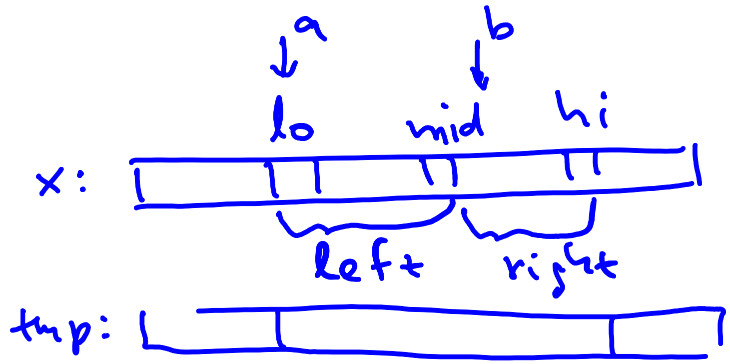
Mergesort Code

x [lo... hi]

```
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);
}

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

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++ ) {
    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];
}
```

< loop invariant *

to make it stable

copy from tmp to x

Loop invariant:

* tmp[lo..k-1] • sorted

• contains elements from x[lo...a-1] and x[mid+1...b-1]

• other elements of x[lo..hi] are ≥ to tmp[lo..k-1]

Sample Merge Steps

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

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 **
```

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

```
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.

depends on the exact definition

Memory:

Not easy to implement without using $\Omega(n)$ extra space, so it is not viewed as an in-place sort.

+ memory on call stack: $\Theta(\log n)$

→ no array elements on call stack

(call stack only stores indices)

Quicksort (C.A.R. Hoare 1961)

In practice, one of the fastest sorting algorithms.

1. Pick a **pivot**

a → ← b

x :

2	-4	6	1	5	-3	3	7

Hoare's partition alg.

- scan with a until you find $x[a] \geq$ pivot
- scan with b until you find $x[b] <$ pivot
- swap$(x[a], x[b])$ and repeat.

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

-4	1	-3	2	6	5	3	7

left partition pivot right partition

in right place (will not move anymore)

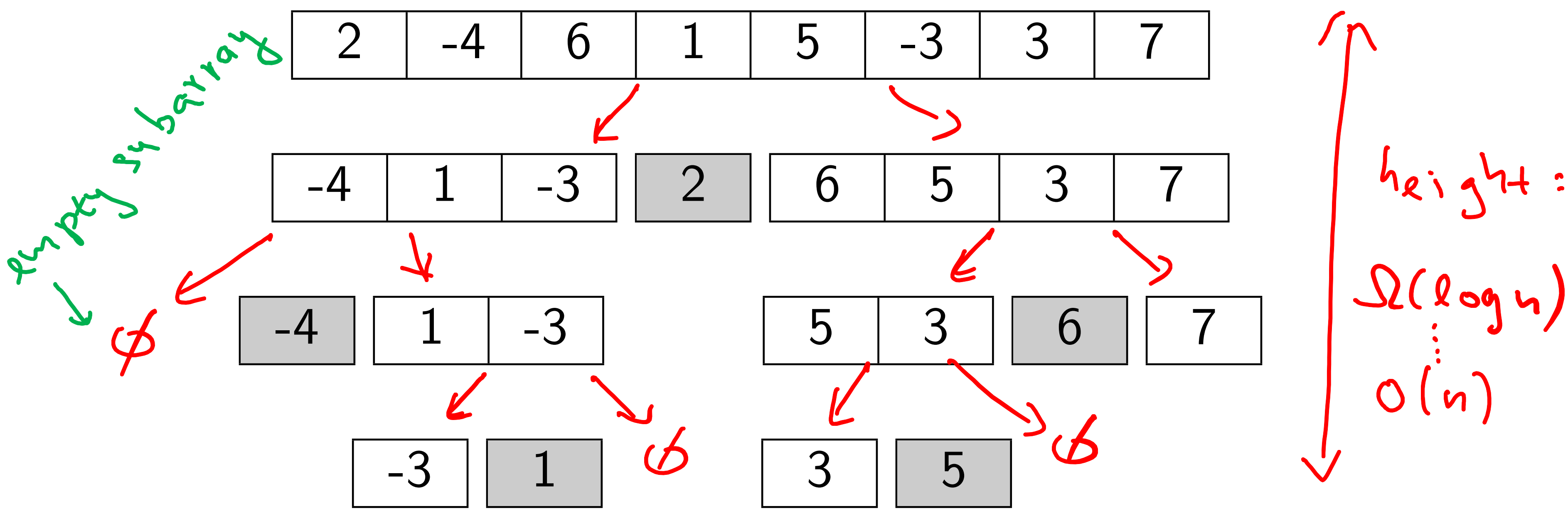
3. Recursively sort each partition.

