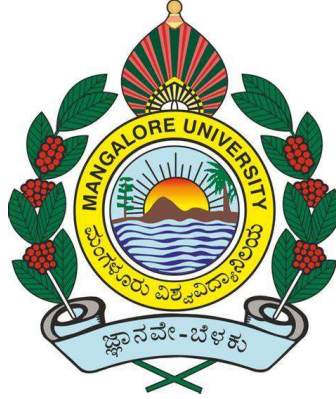


# MANGALORE UNIVERSITY



## DEPARTMENT OF ELECTRONICS

### PRACTICAL JOURNAL

NAME : \_\_\_\_\_

SEMESTER : \_\_\_\_\_

COURSE : \_\_\_\_\_

REG. NO. : \_\_\_\_\_

# MANGALORE UNIVERSITY



## DEPARTMENT OF ELECTRONICS CERTIFICATE

This is to certify that Mr/Ms \_\_\_\_\_  
University Reg. No: \_\_\_\_\_ has successfully  
completed his/her course practical in CSCP 460: Design and Analysis  
of Algorithms Laboratory prescribed by The Mangalore University  
for the Second Semester in the laboratory under this department  
during 2022.

\_\_\_\_\_  
Lab in Charge

\_\_\_\_\_  
Head of the department

Date:

\_\_\_\_\_  
Examiners:

1.

Department Seal

2.

Exp. No.	<b>Laboratory Experiments Description</b>	Page No.
	CSCP 460: Design and Analysis of Algorithms Laboratory	
1.	Hash Table	4-9
2.	Greatest Common Divisor (GCD) using Euclidean Algorithm	10-11
3.	Generating Random Prime Number using Fermat's Little Test	12-15
4.	Multiplicative Inverse using Extended-Euclidean Algorithm	16-20
5.	Integer Factoring problem using Fermat's method	21-23
6.	Repeated Squaring Algorithm	24-26
7.	RSA ALGORITHM (Rivest -Shamir -Adleman Algorithm)	27-32
8.	Find The Median Using Divide and Conquer Method	33-35
9.	Depth First Search Algorithm	36-42
10.	Breadth-First Search Algorithm	43-46
11.	Dijkstra's Algorithm	47-53

# Hash Table

## Definition:

Hash table is a data structure that is used to store key-value items. The idea of hash table is to provide a direct access to its items. It calculates 'hash code' of key and uses it to store item, instead of key itself. Only one hash code per key.

## Theory:

In hash table, data is stored in array format, where each data value has its own unique index value. Access of data becomes very fast if we know the index of desired data.

Index	0 to n-1
Item	54, 26, 93, 17, 77, 31

- size = 1.3 times the key (large prime number)
- hash item = item % size

Assume the size is 11 then

Key	value
$54\%11$	10
$26\%11$	4
$93\%11$	5
$17\%11$	6
$77\%11$	0
$31\%11$	9
$44\%11$	0

**Hash collision:** When the hash function generates the same index for multiple keys, there will be a conflict (what value to be stored in that index). This is called a **hash collision**.

Different collision resolution techniques such as:

- Open Hashing (Separate chaining)
- Closed Hashing (Open Addressing)
  - Liner Probing
  - Quadratic probing

### 1. Separate Chaining-

- This technique creates a linked list to the slot for which collision occurs.
- The new key is then inserted in the linked list.
- These linked lists to the slots appear like chains.
- That is why, this technique is called as separate chaining.

### 2. Closed Hashing (Open Addressing)

This collision resolution technique requires a hash table with fixed and known size. During insertion, if a collision is encountered, alternative cells are tried until an empty bucket is found. These techniques require the size of the hash table to be supposedly larger than the number of objects to be stored

There are various methods to find these empty buckets:

#### i) **Linear probing:**

A hash table in which a collision is resolved by putting the item in the next empty place in the array following the occupied place.

