1 Functions

We know that all the code that we want to execute has to be written within the 'main' body. Now assume that we want to check for two numbers if they are prime or not. For that we need to write the code for prime twice in the same program.

```
public static void main(String args[]) {

	Scanner scn = new Scanner(System.in);

	int num1, num2;

	num1 = scn.nextInt();
	num2 = scn.nextInt();

	boolean flag1 = true, flag2 = true;

	// checking prime for number 1
	for (int i = 2; i<num1; i++) {
		if (num1 % i == 0) {
			flag1 = false;
			break;
		}
	}

	// checking prime for number 2
	for (inti = 2; i<num2; i++) {
		if (num2 % i == 0) {
			flag2 = false;
			break;
		}
	}

	if (flag1)
		System.out.println(num1 + " is prime");
	else
		System.out.println(num1 + " is not prime");

	if (flag2)
		System.out.println(num2 + " is prime");
	else
		System.out.println(num2 + " is not prime");
}
```

What are the visible problems with this code?

- **Repetition of Logic:** The logic for checking prime for first number is same as that for second number.
- **Error Correction:** If there is any error in the logic for checking prime, then we will have to correct it twice.

This is when Functions come into play. Functions help in reducing the redundancy in codes and considerably reduce the lines of code (LOC), there by increasing modularity.

A function is a set of code which is referred to by a name and can be called (invoked) at any point in the program by using the function's name. It is like a subprogram where we can pass parameters to it and it can even return a value. All functions are written outside of the main body and can be called from the main body or even other functions. Thus, our main function doesn't get cluttered with all the code, improving readability. In fact 'main' is also a function, a special function which is called by the JVM at the runtime.

Basic Syntax for a Function :

```
// public static returnType nameOfFunction(Parameters)
public static int func() {
	// do your work here
	// return is compulsory
	return 0;
}
public static void func() {
	// do your work here
	// return is not compulsory
	return;
 }
```

For now, begin all your functions with the keywords 'public static' till we understand their significance in detail. After that the return type of the function is specified.

By return type we mean that we specify the data type of the return value. It may also happen sometimes that the function does not return anything. In that case the return type of the function is **'**void**'** and it is not compulsory to write the 'return' keyword in this case. After the return type we write the name of the function followed by two parentheses where we give the parameters to the function. We enclose the entire body of the function within two curly braces. Inside that we can write the function body. Generally the last statement in the function is the return statement.

Now just making the function would not execute it when we run the program. To execute any function we need to invoke it. This is called calling the function. Function can be called from the main function and also from other functions. If there are any parameters to be given to the function,they are also passed within the parenthesis. If the function returns any value it can be stored in a variable. Now let us write two functions and see how they can be called!

```
public static void main(String[] args) {
        System.out.println("this is main");
        // Calling functions from the main
        func1();
        func2();
}

public static void func1() {
        System.out.println("this is func1");
}

public static void func2() {
        System.out.println("this is func2");
        // Calling another function inside a function
        func1();
}
```

Output :

this is main
this is func1
this is func2
this is func1

Function with Parameter:

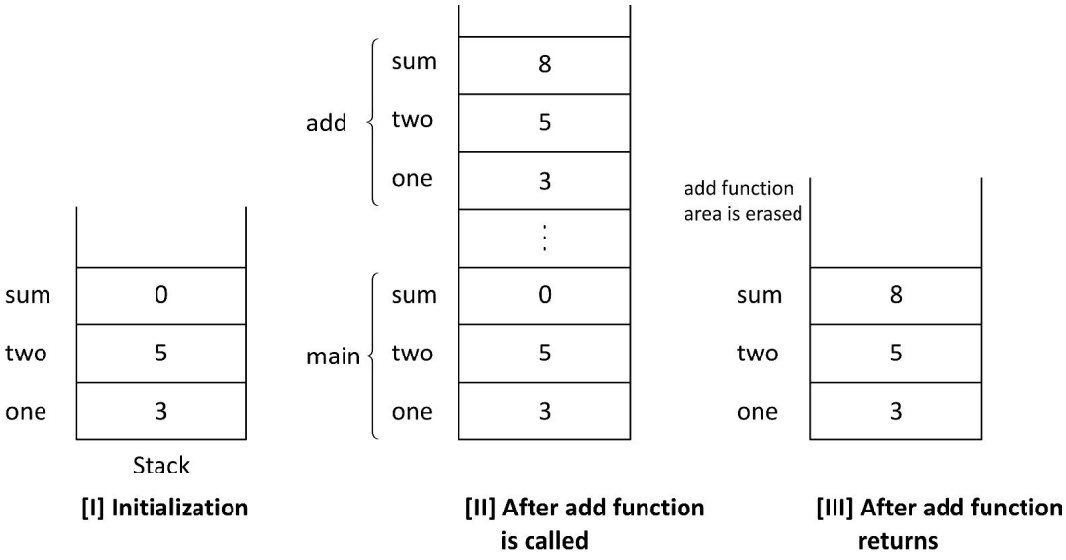
We can also pass parameters to the functions. When we are defining the function, we specify the data type of the parameter followed by a name of the parameter within the parenthesis. This parameter would be referred by that name within the function body. We can pass as many parameters as we want. When the function is called, we need to pass the parameters of the same data type as specified in its definition. Note that while calling the function we only pass in the names of the variables or values and don't specify its data type.

```
public static void main(String[] args) {
        intone = 3, two = 5, sum;
        sum = add(one, two);
        System.out.println(sum + " is the sum");
        // OUTPUT: 8 is the sum
}
public static int add(int one, int two) {
        int sum;
        sum = one + two;
        return sum;
}
```

In the previous code, we have written a function for adding two numbers. An important point to note here is that the variables present in the main function are different from those that are present in the add function, though they have the same names. To understand this better let us learn about the concept of variables and their scope.

Variables and Scope:

During runtime a memory pool is created by the JVM to store all the variables, codes for functions, etc. In this memory pool there are majorly two types of memory – the **stack memory** and the **heap memory**. Note that the values of all the primitive data types are stored on the stack whereas the non primitive data types are made on the heap and their reference is stored on the stack. All the functions that are called occupy their area on the stack where all their variables are stored. The functions keep stacking one above the other in the order of their calls. When the function returns, it is erased from the stack and all its local variables are destroyed.



The variables that are present in one function are local to only that function. It cannot be accessed outside that function. For instance a variable made in the main function is not available to the add function unless we pass it as a parameter to it. Even then only the value would be passed (if the variable is of primitive data type) and that would be stored in another variable which is local to the add function. Thus in the last code, the 'one' and 'two' variables in the main function are different from the 'one' and 'two' variables of the add function. So any changes to the variables in the add function would not alter the values of the variables in the main function.

But this is always not the case when the data type is non primitive, the reference to the actual variable is stored on heap. Now when this variable is passed onto a function, the reference stored on the stack would be passed. Thus the variable in the function would have the same reference as that in the main function and both would point to the same variable present on heap. Thus if the function alters the value of the variable on the heap, it will be reflected in the variable of the main function also.

Now let us talk about another type of variable which no matter what is always made on the heap.It is the global variable. Global variable is the variable which is available to all the functions of the program. It is not local to any function and thus its value cannot be stored on the stack. If its value had to be stored on the stack then it would have been local to that function in whose area it is stored. As it local to none, thus it is stored on the heap memory and can be accessed by all the functions.

Let us now modify the program of checking two numbers whether they are prime or not using functions.

```
public static void main(String[] args) {
	Scanner scn = new Scanner(System.in);
	int num1, num2;
	num1 = scn.nextInt();
	num2 = scn.nextInt();

	// function call
	if (isPrime(num1))
		System.out.println(num1 + " is prime");
	else
		System.out.println(num1 + " is not prime");

	// function call
	if (isPrime(num2))
		System.out.println(num2 + " is prime");
	else
		System.out.println(num2 + " is not prime");

}

// function definition
public static boolean isPrime(intnum) {

	// checking prime
	for (int i = 2; i<num; i++) {
		if (num % i == 0) {
			return false;
		}
	}
	return true;
}
```

As you can see here, the code for checking prime is written only once but used twice, first for num1 and then for num2.

Here, the function declaration comprised of several components:

- Return type: Boolean
- Function name: isPrime
- Number of arguments: 1
- Arguments: a variable of int type

In general, method declaration has 5 components:

(a) Modifier : Defines **access type** of the method i.e. from where it can be accessed in your application. In Java, there 4 type of the access specifiers.

- Public: accessible in all the classes in your application.
- Protected: accessible within the class in which it is defined and in its **subclass(es)**.
- Private: accessible only within the class in which it is defined.
- Default (declared/defined without using any modifier): accessible within the package in which its class is defined.

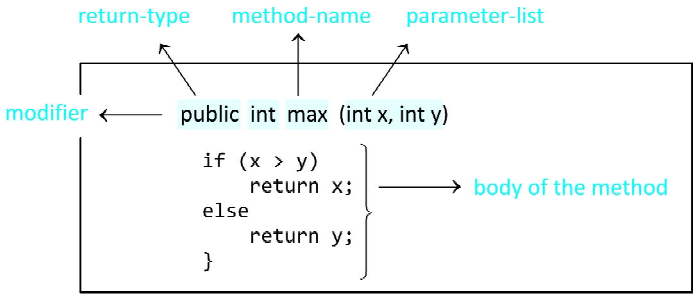
(b) Return Type: The data type of the value returned by the method or void if does not return any value.

(c) Method Name : the rules for field names apply to method names as well, but the convention is a little different.

(d) Parameter List : Comma separated list of the input parameters are defined, preceded with their data type, within the enclosed parenthesis. If there are no parameters, empty parentheses () are used.

(e) Exception List : The exceptions you expect that the method can throw, you can specify these exception(s).

(f) Method Body : it is enclosed between braces. The code that needs to be executed to perform your intended operations.



PRACTICE QUESTIONS:

1. Given lower limit and upper limit, write a function to print all primes within that range.

   Input: 2 8

   Output: 2 3 5 7

2. Given lower limit and upper limit , Write a function to print all the Armstrong numbers within that range.
   A positive integer of **n digits** is called an Armstrong number of **order n** (order is number of digits) if abcd... = pow(a,n) + pow(b,n) + pow(c,n) + pow(d,n) + ...153 is an Armstrong number because 1*1*1 + 5*5*5 + 3*3*3 = 153

Input: 1 1000

Output: 1 153 370 371 407

3. Write a function which returns Log of X to the base N. Print the value returned (without using inbuilt function).

   Input: 100010

   Output: 3

4. Write a function that prints the Nth fibonacci.

   Input: 7

   Output: 13

   Explanation: 0 1 1 2 3 5 8 13 where 0 is 0^{th} Fibonacci, 1 is 1^{st} and so on.

