Recursion

Dr. Anirban Ghosh

School of Computing
University of North Florida

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:

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

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

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

```java
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

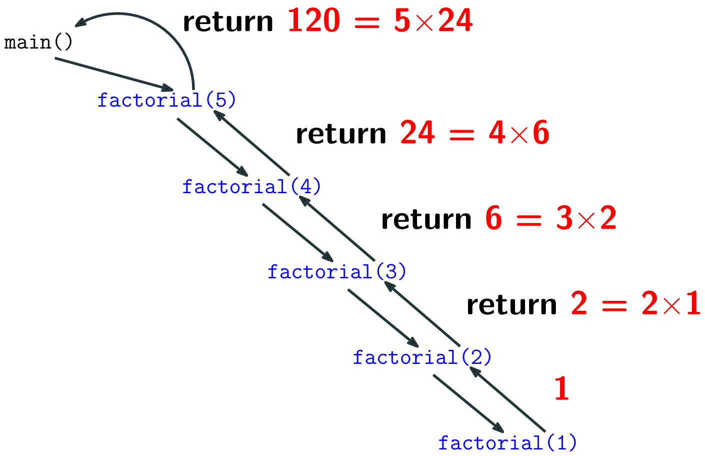
```
public class Factorial {
   public static long factorial(int n) {
      if( n < 0 )
         throw new IllegalArgumentException("n must non-negative!");
      else if( n == 0 || n == 1 ) // base cases
         return 1;
      else
         return n * factorial(n-1); // recursive call
   }

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

Every recursive method contains the following two things:

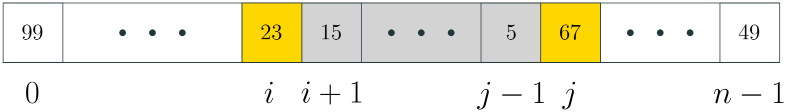
1. **Base case(s)**. the case(s) for which we know how to calculate the answer without recursion; at least one base case is always required; every possible chain of recursive calls must eventually reach a base case.
2. **Recursive call(s)**. these are the calls to the current method. Each recursive call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

Illustration

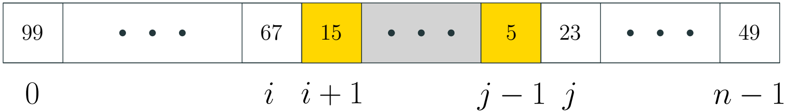


☞ The system uses a **stack** in the background to run recursive code

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



Swap $A[i]$ and $A[j]$



Reverse the subarray $A[i+1 \ldots j-1]$

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

```
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 A[i] + add(A, i-1); // recursive call
   }

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

☞ This is quite similar to the student counting example shown earlier

- 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 (asssumption: 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
- Othewise, search recursively in the right half

2	4	5	7	8	9	12	**14**	17	19	22	25	27	28	33	37
low							mid								high

2	4	5	7	8	9	12	14	17	19	22	**25**	27	28	33	37
								low			mid				high

2	4	5	7	8	9	12	14	17	**19**	22	25	27	28	33	37
								low	mid	high					

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

Searching for **22** in the array

Code

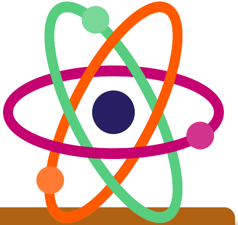
```java
public class BinarySearch {
   public static boolean binarySearchRec(int[] A, int target, int low, int high) {
      if( low > high )
         return false;
      else {
         int mid = (low + high) / 2; // mid takes the floor of (low + high) / 2
         if( target == A[mid] )         return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

Time complexity

- At every recursive call, we discard approximately half of the array
- Also, at every recursive call, we do 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!

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 $n-1$ **comp**s, **computing**, and $n-1$ **ting**s; here are few examples for you...

