CPSC 221
Basic Algorithms and Data Structures

Sorting

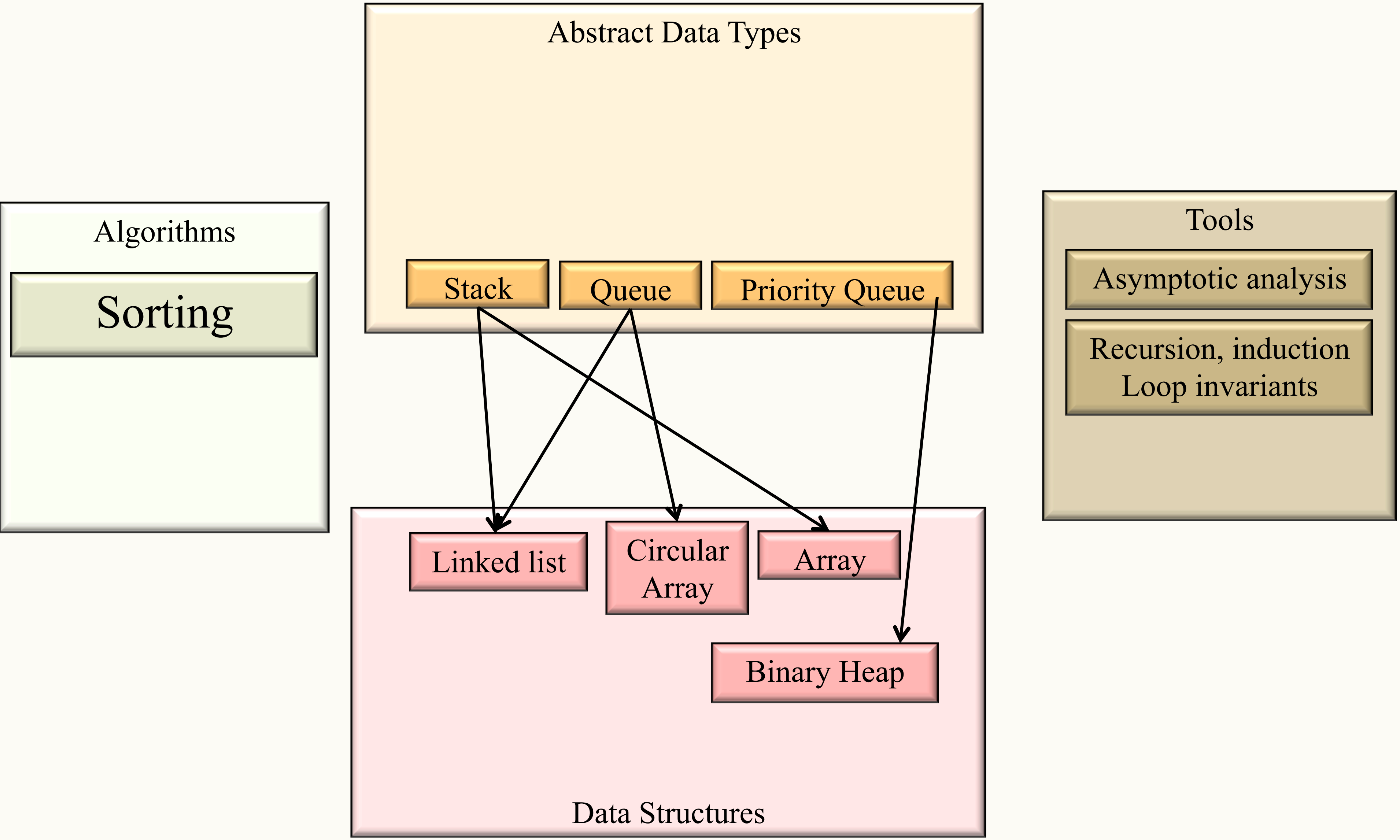
Textbook References:
Koffman: 10.1-10.4, 10.7-10.10
EPP 3rd edition: 9.5
EPP 4th edition: 11.5

Hassan Khosravi
January – April 2015

Learning Goals

- Describe and apply various sorting algorithms:
  - Insertion Sort, Selection Sort, MergeSort, and QuickSort, Bubble Sort, Heapsort
- Compare and contrast the tradeoffs of these algorithms.
- State differences in performance for large files versus small files on various sorting algorithms.
- Analyze the complexity of these sorting algorithms.
- Manipulate data using various sorting algorithms (irrespective of any implementation).

CPSC 221 Journey



Categorizing Sorting Algorithms

- Computational complexity
  - Average case behaviour: Why do we care?
  - Worst/best case behaviour: Why do we care?
- Stability: stable sorting algorithms maintain the relative order of records with equal keys.

Tyrion Lannister	Sort by l name →	Cersei Lannister
Cersei Lannister		Jaime Lannister
Daenerys Targaryen	Break ties with f name	Tyrion Lannister
Jaime Lannister		Daenerys Targaryen

- Memory Usage: How much *extra* memory is used?

Selection Sort

- Sorts an array by repeatedly finding the smallest element of the unsorted tail region and moving it to the front.

	8
	5
	2
	6
	9
	3
	1
	4
	0
	7

Selection Sort

- Find the smallest and swap it with the first element

5	9	17	11	12

- Find the next smallest. It is already in the correct place

5	9	17	11	12

- Find the next smallest and swap it with first element of unsorted portion

5	9	11	17	12

- Repeat

5	9	11	12	17

- When the unsorted portion is of length 1, we are done

5	9	11	12	17

Selection Sort

```
/*
 Purpose: Find the position of the minimum value
          in part of an array
 Param:  data - integer array to be sorted
         from - starting index
         to   - ending index
 returns  - index of minimum value between from and to
 */

int min_position(int data[], int from, int to)
{
    int min_pos = from;
    int i;
    for (i = from + 1; i <= to; i++)
        if (data[i] < data[min_pos])
           min_pos = i;
    return min_pos;
}
```

Selection Sort

```
/*
 Purpose: sorts elements of an array of integers using
          selection sort
 Param:  data - integer array to be sorted
         size - size of the array
 */

void selection_sort(int data[], int size)
{
    int next; // The next position to be set to minimum
    for (next = 0; next < size - 1; next++)
    {
        int min_pos = min_position(data, next, size-1);
        if (min_pos != next)
            swap(data[min_pos], data[next]);
    }
}
```

In-Class Exercise

- Write out all the steps that selection sort takes to sort the following sequence:

91	5	11	90	6	16	31	88

In-Class Exercise

- Write out all of the steps that selection sort takes to sort the following sequence:

91	5	11	90	6	16	31	88
5	91	11	90	6	16	31	88
5	6	11	90	91	16	31	88
5	6	11	90	91	16	31	88
5	6	11	16	91	90	31	88
5	6	11	16	31	90	91	88
5	6	11	16	31	88	91	90
5	6	11	16	31	88	90	91

Clicker Question

- What is the time complexity of selection sort in the best and worst case.
- *A*: $O(n^2)$, $O(n^2)$
- *B*: $O(n)$, $O(n^2)$
- *C*: $O(n \lg n)$, $O(n \lg n)$
- *D*: $O(n \lg n)$, $O(n^2)$
- *E*: $O(n)$, $O(n \lg n)$

Clicker Question (answer)

- What is the time complexity of selection sort in the best and worst case.
- *A: $O(n^2)$, $O(n^2)$*
- *B: $O(n)$, $O(n^2)$*
- *C: $O(n \lg n)$, $O(n \lg n)$*
- *D: $O(n \lg n)$, $O(n^2)$*
- *E: $O(n)$, $O(n \lg n)$*

Clicker Question

- Is selection sort stable?
- *A: Yes*
- *B: No*
- *C: I don't know*

Clicker Question

- Is selection sort stable?
- *A: Yes*
- *B: No*
- *C: I don't know*

90	5	11	90	6	16	2	88

2	5	11	90	6	16	90	88

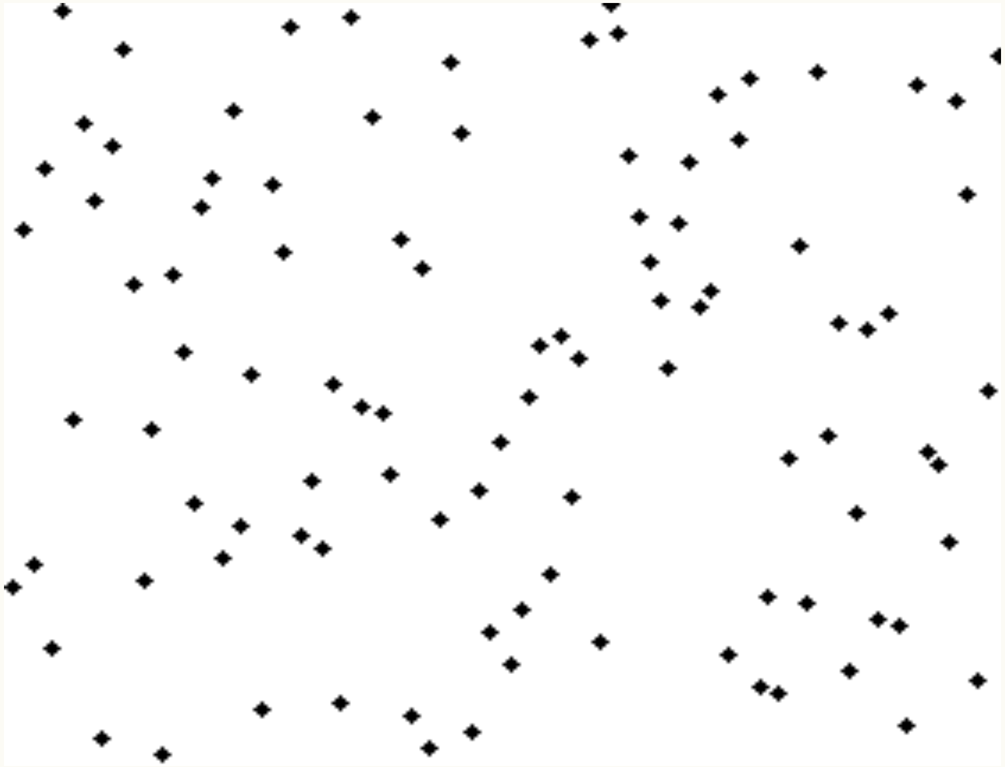
When is the Selection Sort algorithm used?

