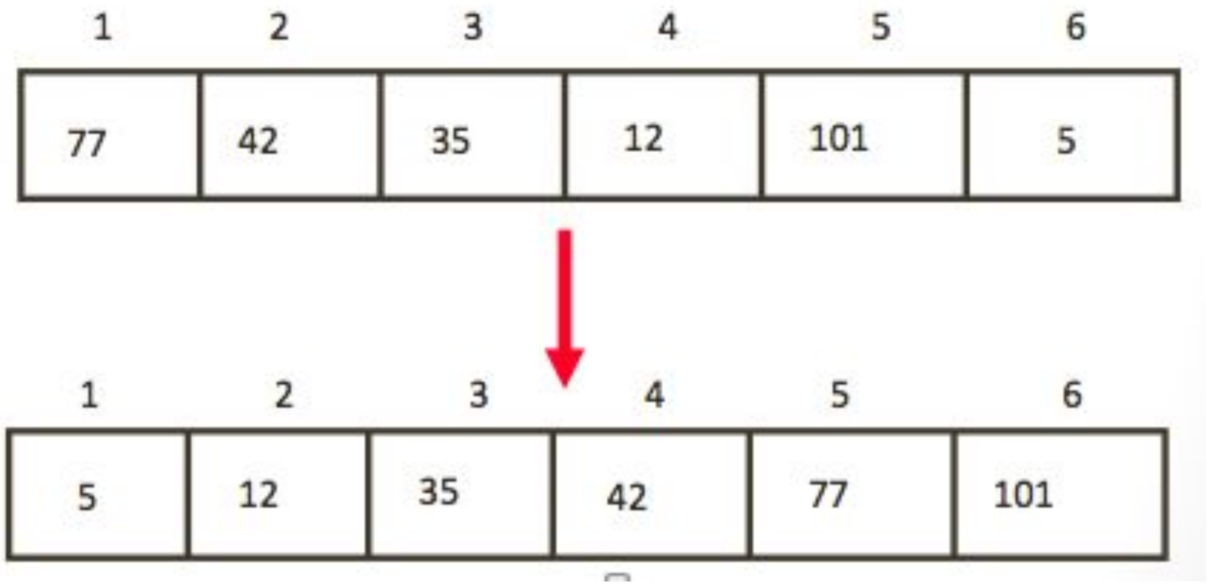
Lecture 13-15

Sorting: checksort, selection, insertion, merge, quick, compareTo

Sorting

Sorting

- Sorting takes an unordered collection and makes it an ordered one.

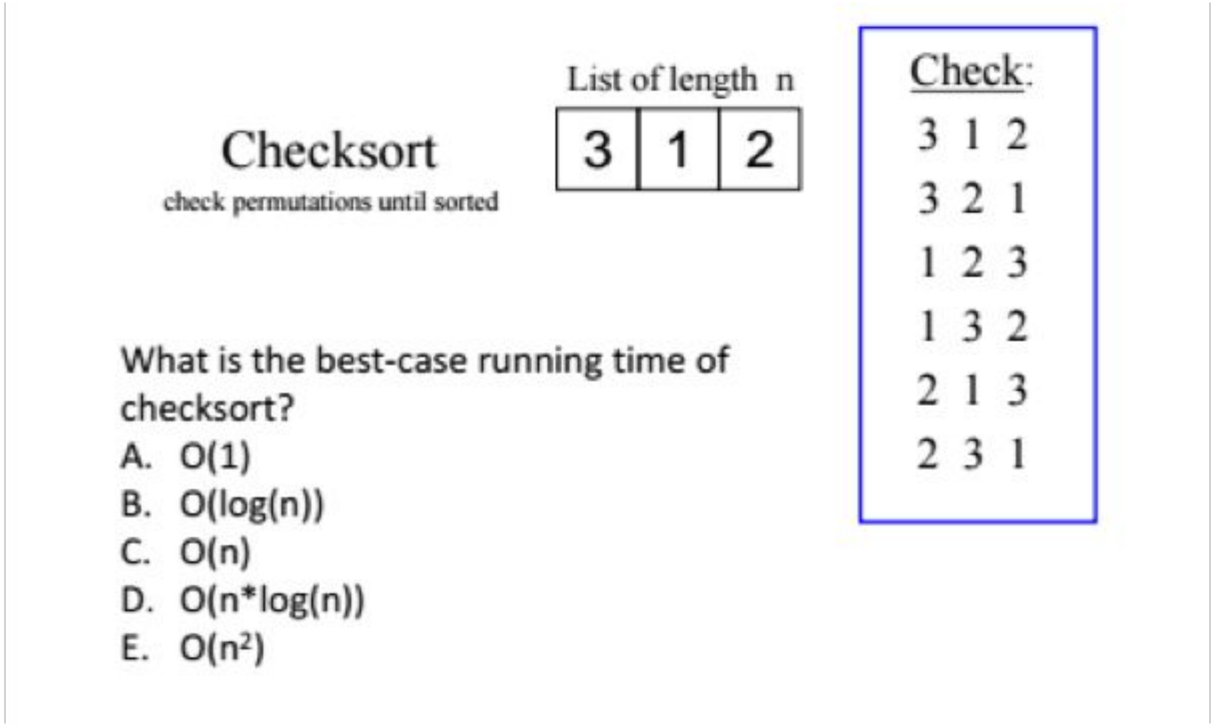


Performance Definitions

- Time complexity

- Space complexity
  - the “extra” memory usage of an algorithm

Check Sort



Checksort

check permutations until sorted

List of length n

3	1	2

Check:

3 1 2

3 2 1

1 2 3

1 3 2

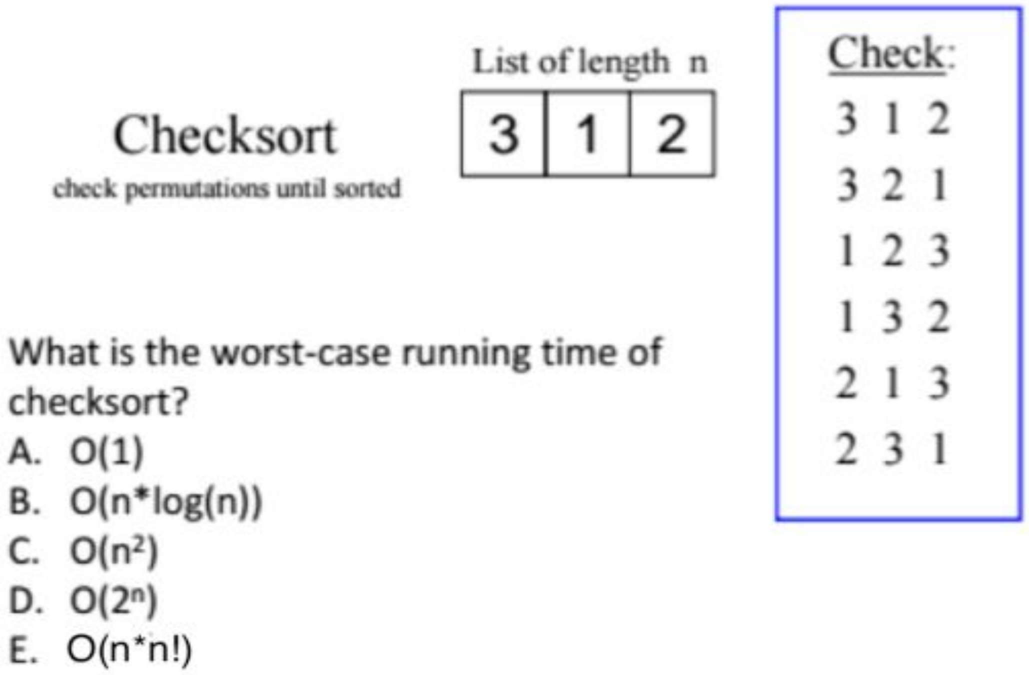
2 1 3

2 3 1

What is the best-case running time of checksort?

A. O(1)
B. O(log(n))
C. O(n)
D. O(n*log(n))
E. O(n^2)

Check Sort



Checksort

check permutations until sorted

List of length n

3	1	2

Check:

3 1 2

3 2 1

1 2 3

1 3 2

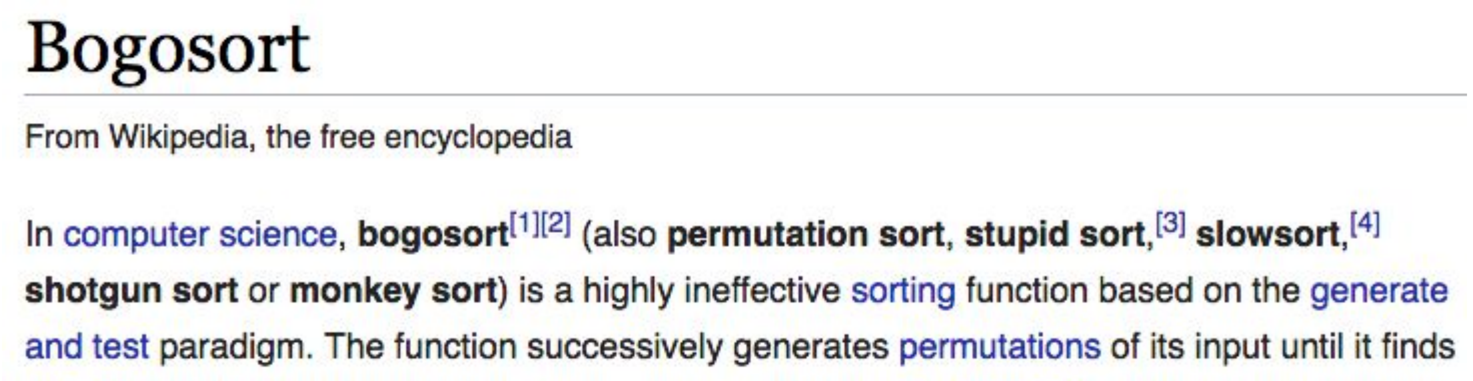
2 1 3

2 3 1

What is the worst-case running time of checksort?