2 Arrays

Until now, we have learnt that the only way to store values is using variables. But let us suppose that we now want to store the marks of 30 students of a class and later calculate their average. Do you think creating variables for 30 students is a feasible task?

In such cases we would need Arrays.

Java provides a data structure, array, which stores a **fixed-size sequential collection of elements of the same type**.

An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as marks1, marks2, ..., and marks30, we declare one array variable such as **marks** and use marks[0], marks[1], and ..., marks[29] to represent individual variables.

Syntax:

1. Declaration of an array

```
    datatype[] arr; // Preferred way
    datatype arr[]; // Works too but not preferred way

 At the time of multiple declaration they convey different meaning to the compiler.
    // In this statement arr is declared as an array but b is declared as a variable
    int arr[], b;
    // In this statement arr and b both are declared as an array
        int[] arr, b;
```

2. Instantiation of an array

```
    array_reference_variable = new datatype[size];
    arr = newint[10]; // Array of name arr is instantiated with 10 elements
    int[] arr = new int[5]; // Declared an int array called arr with 5 elements.
    long[] arr = new long[10]; // Declared a long array called arr with 10 elements.
    double[] arr = new double[7]; // Declared a double array called arr with 7 elements.
    int[] arr = { 1, 2, 3, 4 }; // Initialization at the time of declaration.
    We can also provide the size of the array at the runtime. Eg:
    int n = scn.next Int(); // Taking input of an int type variable n
    int[] arr = new int[n]; // Here, size of the array is provided at the run time
```

In Java, when an array is declared, all the elements are initialized with the default value of that datatype. For int, the default value is 0.

You can refer to an element of an array via its index (or subscript) enclosed within square brackets []. The array index's begins with 0. Let's take an example.

```
public static void main(String[] args) {
	// an array arr of size 5 declared and initialized with 0 at all indexes
	int[] arr = new int[5];

	// values are updated at various indexes
	arr[0] = 10;
	arr[1] = 20;
	arr[2] = 30;
	arr[3] = 40;
	arr[4] = 50;

	// accessing the values from a particular index
	System.out.println(arr[0]);
	System.out.println(arr[3]);
}
```

To create an array, you need to know the length (or size) of the array in advance and allocate accordingly. Once an array is created, its length is fixed and cannot be changed. At times, it is hard to ascertain the length of an array. Nonetheless, you need to estimate the length and allocate an upper bound. This is probably the major drawback of using an array. Java has an ArrayList class which supports dynamic resizable arrays.

If we want to know the size of the array or the total number of elements in the array we can use the **array.length** property and use it on the name of the array.

```
public class Main {

public static void main(String[] args) {
	int[] arr = { 1, 3, 9, 8, 11, 12 };

	// Accessing each element with its property
	for (int num = 0; num<arr.length; num++) {
		// Printing each element
		System.out.print(arr[num] + " ");
	}
    }
}
```

Output: 1 3 9 8 11 12

Java perform array index-bound check. In other words, if the index being accessed is beyond the array's bounds, it issues a warning/error like (java.lang.ArrayIndexOutOfBoundException).

```
public class Main {

public static void main(String[] args) {

      int[] arr = new int[5];

      // 0 is inside the bound of the array
      arr[0] = 10;

      // 6 is not in the bound of the array (will show an error)
      arr[6] = 20;

      // Printing 0th element of the array viz. 10
      System.out.println(arr[0]);

      // Java checks both of the bounds (upper bound and lower bound)
      // Again it will show an error message
      System.out.println(arr[-2]);

   }
}
```

Output:

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 6 at Main.main(Main.java:11)

Array and Loop

It would be quite difficult to process each of the array's elements individually as we have done in the above example. Since these are repetitive tasks, looping statements like **for** and **while** come to our rescue.

Iterating over an array means accessing each element of array one by one. There are many ways of iterating over an array in Java. The common approach is to use a **for** loop with an iterator used to keep track of the index number and iterate through the entire array.

```
// To print the sum of all the marks (maximum marks 100)
 public class Main {

 public static voidmain(String[] args) {

 int[] marks = { 10, 20, 30, 40, 50, 60 };

 int sum = 0;
 for (int i = 0; i< 6; i++) {
    sum = sum + marks[i]; // adding each element of the array to the sum variable
 }
       System.out.println("Sum is: " + sum); // printing the total sum
}
}
```

Output:

Sum is: 210

Enhanced For Loop

In addition to the conventional for-loop, we also have an enhanced for-loop where the iteration variable in the enhanced for loop receives every element of an array one at a time starting from first element to last element. In the first iteration, it gets the first element. In the second iteration, it gets the second element and so on. Thus it iterates over all elements of an array. The type of iteration variable must be compatible with the type of array or collection. Note that the iteration variable in the conventional for-loop received indexes and not the actual elements.

```
for (datatype_of_each_elementref_name :array_ref_name) {
// accessing
}
// testing theenhanced for loop
class Main {

public static voidmain(String[] args) {

       int[] numbers = { 11, 22, 33, 44, 55 };

       for (int num :numbers) { // Enhanced For Loop
              System.out.print(num + " ");
       }
  }
}
```

Output:

11 22 33 44 55

Limitations of Enhanced For Loop

1. The traversal must start with the first element and visit the successor (One cannot start from any desired index like in the normal for-loop).
2. The traversal visits every element unless the programmer breaks or returns.
3. One cannot modify the elements of the array using enhanced for loop.

```
public classArrayDemo {
       public static voidmain(String[] args) {
              int[] arr; // array declaration
              arr = new int[5]; // space is alloted to 5 integers in heap
              System.out.println(arr); // prints base address like [I@7852e922
              System.out.println(arr[0]); // prints 1st element viz 0
              System.out.println(arr[1]); // prints 2nd element viz 0
              System.out.println(arr[2]); // prints 3rd element viz 0
              System.out.println(arr[3]); // prints 4th element viz 0
              System.out.println(arr[4]); // prints 5th element viz 0
              // prints 6th element viz not present (Runtime Error)
```

```
                System.out.println(arr[5]);
                // update the values at index
                arr[0] = 10;
                arr[1] = 20;
                arr[2] = 30;
                arr[3] = 40;
                arr[4] = 50;

                // get the values
                System.out.println(arr[0]); // 10
                System.out.println(arr[1]); // 20
                System.out.println(arr[2]); // 30
                System.out.println(arr[3]); // 40
                System.out.println(arr[4]); // 50
                System.out.println(arr.length); // 5

                // print the array using for loop
                for (int i = 0; i<arr.length; i++) {
                        System.out.print(arr[i] + " "); // 10 20 30 40 50
                }

                // print the array using enhanced for loop
                for (int val :arr) {
                        System.out.print(val + " "); // 10 20 30 40 50
                }
        }
}
```

Function and Arrays

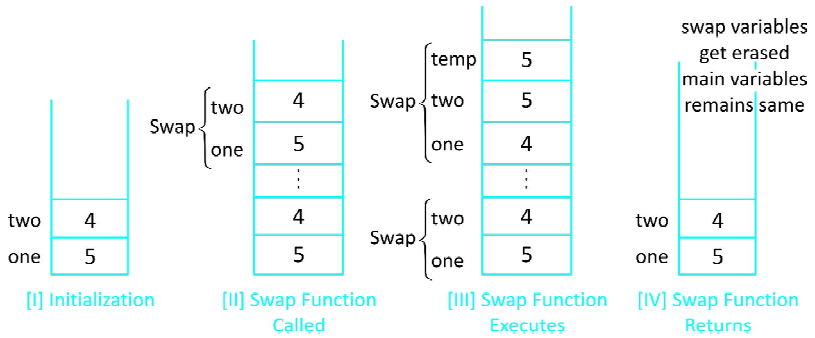
In the function definition we specify the type of the array along with its name whereas in the function call we just pass the name of the array. As we know that array is a non primitive data type so, only its reference is stored in the stack and when passed in a function call, the reference is passed.Thus if the function makes any changes in the array, they would be reflected in the original array also as both the functions (the calling function and the called function) share the reference to the same array. Let us verify this thing by writing a simple program of swapping two values.

```
public class ArrayDemo {

public static void main(String[] args) {
        int one = 5, two = 4;

        int[] arr = { 1, 2, 3, 4, 5 };

        System.out.println(one + " " + two); // 5 4
```

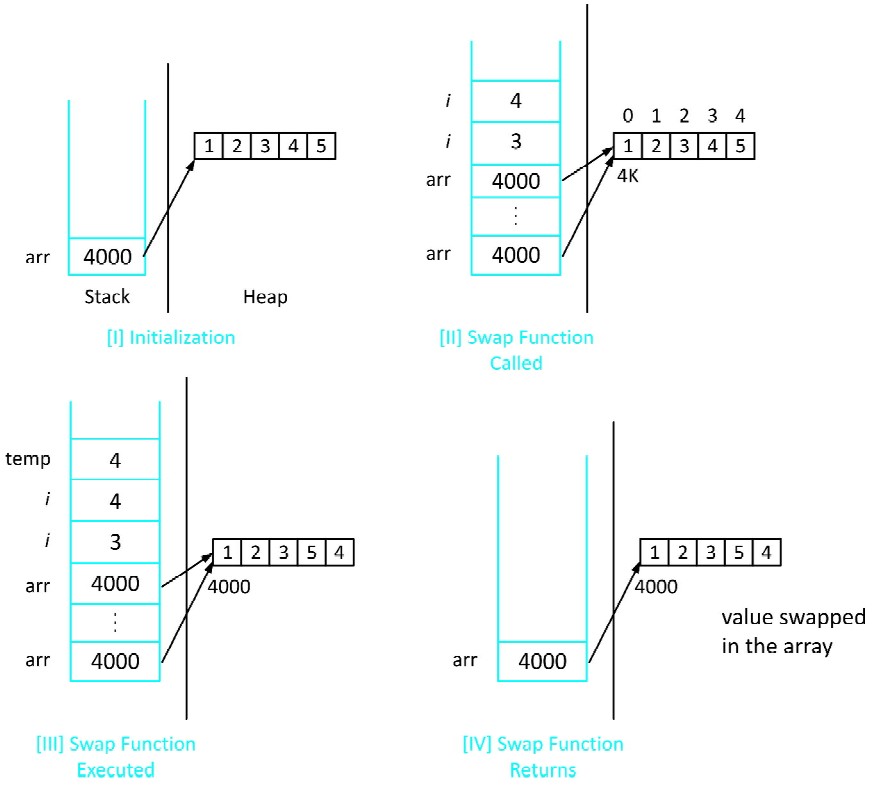
```
        swapIntegers(one, two);
        System.out.println(one + " " + two); // 5 4
        // Numbers not reversed

        System.out.println(arr[3] + " " + arr[4]); // 4 5
        swapArrayNums(arr, 3, 4);
        System.out.println(arr[3] + " " + arr[4]); // 5 4
        // Numbers reversed
    }
    public static void swapIntegers(int one, int two) {
        int temp = one;
        one = two;
        two = temp;
    }

    public static void swapArrayNums(int[] arr, int i, int j) {
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }

}
```

In the first case, where the variables were primitive, only the copies of the values were passed. Thus after the returning of the swap function, the changed variables are destroyed and hence no changes are retained. But in the second case, changes are made in the same array as both the functions' array variable pointed to the same array. Thus even after the function returns, the changes are retained. Remember that there is no such thing as pointers in java!

