

# Recursion

---

Dr. Anirban Ghosh

**School of Computing**  
**University of North Florida**



## What is recursion?

**Recursion** is a technique for solving a computational problem where the final solution to the problem is constructed using the solutions of smaller subproblems, obtained recursively.

# Factorial

For a non-negative integer  $n$ , we define  $n!$  (read as  $n$  factorial) as:

$$n! = 1 \times 2 \times \dots \times n$$

Factorial can also be defined recursively as:

$$n! = \begin{cases} 1 & \text{if } n = 0, 1 \\ n \cdot (n-1)! & \text{otherwise} \end{cases}$$

Expressing using functions we obtain:

$$f(n) = \begin{cases} 1 & \text{if } n = 0, 1 \\ n \cdot f(n-1) & \text{otherwise} \end{cases}$$

# Recursive code

$$f(n) = \begin{cases} 1 & \text{if } n = 0, 1 \\ n \cdot f(n-1) & \text{otherwise} \end{cases}$$

```
public class Factorial {  
    public static long factorial(int n) {  
        if( n < 0 )  
            throw new IllegalArgumentException("n must non-negative!");  
        else if( n == 0 || n == 1 ) // base cases  
            return 1;  
        else  
            return n * factorial(n-1); // recursive call  
    }  
  
    public static void main(String[] args) {  
        System.out.println( factorial(5) );  
    }  
}
```

# Recursive code

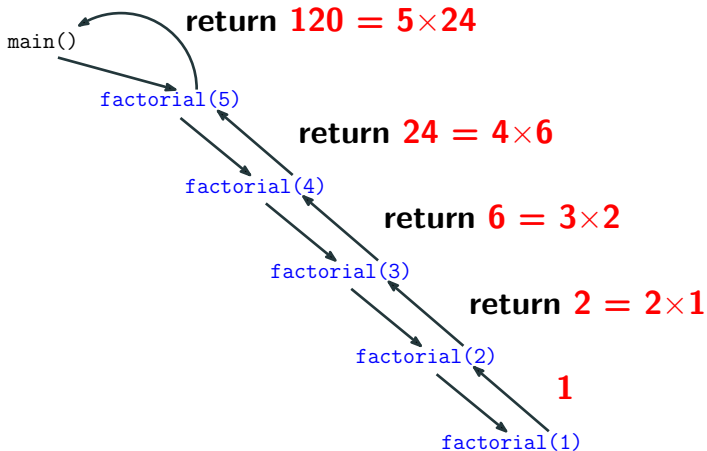
```
public class Factorial {
    public static long factorial(int n) {
        if( n < 0 )
            throw new IllegalArgumentException("n must non-negative!");
        else if( n == 0 || n == 1 ) // base cases
            return 1;
        else
            return n * factorial(n-1); // recursive call
    }

    public static void main(String[] args) {
        System.out.println( factorial(5) );
    }
}
```

Every recursive method contains the following two things:

- 1 **Base case(s)**. the case(s) for which we know how to calculate the answer without recursion; at least one base case is always required; every possible chain of recursive calls must eventually reach a base case.
- 2 **Recursive call(s)**. these are the calls to the current method. Each recursive call should be defined so that it makes progress towards a base case.

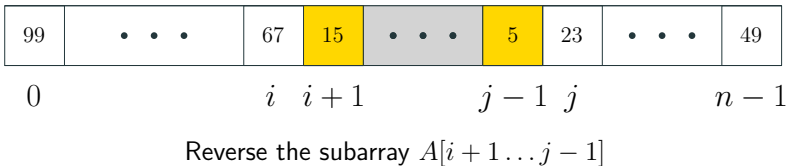
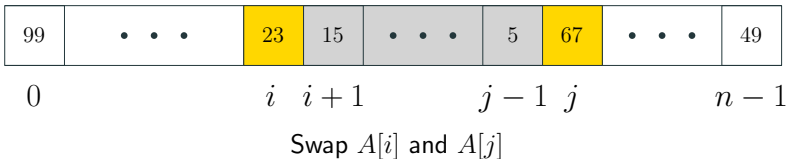
# Illustration



☞ The system uses a **stack** in the background to run recursive code

## Reversing an array

How to recursively reverse the subarray that starts at index  $i$  and ends at index  $j$ ?