A. O(1)
B. O(n*log(n))
C. O(n^2)
D. O(2^n)
E. O(n*n!)

Bogosort



Bogosort

From Wikipedia, the free encyclopedia

In computer science, **bogosort**[1][2] (also **permutation sort**, **stupid sort**,[3] **slowsort**,[4] **shotgun sort** or **monkey sort**) is a highly ineffective sorting function based on the generate and test paradigm. The function successively generates permutations of its input until it finds

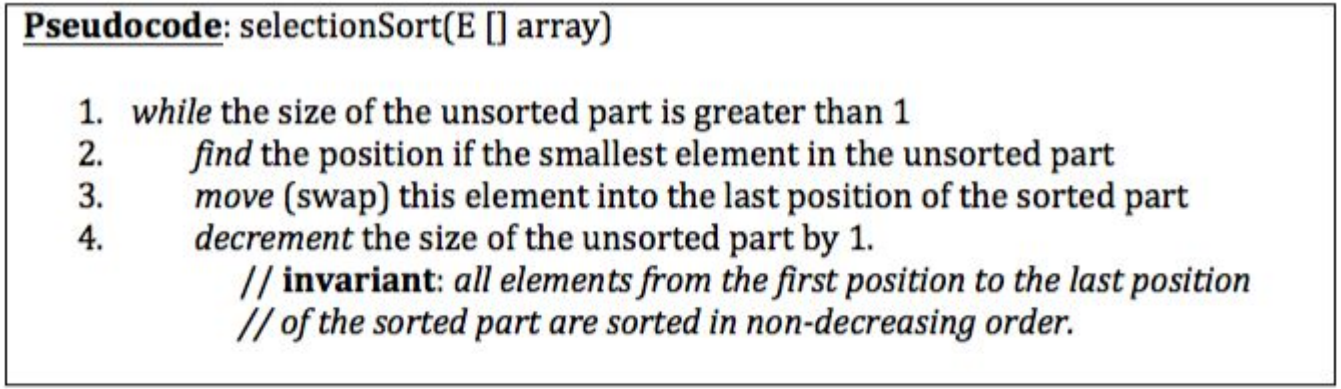
```
while not isInOrder(deck):
    shuffle(deck)
```

reminder

- mic

Selection Sort

Selection Sort. Complexity



Pseudocode: selectionSort(E [] array)

```
1. while the size of the unsorted part is greater than 1
2.       find the position if the smallest element in the unsorted part
3.       move (swap) this element into the last position of the sorted part
4.       decrement the size of the unsorted part by 1.
             // invariant: all elements from the first position to the last position
             // of the sorted part are sorted in non-decreasing order.
```

Complexity of **selection** sort (worst)?

A: Θ(Log N)

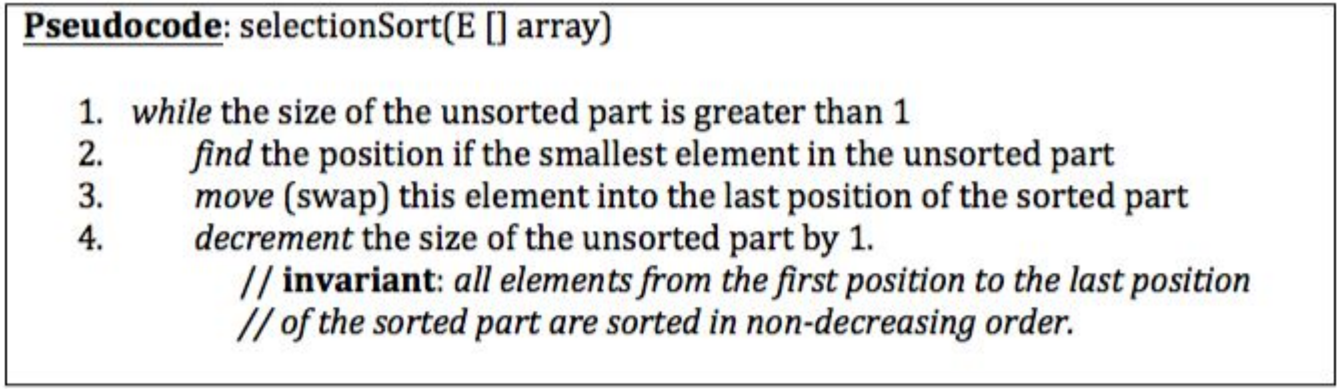
B: Θ(N)

C: Θ(N log N)

D: Θ(N^2)

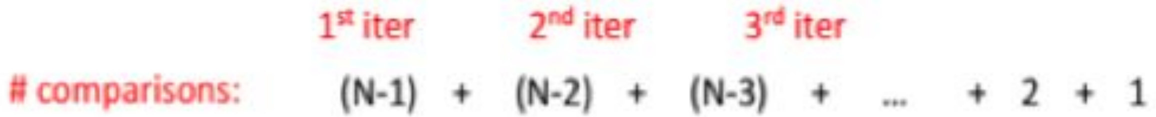
E: Other

Selection Sort. Complexity



Pseudocode: selectionSort(E [] array)

```
1.  while the size of the unsorted part is greater than 1
2.        find the position if the smallest element in the unsorted part
3.        move (swap) this element into the last position of the sorted part
4.        decrement the size of the unsorted part by 1.
              // invariant: all elements from the first position to the last position
              // of the sorted part are sorted in non-decreasing order.
```



	1st iter		2nd iter		3rd iter						
# comparisons:	(N-1)	+	(N-2)	+	(N-3)	+	...	+	2	+	1

Best case running time

for Selection sort?

A: $\Theta(1)$

B: $\Theta(N)$

C: $\Theta(\log N)$

D: $\Theta(N^2)$

E: None of the above

Costs

	Best Case Running time	Worst Case Running time	Additional Space
Selection Sort	$\Theta(N^2)$	$\Theta(N^2)$	$\Theta(1)$

Insertion Sort

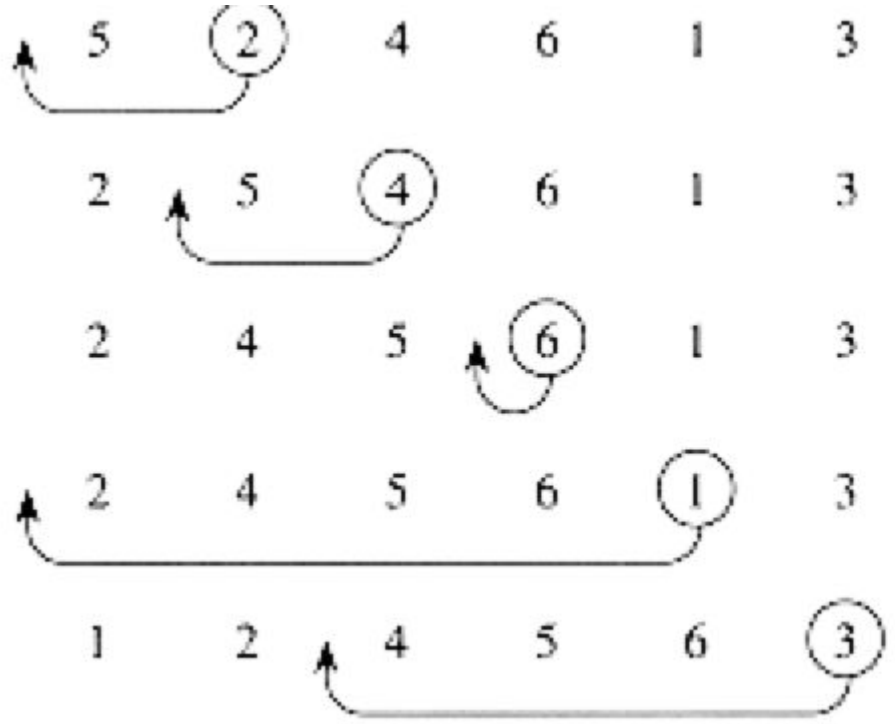
Insertion Sort

General strategy:

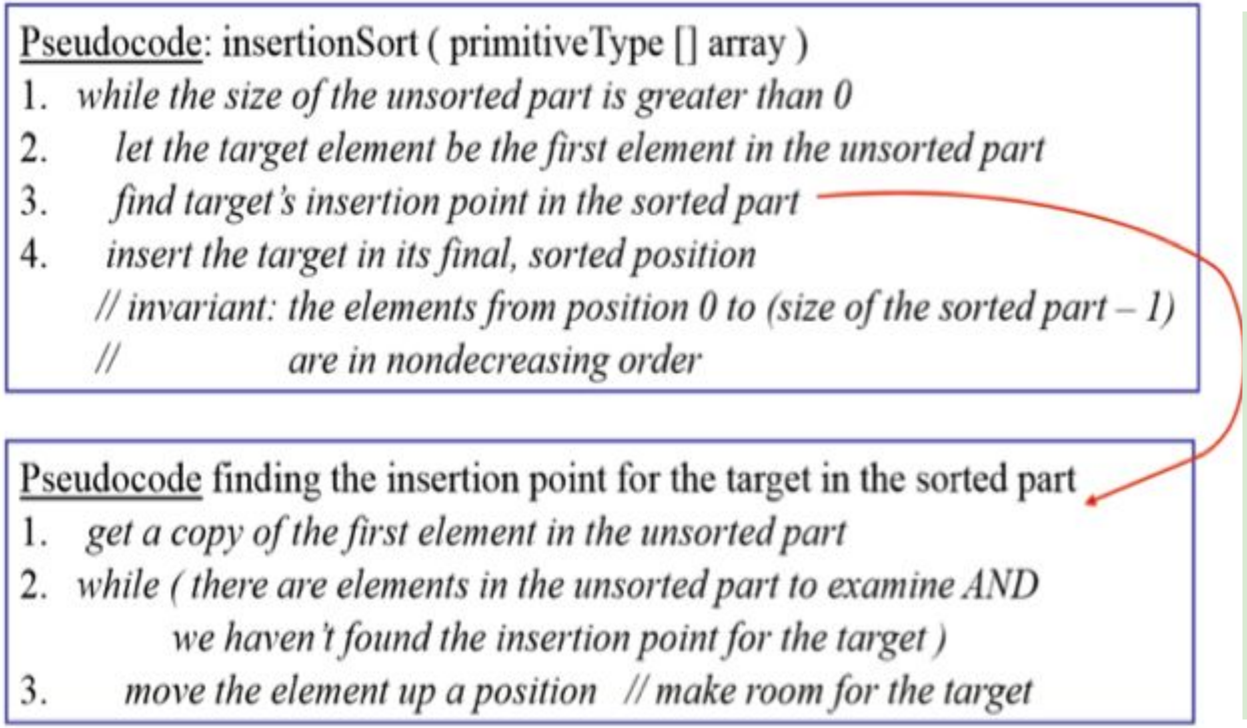
- Starting with an empty output sequence.
- Add each item from input, inserting into output at right point.

