



Explore | Expand | Enrich



Explore | Expand | Enrich



Explore | Expand | Enrich

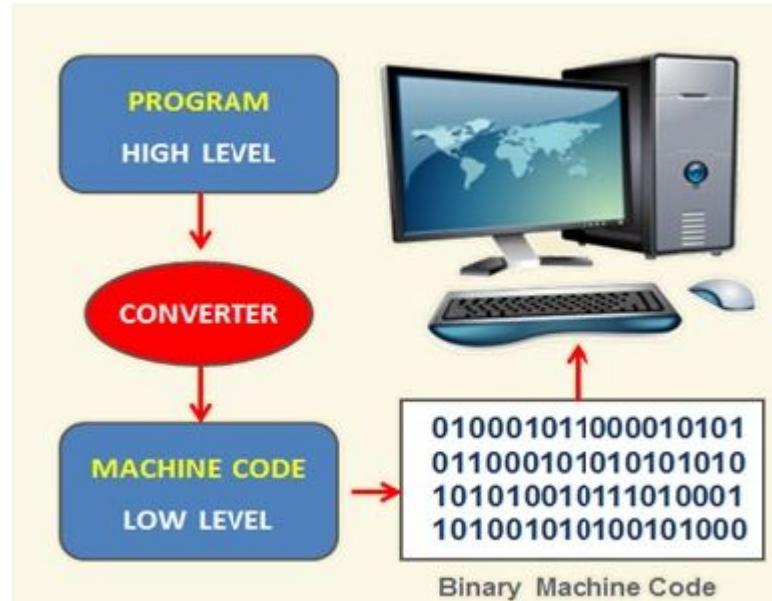
Helping Students  
to Realize Their Dreams  
for the Last 14 Years

## Introduction to Computer Programs

It is a step by step process of designing and developing various sets of computer programs to accomplish a specific computing outcome.

## Elements of Computer Programs

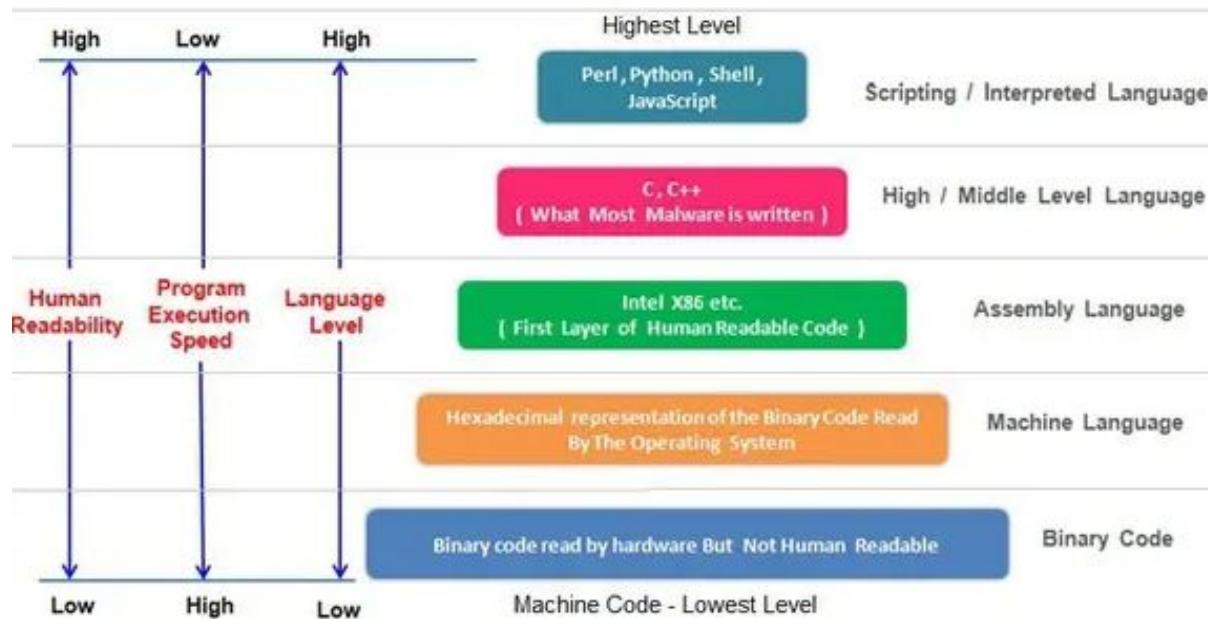
- Programming Environment
- Data Types
- Variables
- Keywords
- Logical and Arithmetical Operators
- If else conditions
- Loops
- Numbers, Characters and Arrays
- Functions
- Input and Output Operations



# Fundamentals of Programming



## Types of Programming Languages



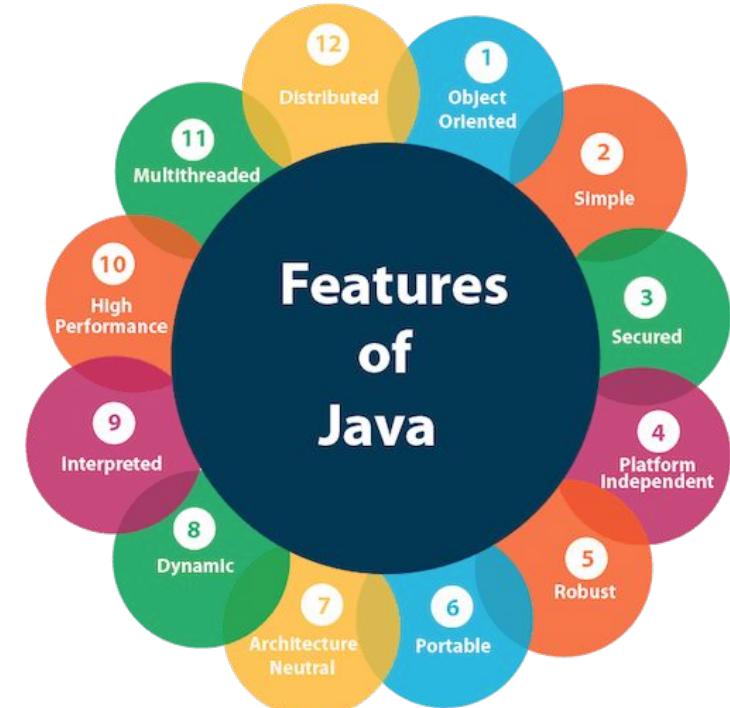
## Competitive Programming

- Competitive programming is a mind sport usually held over the Internet or a local network, involving participants trying to program according to provided specifications.
- One of the oldest contests known is ICPC which originated in the 1970s, and has grown to include 88 countries in its 2011 edition.
- The aim of competitive programming is to write source code of computer programs which are able to solve given problems.
- Typical such tasks belong to one of the following categories: combinatorics, number theory, graph theory, algorithmic game theory, computational geometry, string analysis and data structures
- Competitive programming is typically the first or second round of hiring for SDE roles



## Introduction to Java

- Java is a **programming language** and a **platform**. It is a high level, secure, robust and an Object Oriented programming language
- The syntax of Java is based on C
- There are no pointers in Java and there is automatic garbage collection
- The features of Java are as given in the diagram



# Getting Started with Java



## Example and Structure

```
JavaExample.java ✘
1 package com.beginnersbook;
2 import java.util.Scanner;
3 public class JavaExample
4 {
5     public static void main(String args[])
6     {
7         float p, r, t, sinterest;
8         Scanner scan = new Scanner(System.in);
9         System.out.print("Enter the Principal : ");
10        p = scan.nextFloat();
11        System.out.print("Enter the Rate of interest : ");
12        r = scan.nextFloat();
13        System.out.print("Enter the Time period : ");
14        t = scan.nextFloat();
15        scan.close();
16        sinterest = (p * r * t) / 100;
17        System.out.print("Simple Interest is: " +sinterest);
18    }
19 }
```

```
Problems @ Javadoc Declaration Console ✘ Progress Cover
<terminated> JavaExample [Java Application] /Library/Java/JavaVirtualMachines/jdk-9.jdk
Enter the Principal : 2000
Enter the Rate of interest : 6
Enter the Time period : 3
Simple Interest is: 360.0
```

### Documentation Section

#### Package Statement

#### Import Statements

#### Interface Statements

#### Class Definitions

#### main method class

{

#### main method definition

## Structure of Java Program

# Getting Started with Java

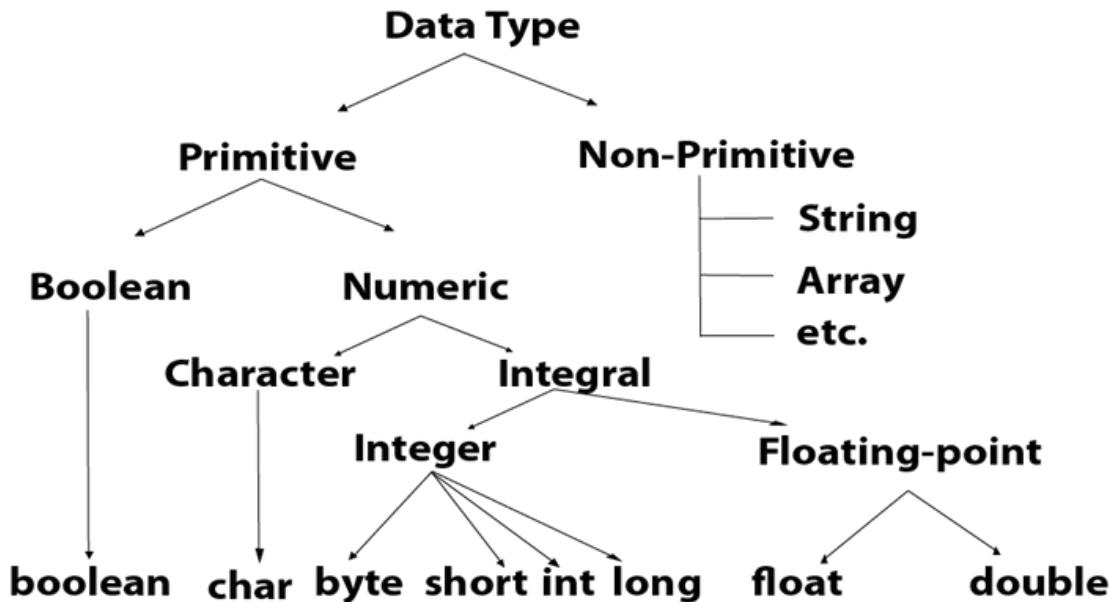


## Java in a nutshell

- Data Types: boolean, int, float, char
- I/O: System.out and Scanner
- Operators: Unary, Arithmetic etc.
- Decision Statements: If, else, if-else, switch
- Looping: while, do-while, for, for-each etc

Operator	Category	Precedence
<b>Unary Operator</b>	postfix	expression++ expression--
	prefix	++expression --expression +expression -expression ~!
<b>Arithmetic Operator</b>	multiplication	* / %
	addition	+ -
<b>Shift Operator</b>	shift	<< >> >>>
<b>Relational Operator</b>	comparison	< > <= >= instanceof
	equality	== !=
<b>Bitwise Operator</b>	bitwise AND	&
	bitwise exclusive OR	^
	bitwise inclusive OR	
<b>Logical Operator</b>	logical AND	&&
	logical OR	
<b>Ternary Operator</b>	ternary	? :
<b>Assignment Operator</b>	assignment	= += -= *= /= %= ^=  = <<= >>= >>>=

## Data Types in Java





# Java: Data Types



Explore | Expand | Enrich

## Primitive Data Types

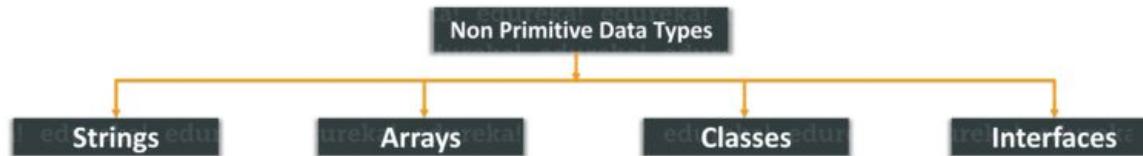
- boolean
- byte
- char
- short
- int
- long
- float
- double

Type	Size in bytes	Range	Default Value
byte	1 byte	-128 to 127	0
short	2 bytes	-32,768 to 32,767	0
int	4 bytes	-2,147,483,648 to 2,147,483, 647	0
long	8 bytes	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	0
float	4 bytes	approximately $\pm 3.40282347E+38F$ (6-7 significant decimal digits) Java implements IEEE 754 standard	0.0f
double	8 bytes	approximately $\pm 1.79769313486231570E+308$ (15 significant decimal digits)	0.0d
char	2 bytes	0 to 65,536 (unsigned)	'\u0000'
boolean	Not precisely defined*	true or false	false



## Non-Primitive Data Types

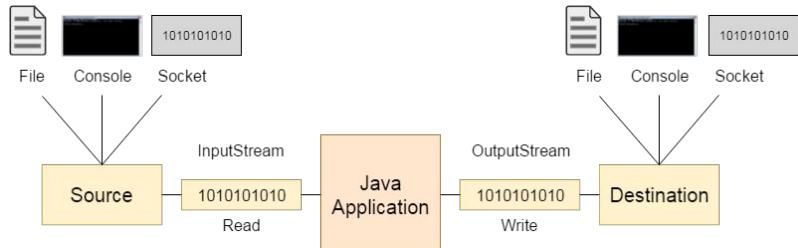
- Strings: String is a sequence of characters. But in Java, a string is an object that represents a sequence of characters. The `java.lang.String` class is used to create a string object.
- Arrays: Arrays in Java are homogeneous data structures implemented in Java as objects. Arrays store one or more values of a specific data type and provide indexed access to store the same. A specific element in an array is accessed by its index.
- Classes: A class in Java is a blueprint which includes all your data. A class contains fields(variables) and methods to describe the behavior of an object.
- Interface: Like a class, an interface can have methods and variables, but the methods declared in interface are by default abstract (only method signature, no body).



# Input and Output

## Java IO

- A stream is a sequence of data. In Java, a stream is composed of bytes.
- There are three default streams in Java:-
  - System.out: standard output stream. Eg: System.out.println("simple message");
  - System.in: standard input stream. int i=System.in.read();
  - System.err: standard error stream. Eg: System.err.println("error message");
- Scanner is a class in java.util package used for obtaining the input of the primitive types like int, double, etc. and strings. It is popularly used for reading from the input stream





# Input and Output



## Java IO - Example

```
1 import java.util.Scanner;
2
3 class Scan {
4     public static void main(String[] ar) {
5
6     Scanner sc = new Scanner(System.in);
7
8     // String input
9     System.out.println("Enter your Name:");
10    String name = sc.nextLine();
11
12    // Character input
13    System.out.println("Enter your Gender:");
14    char gender = sc.next().charAt(0); // it will take the first character
15
16    // Numerical data input
17    System.out.println("Enter your Age:");
18    int age = sc.nextInt();
19    System.out.print("Enter your Phone Number:+91 ");
20    long phoneNo = sc.nextLong();
21    System.out.println("Enter your CGPA:");
22    double CGPA = sc.nextDouble();
23
24    // Printing all the values
25    System.out.println("Name: " + name);
26    System.out.println("Gender: " + gender);
27    System.out.println("Age: " + age);
28    System.out.println("Mobile Number: +91 " + phoneNo);
29    System.out.println("CGPA: " + CGPA);
30 }
31 }
```



# Operators

## Operators in Java

Precedence	Operator	Operand type	Description
1	<code>++, --</code>	Arithmetic	Increment and decrement
1	<code>+, -</code>	Arithmetic	Unary plus and minus
1	<code>~</code>	Integral	Bitwise complement
1	<code>!</code>	Boolean	Logical complement
1	<code>( type )</code>	Any	Cast
2	<code>*, /, %</code>	Arithmetic	Multiplication, division, remainder
3	<code>+, -</code>	Arithmetic	Addition and subtraction
3	<code>+</code>	String	String concatenation
4	<code>&lt;&lt;</code>	Integral	Left shift
4	<code>&gt;&gt;</code>	Integral	Right shift with sign extension
4	<code>&gt;&gt;&gt;</code>	Integral	Right shift with no extension
5	<code>&lt;, &lt;=, &gt;, &gt;=</code>	Arithmetic	Numeric comparison
5	<code>instanceof</code>	Object	Type comparison
6	<code>==, !=</code>	Primitive	Equality and inequality of value
6	<code>==, !=</code>	Object	Equality and inequality of reference
7	<code>&amp;</code>	Integral	Bitwise AND
7	<code>&amp;</code>	Boolean	Boolean AND
8	<code>^</code>	Integral	Bitwise XOR
8	<code>^</code>	Boolean	Boolean XOR
9	<code> </code>	Integral	Bitwise OR
9	<code> </code>	Boolean	Boolean OR
10	<code>&amp;&amp;</code>	Boolean	Conditional AND
11	<code>  </code>	Boolean	Conditional OR
12	<code>?:</code>	N/A	Conditional ternary operator
13	<code>=</code>	Any	Assignment



# Operators



## Associativity and Precedence of Operators

Operator	Precedence
Postfix	Expression++, expression--
Unary	++expression, --expression, +expression, -expression, !
Multiplication	* (multiply), / (divide), % (remainder)
Addition	+ (add), - (subtract)
Relational	<, >, <=, >=
Equality	==, !=
Logical AND	&&
Logical OR	
Assignment	=, +=, -=, *=, /=, %=





# Decision Making



## Decision Making in Java

The decision making statements in Java include the following keywords:-

- if Statement
  - if: An if statement consists of a boolean expression followed by one or more statements.
  - if-else: An if statement can be followed by an optional else statement, which executes when the boolean expression is false.
  - nested-if: You can use one if or else if statement inside another if or else if statement(s).
- switch statement
- ternary operator



## Decision Making in Java

### Condition is true

```
int number = 5;  
  
if (number > 0) {  
    // code  
}  
else {  
    // code  
}  
  
// code after if...else
```

### Condition is false

```
int number = 5;  
  
if (number < 0) {  
    // code  
}  
else {  
    // code  
}  
  
// code after if...else
```

## Ternary operator and Switch Statement

- Ternary operator in Java will be in the form of **condition?true-expression:false-expression**
- Switch statement executes one statement from multiple conditions.
- The switch statement works with byte, short, int, long, enum types, String and some wrapper types like Byte, Short, Int, and Long.
- Since Java 7, you can use strings in the switch statement.

```
switch(expression) {  
    case x:  
        // code block  
        break;  
    case y:  
        // code block  
        break;  
    default:  
        // code block  
}
```



# Introduction to Algorithms



Explore | Expand | Enrich

## Algorithms

An algorithm is a step-by-step method for solving some problem.

**Definition:** “In mathematics and computer science, an algorithm is a finite sequence of well-defined, computer-implementable instructions, typically to solve a class of specific problems or to perform a computation.”

We need algorithms because of the following reasons:

- Scalability: It helps us to understand the scalability. When we have a big real-world problem, we need to scale it down into small-small steps to easily analyze the problem.
- Performance: The real-world is not easily broken down into smaller steps. If the problem can be easily broken into smaller steps means that the problem is feasible.





# Introduction to Algorithms

## Characteristics of Algorithm

Algorithms generally have the following characteristics:

- **Clear and Unambiguous:** Algorithm should be clear and unambiguous. Each of its steps should be clear in all aspects and must lead to only one meaning.
- **Well-Defined Inputs:** If an algorithm says to take inputs, it should be well-defined inputs.
- **Well-Defined Outputs:** The algorithm must clearly define what output will be yielded and it should be well-defined as well.
- **Finiteness:** The algorithm must be finite, i.e. it should not end up in an infinite loops or similar.
- **Feasible:** The algorithm must be simple, generic and practical, such that it can be executed upon will the available resources.
- **Language Independent:** The Algorithm designed must be language-independent, i.e. it must be just plain instructions that can be implemented in any language, and yet the output will be same, as expected.

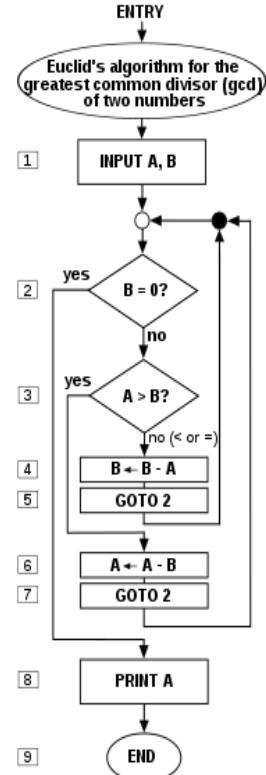


# Introduction to Algorithms

## Dataflow of an Algorithm

- **Problem:** A problem can be a real-world problem or any instance from the real-world problem for which we need to create a program or the set of instructions. The set of instructions is known as an algorithm.
- **Algorithm:** An algorithm will be designed for a problem which is a step by step procedure.
- **Input:** After designing an algorithm, the required and the desired inputs are provided to the algorithm.
- **Processing unit:** The input will be given to the processing unit, and the processing unit will produce the desired output.
- **Output:** The output is the outcome or the result of the program.

On the right side, the flow of Euclid's algorithm to calculate GCD of two numbers are given.



## Issues while designing Algorithms

### **How to design algorithms:**

As we know that an algorithm is a step-by-step procedure so we must follow some steps to design an algorithm.

Designing algorithm is not a one step process, we need different approaches for different algorithms

### **How to analyse algorithm efficiency**

We cannot determine efficiency of an algorithm by benchmarking its speed. We need a sophisticated method to analyse an efficiency of an algorithm.

## Issues while designing Algorithms

### **How to design algorithms:**

As we know that an algorithm is a step-by-step procedure so we must follow some steps to design an algorithm.

Designing algorithm is not a one step process, we need different approaches for different algorithms

### **How to analyse algorithm efficiency**

We cannot determine efficiency of an algorithm by benchmarking its speed. We need a sophisticated method to analyse an efficiency of an algorithm.

Types of approaches to solve while designing an algorithm:-

- 1. Brute force approach
- 2. Divide and Conquer
- 3. Greedy approach
- 4. Dynamic programming
- 5. Branch and Bound
- 6. Randomized Algorithm
- 7. Backtracking



## Algorithm Complexity

We need to learn how to compare the performance different algorithms and choose the best one to solve a particular problem.

While analyzing an algorithm, we mostly consider time complexity and space complexity.

Time complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the input.

Space complexity of an algorithm quantifies the amount of space or memory taken by an algorithm to run as a function of the length of the input.





## Time and Space Complexity



Explore | Expand | Enrich

Time and space complexity depends on lots of things like hardware, operating system, processors, etc.

However, we don't consider any of these factors while analyzing the algorithm. We will only consider the execution time of an algorithm.

Lets start with a simple example. Suppose you are given an array  $A$  and an integer  $x$  and you have to find if  $x$  exists in array  $A$ .

- Simple solution to this problem is traverse the whole array  $A$  and check if the any element is equal to  $x$

```
for i : 1 to length of A  
    if A[i] is equal to x  
        return TRUE  
    return FALSE
```





# Time and Space Complexity



```
for i : 1 to length of A  
    if A[i] is equal to x  
        return TRUE  
return FALSE
```

Consider the above problem.

Each of the operation in computer take approximately constant time.

- Let each operation takes 'c' time. The number of lines of code executed is actually depends on the value of 'x'.

The worst case scenario: The if condition will run 'x' if the array isn't found

Best Case scenario: The if case will be run only once (element will be found in the first pass

The space requirement is constant. It is so because we store only an array and no additional space is required





# Big O



Explore | Expand | Enrich

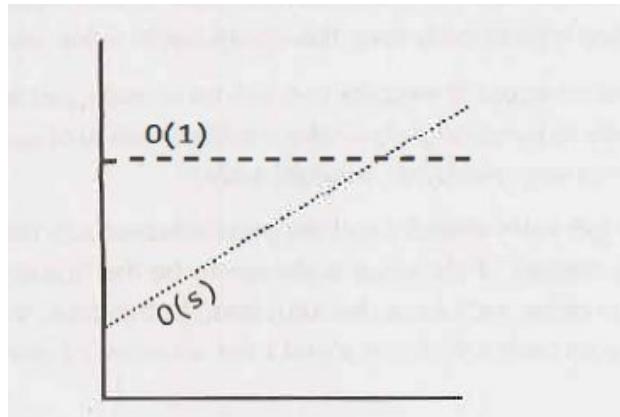
No matter how big the constant is and how slow the linear increase is, linear will at some point surpass constant.

There are many more runtimes than this. Some of the most common ones are  $O(\log N)$ ,  $O(N \log N)$ ,  $O(N)$ ,  $O(N^2)$  and  $O(2^N)$ .

There's no fixed list of possible runtimes, though.

You can also have multiple variables in your runtime. For example, the time to paint a fence that's  $w$  meters wide and  $h$  meters high could be described as  $O(w h)$ .

If you needed  $p$  layers of paint, then you could say that the time is  $O(w h p)$ .





# Big 0, Big Theta, and Big Omega



big 0, big theta, and big omega are used to describe runtimes.

O (big Oh): Big O describes an upper bound on the time. An algorithm that prints all the values in an array could be described as  $O(N)$ , but it could also be described as  $O(N^2)$ ,  $O(N^3)$ , or  $O(2N)$  (or many other big O times).

The algorithm is at least as fast as each of these; therefore they are upper bounds on the runtime. This is similar to a less-than-or-equal-to relationship.

- A simple algorithm to print the values in an array is  $O(N)$

big omega: Omega is the equivalent concept but for lower bound. Printing the values in an array is  $O(N)$  as well as  $O(\log N)$  and  $O(1)$ . After all, you know that it won't be faster than those runtimes.

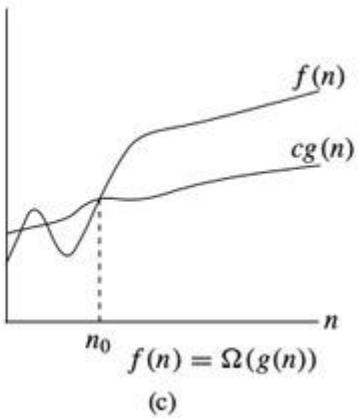
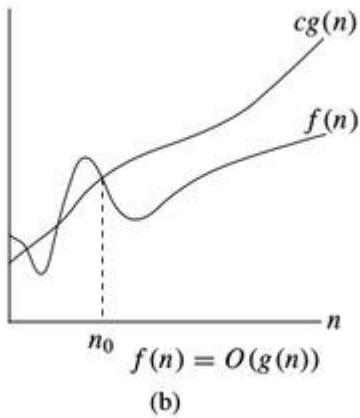
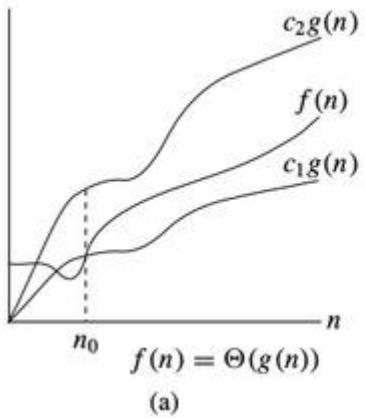
big theta: Theta means both O and Omega. That is, an algorithm is  $\Theta(N)$  if it is both  $O(N)$  and  $\Omega(N)$



# Big 0, Big Theta, and Big Omega



Explore | Expand | Enrich





# Time Complexity



Explore | Expand | Enrich

The time complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the input.