- One advantage of selection sort is that it requires only *$O(n)$* write operations. If we have a system where write operations are extremely expensive and read operations are not, then selection sort could be ideal. One such scenario would be if we are sorting a file in-place on flash memory or an external hard drive.

Name	Best	Average	Worst	Stable	Memory
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	challenging	$O(1)$

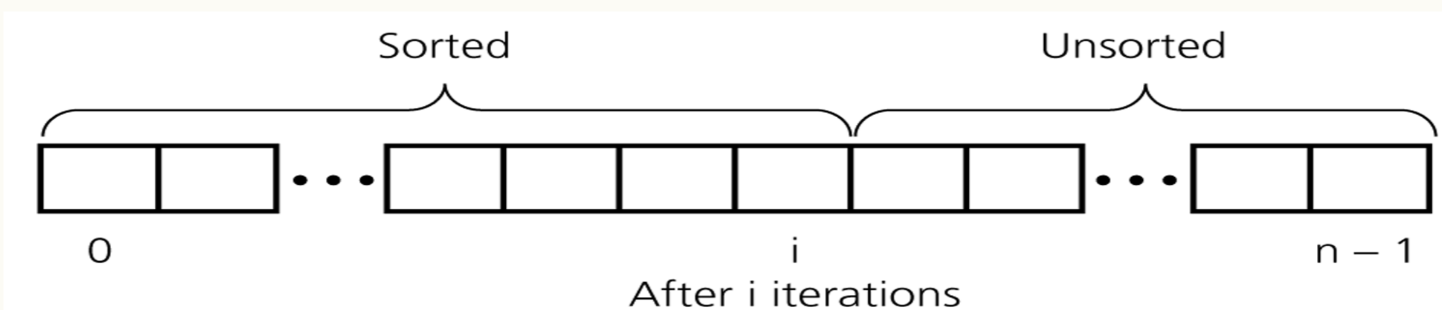
Insertion Sort

- Given a list, take the current element and insert it at the appropriate position of the list, adjusting the list every time you insert



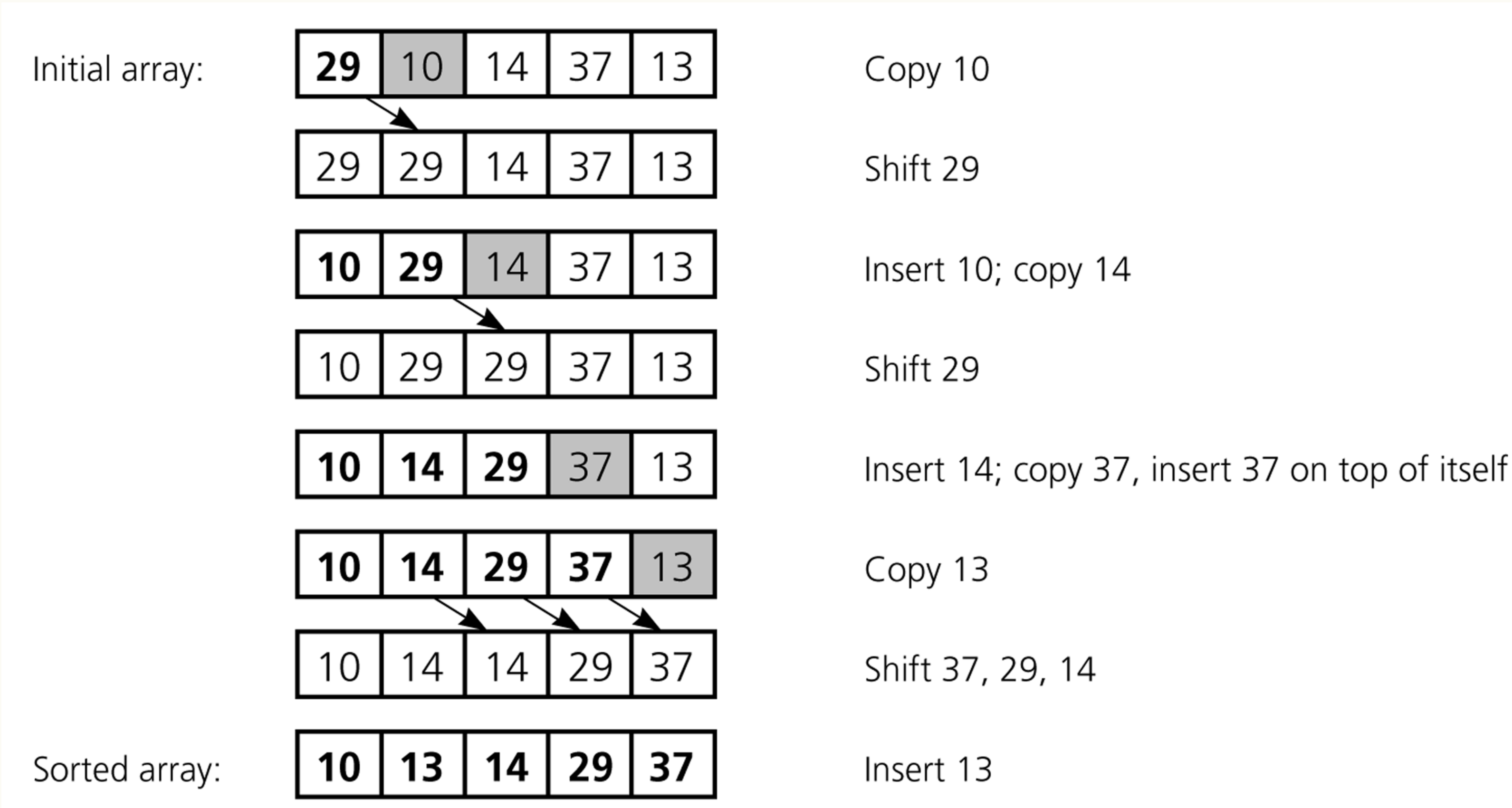
- while some elements unsorted:
  - Using linear search, find the location in the sorted portion where the 1st element of the unsorted portion should be inserted
  - Move all the elements after the insertion location up one position to make space for the new element

the fourth iteration of this loop is shown here



In-class exercise

- Write out all of the steps that insertion sort takes to sort the following sequence: 29 10 14 37 13



Insertion Sort

```
/*
 Purpose: sorts elements of an array of integers using
insertion sort

 Param:  data   - integer array to be sorted
        length - size of the array
 */

void insertion_sort(int data[], int length){
  for (int i = 1; i < length; i++){
    int val = data [i];
    int newIndex = bSearch(data, val, 0, i);
    for (int j = i; j > newIndex; j--)
      data [j] = data [j-1];

    data [newIndex] = val;
  }
}
```

Clicker question

- What is the time complexity of Insertion Sort in the best and worst case.
- *A*: $O(n^2)$, $O(n^2)$
- *B*: $O(n)$, $O(n^2)$
- *C*: $O(n \lg n)$, $O(n \lg n)$
- *D*: $O(n \lg n)$, $O(n^2)$
- *E*: $O(n)$, $O(n \lg n)$

Clicker question (answer)

- What is the time complexity of Insertion Sort in the best and worst case.

a1	a2	a3	a4	a5

B: $O(n)$, $O(n^2)$

- Best case $\sum_{i=1}^{n} 1 = n \in O(n)$

- Worst case $\sum_{i=1}^{n} i = n(n+1)/2 \in O(n^2)$

- Average case $\sum_{i=1}^{n} i/2 = n(n+1)/4 \in O(n^2)$

Clicker Question

- Suppose we are sorting an array of ten integers using a sorting algorithm. After four iterations of the algorithm's main loop, the array elements are ordered as shown here:

1 2 3 4 5 0 6 7 8 9

A. The algorithm might be either selection sort or insertion sort.

B. The algorithm might be selection sort, but could not be insertion sort.

C. The algorithm might be insertion sort, but could not be selection sort.

D. The algorithm is neither selection sort nor insertion sort.

Clicker Question (answer)

- Suppose we are sorting an array of ten integers using a sorting algorithm. After four iterations of the algorithm's main loop, the array elements are ordered as shown here:

1 2 3 4 5 0 6 7 8 9

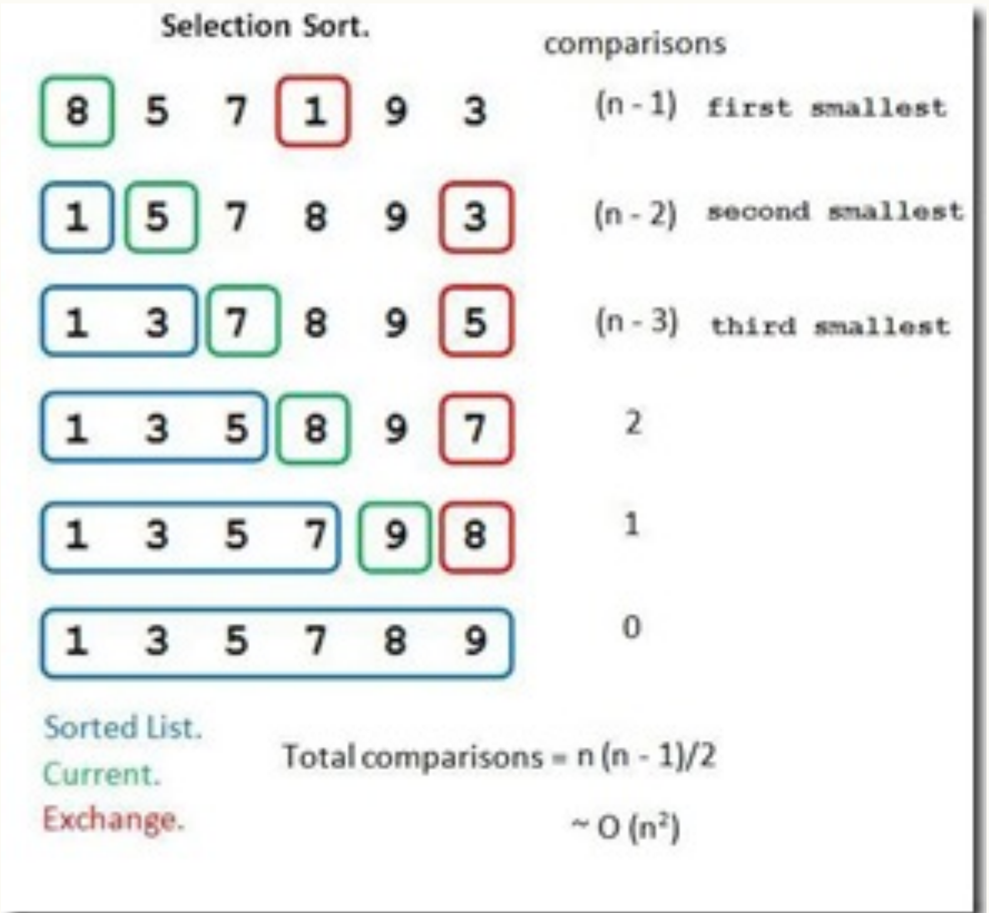
A. The algorithm might be either selection sort or insertion sort.

