CS221: Algorithms and Data Structures

Sorting Takes Priority

Steve Wolfman

(minor tweaks by Alan Hu)

Today’s Outline

- Sorting with Priority Queues, Three Ways

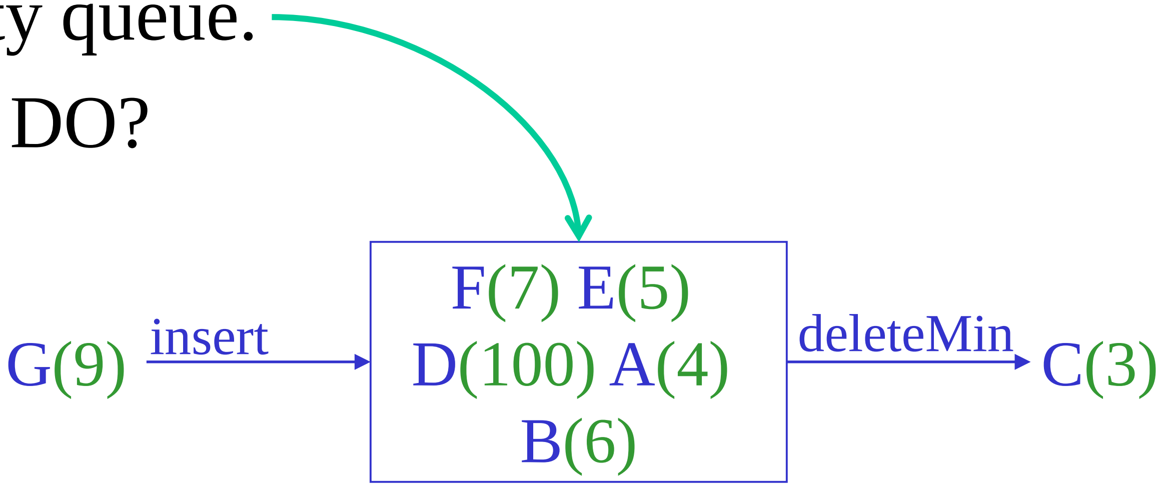
How Do We Sort with a Priority Queue?

You have a bunch of data.

You want to sort by priority.

You have a priority queue.

WHAT DO YOU DO?



“PQSort”

```
Sort(elts):
  pq = new PQ
  for each elt in elts:
    pq.insert(elt);
  sortedElts = new array of size elts.length
  for i = 0 to elts.length – 1:
    sortedElts[i] = pq.deleteMin
  return sortedElts
```

What sorting algorithm is this?

a. Insertion Sort
b. Selection Sort
c. Heap Sort
d. Merge Sort
e. None of these

“PQSort”

```
Sort(elts):
  pq = new PQ
  for each elt in elts:
    pq.insert(elt);
  sortedElts = new array of size elts.length
  for i = 0 to elts.length – 1:
    sortedElts[i] = pq.deleteMin
  return sortedElts
```

What sorting algorithm is this?

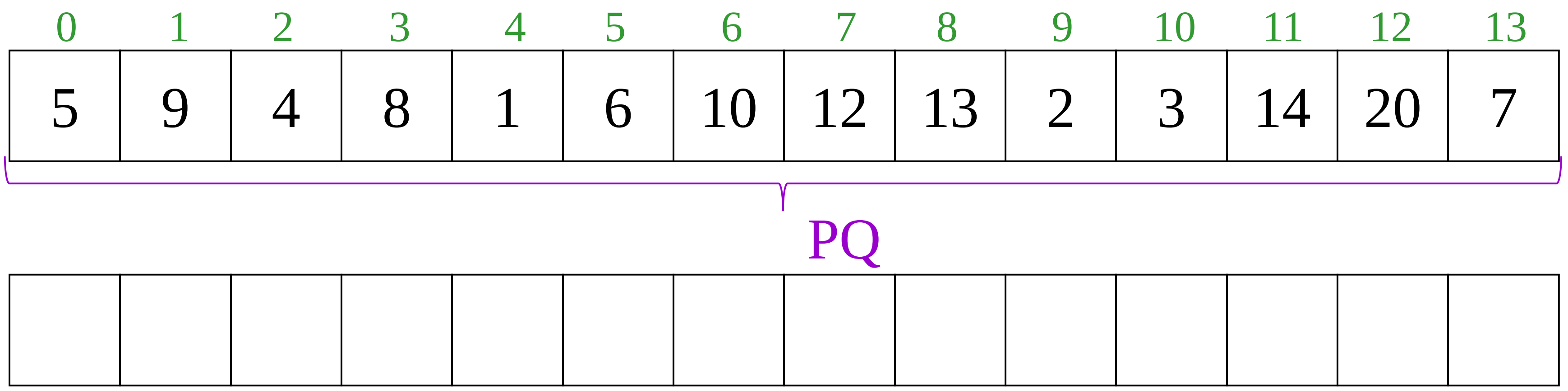
a. Insertion Sort
b. Selection Sort
c. Heap Sort
d. Merge Sort
e. None of these

Abstract Data Type
vs.
Data Structure That Implements It

Reminder:
Naïve Priority Q Data Structures

- Unsorted list:
  - *insert:* worst case O(1)
  - *deleteMin:* worst case O(n)
- Sorted list:
  - *insert:* worst case O(n)
  - *deleteMin:* worst case O(1)