Note that the time to run is a function of the length of the input and not the actual execution time of the machine on which the algorithm is running on.

This is what the concept of asymptotic runtime, or big O time, means. We could describe the data transfer "algorithm" runtime as:

Electronic Transfer:  $O(s)$ , where  $s$  is the size of the file. This means that the time to transfer the file increases linearly with the size of the file. (Yes, this is a bit of a simplification, but that's okay for these purposes.)

Airplane Transfer:  $O(1)$  with respect to the size of the file. As the size of the file increases, it won't take any longer to get the file to your friend. The time is constant.



# Space Complexity



Time is not the only thing that matters in an algorithm. We might also care about the amount of memory or space-required by an algorithm.

Space complexity is a parallel concept to time complexity. If we need to create an array of size  $n$ , this will require  $O(n)$  space.

If we need a two-dimensional array of size  $n \times n$ , this will require  $O(n^2)$  space.

```
int sum(int n) { /* Ex 1.*/
    if (n <= 0) {
        return 0;
    }
    return n + sum(n-1);
}
```

```
int pairSumSequence(int n) { /* Ex 2.*/
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += pairSum(i, i + 1);
    }
    return sum;
}

int pairSum(int a, int b) {
    return a + b;
}
```

Ex.1: Each call adds to the call stack ' $n$ ' times. Hence space complexity is  $O(n)$ . However, just because you have  $n$  calls total doesn't mean it takes  $O(n)$  space. Consider Ex 2. There will be roughly  $O(n)$  calls to `pairSum`. However, those calls do not exist simultaneously on the call stack, so you only need  $O(1)$  space.

# Conditions



It is very possible for  $O(N)$  code to run faster than  $O(1)$  code for specific inputs. Big O just describes the rate of increase.

For this reason, we drop the constants in runtime. An algorithm that one might have described as  $O(2N)$  is actually  $O(N)$ .

For Eg:

```
for(i=0;i<N;i++){  
    print(x[1])  
    print(y[1])  
}
```

```
for(i=0;i<N;i++){  
    print(x[1])  
}  
for(i=0;i<N;i++){  
    print(y[1])  
}
```

Both can be considered the same. The first is  $O(n)$  and second is  $O(2n)$  which can be represented as  $O(n)$

The second rule is that you can drop the terms which aren't significant. In this case:-

- $O(N^2 + N)$  becomes  $O(N^2)$ .
- $O(N + \log N)$  becomes  $O(N)$ .
- $O(5*2^N + 1000N^{100})$  becomes  $O(2^N)$ .

# Examples

The Time complexity of this code will be  $O(N^2)$  as for  $i=0$ , it will run 0 times,  $i=2$  it will run 2 times and hence  $O(N^2)$

$$; 0 + 1 + 2 + \dots + (N - 1) = \frac{N*(N-1)}{2}$$

```
int count = 0;
for (int i = 0; i < N; i++)
    for (int j = 0; j < i; j++)
        count++;
```

An algorithm like binary search takes  $O(\log N)$  as the time complexity as in every iteration array  $N$  is divided into  $N/2$  times.

- When  $i=N$ , it will do  $N/2$  comparisons in the worst case.  
When  $i=N/2$ , it will do  $N/4$  comparisons in the worst case

Hence the time complexity of binary search is  $O(\log N)$ .

```
function binary_search(A, n, T) is
    L := 0
    R := n - 1
    while L ≤ R do
        m := floor((L + R) / 2)
        if A[m] < T then
            L := m + 1
        else if A[m] > T then
            R := m - 1
        else:
            return m
    return unsuccessful
```

# Examples



```
int count = 0;  
for (int i = N; i > 0; i /= 2)  
    for (int j = 0; j < i; j++)  
        count++;
```

This is a tricky case. In the first look, it seems like the complexity is  $O(N * \log N)$ .  $N$  for the  $j$ 's loop and  $\log N$  for  $i$ 's loop. But its wrong. Lets see why.

Think about how many times `count++` will run.

When  $i = N$ , it will run  $N$  times.

When  $i = N/2$ , it will run  $N/2$  times.

When  $i = N/4$ , it will run  $N/4$  times and so on.

Total number of times `count++` will run is  $N + N/2 + N/4 + \dots + 1 = 2 * N$ . So the time complexity will be  $O(N)$ .



# Sieve of Eratosthenes



Explore | Expand | Enrich

## Introduction

The sieve of Eratosthenes is an ancient algorithm for finding all prime numbers up to any given limit.

It does so by iteratively marking as composite (i.e., not prime) the multiples of each prime, starting with the first prime number, 2.

The multiples of a given prime are generated as a sequence of numbers starting from that prime, with constant difference between them that is equal to that prime

- This is the sieve's key distinction from using trial division to sequentially test each candidate number for divisibility by each prime.

Once all the multiples of each discovered prime have been marked as composites, the remaining unmarked numbers are primes.





# Sieve of Eratosthenes

## The Sieve in Action

The steps of the algorithm to find all primes below 121

	2	3	4	5	6	7	8	9	10	Prime numbers
11	12	13	14	15	16	17	18	19	20	
21	22	23	24	25	26	27	28	29	30	
31	32	33	34	35	36	37	38	39	40	
41	42	43	44	45	46	47	48	49	50	
51	52	53	54	55	56	57	58	59	60	
61	62	63	64	65	66	67	68	69	70	
71	72	73	74	75	76	77	78	79	80	
81	82	83	84	85	86	87	88	89	90	
91	92	93	94	95	96	97	98	99	100	
101	102	103	104	105	106	107	108	109	110	
111	112	113	114	115	116	117	118	119	120	

Explore | Expand | Enrich



# Sieve of Eratosthenes



Explore | Expand | Enrich

## Algorithm

To find all the prime numbers less than or equal to 30, proceed as follows.

First, generate a list of integers from 2 to 30

2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30

The first number in the list is 2; cross out every 2nd number in the list after 2 by counting up from 2 in increments of 2 (these will be all the multiples of 2 in the list):

2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ 9 ~~10~~ 11 ~~12~~ 13 ~~14~~ 15 ~~16~~ 17 ~~18~~ 19 ~~20~~ 21 ~~22~~ 23 ~~24~~ 25 ~~26~~ 27 ~~28~~ 29 30

The next number in the list after 2 is 3; cross out every 3rd number in the list after 3 by counting up from 3 in increments of 3 (these will be all the multiples of 3 in the list)

2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ 9 ~~10~~ 11 ~~12~~ 13 ~~14~~ ~~15~~ ~~16~~ 17 ~~18~~ 19 ~~20~~ ~~21~~ ~~22~~ 23 ~~24~~ 25 ~~26~~ ~~27~~ ~~28~~ 29 30

The next number not yet crossed out in the list after 3 is 5; cross out every 5th number in the list after 5 by counting up from 5 in increments of 5 (i.e. all the multiples of 5):

2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ 9 ~~10~~ 11 ~~12~~ 13 ~~14~~ ~~15~~ ~~16~~ 17 ~~18~~ 19 ~~20~~ ~~21~~ ~~22~~ 23 ~~24~~ ~~25~~ ~~26~~ ~~27~~ ~~28~~ 29 30

The next number not yet crossed out in the list after 5 is 7; the next step would be to cross out every 7th number in the list after 7, but they are all already crossed out at this point, as these numbers (14, 21, 28) are also multiples of smaller primes because  $7 \times 7$  is greater than 30. The numbers not crossed out at this point in the list are all the prime numbers below 30:

2 3 5 7 11 13 17 19 23 29





# Sieve of Eratosthenes



## Algorithm

algorithm Sieve of Eratosthenes is

  input: an integer  $n > 1$ .

  output: all prime numbers from 2 through  $n$ .

let  $A$  be an array of Boolean values, indexed by integers 2 to  $n$ ,  
initially all set to true.

for  $i = 2, 3, 4, \dots$ , not exceeding  $\sqrt{n}$  do

  if  $A[i]$  is true

    for  $j = i^2, i^2+i, i^2+2i, i^2+3i, \dots$ , not exceeding  $n$  do

$A[j] := \text{false}$

return all  $i$  such that  $A[i]$  is true.

Time Complexity:  $O(n \log(\log n))$





# Sieve of Eratosthenes



Explore | Expand | Enrich

Program: Sieve of Eratosthenes in Java

## Sample Input/Output

Input : n =10

Output : 2 3 5 7

Input : n = 20

Output: 2 3 5 7 11 13 17 19





# Sieve of Eratosthenes



## Other types of Sieves

The simple sieve algorithm produces all primes not greater than  $n$ .

It includes a common optimization, which is to start enumerating the multiples of each prime  $i$  from  $i^2$ .

The time complexity of this algorithm is  $O(n \log \log n)$ , provided the array update is an  $O(1)$  operation, as is usually the case.

The problem with the sieve of Eratosthenes is not the number of operations it performs but rather its memory requirements.

For large  $n$ , the range of primes may not fit in memory; worse, even for moderate  $n$ , its cache use is highly suboptimal.

The simple sieve algorithm walks through the entire array  $A$ , exhibiting almost no locality of reference

**Locality of Reference:** In computer science, locality of reference, also known as the principle of locality, is the tendency of a processor to access the same set of memory locations repetitively over a short period of time





# Segmented Sieves



Explore | Expand | Enrich

## Introduction

A solution to these problems is offered by segmented sieves, where only portions of the range are sieved at a time.

The idea of a segmented sieve is to divide the range  $[0..n-1]$  in different segments and compute primes in all segments one by one.

This algorithm first uses Simple Sieve to find primes smaller than or equal to  $\sqrt{n}$ .

- In Simple Sieve, we needed  $O(n)$  space which may not be feasible for large  $n$ .

Here we need  $O(\sqrt{n})$  space and we process smaller ranges at a time (locality of reference)





# Segmented Sieves



Explore | Expand | Enrich

## Algorithm: Segmented Sieves

- Use Simple Sieve to find all primes up to the square root of 'n' and store these primes in an array "prime[]". Store the found primes in an array 'prime[]'.
- We need all primes in the range [0..n-1]. We divide this range into different segments such that the size of every segment is at-most  $\sqrt{n}$
- Do following for every segment [low..high]
  - Create an array mark[high-low+1]. Here we need only  $O(x)$  space where x is a number of elements in a given range.
  - Iterate through all primes found in step 1. For every prime, mark its multiples in the given range [low..high].



# Segmented Sieves



Program: Segmented Sieves

Sample I/O

Input

n=100

Output

2 3 5 7 11 13 17 19 23 29 31 37 41  
43 47 53 59 61 67 71 73 79 83 89 97

Note

The time complexity (or a number of operations) by Segmented Sieve is the same as Simple Sieve.

It has advantages for large 'n' as it has better locality of reference thus allowing better caching by the CPU and also requires less memory space.





# Incremental Sieve



Explore | Expand | Enrich

## Introduction

The standard Sieve of Eratosthenes requires that you specify an upper bound before you start

But often, programs that use prime numbers don't know the upper bound in advance, or don't want to pre-compute and store all of the primes up to their bound.

In such cases, Incremental Sieve algorithm can be used

An incremental formulation of the sieve generates primes indefinitely (i.e. without an upper bound) by

- interleaving the generation of primes with the generation of their multiples (so that primes can be found in gaps between the multiples), where the multiples of each prime  $p$  are generated directly, by counting up from the square of the prime in increments of  $p$  (or  $2p$  for odd primes)

The generation must be initiated only when the prime's square is reached, to avoid adverse effects on efficiency. It can be expressed symbolically under the dataflow paradigm as

$$\text{primes} = [2, 3, \dots] \setminus [[p^2, p^2+p, \dots] \text{ for } p \text{ in primes}]$$

Where  $\setminus$  denotes the set subtraction of arithmetic progressions of numbers.





# Euler's Phi Algorithm



Explore | Expand | Enrich

## Introduction

Euler's totient function, also known as phi-function  $\phi(n)$ , counts the number of integers between 1 and  $n$  inclusive, which are coprime to  $n$ .

Two numbers are coprime if their greatest common divisor equals 1 (1 is considered to be coprime to any number).

Here are values of  $\phi(n)$  for the first few positive integers:

$n$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
$\phi(n)$	1	1	2	2	4	2	6	4	6	4	10	4	12	6	8	8	16	6	18	8	12

Examples:

$$\Phi(4) = 2$$

$\text{gcd}(1, 4)$  is 1 and  $\text{gcd}(3, 4)$  is 1

$$\Phi(5) = 4$$

$\text{gcd}(1, 5)$  is 1,  $\text{gcd}(2, 5)$  is 1,  
 $\text{gcd}(3, 5)$  is 1 and  $\text{gcd}(4, 5)$  is 1

$$\Phi(6) = 2$$

$\text{gcd}(1, 6)$  is 1 and  $\text{gcd}(5, 6)$  is 1,





# Euler's Phi Algorithm



Explore | Expand | Enrich

## Basic Algorithm

A simple solution is to iterate through all numbers from 1 to  $n-1$  and count numbers with gcd with  $n$

EulerPhi1.java

Time Complexity:  $O(N \log N)$





## Euler's Phi Algorithm



Explore | Expand | Enrich

### Better Solution

The idea is based on Euler's product formula which states that the value of totient functions is below the product overall prime factors p of n.

$$\phi(n) = n \prod_{p|n} \left(1 - \frac{1}{p}\right)$$

- The formula basically says that the value of  $\Phi(n)$  is equal to n multiplied by-product of  $(1 - 1/p)$  for all prime factors p of n. For example value of  $\Phi(6) = 6 * (1-1/2) * (1 - 1/3) = 2$ .





# Euler's Phi Algorithm



Explore | Expand | Enrich

## Better Solution - 1

Program: EulerPhi2.java

### Algorithm

- 1) Initialize : result = n
- 2) Run a loop from 'p' = 2 to  $\sqrt{n}$ , do following for every 'p'.
  - a) If p divides n, then
    - Set:  $\text{result} = \text{result} * (1.0 - (1.0 / (\text{float}) p))$ ;
    - Divide all occurrences of p in n.
- 3) Return result



# Euler's Phi Algorithm



## Better Solution - 2

Program: EulerPhi3.java

We can avoid floating-point calculations in the above method. The idea is to count all prime factors and their multiples and subtract this count from n to get the totient function value  
(Prime factors and multiples of prime factors won't have GCD as 1)

### Algorithm

- - 1) Initialize result as n
  - 2) Consider every number 'p' (where 'p' varies from 2 to  $\Phi n$ ).
    - If p divides n, then do following
      - a) Subtract all multiples of p from 1 to n [all multiples of p will have gcd more than 1 (at least p) with n]
      - b) Update n by repeatedly dividing it by p.
  - 3) If the reduced n is more than 1, then remove all multiples of n from result.





# Strobogrammatic Number



Explore | Expand | Enrich

## Introduction

Strobogrammatic Number is a number whose numeral is rotationally symmetric so that it appears the same when rotated 180 degrees.

In other words, Strobogrammatic Number appears the same right-side up and upside down.

When written using standard characters (ASCII), the numbers, 0, 1, 8 are symmetrical around the horizontal axis, and 6 and 9 are the same as each other when rotated 180 degrees.

- In such a system, the first few strobogrammatic numbers are:

0, 1, 8, 11, 69, 88, 96, 101, 111, 181, 609, 619, 689, 808, 818, 888.....

The years 1881 and 1961 were the most recent strobogrammatic years; the next strobogrammatic year will be 6009.





# Strobogrammatic Number



Explore | Expand | Enrich

## Examples

0 after 180° rotation : (0 → 0)

1 after 180° rotation : (1 → 1)

8 after 180° rotation : (8 → 8)

6 after 180° rotation : (6 → 9)

9 after 180° rotation : (9 → 6)





# Strobogrammatic Number



Explore | Expand | Enrich

## Program

StrobogrammaticNumber.java

For the given length n, find all n-length Strobogrammatic numbers.





# Strobogrammatic Number



Explore | Expand | Enrich

## Program

StrobogrammaticNumber2.java

Given number n, check if it is a Strobogrammatic Number.

### Sample IO

Input: "69"

- Output: true

Input: "88"

Output: true

Input: "962"

Output: false





# Remainder Theorem



Explore | Expand | Enrich

## Introduction

Also known as the Chinese Remainder Theorem

The Chinese remainder theorem states that if one knows the remainders of the Euclidean division of an integer  $x$  by several integers, then one can determine uniquely the remainder of the division of  $x$  by the product of these integers, under the condition that the divisors are pairwise coprime.

We are given two arrays  $\text{num}[0..k-1]$  and  $\text{rem}[0..k-1]$ .

- In  $\text{num}[0..k-1]$ , every pair is coprime (gcd for every pair is 1). We need to find minimum positive number  $x$  such that:

$$\begin{aligned}x \% \text{num}[0] &= \text{rem}[0], \\x \% \text{num}[1] &= \text{rem}[1],\end{aligned}$$

.....  
.....  
.....

$$x \% \text{num}[k-1] = \text{rem}[k-1]$$

Note that the integers in the  $\text{num}$  array would be pairwise coprime.





# Remainder Theorem



Explore | Expand | Enrich

## Explanation

Basically, we are given  $k$  numbers which are pairwise coprime, and given remainders of these numbers when an unknown number  $x$  is divided by them.

We need to find the minimum possible value of  $x$  that produces given remainders.

Chinese Remainder Theorem states that there always exists an  $x$  that satisfies given congruences

Let  $\text{nums} = [n_1, n_2, n_3 \dots n_k]$  and  $\text{rems} = [a_1, a_2, a_3 \dots a_k]$ . Then, for any given sequence of integers  $\text{rem}[0]$ ,

- there exists an integer  $x$  solving the following system of simultaneous congruences.

### THEOREM

#### Chinese Remainder Theorem

Given pairwise coprime positive integers  $n_1, n_2, \dots, n_k$  and arbitrary integers  $a_1, a_2, \dots, a_k$ , the system of simultaneous congruences

$$x \equiv a_1 \pmod{n_1}$$

$$x \equiv a_2 \pmod{n_2}$$

⋮

$$x \equiv a_k \pmod{n_k}$$

has a solution, and the solution is unique modulo  $N = n_1 n_2 \cdots n_k$ .





# Remainder Theorem



## Program

ChineseRemainderThm.java

### Sample IO

#### Input

num= [5, 7]  
rem = [1, 3]

#### Output

31

#### Input

num= [3, 4, 5]  
rem = [2, 3, 1]

#### Output

11

### Explanation

31 is the smallest number such that:

- (1) When we divide it by 5, we get remainder 1.
- (2) When we divide it by 7, we get remainder 3.

### Explanation

11 is the smallest number such that:

- (1) When we divide it by 3, we get remainder 2.
- (2) When we divide it by 4, we get remainder 3.
- (3) When we divide it by 5, we get remainder 1.



## Introduction

You are given an integer 'X', you need to convert the integer to binary format and check if the binary format is palindrome or not

For Example, 5 i.e. 101, 27 i.e. 11011 are numbers whose binary representations are palindromes. Whereas 10 i.e. 1011 and 20 i.e. 10100 are not palindromes

- The problem is very similar to checking whether a string is palindrome or not. Hence the binary representation must be strings.

We start from leftmost and rightmost bits and compare bits one by one. If we find a mismatch, then return false.

We can use the regular palindrome program. Converting integer 'val' to binary is as easy as `Integer.toBinaryString(val)`



# Binary Palindrome



## Program

Check whether a string is palindrome or not

palindrome1.java

### Sample IO

Input: madam

Output: The string is palindrome

Input: deed

Output: The string is palindrome

Input: algorithm

Output: The string is not palindrome





## Binary Palindrome



Explore | Expand | Enrich

Find the nth number whose binary representation is a palindrome

Find the nth number whose binary representation is a palindrome.

A special caution that you should not consider the leading zeros, while considering the binary representation.

Approach: Traverse through all the integers from 1 to  $2^{31} - 1$  and increment palindrome count, if the number is a palindrome. When the palindrome count reaches the required n, break the loop and return the current integer.



# Binary Palindrome



Explore | Expand | Enrich

## Programs

### palindrome2.java

#### Sample IO

Input : 1

Output : 1

1st Number whose binary representation  
is palindrome is 1 (1)



Input : 9

Output : 27

9th Number whose binary representation  
is palindrome is 27 (11011)





# Booth's Algorithm



## Introduction

The booth algorithm is a multiplication algorithm that allows us to multiply the two signed binary integers in 2's complement, respectively.

It is also used to speed up the performance of the multiplication process. It is very efficient too.

- It works on the string bits 0's in the multiplier that requires no additional bit only shift the right-most string bits and a string of 1's in a multiplier bit weight  $2^k$  to weight  $2^m$  that can be considered as  $2^k + 1 - 2^m$ .
- 



# Booth's Algorithm



## Flowchart

As in all multiplication schemes, booth algorithm requires examination of the multiplier bits and shifting of the partial product.

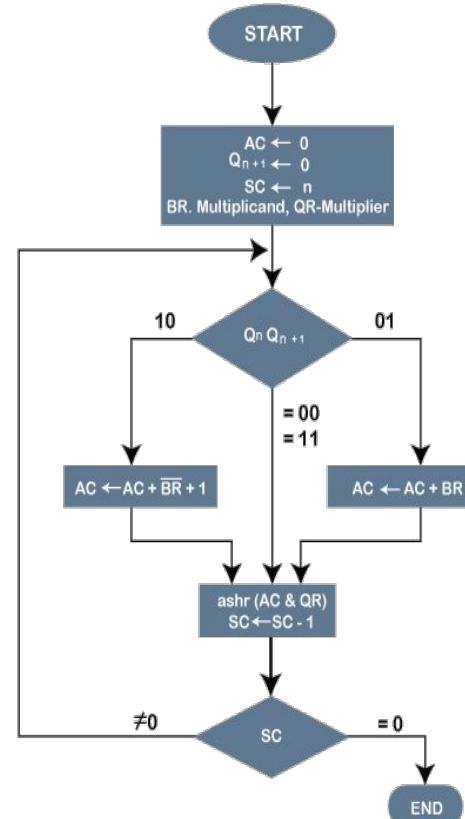
Prior to the shifting, the multiplicand may be added to the partial product, subtracted from the partial product, or left unchanged according to following rules:

1. The multiplicand is subtracted from the partial product upon encountering the first least significant 1 in a string of 1's in the multiplier
2. The multiplicand is added to the partial product upon encountering the first 0 (provided that there was a previous '1') in a string of 0's in the multiplier.
3. The partial product does not change when the multiplier bit is identical to the previous multiplier bit.

# Booth's Algorithm

## Flowchart

- Initially, AC and Q<sub>n+1</sub> bits are set to 0
- The SC is a sequence counter that represents the total bits set n, which is equal to the number of bits in the multiplier.
- There are BR that represent the multiplicand bits, and QR represents the multiplier bits.
- After that, we encountered two bits of the multiplier as Q<sub>n</sub> and Q<sub>n+1</sub>, where Q<sub>n</sub> represents the last bit of QR, and Q<sub>n+1</sub> represents the incremented bit of Q<sub>n</sub> by 1.
- Suppose two bits of the multiplier is equal to 10; it means that we have to subtract the multiplier from the partial product in the accumulator AC and then perform the arithmetic shift operation (ashr).
- If the two of the multipliers equal to 01, it means we need to perform the addition of the multiplicand to the partial product in accumulator AC and then perform the arithmetic shift operation (ashr), including Q<sub>n+1</sub>.
- The arithmetic shift operation is used in Booth's algorithm to shift AC and QR bits to the right by one and remains the sign bit in AC unchanged.
- And the sequence counter is continuously decremented till the computational loop is repeated, equal to the number of bits (n).





# Booth's Algorithm

## Algorithm

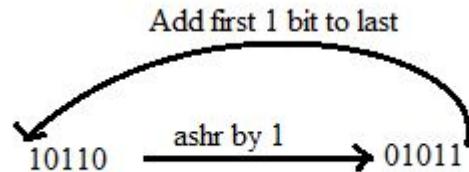
1. Set the Multiplicand and Multiplier binary bits as M and Q, respectively.
2. Initially, we set the AC and  $Q_n + 1$  registers value to 0.
3. SC represents the number of Multiplier bits (Q), and it is a sequence counter that is continuously decremented till equal to the number of bits (n) or reached to 0.
4. A  $Q_n$  represents the last bit of the Q, and the  $Q_{n+1}$  shows the incremented bit of  $Q_n$  by 1.
5. On each cycle of the booth algorithm,  $Q_n$  and  $Q_{n+1}$  bits will be checked on the following parameters as follows:
  1. When two bits  $Q_n$  and  $Q_{n+1}$  are 00 or 11, we simply perform the arithmetic shift right operation (ashr) to the partial product AC. And the bits of  $Q_n$  and  $Q_{n+1}$  is incremented by 1 bit.
  2. If the bits of  $Q_n$  and  $Q_{n+1}$  is shows to 01, the multiplicand bits (M) will be added to the AC (Accumulator register). After that, we perform the right shift operation to the AC and QR bits by 1.
  3. If the bits of  $Q_n$  and  $Q_{n+1}$  is shows to 10, the multiplicand bits (M) will be subtracted from the AC (Accumulator register). After that, we perform the right shift operation to the AC and QR bits by 1.
6. The operation continuously works till we reached  $n - 1$  bit in the booth algorithm.
7. Results of the Multiplication binary bits will be stored in the AC and QR registers.

# Booth's Algorithm

## Methods used in Booth's algorithm

### RSC (Right Shift Circular)

It shifts the right-most bit of the binary number, and then it is added to the beginning of the binary bits.



### RSA (Right Shift Arithmetic)

It adds the two binary bits and then shift the result to the right by 1-bit position.

Example:  $0100 + 0110 \Rightarrow 1010$ , after adding the binary number shift each bit by 1 to the right and put the first bit of resultant to the beginning of the new bit.

# Booth's Algorithm



## Example

A	Q	$Q_{-1}$	M		
0000	0011	0	0111	Initial values	
1001	0011	0	0111	$A \leftarrow A - M$	First cycle
1100	1001	1	0111	Shift	
1110	0100	1	0111	Shift	Second cycle
0101	0100	1	0111	$A \leftarrow A + M$	Third cycle
0010	1010	0	0111	Shift	
0001	0101	0	0111	Shift	Fourth cycle



# Booth's Algorithm



Explore | Expand | Enrich

## Program

### Booth1.java

#### Sample IO

##### Input

Enter two integer numbers

7 -7

##### Output

A : 0111 0000 0

S : 1001 0000 0

P : 0000 1001 0

P : 1100 1100 1

P : 0001 1110 0

P : 0000 1111 0

P : 1100 1111 1

Result :  $7 * -7 = -49$





## Euclid's Algorithm



### Introduction

Euclidean algorithm or Euclid's algorithm, is an efficient method for computing the greatest common divisor (GCD) of two integers (numbers), the largest number that divides them both without a remainder.