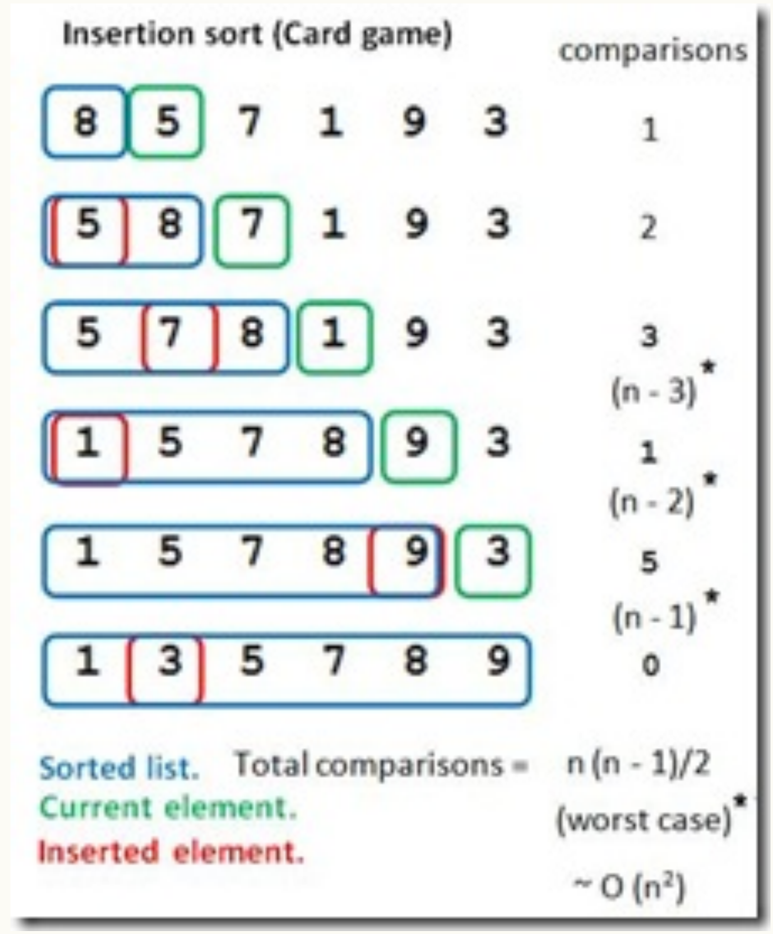
B. The algorithm might be selection sort, but could not be insertion sort.

C. The algorithm might be insertion sort, but could not be selection sort.

D. The algorithm is neither selection sort nor insertion sort.

Selection Sort vs. Insertion Sort





Source

When is the Insertion Sort algorithm used?

- Insertion Sort is the algorithm of choice either when the data is nearly sorted (because it is adaptive) or when the problem size is small (because it has low overhead).

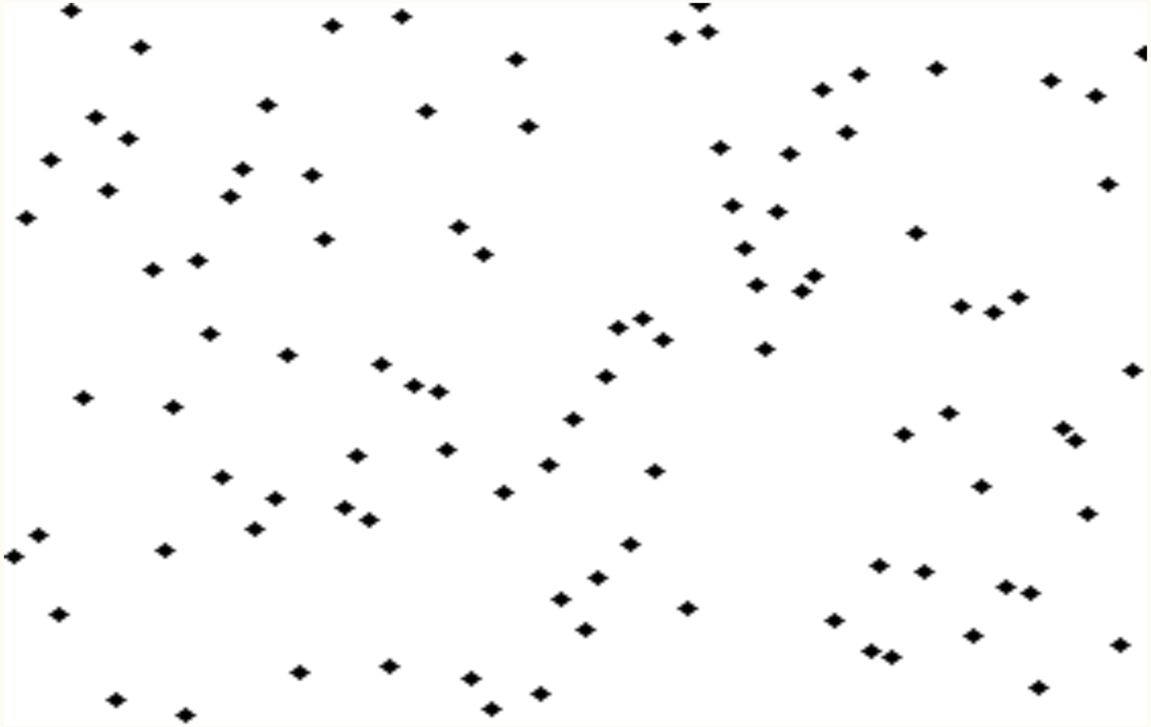
Name	Best	Average	Worst	Stable	Memory
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	challenging	O(1)
Insertion Sort	O(n)	$O(n^2)$	$O(n^2)$	Yes	O(1)

MergeSort

- MergeSort is an example of a divide-and-conquer algorithm that recursively splits the problem into branches, and later combines them to form the solution.
- **Key Steps in MergeSort**:

1. Split the array into two halves.
2. Recursively sort each half.
3. Merge the two (sorted) halves together to produce a bigger, sorted array.
    - Note: The time to merge two sorted sub-arrays of sizes m and n is linear: $O(m + n)$.

Mergesort



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;
}
```

Merging two sorted arrays

- *Divide an array in half and sort each half*

5	9	10	12	17	1	8	11	20	32

- *Merge the two sorted arrays into a single sorted array*

5	9	10	12	17	1	8	11	20	32	1									
5	9	10	12	17	1	8	11	20	32	1	5								
5	9	10	12	17	1	8	11	20	32	1	5	8							
5	9	10	12	17	1	8	11	20	32	1	5	8	9						
5	9	10	12	17	1	8	11	20	32	1	5	8	9	10					
5	9	10	12	17	1	8	11	20	32	1	5	8	9	10	11				
5	9	10	12	17	1	8	11	20	32	1	5	8	9	10	11	12			
5	9	10	12	17	1	8	11	20	32	1	5	8	9	10	11	12	17		
5	9	10	12	17	1	8	11	20	32	1	5	8	9	10	11	12	17	20	
5	9	10	12	17	1	8	11	20	32	1	5	8	9	10	11	12	17	20	32

Merge by Jon Bentley

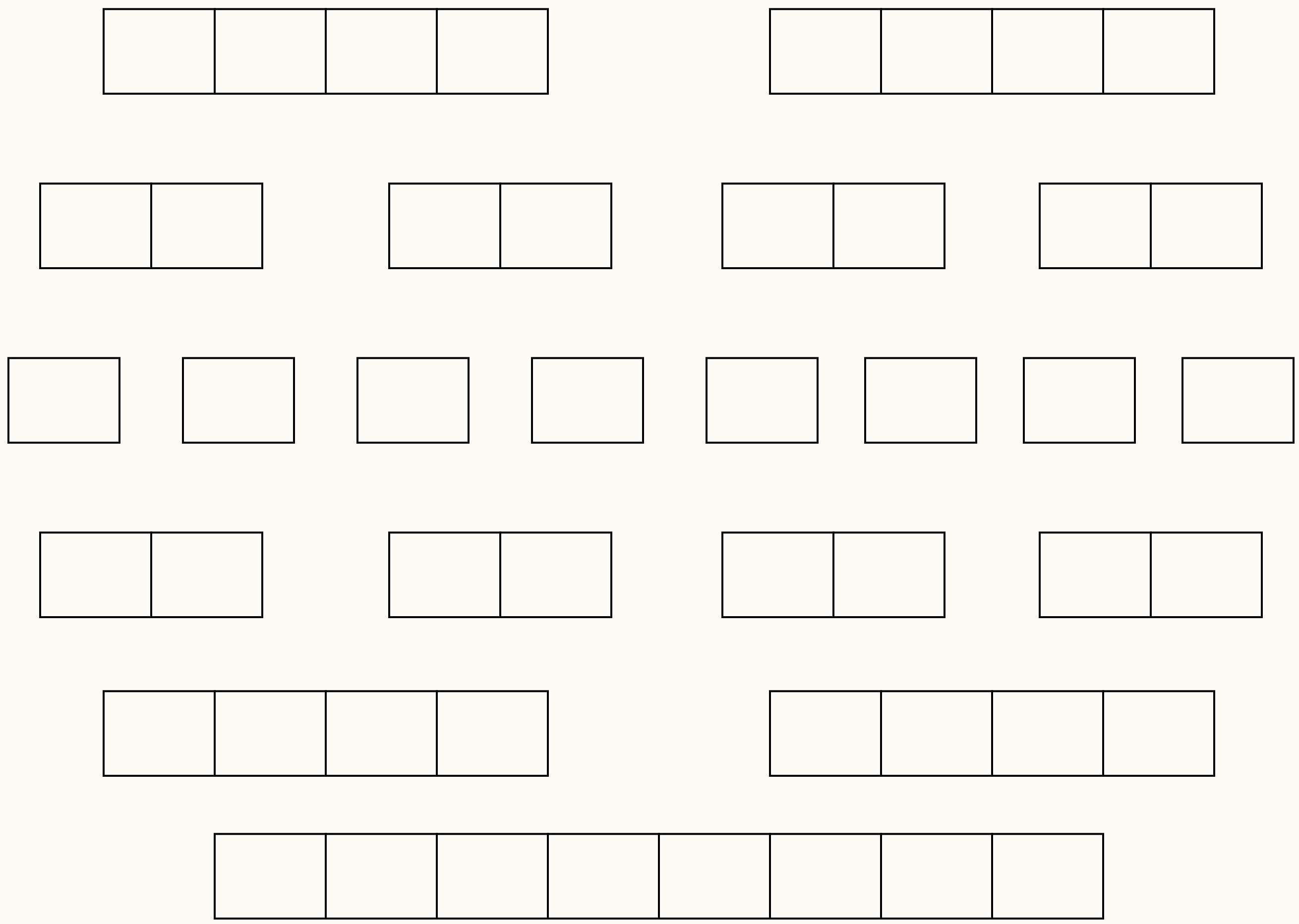
```
/*
 Purpose:  Merges two adjacent ranges in an array