Demo (Link)

Invariant: Partly Sorted



Insertion Sort



```
Pseudocode: insertionSort ( primitiveType [] array )
1.  while the size of the unsorted part is greater than 0
2.      let the target element be the first element in the unsorted part
3.      find target's insertion point in the sorted part
4.      insert the target in its final, sorted position
    // invariant: the elements from position 0 to (size of the sorted part – 1)
    //            are in nondecreasing order
```

```
Pseudocode finding the insertion point for the target in the sorted part
1.  get a copy of the first element in the unsorted part
2.  while ( there are elements in the unsorted part to examine AND
            we haven't found the insertion point for the target )
3.      move the element up a position   // make room for the target
```

Complexity of insertion sort (worst)?

A: Θ(Log N)

B: Θ(N)

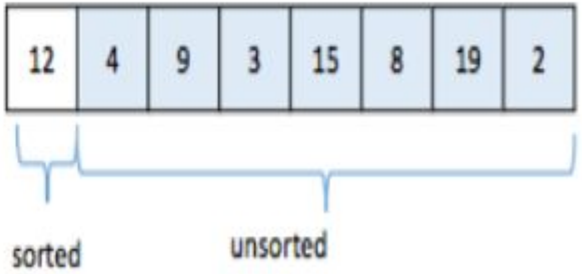
C: Θ(N log N)

D: $\Theta(N^2)$

E: Other

Insertion sort: Worst case analysis

$$\begin{array}{lccccc} & \text{1st iter} & \text{2nd iter} & \text{3rd iter} & & \\ \text{\# comparisons:} & 1\ + & 2\ + & 3\ + & \ldots\ + (N-2) & + (N-1) \end{array}$$



Insertion sort: Best case analysis

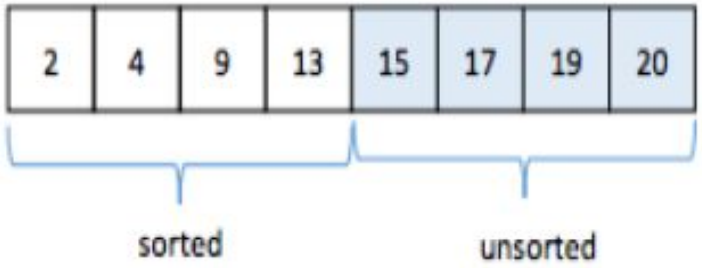
- Approximately how many steps does it take to insert the element into the sorted part each time through the loop (in the BEST case)?

A. 1

B. N

C. N^2

D. It depends on the length of the sorted part



Insertion Sort

- In most cases the insertion sort is the best of the elementary sorts.
- It still executes in Θ(N^2) time, but it's about twice as fast as the bubble sort and somewhat faster than the selection sort in normal situations.
- It's also not too complex, although it's slightly more involved than the bubble and selection sorts.
- It's often used as the final stage of more sophisticated sorts, such as quicksort.
- Works the best if the data is sorted or *partly sorted*.

Costs

	Best Case Running time	Worst Case Running time	Additional Space
Selection Sort	$\Theta(N^2)$	$\Theta(N^2)$	$\Theta(1)$ - in place
Insertion Sort	$\Theta(N)$	$\Theta(N^2)$	$\Theta(1)$ - in place

Which algorithm could produce this list after 3 iterations:

4   5   12   62   45   21   47   30

A. Selection sort
B. Insertion sort
C. Both
D. Neither

Which algorithm could produce this list after 3 iterations:

4 15 12 58 45 62 68 75

A. Selection sort
B. Insertion sort
C. Both
D. Neither

Which algorithm could produce this list after 3 iterations:

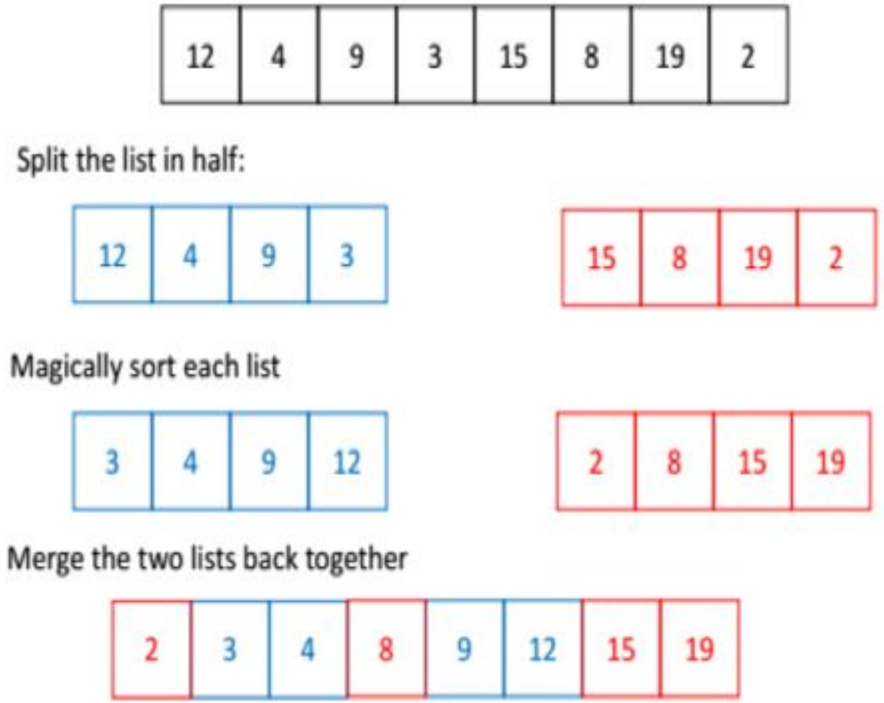
45 53 59 95 14 15 12 64

A. Selection sort
B. Insertion sort
C. Both
D. Neither

Merge Sort

MergeSort: The Magic of Recursion

- Consider this magical way of sorting lists:


Split the list in half:

Magically sort each list

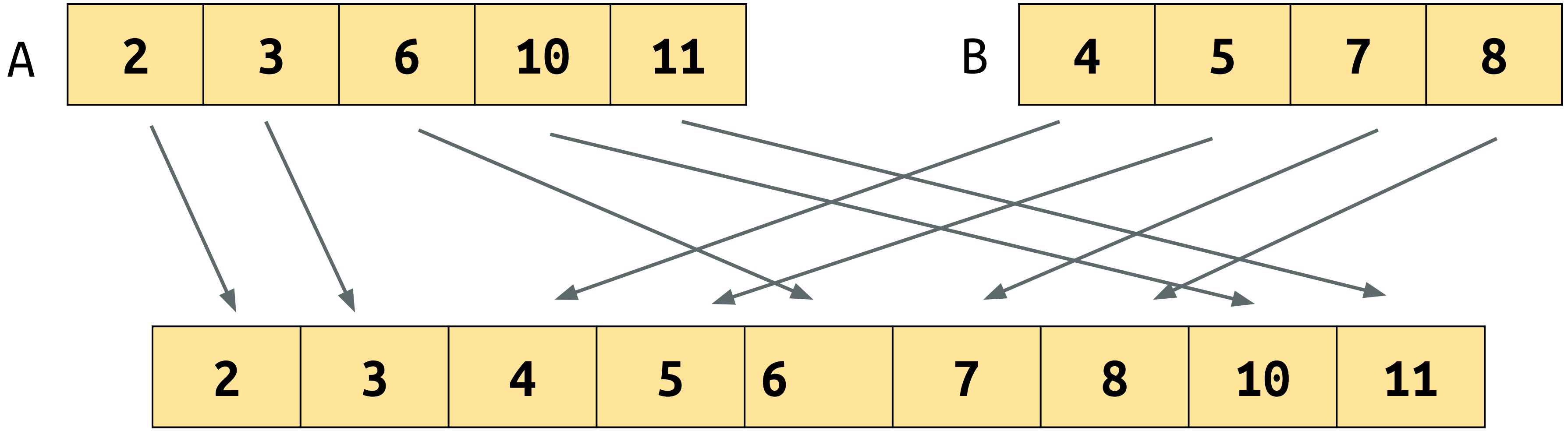
Merge the two lists back together

MergeSort

Mergesort: Demo

- Split items into 2 roughly even pieces.
- Mergesort each half (steps not shown, this is a recursive algorithm!)
- Merge the two sorted halves to form the final result.

Array Merging of N Total Items (A.length + B.length = N)



What is the runtime for **merge step**?

A: Θ(1)

B: Θ(N)

C: $\Theta(N^2)$

D: Θ(N log N)

E: None of the above

Reminder

- mic

```
public void mergeSort( int[] toSort )
{
  int mid = toSort.length / 2;
  int[] firstHalf = makeList(toSort, 0, mid);
  int[] secondHalf = makeList( toSort, mid, toSort.length );
  mergeSort( firstHalf );
  mergeSort( secondHalf );
  merge( firstHalf, secondHalf, toSort );

}
```

Makes a new array and copies the elements from the specified range

Merges the first two lists together into the third, Maintaining sorted order

Does this mergeSort method work?