Memory maps of swap function:
Non-Working Swap:



Working Swap:



Multi-Dimensional Array

In Java, a multi-dimensional array is nothing but an array of arrays.

2D array: A two-dimensional array in Java is represented as an array of one-dimensional arrays of the same type. Mostly, it is used to represent a table of values with rows and columns (also known as matrix form).

Syntax

1. Declaration of a 2-D array

```
dataType[][] arrayRefVar; // preferred way
dataTypearrayRefVar[][]; // works too but generally not preferred
```

2. Instantiation of an 2-D array

```
array_ref_name = new datatype[no_of_rows][no_of_columns];
// 2-D array of int called arr of 9 elements containing 3 rows and 3 columns
int[][] arr = new int[3][3];

// 2-D array of long called arr of 8 elements containing 4 rows and 2 columns.
long[][] arr = new long[4][2];

// Declaration and Initialization at the same time
int[][] arr = { { 1, 2, 3 }, { 4, 5, 6 } };
```

Like in one-dimensional array,here also we can provide both of the bounds of rows and columns at the run time.

```
//Assigning bounds of rows and column at the run time
public classArrayDemo {

public static voidmain(String[] args) {

        Scanner scn = new Scanner(System.in); // Scanner class to take input
        int rows = scn.nextInt(); // Taking input of a rows
        int cols = scn.nextInt(); // Taking input of columns
        int[][] arr = new int[rows][cols]; // Size declared at the run time

}
}
```

In 2-D arrays also the indexing of each of the bound begins with zero.
We can access each element of a 2-D array in the same way as mentioned above for the 1-D array

```
int[][] arr = new int[5][2];
arr[0][0] = 80; // element of 0th row and 0th column initialized with value 80
arr[0][2] = 10; // element of 0th row and 2nd column initialized with value 10
```

So on…

To find the number of rows in the 2D array:arr.length

To find the number of columns in the 2D array:arr[0].length

```
int[][] arr = new int[3][4];
        System.out.println(arr.length); // prints rows viz. 3
        System.out.println(arr[0].length); // prints cols viz. 4
public class ArrayDemo {

        public static void main(String[] args) {

                Scanner scn = new Scanner(System.in);

                int rows = scn.nextInt();
                int cols = scn.nextInt();

                // Declaring 2D array
                int arr[][] = new int[rows][cols];

                // Taking inputof 2D array
                for (int i = 0; i<rows; i++) {
                        for (int j = 0; j<cols; j++) {
```

```
                        arr[i][j] = scn.nextInt();
                }
        }

        // Displaying 2D array
        System.out.println("Content of 2D Array");
        for (int i = 0; i<rows; i++) {
                for (int j = 0; j<cols; j++) {
                        System.out.print(arr[i][j] + " ");
                }
                System.out.println();
        }
  }
}
```

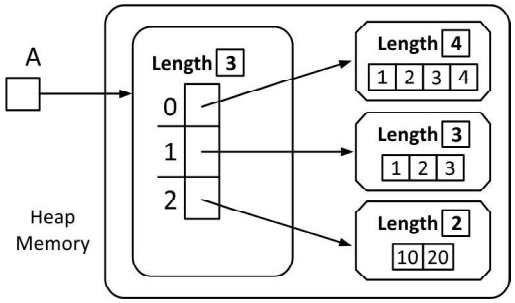
Jagged Array

Now while instantiating an array, it is mandatory to specify the number of rows. The number of columns may not be mentioned. We have seen the case when we specify both the rows and columns. But in case we want to have different number of columns for every row, we may skip the number of columns while specifying number of rows and instantiate columns later.

Formally, Jagged array is array of arrays such that member arrays can be of different sizes, i.e., we can create a 2-D array with variable number of columns in each row.

To find the number of rows in the jagged array: arr.length

To find the number of columns in specific row:arr[row].length



Program to demonstrate jagged array in Java

```
public class ArrayDemo {

public static voidmain(String[] args) {

        // Declaring 2-D array with 2 rows and cols not specified
        int arr[][] = new int[2][];
```

```
        // Making the above array jagged by instantiating columns later
        arr[0] = new int[3]; // First row has 3 columns
        arr[1] = new int[2]; // Second row has 2 columns

        int count = 1;

        // Taking input of jagged array
        for (int i = 0; i<arr.length; i++) {
            for (int j = 0; j<arr[i].length; j++) {
                arr[i][j] = count++; // updating the values
            }
        }
        // Displaying 2D Jagged Array
        for (int i = 0; i<arr.length; i++) {
            for (int j = 0; j<arr[i].length; j++) {
                System.out.print(arr[i][j] + " "); // Displaying Array
            }
            System.out.println();
        }
    }
}
```

Output:

1 2 3
4 5

PRACTICE QUESTIONS

1. Write a program to find the max of an array.
2. Write a program to reverse an array.
3. Write a program to rotate an array with k degree.

Original array :

1	2	3	4	5	6	7

Array after rotation of 5 degree

3	4	5	6	7	1	2

4. Write a program to find the sum of contiguous subarray within a one-dimensional array of numbers which has the largest sum.

Largest Subarray Sum Problem

–2	–3	4	–1	–2	1	5	–3
0	1	2	3	4	5	6	7

4 + (–1) + (–2) + 1 + 5 = 7

Maximum Contiguous Array Sum is 7

3 Searching Algorithm

Linear Search

Linear search, as the name suggests, is a searching algorithm where the search for an item is performed linearly in arrays. We start by searching from the first element and keep moving to the next element until either we find the item to be searched, or all the elements in the array are exhausted.

Algorithm:

Step 1: Read the search element from the user

Step 2: Compare, the search element with the first element in the list.

Step 3: If both are matching, then our task is completed. Print the matching index and terminate the function

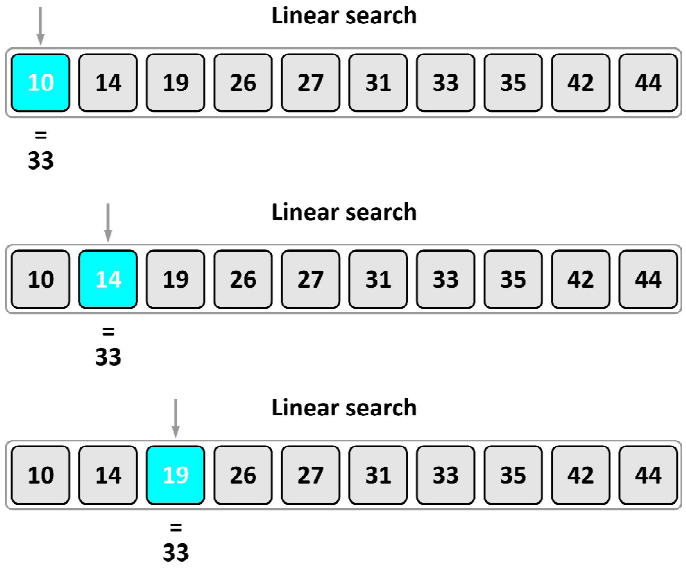
Step 4: If both are not matching, then compare search element with the next element in the list.

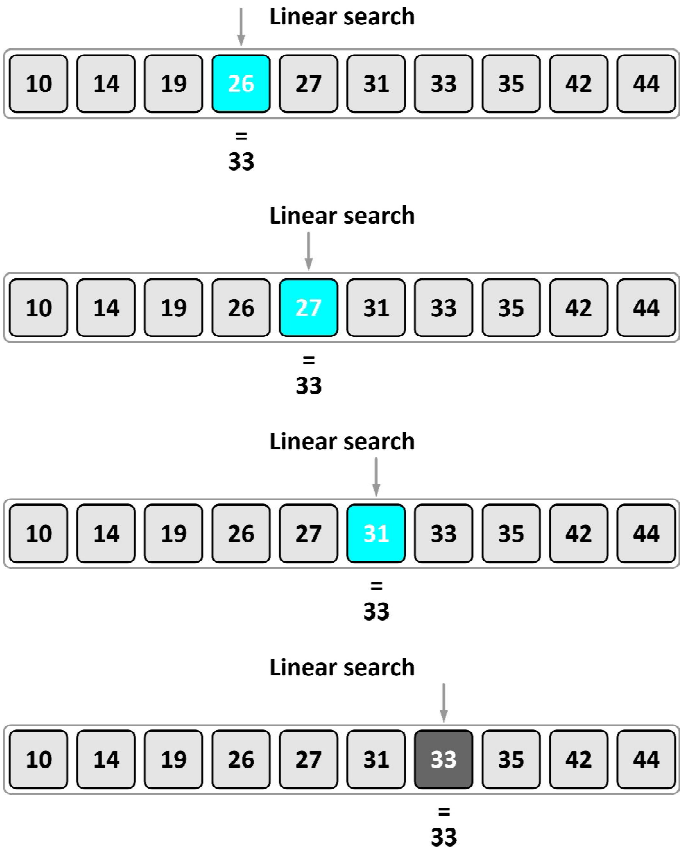
Step 5: Repeat steps 3 and 4 until the search element is compared with the last element in the list.

Step 6: If the last element in the list also doesn't match, then display "Element not found!!!" and terminate the function.

Working :

Item to be searched: 33





Item found at index 6.