 param      x    - the array with the elements to merge
           low   - the start of the first range
           mid   - the end of the first range
           hi    - end of the second range
           tmp[]- temp memory used for sorting
 */
void merge(int x[], int lo, int mid, int hi, int tmp[]) {
    int a = lo, b = mid+1, k;
    for( k = lo; k <= hi; k++ )
        if( a <= mid && ( b > hi || x[a] < x[b] ) )
            tmp[k] = x[a++];  /* store x[a] then a++ */
        else
            tmp[k] = x[b++]; /* store x[b] then b++ */
    for( k = lo; k <= hi; k++ )
        x[k] = tmp[k];
}
```

Elegant & brilliant…
but not how I'd write it.

In-class exercise

- Write out all the steps that MergeSort takes to sort the following sequence:

3	-4	7	5	9	6	2	1



In-class exercise

- Write out all the steps that MergeSort takes to sort the following sequence:

3	-4	7	5	9	6	2	1

[3 | -4 | 7 | 5]   [9 | 6 | 2 | 1]

[3 | -4]   [7 | 5]   [9 | 6]   [2 | 1]

*

[3]   [-4]   [7]   [5]   [9]   [6]   [2]   [1]

[-4 | 3]   [5 | 7]   [6 | 9]   [1 | 2]

**

[-4 | 3 | 5 | 7]   [1 | 2 | 6 | 9]

[-4 | 1 | 2 | 3 | 5 | 6 | 7 | 9]

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

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 ** in previous slide*/
```

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 );        /* the final step */
```

Clicker question

- MergeSort makes two recursive calls. Which statement is true after these recursive calls finish, but before the merge step?

  A. The array elements form a heap.

  B. Elements in each half of the array are sorted amongst themselves.

  C. Elements in the first half of the array are less than or equal to elements in the second half of the array.

  D. None of the above

Clicker question

- MergeSort makes two recursive calls. Which statement is true after these recursive calls finish, but before the merge step?

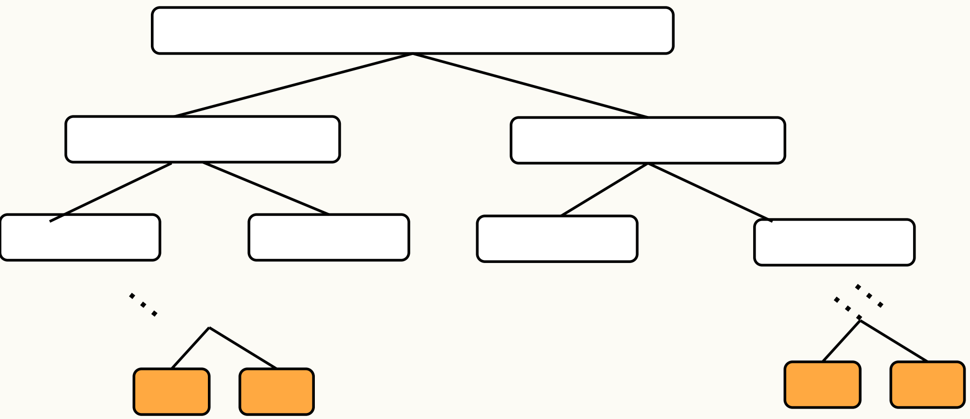
  A. The array elements form a heap.

  B. Elements in each half of the array are sorted amongst themselves.

  C. Elements in the first half of the array are less than or equal to elements in the second half of the array.

  D. None of the above

Analyzing the MergeSort Algorithm



O(n) operations at each level

We have lg n levels therefore,

O(n lg n)

depth	# instances	Size of instances	# read/write operations
0	1	n	n → n
1	2	n/2	2 * n/2 → n
2	4	n/4	4 * n/4 → n
…	…	…	
k	2^k	$n/2^k$	2^k * $n/2^k$ → n
…	…	…	
lg n	$2^{\lg n}$ → n	$n/2^{\lg n}$ → 1	$2^{\lg n}$ *1 → n

MergeSort Performance Analysis

$T(1) = 1$

$$
\begin{aligned}
T(n) &= 2T(n/2) + n \\
&= 4T(n/4) + 2(n/2) + n \\
&= 8T(n/8) + 4(n/4) + 2(n/2) + n \\
&= 8T(n/8) + n + n + n = 8T(n/8) + 3n \\
&= 2^i T(n/2^i) + in.
\end{aligned}
$$

Let $i = \lg n$

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

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

Clicker Question

- Is MergeSort stable?
- A: Yes
- B: No
- C: I don’t know

Clicker Question

- Is MergeSort stable?
- A: Yes
- B: No
- C: I don't know

prefer the "left" of the two sorted sublists on ties

When is the MergeSort algorithm used?

- External sorting is a term for a class of sorting algorithms that can handle massive amounts of data. External sorting is required when the data being sorted do not fit into the main memory of a computing device (usually RAM) and instead they must reside in the slower external memory (usually a hard drive).

- MergeSort is suitable for external sorting.

Name	Best	Average	Worst	Stability	Memory
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	challenging	$O(1)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	Yes	$O(1)$
MergeSort	$O(n \lg n)$	$O(n \lg n)$	$O(n \lg n)$	Yes	$O(n)$

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:

-4 1 -3 2 3 5 4 7

left partition ***pivot*** ***right partition***

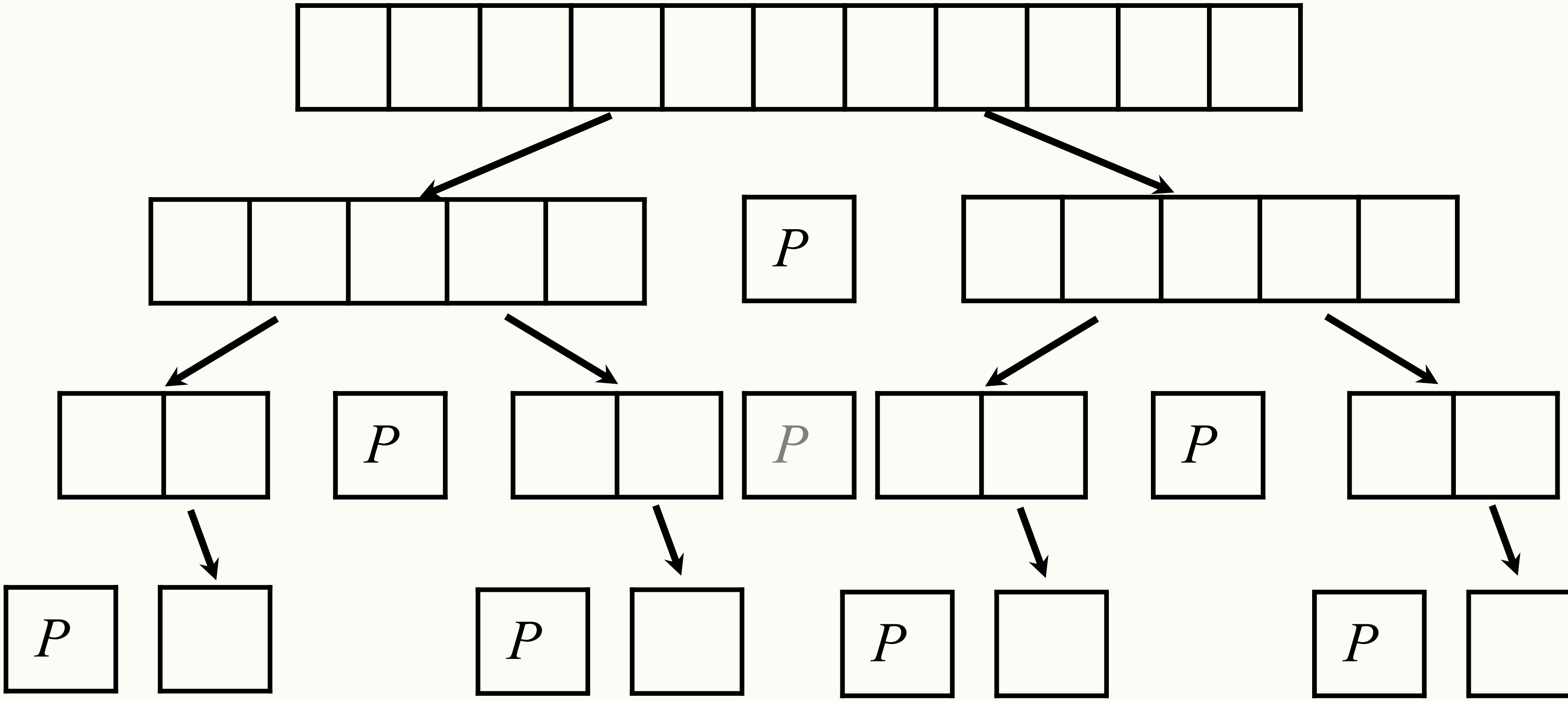
Quicksort example

- http://visualgo.net/sorting.html
  - Initialize array with
    - 25, 44, 38, 5, 47, 15, 36, 26, 27, 2, 46, 4, 19, 50, 48

qSort code

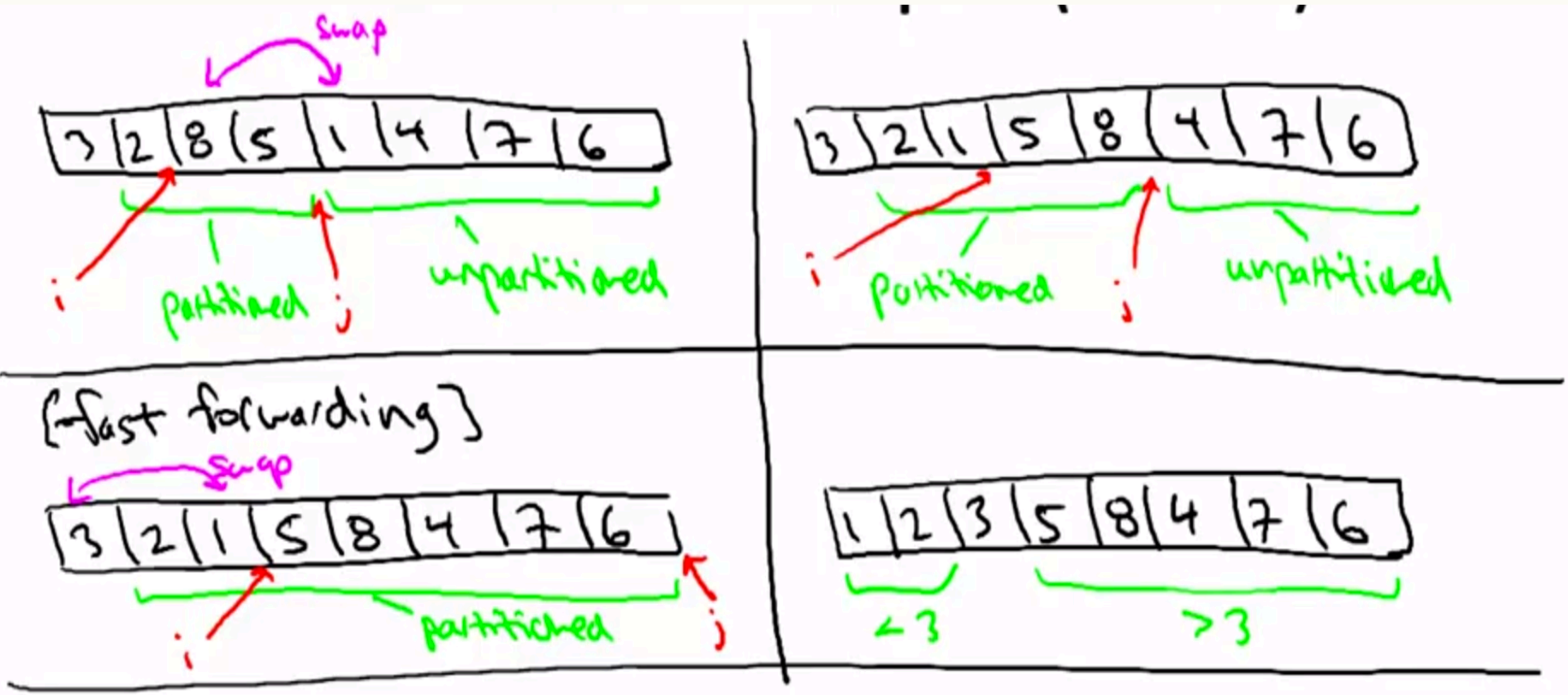
```
/*
 Purpose: sorts elements of an array of integers using Quicksort