A. Yes

B. No

Lost in Recursion Land

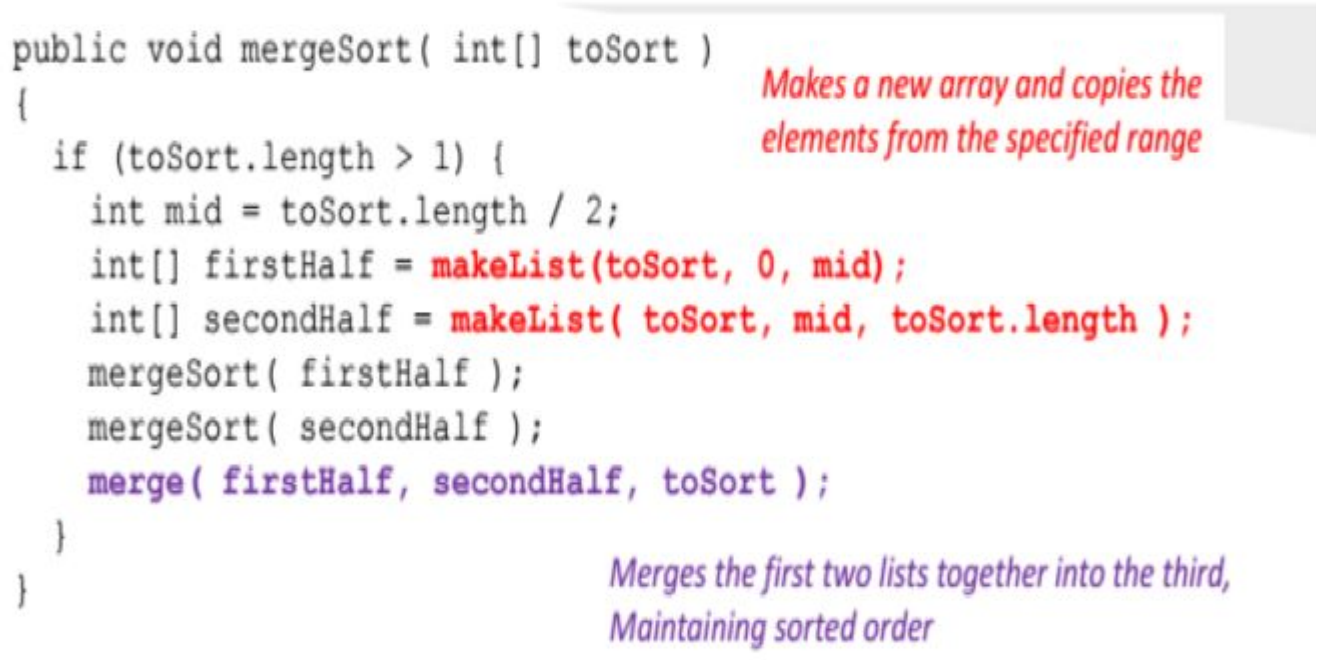
- Beginners often fail to appreciate that a recursion must have a conditional statement or conditional expression that checks for the "bottom-out" condition of the recursion and terminates the recursive descent

- We call the bottom-out condition

the "base case" of the recursion

Warning!!

If you fail to do this properly, you end up lost in Recursion Land and you never return!



```
public void mergeSort( int[] toSort )
{
  if (toSort.length > 1) {
    int mid = toSort.length / 2;
    int[] firstHalf = makeList(toSort, 0, mid);
    int[] secondHalf = makeList( toSort, mid, toSort.length );
    mergeSort( firstHalf );
    mergeSort( secondHalf );
    merge( firstHalf, secondHalf, toSort );
  }
}
```

Makes a new array and copies the elements from the specified range

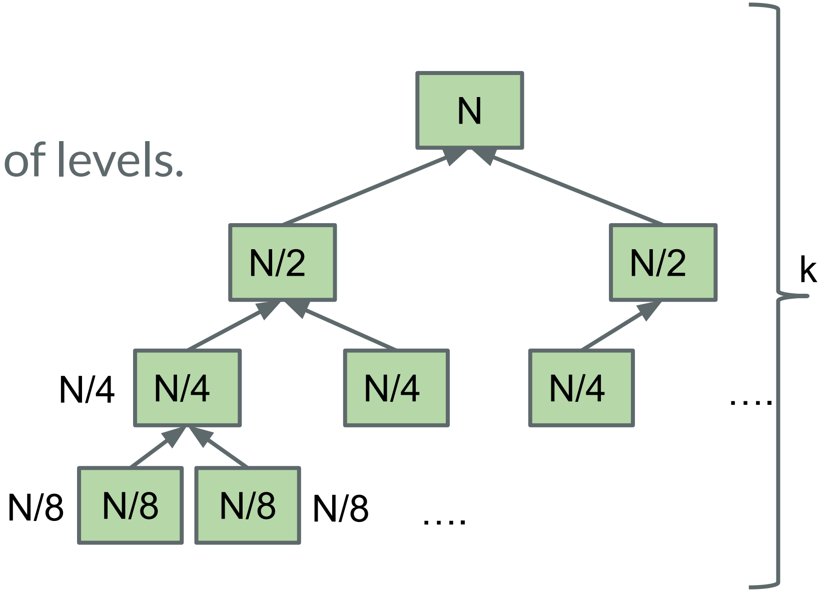
Merges the first two lists together into the third, Maintaining sorted order

This mergeSort works!!

Merge Sort: More General

Intuitive explanation:

- Every level does N work
  - Top level does N work.
  - Next level does N/2 + N/2 = N.
  - One more level down: N/4 + N/4 + N/4 + N/4 = N.
- Thus work is just Nk, where k is the number of levels.
  - How many levels? Goes until we get to size 1.
  - k = lg(N)
- Overall runtime is N log N.



MergeSort

Mergesort: Demo

- Split items into 2 roughly even pieces.
- Mergesort each half (steps not shown, this is a recursive algorithm!)
- Merge the two sorted halves to form the final result.

Space complexity with aux array: Costs Θ(N) memory.

Also possible to do in-place merge sort, **but** algorithm is very complicated, and runtime performance suffers by a significant constant factor.

Which input wins for merge sort?

A: Random

B: (Nearly) Sorted

C: Duplicates

D: Reversed

E: Does not matter

https://www.toptal.com/developers/sorting-algorithms

If time permits

https://www.youtube.com/watch?v=XaqR3G_NVoo

https://www.youtube.com/watch?v=semGJAJ7i74&list=PLOmdoKois7_FK-ySGwHBkltzB11snW7KQ

Costs

	Best Case Running time	Worst Case Running time	Additional Space
Selection Sort	$\Theta(N^2)$	$\Theta(N^2)$	$\Theta(1)$ - in place
Insertion Sort	$\Theta(N)$	$\Theta(N^2)$	$\Theta(1)$ - in place
Merge Sort	$\Theta(N \text{ Log } N)$	$\Theta(N \text{ Log } N)$	$\Theta(N)$ - not in place

Quick Sort

The Core Idea: Partitioning

To partition an array a[] on element x=a[i] is to rearrange a[] so that:

- x moves to position j (may be the same as i)
- All entries to the left of x are <= x.
- All entries to the right of x are >= x.

Called the ‘pivot’.

		i				
5	550	10	4	10	9	330

A.

			j			
4	5	9	10	10	330	550

B.

			j			
5	4	9	10	10	550	330

C.

				j		
5	9	10	4	10	550	330

D.

		j				
5	9	10	4	10	550	330

E. More than one

Which partitions are valid?

Interview Question (Partitioning)

Given an array of colors where the 0th element is white, and the remaining elements are red (less) or blue (greater), rearrange the array so that all red squares are to the left of the white square, and all blue squares are to the right. Your algorithm must complete in Θ(N) time (no space restriction).

- Relative order of red and blues does NOT need to stay the same.

Input

6	8	3	1	2	7	4

Example of a valid output

3	1	2	4	6	8	7

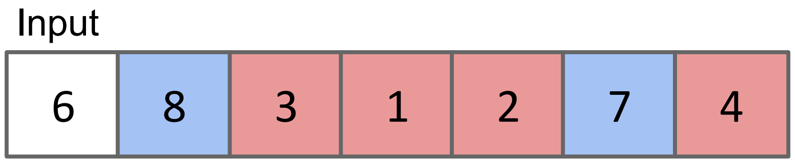
Another example of a valid output

3	4	1	2	6	7	8

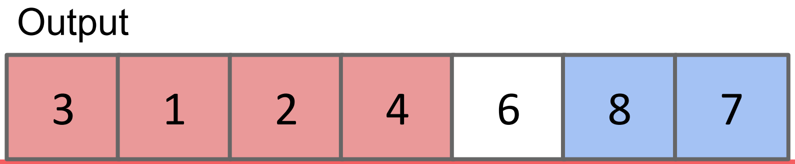
Simplest (but not fastest) Answer: 3 Scan Approach

Given an array of colors where the 0th element is white, and the remaining elements are red (less) or blue (greater), rearrange the array so that all red squares are to the left of the white square, and all blue squares are to the right. Your algorithm must complete in Θ(N) time (no space restriction).

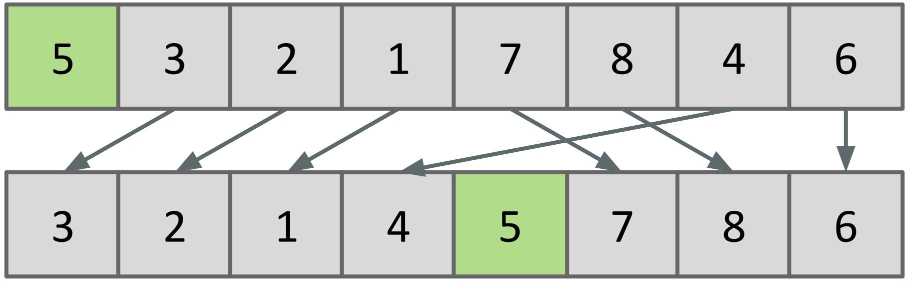
- Relative order of red and blues does NOT need to stay the same.



Algorithm: Create another array. Scan and copy all the red items to the first R spaces. Then scan for and copy the white item. Then scan and copy the blue items to the last B spaces.



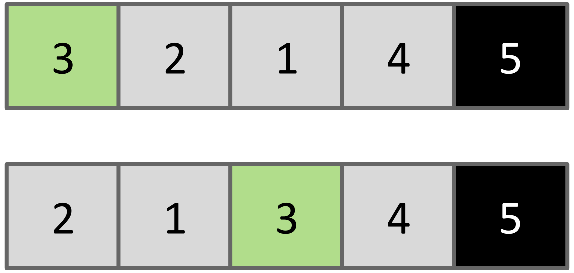
Partition Sort, a.k.a. Quicksort

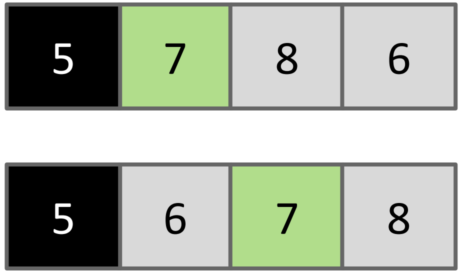


Q: How would we use this operation for sorting?