# Reversing an array

```
import java.util.Arrays;

public class ReverseArray{
    public static void reverseArray(int[] A, int i, int j) {
        if (i > j)
            throw new IllegalArgumentException("i <= j is required.");

        int hold = A[i];
        A[i] = A[j];
        A[j] = hold;

        if( i + 1 < j - 1 )
            reverseArray(A, i + 1, j - 1); // recursive call
    }

    public static void main(String[] args) {
        int[] arr = {10, 20, 30, 40, 50};
        reverseArray(arr, 0, arr.length-1);
        System.out.print(Arrays.toString(arr));
    }
}
```



# Summing up an array

## Recursive idea

To add the numbers inside the subarray  $A[0]$  to  $A[i]$ , first **recursively** add the numbers inside the subarray  $A[0]$  to  $A[i-1]$  and then add the number  $A[i]$  to the result.

```
public class ArraySummer {  
  
    public static int add(int[] A, int i) {  
        if ( i < 0 )  
            throw new IllegalArgumentException("i should be non-negative.");  
        else if( i == 0 )  
            return A[0];  
        else  
            return add(A, i-1) + A[i]; // recursive call  
    }  
  
    public static void main(String[] args) {  
        int[] arr = {10, 20, 30, 40, 50};  
        System.out.print( add(arr,arr.length-1) );  
    }  
}
```

## Binary search

- Given a **sorted** array  $A$  of  $n$  items, how fast can you search a given element?
- One can search by scanning  $A$  from left to right (**linear search**), but this takes  $O(n)$  time
- Can we do it faster? Use the fact that the array is already sorted
- Yes, we can using binary search; runs in  $O(\log n)$  time

# Binary search

## Recursive algorithm (assumption: $A$ is sorted)

- If the target equals  $A[\text{mid}]$ , then we have found the target!
- If the target is less than  $A[\text{mid}]$ , search recursively in the left half
- Otherwise, search recursively in the right half

2	4	5	7	8	9	12	<b>14</b>	17	19	22	25	27	28	33	37
low							mid	high							
2	4	5	7	8	9	12	14	17	19	22	<b>25</b>	27	28	33	37
low								mid				high			
2	4	5	7	8	9	12	14	17	<b>19</b>	22	25	27	28	33	37
low				mid		high									
2	4	5	7	8	9	12	14	17	19	<b>22</b>	25	27	28	33	37
low = mid = high															

Searching for **22** in the array

# Code

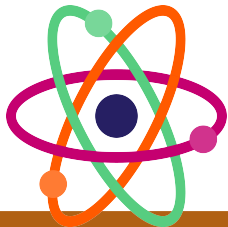
```
public class BinarySearch {
    public static boolean binarySearchRec(int[] A, int target, int low, int high) {
        if( low > high )
            return false;
        else {
            int mid = (low + high) / 2; // mid takes the floor of (low + high) / 2
            if( target == A[mid] )        return true;
            else if( target < A[mid] )    return binarySearchRec(A, target, low, mid - 1 ); // recursive call
            else                          return binarySearchRec(A, target, mid + 1, high ); // recursive call
        }
    }

    public static boolean binarySearch(int[] A, int target) {
        return binarySearchRec(A, target, 0, A.length-1);
    }

    public static void main(String[] args) {
        int[] A = {2,4,5,7,8,9,12,14,17,19,22,25,27,28,33,37};
        System.out.println(binarySearch(A,22)); // prints true; search successful
        System.out.println(binarySearch(A,21)); // prints false; search unsuccessful
    }
}
```

## Time complexity

- At every recursive call, approximately half of the array is discarded
- Also, at every recursive call, it does a constant amount of work –  $O(1)$
- Let  $m$  be the number of recursive calls made
- At every recursive call, array size gets halved
- After  $m$  recursive calls, array size equals  $n/2^m$
- In the worst case, we stop when  $n/2^m = 1 \implies 2^m = n$
- Taking log of both sides we obtain,  $m = \log_2 n = O(\log n)$
- **Time complexity.**  $O(\log n) \times O(1) = O(\log n)$



### Fun fact

Number of atoms in this universe:  $10^{80} \approx 2^{266}$

Even if we have a dataset as large as this, binary search will make just  $\log(2^{266}) = 266 \cdot \log_2 2 = 266 \cdot 1 = 266$  recursive calls in the worst case!

## Suggested exercise

Write a non-recursive (iterative) binary search

## Recursive string printer

For a given value of  $n$ , we need to print a string made up of  $n-1$  **comps**, **computing**, and  $n-1$  **tings**; here are few examples for you...