Program:

The following program returns the index at which the item is found, if the item is present in the array. Else, it returns-1.

```
public static int linearSearch(int[] arr, intitem) {

	for (int i = 0; i<= arr.length - 1; i++) {
		if (arr[i] == item) {
			return i;
		}
	}
	return -1;

}
```

Binary Search :

For this algorithm to work properly, the data collection should be in the sorted form. This search algorithm works on the principle of divide and conquer. It halves the array after each unsuccessful iteration. Thus, after every iteration, we are only looking in half of the array from the previous iteration.

Algorithm :

We basically ignore half of the elements just after one comparison.

Step 1: Compare x with the middle element.

Step 2: If x matches with middle element, we return the mid index.

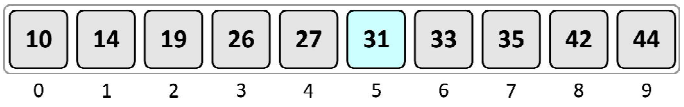
Step 3: Else If x is greater than the mid element, then x can only lie in right half subarray after the mid element. So we traverse only in the right half.

Step 4: Else (x is smaller) traverse only in the left half.

Step 5: Repeat Step 3 & 4 until either element found or the array is completely exhausted.

Working :

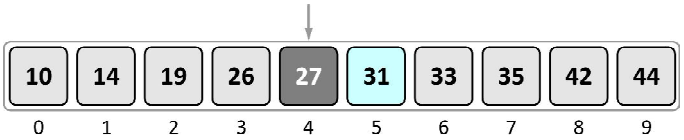
Item to be searched: 31



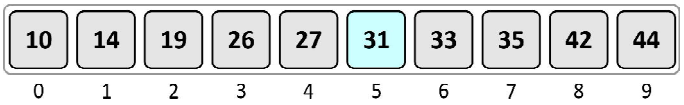
First, we shall determine half of the array by using this formula “

mid = (low +high) / 2

Here it is, (0 + 9) / 2 = 4 (integer value of 4.5). So, 4 is the mid of the array.



Now we compare the value stored at location 4, with the value being searched, i.e. 31. We find that the value at location 4 is 27, which is not a match. As the value to be searched is greater than 27 and we have a sorted array, so we need to search for the target value only in second half of the array.

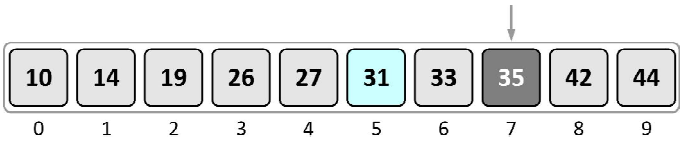


We change our low to mid + 1 and find the new mid value again.

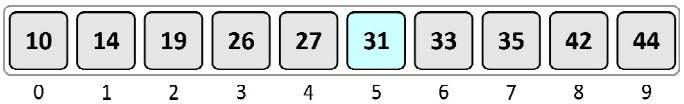
low = mid + 1 i.e. 5

mid = (low + high) / 2 i.e. (5 + 9)/2

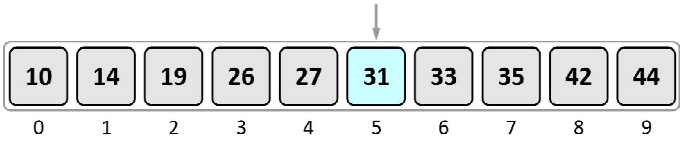
Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.



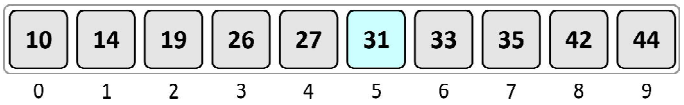
The value stored at location 7 is not a match, rather it is more than what we are looking for. So, the value must be in the lower part from this location. Now, high is changed to mid-1.



Now, low = 5 and high = 6. We calculate the mid again. This time it is 5.



We compare the value stored at location 5 with our target value. We find that it is a match.



We conclude that the target value 31 is stored at location 5.

Program:

```
public static int binarySearch(int[] arr, int item) {

	int lo = 0;
	int hi = arr.length - 1;

	while (lo<= hi) {

		int mid = (lo + hi) / 2;

		if (item>arr[mid]) {
			lo = mid + 1;
		} else if (item<arr[mid]) {
			hi = mid - 1;
		} else {
			return mid;
		}
	}

	return -1;

}
```

The above program performs binary search in a sorted array and returns the index at which the item is found if the item is present. Else, it returns -1.

4 ArrayList

The issue with the array is that it cannot be expanded or made to shrink. Arrays are like rope, they will have a fixed length, cannot be expanded nor reduced from the original length. So, if the array is full you cannot add any more elements to it, likewise ifelements are removed from the array, the memory consumption would be the same as it doesn't shrink.

ArrayList is a dynamic array i.e. its size is not fixed. ArrayList can dynamically grow and shrink after addition and removal of elements. Formally, ArrayList is a data structure that can be stretched to accommodate additional elements within itself and shrink back to a smaller size when elements are removed. It is a very important data structure, useful in handling the dynamic behavior of elements.

The important points about Java ArrayList class are:

- Java ArrayList class can contain duplicate elements.
- Java ArrayList class maintains insertion order.
- Java ArrayList class is non-synchronized.
- Java ArrayList allows random access because array works at the index basis.

While creating an ArrayList, we need to specify the type of value thathas to be stored in the ArrayList within the angular brackets. An important point to note here is that ArrayList can only be made up of the Class and not of primitive data types like 'int', 'boolean', 'float', etc. Thus for creating an ArrayList of integers we need to use the 'Integer' keyword within the angular brackets.

```
public static void main(String args[]) {

	// Creating ArrayList
	ArrayList<String> list = new ArrayList<String>();

	// Adding elements in ArrayList
	list.add("Ravi");
	list.add("Vijay");
	list.add("Harry");
	list.add("Rashid");

	// Processing each element of ArrayList
	for (String item :list) {

		// Printing each element on console
		System.out.println(item);
	}
}
```

Output:

Ravi
Vijay
Harry
Rashid

Inbuilt Operations

ArrayList provides some inbuilt methods to make the efforts easier.

- **add (int index, Object element)** :Used to insert the element at the specified index in a list.
- **boolean add()** : It is used to append the element to the end of list.
- **E get(int index)** : This method returns the element at the specified position in this list.
- **E set(int index, E element)** : This method replaces the element at the specified index with the specified element.
- **E remove(int index)** : This method removes the element at the specified index in the list.
- **boolean isEmpty()**: This method returns true if the list contains no elements.
- **int size()** : This method returns the number of elements in list.
- **void clear()** : It is used to remove all of the elements from list.
- **int indexOf(Object o)** : It is used to return the index of the first occurrence of the specified element and returns -1 if the list does not contain this element.
- **boolean contains(Object o)** : This method returns true if this list contains the specified element.

```
public static void main(String[] args) {

        // Create ArrayList of integers
        ArrayList<Integer> list = new ArrayList<>();

        // Appending elements in the list
        list.add(10);
        list.add(20);
        list.add(40);
        list.add(30);
        list.add(20);
        list.add(1, 50);

        // Enhanced loop
        for (int val :list) {

              // Printing on the console
              System.out.println(val + " ");
        }

        // Return true if contains element 50
        System.out.println(list.contains(50));    // true

        // Update element at index 3
        list.set(3, 70);
```

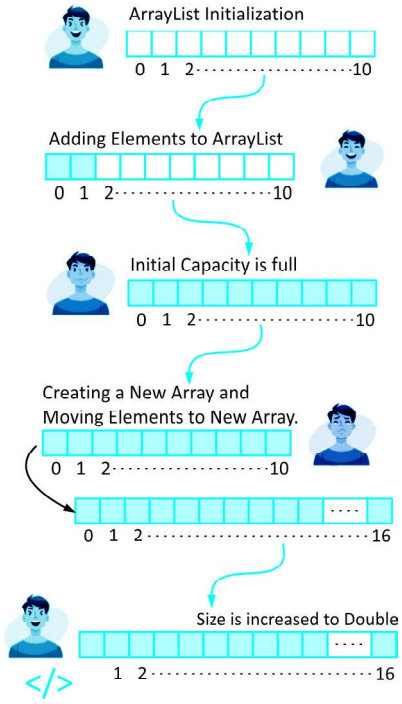
```
        // Returns index of first occurrence of 20
        System.out.println(list.indexOf(20));   // 2

        // Get the element at index 4
        System.out.println(list.get(4));   // 30

        // Return true if list is empty
        System.out.println(list.isEmpty());   // false

        // Return index of last occurrence of 20
        System.out.println(list.lastIndexOf(20));   // 5

        // Return and remove the element at index 2
        System.out.println(list.remove(2));   // 20

        // Return the size of the list
        System.out.println(list.size());   // 5

        // Clear the list
        list.clear();

        // Print the list
        System.out.println(list);  // []
    }
```

Working of ArrayList

In ArrayList, we need to deal with two different terms: Size and Capacity. Size tells the total number of elements which the user has added in ArrayList. Capacity is used internally by the ArrayList to keep track of total number of elements which can be added.

When we add an element to ArrayList, it first verifies whether it has capacity to store the new element or not. If space is available then the element is added at last, in case there is no space left then the new capacity is calculated which is 50% more than the old capacity.A new array will becreated at a new location and all the older elements will be copied in new array.

Increasing the capacity by 1.5 times ensures accessing (get)an element inO(1) and appendtlast in O(1). This has the advantage of wasting slightly less space when the ArrayList is not very full.



PRACTICE QUESTIONS

1. Given two sorted arrays, store their intersection in the arraylist and then print theresulting ArrayList.

   Input :arr1[] = {1, 3, 4, 5, 7}arr2[] = {2, 3, 5, 6}

   Output :Intersection : [3, 5]

2. Given two array **A[0….n-1]** and **B[0….m-1]** of size **n** and **m** respectively, representing two numbers such that every element of arrays represent a digit. The task is to find the sum of both the numbers and store the sum in an ArrayList and print the ArrayList.

   Input :n = 3, m = 3
   a[] = { 1, 2, 3 }
   b[] = { 2, 1, 4 }

   Output : [3, 3, 7]
   Explanation :123 + 214 = 337

5 Strings

If we haveto save a student's name, one way is to create an array of type char and store each letter at one index. But it would be very difficult to use that as every time we will have to process the whole array. Instead we can use a **non primitive datatype** called 'String'. Stringsare a sequence of characters and are treated as objects in java programming language. Every string that you create is an object of type **String**. One important thing to notice about string object is that string objects are immutable that means once a string object is created it cannot be altered.