First the two numbers are subjected to prime factorizations, and the common factors of the two prime factorizations are multiplied to get the GCD

$$\begin{aligned}36 &= 2 \times 2 \times 3 \times 3 \\60 &= 2 \times 2 \times 3 \times 5\end{aligned}$$

$$\begin{aligned}\text{GCD} &= \text{Multiplication of common factors} \\&= 2 \times 2 \times 3 \\&= 12\end{aligned}$$




## Euclid's Algorithm



Explore | Expand | Enrich

### Idea

The algorithm is based on the below facts.

If we subtract a smaller number from a larger (we reduce a larger number), GCD doesn't change. So if we keep subtracting repeatedly the larger of two, we end up with GCD.

Now instead of subtraction, if we divide the smaller number, the algorithm stops when we find remainder 0.



# Binary Palindrome



Explore | Expand | Enrich

## Program

`euclid1.java`

### Sample IO

Input : 10 15

Output : 5

Input : 35 10

Output : 5

Input : 31 2

Output : 1





## Euclid's Algorithm



Explore | Expand | Enrich

### Idea

Extended Euclidean algorithm also finds integer coefficients x and y such that:

$$ax + by = \gcd(a, b)$$

- The extended Euclidean algorithm updates results of  $\gcd(a, b)$  using the results calculated by recursive call  $\gcd(b \% a, a)$ .

Let values of x and y calculated by the recursive call be  $x_1$  and  $y_1$ .

x and y are updated using the below expressions.

$$\begin{aligned}x &= y_1 - \lfloor b/a \rfloor * x_1 \\y &= x_1\end{aligned}$$



## Program

`euclid2.java`

### Sample IO

Same as above, but you can also print the integer coefficients

The extended Euclidean algorithm is particularly useful when  $a$  and  $b$  are coprime.

- With that provision,  $x$  is the modular multiplicative inverse of  $a$  modulo  $b$ , and  $y$  is the modular multiplicative inverse of  $b$  modulo  $a$ .

Similarly, the polynomial extended Euclidean algorithm allows one to compute the multiplicative inverse in algebraic field extensions and, in particular in finite fields of non prime order.

It follows that both extended Euclidean algorithms are widely used in cryptography.

In particular, the computation of the modular multiplicative inverse is an essential step in the derivation of key-pairs in the RSA public-key encryption method.



# Karatsuba Algorithm



Explore | Expand | Enrich

## Introduction

The Karatsuba algorithm is a fast multiplication algorithm. It was discovered by Anatoly Karatsuba in 1960 and published in 1962.

It is a fast multiplication algorithm that uses a divide and conquer approach to multiply two numbers.

The naive algorithm for multiplying 2 numbers has a running time of  $O(n^2)$ , whereas the Karatsuba algorithm has a runtime of

$$\Theta(n^{\log_2 3}) \approx \Theta(n^{1.585})$$

Being able to multiply numbers quickly is very important. Computer scientists often consider multiplication to be a constant time  $\sim O(1)$  operation which is reasonable for small numbers.

Whereas for larger numbers, the actual running times need to be factored in, which is  $O(n^2)$





## Karatsuba Algorithm



Explore | Expand | Enrich

### Naive Algorithm

$$\begin{array}{r} 45 \\ \times 32 \\ \hline \end{array} \quad \begin{array}{r} 45 \\ \times 32 \\ \hline \end{array} \quad \begin{array}{r} 45 \\ \times 32 \\ \hline \end{array} \quad \begin{array}{r} 45 \\ \times 32 \\ \hline \end{array}$$

Four separate multiplication problems are shown, each with a yellow arrow pointing from the tens digit '4' in the first factor to the tens digit '3' in the second factor.

The key idea is to reduce the four sub-problems in multiplication to three unique problems.

Thus, on calculating the three unique sub-problems, the original four sub-problems are solved using addition or subtraction operation.





# Karatsuba Algorithm



Explore | Expand | Enrich

## Explanation

Karatsuba stated that if we have to multiply two n-digit numbers x and y

The two numbers x and y has to be decomposed into 4 numbers x1, x2 and y1,y2

$$x = x1 * B^m + x2$$

$$y = y1 * B^m + y2$$

Eg: 45 \* 32

45 can be represented as  $4*10 + 5$

Where  $x1 = 4$ ,  $x2=5$  with  $m=1$

Note: B is the base (base-10 in this case)





# Karatsuba Algorithm



Explore | Expand | Enrich

## Explanation

The product of x and y after the decomposition can be represented by:-

$$xy = (x_1 * B^m + x_2)(y_1 * B^m + y_2)$$

$$\Rightarrow xy = x_1 * y_1 * B^{2m} + x_1 * y_2 * B^m + x_2 * y_1 * B^m + x_2 * y_2$$

Observe that there are 4 sub-problems:

- $X_1 * Y_1$
- $X_1 * Y_2$
- $X_2 * Y_1$
- $X_2 * Y_2$

We are going to reduce the 4 sub-problems to 3 sub-problems





# Karatsuba Algorithm



## Procedure

Consider the following

- $a = X1 * Y1$
- $b = X1 * Y2 + X2 * Y1$
- $c = X2 * Y2$

Then the following equation

$$xy = x1 * y1 * B^{(2m)} + x1 * y2 * B^m + x2 * y1 * B^m + x2 * y2$$

Becomes  $xy = a * B^{(2m)} + b * B^m + c$

Karatsuba came up with a brilliant idea to calculate b with the following formula

$$b = (x1 + x2)(y1 + y2) - a - c$$



# Karatsuba Algorithm



## Example

Consider the following multiplication:  $47 \times 78$

$$\begin{aligned}x &= 47 \\x &= 4 * 10 + 7\end{aligned}$$

$$\begin{aligned}y &= 78 \\y &= 7 * 10 + 8\end{aligned}$$

$$\begin{aligned}x_1 &= 4 \\x_2 &= 7\end{aligned}$$

$$\begin{aligned}y_1 &= 7 \\y_2 &= 8\end{aligned}$$

The Three subproblems:

$$a = x_1 * y_1 = 4 * 7 = 28$$

$$c = x_2 * y_2 = 7 * 8 = 56$$

$$b = (x_1 + x_2)(y_1 + y_2) - a - c = 11 * 15 - 28 - 56$$

Note:  $11 * 15$  can in turn be multiplied using Karatsuba Algorithm



# Karatsuba Algorithm



## Time Complexity

Assuming that we replace two of the multiplications with only one makes the program faster.

Karatsuba improves the multiplication process by replacing the initial complexity from quadratic to  $\Theta(n^{\log_2 3}) \approx \Theta(n^{1.585})$

Where n is the number of digits of the numbers multiplying.

- The Time Complexity of the algorithm can be represented as follows

$$T(n) = 3 * T(n/2) + O(n)$$



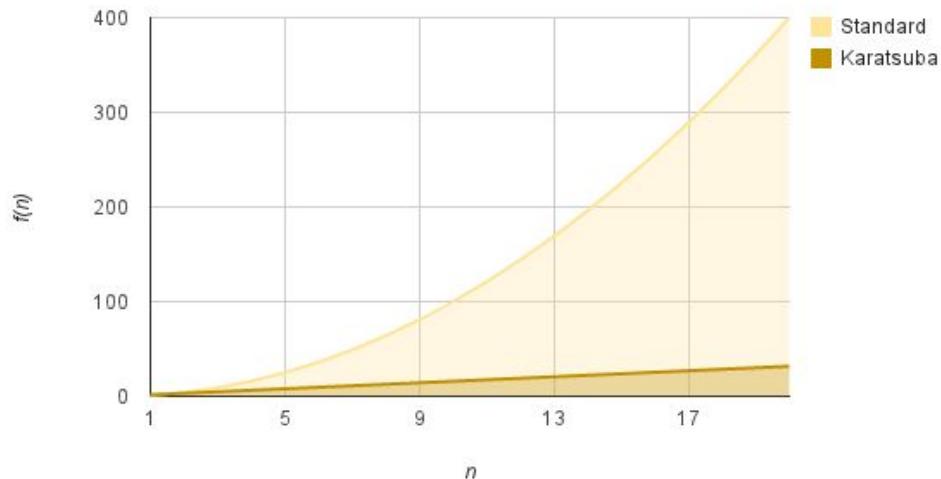


# Karatsuba Algorithm



## Time Complexity

$$T(n) = 3 * T(n/2) + O(n)$$





# Karatsuba Algorithm



Explore | Expand | Enrich

## Implementation

We shall look at implementing Karatsuba algorithm for four digits.

## Algorithm

1. Compute starting set ( $a*c$ )
2. Compute set after starting set may it be ending set ( $b*d$ )
3. Compute starting set with ending sets
4. Subtract values of Step 3 from Step 2 from Step 1
5. Add all the values with the following modifications:-
  1. Pad up 4 zeros to the number obtained from Step 1
  2. Step 2 value unchanged
  3. Pad up two zeros to value obtained from Step 4.





# Karatsuba Algorithm



Explore | Expand | Enrich

## Program

karatsuba1.java

### Sample IO

Karatsuba Multiplication Algorithm Test

Enter two integer numbers:

24061994 28563

Product : 687282734622





## Longest Sequence of 1s after flip



Explore | Expand | Enrich

### Introduction

In this program you are assumed to have a sequence of bits 0s and 1s

The sequence can be in the form of a binary string (“110011”) or as a binary array([1,0,1,1])

- Given that you can flip any one bits in the sequence, you have to do it such that you get the longest sequence of 1s.





## Longest Sequence of 1s after flip



Explore | Expand | Enrich

### Example

Consider the sequence 11011101111

To get the longest sequence of bits, out of all the possible flips, flipping the highlighted digit in the sequence 110111**0**1111.

- This will yield the following sequence: 1101111111

The output of the program would be 8 since there are 8 consecutive bits in the sequence

Additionally you may be given with a decimal which you would have to convert to a bit string to apply the algorithm too





# Longest Sequence of 1s after flip



## Solution

Given a number, provide the longest sequence of 1s after a flip of its bit sequence.

A simple solution is to store the binary representation of a given number in a binary array.

An efficient solution is to walk through the bits in the binary representation of the given number.

We keep track of the current 1's sequence length and the previous 1's sequence length. When we see a zero, update the previous Length:

- If the next bit is a 1, the previous Length should be set to the current Length.
- If the next bit is a 0, then we can't merge these sequences together. So, set the previous Length to 0.

We update max length by comparing the following two:

- The current value of max-length
- Current-Length + Previous-Length .

Result = return max-length+1 (// add 1 for flip bit count )





# Longest Sequence of 1s after flip



Explore | Expand | Enrich

## Program

bitflip1.java

### Sample IO

Enter a number: 13

Answer is: 4

Enter a number: 1775

Answer is: 8

Enter a number: 15

Answer is: 5

Time Complexity: O(n)

Where n is the number of bits





# Longest Sequence of 1s after flip



Explore | Expand | Enrich

## Program

bitflip2.java, bitflip3.java

Given a binary array `nums` and an integer `k`, return the maximum number of consecutive 1's in the array if you can flip at most `k` 0's.

### Sample IO

Input: `nums = [1,1,1,0,0,0,1,1,1,0]`, `k = 2`

Output: 6

Explanation: `[1,1,1,0,0,1,1,1,1,1]`

Bolded numbers were flipped from 0 to 1. The longest subarray is underlined.

Input: `nums = [0,0,1,1,0,0,1,1,1,0,1,1,0,0,0,1,1,1,1]`, `k = 3`

Output: 10

Explanation: `[0,0,1,1,1,1,1,1,1,1,0,0,0,1,1,1,1]`

Bolded numbers were flipped from 0 to 1. The longest subarray is underlined.

Time Complexity:  $O(n)$

Where  $n$  is the number of bits





# Swap two nibbles in a byte



## Introduction

A nibble consists of four bits. It is a four-bit aggregation, or half an octet.

There are two nibbles in a byte.

For example, 64 is to be represented as 01000000 in a byte (or 8 bits). The two nibbles are (0100) and (0000).

## Nibble Swap

Suppose you have a number say 100.

100 is to be represented as 01100100 in a byte (or 8 bits)

The two nibbles are 0110 and 0100.

After swapping the nibbles, we get 01000110 which is 70 in decimal.

To swap the nibbles, We will use bitwise operators &, |, << and >> to swap the nibbles in a byte.



# Swap two nibbles in a byte



## Procedure

The snippet of code for nibble swap is as follows:

$$(x \& 0x0F) << 4 | (x \& 0xF0) >> 4$$

## Explanation

As we know binary of 100 is 01100100. To swap the nibble we split the operation in two parts, in first part we get last 4 bits and in second part we get first 4 bit of a byte.

**First operation:** The expression “data & 0x0F” gives us the last 4 bits of data and result would be 00000100.

Using bitwise left shift operator ‘<<’, we shift the last four bits to the left 4 times and make the new last four bits as 0. The result after the shift is 01000000.

**Second operation:** The expression “data & 0xF0” gives us first four bits of data and result would be 01100000.

Using bitwise right shift operator ‘>>’ , we shift the digit to the right 4 times and make the first four bits as 0. The result after the shift is 00000110.

After completing the two operation we use the bitwise OR ‘|’ operation on them. After OR operation you will find that first nibble to the place of last nibble and last nibble to the place of first nibble



## Swap two nibbles in a byte



Explore | Expand | Enrich

### Program

nickname1.java

### Sample IO

Input: 100

Output: 70

Input: 200

Output: 140

Input: 67

Output: 52



## Introduction

The block swap algorithm for array rotation is an efficient algorithm that is used for array rotation.

It can do the work in  $O(n)$  time complexity

- We would be given an array arr of size n, and an integer d

One would have to rotate the array arr by d elements

We have both recursive and iterative versions of the algorithms

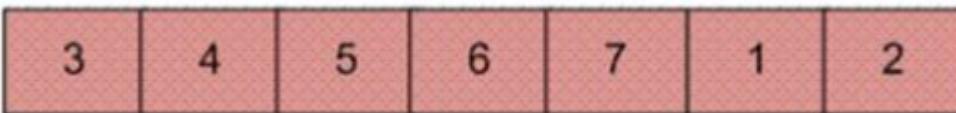


## Block Swap Algorithm



### Demo

Write an algorithm to rotate the following array by 2



## Algorithm

Initialize A = arr[0..d-1] and B = arr[d..n-1]

- 1) Do following until size of A is equal to size of B
  - a) If A is shorter, divide B into BI and Br such that Br is of same length as A. Swap A and Br to change ABIBr into BrBIA. Now A is at its final place, so recur on pieces of B.
  - b) If A is longer, divide A into AI and Ar such that AI is of same length as B Swap AI and B to change AIArB into BArAI. Now B is at its final place, so recur on pieces of A.
- 2) Finally when A and B are of equal size, block swap them.



# Block Swap Algorithm



Explore | Expand | Enrich

## Programs

blockswap1.java

blockswap2.java



## Subarray vs Subsequence

Subsequence: A subsequence is a sequence that can be derived from another sequence by zero or more elements, without changing the order of the remaining elements.

Subarray: A subarray is a contiguous part of array. An array that is inside another array.

A Subsequence cannot be a subarray

For instance, let arr = [1,2,3,4,5]

Examples for Subarrays: [1], [1,2], [2,3,4] etc

Examples for Subsequences: [1], [1,3], [1,2,3], [2,3,4]



# Maximum Product Subarray



Explore | Expand | Enrich

## Introduction

You would be given a array of integers and you have to find a subarray which has the largest product

The array can contain both positive and negative integers

As discussed earlier -- A subarray is a contiguous subsequence of the array.

The solution must should be preferably having  $O(n)$  time complexity and constant space complexity





# Maximum Product Subarray



Explore | Expand | Enrich

## Examples

### Example-1

Input: nums = [2,3,-2,4]

Output: 6

Explanation: [2,3] has the largest product 6.

### Example-2

Input: nums = [-2,0,-1]

Output: 0

Explanation: The result cannot be 2, because [-2,-1] is not a subarray.

### Example-3

Input: nums = [6, -3, -10, 0, 2]

Output: 180

Explanation: The subarray is [6, -3, -10] with largest product as 180





# Maximum Product Subarray



Explore | Expand | Enrich

## Program

mps1.java

This solution is a naive solution. The idea is to traverse over every contiguous subarrays, find the product of each of these subarrays and return the maximum product from these results.

### Constraints:

Time Complexity:  $O(N^2)$

Auxiliary Space:  $O(1)$





# Maximum Product Subarray



Explore | Expand | Enrich

## Program

mps2.java

This solution is an improvement to the previous naive solution.  
It achieves the goal of linear time complexity constraint

### Constraints:

Time Complexity:  $O(N)$

Auxiliary Space:  $O(1)$



# Maximum sum of hourglass in matrix



Explore | Expand | Enrich

## Introduction

In this problem, we are given a 2D array (matrix) of any order.

Our task is to create a program that finds the maximum sum of the hourglass.

Out of all the possible hourglass sums of a given  $m \times n$  ordered matrix, we have to print the maximum of all the sums

In our problem, Hourglass is a 7 element shape made in the matrix in the following form:

```
X X X  
  X  
X X X
```



# Maximum sum of hourglass in matrix

## Example

Consider the matrix:-

```
2 4 0 0  
0 1 1 0  
4 2 1 0  
0 3 0 1
```



So, an hourglass can be created using the following indexes:-

matrix[i][j]	matrix[i][j+1]	matrix[i][j+2]
matrix[i+2][j]	matrix[i+1][j+1]	matrix[i+2][j+1]
	matrix[i+2][j+1]	matrix[i+2][j+2]

The hourglasses are :

```
2 4 0 0 1 1  
1 1 2  
4 2 1 0 3 0
```

```
4 0 0 1 1 0  
1 1  
2 1 0 3 0 1
```

We will find the sum of all these elements of the array from [0][0] to [R2][C-2] starting points. And then find the maxSum for all these hourglasses created from array elements.



# Maximum sum of hourglass in matrix



Explore | Expand | Enrich

## Program

hourglass1.java

The first line contains m and n which are the number of rows and columns of the matrix.

The next m lines contain n space seperated integers which are the elements of the matrix

### Test Cases

#### Input

```
5 5  
1 2 4 5 6  
7 5 2 3 6  
0 0 0 0 0  
7 5 1 3 5  
0 0 0 0 0
```

#### Output

```
27
```

### Explanation

All the possible hourglasses are given. The bold hourglass is the one with the maximum sum.

1	2	4	2	4	5	4	5	6
			5		2		3	
			0	0	0	0	0	0

<b>7</b>	<b>5</b>	<b>2</b>	<b>5</b>	<b>2</b>	<b>3</b>	<b>2</b>	<b>3</b>	<b>6</b>
			0		0		0	
			<b>7</b>	<b>5</b>	<b>1</b>	<b>5</b>	<b>1</b>	<b>3</b>

0	0	0	0	0	0	0	0	0
			5		1		3	
			0	0	0	0	0	0



## Maximum equilibrium sum in an array



Explore | Expand | Enrich

### Introduction

You have been given an array arr[]

Your task is to find maximum value of prefix sum which is also suffix sum for index i in arr[].

- Example:

arr[] = {1, 2, 3, 5, 3, 2, 1}, then output is 11 as:-

- Prefix sum =  $\text{arr}[0..3] = 1 + 2 + 3 + 5 = 11$
- Suffix sum =  $\text{arr}[3..6] = 5 + 3 + 2 + 1 = 11$





# Maximum equilibrium sum in an array



Explore | Expand | Enrich

## Program: Simple Approach

maxeqsum1.java

Check one by one the given condition (prefix sum equal to suffix sum) for every element and returns the element that satisfies the given condition with maximum value.

Time Complexity:  $O(n^2)$

Auxiliary Space Complexity:  $O(n)$

Sample IO #1

Input

$\text{arr[]} = \{-1, 2, 3, 0, 3, 2, -1\}$

Output

4

Explanation

Prefix sum of  $\text{arr}[0..3] = \text{Suffix sum of } \text{arr}[3..6]$

Sample IO #2

Input

$\text{arr[]} = \{-2, 5, 3, 1, 2, 6, -4, 2\}$

Output

7

Explanation

Prefix sum of  $\text{arr}[0..3] = \text{Suffix sum of } \text{arr}[3..7]$



### Better Approach: Algorithm

- Traverse the array and store prefix sum for each index in array `presum[]`, in which `presum[i]` stores sum of subarray  $\text{arr}[0..i]$
- Traverse array again and store suffix sum in another array `suffsum[]`, in which `suffsum[i]` stores sum of subarray  $\text{arr}[i..n-1]$
- For each index check if `presum[i]` is equal to `suffsum[i]` and if they are equal then compare their value with overall maximum so far



# Maximum equilibrium sum in an array



Explore | Expand | Enrich

## Program: Better Approach

maxeqsum2.java

Based on the algorithm introduced earlier

Time Complexity: O(n)

Auxiliary Space Complexity: O(n)

- Sample IO #1

Input

arr[] = {-1, 2, 3, 0, 3, 2, -1}

Output

4

Explanation

Prefix sum of arr[0..3] = Suffix sum of arr[3..6]

- Sample IO #2

Input

arr[] ={-2, 5, 3, 1, 2, 6, -4, 2}

Output

7

Explanation

Prefix sum of arr[0..3] = Suffix sum of arr[3..7]





# Maximum equilibrium sum in an array



## Program: Even Better Approach

maxeqsum3.java

We can avoid the use of extra space by first computing the total sum, then using it to find the current prefix and suffix sums.

Time Complexity: O(n)

Auxiliary Space Complexity: O(n)

Sample IO #1

Input

$\text{arr[]} = \{-1, 2, 3, 0, 3, 2, -1\}$

Output

4

Explanation

Prefix sum of  $\text{arr}[0..3] = \text{Suffix sum of } \text{arr}[3..6]$

Sample IO #2

Input

$\text{arr[]} = \{-2, 5, 3, 1, 2, 6, -4, 2\}$

Output

7

Explanation

Prefix sum of  $\text{arr}[0..3] = \text{Suffix sum of } \text{arr}[3..7]$



### Introduction

A leader in an array is an element which is greater than all the elements on its right side in the array.

There are two methods for finding leaders in an array.

- Two types of elements are leaders by default.
  - Last element of an array is a leader by default because there is no element on the right side of it and so its right element is NULL.
  - Greatest element of an array is also a leader by default.





## Leaders in an array



Explore | Expand | Enrich

### Example

Consider an array arr = {2,5,8,7,3,6}. The elements 6,7,8 are the leaders

- 6 - It is a leader by default because it is the last element of the array.
- 7 - It is a leader because it is greater than the elements on its right, i.e., 3,6.
- 8 - It is also a leader because it is greater than all the elements on its right, i.e., 7,3,6.





### Program: Easy Method

leader1.java

Use two loops. The outer loop runs from 0 to size – 1 and one by one picks all elements from left to right.

- The inner loop compares the picked element to all the elements to its right side. If the picked element is greater than all the elements to its right side, then the picked element is the leader.

Time Complexity:  $O(n^2)$



# Leaders in an array



## Program: Easy Method

leader1.java

### Test Case #1

#### Input

n = 6

A[] = {16,17,4,3,5,2}

#### Output

17 5 2

### Test Case #2

#### Input

n = 5

A[] = {1,2,3,4,0}

#### Output

4 0

### Explanation of TC#1

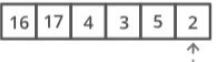
The first leader is 17 as it is greater than all the elements to its right. Similarly, the next leader is 5. The right most element is always a leader so it is also included.



# Leaders in an array

## A Better Method

Initially:



max\_from\_right = 2

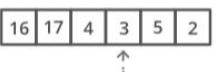
Rightmost element is always leader. So, print 2

Step 1:



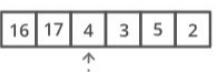
Here,  $a[i] > \text{max\_from\_right}$ . make  
 $\text{max from right} = 5$  and print 5

Step 2:



Here,  $a[i] < \text{max\_from\_right}$ . do nothing

Step 3:



Here,  $a[i] < \text{max\_from\_right}$ . do nothing

Step 4:



Here,  $a[i] > \text{max\_from\_right}$ . make  
 $\text{max from right} = 17$  and print 17

Step 5:



Here,  $a[i] < \text{max\_from\_right}$ . So, do nothing

Explore | Expand | Enrich



## Leaders in an array



Explore | Expand | Enrich

### Program: Better Method

leader2.java

Scan all the elements from right to left in an array and keep track of maximum till now. When maximum changes its value, print it.

Time Complexity

$O(n)$





# Majority element



Explore | Expand | Enrich

## Introduction

You are given an array of integers. The task is to print the majority element.

A majority element in an array  $\text{arr}[]$  of size  $n$  is an element that appears more than  $n/2$  times.

Thus there can only be utmost only one majority element.

Example:

$\text{arr}[] = \{3, 3, 4, 2, 4, 4, 2, 4, 4\}$

Majority Element: 4

$\text{arr}[] = \{3, 3, 4, 2, 4, 4, 2, 4\}$

Majority Element: null





## Majority element



Explore | Expand | Enrich

### Trivial Solution

The basic solution is to have two loops and keep track of the maximum count for all different elements.

If maximum count becomes greater than  $n/2$ , then break the loops and return the element having maximum count.

- If the maximum count doesn't become more than  $n/2$  then the majority element doesn't exist.



## Trivial Solution: Algorithm

- Create a variable to store the max count,  $\text{count} = 0$
- Traverse through the array from start to end.
- For every element in the array run another loop to find the count of similar elements in the given array.
- If the count is greater than the max count update the max count and store the index in another variable.
- If the maximum count is greater than the half the size of the array, print the element. Else print there is no majority element.



# Majority element



Explore | Expand | Enrich

## Program: Trivial Solution

majele1.java

Time Complexity:  $O(n^2)$

Auxiliary Space:  $O(1)$



## BST Solution

Insert elements in Binary Search Tree (BST) one by one and if an element is already present then increment the count of the node. At any stage, if the count of a node becomes more than  $n/2$  then return.

## Algorithm

- Create a binary search tree, if same element is entered in the binary search tree the frequency of the node is increased.
- traverse the array and insert the element in the binary search tree.
- If the maximum frequency of any node is greater than the half the size of the array, then perform a inorder traversal and find the node with frequency greater than half
- Else print No majority Element.



## Majority element



Explore | Expand | Enrich

Program: BST Solution

majele2.java

Time Complexity:  $O(n^2)$  in case of BST or  $O(n \log n)$  in case of balanced BT

Auxiliary Space:  $O(n)$



## Linear Solution

In Hashmap(key-value pair), at value, maintain a count for each element(key) and whenever the count is greater than half of the array length, return that key(majority element).