"PQSort" deleteMins with Unsorted List PQ



“PQSort” deleteMins with Unsorted List PQ

0	1	2	3	4	5	6	7	8	9	10	11	12	13
5	9	4	8	1	6	10	12	13	2	3	14	20	7

PQ

“PQSort” deleteMins with Unsorted List PQ

0	1	2	3	4	5	6	7	8	9	10	11	12	13
5	9	4	8	6	10	12	13	2	3	14	20	7	

PQ

1													

"PQSort" deleteMins with Unsorted List PQ

0	1	2	3	4	5	6	7	8	9	10	11	12	13
5	9	4	8	6	10	12	13	2	3	14	20	7	

PQ

1													

“PQSort” deleteMins with Unsorted List PQ

0	1	2	3	4	5	6	7	8	9	10	11	12	13
5	9	4	8	6	10	12	13	3	14	20	7		

PQ

1	2												

“PQSort” deleteMins with Unsorted List PQ

0	1	2	3	4	5	6	7	8	9	10	11	12	13
5	9	4	8	6	10	12	13	14	20	7			

PQ

1	2	3											

"PQSort" deleteMins with Unsorted List PQ

0	1	2	3	4	5	6	7	8	9	10	11	12	13
5	9	8	6	10	12	13	14	20	7				

PQ

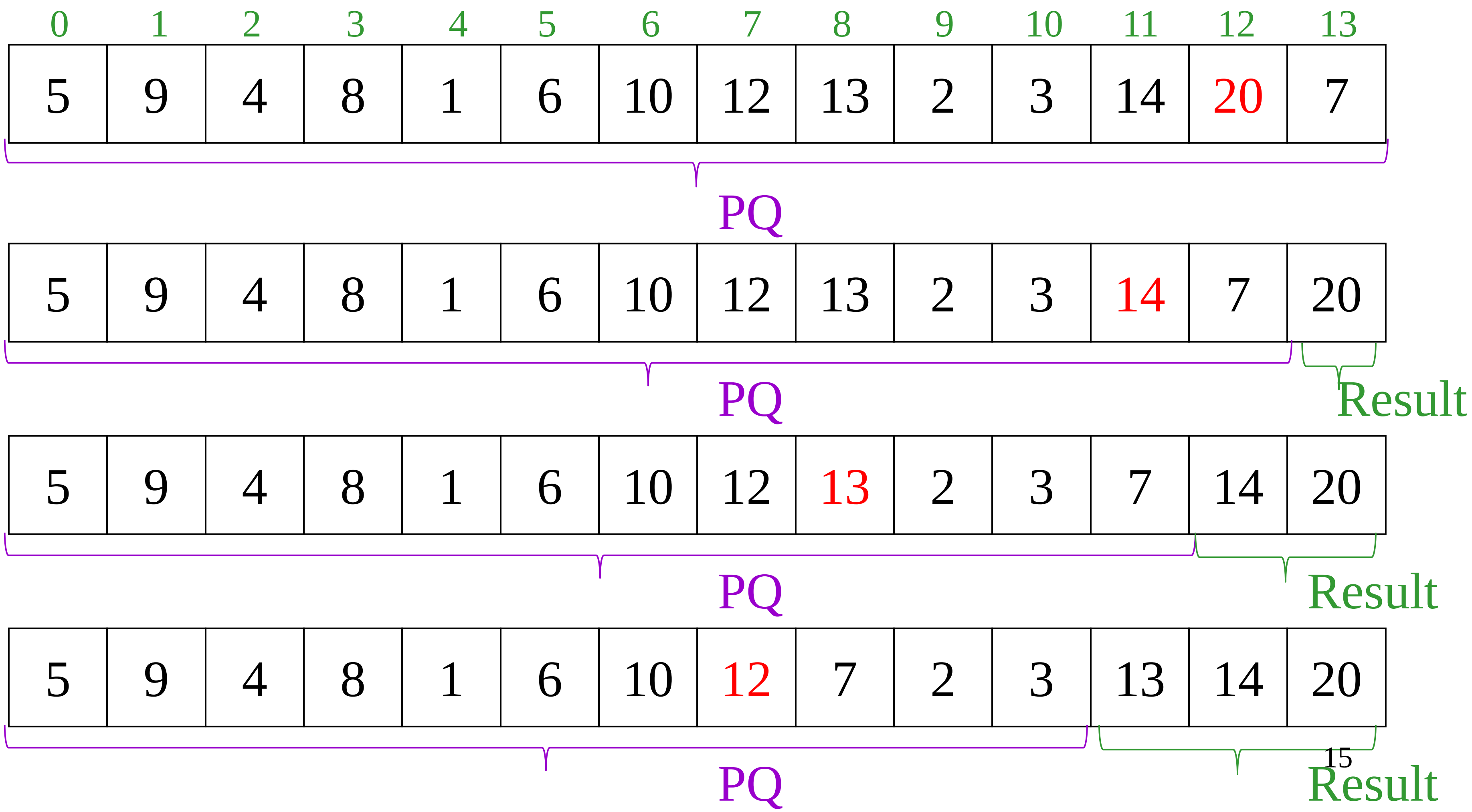
1	2	3	4										

Two PQSort Tricks

1) Use the array to store both your results and your PQ. No extra memory needed!

2) Use a max-heap to sort in increasing order (or a min-heap to sort in decreasing order) so your heap doesn’t “move” during deletions.

“PQSort” deleteMaxes with Unsorted List MAX-PQ



0	1	2	3	4	5	6	7	8	9	10	11	12	13
5	9	4	8	1	6	10	12	13	2	3	14	20	7

PQ

5	9	4	8	1	6	10	12	13	2	3	14	7	20

PQ Result

5	9	4	8	1	6	10	12	13	2	3	7	14	20

PQ Result

5	9	4	8	1	6	10	12	7	2	3	13	14	20

PQ Result

“PQSort” deleteMaxes with Unsorted List MAX-PQ

How long does “build” take? No time at all!

How long do the deletions take? Worst case: $O(n^2)$ ☹

What algorithm is this?

a. Insertion Sort

b. Selection Sort

c. Heap Sort

d. Merge Sort

e. None of these

$$\underbrace{\boxed{5}\boxed{9}\boxed{4}\boxed{8}\boxed{1}\boxed{6}\boxed{10}\boxed{12}\boxed{7}\boxed{2}\boxed{3}}_{\text{PQ}}\underbrace{\boxed{13}\boxed{14}\boxed{20}}_{\text{Result}}$$

“PQSort” insertions with Sorted List MAX-PQ

0	1	2	3	4	5	6	7	8	9	10	11	12	13
5	9	4	8	1	6	10	12	13	2	3	14	20	$\underbrace{7}_{PQ}$

5	9	4	8	1	6	10	12	13	2	3	14	$\underbrace{7 \quad 20}_{PQ}$	

5	9	4	8	1	6	10	12	13	2	3	$\underbrace{7 \quad 14 \quad 20}_{PQ}$		

5	9	4	8	1	6	10	12	13	2	$\underbrace{3 \quad 7 \quad 14 \quad 20}_{PQ}$			

“PQSort” insertions with Sorted List MAX-PQ

How long does “build” take? Worst case: $O(n^2)$ ☹

How long do the deletions take? No time at all!

What algorithm is this?

a. Insertion Sort

b. Selection Sort

c. Heap Sort

d. Merge Sort

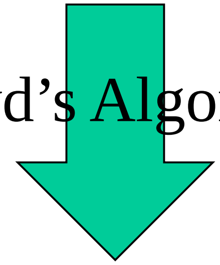
e. None of these

5	9	4	8	1	6	10	12	13	2	3	7	14	20

$\underbrace{3 \quad 7 \quad 14 \quad 20}_{\text{PQ}}$

"PQSort" Build with Heap MAX-PQ

0	1	2	3	4	5	6	7	8	9	10	11	12	13
5	9	4	8	1	6	10	12	13	2	3	14	20	7


Floyd's Algorithm

20	13	14	12	3	6	10	9	8	2	1	4	5	7

$\underbrace{\hspace{20em}}_{PQ}$

Takes only O(n) time!