 Param:  a  - integer array to be sorted
         lo - the start of the sequence to be sorted.
         hi - the end of the sequence to be sorted.
 */

void qSort( int a[], int lo, int hi ){
    int pivotElement;

    if(lo < hi){
        pivotElement = pivot(a, lo, hi);
        qSort(a, lo, pivotElement-1);
        qSort(a, pivotElement+1, hi);
    }
}
```

QuickSort Visually



Sorted!

Partitioning example



Partitioning

```
/*
 Purpose: find and return the index of pivot element such that all items
          left of partition are smaller and right of partition are bigger

 Param:   x  - integer array to be sorted
          lo - the start of the sequence to be sorted.
          hi - the end of the sequence to be sorted.
*/
int pivot(int x[], int lo, int hi){
    int  p = lo;
    int pivotElement = x[lo];

    for(int i = lo+1 ; i <= hi; i++){
        if(x[i] <= pivotElement){
            p++;
            swap(x[i], x[p]);
        }
    }
    swap(x[p], x[lo]);
    return p;
}
```

QuickSort Example

2 -4 6 1 5 -3 3 7

Clicker question

- Here is an array which has just been partitioned by the first step of QuickSort:

  3, 0, 2, 4, 5, 8, 7, 6, 9

Which of these elements could be the pivot?

- a. 3
- b. 4
- c. 5
- d. 6
- e. (b) or (c)

Clicker question (answer)

- Here is an array which has just been partitioned by the first step of QuickSort:

  3, 0, 2, 4, 5, 8, 7, 6, 9

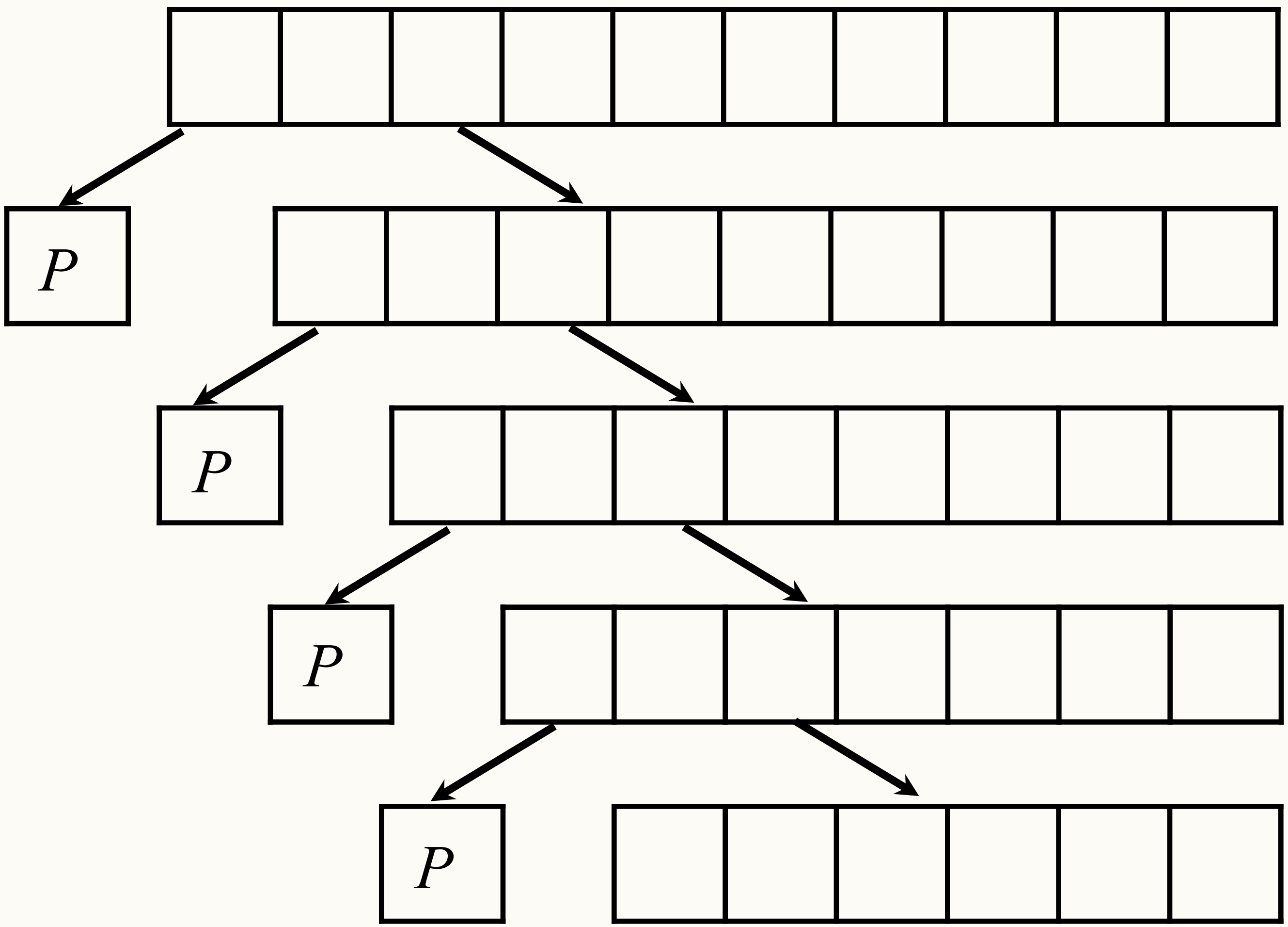
Which of these elements could be the pivot?

- a. 3
- b. 4
- c. 5
- d. 6
- e. (b) or (c)

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 $\Theta(n \lg n)$.
- 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
  - :

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

…or $O(n^2)$

MergeSort vs. QuickSort

- QuickSort is also O(n lg n). But in practice, it tends to run faster than MergeSort. It can work in-place on the original array and does not require extra space
- But: its worst-case complexity is $O(n^2)$.
- That worst-case behaviour can usually be avoided by using more clever ways of finding the pivot (not just using the first element).
  - Randomized algorithms can be used to prove that the average case for Quicksort is *O(n lg n)*

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)

Comparison of different sorting algorithms

- Quicksort algorithm is one of the best sorting algorithms and is widely used

Name	Best	Average	Worst	Stability	Memory
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	challenging	$O(1)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	Yes	$O(1)$
MergeSort	$O(n \lg n)$	$O(n \lg n)$	$O(n \lg n)$	Yes	$O(n)$
QuickSort	$O(n \lg n)$	$O(n \lg n)$	$O(n^2)$	Challenging	$O(\lg n)$

Bubble Sort

- **Bubble sort**, works by repeatedly comparing each pair of adjacent items and swapping them if they are in the wrong order.

In-class exercise

Write out all the steps that bubble sort takes to sort the following sequence: (**5 1** 4 2 8)

First Pass:

(**5 1** 4 2 8) → (**1 5** 4 2 8), Swap since 5 > 1
(1 **5 4** 2 8) → (1 **4 5** 2 8), Swap since 5 > 4
(1 4 **5 2** 8) → (1 4 **2 5** 8), Swap since 5 > 2
(1 4 2 **5 8**) → (1 4 2 **5 8**),

Second Pass:

(**1 4** 2 5 8) → (**1 4** 2 5 8)
(1 **4 2** 5 8) → (1 **2 4** 5 8), Swap since 4 > 2
(1 2 **4 5** 8) → (1 2 **4 5** 8)

Third Pass:

(**1 2** 4 5 8) → (**1 2** 4 5 8)
(1 **2 4** 5 8) → (1 **2 4** 5 8)

Fourth Pass: /* could be avoided */

(1 **2 4** 5 8) → (1 **2 4** 5 8)

Bubble Sort

- Consider the following implementation for