Observations:

- 5 is "in its place." Exactly where it'd be if the array were sorted.
- Can sort two halves separately, e.g. through recursive use of partitioning.

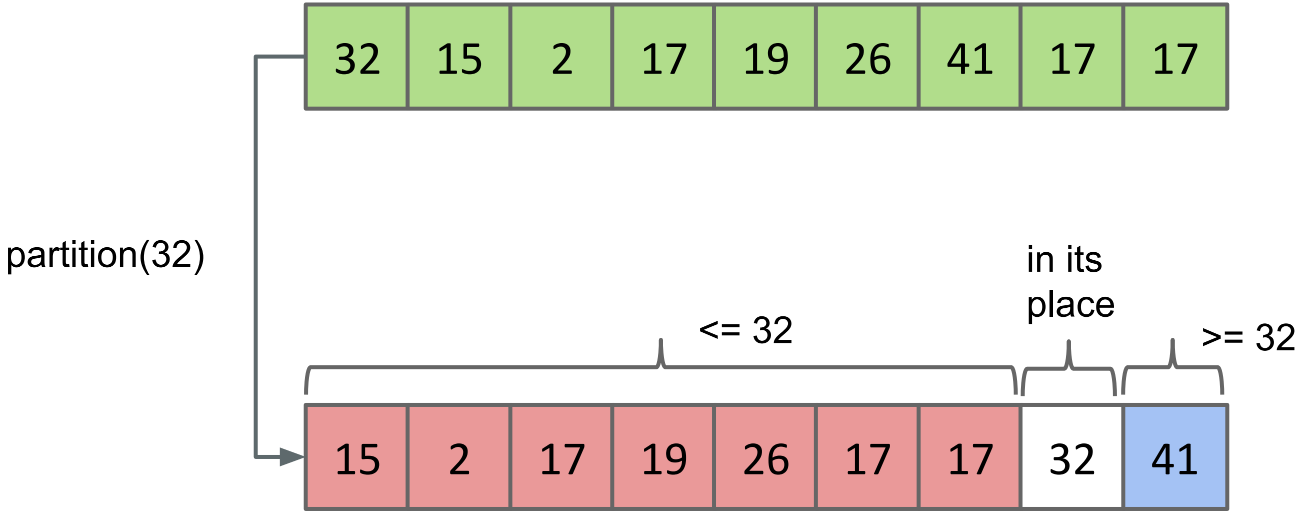




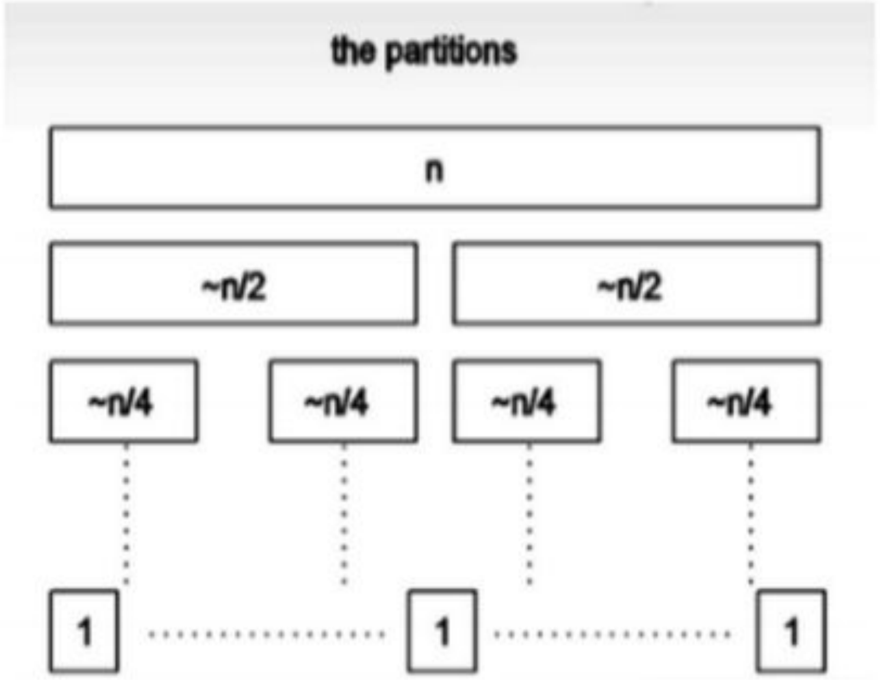
Partition Sort, a.k.a. Quicksort

Quicksorting N items: (Demo)

- Partition on leftmost item.
- Quicksort left half.
- Quicksort right half.



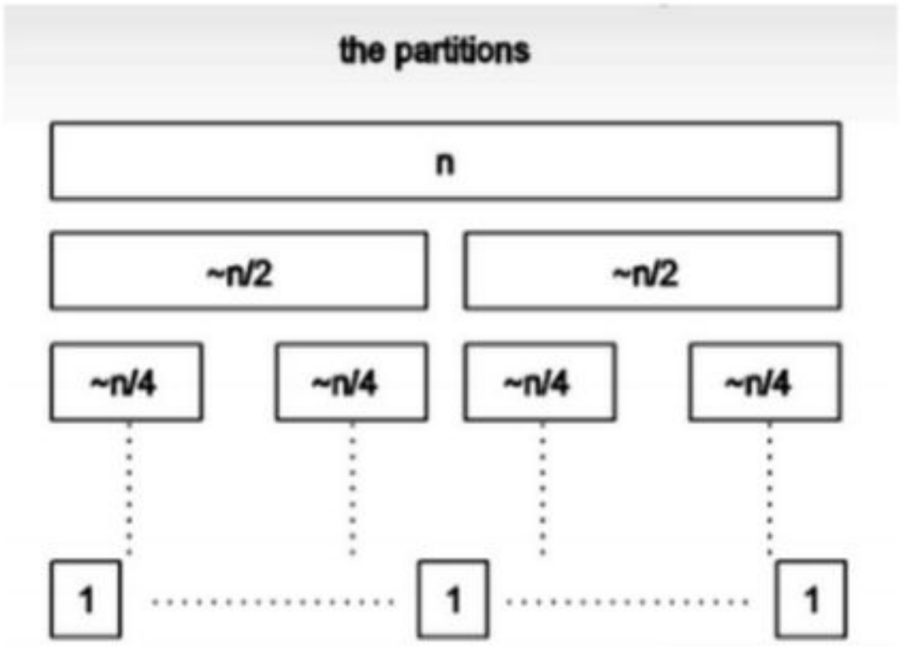
Using a good pivot



How many levels will there be if you choose a pivot that divides the list in half?

A. 1
B. log(N)
C. N
D. N*log(N)
E. N^2

Using a good pivot



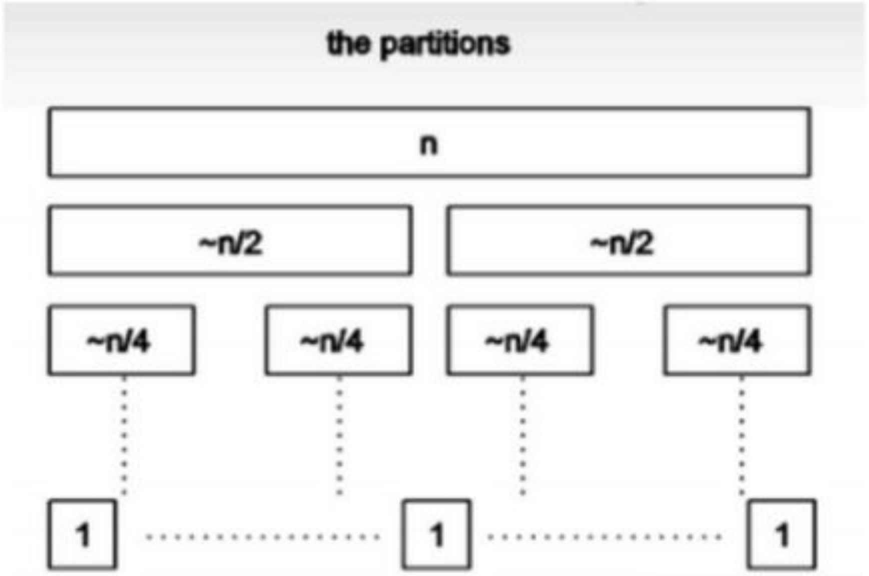
If the time to partition on each level takes N comparisons, how long does Quicksort take with a good partition?

A. O(1)
B. O(log(N))
C. O(N)
D. O(N*log(N))
E. $O(N^2)$

reminder

Record

Using a good pivot



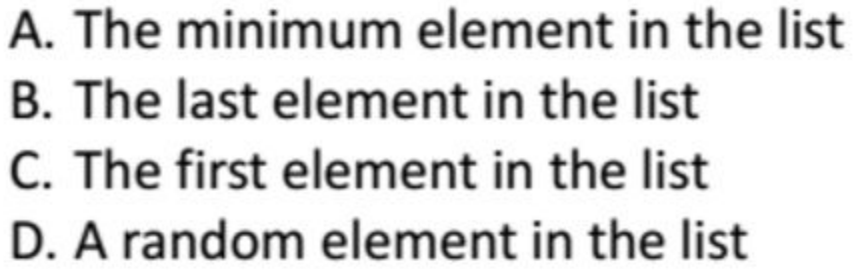
Space complexity: $O(\log_2 n)$ for the runtime stack activation records

If the time to partition on each level takes N comparisons, how long does Quicksort take with a good partition?