"PQSort" deleteMaxes with Heap MAX-PQ

0	1	2	3	4	5	6	7	8	9	10	11	12	13
20	13	14	12	3	6	10	9	8	2	1	4	5	7

PQ

14	13	10	12	3	6	7	9	8	2	1	4	5	20

PQ

13	12	10	9	3	6	7	5	8	2	1	4	14	20

PQ

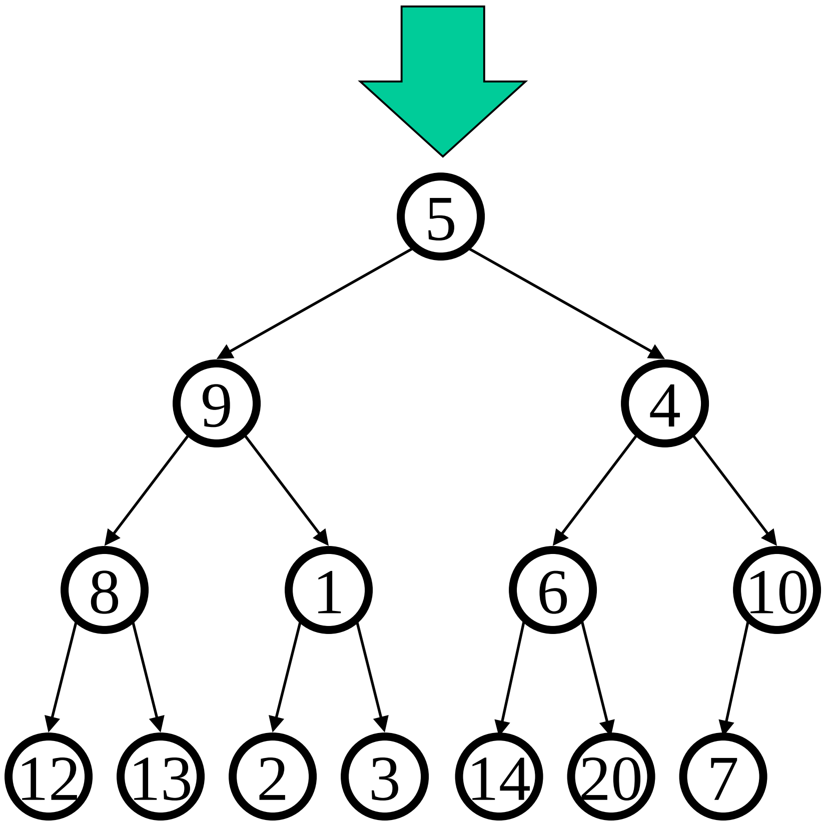
12	9	10	8	3	6	7	5	4	2	1	13	14	20

PQ

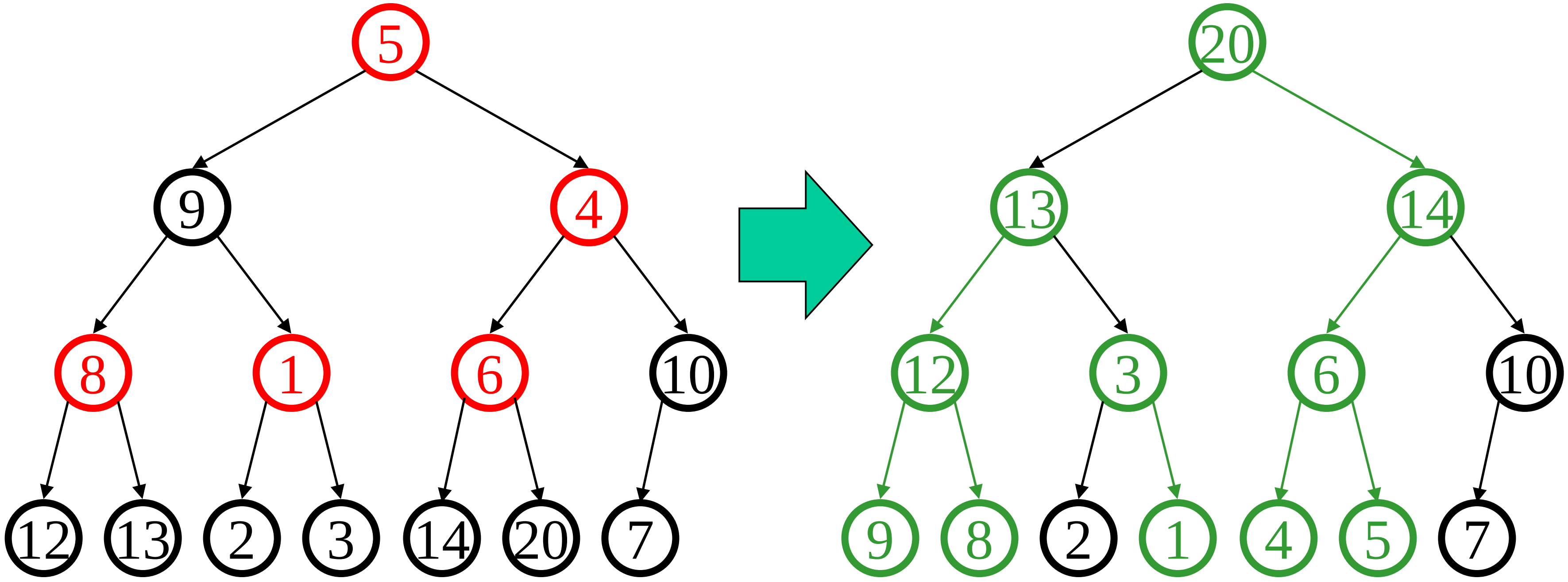
Totally incomprehensible as an array!

"PQSort" deleteMaxes with Heap MAX-PQ

5	9	4	8	1	6	10	12	13	2	3	14	20	7



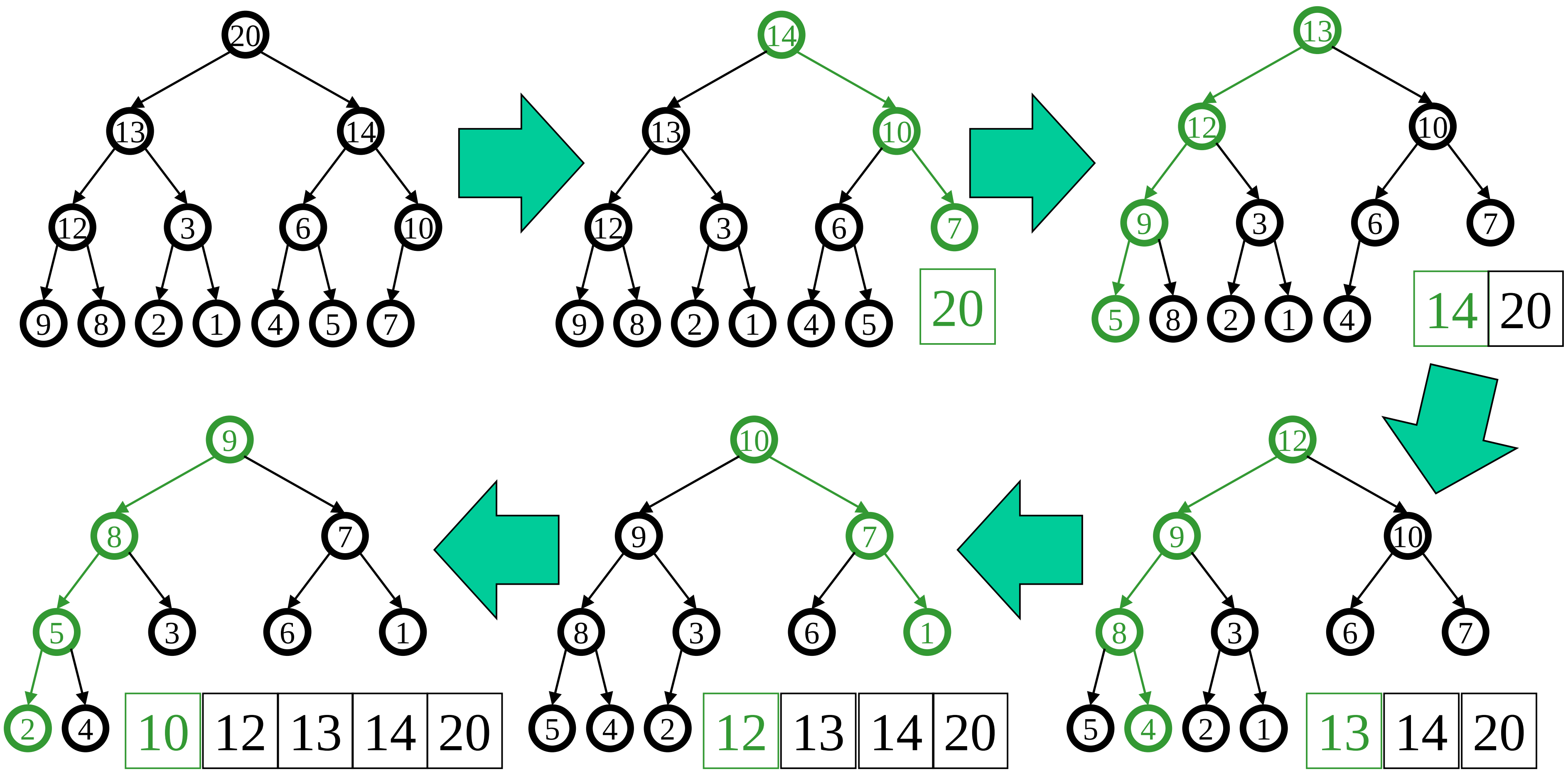
"PQSort" deleteMaxes with Heap MAX-PQ



Build Heap

Note: 9 ends up being perc'd down as well since its invariant is violated by the time we reach it.

"PQSort" deleteMaxes with Heap MAX-PQ



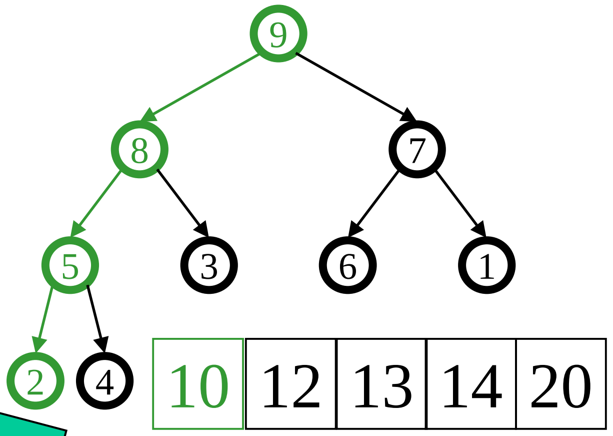
“PQSort” with Heap MAX-PQ

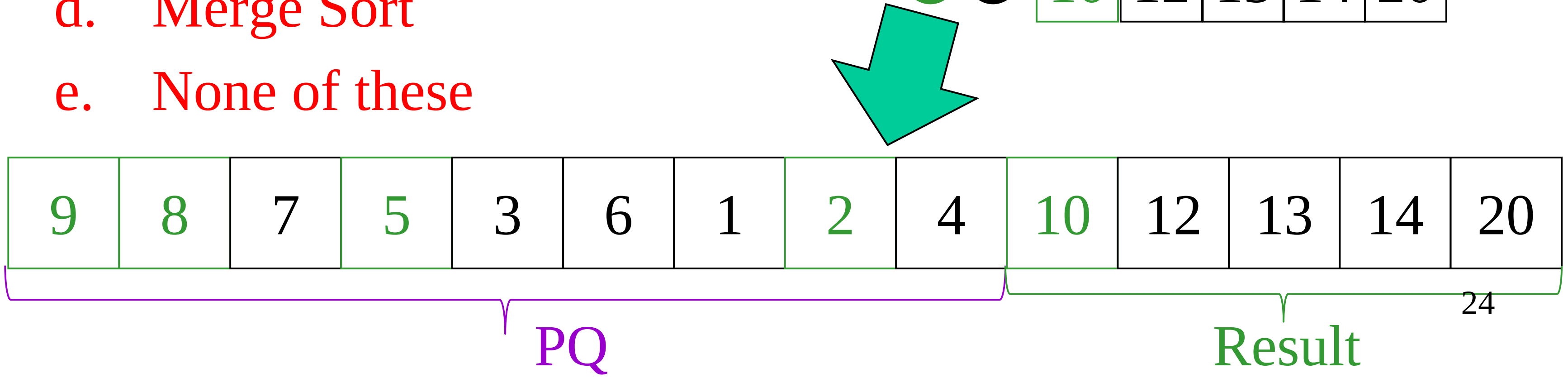
How long does “build” take? Worst case: O(n) ☺