## Algorithm

- Create a hashmap to store a key-value pair, i.e. element-frequency pair.
- Traverse the array from start to end.
- For every element in the array, insert the element in the hashmap if the element does not exist as key, else fetch the value of the key ( `array[i]` ), and increase the value by 1
- If the count is greater than half then print the majority element and break.
- If no majority element is found print “No Majority element”



# Majority element



Explore | Expand | Enrich

## Program: Linear Solution

majele3.java

Time Complexity:  $O(n)$

Auxiliary Space:  $O(n)$





# Majority element



Explore | Expand | Enrich

## Boyer-Moore Solution: Two step process

- The first step gives the element that maybe the majority element in the array. If there is a majority element in an array, then this step will definitely return majority element, otherwise, it will return candidate for majority element.
  - Check if the element obtained from the above step is majority element. This step is necessary as there might be no majority element.
- Algorithm

Initialize an element  $m$  and a counter  $i = 0$

for each element  $x$  of the input sequence:

```
if  $i = 0$ , then  
    assign  $m = x$  and  $i = 1$   
else  
    if  $m = x$ , then assign  $i = i + 1$   
    else  
        assign  $i = i - 1$   
return  $m$ 
```





# Majority element



Explore | Expand | Enrich

Program: Boyer-Moore Solution

majele4.java

Time Complexity:  $O(n)$

Auxiliary Space:  $O(n)$



### Introduction

Imagine you have been given a string  $s$  of alphabets

The task is to rearrange the characters of the given string to form a lexicographically first palindromic string

#### Lexicographically First/Lexicographically Minimum

String  $x$  is lexicographically less than string  $y$

- If either  $x$  is a prefix of  $y$  (and  $x \neq y$ ) or
- There exists such  $i$  ( $1 \leq i \leq \min(|x|, |y|)$ ), that  $x_i < y_i$ , and for any  $j$  ( $1 \leq j < i$ )  $x_j = y_j$

The lexicographic comparison of strings is implemented by operator `<` in modern programming languages.





## Lexicographically first palindromic string



Explore | Expand | Enrich

### Example

Our output string must be lexicographically minimum or lexicographically first.

#### Sample Input

str = “aabcc”

#### Sample Output

acbca

Print -1 if there are no such lexicographically minimum substring.





### Simple Approach

1. Sort the string characters in alphabetical(ascending) order.
2. One by one find lexicographically next permutation of the given string.
3. The first permutation which is palindrome is the answer.



### Improved Approach

We know that, In a palindrome of odd characters (Eg. “actac”, there will be only one character with frequency = 1, all others will have a frequency of 2)

In a Palindrome of even characters (“aacc”, all characters will have a frequency of 2)

Hence, we shall modify the algorithm as shown below

1. If length of string is even, then the frequency of each character in the string must be even.
2. If the length is odd then there should be one character whose frequency is odd and all other chars must have even frequency and at-least one occurrence of the odd character must be present in the middle of the string.

### Algorithm

1. Store frequency of each character in the given string
2. Check whether a palindromic string can be formed or not using the properties of palindromic string mentioned above.
3. If palindromic string cannot be formed, return “No Palindromic String”.
4. Else we create three strings and then return `front_str + odd_str + rear_str`.

Note that:-

- `odd_str` : It is empty if there is no character with odd frequency. Else it contains all occurrences of odd character.
- `front_str` : Contains half occurrences of all even occurring characters of string in increasing order.
- `rear_str` Contains half occurrences of all even occurring characters of string in reverse order of `front_str`.



## Lexicographically first palindromic string



Explore | Expand | Enrich

### Program

`lexfirstpal.java`

Time Complexity:  $O(n)$

Auxillary Space Complexity: constant



## Introduction

Natural sorting is the sorting of text that does more than rely on the order of individual characters codes to make the finding of individual strings easier for a human reader.

## Example

In alphabetical sorting, "z11" would be sorted before "z2" because the "1" in the first string is sorted as smaller than "2", while in natural sorting "z2" is sorted before "z11" because "2" is treated as smaller than "11".





## Natural Sort Order



Explore | Expand | Enrich

### Rules

There is no "one true way" to do this, but for the purpose of this task 'natural' orderings might include:

1. Ignore leading, trailing and multiple adjacent spaces
2. Make all whitespace characters equivalent.
3. Sorting without regard to case.
4. Sorting numeric portions of strings in numeric order.

That is split the string into fields on numeric boundaries, then sort on each field, with the rightmost fields being the most significant, and numeric fields of integers treated as numbers.

foo9.txt before foo10.txt

As well as ... x9y99 before x9y100, before x10y0

... (for any number of groups of integers in a string).

5. Title sorts: without regard to a leading, very common, word such as 'The' in "The thirty-nine steps".
6. Sort letters without regard to accents.
7. Sort ligatures as separate letters.
8. Replacements:

Sort German eszett or scharfes S (ß) as ss

Sort f, LATIN SMALL LETTER LONG S as s

Sort ȝ, LATIN SMALL LETTER EZH as s

...



### Rules - Simplified

#### Alphabetical sorting:

z11

z2

#### Natural sorting:

z2

z11

During the 1996 MacHack conference, the Natural Order Mac OS System Extension was conceived and implemented overnight on-site as an entry for the Best Hack contest.

Dave Koelle wrote the Alphanum Algorithm in 1997 and Martin Pool published Natural Order String Comparison in 2000.





## Natural Sort Order



Explore | Expand | Enrich

### Program

naturalorder1.java



### Introduction

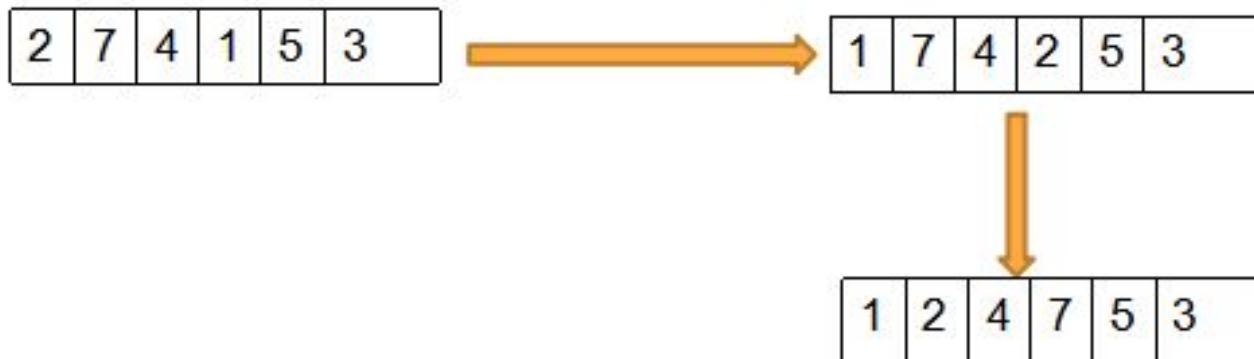
The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning.

- The algorithm maintains two sub arrays in a given array.
  - 1) The sub array which is already sorted.
  - 2) Remaining sub array which is unsorted.

## Selection Sort

Example:

$\text{arr[]} = \{2, 7, 4, 1, 5, 3\}$



### Details

Time Complexity:  $T(n) = O(n^2)$

Auxiliary Space:  $O(1)$

Algorithmic Paradigm: Divide and Conquer

- Sorting In Place: Yes

Stable: No





## Merge Sort



Explore | Expand | Enrich

### Introduction

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 halves. The `merge(arr, l, m, r)` is key process that assumes that `arr[l..m]` and `arr[m+1..r]` are sorted and merges the two sorted sub-arrays into one





## Merge Sort



Algorithm: MergeSort(arr[], l, r)

If  $r > l$

1. Find the middle point to divide the array into two halves:

$$\text{middle } m = (l+r)/2$$

2. Call mergeSort for first half:

- Call mergeSort(arr, l, m)

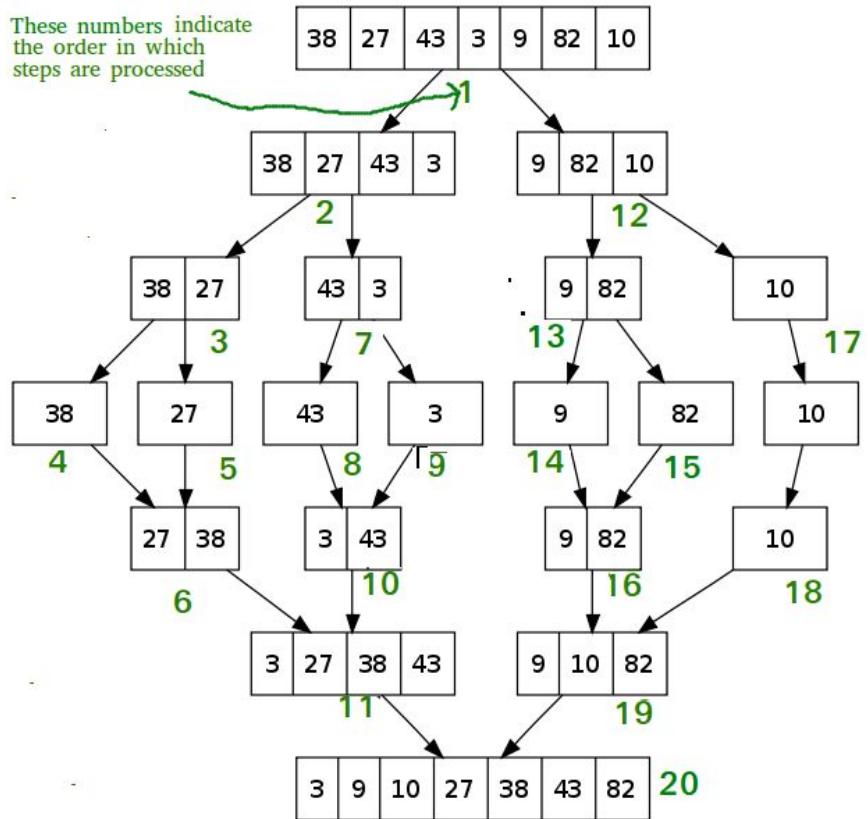
3. Call mergeSort for second half:

- Call mergeSort(arr, m+1, r)

4. Merge the two halves sorted in step 2 and 3:

- Call merge(arr, l, m, r)

# Merge Sort



## Details

Time Complexity:  $T(n) = 2T(n/2) + \theta(n)$

Auxiliary Space:  $O(1)$

Algorithmic Paradigm: Divide and Conquer

• Sorting In Place: No in a typical implementation

Stable: Yes



## Introduction

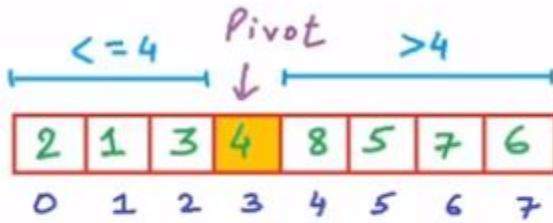
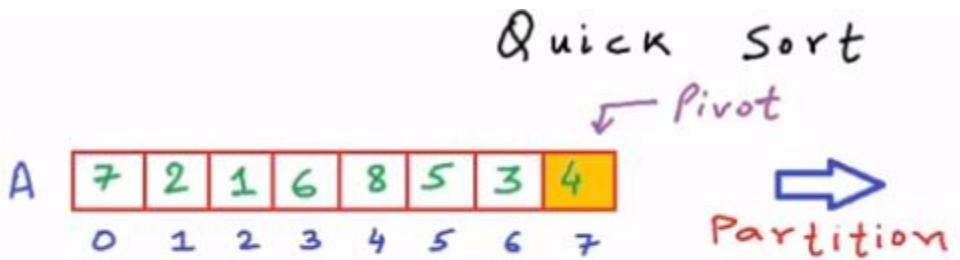
Like Merge Sort, QuickSort is a Divide and Conquer algorithm.

Picks an element as pivot and partitions the given array around the picked pivot.

```
/* low --> Starting index, high --> Ending index */  
quickSort(arr[], low, high)  
{  
    if (low < high)  
    {  
        /* pi is partitioning index, arr[pi] is now  
         * at right place */  
        pi = partition(arr, low, high);  
  
        quickSort(arr, low, pi - 1); // Before pi  
        quickSort(arr, pi + 1, high); // After pi  
    }  
}
```



# Quick Sort



### Details

#### Analysis of QuickSort

Time taken by QuickSort in general can be written as following.

$$T(n) = T(k) + T(n-k-1) + (n)$$

Time Complexity:  $O(n^2)$

• Auxiliary Space:  $O(1)$

Boundary Cases: Insertion sort takes maximum time to sort if elements are sorted in reverse order. And it takes minimum time (Order of n) when elements are already sorted.

Algorithmic Paradigm: Incremental Approach

Sorting In Place: Yes





## Quick Sort



Explore | Expand | Enrich

### Programs

selectionsort.java  
mergesort.java  
quicksort.java



### Introduction

Given a string P consisting of small English letters and a string Q consisting of weight of all characters of English alphabet

Each alphabet will have a weight w where  $0 \leq w \leq 9$

- The task is to find the total numbers of unique substring with sum of weights atmost K.

Weight of a string s can be defined as

$\text{Weight}[S[0]] + \text{Weight}[S[1]] + \text{Weight}[S[2]] + \dots + \text{Weight}[S[\text{Len}-1]]$  where Len is length of string.



# Weighted Substring



## Examples

### Sample IO #1

#### Input

P = "ababab", Q =

"12345678912345678912345678", K = 5

#### Output

7

#### Explanation

The substrings with the sum of weights of individual characters  $\leq 5$  are:

"a", "ab", "b", "bc", "c", "d", "e"

The idea is to use HashSet to solve it

### Sample IO #2

#### Input

P = "acbabcacaa", Q =

"12300045600078900012345000", K = 2

#### Output

3

#### Explanation

The substrings with the sum of weights of individual characters  $\leq 2$  are:

"a", "b", "aa"





# HashSet (For Weighted Substring)



Explore | Expand | Enrich

## Introduction

HashSet class is used to create a collection that uses a hash table for storage.

A Set contains unique elements only.

The underlying data structure for HashSet is Hashtable.

As it implements the Set Interface, duplicate values are not allowed.

Objects that you insert in HashSet are not guaranteed to be inserted in the same order as they are inserted based on their hash code.

## Example

```
1 import java.util.*;
2
3 class EthCode {
4     public static void main(String[] args) {
5         HashSet < String > h = new HashSet < String > ();
6
7         // Adding elements into HashSet usind add()
8         h.add("A");
9         h.add("B");
10        h.add("C");
11        h.add("C"); // adding duplicate elements
12
13        // Displaying the HashSet
14        System.out.println(h);
15        System.out.println(h.contains("A"));
16
17        // Removing items from HashSet using remove()
18        h.remove("B");
19        System.out.println(h);
20
21        // Iterating over hash set items
22        Iterator < String > i = h.iterator();
23        while (i.hasNext())
24            System.out.println(i.next());
25    }
26 }
```

## Output

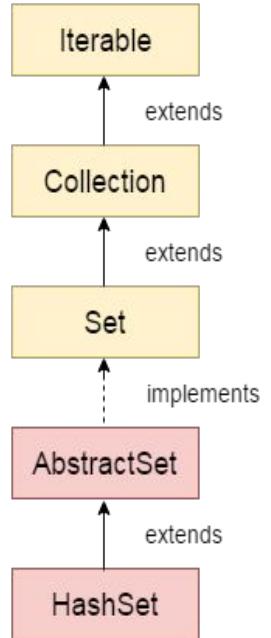
[A, B, C]

true

[A, C]

A

C



### Algorithm

1. Iterate over all the substrings using the nested loops and maintain the sum of the weight of all the characters encountered so far.
2. If the sum of characters is not greater than K, then insert it in a hashmap.
3. Finally, output the size of the hashmap.

# Program

weightedsubstring1.java

Time Complexity: Quadratic a.k.a.  $O(n^*n)$



## Introduction: Variation

This is the variation of the previous program

Given string S which contains only lowercase characters and an integer K

You have to find number of substrings having weight equal to K

- Sample IO #1

Input

str="abcdef", K=5

Output

2

Sample IO #2Input

str="abcdef", K=4

Output

1

```
Weight['a']=1
Weight['b']=2
Weight['c']=3
Weight['d']=4
Weight['e']=5
Weight['f']=6
Weight['g']=7
Weight['h']=8
Weight['i']=9
Weight['j']=10
Weight['k']=11
Weight['l']=12
Weight['m']=13
Weight['n']=14
Weight['o']=15
Weight['p']=16
Weight['q']=17
Weight['r']=18
Weight['s']=19
Weight['t']=20
Weight['u']=21
Weight['v']=22
Weight['w']=23
Weight['x']=24
Weight['y']=25
Weight['z']=26
```

# Program

weightedsubstring2.java

Time Complexity: Quadratic a.k.a.  $O(n^*n)$





## Move hyphen to the beginning



Explore | Expand | Enrich

### Introduction

You are given a string that has set of words and hyphens

Write a program to move all hyphens to front of string

The condition is that you have to do it by traversing the string only once.

Examples:

“hello-world-” becomes “--helloworld”

“I - love--Java” becomes “---I loveJava”



### Approach-1: Using Swap

1. Initialize two indices i and j for traversals from beginning to end
2. If the current index contains hyphen, swap chars in index i with index j
3. Decrement the value by 1 and continue the advancement of j
4. At the end of this method, all hyphens will bubble up to the beginning



## Move hyphen to the beginning



Explore | Expand | Enrich

### Program

movhyptobegin1.java

- Time complexity:  $O(n)$   
Auxiliary Space:  $O(1)$

### Approach-2: Not Using Swap

The idea is to copy all characters except hyphen to the end

Copy the hyphens at the end.





## Move hyphen to the beginning



Explore | Expand | Enrich

### Program

movhyptobegin2.java

- Time complexity:  $O(n)$   
Auxiliary Space:  $O(1)$

### Introduction

Given a string, find the longest substring which is palindrome.

Here, the task is to find the Longest Palindromic Substring given a string

Manacher's algorithm solves this task in Linear Time

### Examples

- if the given string is “abaaba”, the output should be “abaaba”
- if the given string is “abababa”, the output should be “abababa”

### Brute Force

The simple approach is to check each substring whether the substring is a palindrome or not.

To do this first, run three nested loops:-

- The outer two loops pick all substrings one by one by fixing the corner characters
- The inner loop checks whether the picked substring is palindrome or not.



## Manacher's Algorithm



Explore | Expand | Enrich

### Program

manacher1.java

- Time complexity:  $O(n^3)$   
Auxiliary Space:  $O(1)$

## Better Solution

1. The idea is to generate all even length and odd length palindromes and keep track of the longest palindrome seen so far.
2. To generate odd length palindrome, Fix a center and expand in both directions for longer palindromes, i.e. fix  $i$  (index) as the center and two indices as  $i_1 = i+1$  and  $i_2 = i-1$
3. Compare  $i_1$  and  $i_2$  if equal then decrease  $i_2$  and increase  $i_1$  and find the maximum length.
4. Use a similar technique to find the even-length palindrome.
5. Take two indices  $i_1 = i$  and  $i_2 = i-1$  and compare characters at  $i_1$  and  $i_2$  and find the maximum length till all pairs of compared characters are equal and store the maximum length.
6. Print the maximum length.

# Program

manacher2.java

TC: Two nested traversals are needed.

SC: Matrix of size  $n^2$  is needed to store the dp array.

Time complexity:  $O(n^2)$

Auxiliary Space:  $O(1)$



## Manacher's Algorithm



Explore | Expand | Enrich

### Manacher Algorithm to the rescue!

Manacher's Algorithm helps us find the longest palindromic substring in the given string.

It optimizes over the brute force solution by using some insights into how palindromes work



### Notations

Let's focus on odd length palindromes for simplicity

Let  $c$  be the center of the longest length palindrome we have encountered till now.

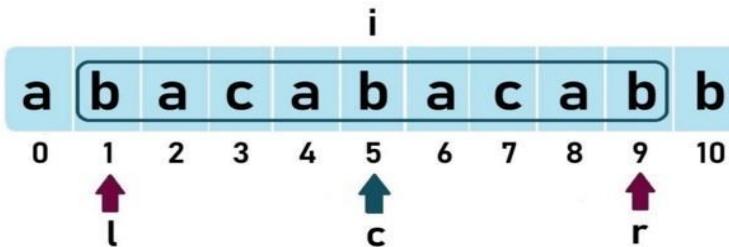
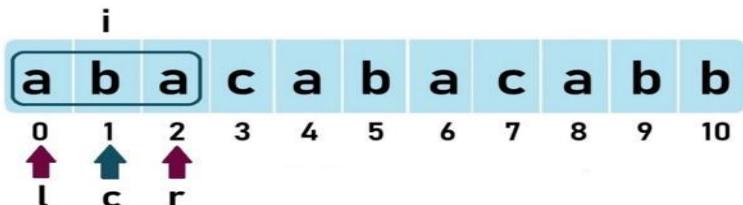
- Let  $l$  and  $r$  be the left and right boundaries of this palindrome, i.e., the left-most character index and the right-most character index respectively.

Now, let's take an example to understand  $c$ ,  $l$ , and  $r$

Algorithm. Take an example  
“abacabacabb”

When going from left to right, when  $i$  is at index 1, the longest palindromic substring is “aba” (length = 3)

The answer for the given string is 9 when the palindrome is centered at index 5;  $c$ ,  $l$ , and  $r$  are as follows:



## Concept: Mirror Index

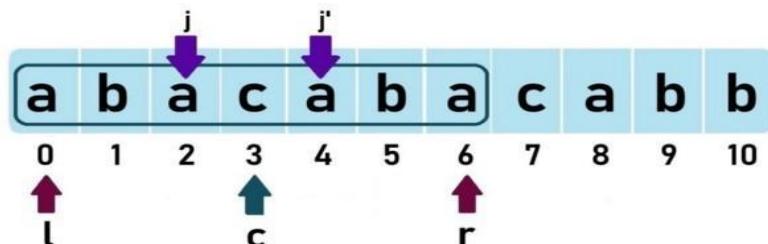
For any palindrome centered at a center  $c$ , the mirror of index  $j$  is  $j'$  such that

- For palindrome “abacaba”, the mirror of  $j$  is  $j'$  and the mirror of  $j'$  is  $j$ .

$$c - j = j' - c$$

So, mirror index of  $j$ :

$$j' = (2 * c) - j$$



Now, come back to the algorithm

### Algorithm Continued

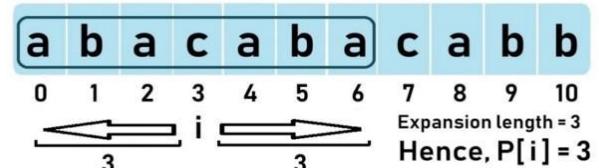
When we move  $i$  from left to right, we try to “expand” the palindrome at each  $i$ .

Expand means, that we'll check whether there exists a palindrome centered at  $i$  and if there exists one, we'll store the “expansion length” to the left or to the right in a new array called  $P[]$  array or (some prefer)  $LPS[]$ .

If the palindrome at  $i$  expands beyond the current right boundary  $r$ , then  $c$  is updated to  $i$  and new  $l, r$  are found and updated.

## Example

Let's take the previously discussed palindrome “abacaba” which is centered at  $i = 3$ .



Therefore, the  $P[]$  array stores the expansion length of the palindrome centered at each index.

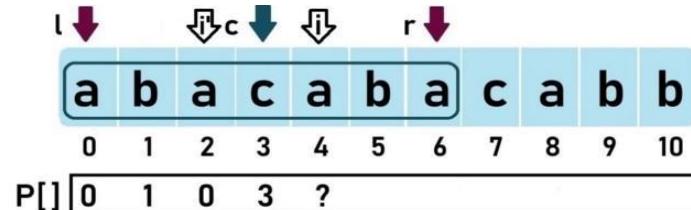
But we don't need to manually go to each index and expand to check the expansion length every time.

This is exactly where Manacher's algorithm optimizes better than brute force, by using some insights on how palindromes work.

## Optimization explained

Remember that  $c$  indicates the center of the current longest odd length palindrome. And  $l$ ,  $r$  denote the palindrome's left and right boundaries

Let's see the  $P[]$  array for the string "abacaba".



When  $i = 4$ , the index is inside the scope of the current longest palindrome, i.e.,  $i < r$ .

So, instead of naively expanding at  $i$ , we want to know the minimum expansion length that is certainly possible at  $i$ , so that we can expand on that minimum  $P[i]$  and see, instead of doing from start. So, we check for mirror  $i'$

### Optimization

As long as the palindrome at index  $i'$  does NOT expand beyond the left boundary ( $l$ ) of the current longest palindrome, we can say that the minimum certainly possible expansion length at  $i$  is  $P[i']$ .

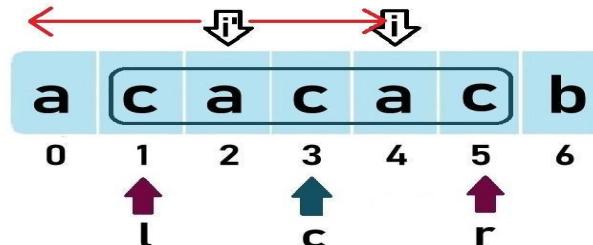
Remember that we are only talking about the minimum possible expansion length, the actual expansion length could be more and, we'll find that out by expanding later on. In this case,  $P[4] = P[2] = 0$ . We try to expand but still,  $P[4]$  remains 0.

Now, if the palindrome at index  $i'$  expands beyond the left boundary ( $l$ ) of the current longest palindrome, we can say that the minimum certainly possible expansion length at  $i$  is  $r-i$ .

## Manacher's Algorithm

### Example

So,  $P[4] = 5 - 4 = 1$



You could ask but why can't the palindrome centered at  $i$  expand after  $r$  in this case? If it did, then it would have already been covered with the current center  $c$  only. But, it didn't. So,  $P[i] = r - i$

If palindrome at  $i$  were to expand beyond  $r$ , then the character at index 6 should have been 'a'. But if that were to happen, the current palindrome centered at  $c$  would NOT have been "cacac" but "acacaca"

```
if(i < r){  
    P[i] = Math.min(r - i, P[mirror]);  
}
```

So, we can sum the two quoted points above as the image at the right:

## Algorithm Continued

Now the only thing left is to expand at  $i$  after  $P[i]$ , so we check characters from index  $(P[i] + 1)$  and keep expanding at  $i$ .

If this palindrome expands beyond  $r$ , update  $c$  to  $i$  and  $r$  to  $(i + P[i])$ .

The  $P[]$  array is now filled and the maximum value in this array gives us the longest palindromic substring in the given string

This works for odd lengthed strings only. we simply append  $N + 1$  special characters (say '#') in between every two characters, just to make sure that our modified string is always of odd length.