$n$	Output
1	computing
2	compcomputingting
3	compcompcomputingtingting
4	compcompcompcomputingtingtingting
5	compcompcompcompcomputingtingtingtingting

# Code

```
public class RecursiveStringPrinter {  
  
    public static String printer(int n) {  
        if( n <= 0)      return null;  
        else if( n == 1) return "computing";  
        else             return "comp" + printer(n-1) + "ting";  
    }  
  
    public static void main(String[] args) {  
        System.out.print(printer(5));  
    }  
}
```



# Self-referential classes

```
private static class Node<E> {  
    private E element;  
    private Node<E> prev, next; // defined recursively  
  
    // ...  
}
```

A **self-referential class** contains an instance variable that refers to another object of the same class type

# Using recursion for linked-lists

```
public class DoublyLinkedList<E> implements Iterable<E>{
    // other methods, variables, classes
    public String print() {
        return (printRecursive(head)).toString();
    }

    private StringBuilder printRecursive(Node<E> n) {
        if( n == null )
            return new StringBuilder();

        StringBuilder s = new StringBuilder(n.element.toString() + " ");
        s.append(printRecursive(n.next));
        return s;
    }
    // other methods, variables, classes
}
```

# Fractals

**What are fractals?** <https://en.wikipedia.org/wiki/Fractal>

Fascinating geometric figures that can be drawn recursively



Sierpiński triangle (source: Wikipedia)



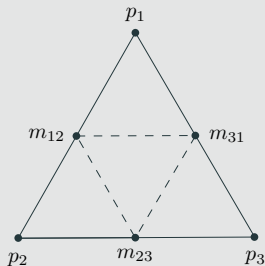
Wacław Sierpiński (source: Wikipedia)

# Pseudo-code



Sierpiński triangle (source: Wikipedia)

```
private static void drawTriangles(Graphics g, int d, Point p1, Point p2, Point p3) {  
    if (d == 0) { // depth is 0, draw the triangle; base case  
        Polygon P = new Polygon();  
        P.addPoint(p1.x,p1.y); P.addPoint(p2.x,p2.y); P.addPoint(p3.x,p3.y);  
        g.fillPolygon(P); // draws a filled triangle  
        return;  
    }  
  
    Point m12 = midpoint(p1,p2);  
    Point m23 = midpoint(p2,p3);  
    Point m31 = midpoint(p3,p1);  
  
    // Draw 3 Sierpinski triangles recursively of depth d-1  
    drawTriangles(g, d - 1, p1, m12, m31); // recursive call 1  
    drawTriangles(g, d - 1, m12, p2, m23); // recursive call 2  
    drawTriangles(g, d - 1, m31, m23, p3); // recursive call 3  
}
```



# Merge sort

# Merge sort

- Merge sort runs in  $O(n \log n)$  time
- It uses a linear-time algorithm known as **merging** for sorting the input
- Let us begin by understanding what is meant by merging two sequences ...

## Merging two sorted sequences

Given two **sorted** sequences  $S_1, S_2$ , how fast can you **merge** them into one final sorted sequence  $S$ ?

$S_1$	244	311	478
-------	-----	-----	-----

$S_2$	324	415	499	505	666
-------	-----	-----	-----	-----	-----

$S$	244	311	324	415	478	499	505	666
-----	-----	-----	-----	-----	-----	-----	-----	-----

Assume that  $S_1$  has  $k_1$  elements and  $S_2$  has  $k_2$  elements  
Clearly,  $S$  has  $k_1 + k_2$  elements