**Algorithm:**

1. START
2.  $i=0, j=0, h=0,$   
 $k=5, a[k]=\{8,20,52,61,92\}, \text{size}=\text{int}(1.3*k), b[\text{size}];$
3. for  $i=0$  to  $i=6$   
 $b[i]=0$
4. while  $i < k$   
 $h=a[i]\% \text{size}$   
if  $b[h]==0$   
 $b[h]=a[i]$   
else  
while  $b[h] \neq 0$   
 $h=(h+1)\% \text{size}$   
 $b[h]=a[i]$   
 $i=i+1$
5. STOP

**Program:**

```
a=[8,20,52,61,92]
k=len(a)
size=int(1.3*k)
b=[]
for i in range(0,size):
    b.append(0)
i=0
j=0
print("empty table")
for i in b:
    print(j,"=",i)
    j+=1
i=0
```

```

while i<k:
    h=a[i]%size
    if b[h]==0:
        b[h]=a[i]
    else:
        while b[h]!=0:
            h=(h+1)%size
        b[h]=a[i]
    i=i+1
j=0
print("hash table")
for i in b:
    print(j,"=",i)
    j+=1

```

### Output:

```

empty table
0 = 0
1 = 0
2 = 0
3 = 0
4 = 0
5 = 0
hash table
0 = 0
1 = 61
2 = 8
3 = 20
4 = 52
5 = 92

```

### ii). Quadratic probing:

Quadratic probing operates by taking the original hash index and adding successive values of an arbitrary quadratic polynomial until an open slot is found.

### Algorithm:

1. START
2.  $i=0, j=0, h=0, k=5, a[k]=\{8,20,52,61,92\}, \text{size}=\text{int}(1.3*k), b[\text{size}]$
3. for  $i=0$  to  $i=6$   
     $b[i]=0$
4. while  $i < k$   
     $h = a[i] \% \text{size}$   
    if  $b[h] == 0$   
         $b[h] = a[i]$   
    else  
        while  $b[h] \neq 0$   
             $h = (h + i * 2) \% \text{size}$   
         $b[h] = a[i]$   
     $i = i + 1$
5. STOP

### Program:

```
a=[8,20,52,61,92]
k=len(a)
size=int(1.3*k)
b=[]
for i in range(0,size):
    b.append(0)
i=0
j=0
print("empty table:")
for i in b:
    print(j,'=',i)
    j+=1
i=0
while i<k:
    h=a[i]%size
    if b[h]==0:
        b[h]=a[i]
```



```

        else:
            while b[h]!=0:
                h=(h+i**2)%size
            b[h]=a[i]
        i=i+1
j=0
print("hash table:")
for i in b:
    print(j,"=",i)
    j=j+1

```

### Output:

```

empty table:
0 = 0
1 = 0
2 = 0
3 = 0
4 = 0
5 = 0
hash table:
0 = 92
1 = 61
2 = 8
3 = 20
4 = 52
5 = 0

```

# **Greatest Common Divisor (GCD) using Euclidean Algorithm**

**Definition:** To Calculate Greatest Common Divisor (GCD)

## **Theory:**

GCD is the abbreviation for Greatest Common Divisor which is a mathematical equation to find the largest number that can divide both the numbers given by the user. Sometimes this equation is also referred as the greatest common factor. For example, the greatest common factor for the numbers 20 and 15 is 5, since both these numbers can be divided by 5. This concept can easily be extended to a set of more than 2 numbers as well, where the GCD will be the number which divides all the numbers given by the user.

## **Using Euclidean Algorithm:**

The Euclid's algorithm (or Euclidean Algorithm) is a method for efficiently finding the greatest common divisor (GCD) of two numbers. The GCD of two integers X and Y is the largest number that divides both of X and Y (without leaving a remainder).

## **Pseudo Code of the Algorithm-**

1. Let a, b be the two numbers
2.  $a \bmod b = R$
3. Let  $a = b$  and  $b = R$
4. Repeat Steps 2 and 3 until  $a \bmod b$  is greater than 0
5.  $GCD = b$
6. Finish

## **Python code:**

```
# Method to compute gcd (Euclidean algorithm)
def computeGCD(x, y):
```

```

while(y):
    x, y = y, x % y
return x
a = int(input("Enter the value for a: "))
b = int(input("Enter the value for b: "))
# prints 12
print ("The gcd of ",a,"and",b," is : ",computeGCD(a,b))

```

### Output:

```

root@kali:~/Desktop# python3 euc.py
Enter the value for a: 49
Enter the value for b: 79
The gcd of 49 and 79 is : 1
root@kali:~/Desktop# python3 euc.py
Enter the value for a: 49
Enter the value for b: 21
The gcd of 49 and 21 is : 7

```

```

root@kali:~/Desktop# python3 euc.py
Enter the value for a: 15458
Enter the value for b: 48654
The gcd of 15458 and 48654 is : 2
root@kali:~/Desktop# python3 euc.py
Enter the value for a: 485648
Enter the value for b: 958456
The gcd of 485648 and 958456 is : 8

```

# Generating Random Prime Number using Fermat's Little Test

**Definition:** Given a positive integer number  $N$ , Task is to check whether given number prime or not.

**Theory:** Fermat's little theorem If  $p$  is prime, then for every  $1 \leq a < p$ ,

$$a^{N-1} \equiv 1 \pmod{N}$$

If  $N$  is prime, then  $a^{N-1} \equiv 1 \pmod{N}$ , for all  $a < N$ .