Examples:

aba -> #a#b#a#

abba-> #a#b#b#a#



## Manacher's Algorithm



Explore | Expand | Enrich

### Program

manacher3.java

- Time Complexity:  $O(N)$
- Space Complexity:  $O(N)$



## Introduction

In this programming problem, we are given a string str

We have to print the distinct sorted permutations of the string elements that can be formed.

- Condition: The string may contain a character that will arise more than one time

Permutation is rearranging all the elements of a set based on a specific sequence or type. The set may be ordered or may not be ordered.



### Example

#### Input

RST

#### Output

RST, RTS, SRT, STR, TRS, TSR

#### Explanation

From this string total  $3! = 6$  permutations can be formed.

Let's fix R and find permutations from s and t. This will give 2 strings RST, RTS.  
Similarly fixing S will give SRT, STR. And fixing T will give TRS, TSR.

### A few more examples

#### Example-1

##### Input

BAC

##### Output

ABC ACB BAC BCA CAB CBA

#### Example-2

##### Input

AAB

##### Output

AAB ABA BAA

#### Example-3

##### Input

DBCA

##### Output

ABCD ABDC ACBD  
ACDB ADBC ADCB  
BACD BADC BCAD  
BCDA BDAC BDCA  
CABD CADB CBAD  
CBDA CDAB CDBA  
DABC DACB DBAC  
DBCA DCAB DCBA

### For repetitions

Input

AAL

Output

AAL ALA LAA

Explanation

Length of Str: 3

A is repeated 2 times:  $3!/2 = 3$



### Algorithm

1. Sort the string, if the input isn't sorted already
2. Obtain the total number of permutations which can be formed from that string
3. Print the sorted string and then loop for the number of (permutations-1) times as 1st string is already printed
4. Find the next greater string

### Program

sup1.java

- Time Complexity:  $O(P*Q)$ , where P is the size of the string and Q is the number of permutations possible
- Space Complexity:  $O(N)$

### Introduction

The problem is also called Maneuvering a Cave

You are given a  $m \times n$  matrix as input

- The task is to count all the possible paths from top left to bottom right of a  $m \times n$  matrix

### Constraints

You start from  $0 \times 0$  and end at  $m-1 \times n-1$

From each cell you can either move only to right or down.



## Examples

### Input

$m = 2, n = 2$

### Output

2

### Explanation

There are two paths

$(0, 0) \rightarrow (0, 1) \rightarrow (1, 1)$   
 $(0, 0) \rightarrow (1, 0) \rightarrow (1, 1)$

### Input

$m = 2, n = 3$

### Output

3

### Explanation

There are three paths

$(0, 0) \rightarrow (0, 1) \rightarrow (0, 2) \rightarrow (1, 2)$   
 $(0, 0) \rightarrow (0, 1) \rightarrow (1, 1) \rightarrow (1, 2)$   
 $(0, 0) \rightarrow (1, 0) \rightarrow (1, 1) \rightarrow (1, 2)$

# Program

## Solution-1

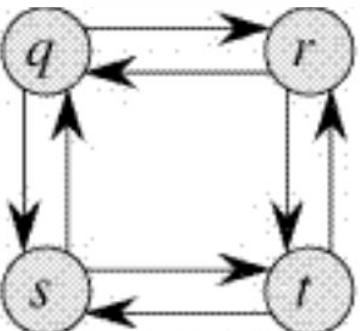
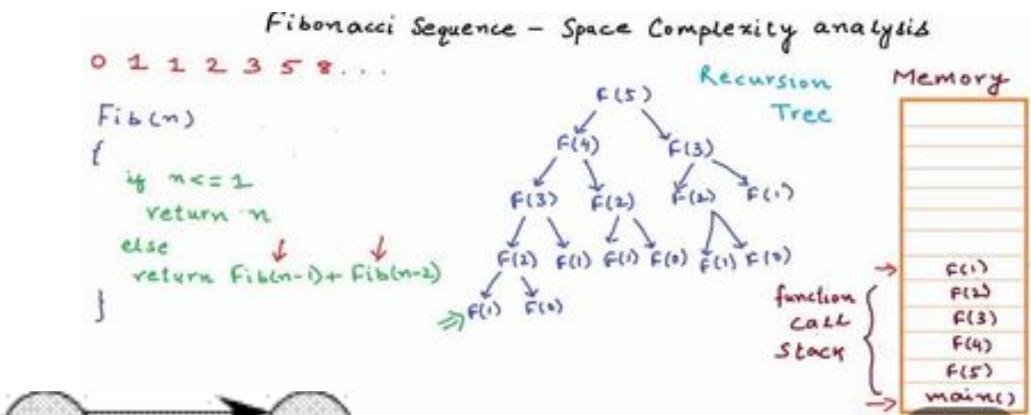
manCave1.java

- Time Complexity: Exponential
- 

## Solution-2: DP

### Dynamic Programming Properties

- 1) Overlapping Subproblems
  - a) Memoized (Top Down)
  - b) Tabulation (Bottom Up)
- 2) Optimal Substructure



- Optimal Substructure
  - An optimal solution to a problem contains within it an optimal solution to subproblems
  - Optimal solution to the entire problem is build in a bottom-up manner from optimal solutions to subproblems
- Overlapping Subproblems
  - If a recursive algorithm revisits the same subproblems over and over  $\Rightarrow$  the problem has overlapping subproblems

# Program

## Solution-2

manCave2.java

- Time Complexity:  $O(mn)$
- 



## Maneuvering Problem



Explore | Expand | Enrich

### Program

#### Solution-3

manCave3.java

This is an optimization over the DP variant of the problem

- We will be using a 1D array here

Time Complexity:  $O(n)$



## Introduction

A combination is a subset of elements from a given set

Example: In a card game, we have to deal 5 cards out of the pack consisting of 52 cards. We have no interest in the order in which the 5 cards were selected. Rather, we only care which cards are present in the hand

$$C(n, k) = \binom{n}{k} = \frac{n!}{k! (n - k)!}$$

Program: Given an array of size n, generate and print all possible combinations of r elements in array. For example, if input array is {1, 2, 3, 4} and r is 2, then output should be {1, 2}, {1, 3}, {1, 4}, {2, 3}, {2, 4} and {3, 4}.

### Technique-1

We create a temporary array 'data' which stores all outputs one by one.

Start with the first index in data, one by one fix elements at this index and recur for remaining indexes.

Example: arr = [1, 2, 3, 4, 5] r = 3

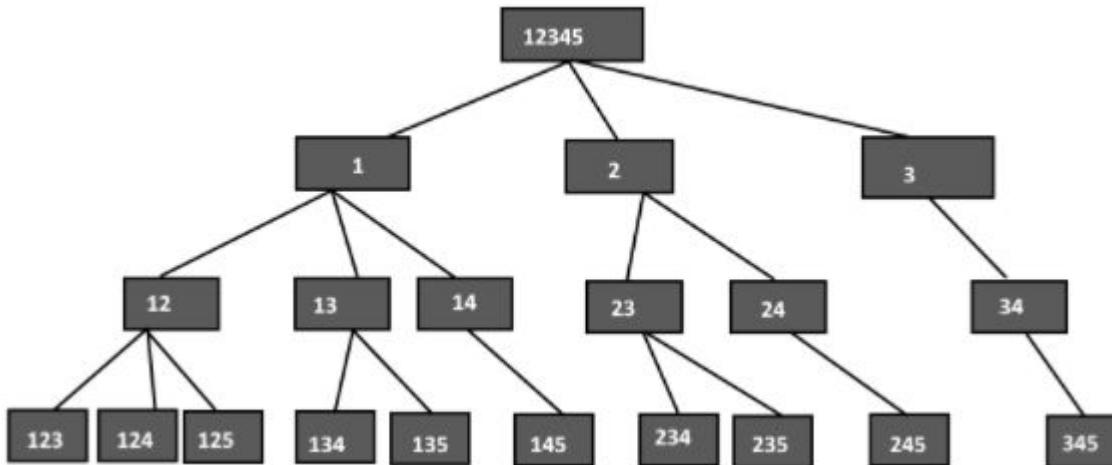
We first fix 1 at index 0 in data then recur for remaining indexes, then we fix 2 at index 0 and recur.

Then we fix 3 and recur for remaining indexes

When number of elements in data becomes equal to r (size of a combination), we print data

# Combinations

## Technique-1



# Program

comb1.java

Time Complexity:  $O(n)$



### Technique-1: Handling Duplicates

If input array is [1 , 2, 1] and r is 2, then the program prints [1, 2] and [2, 1] as two different combinations

We can avoid duplicates by adding following to our code

- 1) Add code to sort the array before calling combinationUtil() in printCombination()
- 2) Add following lines at the end of for loop in combinationUtil()

# Program

comb2.java

Time Complexity:  $O(n)$



## Technique-2

The idea of technique-2 is similar to the subset-sum problem and is based on Pascal's identity

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

- We one by one consider every element of input array, and recur for two cases:
  - 1) The element is included in current combination  
(We put the element in data and increment next available index in data)
  - 2) The element is excluded in current combination (We do not put the element and do not change index)

# Program

comb3.java

Time Complexity:  $O(n)$



### Introduction

This problem goes back to Josephus Flavius, a 1st century Roman historian

- 41 rebels are trapped in a cave, surrounded by enemy troops. They decide to commit suicide: they line up in a circle and systematically kill every other one, going around and around, until only one rebel is left -- who supposedly kills himself (fat chance).

Who is the lone survivor?



### Introduction

This problem goes back to Josephus Flavius, a 1st century Roman historian

- 41 rebels are trapped in a cave, surrounded by enemy troops. They decide to commit suicide: they line up in a circle and systematically kill every other one, going around and around, until only one rebel is left -- who supposedly kills himself (fat chance).

Who is the lone survivor?





## Josephus Problem



Explore | Expand | Enrich

### Task

Given the total number of persons  $n$  and a number  $k$  which indicates that  $k-1$  persons are skipped and  $k$ th person is killed in circle.

The task is to choose the place in the initial circle so that you are the last one remaining and so survive.



### Examples

$n = 5$  and  $k = 2$

The safe position is 3. Firstly, the person at position 2 is killed, then person at position 4 is killed, then person at position 1 is killed. Finally, the person at position 5 is killed. So the person at position 3 survives

$n = 7$  and  $k = 3$

The safe position is 4. The persons at positions 3, 6, 2, 7, 5, 1 are killed in order, and person at position 4 survives



### Thinking

Clearly, we can represent the rebels by a list of numbers  $\{1, 2, \dots, 41\}$  and we can simulate the demise of the corresponding rebel by manipulating this list.

For example, we could just mark the dead rebels by switching item  $i$  to  $-i$  (a standard hack, and a true abuse of negative numbers).

Actually, it will be more interesting to generalize slightly: let us deal with the problem for an arbitrary number  $n$  of rebels.

So, our original list is  $\{1, 2, \dots, n\}$ , and we have to make  $n-1$  deletions in the right place to find the survivor.

### Structure

The problem has following recursive structure

$$\begin{aligned} \text{josephus}(n, k) &= (\text{josephus}(n - 1, k) + k - 1) \% n + 1 \\ \text{josephus}(1, k) &= 1 \end{aligned}$$

- After the first person (kth from beginning) is killed, n-1 persons are left.

So we call  $\text{josephus}(n - 1, k)$  to get the position with n-1 persons.

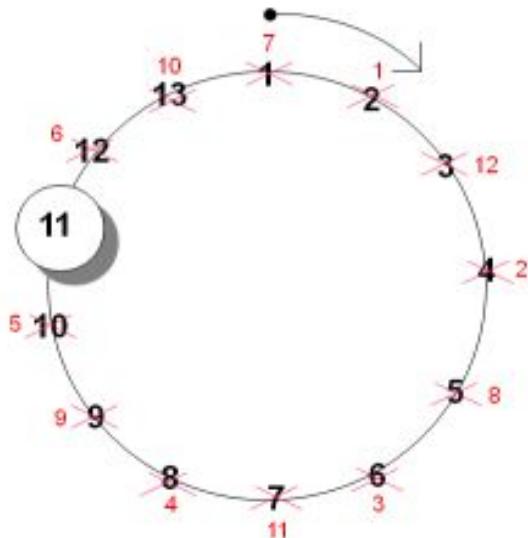
But the position returned by  $\text{josephus}(n - 1, k)$  will consider the position starting from  $k \% n + 1$ .

So, we must make adjustments to the position returned by  $\text{josephus}(n - 1, k)$ .

# Josephus Problem

## Example

In the above image the execution starts from 2 and finally the free person will be 11.





## Josephus Problem



Explore | Expand | Enrich

### Program

josephus1.java

#### IO Format

#### Input

$n = 14; k = 2$

#### Output

13

#### Explanation

The chosen place is 13

#### Time Complexity

$O(n)$



### Another Solution

Let's take an example of there are 5 people in a circle and execution starts clockwise at position 3

So there are five people – 1 2 3 4 5 and execution starts clockwise at position 3. Therefore if you kill 3 then are left with the list something like – 1 2 4 5.

Now again you have to kill the person at position 3 clockwise counting from the last killed person.

### Another Solution

Now you need to execute the person at position 1. So executing the person at 1 you are left with the list 2 4 5.

After executing another person you get list 2 4.

Now you are left with 2 persons in the list. So you have to calculate the position wisely to execute the person.

Therefore the calculated position would be  
(starting position of the execution)% (no of people remaining in the list less than starting position)

i.e.  $3\%2 = 1$ .

So finally the winner is 4.





## Josephus Problem



### Program

josephus2.java

#### IO Format

#### Input

$n = 14; k = 2$

#### Output

13

#### Explanation

The chosen place is 13

#### Time Complexity

$O(n)$



### Introduction to Backtracking

The Backtracking is an algorithmic-method to solve a problem with an additional way.

It uses a recursive approach to explain the problems.

- We can say that the backtracking is needed to find all possible combination to solve an optimization problem.

Backtracking is a systematic way of trying out different sequences of decisions until we find one that "works."

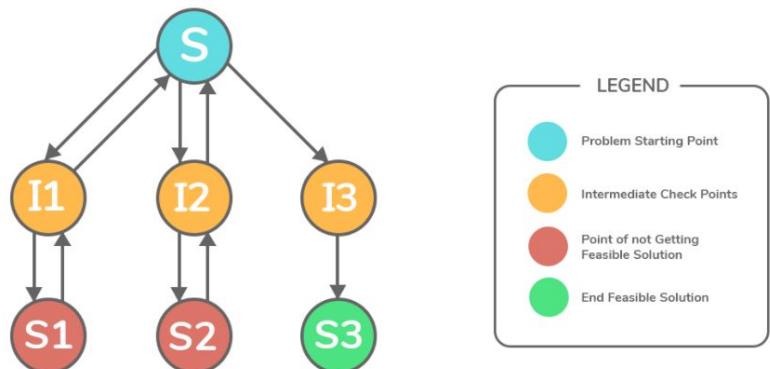
## Introduction to Backtracking

In Backtracking, we "explore" each node, as follows:

Step 1 – if current point is a feasible solution, return success

Step 2 – else if all paths are exhausted (i.e current point is an end point), return failure, since we have no feasible solution.

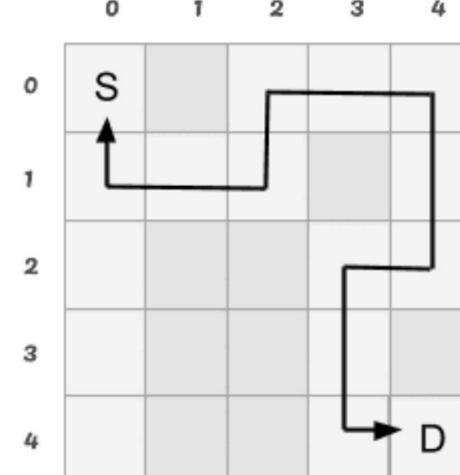
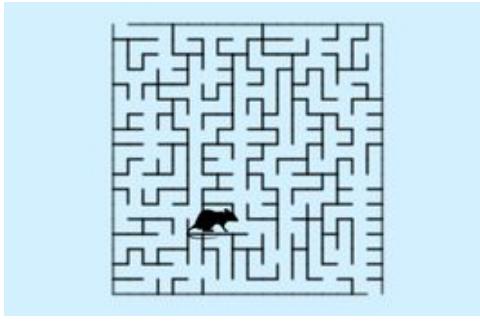
Step 3 – else if current point is not an end point, backtrack and explore other points and repeat above steps.



## Introduction to Rat in a Maze

A Maze is given as  $N \times N$  binary matrix of blocks where source block is the upper left most block  $\text{maze}[0][0]$  and destination block is lower rightmost block  $\text{maze}[N-1][N-1]$ .

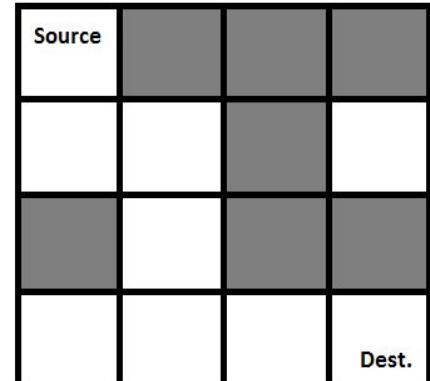
- A rat starts from source and has to reach the destination. The rat can move only in two directions: forward and down.



## Introduction to Rat in a Maze

Representing the Maze on the left as the matrix given below

- [1, 0, 0, 0  
1, 1, 0, 1  
0, 1, 0, 0  
1, 1, 1, 1]



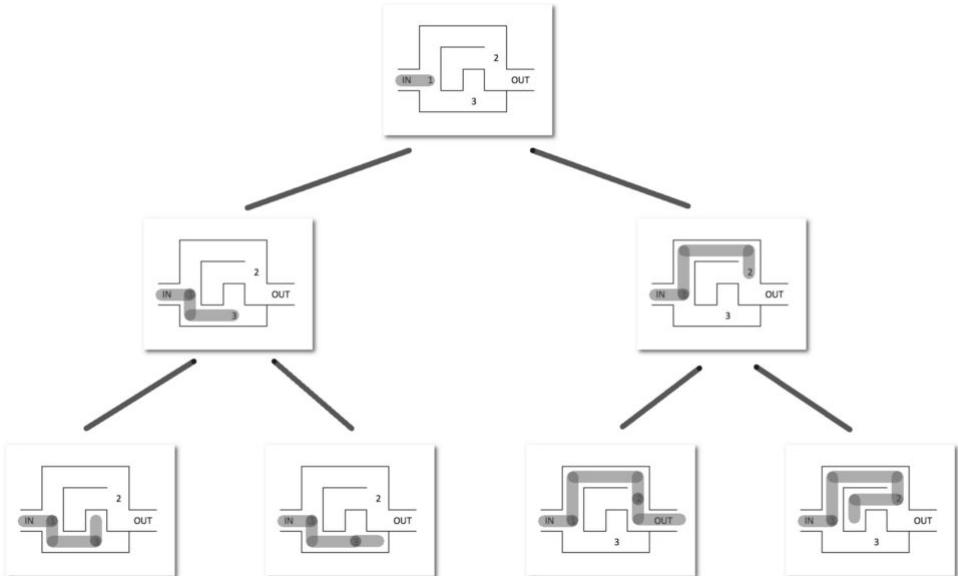
The Solution Matrix is given below where 1's represent the path

- [1, 0, 0, 0  
1, 1, 0, 0  
0, 1, 0, 0  
0, 1, 1, 1]

## Approach

Form a recursive function, which will follow a path and check if the path reaches the destination or not.

- If the path does not reach the destination then backtrack and try other paths.



## Algorithm

1. Create a solution matrix, initially filled with 0's.
2. Create a recursive function, which takes initial matrix, output matrix and position of rat  $(i, j)$ .
3. if the position is out of the matrix or the position is not valid then return.
4. Mark the position  $\text{output}[i][j]$  as 1 and check if the current position is destination or not. If destination is reached print the output matrix and return.
5. Recursively call for position  $(i+1, j)$  and  $(i, j+1)$ .
6. Unmark position  $(i, j)$ , i.e  $\text{output}[i][j] = 0$ .

## Program

ratinmaze1.java

### Sample IO

#### Input

```
int maze[][] = { { 1, 0, 0, 0 },
                 { 1, 1, 0, 1 },
                 { 0, 1, 0, 0 },
                 { 1, 1, 1, 1 } };
```

#### Output

```
1 0 0 0
1 1 0 0
0 1 0 0
0 1 1 1
```

### Time Complexity

$O(2^{n \cdot n})$

### Explanation

The recursion can run upper-bound  $2^{n \cdot n}$  times.

### Space Complexity

$O(n \cdot n)$

## Introduction

N Queens is another example problem that can be solved using Backtracking

The N Queens is the problem of placing N chess queens on an N×N chessboard so that no two queens attack each other

- The solution can be a matrix indicating the position of the queens or the index numbers of the cells in which the queens have been placed

0	1	2	3	4	5	6	7
0	Q						
1							Q
2				Q			
3							Q
4	Q						
5			Q				
6					Q		
7		Q					

Solution { 0 4 7 5 2 6 1 3 }

0	1	2	3	
0			Q	
1	Q			
2				
3				Q

Solution { 1 3 0 2 }



### Backtracking Method of solving

The idea is to place queens one by one in different columns, starting from the leftmost column.

When we place a queen in a column, we check for clashes with already placed queens.

- In the current column, if we find a row for which there is no clash, we mark this row and column as part of the solution.

If we do not find such a row due to clashes then we backtrack and return false.



## Backtracking Algorithm

- 1) Start in the leftmost column
- 2) If all queens are placed return true
- 3) Try all rows in the current column.
  - Do following for every tried row.
    - a) If the queen can be placed safely in this row then mark this [row, column] as part of the solution and recursively check if placing queen here leads to a solution.
    - b) If placing the queen in [row, column] leads to a solution then return true.
    - c) If placing queen doesn't lead to a solution then unmark this [row, column] (Backtrack) and go to step (a) to try other rows.
  - 3) If all rows have been tried and nothing worked, return false to trigger backtracking.



# N-Queens Problem



Explore | Expand | Enrich

## Program

nqueens1.java

### Sample IO

#### Input

8

#### Output

```
Q - - - - - - -  
- - - - - - Q -  
- - - - Q - - -  
- - - - - - - Q  
- Q - - - - - -  
- - - Q - - - -  
- - - - - Q - -  
- - Q - - - - -
```



### Optimization

The idea is not to check every element in right and left diagonal instead use property of diagonals:

1. The sum of  $i$  and  $j$  is constant and unique for each right diagonal where  $i$  is the row of element and  $j$  is the column of element.
2. The difference of  $i$  and  $j$  is constant and unique for each left diagonal where  $i$  and  $j$  are row and column of element respectively.

# Program

nqueens2.java

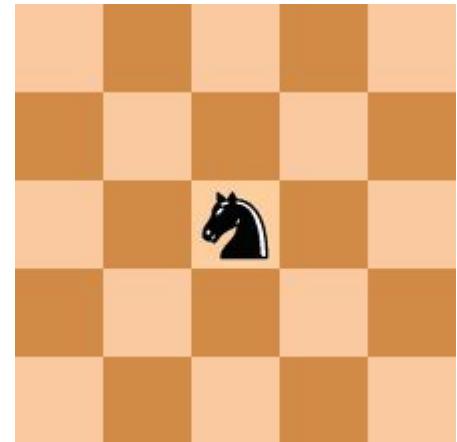
The board size is fixed to 4 here.

## Introduction

A knight's tour is a sequence of moves of a knight on a chessboard such that the knight visits every square exactly once.

- If the knight ends on a square that is one knight's move from the beginning square, the tour is closed; otherwise, it is open

The knight's tour problem is the problem of finding a knight's tour.





## A Knight's Tour

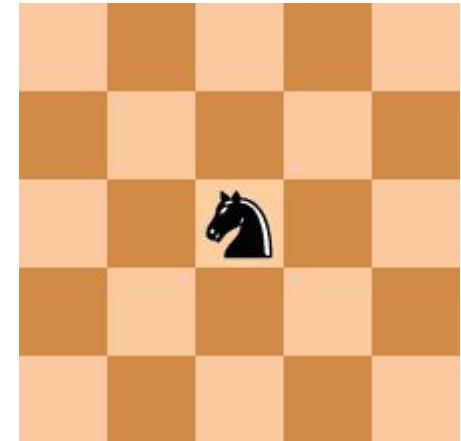


### Problem

Given a  $N \times N$  board with the Knight placed on the first block of an empty board.

Moving according to the rules of chess knight must visit each square exactly once.

Print the order of each cell in which they are visited.



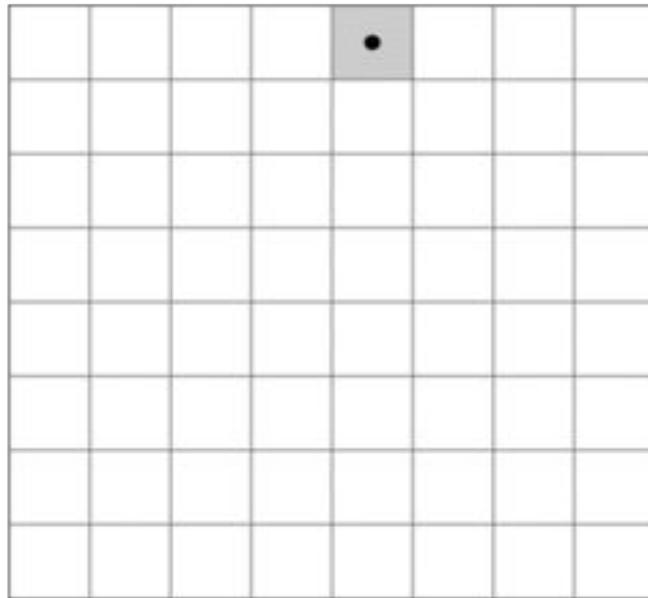


## A Knight's Tour



Explore | Expand | Enrich

An open knight's tour of a chessboard



# A Knight's Tour



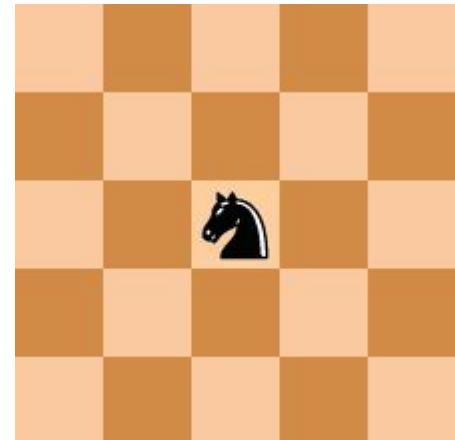
## Sample IO

### Input

N = 8

### Output

1 50 45 62 31 18 9 64  
46 61 32 49 10 63 30 17  
51 2 47 44 33 28 19 8  
60 35 42 27 48 11 16 29  
41 52 3 34 43 24 7 20  
36 59 38 55 26 21 12 15  
53 40 57 4 23 14 25 6  
58 37 54 39 56 5 22 13



### Explanation

The tour is started from the leftmost corner (marked as 1)

## Naive Algorithm

---

```
1 while there are untried tours
2 {
3     generate the next tour
4     if this tour covers all squares
5     {
6         print this path;
7     }
8 }
```

## Backtracking Algorithm

```
1 if all squares are visited
2     print the solution
3 else
4     a) Add one of the next moves to solution vector and recursively
5         check if this move leads to a solution. (A Knight can make maximum
6         eight moves. We choose one of the 8 moves in this step).
7     b) If the move chosen in the above step doesn't lead to a solution
8         then remove this move from the solution vector and try other
9         alternative moves.
10    c) If none of the alternatives work then return false (Returning false
11        will remove the previously added item in recursion and if false is
12        returned by the initial call of recursion then "no solution exists")
13
```



## A Knight's Tour



Explore | Expand | Enrich

### Program

knightstour1.java

The board size is fixed to 8 here

### Output

1	50	45	62	31	18	9	64
46	61	32	49	10	63	30	17
51	2	47	44	33	28	19	8
60	35	42	27	48	11	16	29
41	52	3	34	43	24	7	20
36	59	38	55	26	21	12	15
53	40	57	4	23	14	25	6
58	37	54	39	56	5	22	13



### Problem

Warnsdorff's algorithm for Knight's tour problem

Same problem statement: A knight is placed on the first block of an empty board and, moving according to the rules of chess, must visit each square exactly once.