We need to do it in  $O(k_1 + k_2)$  time

## Merging two sorted sequences

$S_1$ 

244	311	478
-----	-----	-----

$S_2$ 

324	415	499	505	666
-----	-----	-----	-----	-----

$S$ 

--	--	--	--	--	--	--	--



## Merging two sorted sequences

$S_1$ 

<u>244</u>	311	478
------------	-----	-----

$S_2$ 

<u>324</u>	415	499	505	666
------------	-----	-----	-----	-----

$S$ 

--	--	--	--	--	--	--	--

## Merging two sorted sequences

$S_1$ 

244	<u>311</u>	478
-----	------------	-----

$S_2$ 

<u>324</u>	415	499	505	666
------------	-----	-----	-----	-----

$S$ 

244							
-----	--	--	--	--	--	--	--

## Merging two sorted sequences

$S_1$    244   311   478

$S_2$    324   415   499   505   666

$S$    244   311   

--	--	--	--	--	--

## Merging two sorted sequences

$S_1$    244   311   478

$S_2$    324   415   499   505   666

$S$    244   311   324   

--	--	--	--	--

## Merging two sorted sequences

$S_1$    244   311   478

$S_2$    324   415   499   505   666

$S$    244   311   324   415   

--	--	--	--

## Merging two sorted sequences

$S_1$ 

244	311	478
-----	-----	-----

$S_2$ 

324	415	<u>499</u>	505	666
-----	-----	------------	-----	-----

$S$ 

244	311	324	415	478			
-----	-----	-----	-----	-----	--	--	--

## Merging two sorted sequences

$S_1$ 

244	311	478
-----	-----	-----

$S_2$ 

324	415	499	<u>505</u>	666
-----	-----	-----	------------	-----

$S$ 

244	311	324	415	478	499		
-----	-----	-----	-----	-----	-----	--	--

## Merging two sorted sequences

$S_1$ 

244	311	478
-----	-----	-----

$S_2$ 

324	415	499	505	<u>666</u>
-----	-----	-----	-----	------------

$S$ 

244	311	324	415	478	499	505	
-----	-----	-----	-----	-----	-----	-----	--



## Merging two sorted sequences

$S_1$ 

244	311	478
-----	-----	-----

$S_2$ 

324	415	499	505	666
-----	-----	-----	-----	-----

$S$ 

244	311	324	415	478	499	505	666
-----	-----	-----	-----	-----	-----	-----	-----

Merging takes time proportional to the total number of blue cursors movements, and they move  $k_1 + k_2$  times to the right in total. Further, at each cursor movement, we spend  $O(1)$  time for comparing two elements, sending an element to  $S$ , and incrementing a blue pointer.

So, the time complexity amounts to  
$$(k_1 + k_2) \times O(1) = O(k_1 + k_2)$$

# Merge sort

- It is a recursive **divide and conquer** sorting algorithm
- Runs in  $O(n \log n)$  time (faster than Insertion, Bubble, and Selection sorts)

## The algorithm

Let the input be denoted by  $S$

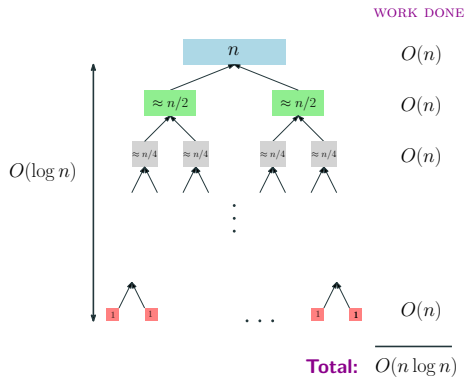
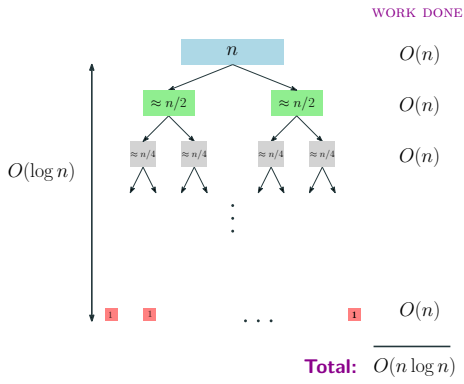
- 1 **Divide. Split** the array into two halves  $S_1, S_2$
- 2 **Conquer.**
  - 1 **Recursively** sort the left half  $S_1$
  - 2 **Recursively** sort the right half  $S_2$
- 3 **Combine. Merge** the two sorted halves  $S_1, S_2$  into  $S$

# Visualization

<https://opensa-server.cs.vt.edu/embed/mergesortAV>

Try: 85 24 63 45 17 31 96 50 67 88 11

# Time complexity of merge sort



Left: total time spent to create the two halves  $S_1, S_2$  at every recursive call;

Right: total time spent for merging.

➡ Merge sort runs in  $O(n \log n) + O(n \log n) = 2 \times O(n \log n) = O(n \log n)$  time

## Space complexity

**Space complexity** of an algorithm refers to the amount extra space the algorithm needs (apart from the input) for its execution.