How long do the deletions take? Worst case: O(n lg n) ☺

What algorithm is this?

a. Insertion Sort
b. Selection Sort
c. Heap Sort
d. Merge Sort
e. None of these





“PQSort”

What sorting algorithm is this?

a. Insertion Sort
b. Selection Sort
c. Heap Sort
d. Merge Sort
e. None of these

```
Sort(elts):
  pq = new PQ
  for each elt in elts:
    pq.insert(elt);
  sortedElts = new array of size elts.length
  for i = 0 to elements.length – 1:
    sortedElts[i] = pq.deleteMin
  return sortedElts
```

CS221: Algorithms and Data Structures

Sorting Things Out

(slides stolen from Steve Wolfman with minor tweaks by Alan Hu)

Today’s Outline

- Categorizing/Comparing Sorting Algorithms
  - PQSorts as examples
- MergeSort
- QuickSort
- More Comparisons
- Complexity of Sorting

Categorizing Sorting Algorithms

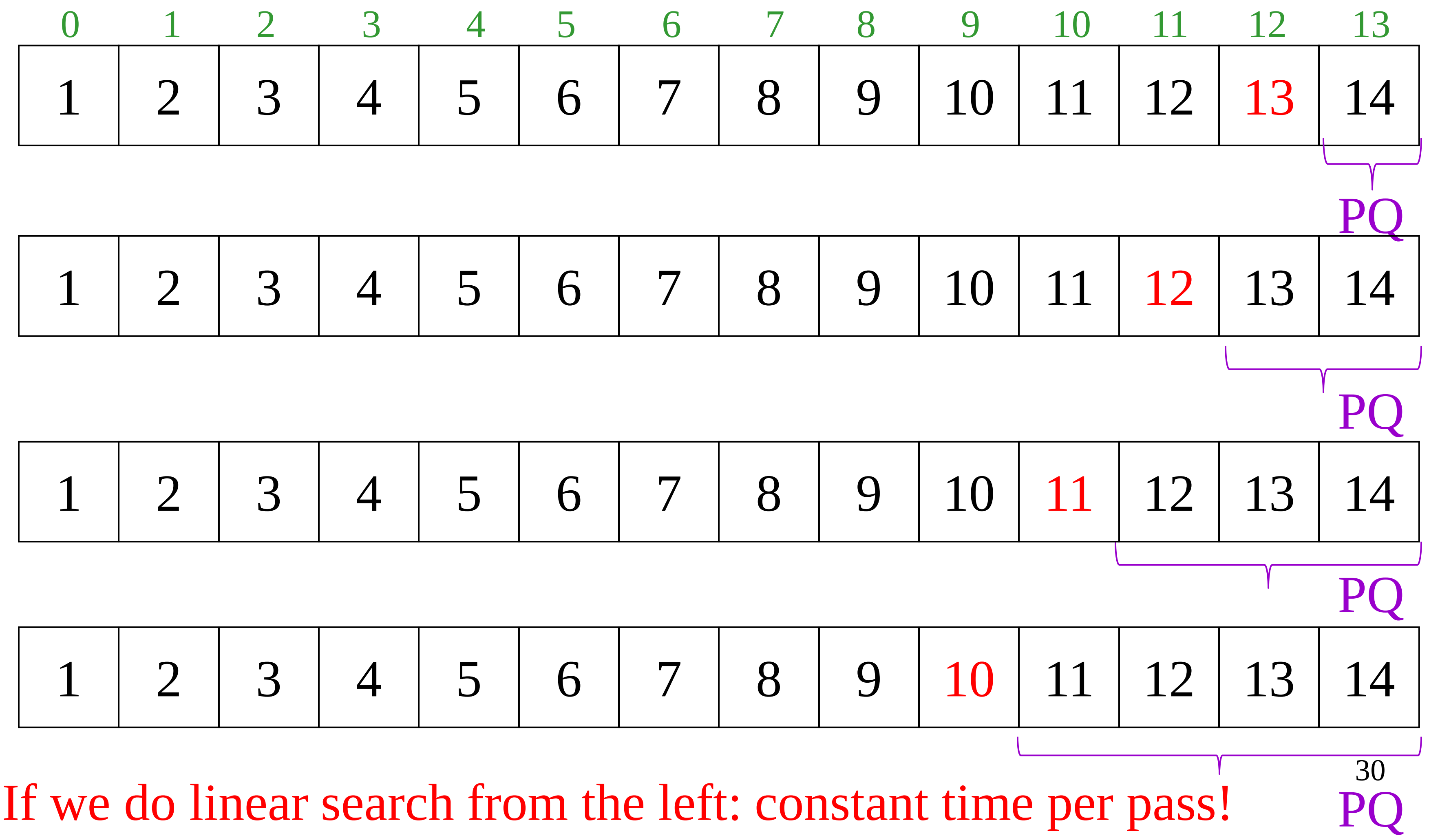
- Computational complexity
  - Average case behaviour: Why do we care?
  - Worst/best case behaviour: Why do we care? How often do we resort sorted, reverse sorted, or “almost” sorted (k swaps from sorted where k << n) lists?
- Stability: What happens to elements with identical keys?
- Memory Usage: How much *extra* memory is used?

Comparing our "PQSort" Algorithms

- Computational complexity
  - Selection Sort: *Always* makes n passes with a "triangular" shape. Best/worst/average case $\Theta(n^2)$
  - Insertion Sort: *Always* makes n passes, but if we're lucky (and do linear search from left), only constant work is needed on each pass. Best case $\Theta(n)$; worst/average case: $\Theta(n^2)$
  - Heap Sort: *Always* makes n passes needing O(lg n) on each pass. Best/worst/average case: $\Theta(n \lg n)$.

Note: best cases assume *distinct* elements.
With identical elements, Heap Sort can get $\Theta(n)$ performance.