There are two ways to create String object:

1. By string literal
2. By new keyword

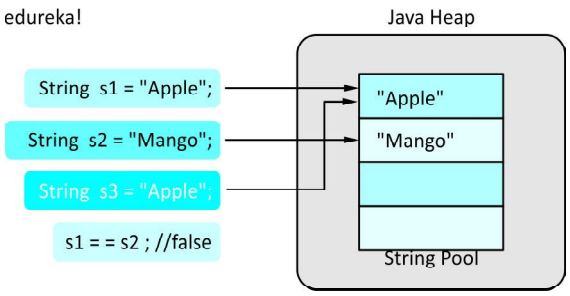
By String Literal

Java String literal is created by using double quotes.

String ref_name = "Any String";

String s = "Welcome";

Java String Pool: A part of heap memory is reserved to store Strings. This part of memory is called String Pool or Intern Pool. Whenever a new string is created, JVM check if the string is already present in the intern pool or not. If it is present, then same reference is returned to the variable else new string will be created in the String pool and the respective reference will be returned. Refer to the diagrammatic representation for better understanding:



In the above image, two Strings are created using literal,"Apple" and "Mango". Now, when third String is created with the value "Apple", instead of creating a new object, the already present object reference is returned. The advantage of returning the same reference is that we are saving heap space.

Before we go ahead, one key point to note is that unlike other data types in Java, Strings are immutable. By immutable, we mean that there is no way via which we can change the string at the original location in memory.

JVM checks if the object already exists in the intern pool, if it does, then a new Object is not created rather it assigns the same old reference. Suppose we create 5 strings (str1, str2, str3, str4, str5) as shown below.

String str1 = “Hello”;

String str2 = “Hello”;

String str3 = “Hello”;

String str4 = “Hello”;

String str5 = “Hello”;

Let’s see how the statements get executed, When String str1 = “Hello”gets executed thenJVM checks if hello is present in intern pool or not, as of now there is no string with content “Hello” in intern pool, so a new string gets created and its address is returned to str1 on stack. WhenString str2 = “Hello”gets executed then JVM checks in intern pool if “Hello”exists or not, as of now “Hello”is present in intern pool so the older address is returned and new string is not created. So, in a way for str2, str3, str4, str5 olderaddress/reference is returned and new string is not created. It means that in memory there is only one object having the content “Hello”and all the 5 stringwould be pointing to the same object.

What if we want to have two different objects with the same string value? For that we would need to create strings using **new keyword**.

By New Keyword

Java String can also be created by using a keyword **new**.

```
String ref_name = newString(“Any String”);
String s = new String(“Welcome”);
```

String created using new keyword will always be created outside intern pool and inside heap.

```
String str6 = new String(“Hello”);
String str7 = new String(“Hello”);
```

When the first statement of str6 gets executed then a string with content “Hello” will be created at a new memory address. And when str7 gets executed then again a string “Hello”will be created at a new memory address. After execution of both the statements, two strings with content “Hello”will be present in heap.

The java.lang.String class provides many useful methods to perform operations on sequence of char values.

Functions :

Now let us learn about different function that can be used to process strings.

- **charAt(index)** : Function returns the character that is present at the specified index. If the index is not valid it will give a run time error.
- **substring(si,ei)** :Function returns a substring of the original string from si (start index) to ei (end index), including si and excluding ei.

- **substring(si)** :This function returns the string from si till the end of the string. If the parameters is invalid, it will give a run time error.
- **concat(str)**: Function joins the string passed as a parameter to string on which the function is invoked.
- **indexOf(ch)** :Function returns the first index at which the character is found and returns -1 if the character is not found.
- **startsWith(str)** :Function checks if the string on which the function is invoked starts with the string passed as a parameter or not. It returns a Boolean value - true or false

For comparison of two values we have always used '==' operator. But in the case of string it doesn't always gives the correct answers. This is because == operator just compares the value present on the stack. As string is a non primitive, the reference to the string object is stored on stack. So if the reference is same it returns true otherwise false. But there might be a case when the reference in the stack is different but the values present on the actual object is same, but even then it would return false. To overcome this problem, we use the **'equals'** method on the string. This method compares the actual content of the strings character by character. Therefore we should always use s1.equals(s2) for checking if string s1 is equal to s2 or not.

StringDemo

```
public static void main(String[] args) {
String str = "Hello";

// Part-1 : Length of String
System.out.println(str.length());  // 5

// Part-2 : CharAt Limit:0->length-1
char ch = str.charAt(0);
System.out.println(ch);  // H
System.out.println(str.charAt(str.length())); // Exception thrown - Index out of bound
System.out.println(str.charAt(str.length() - 1));  // o

// Part-3 : Substring Limit:0->length
System.out.println(str.substring(2, 4)); // ll
System.out.println(str.substring(0, 3)); // Hel
System.out.println(str.substring(0, 5)); // Hello
System.out.println(str.substring(0, 6)); // exception
System.out.println(str.substring(3, 4)); // l
System.out.println(str.substring(4, 4)); // blank
System.out.println(str.substring(2)); // llo
System.out.println(str.substring(5, 4)); // Exception thrown : Index out of bound

// Part-4 : Concatenate Two Strings
```

```
    String s1 = "hi";
    String s2 = "bye";
    String s3 = s1 + s2;
    System.out.println(s1 + ", " + s2 + ", " + s3);   // hi, bye, hibye
    String s4 = s1.concat(s2);
    System.out.println(s1 + ", " + s2 + ", " + s4);   // hi, bye, hibye

    // Part-5 :IndexOf
    System.out.println(str.indexOf("el"));   // 1
    System.out.println(str.indexOf("eL"));   // -1

    // Part-6 : StartsWith
    System.out.println(str.startsWith("He"));   // true
    System.out.println(str.startsWith("he"));   // false

    // Part-7 : Equals and ==
    s1 = "Hello";
    s2 = s1;
    s3 = "Hello";
    s4 = new String("Hello");

    System.out.println((s1 == s2) + ", " + s1.equals(s2)); // true true
    System.out.println((s1 == s3) + ", " + s1.equals(s3)); // true true
    System.out.println((s1 == s4) + ", " + s1.equals(s4)); // false true
}
```

PRACTICE QUESTIONS

1. Write a program to print the no of palindromic substrings in a given String.

   Input: "nitin"

   Output: 7

   Explanation: [n , i , t , i , n , iti , nitin]

2. Given a list of sorted characters consisting of both Uppercase and Lowercase alphabets and a particular target value, say **K**, the task is to find the smallest element in the list that is larger than K.

   Input: Letters = ["D", "J", "K"]

   K = "B"

   Output: 'D'

3. Take as input S, a string. Write a function that removes all consecutive duplicates. Print the value returned.

   Input: aabccba

   Output: abcba

4. Given an array of numbers, arrange them in a way that yields the largest value.
   Input: {1, 34, 3, 98, 9, 76, 45, 4}
   Output: 998764543431

5. Given a string, check if it is palindrome or not. A string is said to be palindrome if it has consecutive 3 alphabets followed by the reverse of these 3 alphabets and so on.
   Input: cappaccappac Output : String is palindrome
   Input : mollomaappaa Output : String is palindrome

6 String Builder

We already know that strings are immutable. So, if we want to change any string or add a character to a string, first the contents of the original string is copied and then the necessary changes are made. So this is a very costly operation in terms of time. To save time, we can use String Builder which is mutable and hence is very efficient. StringBuilder class should be used for String manipulation.

StringBuilder are mutable objects in java and provide append(), insert(), delete() and substring() methods for String manipulation.

Function:

- **StringBuilder append(String s)** : Used to append the specified string with the string on which function is invoked.
- **StringBuilder insert(int offset, String s)** : Used to insert the specified string at the given index.
- **StringBuilder replace(int startIndex, int endIndex, String str)** :Used to replace the string from startIndex to endIndex with the specified string.
- **StringBuilder delete(int startIndex, int endIndex)** :Used to delete the string from specified startIndex and endIndex.
- **StringBuilder reverse()**: Used to reverse the string.
- **char charAt(int index)** :Used to return the character at the specified position.
- **int length()** : Used to return the length of the string i.e. total number of characters.
- **String substring(int beginIndex)** :Used to return the substring from the specified beginIndex to end.
- **String substring(int beginIndex, int endIndex)** :is used to return the substring from the specified beginIndex and endIndex.

```
public static void main(String[] args) {

        // Instantiation
        StringBuilder sb = newStringBuilder();

        // Appends the string in the last
        sb.append("Coding Blocks");
        sb.append(" Course");

        // Prints String Builder
        System.out.println(sb); // Coding Blocks Course
```

```
        // Insert string at 0th index
        sb.insert(0, "The ");
        System.out.println(sb); // The Coding Blocks Course

        // Replace the string from 11 to 13 index with "Java"
        sb.replace(11, 14, "Java");
        System.out.println(sb); // The Coding Javacks Course

        // Deletes the string from 17 index to 19 index
        sb.delete(17, 20);
        System.out.println(sb); // The Coding Javackourse

        // Prints length of current StringBuilder
        System.out.println(sb.length()); // 22

        // Returns substring after 18th index
        System.out.println(sb.substring(19)); // rse

        // Returns substring from 4th index to 15th index
        System.out.println(sb.substring(4, 16)); // rse
    }
```

PRACTICE QUESTIONS

1. Given a String, write a program to toggle all characters.

   Input: aBcDe

   Output: AbCdE

2. Given a string str of lowercase characters. The task is to count the number of deletions required to reduce the string to its shortest length. In each delete operation, you can select a pair of adjacent lowercase letters that match, and then delete them. The task is to print the count of deletions done.

   Input: str = "aaabccddd"

   Output: 3

   Explanation: aaabccddd -> abccddd -> abddd -> abd

3. Given a string S, comprising of only lowercase English alphabets, the task is to update the string by eliminating such vowels from the string that occur between two consonants.

   Input: bab

   Output: bb

4. Write a program to reverse the words in the given String without reversing the words internally.

   Input: The Sky is Blue

   Output: Blue is Sky The

7 Recursion

Recursion is a technique (just like loops) where the solution to a problem depends on the solution to smaller instances of that problem. In cases where we need to solve a bigger problem, we try to reduce that bigger problem in a number of smaller problems and then solve them. For instance we need to calculate 2^8.We can do this by first calculating 2^4 and then multiplying 2^4 with 2^4. Now to calculate 2^4, we can calculate 2^2 which in turn depends on just 2^1. So for calculating 2^8, we just need to calculate 2^1 and then keep on multiplying the obtained numbers until we reach the result.