- To find space complexity, focus on the additional defined data structures (arrays, stacks, queues, lists, etc.) whose sizes are dependent on  $n$ . For recursive code, also consider the stack depth of the call stack.
- Count the total number of data elements stored in those data structures in the worst case
- Let  $s$  be total number of such data elements
- Space complexity is  $O(s)$
- If no such data structures are used, space complexity is  $O(1)$  (constant amount of extra space is used)

## Examples

- The space complexity of the ExpressionChecker implementation is  $O(n)$  where  $n$  is the number of symbols since it uses a stack whose size is  $n$  in the worst case
- The space complexity of bubble sort/insertion sort/selection sort is  $O(1)$  since they use just a constant amount of extra space (size independent of  $n$ ) for maintaining a bunch of variables
- Let us say a method uses a doubly linked list having at most  $n$  nodes and a bunch of variables for processing. The space complexity of the method is  $O(n)$
- If a method uses a linked list of size  $n$  and a  $n \times n$  matrix of size  $n^2$ . The space complexity of the method amounts to  $O(n) + O(n^2) = O(n^2)$

# Space complexity of merge sort

## Merge sort

- For creating the two subsequences  $S_1, S_2$  we need  $O(n/2) + O(n/2) = O(n)$  extra space.
- So, the total amount of extra space needed by a series of recursive call from the root to a leaf of the recursion tree also amounts to  $O(n)$  since

$$O(n) + O\left(\frac{n}{2}\right) + O\left(\frac{n}{4}\right) + \dots + O(1) = O\left(n + \frac{n}{2} + \frac{n}{4} + \dots + 1\right) = O(2n) = O(n)$$

- For recursion, a stack is needed of size  $O(\log n)$
- Total space complexity:  $O(n) + O(\log n) = O(n)$



# The Comparable interface in Java

<https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Comparable.html>

## Why you should use the Comparable interface?

If we ever need to compare two objects of a class, there must be a comparison method for the class. This interface forces the class to define such a method if it is not already defined inside it. For the wrapper classes such as **Integer**, **Double**, **Character** etc. comparison methods are already defined. Generic sorting methods in Java use the comparison method for sorting by comparisons.

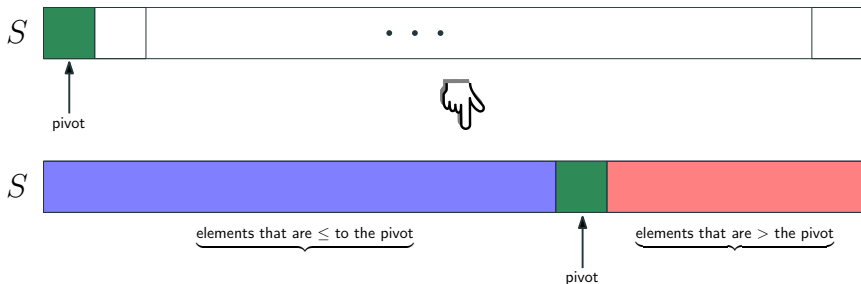
☞ The comparison method must be named **compareTo**, as declared inside the **Comparable** interface

☞ **obj1.compareTo(obj2) < 0** if obj1 is **less than** obj2;  
**obj1.compareTo(obj2) == 0** if obj1 is **equals** obj2;  
**obj1.compareTo(obj2) > 0** if obj1 is **greater than** obj2;

See the class `MergeSort`

# Quick sort

# Partitioning an array

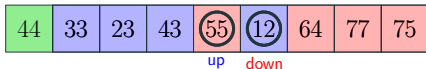
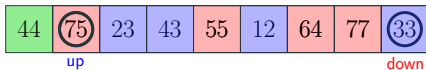
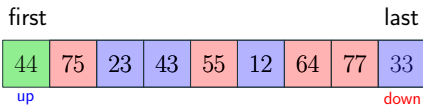


## The algorithm

Denoted the input by  $S$

- 1 Select the first element in  $S$ ; call it **pivot**
- 2 Find the elements in  $S$  that are less than equal to pivot and send them to the left part of  $S$  and the ones that are greater than the pivot to the right part of  $S$
- 3 Put the pivot at the appropriate location in  $S$ , meaning put it at the location where it would appear if  $S$  is sorted