If  $N$  is not prime, then  $a^{N-1} \equiv 1 \pmod{N}$ , for at most half the values of  $a < N$ .

The algorithm of Figure 1.7 therefore has the following probabilistic behavior.

$$\Pr(\text{Algorithm 1.7 returns yes when } N \text{ is prime}) = 1$$

$$\Pr(\text{Algorithm 1.7 returns yes when } N \text{ is not prime}) \leq \frac{1}{2}$$

We can reduce this one-sided error by repeating the procedure many times, by randomly picking several values of  $a$  and testing them all (Figure 1.8).

$$\Pr(\text{Algorithm 1.8 returns yes when } N \text{ is not prime}) \leq \frac{1}{2^k}$$

This probability of error drops exponentially fast, and can be driven arbitrarily low by choosing  $k$  large enough. Testing  $k = 100$  values of  $a$  makes the probability of failure at most  $2^{-100}$ .

## Algorithm:

Input: Positive integer N

Output: YES / NO

Pick positive integers  $a_1, a_2, \dots, a_k < N$  at random

if  $a_i^{N-1} \equiv 1 \pmod{N}$  { for all  $i = 1, 2, \dots, k$ }

return YES

else:

return NO

- Pick a random n-bit number N.
- Run a primality test on N.
- If it passes the test, output N; else repeat the process.

## Program:

```
import random as rd

v=2

while(v!=1):

    m=1

    N = rd.randint( 2**2048, 2**2050 )

    while(m<=100):

        a=rd.randint(1,500)

        x = a

        y = N-1

        product = 1

        #####implement (x^y) mod N, modular exponentiation#####

        while y > 0:

            if( y & 1 == 1):#i.e, if last bit of y is 1

                product = (product*x)%N

            y = y>>1;

            x=(x*x)%N;

        v=product

        m=m+1

    if(product!=1):
```

break

#####

if(v==1):

print("prime number =",N)

## Output:

```
prime number= 333903416906577490535035460088176371321414038051237102344742913972917266
4771302063548885486030999055561238013254528402812761418953992556527777599840100746
4183936850799654392721555752572661286966501730585119405299274361640699473374472448
0497172517943664758117116256521570305072023381128019599217229870235776185921992028
2546002411177671581674372825266776079369227786755683752452519124826160546466885060
3164813509665486624267780033139081254747084237842263958388699677775166744990167435
1798760067467141255290707888915110977022996305129011566153014645396881471697093836
55292951334142642345993170580119174952366084369340724
```

```
prime number= 413495499872894763013691774962557077054813238752360223695120342651943513
1971505394851668716706070815666162854080628724436201208870451952364134483200521251
2112289654793823961892290005321557079977915748668225440482330775475839920362545538
2247160172149405776461313551054093431094684058120712976310873605313561613617619104
6402294591373834210848915683171608236565754855529408827428365048463016481600970943
4350375664735546674296451515480985387876638786447800286675871388794753746319964911
951653099199849104130422226817519431087773325476547596048399621489914786716727924
47628262188593372390483210464393081867136656063595160
```

```
prime number= 123681129864173618657469474782366350328881696451683162908323200399042163
2013085101336263527158137561084129119932629419493203381021211162524576720629093240
5295306878227640853031257850167544602754983319708165173538127519988681485622716229
4131548089965949272619141344747241781518860282117956776523335840702928863870882151
7041892085867025651973361320966057221929425884550367984100361467972274286055889435
7969772158152186766516208645946737240549944193816298824319889920104101301681460154
4104965819588575677804212737328927557857765010620826837823687797800341293319339108
505257632739114255074822877243319745847152255806151669
```

## Multiplicative Inverse using Extended-Euclidean Algorithm

**Definition:** To find the multiplicative inverse of two numbers

### Theory:

In math the multiplicative inverse of a number is another number which when multiplied by the original number gives 1 as the product. If 'N' is a natural number, the multiplicative inverse of 'N' will be  $\frac{1}{N}$  or  $N^{-1}$ .