So by dividing the problem in smaller problems, we can solve the original program easily. But we can't go on making our problem smaller infinitely. We need to stop somewhere and terminate. This point is called a **base case**. The base case tells us that the problem doesn't have to be divided further. For instance, in the above case when power of 2 equals to 1 we can return 2. So power being equal to 1 becomes our base case.

```
public static int power(int x, int n) {

	// base case
	if (n == 1) {
		return x;
	}

	// smaller problem
	int smallPow = power(x, n / 2);

	// convert answer of smaller problem to answer of bigger problem
	if (n % 2 == 0) {
		return smallPow * smallPow;
	} else {
		return smallPow * smallPow * x;
	}
}
```

In the next program we will be calculating the factorial of a number.

For calculating the factorial of n, all we need is the factorial for n-1 and then multiply it with the number itself. As we can see that factorial of a bigger number depends on the factorial of a smaller number, thus this problem can be solved by recursion.

So for calculation of factorial of a bigger number, we first calculate factorial of n-1 and then multiply it by n. If the number is 1, we don't need to calculate anything but just return 1. Thus this is our base case.Notice that in the below code the factorial function calls itself on a smaller input and then uses that result to calculate the final answer. Thus in recursion, a function calls itself with a smaller value.

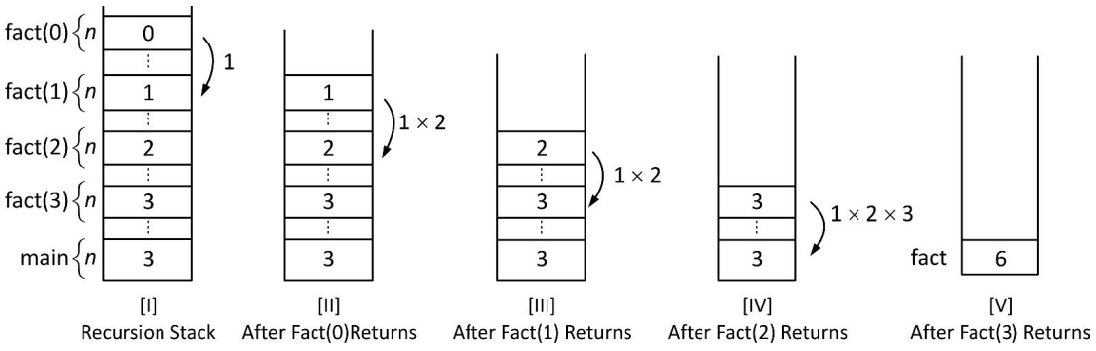
An important point to note here is that in the base case we need to have a return statement compulsorily even if the return type of the function is void. This is because in the base case if we don't have the return statement, the code after the base case would begin to execute and the recursion would execute infinitely. Thus to terminate recursion, we need to have a return statement in the base case.

```
public static int factorial(int n) {

	// base case
	if (n == 1) {
		return 1;
	}

	// smaller problem : factorial of n-1
	int fnm1 = factorial(n - 1);

	// convert answer of smaller problem to answer of bigger problem
	int fn = n * fnm1;

	return fn;

}
```

For factorial of 5:

```
factorial(5)
    factorial(4)
        factorial(3)
            factorial(2)
                factorial(1)
                    return 1
                return 2*1 = 2
            return 3*2 = 6
        return 4*6 = 24
    return 5*24 = 120
```

Let us understand the recursive function calls using memory maps for factorial of 3:



Now we know that every recursive function includes **a base case** and **a call to itself**. So every time the function calls itself we have two opportunities to do our work. First is before calling itself and second is after the call. First opportunity helps to do work on the input whereas second opportunity lets us to do work on the result of the smaller problem. To understand this better, let us write code for printing a series of increasing and decreasing numbers.

```
public static void PDI(int n) {

	if (n == 0) {
		return;
	}

	System.out.println(n);

	PDI(n - 1);

	System.out.println(n);
}
```

The first print statement helps in printing the series in decreasing order while the second print statement helps in printing the series in ascending order.

Some more Programs on Recursion :

1. Find the N^{th} Fibonacci term

```
public static int fibonacci(int n) {

	// base case
	if (n == 0 || n == 1) {
		return n;
	}

	// n-1 fibonacci
	int fnm1 = fibonacci(n - 1);

	// n-2 fibonacci
	int fnm2 = fibonacci(n - 2);

	// nth fibonacci is calculted by sum of (n-1) and (n-2) fibonacci
	int fn = fnm1 + fnm2;

	return fn;

}
```

2. Find first occurrence of an item in array

```
public static int findFirst(int[] arr, int vidx, int item) {

	// base case : array is exhausted
	if (vidx == arr.length) {
		return -1;
```

```
        }

        // if item found then return present index
        if (arr[vidx] == item) {
                return vidx;
        }

        return findFirst(arr, vidx + 1, item);

}
```

3. Display the array in reverse using recursion

```
public static void displayArrReverse(int[] arr, int vidx) {

        // base case : array is exhausted
        if (vidx == arr.length) {
                return;
        }

        // display rest of array in reverse
        displayArrReverse(arr, vidx + 1);

        // display present element
        System.out.print(arr[vidx] + " ");
}
```

Every time a recursive function is called, the called function is pushed into the running program stack over the calling function. This uses up memory equal to the data members of a function. So, using recursion is recommended only up to certain depth. Otherwise too much memory would be wasted in stack space for program.

Limitation of Recursive Call:

There is an upper limit to the number of recursive calls that can be made. To prevent this make sure that your base case is reached before stack size limit exceeds.

So, if we want to solve a problem using recursion, then we need to make sure that:

1. The problem can be broken down into smaller problems of same type.
2. Problem has some base case(s).
3. Base case is reached before the stack size limit exceeds.

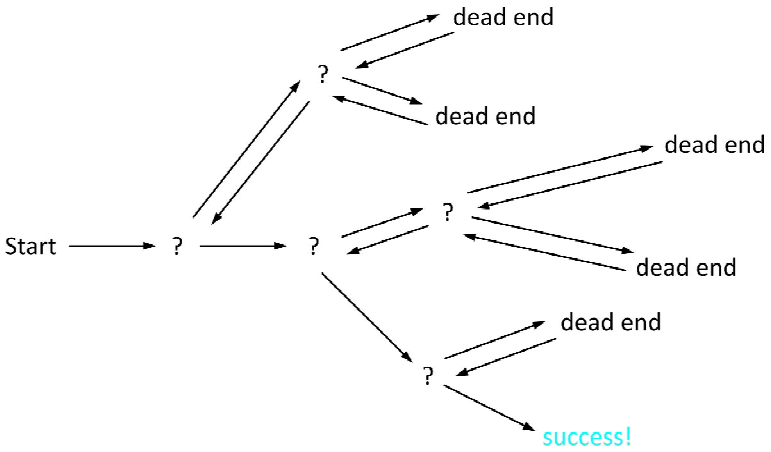
Backtracking:

When we solve a problem using recursion, we break the given problem into smaller ones.Let's say we have a problem PROBand we divided it into three smaller problems P1, P2 and P3. Now it may be the case that the solution to PROB does not depend on all the three sub-problems, in fact we don't even know on which one it depends.

Let's take a situation. Suppose you are standing in front of three tunnels, one of which is having a bag of gold at its end, but you don't know which one. So you'll try all three. First go in tunnel 1 till its end, if that is not the one, then come out of it, and go into tunnel 2, and again if that is not the one, come out of it and go into tunnel 3. So basically in backtracking we attempt solving a sub-problem, and if we don't reach the desired solution, then undo whatever we did for solving that sub-problem, and again try solving another sub-problem.

Basically Backtracking is an algorithmic paradigm that tries different solutions until it finds a solution that "works". Problems which are typically solved using backtracking technique have following property in common. These problems can only be solved by trying every possible configuration and each configuration is tried only once.

Note : We can solve this by using **recursion method**.



Backtracking Algorithm:

If destination is reached
Print the solution
Else
{

(a) Mark current cell in solution matrix as 1.

(b) Move forward in horizontal direction and recursively check if this move leads to a solution.

(c) If the move chosen in the above step doesn't lead to a solution then move down and check of this move leads to a solution.

(d) If none of the above solutions work then unmark this cell as 0 (Backtrack) and return false.

}

Problem based on Backtracking:

N-Queen Problem: Given a chess board having N×N cells, we need to place N queens in such away that no queen is attacked by any other queen. A queen can attack horizontally, vertically anddiagonally.

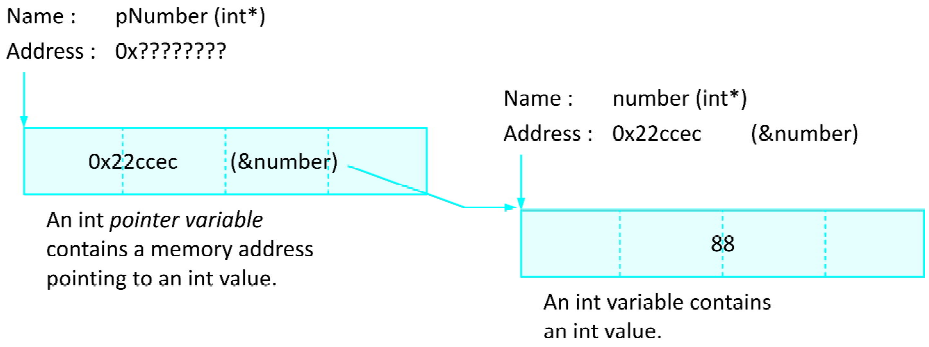
Approach towards the Solution:

We are having N × N un-attacked cells where we need to place N- queens. Let's place the first queen at a cell (i, j), so now the number of un-attacked cells is reduced, and number of queens to be placed is N – 1. Place the next queen at some un-attacked cell. This again reduces the number of un-attacked cells and number of queens to be placed becomes N – 2. Continue doing this, as long as following conditions hold:

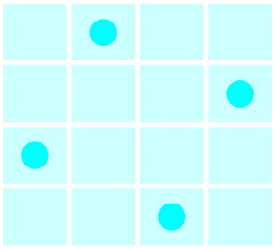
- The number of un-attacked cells is not equal to 0.
- The number of queens to be placed is not equal to 0.

If the number of queens to be placed becomes 0, then it's over, we found a solution. But if thenumber of un-attacked cells become 0, then we need to backtrack, i.e. remove the last placed queen from its current cell, and place it at some other cell.

Let's N = 4, we have to place 4 queen in 4×4 cells.



So, final solution of this problem is



So, clearly, we try solving a sub-problem, if that does not result in the solution, we undo whatever changes were made and solve the next sub-problem. If the solution does not exists (like N=2), then it returns false.