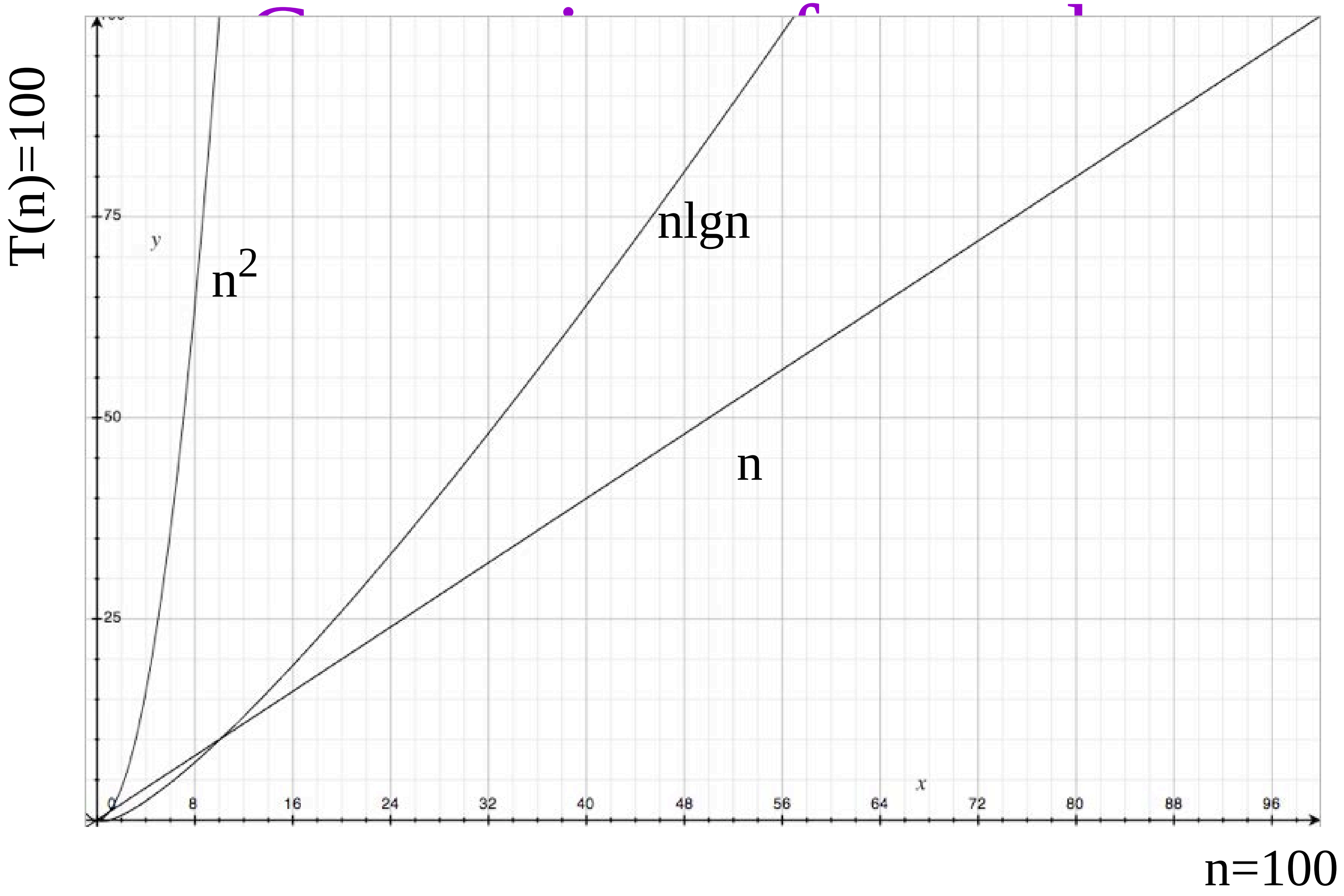
Insertion Sort Best Case



If we do linear search from the left: constant time per pass!

Comparing "PQSort" Algorithms

- Stability
  - Selection: Easily made stable (when building from the right, prefer the rightmost of identical "biggest" keys).
  - Insertion: Easily made stable (when building from the right, find the leftmost slot for a new element).
  - Heap: Unstable ☹
- Memory use: All three are essentially "in-place" algorithms with small O(1) extra space requirements.
- Cache access: Not detailed in 221, but… algorithms that don't "jump around" tend to perform better in modern memory systems. Which of these "jumps around"?


T(n)=100
y
75
50
25
n^2
nlgn
n
x
0
8
16
24
32
40
48
56
64
72
80
88
96
n=100

Today's Outline

- Categorizing/Comparing Sorting Algorithms
  - PQSorts as examples
- MergeSort
- QuickSort
- More Comparisons
- Complexity of Sorting

MergeSort

Mergesort belongs to a class of algorithms known as "divide and conquer" algorithms (your recursion sense should be tingling here...).

The problem space is continually split in half, recursively applying the algorithm to each half until the base case is reached.

MergeSort Algorithm

1. If the array has 0 or 1 elements, it's sorted. Else…
2. Split the array into two halves
3. Sort each half recursively (i.e., using mergesort)
4. Merge the sorted halves to produce one sorted result:
    1. Consider the two halves to be queues.
    2. Repeatedly compare the fronts of the queues. Whichever is smaller (or, if one is empty, whichever is left), dequeue it and insert it into the result.

MergeSort Performance Analysis

1. If the array has 0 or 1 elements, it's sorted. Else… T(1) = 1
2. Split the array into two halves
3. Sort each half recursively (i.e., using mergesort) 2*T(n/2)
4. Merge the sorted halves to produce one sorted result: n
    1. Consider the two halves to be queues.
    2. Repeatedly compare the fronts of the queues. Whichever is smaller (or, if one is empty, whichever is left), dequeue it and insert it into the result.

MergeSort Performance Analysis

$T(1) = 1$

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

$\quad\quad\quad = 4T(n/4) + 2(n/2) + n$

$\quad\quad\quad = 8T(n/8) + 4(n/4) + 2(n/2) + n$

$\quad\quad\quad = 8T(n/8) + n + n + n = 8T(n/8) + 3n$

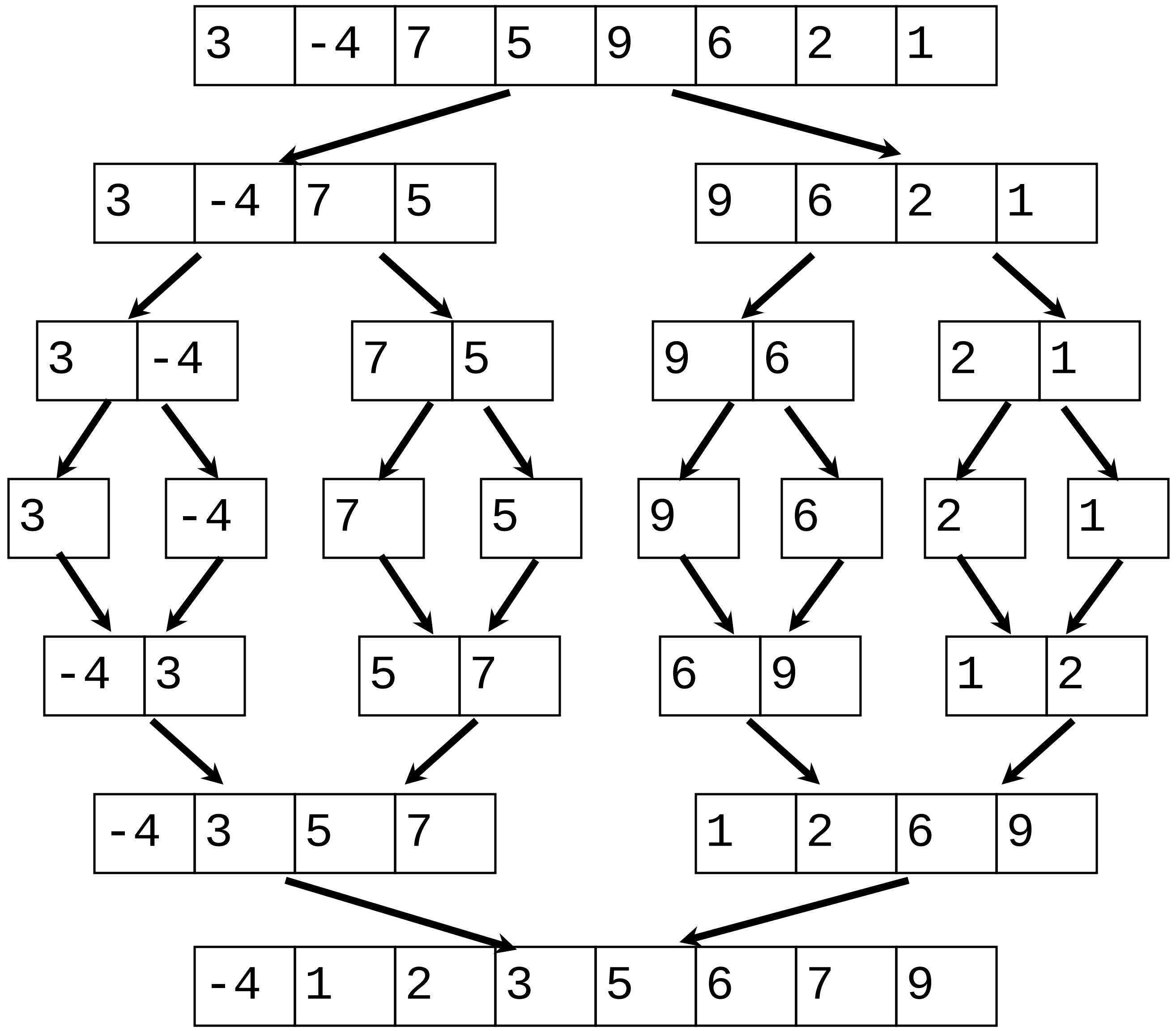
$\quad\quad\quad = 2^i T(n/2^i) + in.$

Let $i = \lg n$

$T(n) \quad = nT(1) + n \lg n = n + n \lg n \in \Theta(n \lg n)$

We ignored floors/ceilings. To prove performance formally, we'd use this as a guess and prove it with floors/ceilings by induction.