So, the multiplicative inverse of a number is the reciprocal of that number.

For example, for  $N = 5$  the reciprocal will be  $\frac{1}{5}$ .

Now,

$$5 \times \frac{1}{5} = 1$$

So,  $\frac{1}{5}$  is the multiplicative inverse of 5.

**Extended Euclidean** algorithm also refers to a very similar algorithm for computing the polynomial greatest common divisor and the coefficients of Bézout's identity of two univariate polynomials.

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.

**Algorithm:**

Given a & b

**Output:**  $\gcd(a,b), x, y \in \mathbb{Z}$  such that

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

**Initialization:**  $x_0=1, x_1=0, y_0=0, y_1=1, \text{sign}=1$

1. While  $b \neq 0$
2.  $r = a \bmod b$  //remainder of the
3.  $q = a / b$
4.  $a = b$
5.  $b = r$
6.  $xx = x_1$
7.  $yy = y_1$
8.  $x_1 = q * y_1 + x_0$
9.  $y_1 = q * y_1 + y_0$
10.  $x_0 = xx$
11.  $y_0 = yy$
12.  $\text{Sign} = -\text{sign}$
13. End while
14.  $x = \text{sign} * x_0$
15.  $y = \text{sign} * y_0$
16.  $\gcd = a$
17. Return  $\gcd(x,y)$

**Program code:**

```
import random as r
import math
import sympy as s
```

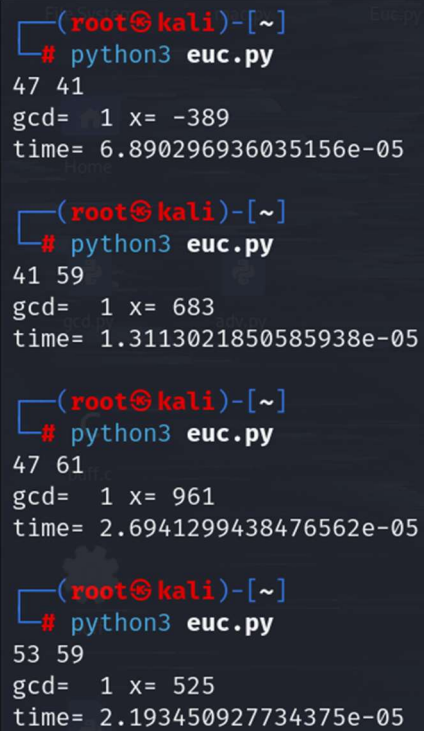
```

m=35
n=65
p=s.randprime(m,n)
q=s.randprime(m,n)
while p==q:
    q=s.randprime(m,n)
print(p, q)
n=p*q
phi=(p-1)*(q-1)
e=r.randint(1,phi)
while math.gcd(e,phi)!=1:
    e=r.randint(1,phi)
a=e
b=phi
xo=1
x1=0
yo=0
y1=1
sign=1
import time
start=time.time()
while(b!=0):
    r=a%b
    q=a//b
    a=b
    b=r
    xx=x1
    yy=y1
    x1=(q*x1)+xo
    y1=(q*y1)+yo
    xo=xx
    yo=yy
    sign=-sign
x=sign*xo
y=-sign*yo

```

```
gcd=a
print("gcd= ",gcd,"x=",x)
print("time= %s" % ((time.time())-start))
```

## **OUTPUT:**



```
(root@kali)~# python3 euc.py
47 41
gcd= 1 x= -389
time= 6.890296936035156e-05

(root@kali)~# python3 euc.py
41 59
gcd= 1 x= 683
time= 1.3113021850585938e-05

(root@kali)~# python3 euc.py
47 61
gcd= 1 x= 961
time= 2.6941299438476562e-05

(root@kali)~# python3 euc.py
53 59
gcd= 1 x= 525
time= 2.193450927734375e-05
```

```
(root@kali)-[~]  
# python3 euc.py  
256994747487543341 464937119869018967  
gcd= 1 x= -42674150739939787391390329088298443  
time= 7.581710815429688e-05
```

```
(root@kali)-[~]  
# python3 euc.py  
652768798767342937 415764404745952937  
gcd= 1 x= 24102757793631538980997794861402653  
time= 5.1021575927734375e-05
```

```
(root@kali)-[~]  
# python3 euc.py  
127140049960100591 38276765078486131  
gcd= 1 x= 1624178383501082021641253895522137  
time= 8.296966552734375e-05
```

```
(root@kali)-[~]  
# python3 euc.py  
323683085621291183 174864319823697613  
gcd= 1 x= 555782749471799848248053448206357  
time= 7.43865966796875e-05
```

```
(root@kali)-[~]  
# python3 euc.py  
29694740965667171 262395572918619419  
gcd= 1 x= 3272686860444478562943568905347507  
time= 9.036064147949219e-05
```

# Integer Factoring problem using Fermat's method

**Definition:** Prime Factorization (or integer factorization) is a commonly used mathematical problem often used to **secure public-key encryption systems**. A common practice is to use very large semi-primes (that is, the result of the multiplication of two prime numbers) as the number securing the encryption.