- Warnsdorff's rule is a heuristic for finding a single knight's tour. The knight is moved so that it always proceeds to the square from which the knight will have the fewest onward moves.

When calculating the number of onward moves for each candidate square, we do not count moves that revisit any square already visited. It is possible to have two or more choices for which the number of onward moves is equal; there are various methods for breaking such ties

### Warnsdorff's Rule

1. We can start from any initial position of the knight on the board.
2. We always move to an adjacent, unvisited square with minimal degree (minimum number of unvisited adjacent).

This algorithm may also more generally be applied to any graph.

Note:

1. A position Q is accessible from a position P if P can move to Q by a single Knight's move, and Q has not yet been visited.
2. The accessibility of a position P is the number of positions accessible from P.

## Warnsdorff's Rule: Algorithm

1. Set  $P$  to be a random initial position on the board
  2. Mark the board at  $P$  with the move number “1”
  3. Do following for each move number from 2 to the number of squares on the board:
    1. let  $S$  be the set of positions accessible from  $P$ .
    2. Set  $P$  to be the position in  $S$  with minimum accessibility
    3. Mark the board at  $P$  with the current move number
  4. Return the marked board — each square will be marked with the move number on which it is visited.
- 

## Warnsdorff's Rule: Algorithm

1. Set  $P$  to be a random initial position on the board
  2. Mark the board at  $P$  with the move number “1”
  3. Do following for each move number from 2 to the number of squares on the board:
    1. let  $S$  be the set of positions accessible from  $P$ .
    2. Set  $P$  to be the position in  $S$  with minimum accessibility
    3. Mark the board at  $P$  with the current move number
  4. Return the marked board — each square will be marked with the move number on which it is visited.
- 



## Warnsdorff's Algorithm



Explore | Expand | Enrich

### Program

knightstour2.java

The board size is fixed to 8 here



### Introduction

Hamiltonian Path in an undirected graph is a path that visits each vertex exactly once.

A Hamiltonian cycle (or Hamiltonian circuit) is a Hamiltonian Path such that there is an edge (in the graph) from the last vertex to the first vertex of the Hamiltonian Path.

The task is to determine whether a given graph contains Hamiltonian Cycle or not

## IO Format

### Input

Adjacency Matrix or Adjacency List representing the graph

If it is an adjacency matrix, it will be a 2D array with '1' at (i,j) if there is a path from i to j.

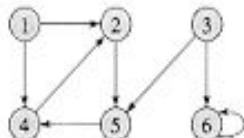
### Output

Array which represents the Hamiltonian Path

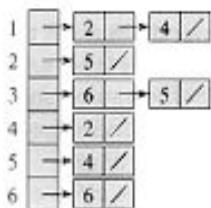
a -> Graph

b -> Adj. List

c -> Adj. Matrix



(a)



(b)

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

(c)

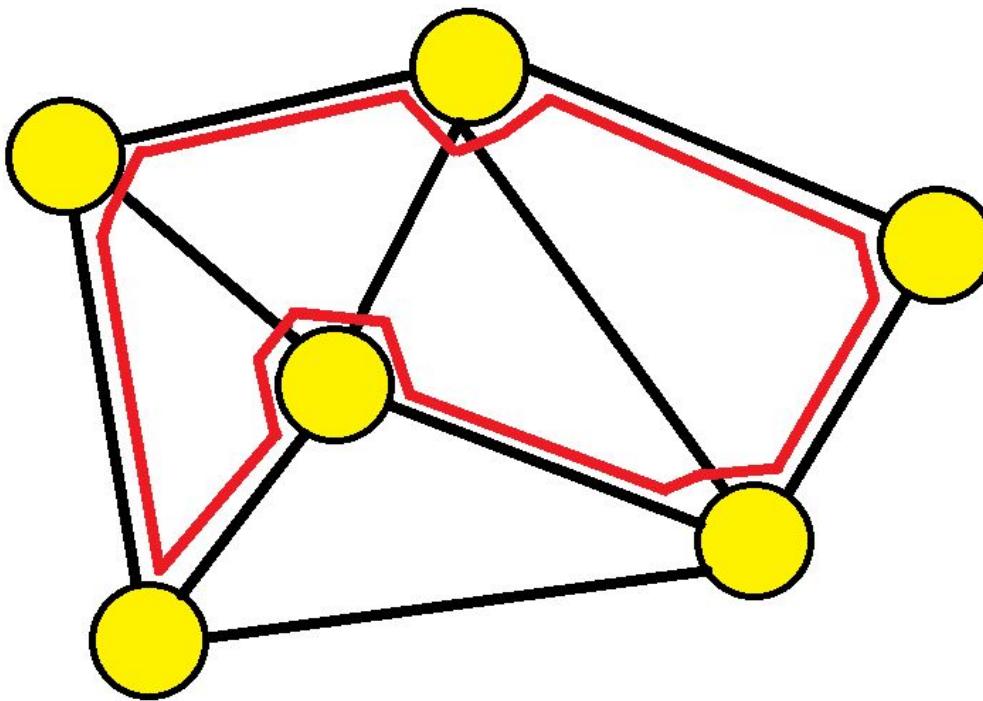


## Hamiltonian Cycle



Explore | Expand | Enrich

### Example



## Backtracking Solution

Naive approach is to generate all possible configurations ( $n!$  configurations in total) of vertices and print a configuration that satisfies the given constraints.

A better approach is to use backtracking

1. Create an empty path array and add vertex 0 to it. Add other vertices, starting from the vertex 1.
2. Before adding a vertex, check for whether it is adjacent to the previously added vertex and not already added.
3. If we find such a vertex, we add the vertex as part of the solution. If we do not find a vertex then we return false.





## Hamiltonian Cycle



Explore | Expand | Enrich

### Program

hamilt1.java

### Sample IO

#### Input

```
8  
0 1 0 1 1 0 0 0  
1 0 1 0 0 1 0 0  
0 1 0 1 0 0 1 0  
1 0 1 0 0 0 0 1  
1 0 0 0 0 1 0 1  
0 1 0 0 1 0 1 0  
0 0 1 0 0 1 0 1  
0 0 0 1 1 0 1 0
```

#### Output

```
0 1 2 3 7 6 5 4 0
```

#### Explanation:

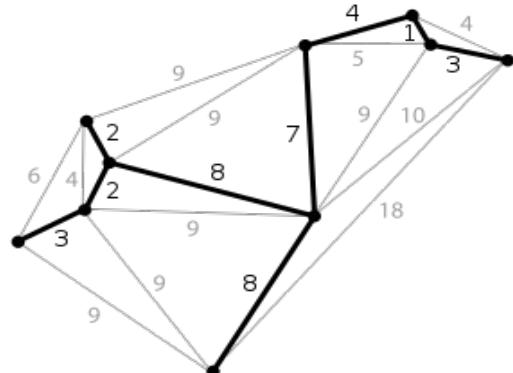
First line in input is the number of vertices  
The next lines would be the matrix



## Minimum Spanning Tree

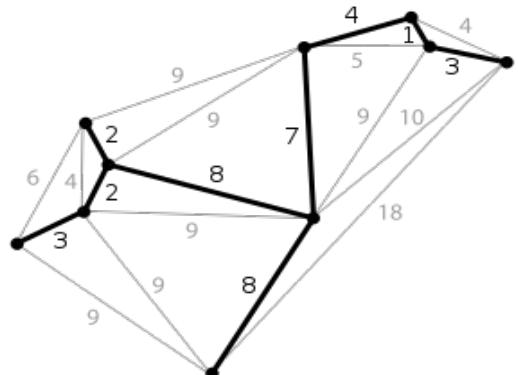
A minimum spanning tree (MST) or minimum weight spanning tree is a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight

- A minimum spanning tree has  $(V - 1)$  edges where  $V$  is the number of vertices in the given graph.



## Algorithm to find MST

1. Sort all the edges in non-decreasing order of their weight.
2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far.
  - If cycle is not formed, include this edge.
  - Else discard it.
3. Repeat step-2 until there are  $(V-1)$  edges in the spanning tree.



Note: Step #2 uses the Union-Find algorithm to detect cycles.

### Algorithm to find MST

1. Sort all the edges in non-decreasing order of their weight.
2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far.
  - If cycle is not formed, include this edge.
  - Else discard it.
3. Repeat step-2 until there are  $(V-1)$  edges in the spanning tree.

Note: Step 2 uses the Union-Find algorithm to detect cycles.



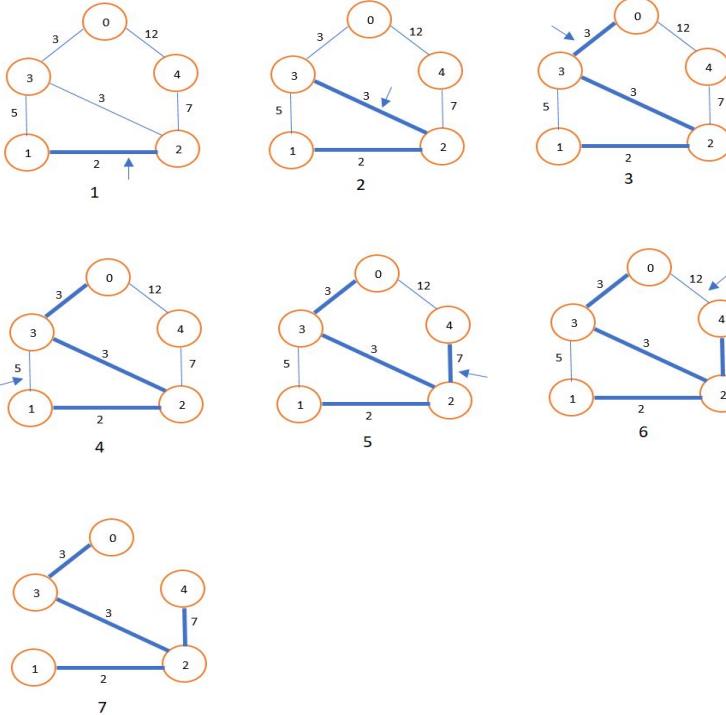
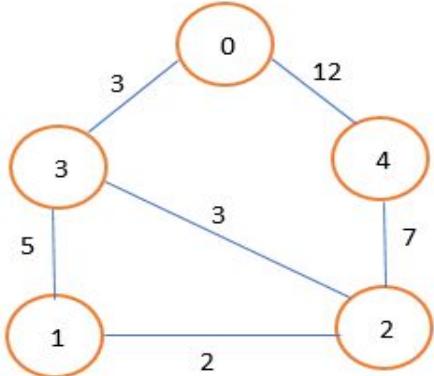


# Kruskal's Algorithm



Explore | Expand | Enrich

## Algorithm in Action



### Disjoint-Set data structure

A disjoint-set data structure that keeps track of a set of elements partitioned into a number of disjoint or non-overlapping subsets.

It provides near-constant-time operations to add new sets, to merge existing sets, and to determine whether elements are in the same set.

Plays a key role in Kruskal's algorithm for finding the minimum spanning tree of a graph.

This can also be used to detect cycle in the graph.



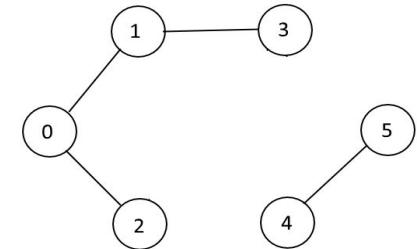
## Disjoint-Set Construction

A disjoint-set forest consists of a number of elements each of which stores an id, a parent pointer

The parent pointers of elements are arranged to form one or more trees, each representing a set.

- If an element's parent pointer points to no other element, then the element is the root of a tree and is the representative member of its set.

A set may consist of only a single element. However, if the element has a parent, the element is part of whatever set is identified by following the chain of parents upwards until a representative element (one without a parent) is reached at the root of the tree.



Disjoint Sets

Set Id: 0, elements: [0, 1, 2, 3]

Set Id: 4, elements: [4, 5]

## Disjoint-Set Operations

The sets shown at the right are disjoint because they have no members in common.

Union-Find supports three basic operations:

- **Find(A):** Determine which subset an element A is in. For example, find(d) would return the subset [ g, d, c ].

**Union(A, B):** Join two subsets that contain A and B into a single subset. For example, union(d, j) would combine [ g, d, c ] and [ i, j ] into the larger set [ g, d, c, i, j ].

**AddSet(A):** Add a new subset containing just that element A. For example, addSet(h) would add a new set [ h ].

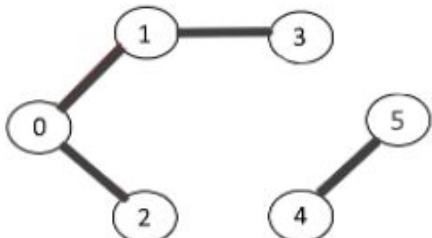
```
[ a, b, f, k ]
[ e ]
[ g, d, c ]
[ i, j ]
```

## Program

disjset1.java

### Sample IO

#### Input



#### Output

Set Id: 0 elements: [0, 1, 2, 3]

Set Id: 4 elements: [4, 5]

#### Explanation:

The Graph is input to the program and it provides a disjoint set output



### Disjoint-Set data structure

A disjoint-set data structure that keeps track of a set of elements partitioned into a number of disjoint or non-overlapping subsets.

It provides near-constant-time operations to add new sets, to merge existing sets, and to determine whether elements are in the same set.

Plays a key role in Kruskal's algorithm for finding the minimum spanning tree of a graph.

This can also be used to detect cycle in the graph.

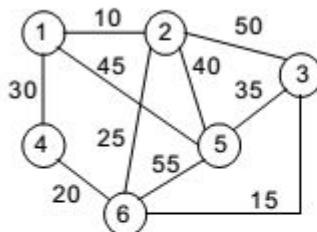




## Kruskal's Algorithm



### Algorithm in Action: Another Example

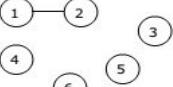
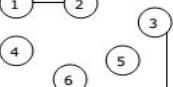
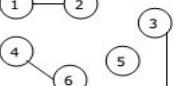
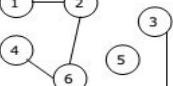
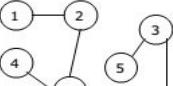


*Arrange all the edges in the increasing order of their costs:*

Cost	10	15	20	25	30	35	40	45	50	55
Edge	(1, 2)	(3, 6)	(4, 6)	(2, 6)	(1, 4)	(3, 5)	(2, 5)	(1, 5)	(2, 3)	(5, 6)



# Kruskal's Algorithm

EDGE	COST	STAGES IN KRUSKAL'S ALGORITHM	REMARKS
(1, 2)	10		The edge between vertices 1 and 2 is the first edge selected. It is included in the spanning tree.
(3, 6)	15		Next, the edge between vertices 3 and 6 is selected and included in the tree.
(4, 6)	20		The edge between vertices 4 and 6 is next included in the tree.
(2, 6)	25		The edge between vertices 2 and 6 is considered next and included in the tree.
(1, 4)	30	Reject	The edge between the vertices 1 and 4 is discarded as its inclusion creates a cycle.
(3, 5)	35		Finally, the edge between vertices 3 and 5 is considered and included in the tree built. This completes the tree. The cost of the minimal spanning tree is 105.



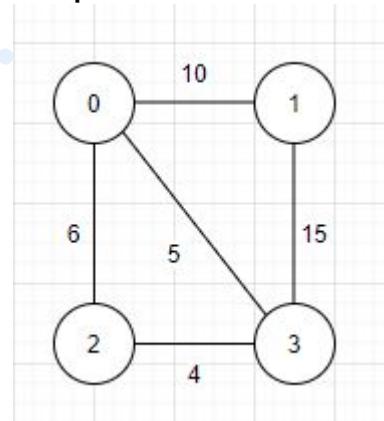
## Kruskal's Algorithm



### Program

kruskal1.java

### Sample IO Input



### Output

Following are the edges in the constructed MST

2 -- 3 == 4

0 -- 3 == 5

0 -- 1 == 10

Minimum Cost Spanning Tree: 19

### Time Complexity

$O(E \log E)$

### Introduction

A greedy algorithm is any algorithm that follows the problem-solving heuristic of making the locally optimal choice at each stage

In many problems, a greedy strategy does not produce an optimal solution, but a greedy heuristic can yield locally optimal solutions that approximate a globally optimal solution in a reasonable amount of time.

Example: In Travelling Sales Problem, the heuristic is "At each step of the journey, visit the nearest unvisited city."

This problem has unreasonably many steps and greedy approach limits them

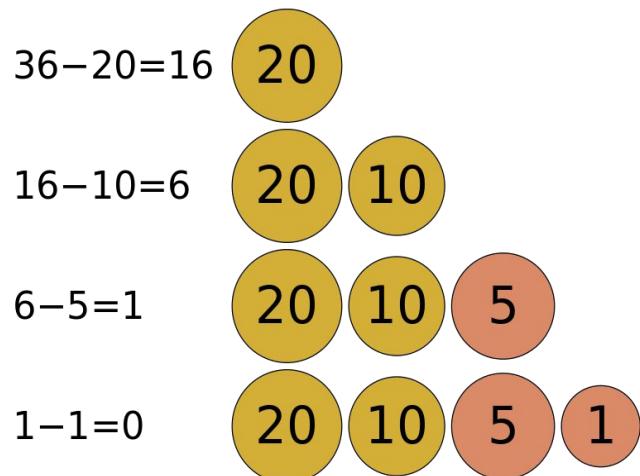


## Example

Greedy algorithms determine the minimum number of coins to give while making change.

The image represents the steps most people would take to emulate a greedy algorithm to represent 36 cents using only coins with values {1, 5, 10, 20}.

The coin of the highest value, less than the remaining change owed, is the local optimum.



# Properties

Greedy Algorithm solves problems by making the best choice that seems best at the particular moment.

A greedy algorithm works if a problem exhibits the following two properties:

- 1. **Greedy Choice Property:** A globally optimal solution can be reached at by creating a locally optimal solution. In other words, an optimal solution can be obtained by creating "greedy" choices.
- 2. **Optimal substructure:** Optimal solutions contain optimal subsolutions. In other words, answers to subproblems of an optimal solution are optimal.

### Introduction

Recap: At every step, we can make a choice that looks best at the moment, and we get the optimal solution of the complete problem.

Activity Selection Problem is a fine example of a Greedy Algorithm

- Problem Statement: You are given  $n$  activities with their start and finish times. Select the maximum number of activities that can be performed by a single person, assuming that a person can only work on a single activity at a time.



## Examples

Consider the following examples

### Input

```
start[] = {10, 12, 20};  
finish[] = {20, 25, 30};
```

### Output

{0,2}

### Explanation

A person can perform at most two activities. The maximum set of activities that can be executed is {0, 2} These are indexes in start[] and finish[]

### Input

```
start[] = {1, 3, 0, 5, 8, 5};  
finish[] = {2, 4, 6, 7, 9, 9};
```

### Output

{0, 1, 3, 4}

### Explanation

A person can perform at most four activities. The maximum set of activities that can be executed is {0, 1, 3, 4}

### Algorithm

The greedy choice is to always pick the next activity whose finish time is least among the remaining activities and the start time is more than or equal to the finish time of the previously selected activity.

- 1) Sort the activities according to their finishing time
- 2) Select the first activity from the sorted array and print it.
- 3) Do the following for the remaining activities in the sorted array.
  - If the start time of this activity is greater than or equal to the finish time of the previously selected activity then select this activity and print it.

# Activity Selection Problem

**Initially :** arr[] = {{ 5,9 }, { 1,2 }, { 3,4 }, { 0,6 }, { 5,7 }, { 8,9 }}

**Step 1:** Sort the array according to finish time  
arr[] = {{ 1,2 }, { 3,4 }, { 0,6 }, { 5,7 }, { 8,9 }, { 5,9 }}

**Step 2:** Print first activity and make i = 0  
print = ({ 1,2 })

**Step 3:** arr[] = {{ 1,2 }, { 3,4 }, { 0,6 }, { 5,7 }, { 8,9 }, { 5,9 }}  
↑  
j

Start[ j ] >= finish[ i ]. print({ 3,4 })  
make i = j , j++

**Step 4:** arr[] = {{ 1,2 }, { 3,4 }, { 0,6 }, { 5,7 }, { 8,9 }, { 5,9 }}  
↑  
j

Start[ j ] < finish[ i ]. j++

**Step 5:** arr[] = {{ 1,2 }, { 3,4 }, { 0,6 }, { 5,7 }, { 8,9 }, { 5,9 }}  
↑  
j

Start[ j ] >= finish[ i ]. print ({ 5,7 })  
make i = j , j++

**Step 6:** arr[] = {{ 1,2 }, { 3,4 }, { 0,6 }, { 5,7 }, { 8,9 }, { 5,9 }}  
↑  
j

make i = j , j++

**Step 6:** arr[] = {{ 1,2 }, { 3,4 }, { 0,6 }, { 5,7 }, { 8,9 }, { 5,9 }}  
Start[ j ] < finish[ i ].

# Program

actsel1.java

Activities are sorted here

- Sample IO

### Input

```
start = {1, 3, 0, 5, 8, 5};  
end = {2, 4, 6, 7, 9, 9};
```

### Output

```
0 1 3 4
```

Time Complexity: O(n)



# Program

actsel2.java

Activities are unsorted here

- Sample IO

### Input

```
start = {1, 3, 0, 5, 8, 5};  
end = {2, 4, 6, 7, 9, 9};
```

### Output

0 1 3 4

Time Complexity:  $O(n \log n + n)$



### Introduction

Graph Coloring is an assignment of labels traditionally called "colors" to elements of a graph subject to certain constraints.

A way of coloring the vertices of a graph such that no two adjacent vertices are of the same color -- a common graph coloring problem

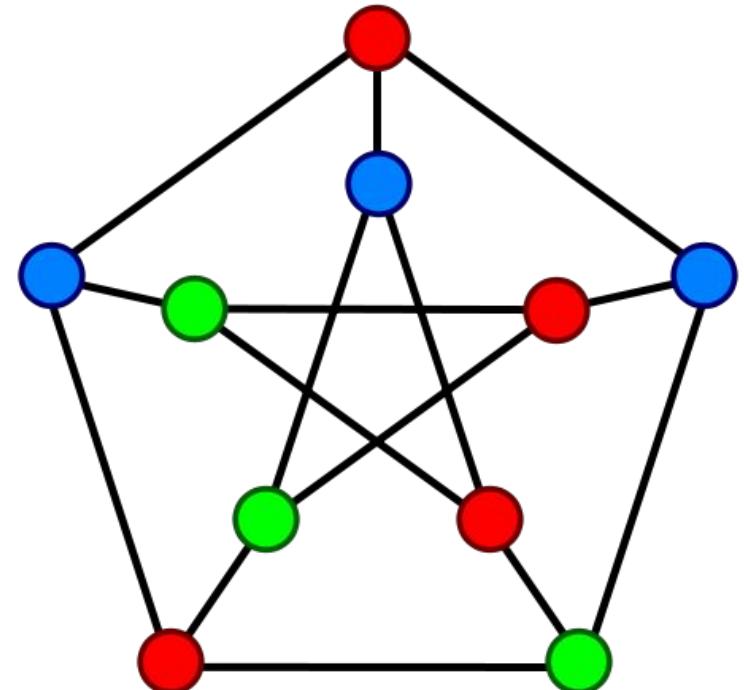
Other kinds of Graph coloring are Edge Coloring and Face coloring

Graph coloring enjoys many practical applications as well as theoretical challenges.

## Example

Vertex coloring of the “Petersen graph” with 3 colors, the minimum number possible

**Chromatic Number:** The smallest number of colors needed to color a graph  $G$  is called its chromatic number. For example, the graph on the right can be colored minimum 3 colors.



# Applications

- 1) Making Schedule or Time Table
- 2) Mobile Radio Frequency Assignment
- 3) Sudoku
- 4) Register Allocation
- 5) Bipartite Graphs
- 6) Map Coloring

### Greedy method to solve it

There is no efficient algorithm available for coloring a graph with minimum number of colors as the problem is a known *NP-Complete* problem

The algorithm doesn't guarantee to use minimum colors, but it guarantees an upper bound on the number of colors.

The basic algorithm never uses more than  $d+1$  colors where  $d$  is the maximum degree of a vertex in the given graph.

### Algorithm

1. Color first vertex with first color.
2. Do following for remaining  $V-1$  vertices.

Consider the currently picked vertex and color it with the lowest numbered color that has not been used on any previously colored vertices adjacent to it.

If all previously used colors appear on vertices adjacent to  $v$ , assign a new color to it



# Graph Coloring



Explore | Expand | Enrich

## Program

graphcoloring1.java

### Output

Coloring of graph 1  
Vertex 0 ---> Color 0  
Vertex 1 ---> Color 1  
• Vertex 2 ---> Color 2  
Vertex 3 ---> Color 0  
Vertex 4 ---> Color 1

### Coloring of graph 2

Vertex 0 ---> Color 0  
Vertex 1 ---> Color 1  
Vertex 2 ---> Color 2  
Vertex 3 ---> Color 0  
Vertex 4 ---> Color 3

### Time Complexity

$O(V^2 + E)$



### Program

graphcoloring1.java

The algorithm doesn't always use minimum number of colors.

The number of colors used sometime depend on the order in which vertices are processed.