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

Code

```java
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

```java
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

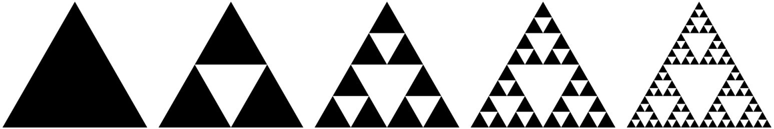
```
public class DoublyLinkedList<E> implements Iterable<E>{
   // other methods, variables, classes
   public String print() {
      return (printRecursive(head)).toString();
   }

   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
}
```

What are these?

Fascinating geometric figures that can be drawn recursively



Sierpiński triangle (source: Wikipedia)

Wacław Sierpiński (source: Wikipedia)

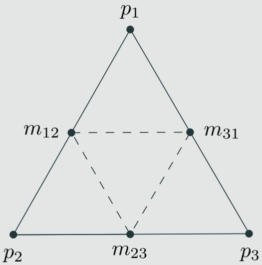
Pseudo-code

Sierpiński triangle (source: Wikipedia)

```java
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);
      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

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	244	311	478					
S_2	324	415	499	505	666			
S								

Merging two sorted sequences

S_1	244	311	478					
S_2	324	415	499	505	666			
S	244							

Merging two sorted sequences

S_1	244	311	<u>478</u>					
S_2	<u>324</u>	415	499	505	666			
S	244	311						

Merging two sorted sequences

S_1	244	311	<u>478</u>					
S_2	324	<u>415</u>	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	499	505	666			
S	244	311	324	415	478			

Merging two sorted sequences

S_1	244	311	478					
S_2	324	415	499	505	666			
S	244	311	324	415	478	499		

Merging two sorted sequences

S_1	244	311	478					
S_2	324	415	499	505	666			
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

How much work are we doing to merge?

We are working proportional to the total number of times the two blue cursors moved

So the time complexity is $O(k_1 + k_2)$

- 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

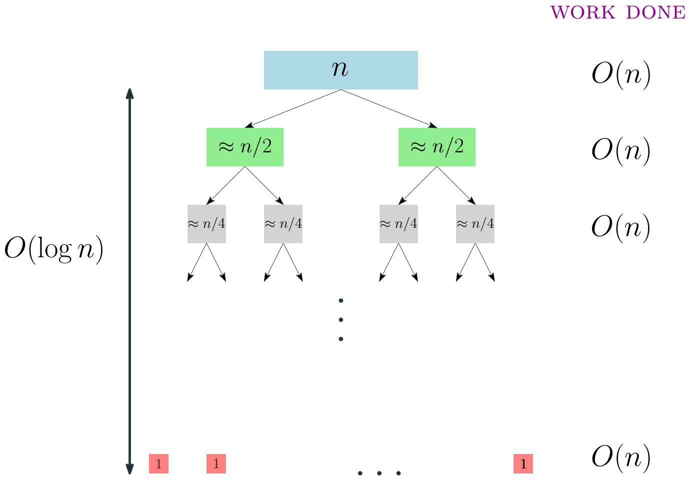
Let the input be denoted by S

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

`https://opendsa-server.cs.vt.edu/embed/mergesortAV`

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

Time complexity of merge sort



Total: $O(n \log n)$

Space complexity

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 `ExpresssionChecker` 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 is $O(1)$ since it uses a constant amount of extra space for maintaining a bunch for variables

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. Further, it can be shown that the total amount of extra space needed by a series of recursive call from the root to a leaf amounts to $O(n)$ as well.
- 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 to use it?

If we 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. For the wrapper classes such as **`Integer`**, **`Double`**, **`Character`** etc. this is already defined.

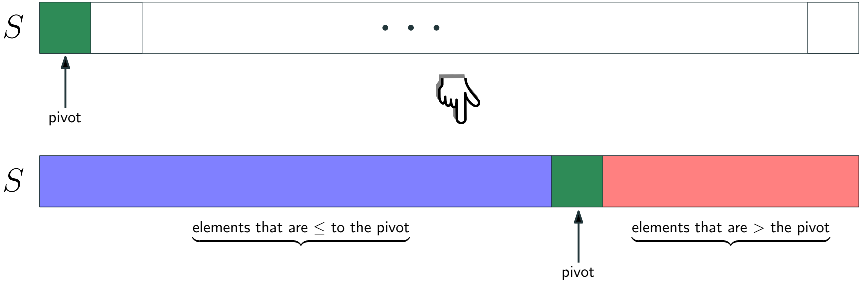
☞ The comparison method must be named as **`compareTo`**, as enforced by the **`Comparable`** interface

☞ **`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

first								last
44	75	23	43	55	12	64	77	33
up								down

44	75	23	43	55	12	64	77	33
	up							down

44	33	23	43	55	12	64	77	75
				up	down			

44	33	23	43	12	55	64	77	75
				down	up			

12	33	23	43	44	55	64	77	75
				down				

Pseudocode

1. $\mathtt{pivot} = S[\mathtt{first}]$, $\quad \mathtt{up} = \mathtt{first}$, $\quad \mathtt{down} = \mathtt{last}$
2. **do**
   - 2.1 Increment up until up selects the first element greater than the pivot value or up has reached **last**
   - 2.2 Decrement down untill down selects the first element less than or equal to the pivot value or down has reached **first**
   - 2.3 if $\mathtt{up} < \mathtt{down}$, exchange $S[\mathtt{up}]$ and $S[\mathtt{down}]$
3. **while** up is to the left of down
4. Exchange $S[\mathtt{first}]$ and $S[\mathtt{down}]$

Partition

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

public 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;
}
```