## Theory:

In number theory, integer factorization is the decomposition of a composite number into a product of smaller integers. If these factors are further restricted to prime numbers, the process is called prime factorization.

When the numbers are sufficiently large, no efficient non-quantum integer factorization algorithm is known. However, it has not been proven that such an algorithm does not exist. The presumed difficulty of this problem is important for the algorithms used in cryptography such as RSA public-key encryption and the RSA digital signature. Many areas of mathematics and computer science have been brought to bear on the problem, including elliptic curves, algebraic number theory, and quantum computing.

## Algorithm:

1. Start
2. Read  $n$
3.  $a = \text{sqrt}(n)$
4.  $i = \text{round}(a) + 1$
5.  $b = (i * i) - N$
6.  $c = \text{sqrt}(b)$
7. if  $c = \text{int value}$ 
  - $\text{print}(N = i * i - c * c)$
  - $p = (i + c)$
  - $q = (i - c)$

```

    print(p)
    print(q)
else:
    i = i + 1
    go to step (5)
8. Stop

```

### **Program code:**

```
import math as m
```

```

def fermat(n):
    a =m.sqrt(n)
    i =round(a)+1
    b = i*i - n
    c = m.sqrt(b)
    while c*c != b:
        i = i + 1
        b = i*i - n
        c =m.sqrt(b)
    p=i+c
    q=i-c

    print('p=',p)
    print('q=',q)
    print('pq=',p*q)
    return p, q

```

```

n=195559655854585
fermat(n)

```

## OUTPUT:

```
In [94]: runfile('C:/Users/Asus/Desktop/if.py', wdir='C:/Users/Asus/Desktop')
p= 15.69041575982343
q= 6.30958424017657
pq= 99.0

In [95]: runfile('C:/Users/Asus/Desktop/if.py', wdir='C:/Users/Asus/Desktop')
p= 55.78404875209022
q= 28.21595124790978
pq= 1574.0

In [96]: runfile('C:/Users/Asus/Desktop/if.py', wdir='C:/Users/Asus/Desktop')
p= 3273.829435256319
q= 3010.170564743681
pq= 9854785.0

In [97]: runfile('C:/Users/Asus/Desktop/if.py', wdir='C:/Users/Asus/Desktop')
p= 46.48999599679679
q= 21.510004003203203
pq= 999.9999999999999

In [98]: runfile('C:/Users/Asus/Desktop/if.py', wdir='C:/Users/Asus/Desktop')
p= 13990567.124582168
q= 13977964.875417832
pq= 195559655854585.0
```

## Repeated Squaring Algorithm

**Definition:** Given 2 positive integer a and b, the task is to find  $a^b$  using Repeated Squaring Algorithm.

### Theory:

Repeated squaring, or repeated doubling is an algorithm that computes integer powers of a number quickly. The general problem is to compute for an arbitrary integer y. The naive method, doing y multiplications of x, is very slow. It can be sped up by repeatedly squaring x until the current power of x exceeds y, and then collecting the "useful" powers.

Repeated Squaring Method takes the advantage of the fact that  $A^x \times A^y = A^{x+y}$ .

Any number can be written as the sum of powers of 2. We Just need convert the number to Binary Number System. Now for each position i for which binary number has 1 in it, add  $2^i$  to the sum.

Consider the number  $7^{19}$ , here  $(19)_{10}$  can be written as binary number i.e.

$$(10011)_2 = (2^4 + 2^1 + 2^0)_{10} = (16 + 2 + 1)_{10}$$

Therefore, in the equation  $a^b$ , we can decompose 'b' to the sum of powers of 2.

The main concept for repeated squaring method is decomposing value 'b' to binary, and then for each position i (we start from 0 and loop till the highest position at binary form of 'b') for which binary of 'b' has 1 in  $i^{\text{th}}$  position, we multiply  $a^{2^i}$  to result. The time complexity of repeated squaring algorithm is  $(m^2 n)$  or  $m^3$ .