base case?

$n = \begin{cases} 0 \\ 1 \end{cases}$

Quicksort Visually

recursion tree:



Quicksort by Jon Bentley

from Programming Pearls
[by Lomato]

```
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);
}

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

pivot

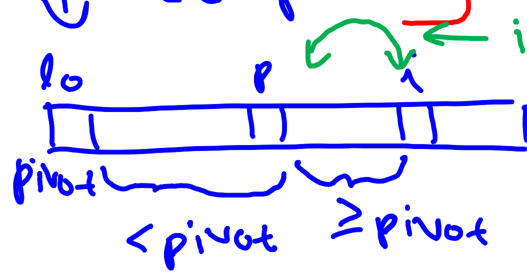
++p vs. p++

return p+1 return p

increment p

loop invariant:

if x[i] < pivot



in words:

$x[lo+1 .. p] < pivot$

$x[p+1 ... i-1] \geq pivot$

Lomato's Partition algorithm for Quicksort (loop invariant)

October 25, 2016 5:49 PM

```
p = lo;
for( i=lo+1; i <= hi; i++ )
  if( x[i] < x[lo] ) swap(x[++p], x[i]);
swap(x[lo], x[p]);
```

Loop invariant:
(a) $x[lo+1..p] < \text{pivot}$
(b) $x[p+1..i-1] \geq \text{pivot}$

Task: Prove and use loop invariant to show correctness of Lomato's alg.

Induction on # of iterations:

Base case (before the first iteration).

$i = lo+1, \; p = lo$

(a) $x[lo+1..p] = x[lo+1..lo]$ = empty array ✓

(b) $x[p+1..i-1] = x[lo+1..lo]$ = empty array ✓ □

Inductive step: We need to consider 2 cases:

i.h.: (a) & (b) hold at the beginning of some iteration.

(1) $x[i] < \underbrace{x[lo]}_{\text{pivot}}$: $i_{new} = i+1, \; p_{new} = p+1$, swap$(x[p+1], x[i])$ } what happens

checking invariant after this iteration

(a) $x[lo+1..\underset{p+1}{p_{new}}] = \underbrace{x[lo+1 \ldots p]}_{\text{by i.h.} < \text{pivot}} \; \underline{x[p+1]}$

holds value of $x[i]$ before the swap

$< \text{pivot} \ldots$ ✓

(b) $x[p_{new}+1 \ldots i_{new}-1] = x[p+2 \ldots i]$

$\underbrace{x[p+1 \ldots\ldots i-1]}_{\geq \text{pivot}}$

by i.h. $\geq$ pivot

$x[i]$ holds value of $x[p+1]$ $\geq$ pivot by i.h. ✓

(2) $x[i] \geq \underset{\text{pivot}}{x[lo]}$: $i_{new} = i+1, \; p_{new} = p$ ← what happens in this case

checking invariant

(a) $x[lo+1 \ldots p_{new}] = x[lo+1 \ldots p] <$ pivot by i.h. ✓

(b) $x[p_{new}+1 \ldots i_{new}-1] = x[\underbrace{p+1 \ldots}_{\geq \text{pivot by i.h.}} i]$ ← $x[i] \geq$ pivot ✓ □

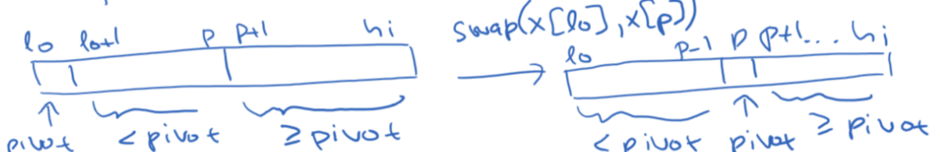
∴ Invariant holds.

Use invariant: Loop will stop when $i = hi+1$.

$x[lo] = $ pivot

invariant ⇒ { (a) $x[lo+1..p] <$ pivot