```
/*
 Purpose: sorts elements of an array of integers using bubble sort

 Param:  x - integer array to be sorted
 n - size of the array
 */
void bubbleSort(int x[], int n){
  int i, j, flag = 1;    // set flag to 1 to start first pass
  for(i = 1; (i <= n) && flag; i++){
    flag = 0;
    for (j=0; j < (n -1); j++){
      if (x[j+1] < x[j]) {
        swap(x[j], x[j+1]);
        flag = 1; // indicates that a swap occurred.
      }
    }
  }
  return;
}
```

Clicker question

- What is the time complexity of Bubble Sort in the best and worst case.
- *A*: $O(n^2)$, $O(n^2)$
- *B*: $O(n)$, $O(n^2)$
- *C*: $O(n \lg n)$, $O(n \lg n)$
- *D*: $O(n \lg n)$, $O(n^2)$
- *E*: $O(n)$, $O(n \lg n)$

Clicker question (answer)

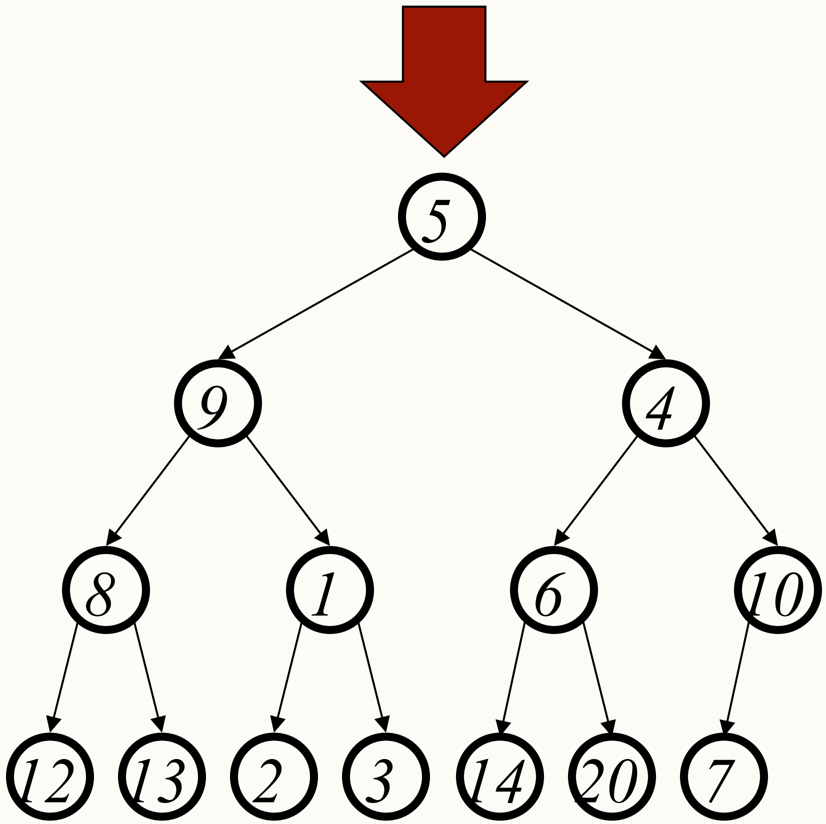
- What is the time complexity of Bubble Sort in the best and worst case.
- *A*: $O(n^2)$, $O(n^2)$
- *B*: $O(n)$, $O(n^2)$
- *C*: $O(n \lg n)$, $O(n \lg n)$
- *D*: $O(n \lg n)$, $O(n^2)$
- *E*: $O(n)$, $O(n \lg n)$

Comparison of different sorting algorithms

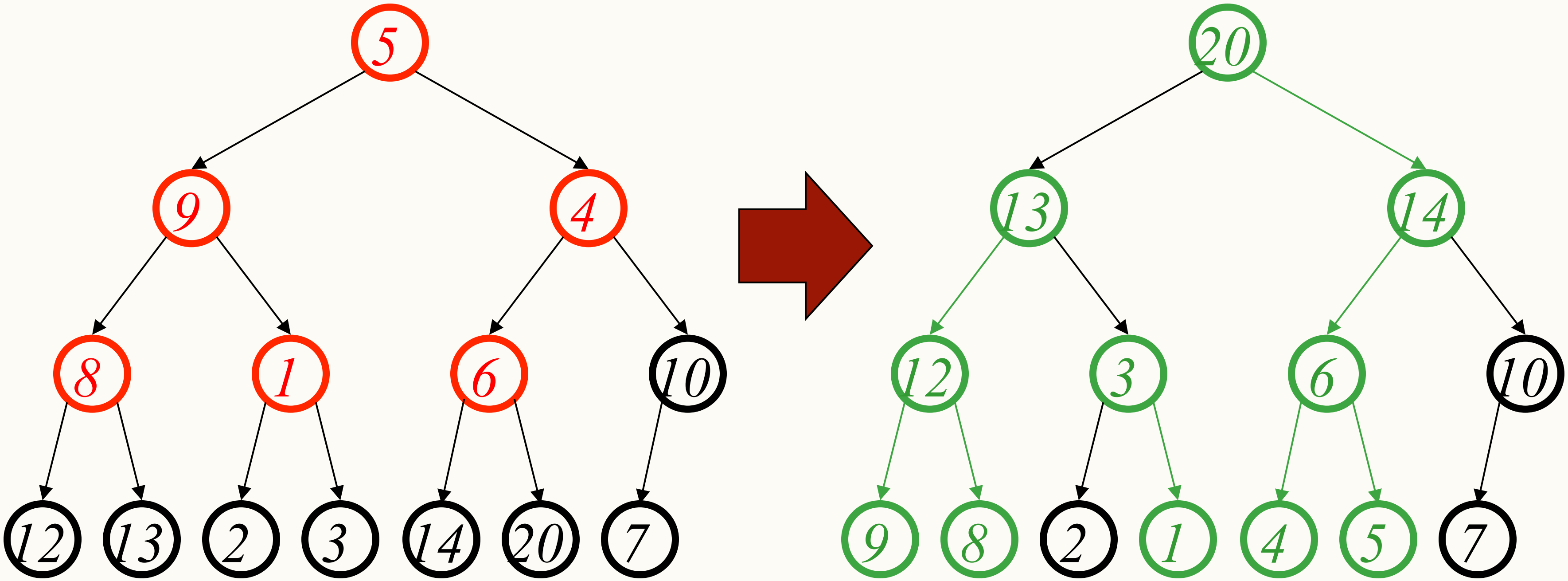
Name	Best	Average	Worst	Stability	Memory
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	Challenging	$O(1)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	Yes	$O(1)$
MergeSort	$O(n \lg n)$	$O(n \lg n)$	$O(n \lg n)$	Yes	$O(n)$
QuickSort	$O(n \lg n)$	$O(n \lg n)$	$O(n^2)$	Challenging	$O(\lg n)$
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	Yes	$O(1)$

Heapsort (revisited)

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

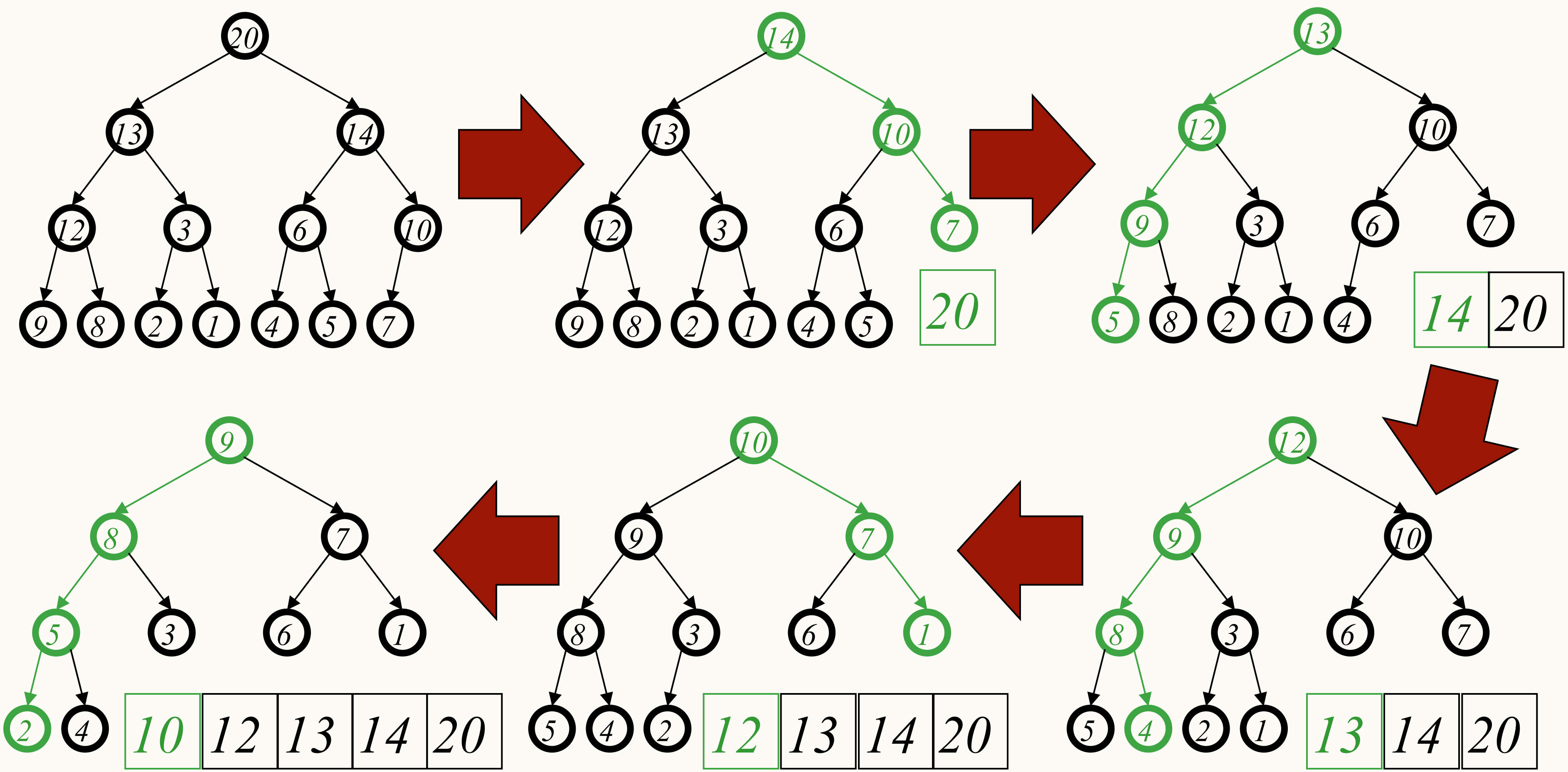