A. O(1)
B. O(log(N))
C. O(N)
D. O(N*log(N))
E. $O(N^2)$

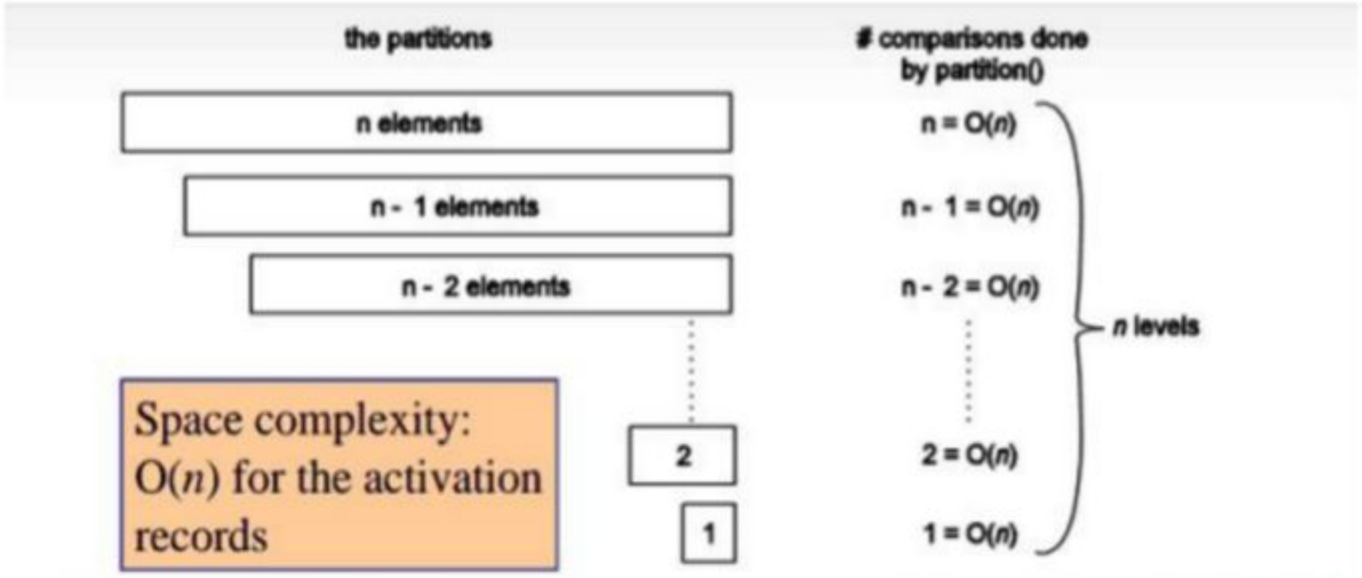
Question

Which of these choices would be the **worst** choice for the pivot?



A. The minimum element in the list
B. The last element in the list
C. The first element in the list
D. A random element in the list

Quick sort with a bad pivot

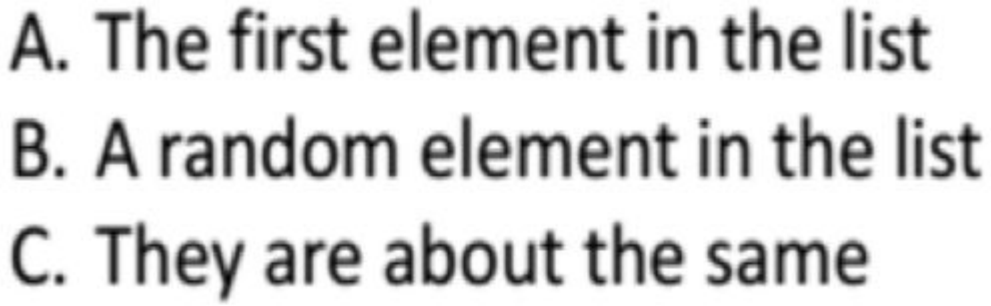


Space complexity: O(n) for the activation records

If the pivot always produces one empty partition and one with $n - 1$ elements, there will be n levels, each of which requires O(n) comparisons: O(n^2) time complexity

Question

Which of these choices is a better choice for the pivot?



A. The first element in the list
B. A random element in the list
C. They are about the same

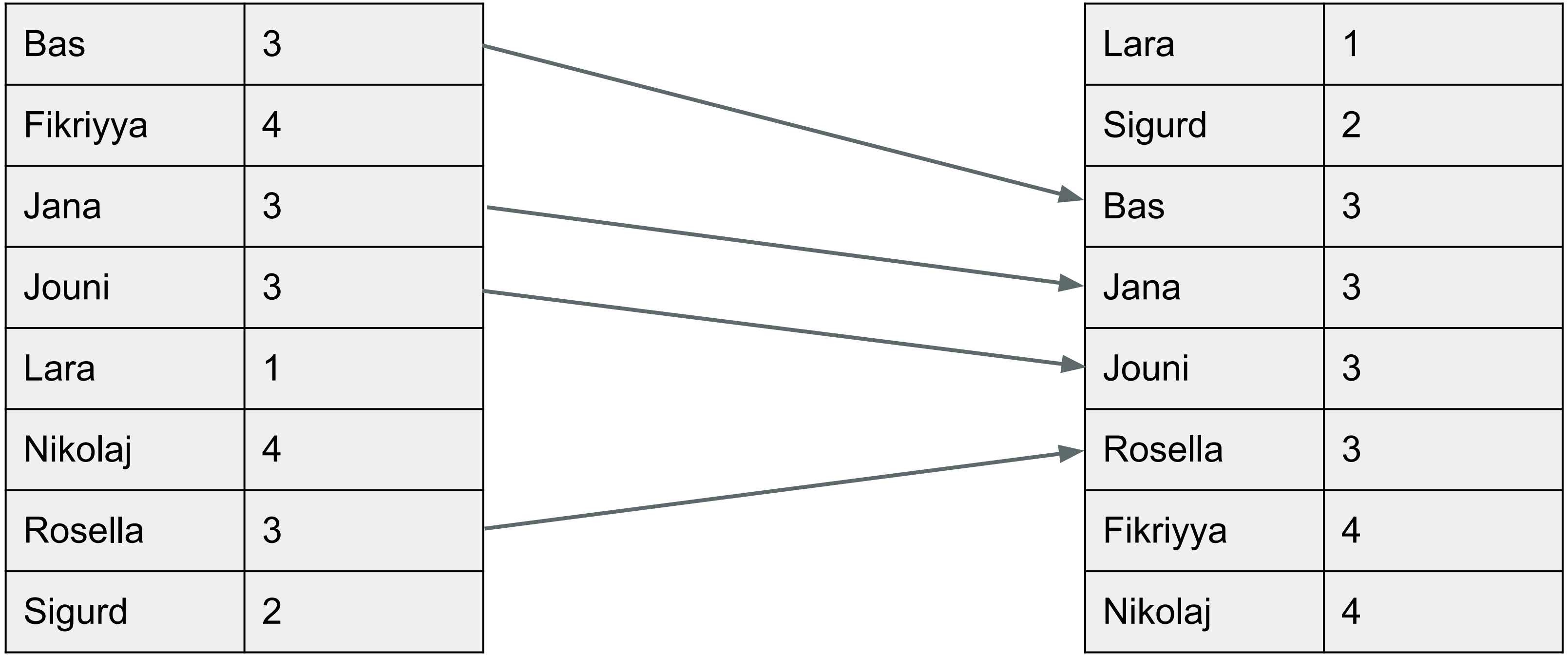
Costs

	Best Case Running time	Worst Case Running time	Additional Space
Selection Sort	Θ(N^2)	Θ(N^2)	Θ(1) - in place
Insertion Sort	Θ(N)	Θ(N^2)	Θ(1) - in place
Merge Sort	Θ(N Log N)	Θ(N Log N)	Θ(N)
Quick Sort	Θ(N Log N)	Θ(N^2)	Θ(log n)

Stability

Other Desirable Sorting Property: Stability

A sort is said to be **stable** if the order of equivalent items is preserved.



sort(studentRecords, BY_NAME);

Bas	3
Fikriyya	4
Jana	3
Jouni	3
Lara	1
Nikolaj	4
Rosella	3
Sigurd	2

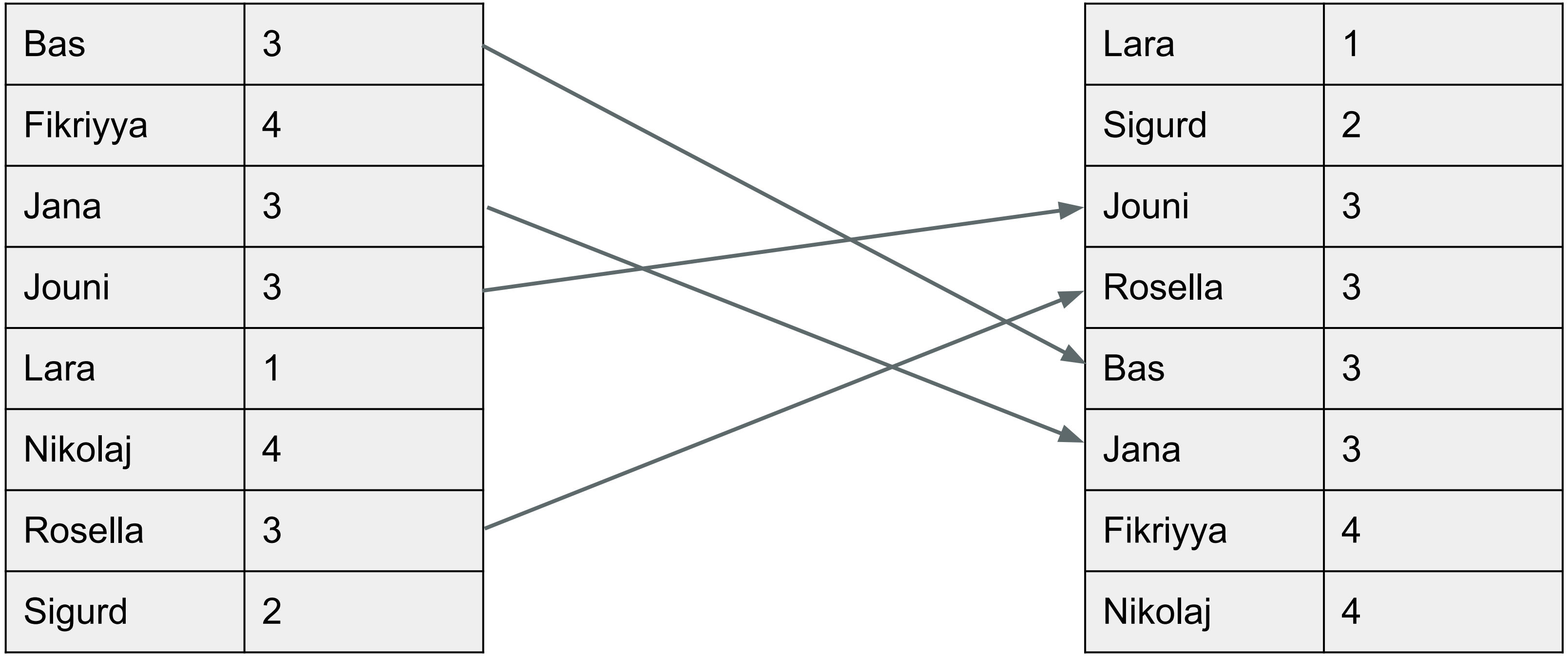
sort(studentRecords, BY_SECTION);

Lara	1
Sigurd	2
Bas	3
Jana	3
Jouni	3
Rosella	3
Fikriyya	4
Nikolaj	4

Equivalent items don't 'cross over' when being stably sorted.