(b) $x[p+1 \ldots hi] \geq$ pivot

the swap after the loop



partitioning accomplished!

Quicksort Example (using Bentley's Algorithm)

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

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

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

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

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

Quicksort Example (using Bentley's Algorithm)

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

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

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

Quicksort Example (using Bentley's Algorithm)

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

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

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

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

-4	-3	1	2	3	5	6	7

Quicksort: Running Time

Running time is proportional to number of comparisons so...
Let's count comparisons.

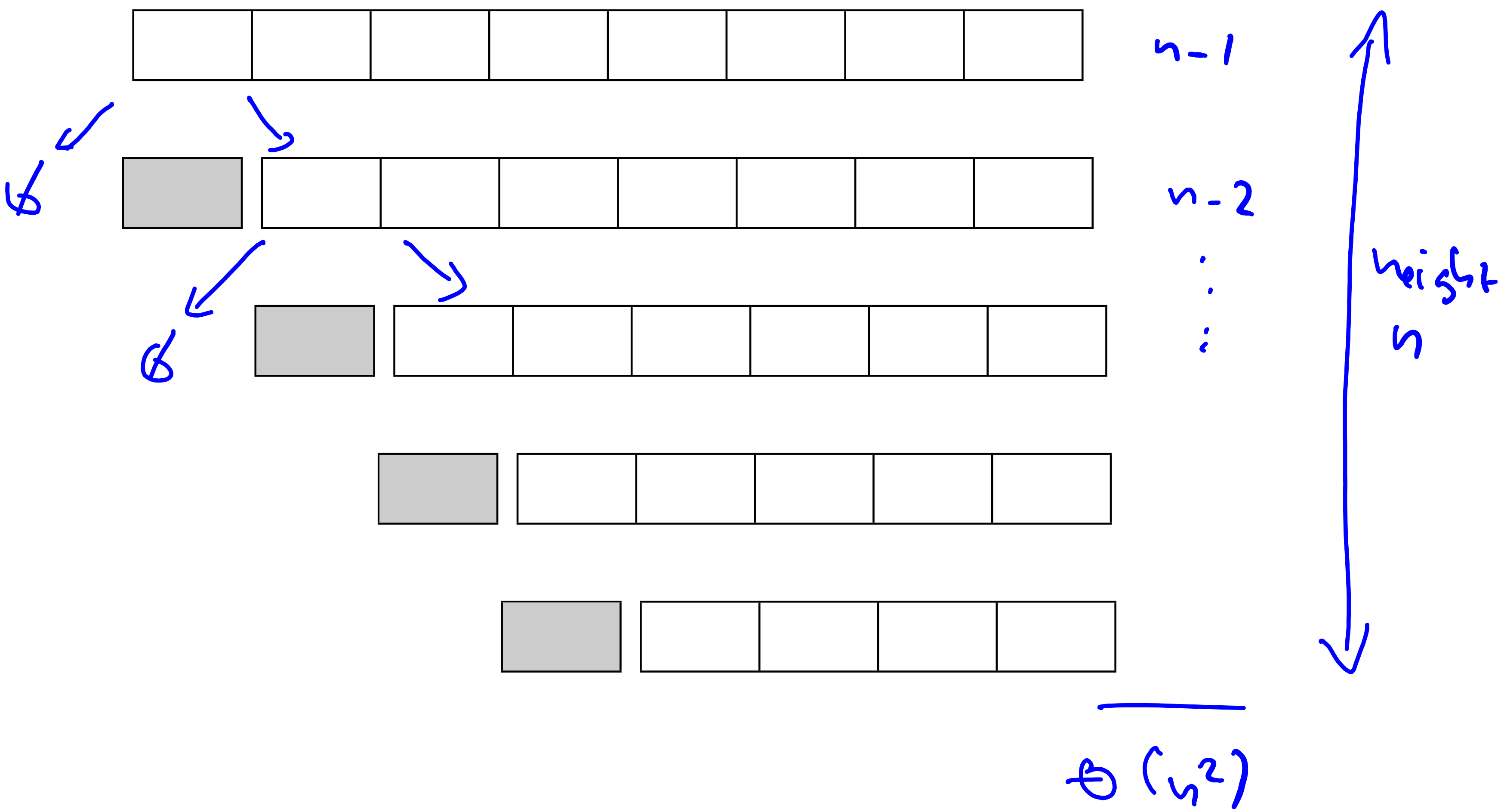