8 Sorting Algorithms

Bubble Sort

Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$ where **n** is the number of items.

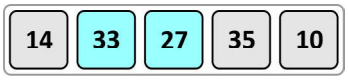
How Bubble Sort Works?

Lets take an unsorted array. Bubble sort takes Ï(n^2) time so we're keeping it short and precise.

Bubble sort starts with very first two elements, comparing them to check which one is greater.

In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.

We find that 27 is smaller than 33 and these two values must be swapped.

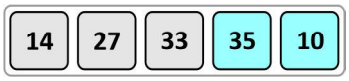


The new array should look like this "

Next we compare 33 and 35. We find that both are in already sorted positions.

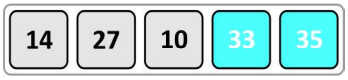
Then we move to the next two values, 35 and 10.

We know then that 10 is smaller 35. Hence they are not sorted.



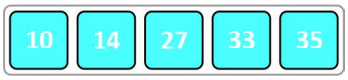
We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this –

To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this –



Notice that after each iteration, at least one value moves at the end.

And when there's no swap required, bubble sorts learns that an array is completely sorted.




```
public static void bubblesort(int[] arr) {
	for (int counter = 0; counter < arr.length - 1; counter++) {
		for (int j = 0; j < arr.length - 1 - counter; j++) {
			if (arr[j] > arr[j + 1]) {
				int temp = arr[j];
				arr[j] = arr[j + 1];
				arr[j + 1] = temp;
			}
		}
	}
}
```

Selection Sort

Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.

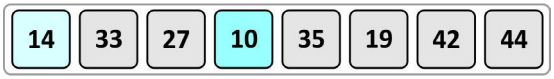
The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one element to the right.

This algorithm is not suitable for large data sets as its average and worst case complexities are of $O(n^2)$, where **n** is the number of items.

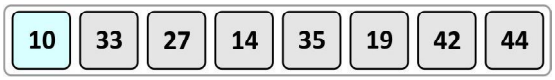
How Selection Sort Works?

Consider the following depicted array as an example.

We wish to place the correct element at first position i.e. 0^{th} index. So, the correct element will be the smallest element of the entire list. The whole list is scanned sequentially to find out the smallest element. We found out that 10 is the smallest element.

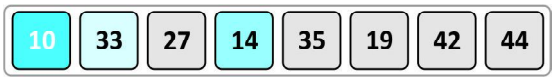


So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of the sorted list.



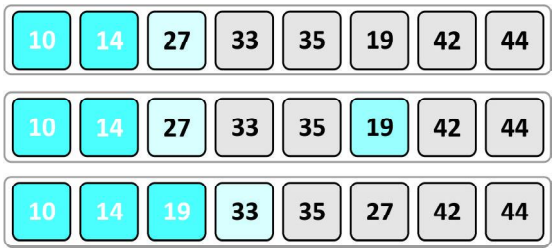
We wish to placer correct element at second position, where 33 is residing, we start scanning the rest of the list in a linear manner.

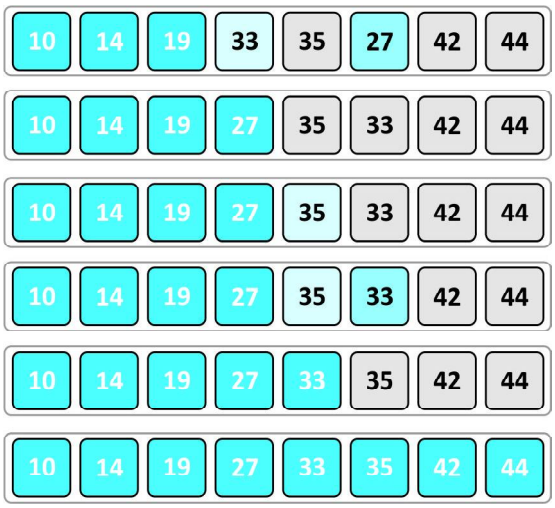
We find that 14 is the second lowest value in the list and it should appear at the second place. We swap these values.



After two iterations, two least values are positioned at the beginning in a sorted manner.

The same process is applied to the rest of the items in the array. Following is a pictorial depiction of the entire sorting process“





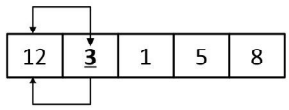
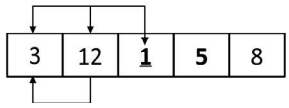
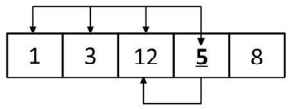
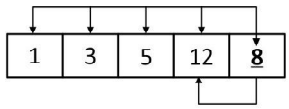
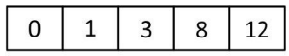

```
public static void selectionsort(int[] arr) {
	for (int counter = 0; counter < arr.length - 1; counter++) {
		int min = counter;
		for (int j = min + 1; j <= arr.length - 1; j++) {
			if (arr[j] < arr[min]) {
				min = j;
			}
		}

		if (min != counter) {
			int temp = arr[min];
			arr[min] = arr[counter];
			arr[counter] = temp;
		}
}
 }
```

Insertion Sort

This is an in-place comparison-based sorting algorithm. Here, a sub-list is maintained which is always sorted. For example, the lower part of an array is maintained to be sorted. An element which is to be 'inserted in this sorted sub-list, has to find its appropriate place and then it has to be inserted there. Hence the name, **insertion sort**.

The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list (in the same array). This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$, where **n** is the number of items.

How Insertion Sort Works?

Set 1		Checking second element of array with element before it and inserting it in proper position. In this case, 3 is inserted in position of 12.
Set 2		Checking third element of array with element before it and inserting it in proper position. In this case, 1 is inserted in position of 3.
Set 3		Checking fourth element of array with elements before it and inserting it in proper position. In this case, 5 is inserted in position of 12.
Set 4		Checking fifth element of array with elements before it and inserting it in proper position. In this case, 8 is inserted in position of 12.
		Sorted Array in Ascending Order

```
public static void insertionsort(int[] arr) {
	for (int counter = 1; counter <= arr.length - 1; counter++) {
		int val = arr[counter];
		int j = counter - 1;
		while (j >= 0 && arr[j] > val) {
			arr[j + 1] = arr[j];
			j--;
		}
		arr[j + 1] = val;
	}
}
```

Merge Sort

Merge Sort is a Divide and Conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. **The merge() function** is used for merging two sorted halves. The merge function takes two sorted arrays as input, merges the two arrays and returns a merged sorted array.

Merge sort is a sorting technique based on divide and conquer technique. Worst-case time complexity of merge-sort is O(n logn).

How Merge Sort Works?

MergeSort(arr[], lo, hi)

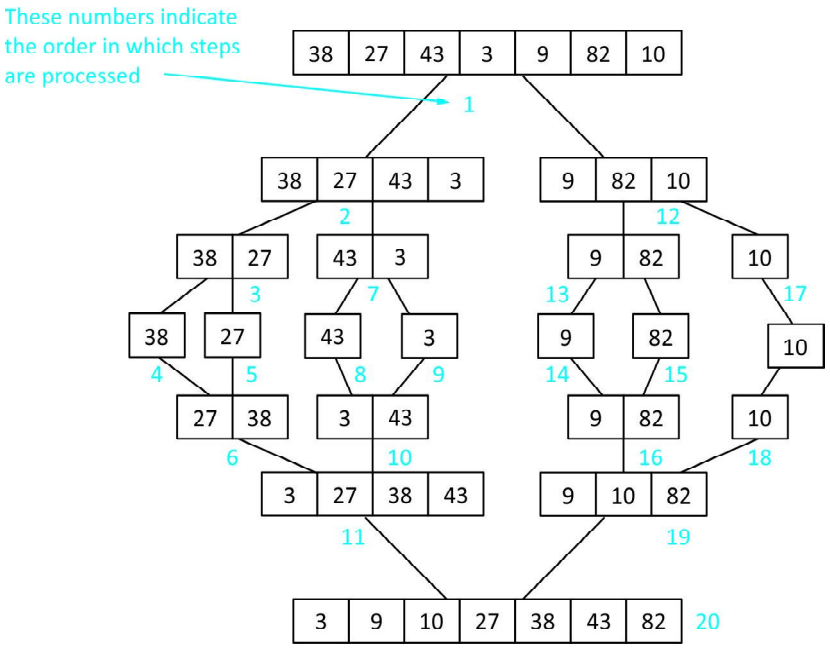
if lo > hi

1. Find the middle point to divide the array into two halves
   mid = (lo+hi) / 2
2. Call mergeSort for first half:
   firstHalf = Call mergeSort(arr, lo, mid)
3. Call mergeSort for second half:
   secondHalf = Call mergeSort(arr, mid+1, hi)
4. Merge the two halves sorted in step 2 and 3:
   sorted = Call merge(firstHalf, secondHalf)

The following diagram from shows the complete merge sort process for an array {38, 27, 43, 3, 9, 82, 10}. If we take a closer look at the diagram, we can see that the array is recursively divided in two halves till the size becomes 1. Once the size becomes 1, the merge processes comes into action and starts merging arrays back till the complete array is merged.

```
public static int[] merge(int[] one, int[] two) {
	int[] sorted = new int[one.length + two.length];
	int i = 0;
	int j = 0;
	int k = 0;
	while (i < one.length && j < two.length) {
		if (one[i] < two[j]) {
			sorted[k] = one[i];
			k++;
			i++;
		} else {
			sorted[k] = two[j];
			k++;
			j++;
		}
	}
	if (i == one.length) {
		while (j < two.length) {
			sorted[k] = two[j];
			k++;
			j++;
		}
	}
	if (j == two.length) {
```

```
            while (i < one.length) {
                  sorted[k] = one[i];
                  k++;
                  i++;
            }
      }

      return sorted;
}

public static int[] mergeSort(int[] arr, int lo, int hi) {

      if (lo == hi) {
            int[] br = new int[1];
            br[0] = arr[lo];
            return br;
      }

      int mid = (lo + hi) / 2;

      int[] firstHalf = mergeSort(arr, lo, mid);
      int[] secondHalf = mergeSort(arr, mid + 1, hi);

      int[] sorted = merge(firstHalf, secondHalf);

      return sorted;
}
```



QuickSort

Like Merge Sort, QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

1. Always pick first element as pivot.
2. Always pick last element as pivot (implemented below)
3. Pick a random element as pivot.
4. Pick median as pivot.