Other Desirable Sorting Property: Stability

A sort is said to be stable if order of equivalent items is **preserved**.



sort(studentRecords, BY_NAME);

Bas	3
Fikriyya	4
Jana	3
Jouni	3
Lara	1
Nikolaj	4
Rosella	3
Sigurd	2

sort(studentRecords, BY_SECTION);

Lara	1
Sigurd	2
Jouni	3
Rosella	3
Bas	3
Jana	3
Fikriyya	4
Nikolaj	4

Sorting instability can be really annoying! Wanted students listed alphabetically by section.

Is Selection Sort Stable? Sort to find out!

Bas	3
Fikriyya	4
Jana	3
Jouni	3
Lara	1
Nikolaj	4
Rosella	3
Sigurd	2

Is Insertion Sort Stable? Sort to find out!

Bas	3
Fikriyya	4
Jana	3
Jouni	3
Lara	1
Nikolaj	4
Rosella	3
Sigurd	2

Is Merge Sort Stable? Sort to find out!

Bas	3
Fikriyya	4
Jana	3
Jouni	3
Lara	1
Nikolaj	4
Rosella	3
Sigurd	2

Suppose we do the following:

- Read 1,000,000 integers from a file into an array of length 1,000,000.
- Mergesort these integers.
- Select one integer randomly and change it.
- Sort using algorithm X of your choice.

Which sorting algorithm would be the fastest choice for X?

A. Selection Sort
B. Mergesort
C. Insertion Sort
D. Quick Sort

Costs

	Best Case Running time	Worst Case Running time	Additional Space	Stability
Selection Sort	$\Theta(N^2)$	$\Theta(N^2)$	$\Theta(1)$ - in place	No
Insertion Sort	$\Theta(N)$	$\Theta(N^2)$	$\Theta(1)$ - in place	Yes
Merge Sort	$\Theta(N \text{ Log } N)$	$\Theta(N \text{ Log } N)$	$\Theta(N)$	Yes
Random Quick Sort	$\Theta(N \text{ Log } N)$ And average	$\Theta(N^2)$	$\Theta(\log n)$	No

Questions?

Empirical Testing

Practical world

- Sometimes calculating instructions is not possible or very difficult.

- As an alternative to describing an algorithm’s performance with a “number of abstract operations”, we can also measure its time empirically using a clock.

Explain these results

Timing for `findMax()` on an array of *`ints`* and an array of *`Integer objects`* (times are in milliseconds)

n	Array of `int`	Array of `Integer`
800,000	2.314	4.329
4,000,000	11.363	21.739
8,000,000	22.727	42.958

For these measurements, what is the likely Θ time cost of findMax() on array of `int`

A: $f(n) = \Theta(\log_2 n)$

B: $f(n) = \Theta(n \log_2 n)$

C: $f(n) = \Theta(n^2)$

D: $f(n) = \Theta(n)$

E: None of these

Explain these results

Timing for `findMax()` on an array of *`ints`* and an array of *`Integers`* (times are in milliseconds)

n	Array of `int`	Array of `Integer`
800,000	2.314	4.329
4,000,000	11.363	21.739
8,000,000	22.727	42.958

For these measurements, what is the likely ϴ time cost of findMax() on array of `Integer`

A: $f(n) = \Theta(\log_2 n)$

B: $f(n) = \Theta(n \log_2 n)$

C: $f(n) = \Theta(n^2)$

D: $f(n) = \Theta(n)$

E: None of these

Explain these results

Timing for `findMax()` on an array of *`ints`* and an array of *`Integer`* (times are in milliseconds)

n	Array of `int`	Array of `Integer`
800,000	2.314	4.329
4,000,000	11.363	21.739
8,000,000	22.727	42.958

Discussion

- Why would `Integer` take more time?
- If `Integer` has more time, why does it have the same Θ cost as `int`?

Reasons to use “real timers”

- As illustrated, counting “abstract operations” can anyway hide real performance differences, e.g., between using int[] and Integer[]

- Many programs and libraries are not open source!
- You have to analyze an algorithm’s performance as a black box.

Procedure for measuring time cost

- Let’s suppose we wish to measure the time cost of algorithm A as a function of its input size n.
- We need to choose a set of values of n that we will test.
- If we make n too big, our algorithm A may never terminate (the input is “too big”).
- If we make n too small, then A may finish so fast that the “elapsed time” is practically 0, and we won’t get a reliable clock measurement.

Procedure for measuring time cost

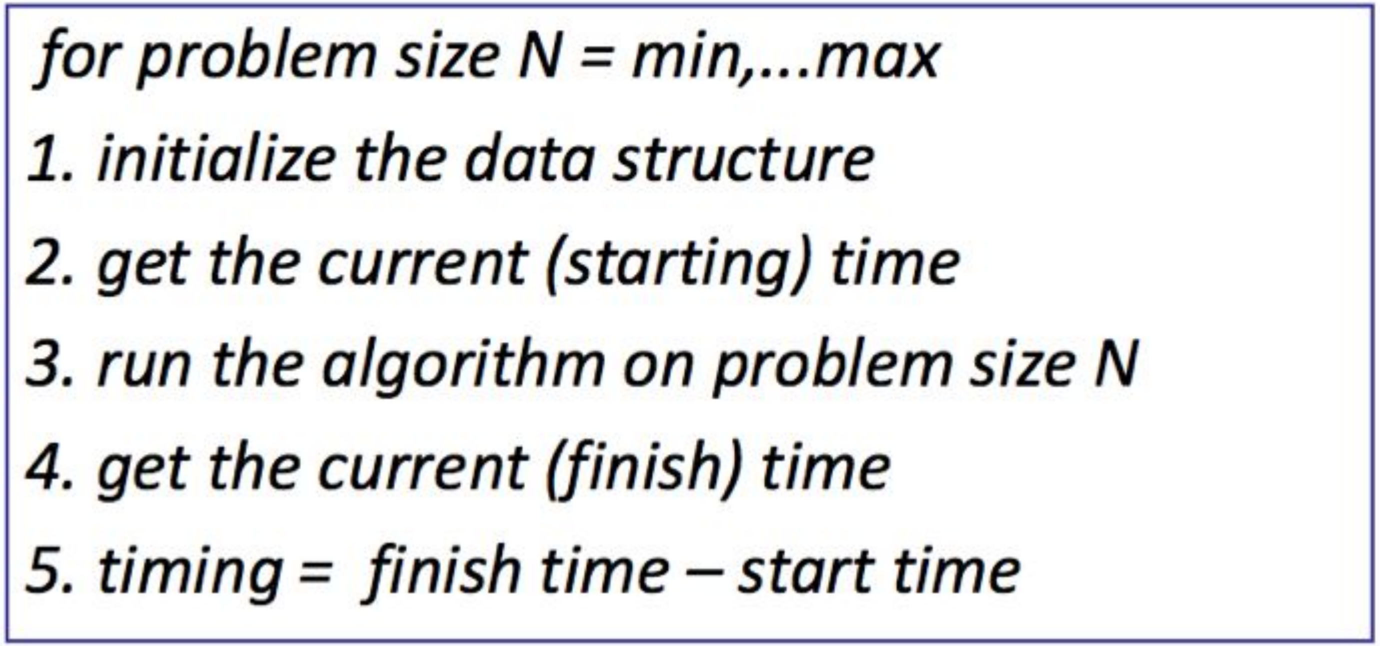
- In practice, one “guesses” a few values for n, sees how fast A executes on them, and selects a range of values for n.

- Let’s define an array of different input sizes, e.g.:

```
int[ ] N = { 1000, 2000, 3000, ..., 10000 };
```

- Now, for each input size N[i], we want to measure A’s time cost.

Basic procedure for benchmarking



```
for problem size N = min,...max
1. initialize the data structure
2. get the current (starting) time
3. run the algorithm on problem size N
4. get the current (finish) time
5. timing =  finish time – start time
```