1. Pick a pivot.
   Zero comparisons
2. Reorder (partition) array around the pivot.
   Quicksort compares each element to the pivot.
   $n - 1$ comparisons
3. Recursively sort each partition.
   Depends on the size of the partitions.

- 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 $n - 1$.

best case: $\Theta(n \log n)$

Quicksort Visually: Worst case

Example: input is sorted



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 $n-1$:

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

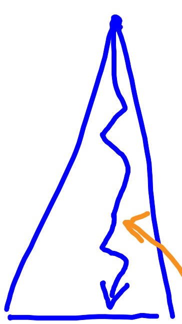
This is $\Theta(n^2)$ comparisons.

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.

	$n/4$	$3n/4$
$<$ pivot	good pivots	$>$ pivot

recursion tree



- Any good pivot creates two partitions of size at most $3n/4$.
- We expect to pick one good pivot every two tries.

every second pivot is bad

- 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.

on this path approx. 1/2 of pivots are bad and approx 1/2 are good

assume no progress

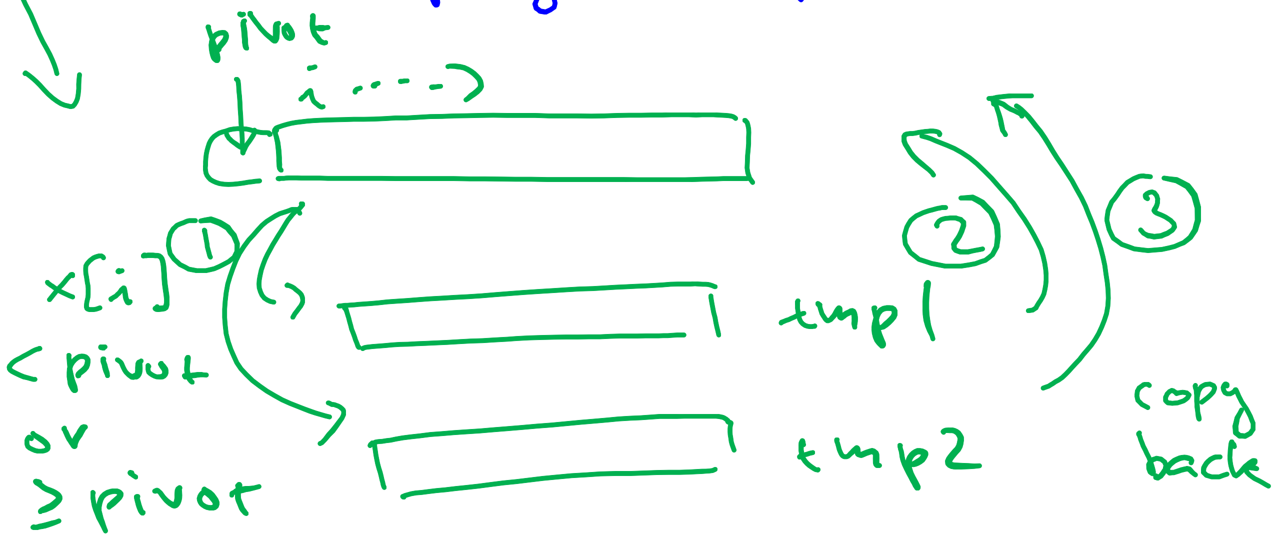
assume $\frac{3}{4}n$ progress