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

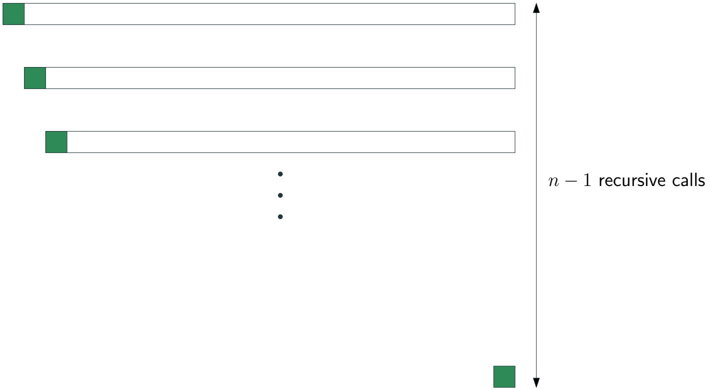
The algorithm

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

1. **Divide**. **Partition** the array so that the pivot value is in its correct place (its index is `pivIndex`)
2. **Conquer**.
   1. **Recursively** sort the subarray $\texttt{first}, \ldots, \texttt{pivIndex-1}$
   2. **Recursively** sort the subarray $\texttt{pivIndex+1}, \ldots, \texttt{last}$
3. **Combine**. The two answers from the two recursive calls are trivially combined with the pivot value to form the final sorted sequence.

See the class `QuickSort`

Time complexity



- 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 paritioning 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 $A[0], A[1], \ldots, A[i]$;

`https://visualgo.net/en/sorting`

Now see the class `SortingSpeedComparison`

Quick sort vs Merge sort, output in some run, $n = 50K$

```
Time taken by QuickSort (O(n^2)): 27 ms
Time taken by MergeSort (O(n*log n)): 40 ms
```

☞ *Quick sort could beat merge sort despite having worse time complexity*

Quick sort performs terribly when the input is already sorted!

Output, $n = 10K$

```
Time taken by QuickSort (O(n^2)) on a random array: 6 ms
Time taken by 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?

```
Time taken by QuickSort (O(n^2)): 28 ms
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)
  .
  .
  .
```

How to avoid `StackOverflowError` exception?

Make it non-recursive using stack ...

See the class `NonRecursiveQuickSort`

How to avoid quadratic runtime in practice?

Choose the pivot randomly ...

This small change exhibits $O(n \log n)$ behavior in practice

And, quadratic runtimes are extremely unlikely

See the class `RandomizedQuicksort`

Then make it non recursive ...

See the class `NonRandomizedQuicksort`

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://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/RecIntro.html