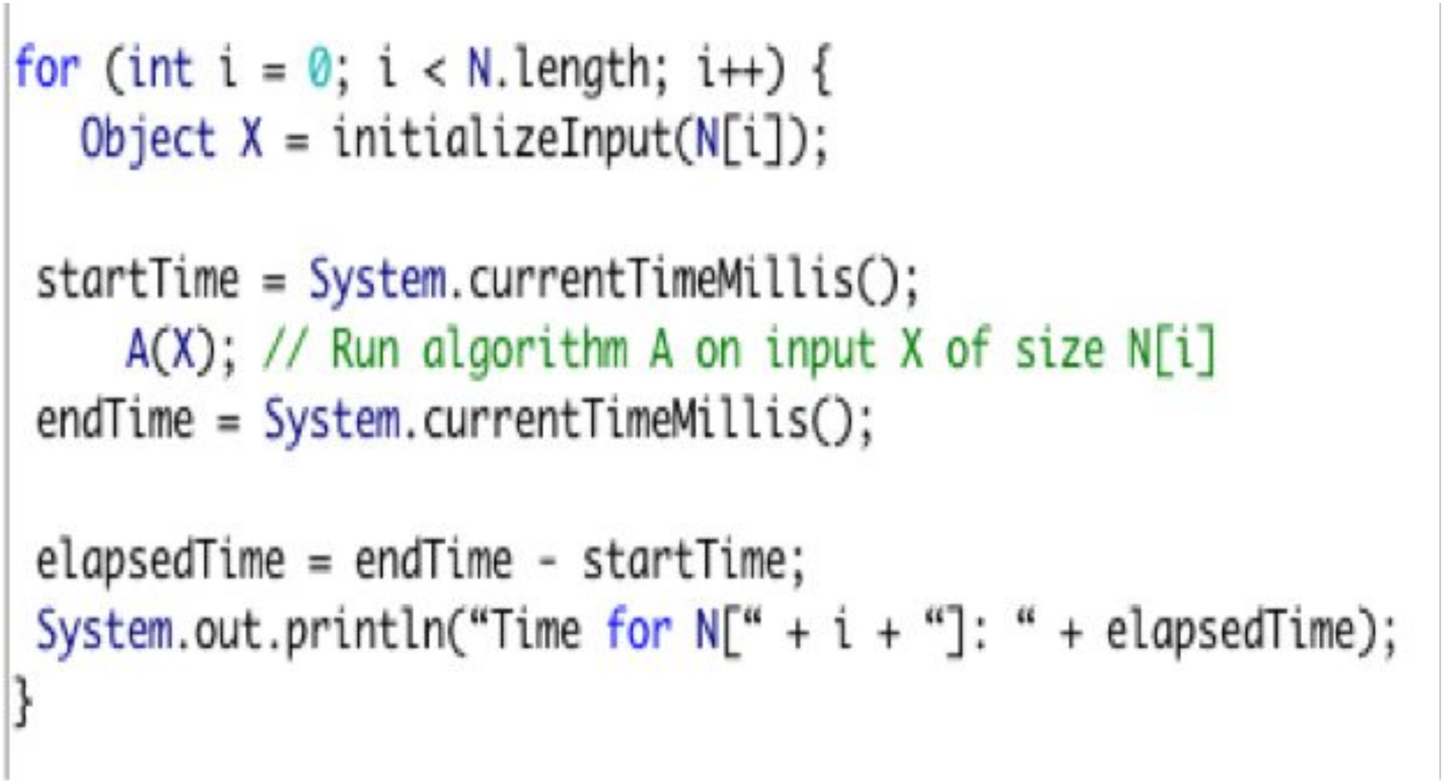
Timer methods in Java

- Java has two static methods in the System class:

```
/** Returns the current time in milliseconds. */
static long System.currentTimeMillis()

/** Returns the current value of the most precise
    available system timer, in nanoseconds */
static long System.nanoTime()
```

- If the algorithm can take less than a millisecond to run, you should use System.nanoTime() !

Draft 1



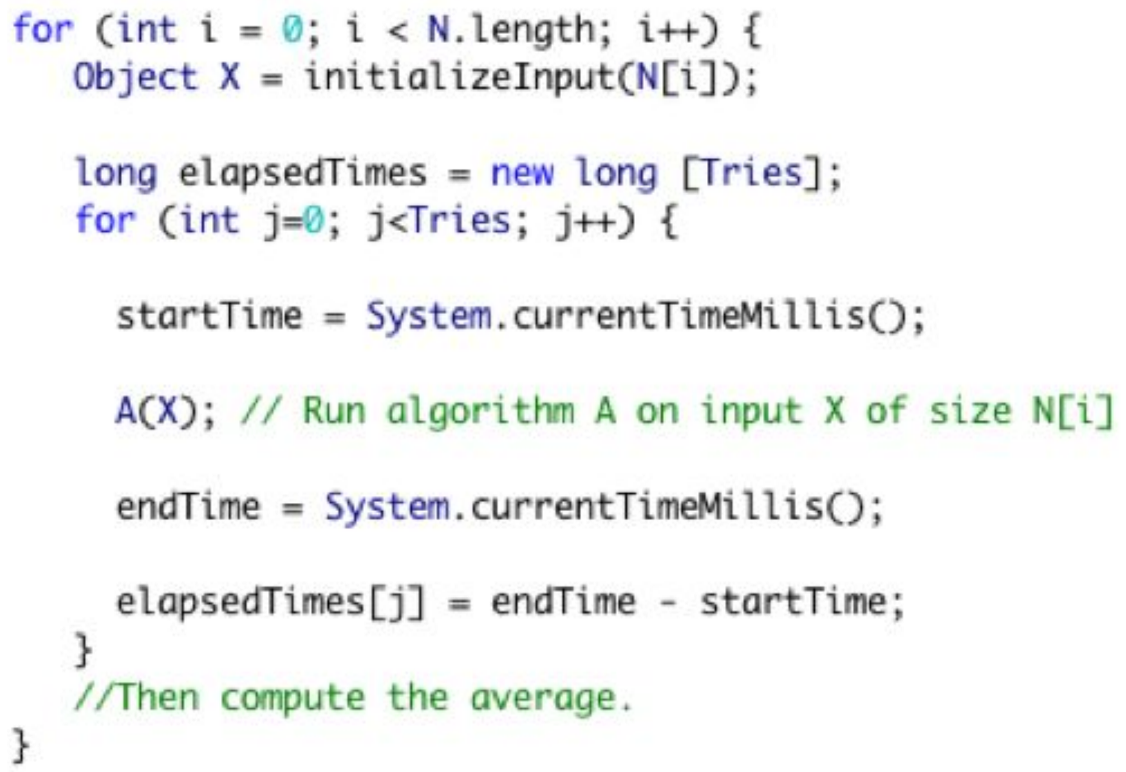
```
for (int i = 0; i < N.length; i++) {
   Object X = initializeInput(N[i]);

 startTime = System.currentTimeMillis();
     A(X); // Run algorithm A on input X of size N[i]
 endTime = System.currentTimeMillis();

 elapsedTime = endTime - startTime;
 System.out.println("Time for N[" + i + "]: " + elapsedTime);
}
```

Is it enough to execute once?

- Unfortunately, in the "real world", each measurement of the time cost of A(X) is corrupted by noise:
    - Garbage collector!
    - Other programs running.
    - Swapping to/from disk.
    - Input/output requests from external devices
- If we measured the time cost of A(X) based on just one measurement, then our estimate of the "true" time cost of A(X) will be very imprecise.
- We might get unlucky and measure A(X) while the computer is doing a "system update".

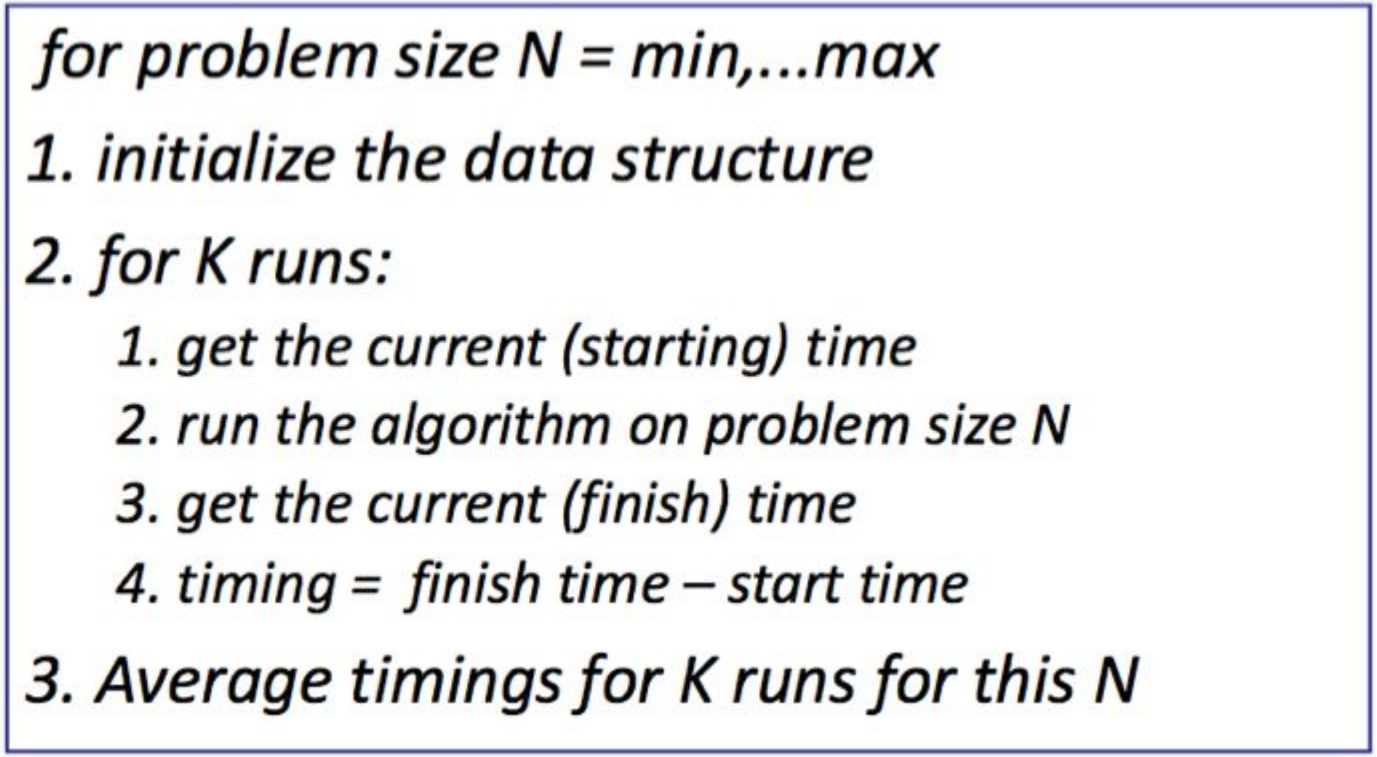
Draft 2

A much-improved procedure for measuring the
time cost of A(X) is to compute the average time
across Tries attempts.



```
for (int i = 0; i < N.length; i++) {
   Object X = initializeInput(N[i]);

   long elapsedTimes = new long [Tries];
   for (int j=0; j<Tries; j++) {

     startTime = System.currentTimeMillis();

     A(X); // Run algorithm A on input X of size N[i]

     endTime = System.currentTimeMillis();

     elapsedTimes[j] = endTime - startTime;
   }
   //Then compute the average.
}
```

IMPROVED procedure for benchmarking



```
for problem size N = min,...max
1. initialize the data structure
2. for K runs:
        1. get the current (starting) time
        2. run the algorithm on problem size N
        3. get the current (finish) time
        4. timing =  finish time – start time
3. Average timings for K runs for this N
```