Quicksort: Stability & Memory

Stable:

Can be made stable, most easily by using more memory.

Example of such implementation



Memory:

In-place sort.

→ Yes

Note. size of call stack ≈ depth in the recursion tree

→ No if counting any memory used by call stack

max size of : $\Omega(\log n) \ldots O(n)$

Compare: Running Times (100 samples)

converging to average case

... to worst-case

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

Code is from lecture notes and labs (not optimized).

Compare: Quick, Merge, Heap, and Insert 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 Quick/Merge implementations use Insert 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 Merge's best case linear.

Compare: Quick, Merge, Heap, and Insert Sort

Stability

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

For all four:

Memory use → Total space: $\Theta(n)$

- $\underbrace{\text{Insert, Heap, Quick}}_{\text{in-place}} <$ Merge

Example when Heapsort might be useful:

- given n elements, print k smallest ones in sorted order (for instance, when displaying files in a window, only k can be shown)

-> then using Heapify + k × DeleteMin we get alg. with runtime $\Theta(n + k \log n)$

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, Heap, Merge, and Quick sort are comparison-based.
Radix sort is not.

Bucket sort

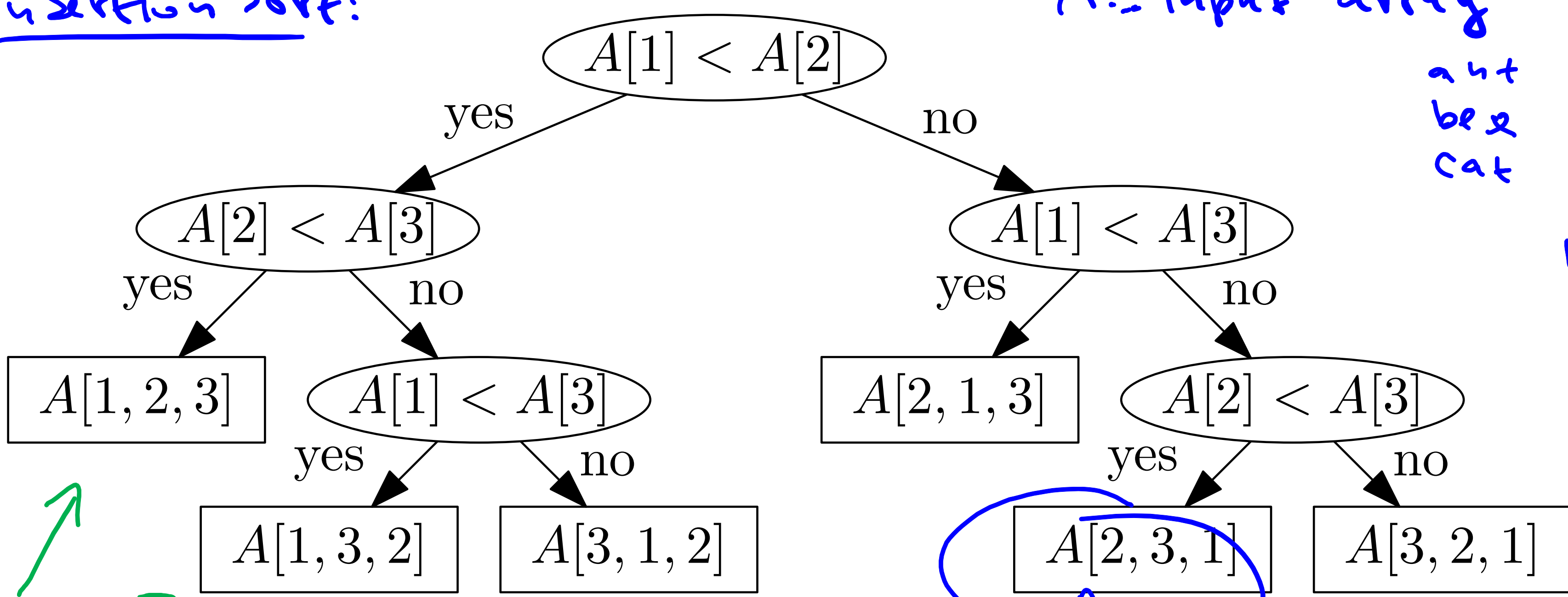
Comparison-based algorithms using a Decision Tree model

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...

Insertion Sort:

A... input array

ant
bee
cat



height=3

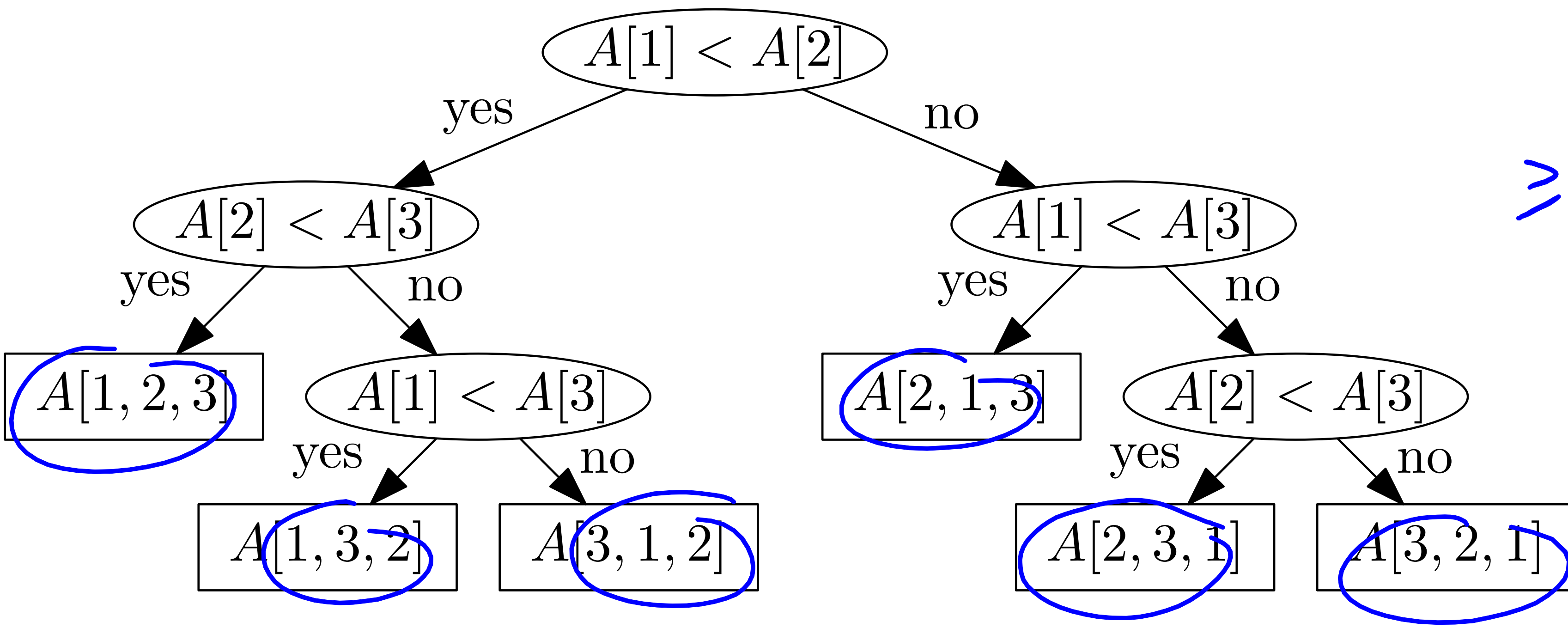
outputs (sorted A)

order

Input: [cat, ant, bee]

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

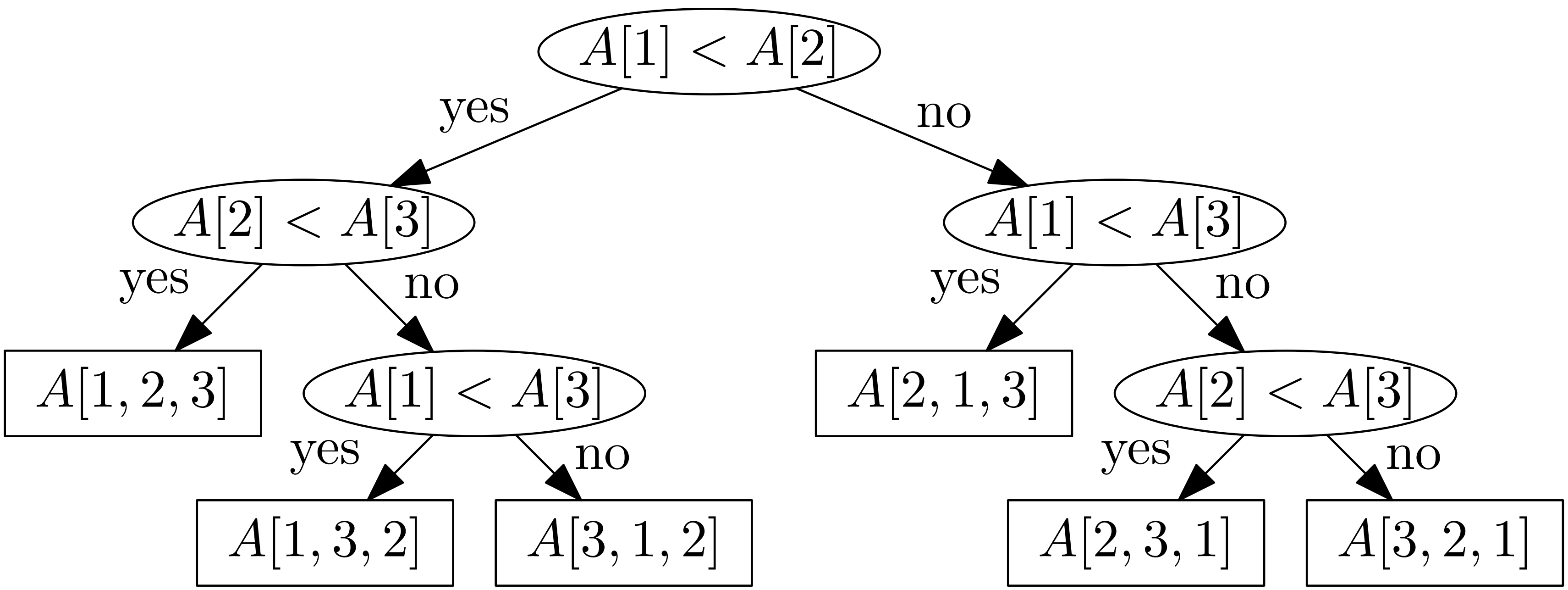


$\geq \lg(n!)$

$\geq n!$

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



Complexity of the Sorting Problem

- The number of leaves is at least $n!$.
- The height of the decision tree is at least $\lceil \lg(n!) \rceil$. $\in \Omega(n \log n)$
- 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** so the complexity of the sorting problem is $\Omega(n \log n)$.