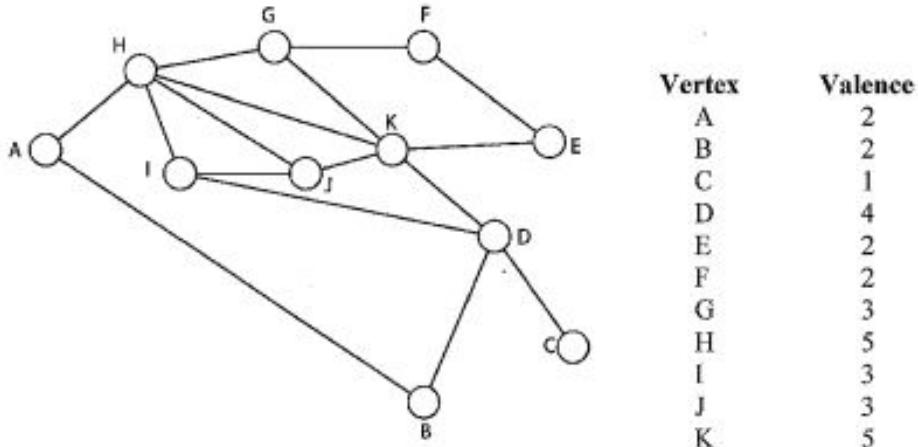
Order in which the vertices are picked is important. There are different ways to do so

The most common is Welsh–Powell Algorithm which considers vertices in descending order of degrees.

## Welsh Powell Algorithm

1. Find the degree of each vertex
2. List the vertices in order of descending degrees.
3. Colour the first vertex with color 1.
4. Move down the list and color all the vertices not connected to the coloured vertex, with the same color.
5. Repeat step 4 on all uncolored vertices with a new color, in descending order of degrees until all the vertices are coloured.

## Welsh Powell Algorithm



Arrange the list of vertices in descending order of valence. If there's a tie, you can randomly choose how to break it. (To get the ordering below, we used alphabetical order to break ties.) The new order will be:

*H, K, D, G, I, J, A, B, E, F, C*

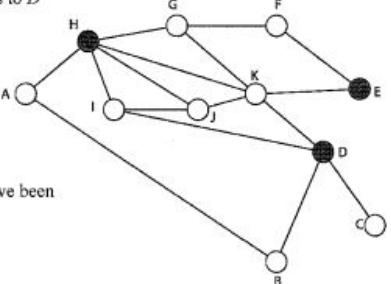
# Welsh Powell Algorithm

## Graph Coloring

Now we color the vertices, in the order listed above. Here's the thought process for the first color:

- H color red
- K don't color red since it connects to H
- D color red
- G don't color red since it connects to H
- I don't color red since it connects to H
- J don't color red since it connects to H
- A don't color red since it connects to H
- B don't color red since it connects to D
- E color red
- F don't color red since it connects to E
- C don't color red since it connects to D

Now the graph should look like this:



If we now ignore the vertices that have been already colored, we're left with:

K, G, I, J, A, B, F, C

We can repeat the process now with a second color (blue):

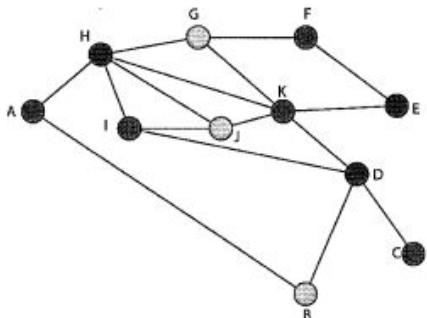
- K color blue
- G don't color blue since it connects with K
- I color blue
- J don't color blue since it connects with I
- A color blue
- B don't color blue since it connects with A
- F color blue
- C color blue

Again, cross out the colored vertices, leaving you with G, J, B, and start at the top of the new list with a new color (yellow):

- G color yellow
- J color yellow
- B color yellow

The final colored graph is shown at right.

We see that by using Welsh-Powell we can get away with using just three colors. In this case, three colors turns out to be the optimal solution since the graph contains at least one triangle (for example, there's the triangle  $HJH$ ), and it will always take at least 3 colors to color a graph with a triangle in it.



## Introduction

Huffman coding is a lossless data compression algorithm.

It forms the basis for file-compression methods

Every information in computer science is encoded as strings of 1s and 0s.

- The objective of information theory is to usually transmit information using fewest number of bits in such a way that every encoding is unambiguous.

We discuss about fixed-length and variable-length encoding along with Huffman Encoding

## Introduction

### Fixed-Length encoding

Every character is assigned a binary code using same number of bits. Thus, a string like “aabacdad” can require 64 bits (8 bytes) for storage or transmission, assuming that each character uses 8 bits.

### Variable-Length encoding

- As opposed to Fixed-length encoding, this scheme uses variable number of bits for encoding the characters depending on their frequency in the given text.

Thus, for a given string like “aabacdad”, frequency of characters ‘a’, ‘b’, ‘c’ and ‘d’ is 4,1,1 and 2 respectively. Since ‘a’ occurs more frequently than ‘b’, ‘c’ and ‘d’, it uses least number of bits, followed by ‘d’, ‘b’ and ‘c’.

### Fixed and Variable encoding scheme

Suppose we randomly assign binary codes to each character as follows  
a - 0, b - 011, c - 111, d - 11

Thus, the string “aabacdad” gets encoded to 0001101111011 (0 | 0 | 011 | 0 | 111 | 11 | 0 | 11) using variable encoding scheme.

- Thus variable encoding scheme can potentially occupy lesser space

But the real problem lies with the decoding phase. There would be ambiguity here.

To prevent such ambiguities during decoding, the encoding phase should satisfy the “prefix rule”

## Prefix rule in variable encoding scheme

States that no binary code should be a prefix of another code

This will produce uniquely decodable codes

In the example, the codes for 'a', 'b', 'c' and 'd' do not follow prefix rule since the binary code for a, i.e. 0, is a prefix of binary code for b i.e 011, resulting in ambiguous decodable codes

Thus, on following the rules, we can reassign the binary codes as  
a - 0, b - 11, c - 101, d - 100

Using the above codes, string “aabacdad” gets encoded to 001101011000100 (0 | 0 | 11 | 0 | 101 | 100 | 0 | 100) which is easier to decode

### Problem Statement

Given a set of symbols to be transmitted or stored along with their frequencies or weights,

One has to find the prefix-free and variable-length binary codes with minimum expected codeword length.

- Equivalently, a tree-like data structure with minimum weighted path length from root can be used for generating the binary codes

Huffman Encoding, developed by David Huffman in 1951 can be used for this

## Steps

1. Building a Huffman tree using the input set of symbols and weight/ frequency for each symbol
  1. A Huffman tree, similar to a binary tree data structure, needs to be created having  $n$  leaf nodes and  $n-1$  internal nodes
  2. Priority Queue is used for building the Huffman tree such that nodes with lowest frequency have the highest priority. (built using Min Heap)
  3. Initially, all nodes are leaf nodes containing the character itself along with the weight/ frequency of that character
  4. Internal nodes, on the other hand, contain weight and links to two child nodes
2. Assigning the binary codes to each symbol by traversing Huffman tree

## Introduction

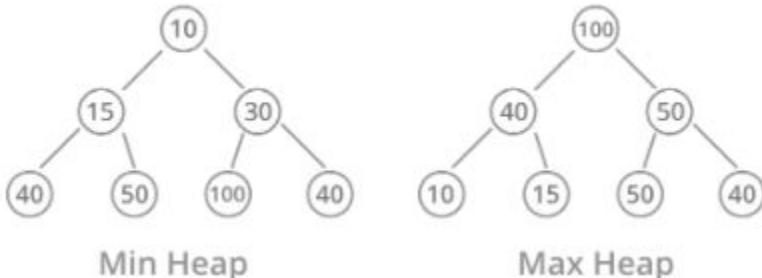
A Binary Heap is a Binary Tree with following properties

- 1) It's a complete tree (All levels are completely filled except possibly the last level and the last level has all keys as left as possible).

- This property of Binary Heap makes them suitable to be stored in an array.

- 2) A Binary Heap is either Min Heap or Max Heap.

In a Min Binary Heap, the key at root must be minimum among all keys present in Binary Heap. The same property must be recursively true for all nodes in Binary Tree. Max Binary Heap is similar to MinHeap.



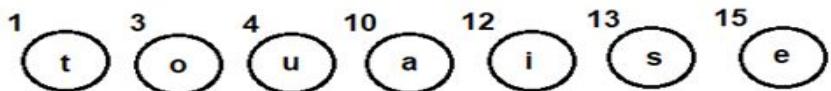
## Algorithm for creating the Huffman Tree

1. Create a leaf node for each character and build a min heap using all the nodes  
(The frequency value is used to compare two nodes in min heap)
2. Repeat Steps 3 to 5 while heap has more than one node
3. Extract two nodes, say x and y, with minimum frequency from the heap
4. Create a new internal node z with x as its left child and y as its right child. Also  
 $\text{frequency}(z) = \text{frequency}(x) + \text{frequency}(y)$
5. Add z to min heap
6. Last node in the heap is the root of Huffman tree

# Huffman Coding

## Example

Create leaf nodes for all the characters and add them to the min heap.



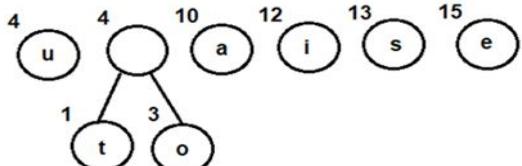
Repeat the following steps till heap has more than one nodes

Extract two nodes, say x and y, with minimum frequency from the heap

- Create a new internal node z with x as its left child and y as its right child  
Add z to min heap

Extract and combine node u with an internal node having 4 as the frequency

Add the new internal node to priority queue



Characters	Frequencies
a	10
e	15
i	12
o	3
u	4
s	13
t	1

# Huffman Coding

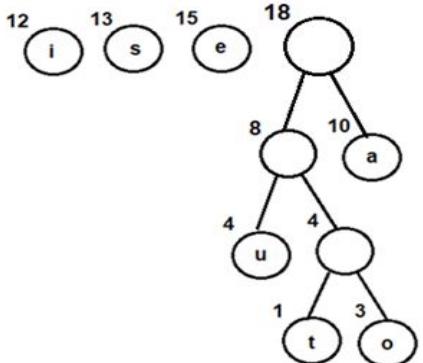
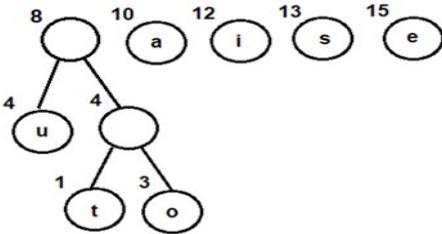
## Example

Extract and Combine node a with an internal node having 8 as the frequency

Add the new internal node to priority queue

Extract and Combine nodes i and s

Add the new internal node to priority queue

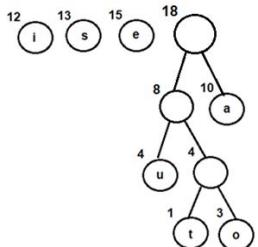


Characters	Frequencies
a	10
e	15
i	12
o	3
u	4
s	13
t	1

# Huffman Coding

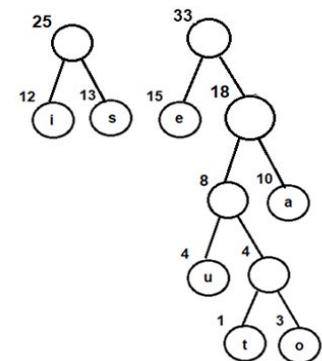
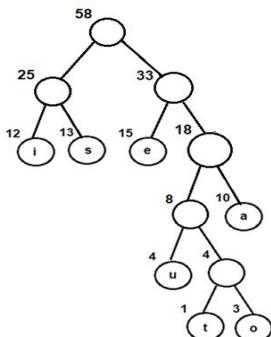
Similarly this goes on to

Extract and Combine nodes i and s from =>



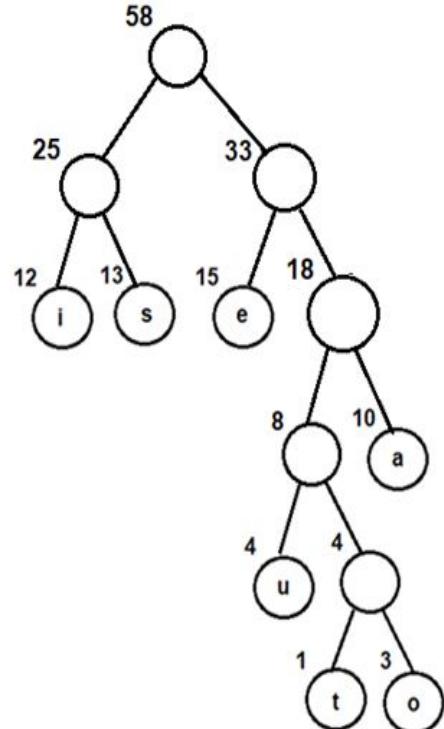
Extract and Combine node ewith an internal node having 18 as the frequency from =====>

Extract and Combine internal nodes having 25 and 33 as the frequency to form



## Algorithm for creating the Huffman Tree

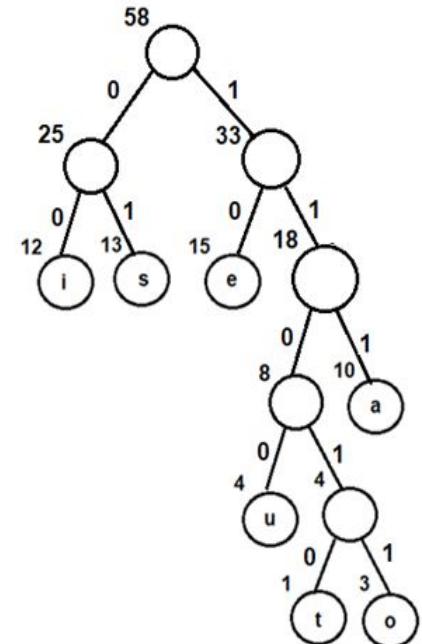
Since internal node with frequency 58 is the only node in the queue, it becomes the root of Huffman tree.



## Steps for traversing the Huffman Tree

1. Create an auxiliary array
2. Traverse the tree starting from root node
3. Add 0 to array while traversing the left child and add 1 to array while traversing the right child
4. Print the array elements whenever a leaf node is found

Following the above steps for Huffman Tree generated above, we get prefix-free and variable-length binary codes with minimum expected codeword length



## Steps for traversing the Huffman Tree

Suppose the string “staeiou” needs to be transmitted from computer A (sender) to computer B (receiver) across a network. Using concepts of Huffman encoding, the string gets encoded to “0111010111100011011110011010” (01 | 11010 | 111 | 10 | 00 | 11011 | 1100 | 11010) at the sender side.

Once received at the receiver’s side, it will be decoded back by traversing the Huffman tree. For decoding each character, we start traversing the tree from root node. Start with the first bit in the string. A ‘1’ or ‘0’ in the bit stream will determine whether to go left or right in the tree. Print the character, if we reach a leaf node.

Characters	Binary Codes
i	00
s	01
e	10
u	1100
t	11010
o	11011
a	111

## Decoding

For “staeiou”



## Program

huffman1.java

### Output

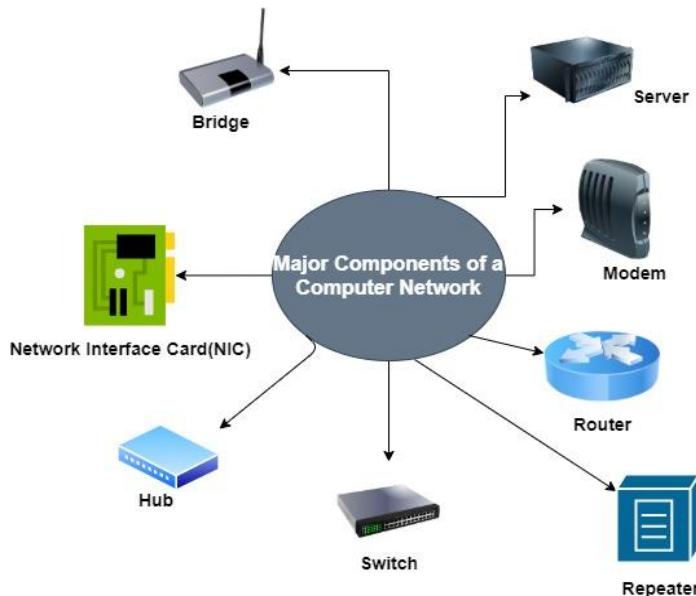
```
f: 0
c: 100
d: 101
a: 1100
b: 1101
e: 111
```

Time Complexity is  $O(n \log n)$  where  $n$  is the number of unique characters. If there are  $n$  nodes, `extractMin()` is called  $2*(n - 1)$  times. `extractMin()` takes  $O(\log n)$  time as it calls `minHeapify()`. So, overall complexity is  $O(n \log n)$ .



## Introduction

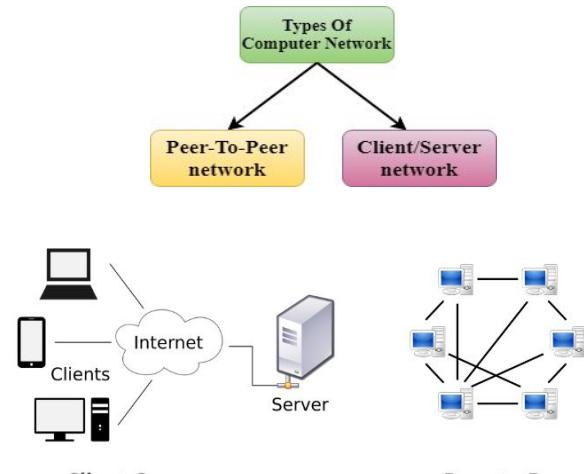
Computer Network is a group of computers connected with each other through wires, optical fibres or optical links so that various devices can interact with each other through a network.



## Architecture

Computer Network Architecture is defined as the physical and logical design of the software, hardware, protocols, and media of the transmission of data. Simply we can say that how computers are organized and how tasks are allocated to the computer.

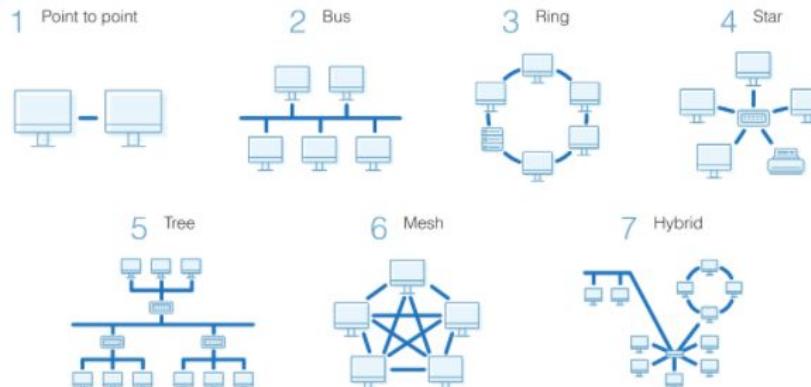
	Peer to Peer Network	Client/Server network
Hardware Cost	It needs no high-end server as the resources are distributed over all clients which reduce cost.	A dedicated computer server (hardware) that distributes resources is needed.
Easy Setup	It is easy to setup mainly if the computers are less than fifty (50).	It is difficult to setup.
Network Operating system	There is no required network operating.	Network operating system is required.
Failure	It can accommodate failure i.e. if one or more Computers (clients) fail the others can still be up.	It cannot accommodate failure if the server fails.
Security	It has security deficiency as clients' administration is not guaranteed.	Very secure because server administration is guaranteed.
Performance	It performs less	Performs very good
Backup	It has decentralized backup that is difficult to access.	It has a centralized data backup with ease of access.



## Topologies

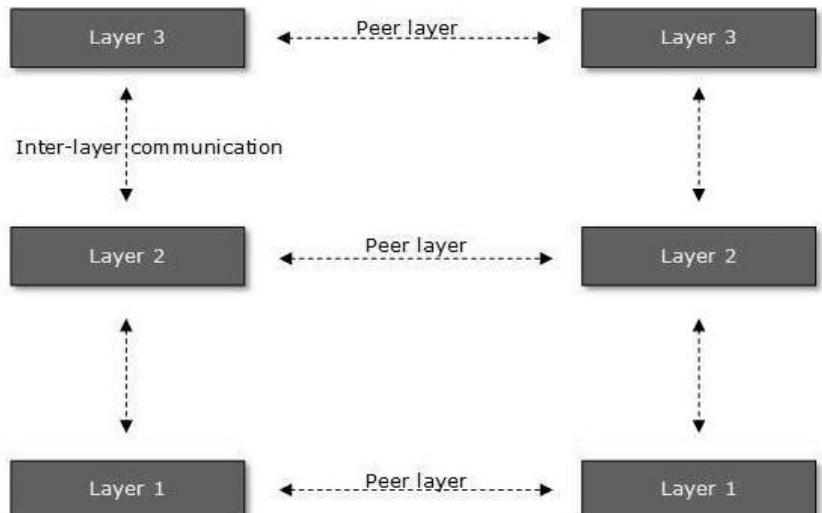
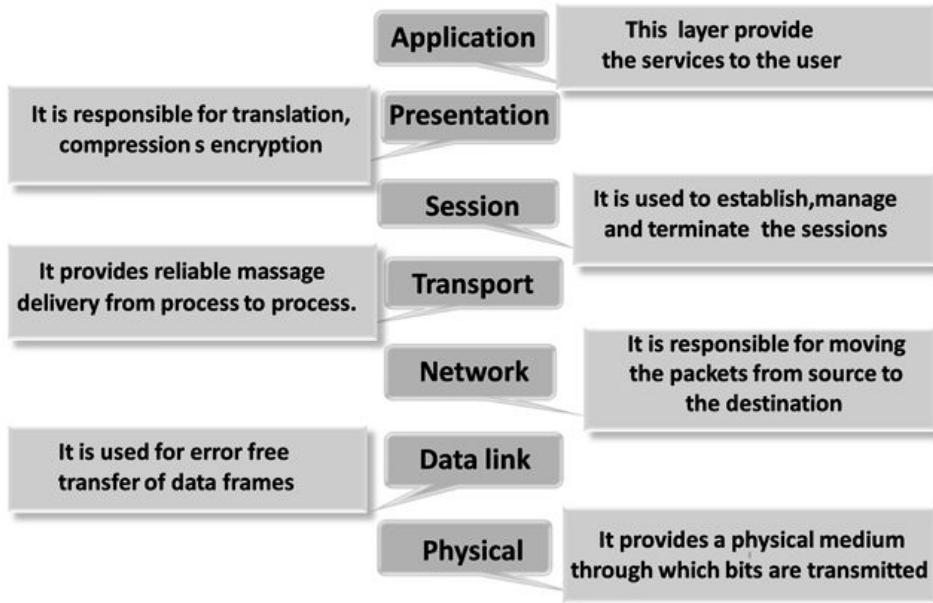
Topology defines the structure of the network of how all the components are interconnected to each other.

There are two types of topology: physical and logical topology.

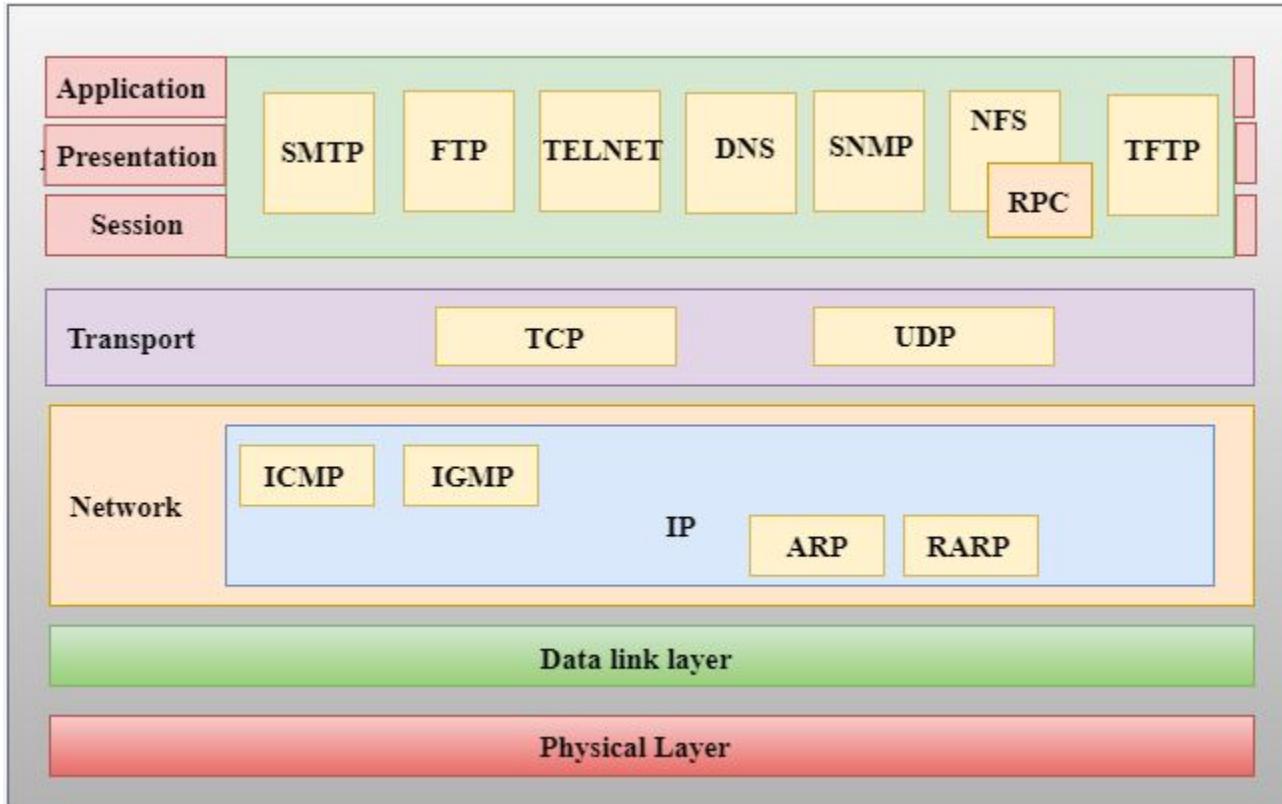


# Networking

## OSI model



# Networking



# Networking



Explore | Expand | Enrich

OSI (Open Source Interconnection) 7 Layer Model

Layer	Application/Example	Central Device/Protocols	DOD4 Model
<b>Application (7)</b>  Serves as the window for users and application processes to access the network services.	<b>End User layer</b> Program that opens what was sent or creates what is to be sent Resource sharing • Remote file access • Remote printer access • Directory services • Network management	User Applications SMTP	
<b>Presentation (6)</b>  Formats the data to be presented to the Application layer. It can be viewed as the "Translator" for the network.	<b>Syntax layer</b> encrypt & decrypt (if needed)  Character code translation • Data conversion • Data compression • Data encryption • Character Set Translation	JPEG/ASCII EBCDIC/TIFF/GIF PICT	Process
<b>Session (5)</b>  Allows session establishment between processes running on different stations.	<b>Synch &amp; send to ports</b> (logical ports)  Session establishment, maintenance and termination • Session support - perform security, name recognition, logging, etc.	Logical Ports  RPC/SQL/NFS NetBIOS names	
<b>Transport (4)</b>  Ensures that messages are delivered error-free, in sequence, and with no losses or duplications.	<b>TCP</b> Host to Host, Flow Control  Message segmentation • Message acknowledgement • Message traffic control • Session multiplexing	F I L T E R E G K E T P A C K E T R O U T E R S TCP/SPX/UDP	Host to Host
<b>Network (3)</b>  Controls the operations of the subnet, deciding which physical path the data takes.	<b>Packets</b> ("letter", contains IP address)  Routing • Subnet traffic control • Frame fragmentation • Logical-physical address mapping • Subnet usage accounting	Routers IP/IPX/ICMP	Internet
<b>Data Link (2)</b>  Provides error-free transfer of data frames from one node to another over the Physical layer.	<b>Frames</b> ("envelopes", contains MAC address) [NIC card —> Switch —> NIC card] (end to end) Establishes & terminates the logical link between nodes • Frame traffic control • Frame sequencing • Frame acknowledgment • Frame delimiting • Frame error checking • Media access control	Switch Bridge WAP PPP/SLIP Land Based Layers	Can be used on all layers
<b>Physical (1)</b>  Concerned with the transmission and reception of the unstructured raw bit stream over the physical medium.	<b>Physical structure</b> Cables, hubs, etc.  Data Encoding • Physical medium attachment • Transmission technique - Baseband or Broadband • Physical medium transmission Bits & Volts	Hub	Network

## Network Layer

**Routing:** When a packet reaches the router's input link, the router will move the packets to the router's output link. For example, a packet from S1 to R1 must be forwarded to the next router on the path to S2.