## An example



## Pseudocode

- ①  $\text{pivot} = S[\text{first}]$ ,    $\text{up} = \text{first}$ ,    $\text{down} = \text{last}$
  - ② **do**
    - 2.1 Increment  $\text{up}$  until  $\text{up}$  selects the first element greater than the pivot value or  $\text{up}$  has reached  $\text{last}$
    - 2.2 Decrement  $\text{down}$  until  $\text{down}$  selects the first element less than or equal to the pivot value or  $\text{down}$  has reached  $\text{first}$
    - 2.3 if  $\text{up} < \text{down}$ , exchange  $S[\text{up}]$  and  $S[\text{down}]$
  - ③ **while**  $\text{up}$  is to the left of  $\text{down}$
  - ④ Exchange  $S[\text{first}]$  and  $S[\text{down}]$
- 👉 Note that the index  $\text{down}$  is a correct place for the pivot since in that case, everything to its left will be less than  $\text{pivot}$  and everything to its right will be greater than  $\text{pivot}$

# Partition

```
private static <K extends Comparable<K>> void swapTheItemsAt(K[] S, int i, int j) {
    K hold = S[i];
    S[i] = S[j];
    S[j] = hold;
}

private static <K extends Comparable<K>> int partition(K[] S, int first, int last) {
    K pivot = S[first];
    int up = first, down = last;

    do {
        while( (up < last) && (pivot.compareTo(S[up]) >= 0))
            up++;

        while( pivot.compareTo(S[down]) < 0)
            down--;

        if( up < down )
            swapTheItemsAt(S, up, down);
    }while(up < down);

    swapTheItemsAt(S, first, down);
    return down;
}
```

# Quick sort

- It is another divide and conquer sorting algorithm
- Runs in  $O(n^2)$  time (explained next)

## The algorithm

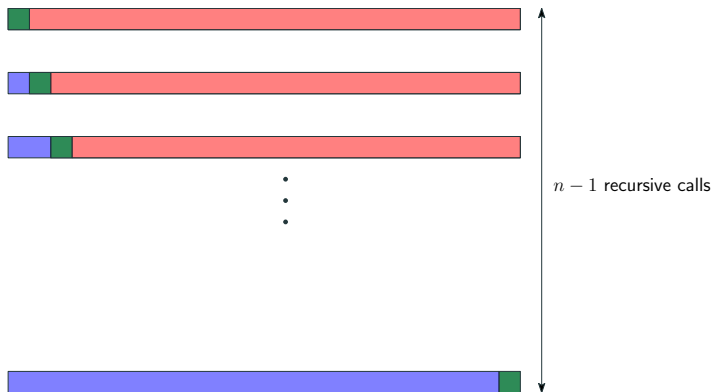
Let the input be denoted by  $S[\text{first}, \dots, \text{last}]$

- 1 **Divide. Partition** the array so that the pivot item reaches its correct place in the array (its index is `pivIndex`)
- 2 **Conquer.**
  - 1 **Recursively** sort the subarray  $\text{first}, \dots, \text{pivIndex}-1$  (the subarray to the left of pivot)
  - 2 **Recursively** sort the subarray  $\text{pivIndex}+1, \dots, \text{last}$  (the subarray to the right of pivot)



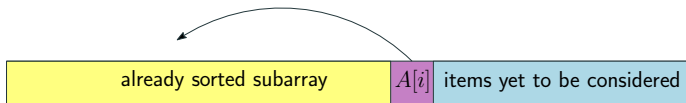
See the class `QuickSort`

## Time and space complexities of quick sort



- When the array is sorted, at every recursive call, we find that all the other elements are bigger than the pivot! This is the worst case in fact
- So, we make  $n - 1 = O(n)$  recursive calls; we spend  $O(n)$  time for partitioning at every level
- Total time taken  $O(n^2)$
- Space complexity:  $O(n)$  since recursion depth can be at most  $n - 1$

## Speed comparison



### Insertion sort, needed for comparison; runs in $O(n^2)$ time

**Input:** An array  $A$  of  $n$  comparable elements

for  $i = 1$  to  $n - 1$  do

    Insert  $A[i]$  at the proper spot within the sorted subarray  $A[0], A[1], \dots, A[i];$

☞ If  $A$  is already sorted then every  $A[i]$  is already in its correct position. As a result, every iteration of the for-loop runs in  $O(1)$  time. Consequently, insertion sort takes  $O(n)$  time when  $A$  is already sorted.