Heapsort (revisited)



Build Heap

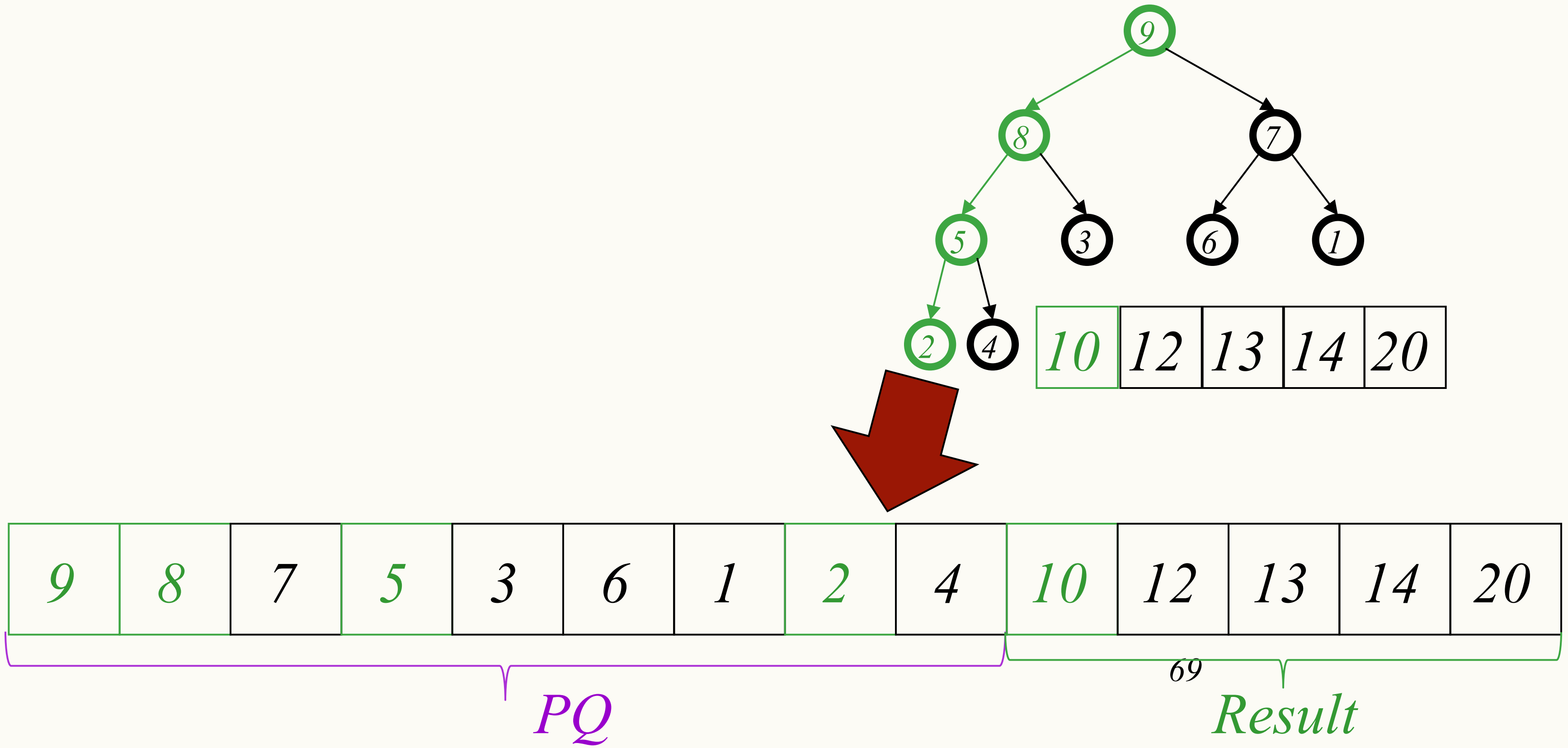
Heapsort (revisited)



Heapsort (revisited)

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

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



Heapsort (revisited)

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

PQ

Takes only O(n) time!

Heapsort (revisited)

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

Comparison of different sorting algorithms

Name	Best	Average	Worst	Stability	Memory
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	Challenging	$O(1)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	Yes	$O(1)$
MergeSort	$O(n \lg n)$	$O(n \lg n)$	$O(n \lg n)$	Yes	$O(n)$
QuickSort	$O(n \lg n)$	$O(n \lg n)$	$O(n^2)$	Challenging	$O(\lg n)$
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	Yes	$O(1)$
Heapsort	$O(n \lg n)$	$O(n \lg n)$	$O(n \lg n)$	No	$O(1)$

Average Case Running Time

~7 min

n	Insertion	Heap	Merge	Quick
100,000	26.86s	0.06s	0.03s	0.03s
200,000	108.05s	0.11s	0.08s	0.06s
400,000	437.27s	0.30s	0.17s	0.14s
800,000	?	0.70s	0.34s	0.31s
1,600,000		1.66s	0.72s	0.66s
3,200,000		3.83s	1.52s	1.38s
6,400,000		8.81s	3.08s	2.88s

- How long would it take the insertion sort algorithm to sort 800,000 values

A: 14 minutes B: 28 minutes C: 56 minutes D: other

Average Case Running Time

~7 min

n	Insertion	Heap	Merge	Quick
100,000	26.86s	0.06s	0.03s	0.03s
200,000	108.05s	0.11s	0.08s	0.06s
400,000	437.27s	0.30s	0.17s	0.14s
800,000	?	0.70s	0.34s	0.31s
1,600,000		1.66s	0.72s	0.66s
3,200,000		3.83s	1.52s	1.38s
6,400,000		8.81s	3.08s	2.88s