Consider the following array of integers:



Mergesort:

```
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:

```
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];
}
```

```
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 );      // will be the final step
```

Today’s Outline

- Categorizing/Comparing Sorting Algorithms
  - PQSorts as examples
- MergeSort
- QuickSort
- More Comparisons
- Complexity of Sorting

QuickSort

In practice, one of the fastest sorting algorithms is Quicksort, developed in 1961 by C.A.R. Hoare.

Comparison-based: examines elements by comparing them to other elements

Divide-and-conquer: divides into “halves” (that may be very unequal) and recursively sorts

QuickSort algorithm

- Pick a pivot
- Reorder the list such that all elements < pivot are on the left, while all elements >= pivot are on the right
- Recursively sort each side

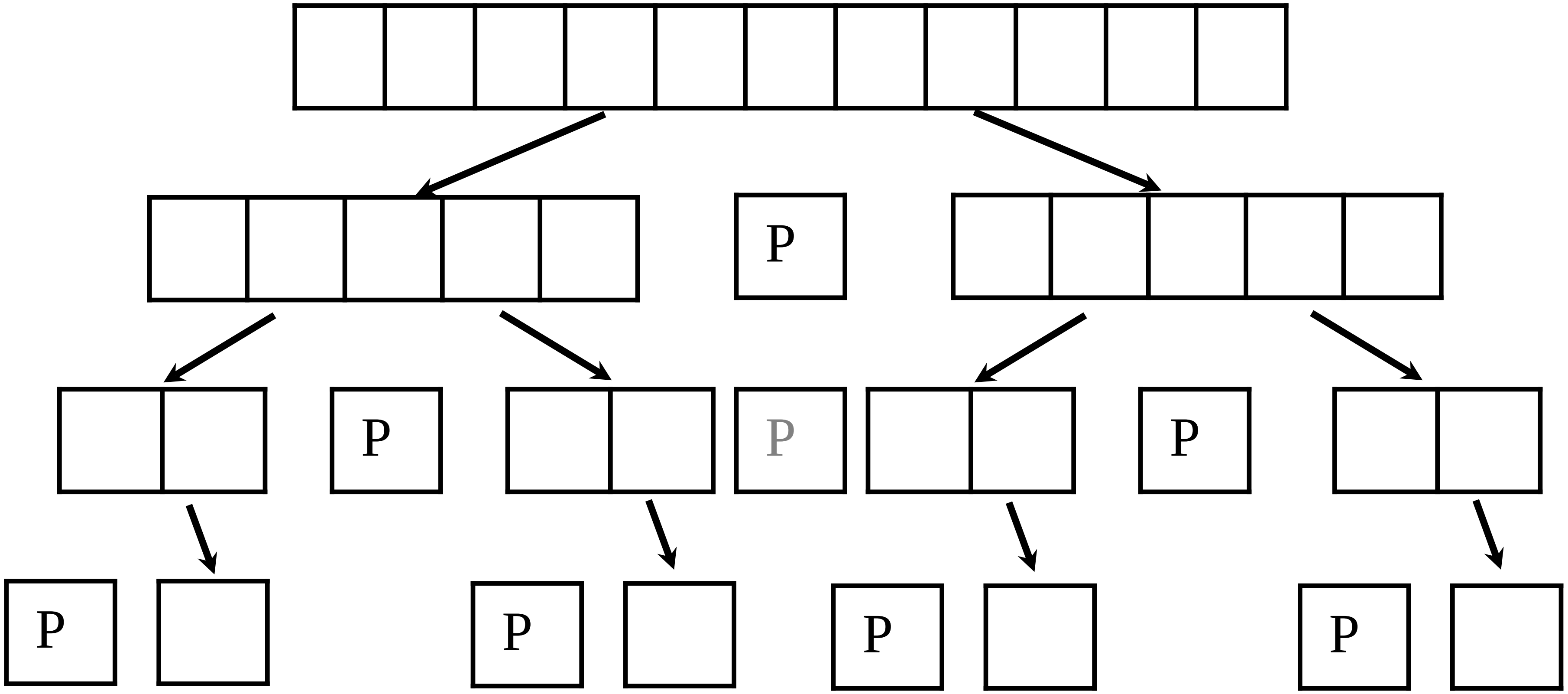
Are we missing a base case?

Partitioning

- The act of splitting up an array according to the pivot is called partitioning
- Consider the following:

$$\underbrace{\mathtt{-4}\quad \mathtt{1}\quad \mathtt{-3}\quad \mathtt{2}}_{\texttt{left partition}}\quad \underset{\text{pivot}}{\mathtt{3}}\quad \underbrace{\mathtt{5}\quad \mathtt{4}\quad \mathtt{7}}_{\texttt{right partition}}$$

QuickSort Visually



Sorted!

QuickSort (by Jon Bentley):

```
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);
}
```

QuickSort (by Jon Bentley): (Loop invariant by Alan!)

```
void qsort(int x[], int lo, int hi)
{
  int i, p;
  if (lo >= hi) return;
  p = lo;
  for( i=lo+1; i <= hi; i++ )
    // x[lo+1..p] contains all elements of
    // x[lo+1..i-1] that are less than x[lo]
    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);
}
```

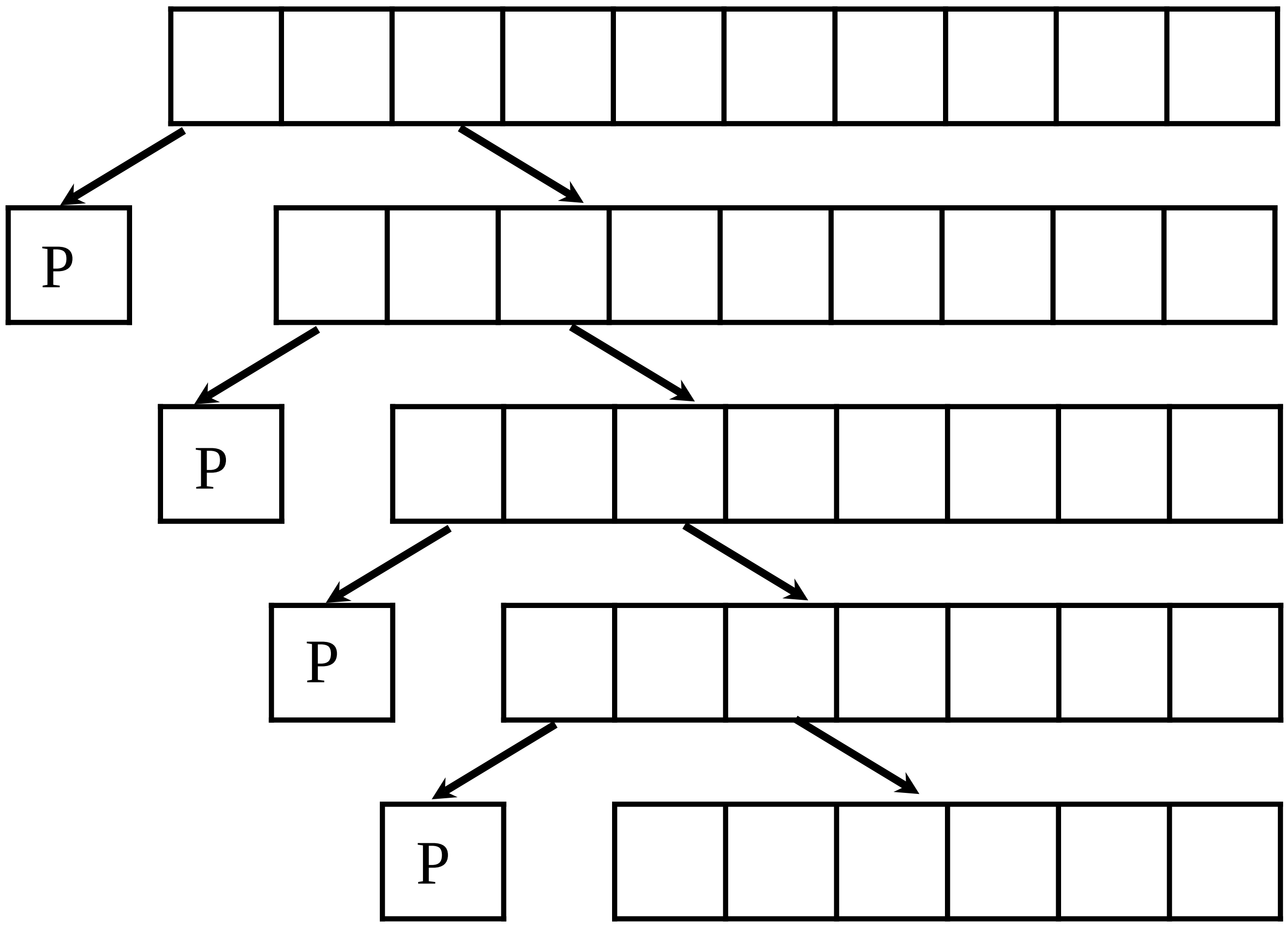
QuickSort Example (using Bentley’s Algorithm)