**Applet.** <https://visualgo.net/en/sorting>

Now see the class [SortingSpeedComparison](#)

## Quick sort vs Merge sort, output in some run, $n = 1M$

$n = 1,000,000$

QuickSort ( $O(n^2)$ ): 624 ms

MergeSort ( $O(n \log n)$ ): 1264 ms

**AWESOME!**

Quick sort **beats** merge sort in practice on randomly ordered arrays despite having worse time complexity!

## Merge sort vs. Quick sort vs. Insertion sort

$n = 100,000$

QuickSort ( $O(n^2)$ ): 85 ms

MergeSort ( $O(n \log n)$ ): 124 ms

InsertionSort ( $O(n^2)$ ): 19002 ms

*Insertion sort is unusable in general when  $n$  is large. However, if the input was already sorted, it could finish in just 10 ms when  $n = 100K$ . In such cases, every iteration of the for-loop takes  $O(1)$  time since  $A[i]$  is already at its correct spot! As result, insertion sort takes  $O(n)$  time on sorted inputs.*

## Quick sort performs terribly when the input is already sorted!

### Output, $n = 10K$

$n = 10,000$

QuickSort ( $O(n^2)$ ) on a random array: 6 ms

QuickSort on a sorted array: 200 ms

👉 *When the input is sorted, quick sort runs in quadratic time*

## What happens when the input size is 50,000?

`n = 50,000`

```
Exception in thread "main" java.lang.StackOverflowError
  at recursion.QuickSort.recurseAndSort(QuickSort.java:12)
  at recursion.QuickSort.recurseAndSort(QuickSort.java:13)
  at recursion.QuickSort.recurseAndSort(QuickSort.java:13)
  at recursion.QuickSort.recurseAndSort(QuickSort.java:13)
  at recursion.QuickSort.recurseAndSort(QuickSort.java:13)
  at recursion.QuickSort.recurseAndSort(QuickSort.java:13)
  at recursion.QuickSort.recurseAndSort(QuickSort.java:13)
  .
  .
  .
```

👉 *The call stack runs out of space since quick sort strives to make around 50K calls in this case!*

# Avoiding StackOverflowError exception

## How to avoid StackOverflowError exception?

- Make quick sort non-recursive by using a stack explicitly

```
• LinkedStack<Pair> stack = new LinkedStack<>();  
  stack.push( new Pair(0,S.length-1) ); // push the left and right indices of the source array  
  
  while( !stack.isEmpty() ) {  
    var currentPair = stack.pop(); // pop a pair from the stack  
  
    if( currentPair.leftIndex >= currentPair.rightIndex ) continue;  
  
    int pivotIndex = partition(S, currentPair.leftIndex, currentPair.rightIndex);  
    stack.push( new Pair(currentPair.leftIndex, pivotIndex - 1) ); // push the left and right indices of the left chunk  
    stack.push( new Pair(pivotIndex+1, currentPair.rightIndex) ); // push the left and right indices of the right chunk  
  }
```

- See the class [NonRecursiveQuickSort](#)
- The non-recursive version is still **slow** on **sorted** inputs but unlike the recursive version, it doesn't crash by throwing a StackOverflowError exception
- Now, [NonRecursiveQuickSort](#) can sort a 50K-sized **sorted** input in 2608 ms



# How to 'almost' avoid quadratic runtime in practice?

## Speeding up quick sort in practice

- Quick sort slows down on sorted inputs because of the bad pivots which generate empty left chunks (all the other elements goes to the right chunks)
- ☞ **Idea.** choose pivots **randomly** instead of sticking to the first element every time
- It will be then **unlikely** a bad pivot is chosen every time a partition is executed
- It can be shown theoretically that this small change will result in  $O(n \log n)$  behavior in practice (proof is out of scope)
- See the class [RandomizedQuicksort](#) (**recursive**)
- ```
Random generator = new Random();  
int pivotIndex = generator.nextInt(first, last + 1); // generate a random index for pivot selection  
K pivot = S[pivotIndex];
```
- The randomized **recursive** version can sort a 50K-sized sorted input in well under 30 ms (previously it took 2608 ms!)

## Avoiding StackOverflowError exceptions

- Although very unlikely, bad pivots can still be chosen resulting in StackOverflowError exceptions since `RandomizedQuicksort` is recursive
- **Solution.** make it non-recursive
- See the class `NonRecursiveRandomizedQuickSort`
- No more stack overflows and painful slowdowns on sorted datasets are unlikely!