- How long would it take the insertion sort algorithm to sort 800,000 values
- $T(n) = n^2 \rightarrow \quad T(2n) = 4n^2$ **B: 28 minutes**

Comparison of different sorting algorithms

- 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
- On very short lists: quadratic sorts may have an advantage (so, some quick/merge implementations "bottom out" to these as base cases)

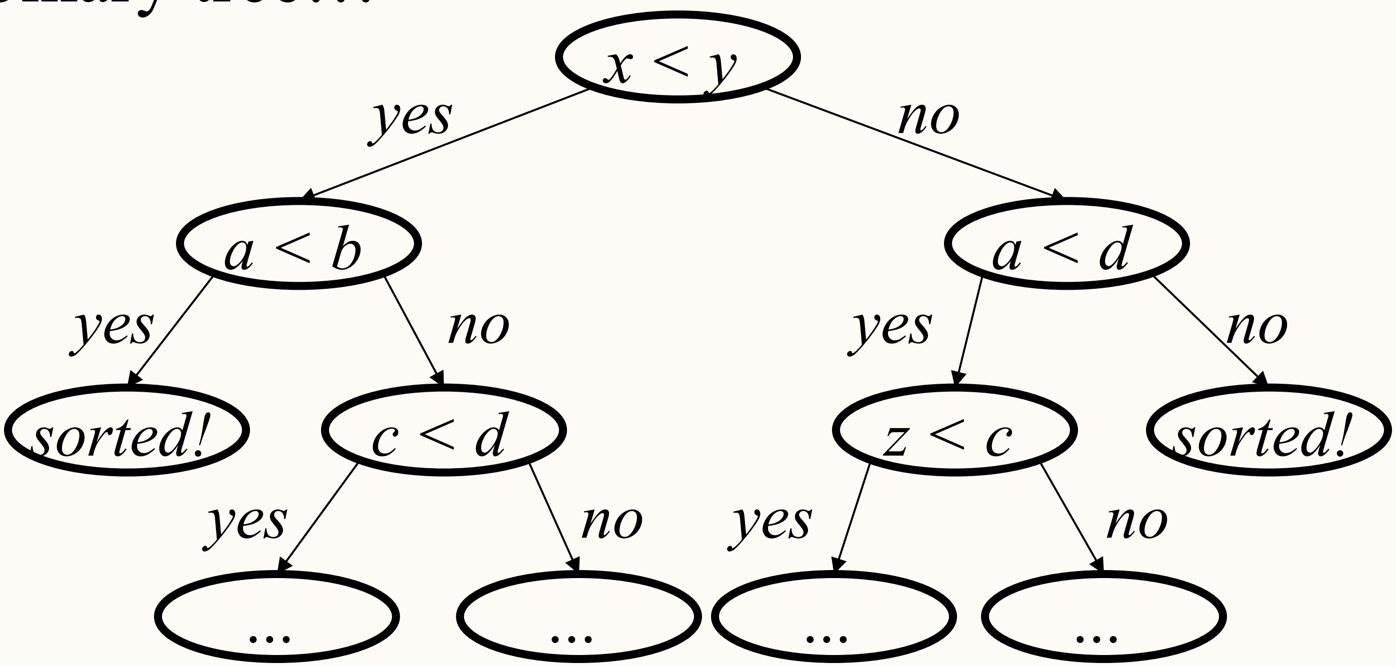
Comparison of different sorting algorithms

- Stability
  - Easily Made Stable: Insert, Merge
  - Challenging to Make Stable: Select, Quick
  - Unstable: Heap

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

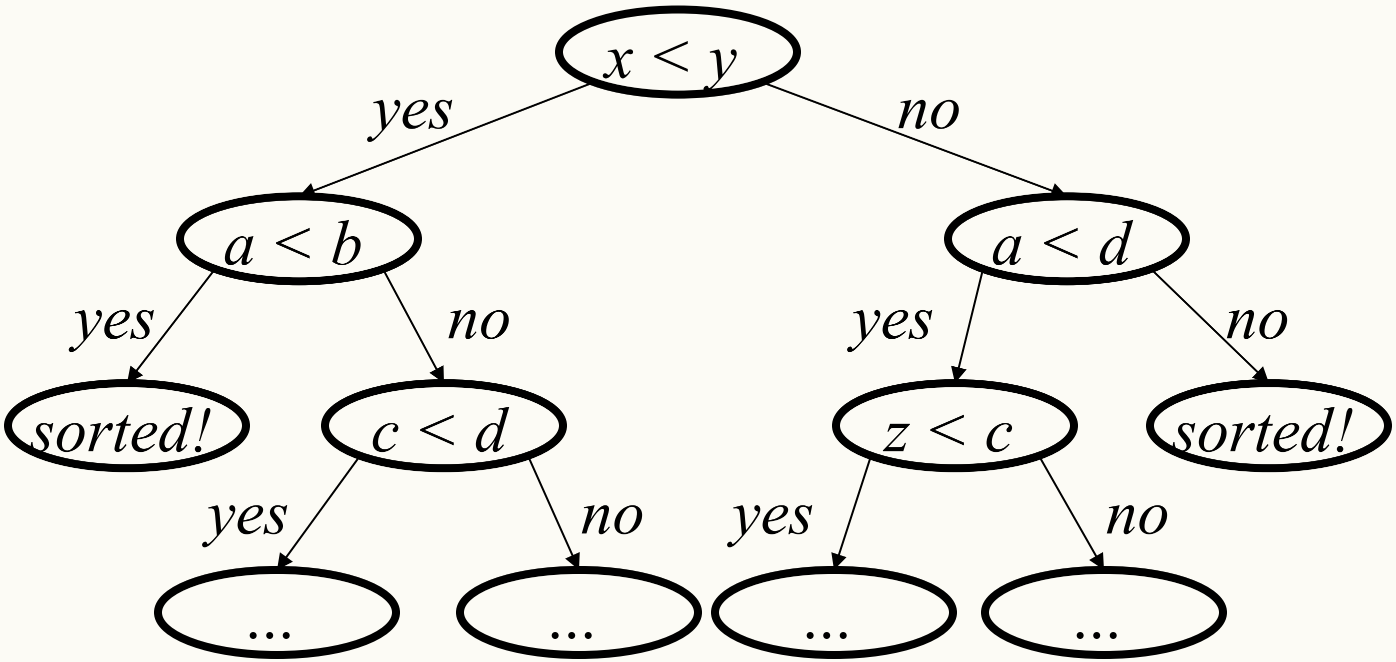
Complexity of Sorting Using Comparisons as a Problem

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…



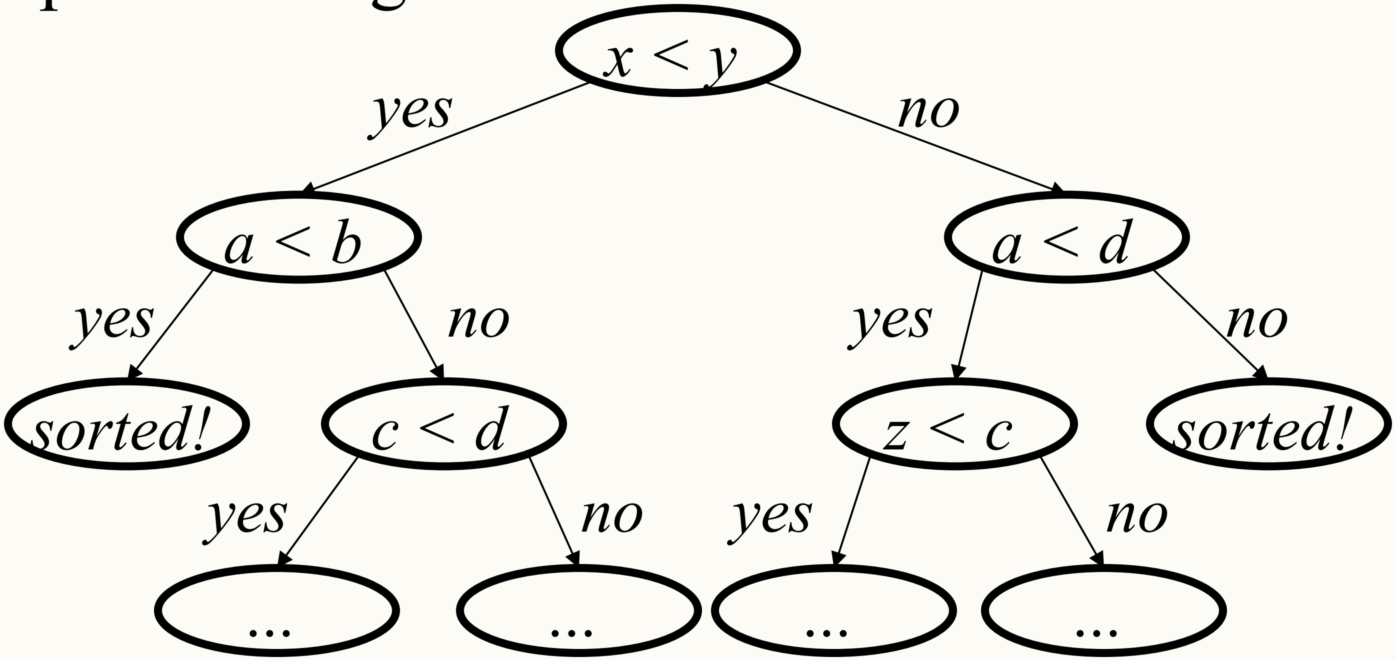
Complexity of Sorting Using Comparisons as a Problem

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



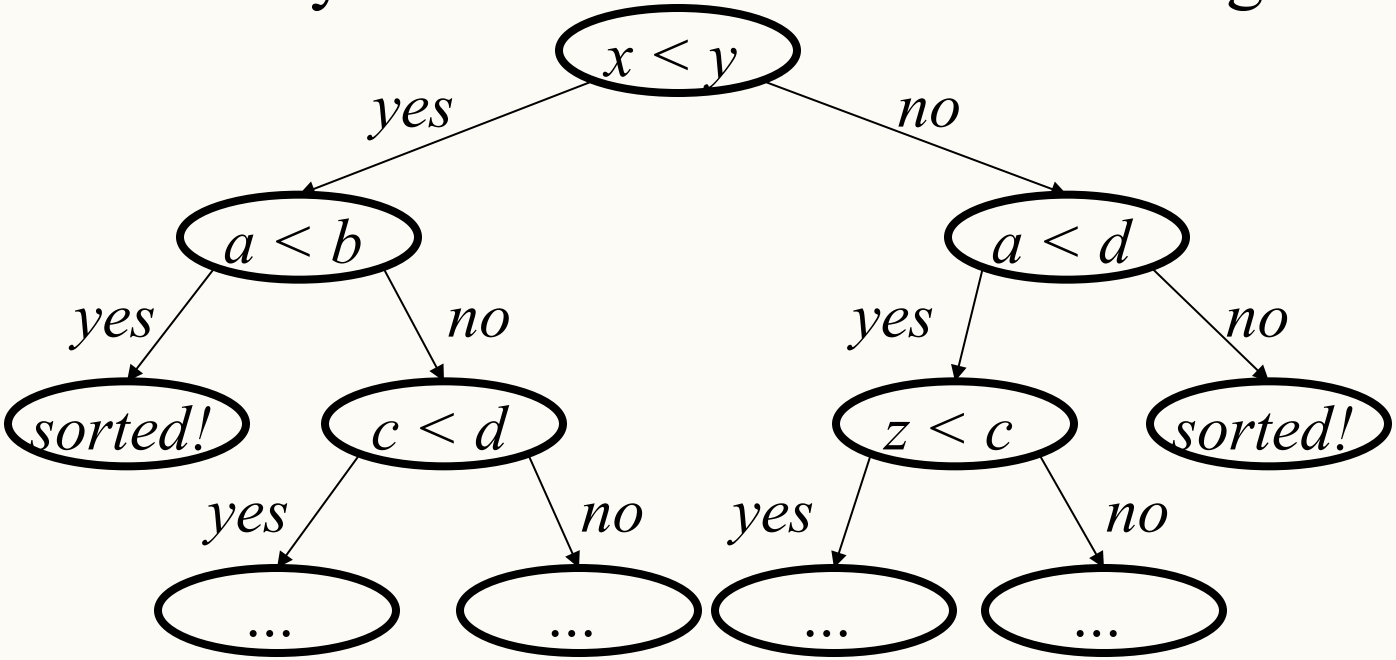
Complexity of Sorting Using Comparisons as a Problem

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.



Learning Goals revisited

- Describe and apply various sorting algorithms:
  - Insertion Sort, Selection Sort, MergeSort, and QuickSort, Bubble Sort, Heapsort
- Compare and contrast the tradeoffs of these algorithms.
- State differences in performance for large files versus small files on various sorting algorithms.
- Analyze the complexity of these sorting algorithms.
- Manipulate data using various sorting algorithms (irrespective of any implementation).