```
2    -4   6    1    5    -3   3    7
```

QuickSort: Complexity

- In our partitioning task, we compared each element to the pivot
  - Thus, the total number of comparisons is N
  - As with MergeSort, if one of the partitions is about half (or *any* constant fraction of) the size of the array, complexity is Θ(n lg n).
- In the worst case, however, we end up with a partition with a 1 and n-1 split

QuickSort Visually: Worst case



QuickSort: Worst Case

- In the overall worst-case, this happens at every step…
  - Thus we have N comparisons in the first step
  - N-1 comparisons in the second step
  - N-2 comparisons in the third step
  - :

$$\boxed{n + (n-1) + \cdots + 2 + 1 = \frac{n(n+1)}{2} = \frac{n^2}{2} + \frac{n}{2}}$$

- …or O(n$^{2)}$

QuickSort: Average Case (Intuition)

- Clearly pivot choice is important
  - It has a direct impact on the performance of the sort
  - Hence, QuickSort is fragile, or at least “attackable”
- So how do we pick a good pivot?

QuickSort: Average Case (Intuition)

- Let’s assume that pivot choice is random
  - Half the time the pivot will be in the centre half of the array
  - Thus at worst the split will be n/4 and 3n/4

QuickSort: Average Case (Intuition)

- We can apply this to the notion of a good split
  - Every “good” split: 2 partitions of size n/4 and 3n/4
    - Or divides N by 4/3
  - Hence, we make up to log4/3(N) splits
- Expected # of partitions is at most 2 * log4/3(N)
  - O(logN)
- Given N comparisons at each partitioning step, we have Θ(N log N)

Today’s Outline

- Categorizing/Comparing Sorting Algorithms
  - PQSorts as examples
- MergeSort
- QuickSort
- More Comparisons
- Complexity of Sorting

How Do Quick, Merge, Heap, Insertion, and Selection Sort Compare?

Complexity

- Best case: Insert < Quick, Merge, Heap < Select
- Average case: Quick, Merge, Heap < Insert, Select
- Worst case: Merge, Heap < Quick, Insert, Select
- Usually on “real” data: Quick < Merge < Heap < I/S *(**not** asymptotic)*