The key process in quickSort is partition(). After the partitioning step, all elements smaller than pivot will be on one side of array and all elements greater than pivot will be on other side of array. Pivot element can be on either sides of array or it can be at its correct position. All this should be done in linear time.

```
public static void quickSort(int[] arr, int lo, int hi) {
		if (lo >= hi) {
		return;
	}
	// partioning step
	int mid = (lo + hi) / 2;
	int pivot = arr[mid];
	int left = lo;
	int right = hi;
	while (left <= right) {
		while (arr[left] < pivot) {
			left++;
		}
		while (arr[right] > pivot) {
			right–;
		}
		if (left <= right) {
			int temp = arr[left];
			arr[left] = arr[right];
			arr[right] = temp;
			left++;
			right–;
		}
	}
	// sort both the halves
	quickSort(arr, lo, right);
	quickSort(arr, left, hi);
}
```

9 Time and Space Complexity

In the world of computing, we want our programs to be fast and efficient. But fast and efficient are loose terms. A software might be fast on a supercomputer but it may be insanely slow on a personal computer. So to define this *fast*ness and *efficiency*, we take the help of time and space complexity analysis.

Complexity Analysis is a way to quantify or *measure* time and space required by an algorithm with respect to the input fetched to it. In time complexity analysis, we are concerned with this growth which remains same on all machines. This analysis doesn't require algorithms to be run. It can be found just by doing a little math.

```
public static void print(int n) {
	for (int i = 0; i<n; i++) {
		System.out.println("Coding Blocks");
	}
}
```

The print function above runs n times. As n increases linearly, the time complexity of the program will also increase linearly. So, time complexity is proportional to n. We say that T(N) = O(N). This is Big-O notation. T(N) here denotes the time complexity when represented in terms of N, the input size.

```
public static void print(int n) {
	for (int i = 0; i< 10; i++) {
		System.out.println("Coding Blocks");
	}
}
```

Irrespective of n, print function runs exactly 10 times. So, we'll say, time complexity is T(N) = O(1). That is, the program runs for a constant time.

```
public static void print(int n) {
	for (int i = 0; i<n; i++) {
		for (int j = i; j<n; j++) {
			System.out.println("Coding Blocks");
		}
	}
}
```

Let us assume n = 4,

When *i* is	0	1	2	3
Then inner loop runs	4	3	2	1

Total printing done = Total time inner loop runs = Sum of first 4 natural numbers (4+3+2+1)

$$T(N) = \text{Sum of first N natural Numbers} = N(N+1)/2 = N^2/2 + N/2$$

To get the Big-O notation, just keep the term with the highest power of N, $N^2/2$ in our case. Remove all constants associated with this term, in our case 1 / 2 will be removed. Remaining term is the proportionality factor. Hence, we will say time complexity is Big-O of N square.$T(N) = O(N^2)$

Constant Complexity: O(1)

A constant task's run time won't change no matter what the input value is. Consider a function that prints a value in an array.

```
public static void printElement(int[] arr, int idx) {
	System.out.println("arr[" + idx + "] = " + arr[idx]);
}
```

No matter which element's value you're asking the function to print, only one step is required. So we can say the function runs in O(1) time; its run-time does not increase. Its order of magnitude is always 1.

Linear Complexity: O(N)

A linear task's run time will vary depending on its input value. If you ask a function to print all the items in a 10-element array, it will require less steps to complete than it would a 10,000 element array.

This is said to run at O(n); it's run time increases at an order of magnitude proportional to n.

Quadratic Complexity: $O(N^2)$

A quadratic task requires a number of steps equal to the square of its input value. Let's look at a function that takes an array and N as its input values where N is the number of values in the array.

If I use a nested loop both of which use N as its limit condition, and I ask the function to print thearray's contents, the function will perform N rounds, each round printing N lines for a total of N^2 printsteps.

```
public static void print(int[] arr, int n) {
	for (int i = 0; i<n; i++) {
		for (int j = 0; j<n; j++) {
			System.out.println(arr[j]);
		}
	}
}
```

Let's look at that practically.

Assume the index length N of an array is 10. If the function prints the content of its array in a nested-loop, it will perform 10 rounds, each round printing 10 lines for a total of 100 print steps. This is said to run in $O(N^2)$ time; its total run time increases at an order of magnitude proportional to N^2.

Exponential: $O(2^N)$

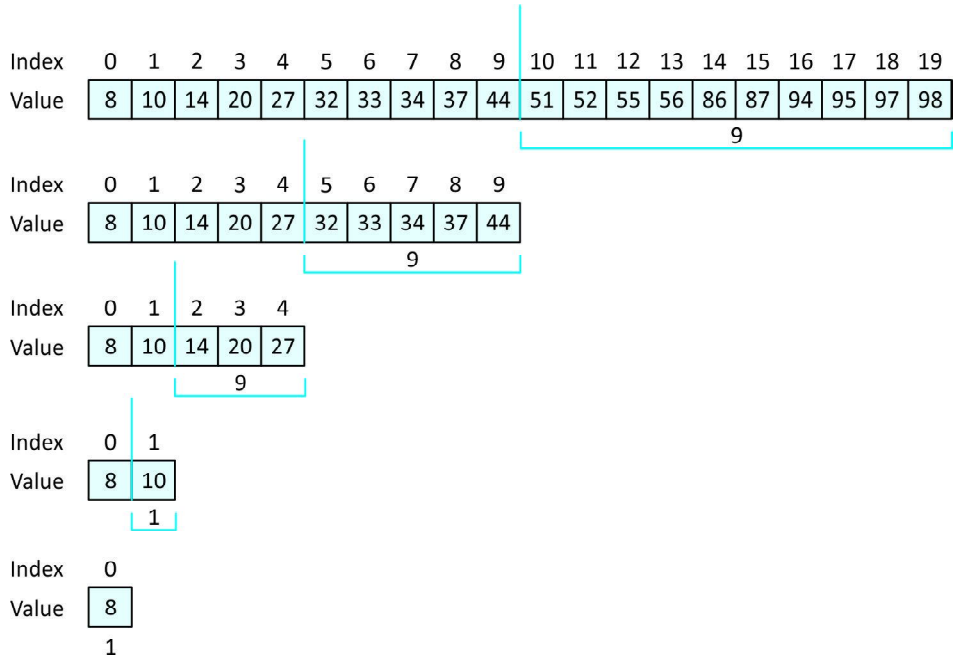
$O(2^N)$ is just one example of exponential growth (among $O(3^N)$, $O(4^N)$, etc.). Time complexity at an exponential rate means that with each step the function performs, its subsequent step will take longer by an order of magnitude equivalent to a factor of N. For instance, with a function whose step-time doubles with each subsequent step, it is said to have a complexity of $O(2^N)$. A function whose step-time triples with each iteration is said to have a complexity of $O(3^N)$ and so on.

Logarithmic Complexity: O(log n)

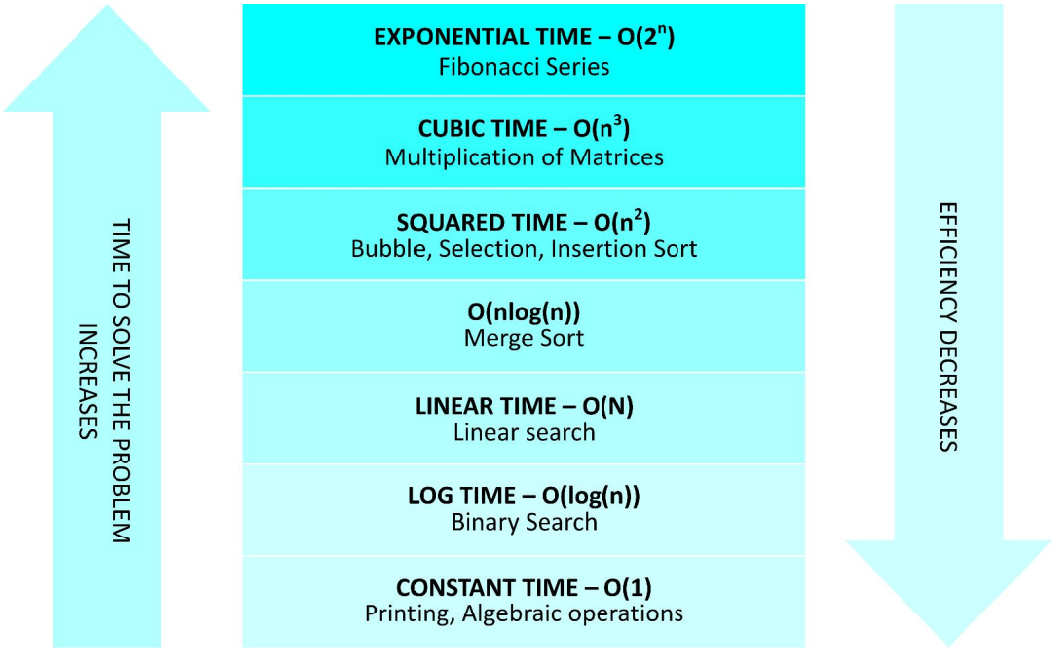
This is the type of algorithm that makes computation blazingly fast. Instead of increasing the time it takes to perform each subsequent step, the time is decreased at magnitude inversely proportional to N.

Let's say we want to search a database for a particular number. In the data set below, we want to search 20 numbers for the number 100. In this example, searching through 20 numbers is no issue. But imagine we're dealing with data sets that store millions of users' profile information. Searching through each index value from beginning to end would be ridiculously inefficient. Especially if it had to be done multiple times.

A logarithmic algorithm that performs a binary search, looks through only half of an increasingly smaller data set per step. Assume we have an ascending ordered set of numbers. The algorithm starts by searching half of the entire data set. If it doesn't find the number, it discards the set just checked and then searches half of the remaining set of numbers.



As illustrated above, each round of searching consists of a smaller data set than the previous, decreasing the time each subsequent round is performed. This makes **log n** algorithms very scalable.



Space Complexity:

Total amount of computer memory required for the execution of an algorithm when expressed in terms of input size is called space complexity. Auxiliary space is the extra space or temporary space used by an algorithm. Space complexity includes both auxiliary space and the space required by the input.

It is always better to use minimum of auxiliary space. Like if we have to calculate the transpose of a 2-D matrix, one way would be to make a new 2-D array and copy the elements from one matrix to another. But this wastes a lot of memory. So a better way would be to convert the original array into its transpose saving a lot of memory.

Well there is always a trade-off between improving time and space complexity. So improving one may lead to bad performance with respect to the other! Thus we need to wisely choose between the two.