# Making quick sort faster in practice

## Median of three heuristic

Use the median of the three items  $S[\text{first}]$ ,  $S[(\text{first}+\text{last})/2]$ ,  $S[\text{last}]$  as the pivot. In this case, median is the second item of the sorted sequence of the above three items.

- In the randomized version, we choose pivots randomly
- Finding a pivot using random number generator is **slower** than computing the median of the above three items since only comparison operators and swapping can be used to select the median

```
int mid = (first + last)/2;
if (S[last].compareTo( S[first] ) < 0)    swapTheItemsAt(S, first, last);
if (S[mid].compareTo( S[first] ) < 0)    swapTheItemsAt(S, mid, first);
if (S[last].compareTo( S[mid] ) < 0)    swapTheItemsAt(S, last, mid);

K pivot = S[mid]; // S[mid] is now the pivot since the median item is now present at S[mid]
```

- Also, this results in careful selection of pivots in practice
- Consequently, quick sort runs surprisingly faster!

## Further optimization

- Insertion sort works very fast for small inputs
- We leverage insertion sort in the algorithm
- When array size is at most 50 (other small numbers may work as well), do not partition anymore, use insertion sort instead

```
if( (currentPair.rightIndex - pivotIndex + 1) > 50)
    stack.push(new Pair(pivotIndex, currentPair.rightIndex));
else
    insertionSort(S,pivotIndex,currentPair.rightIndex);

if( (pivotIndex - currentPair.leftIndex) > 50)
    stack.push(new Pair(currentPair.leftIndex, pivotIndex - 1));
else
    insertionSort(S,currentPair.leftIndex,pivotIndex-1);
```

See the class [MedOfThreeNonRecQuickSort](#)

## Demonstration

### $n = 10,000,000$ , randomly generated integer array

$n = 10,000,000$

NonRecursiveRandomizedQuickSort ( $O(n \log n)$  behavior expected): 6723 ms

MedOfThreeNonRecQuickSort ( $O(n \log n)$  behavior expected): 4971 ms

MergeSort ( $O(n \log n)$ ): 5195 ms

Arrays.sort() ( $O(n \log n)$ ): 5435 ms

### $n = 10,000,000$ , sorted integer array

$n = 10,000,000$

NonRecursiveRandomizedQuickSort ( $O(n \log n)$  behavior expected): 3654 ms

MedOfThreeNonRecQuickSort ( $O(n \log n)$  behavior expected): 1498 ms

MergeSort ( $O(n \log n)$ ): 4411 ms

Arrays.sort() ( $O(n \log n)$ ): 161 ms

👉 **Moral of the story.** Arrays.sort() is hard to beat in general

## Using multiple cores on your machine to sort faster

$n = 10,000,000$ , **randomly generated integer array**

$n = 10,000,000$

`Arrays.sort()` ( $O(n \log n)$ ): 5564 ms

`Arrays.parallelSort()` ( $O(n \log n)$ ): 488 ms

Implement the selection sort algorithm using the  
Comparable interface

[https://en.wikipedia.org/wiki/Selection\\_sort](https://en.wikipedia.org/wiki/Selection_sort)

## Timsort (optional, for algorithm lovers only)

Java uses **Timsort** for sorting an array of non-primitives  
<https://en.wikipedia.org/wiki/Timsort>



# Generic binary search

```
public class GenericBinarySearch {  
    private static <K extends Comparable<K>> boolean binarySearchRec(K[] A, K target, int low, int high) {  
        if( low > high )  
            return false;  
        else {  
            int mid = (low + high) / 2; // mid takes the floor of (low + high) / 2  
            if( target.equals(A[mid]) )        return true;  
            else if( target.compareTo( A[mid] ) < 0 ) return binarySearchRec(A, target, low, mid - 1 ); // recursive call  
            else return binarySearchRec(A, target, mid + 1, high ); // recursive call  
        }  
    }  
  
    public static <K extends Comparable<K>> boolean search(K[] A, K target) {  
        return binarySearchRec(A, target, 0, A.length-1);  
    }  
}
```

The Comparable interface also helps us to implement a generic binary search

## Recursion tips

- Make sure every chain of recursive calls eventually reach at least one base case
- Long chains of recursive calls can throw **StackOverflowError**; be careful!
- If such long chains cannot be avoided, make your code iterative (non-recursive)

# Reading

<https://opensa-server.cs.vt.edu/ODSA/Books/Everything/html/RecIntro.html>