- On very short lists: quadratic sorts may have an advantage (so, some quick/merge implementations “bottom out” to these as base cases)

Some details depend on implementation!
(E.g., an initial check whether the last elt of the left sublist is less than first of the right can make merge’s best case linear.)

How Do Quick, Merge, Heap, Insertion, and Selection Sort Compare?

Stability

- Easily Made Stable: Insert, Select, Merge (prefer the "left" of the two sorted sublists on ties)
- Unstable: Heap
- Challenging to Make Stable: Quick

- Memory use:
  - Insert, Select, Heap < Quick < Merge

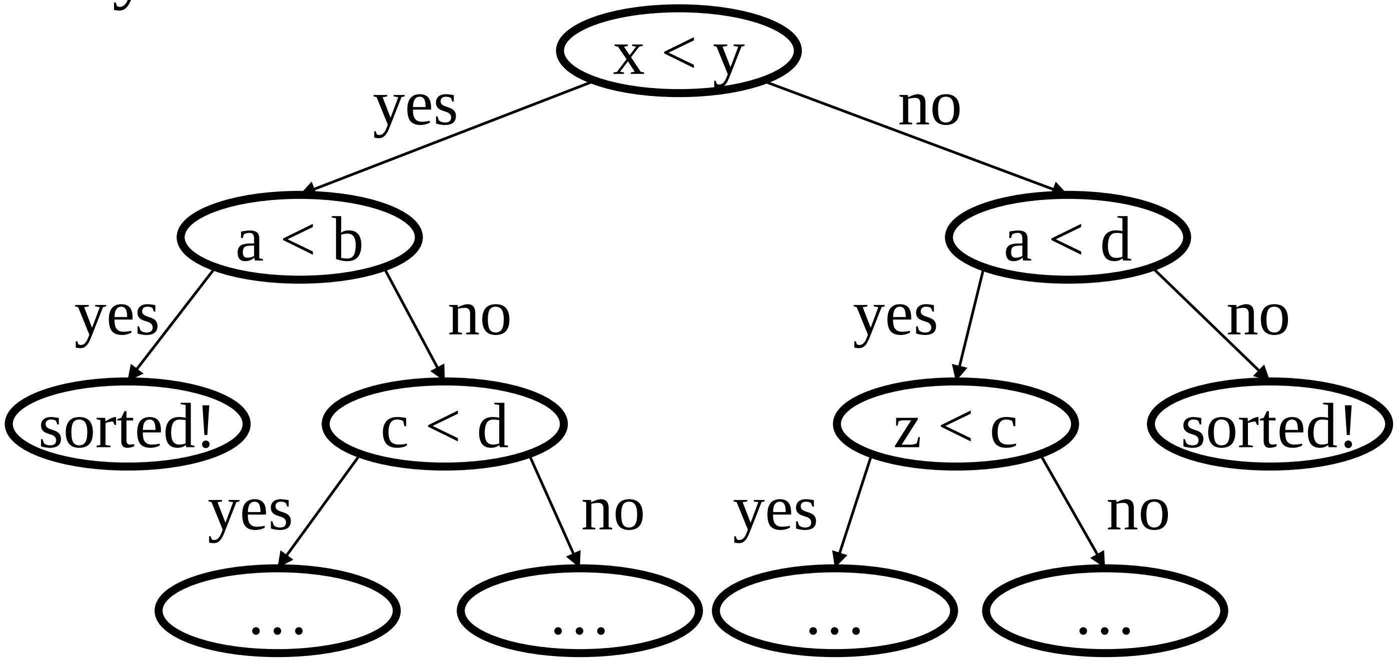
How much stack space does recursive QuickSort use?
In the worst case? Could we make it better?

Today's Outline

- Categorizing/Comparing Sorting Algorithms
  - PQSorts as examples
- MergeSort
- QuickSort
- More Comparisons
- Complexity of Sorting

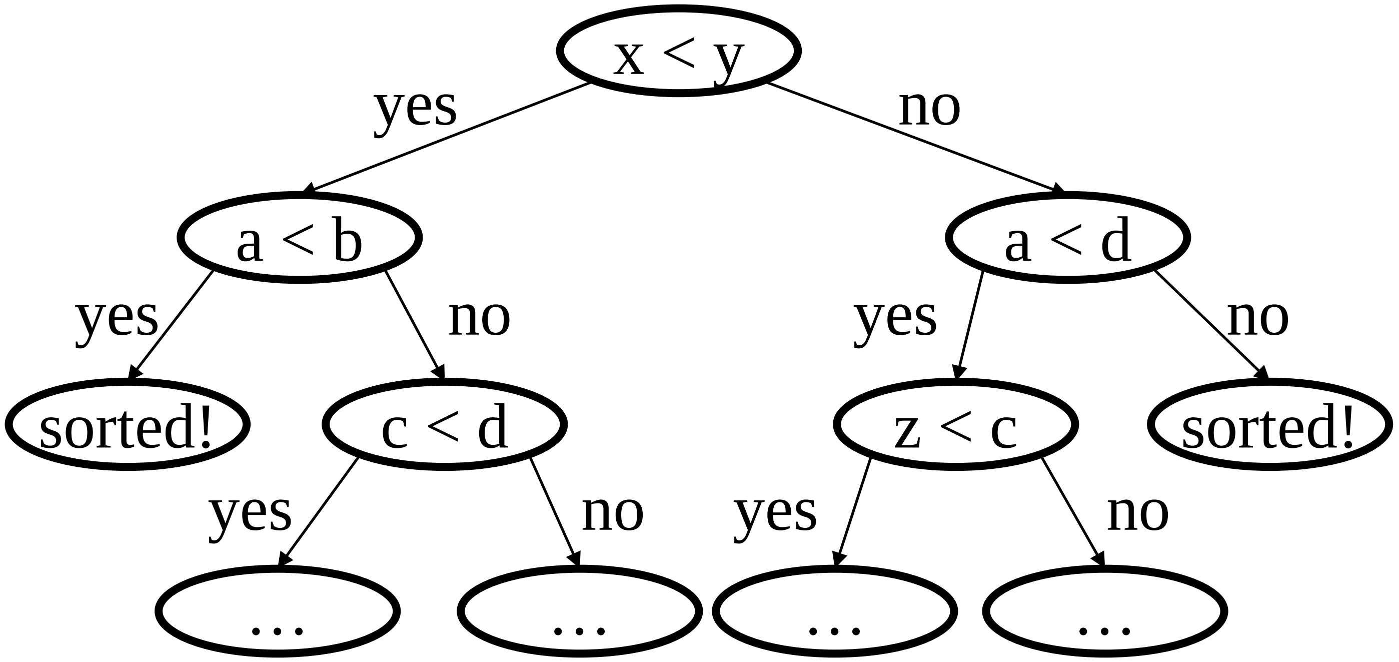
Complexity of Sorting Using Comparisons as a Problem

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



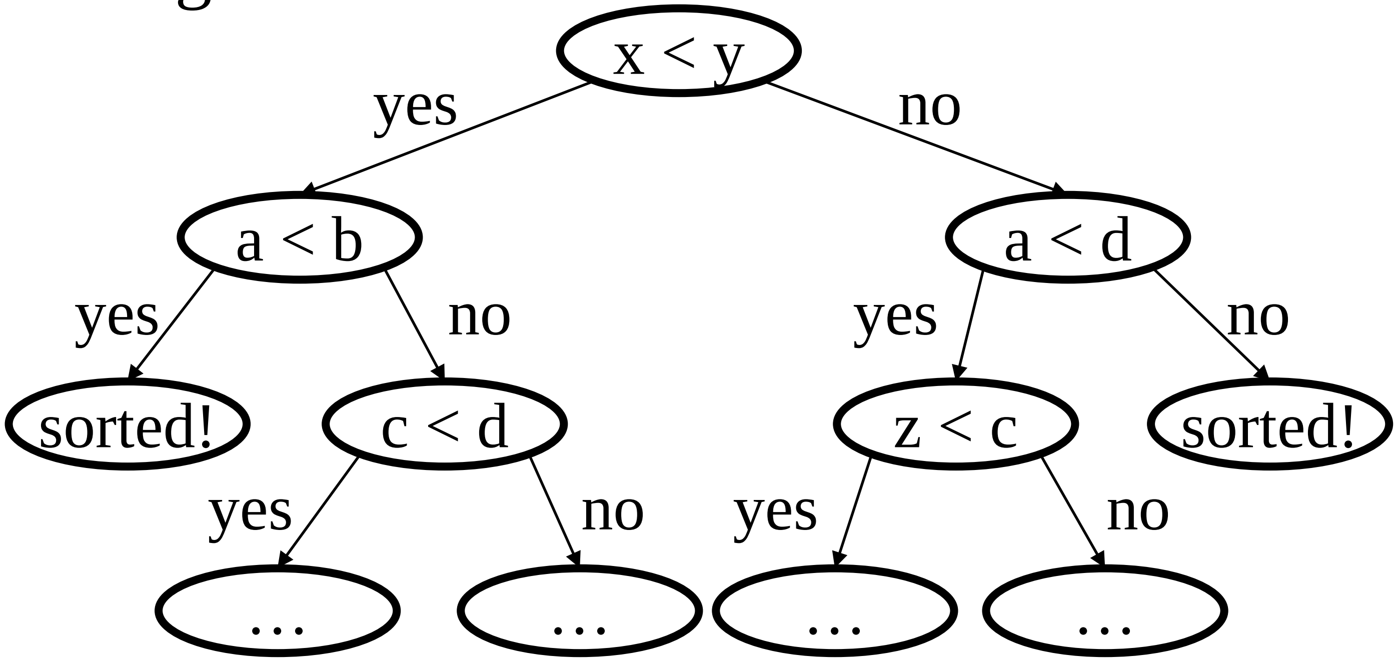
Complexity of Sorting Using Comparisons as a Problem

The algorithm spits out a (possibly different) sorted list at each leaf. What's the maximum number of leaves?



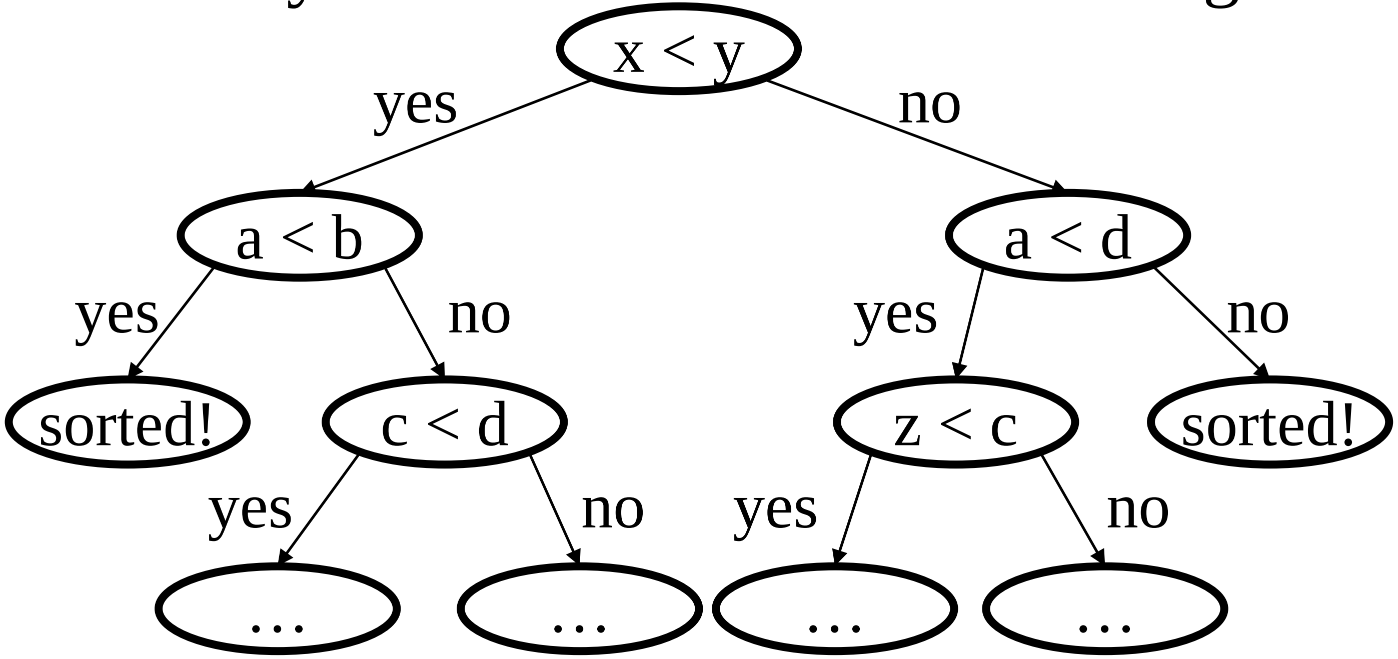
Complexity of Sorting Using Comparisons as a Problem

There are n! possible permutations of a sorted list (i.e., input orders for a given set of input elements). How deep must the tree be to distinguish those input orderings?



Complexity of Sorting Using Comparisons as a Problem

If the tree is *not* at least lg(n!) deep, then there's some pair of orderings I could feed the algorithm which the algorithm does not distinguish. So, it must not successfully sort one of those two orderings.



Complexity of Sorting Using Comparisons as a Problem

QED: The complexity of sorting using comparisons is Ω(n lg n) in the worst case, *regardless of algorithm*!

In general, we can *lower-bound* but not *upper-bound* the complexity of problems.

(Why not? Because I can give as crappy an algorithm as I please to solve any problem.)