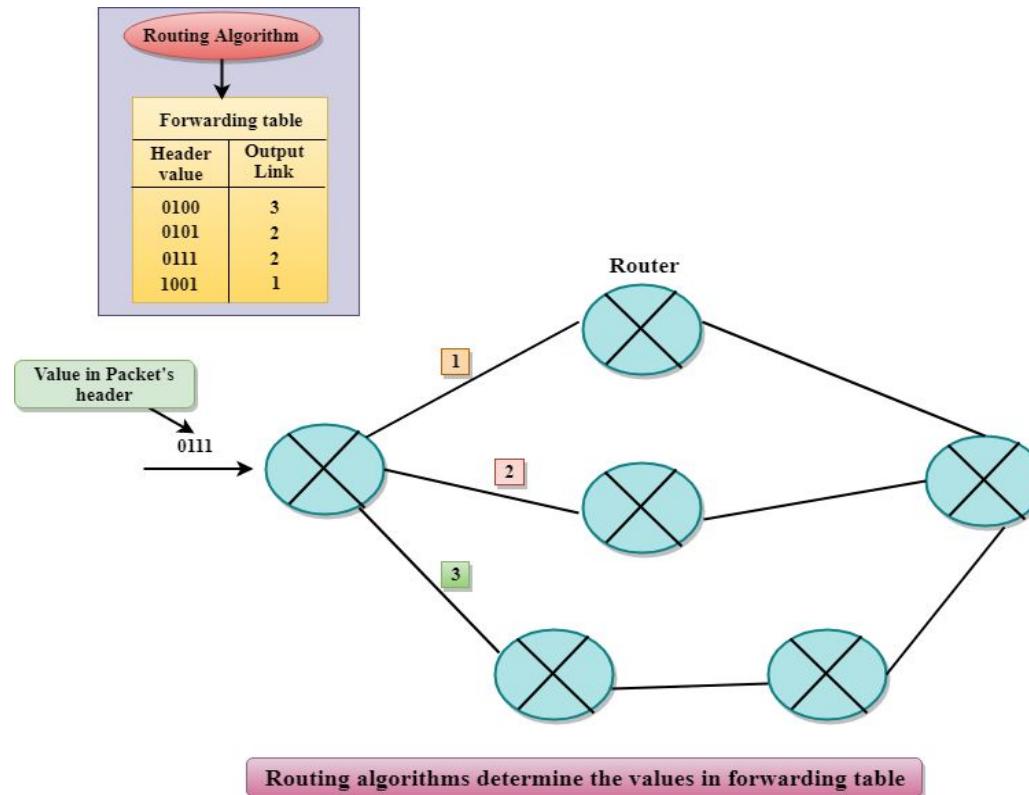
**Logical Addressing:** The data link layer implements the physical addressing and network layer implements the logical addressing. Logical addressing is also used to distinguish between source and destination system. The network layer adds a header to the packet which includes the logical addresses of both the sender and the receiver.

**Internetworking:** This is the main role of the network layer that it provides the logical connection between different types of networks.

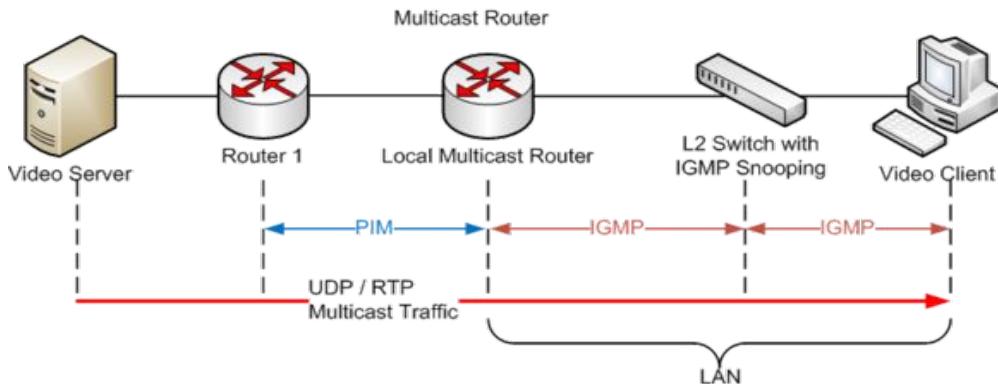
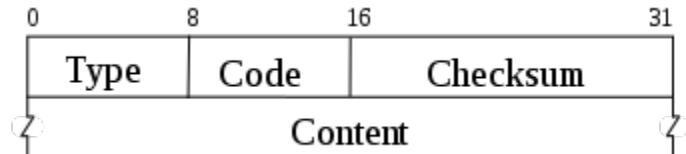
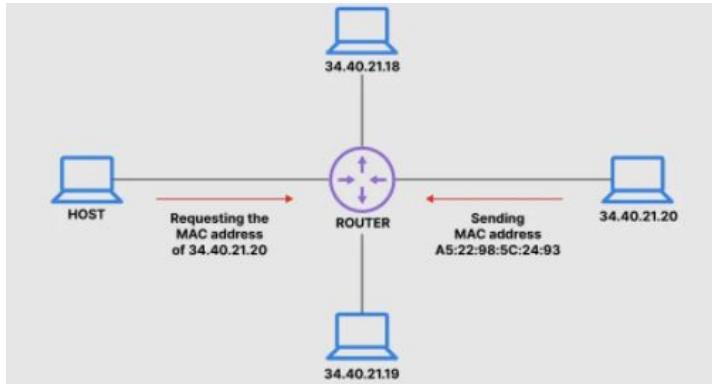
**Fragmentation:** The fragmentation is a process of breaking the packets into the smallest individual data units that travel through different networks.



## Routing in the network layer



## Protocols: ARP, RARP and ICMP



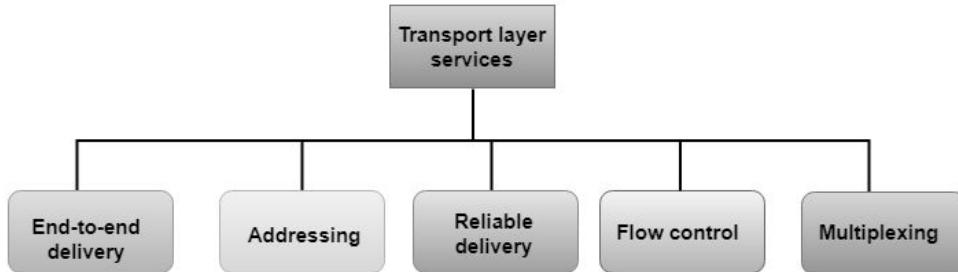
## Transport Layer

The main role of the transport layer is to provide the communication services (logical communication) directly to the application processes running on different hosts

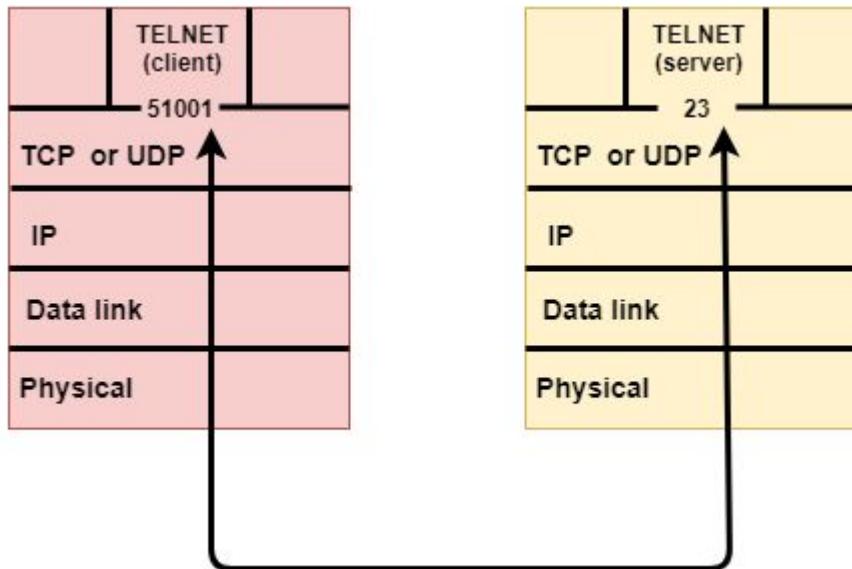
The transport layer protocols are implemented in the end systems but not in the network routers.

All transport layer protocols provide multiplexing/demultiplexing service. It also provides other services such as reliable data transfer, bandwidth guarantees, and delay guarantees.

Applications make use of either TCP/UDP and read and write to the transport layer



## Transport Layer

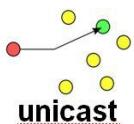


## Transport Layer



TCP

- Slower but reliable transfers
- Typical applications:
  - Email
  - Web browsing

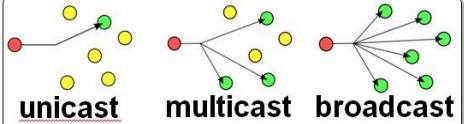


unicast



UDP

- Fast but non-guaranteed transfers ("best effort")
- Typical applications:
  - VoIP
  - Music streaming



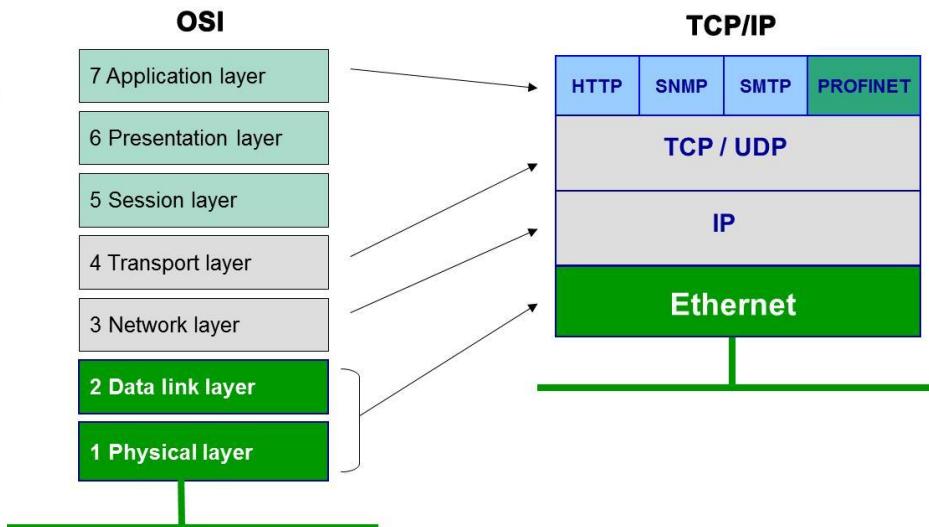
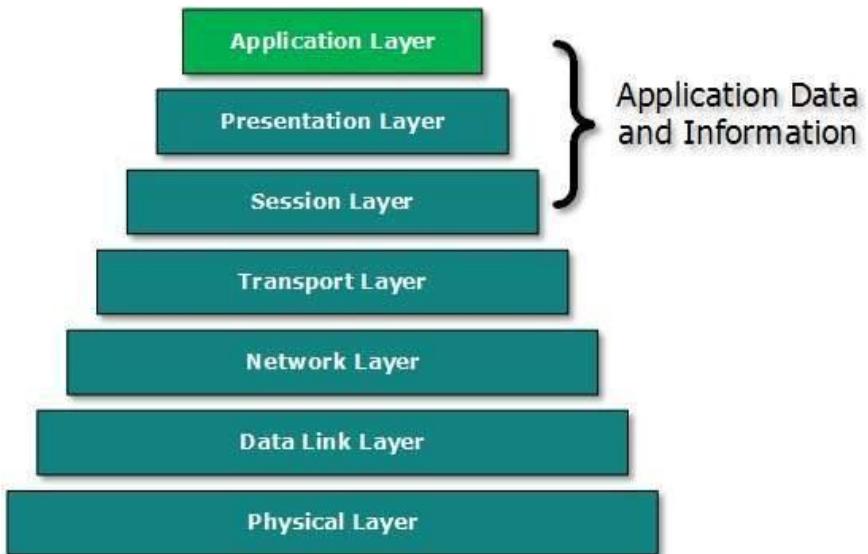
unicast

multicast broadcast

## TCP vs UDP

- |  |   |
|--|---|
| <ul style="list-style-type: none"><li>• Connected</li><li>• State Memory</li><li>• Byte Stream</li><li>• Ordered Data Delivery</li><li>• Reliable</li><li>• Error Free</li><li>• Handshake</li><li>• Flow Control</li><li>• Relatively Slow</li><li>• Point to Point</li><li>• Security: SSL/TLS</li></ul> | <ul style="list-style-type: none"><li>• Connectionless</li><li>• Stateless</li><li>• Packet/Datagram</li><li>• No Sequence Guarantee</li><li>• Lossy</li><li>• Error Packets Discarded</li><li>• No Handshake</li><li>• No Flow Control</li><li>• Relatively Fast</li><li>• Supports Multicast</li><li>• Security: DTLS</li></ul> |
|--|---|

## Application Layer



## Application Layer Protocols

Domain Name System (DNS)

Simple Mail Transfer Protocol (SMTP)

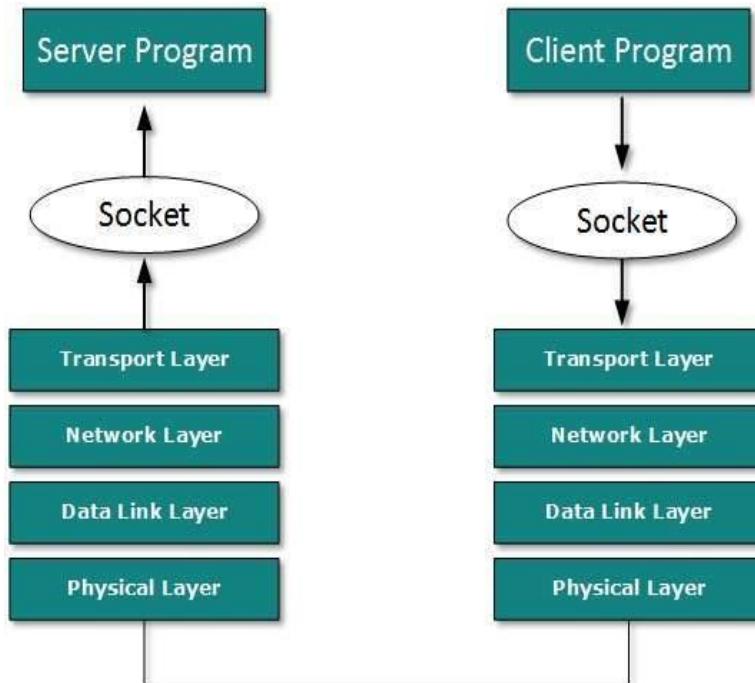
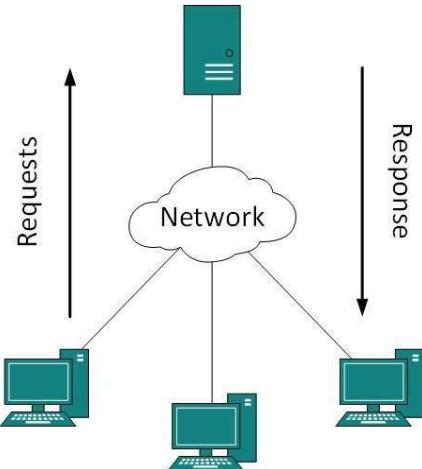
File Transfer Protocol (FTP)

Post Office Protocol (POP)

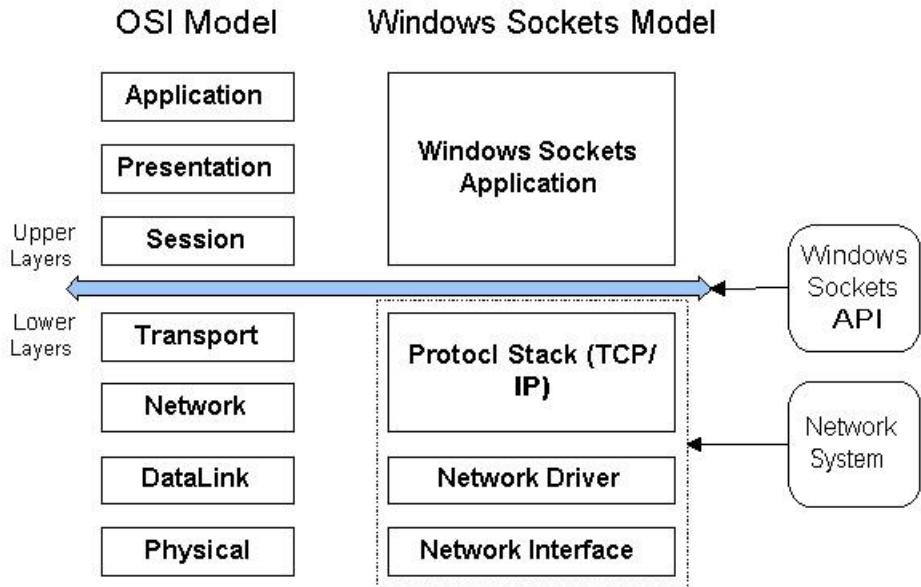
Hyper Text Transfer Protocol (HTTP)

Transport Layer Security (TLS)

## Client Server Model



## Example: The WinSock Network model

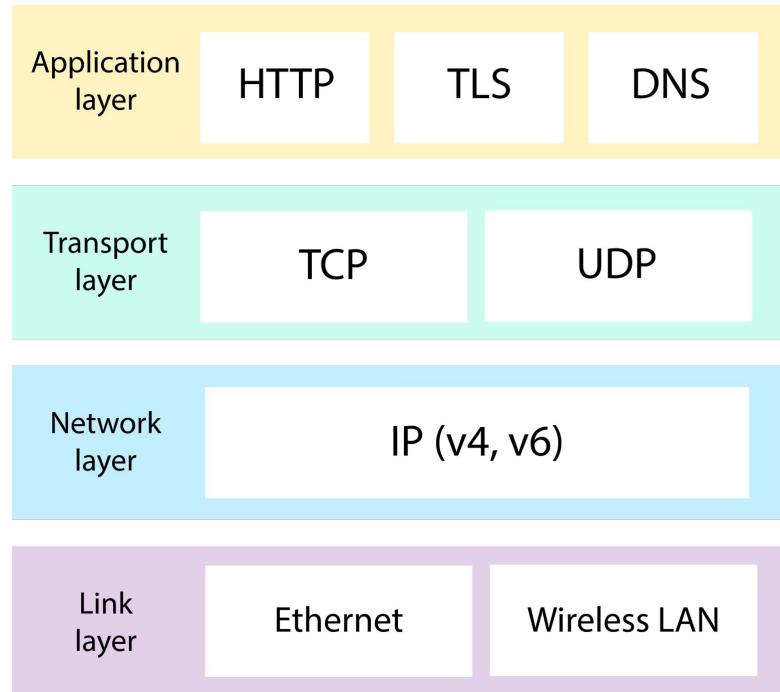


# Networking

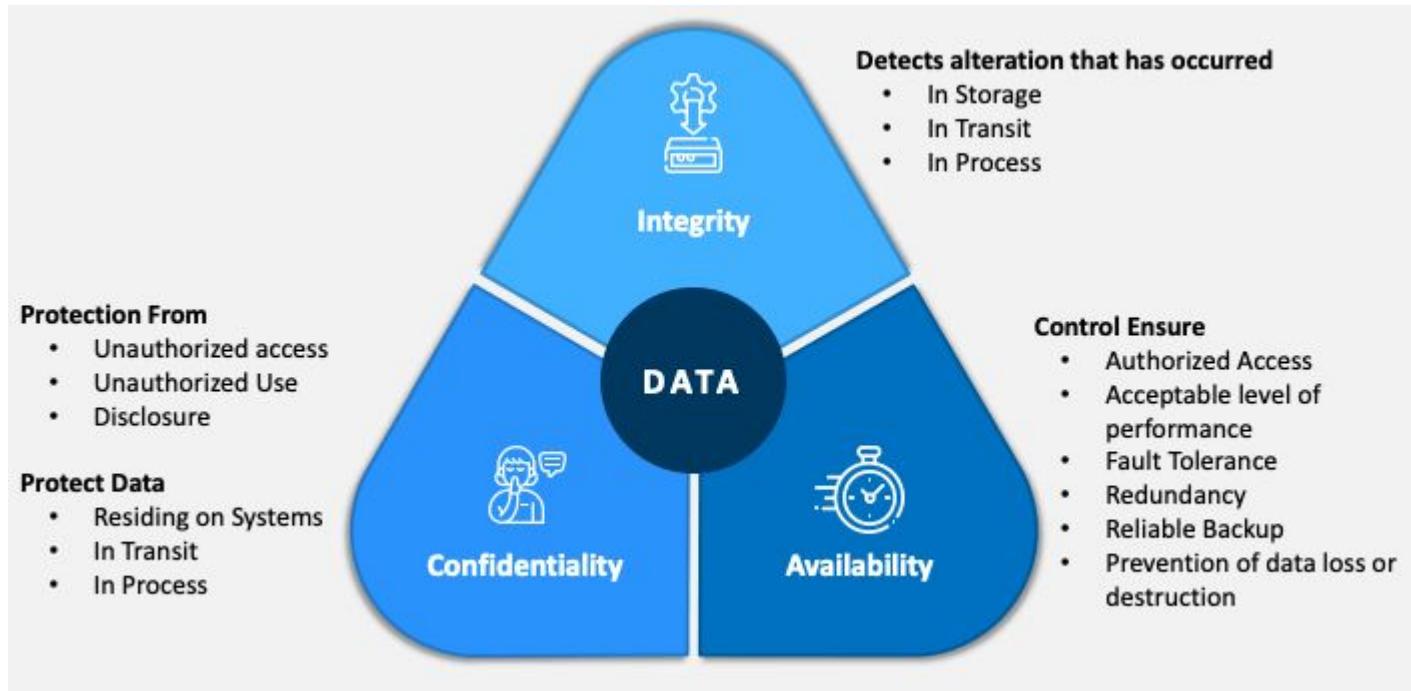


Explore | Expand | Enrich

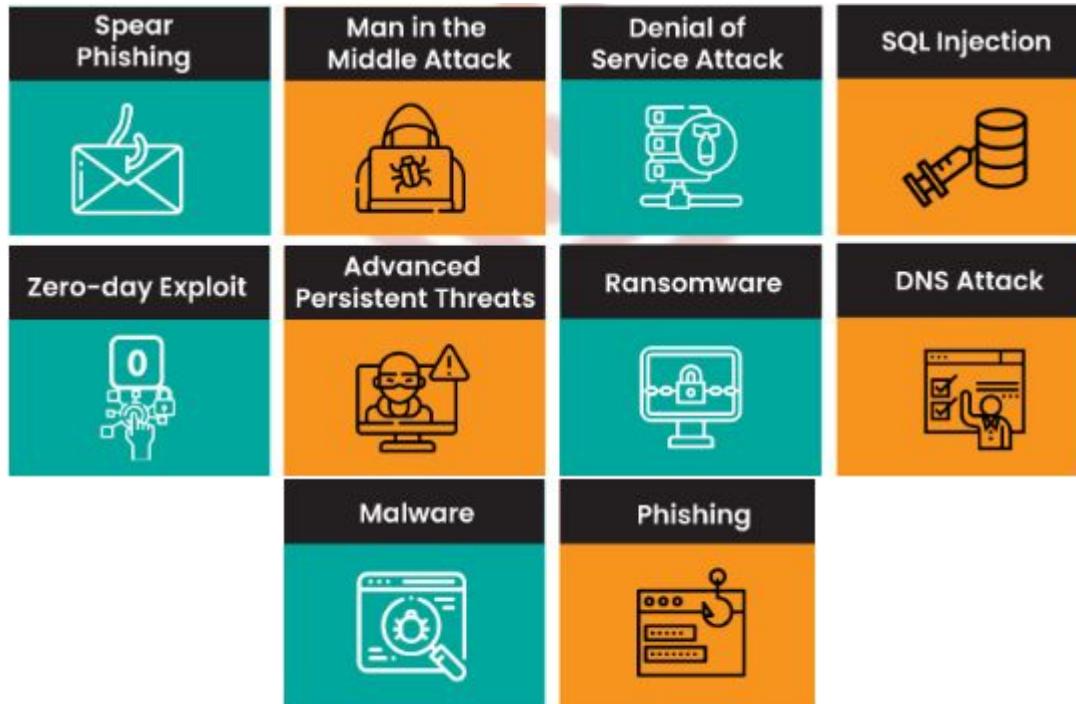
## Summary



## The CIA Triad



## Types of Security Threats





/ethnuscodemithra



Ethnus Codemithra



/ethnus



/code\_mithra

# THANK YOU

<https://www.codemithra.com/>



Explore | Expand | Enrich



codemithra@ethnus.com



+91 7815 095 095



+91 9019 921 340