Consider the number  $3^{13}$ . The binary representation of 13 is  $(1101)_2$ , so  $8 + 4 + 1 = 13$

$3^1$	3
$3^2$	$9 = 3 * 3$
$3^4$	$81 = 9 * 9$
$3^8$	$6561 = 81 * 81$



Here result is the square of the previous result, and hence can be computed in one multiplication.

$$3^1 * 3^4 * 3^8 = 3^{13} = 1594323$$

### **Algorithm:**

Step 1:

Start

Step2:

a=2, b=12, p=1

Step 3:

if(b==0): return 1 {go to step 9}

step 4:

lsb=b%2

step 5:

b=b//2

step 6:

if (lsb==1) {p=p\*a}

Step 7:

A=a\*a {go to step 3}

Step 8:

Print (p)

Step 9:

Stop

### **Program:**

a=2844

n=193

p=1

while (a!=0):

lsb=a%2

a=a>>1

if (lsb==1):

p=p\*n

```
n=n*n
print(p)
```

**Result:**

```
(kali@kali) ~
$ python3 rep5.py
1333480957166071908952621926137879369050810754460893902739202288309018597085983585330395700353304455544612016754202966152500170925530046699671109964647713665051879672027930299600234433168357
5334407030638168489036151975079364514022924519033047809591150170286052772377481481841994347945226502080195472569376316672774811202346824137831677292154787907233945375322695891869783277679712
2479124093481494796547063002506768463365270894538639359444710437197432861848947437680580195840005419905462836442676361444519062226190061756409873992079916452350536747702502994126537260385158
697167814294098448914704474624825764596793878113035701356561059331158117416123423189484556941503206085853563738048502496608481844848088963600313896894400070409535770477829600958889899876692753
8426648832259036016430101154413282821993527874383575523166668097274851627005853213647718005540803247367769393850635446387989851782282715670871985720491630114682521758847251388846550788539
035103941452113782971721816289619773197186890917548428356503220692125706620121323851503794529462801148016408697538633235025489087049171838650462239929689437581134155096802255404046629180187
38466630655378109144731149633662164238319481955907678245502249866418312424439166814049727766163469716766215349585579329003867179591551253377314565749494207004219979342356865315103090851
309357549169409857213894383412506728907937095661176490490415684619831574111382675112738242834725892028316167398047605777700282663726068763172686976928751071398530452843342621551579972010626
659021340955936728677726910911722139897942502788232902583770990437586970301149224970475088739336280928105384374796807170956918673541431402437734683897213179315106054579902937899990218478153
13840414942933820869406738423329743029448003396185196783114621089412209008377628514609486204724342945205508396666378570696946600918817633549901064759365878245895121264316714605498857491060613
595303732667905814095525773639325648734293241332474746166423828685120550910860875363643050273398805086877990814956889139584541102966098004241651494717383750333240487772197314781673159795766
768384763949116044393277969485425971160195090303080177443981495369758068561745075772888860611515993617874111192999767914710294210834858369292322748781607465604133502954336520747038042971057
8526841614868352528298205108507960315967378364771913755992326643004481359949536178830117783223154240221298027461561860524020180815481285370407285358342817609833403658047964314705659703450300
0174817741361539566083814746727365382591459523067239232112257159085158766631169414571595421200956922105184696875113896472635278445181158835123840759183999917860226746339205779076951075399172
435292719091329661437841535894316844542222097629939766370532723320103496605468095779901340855863499378295041447753261304081925493544201994696060521604296548153583501973848485407435122127085
44062404201909230074011835299191076333596320265611341947355534067258313298581404421086842708093739284853346356153027576633255392403542472655899647820184697742457356618322331460026039022493
651949252984595341410783028408927757702896617814067392333860567670389071623222071414487757386985377320160561094235481692279188322581530253808751953218259950172693862585130578188409832079230
413638175860549095062139351046449726768435556665364436811926284708005401497840655221143230725701876746948300150169313018516860124320063863564012306728960024165442302230872398120607570833266
8707361680527978285115761120643572190028946158520755905285407569818665422096712911515573639975138679753296840908786322689708319725132004209007966754657223951513202069295468650112111787297414
550876530113558707103809644611519254292547644204286720400159311994503275743196260490937132828710834477632418265549715518143871523305159447746288238328369547803239406666750422437617404970791
2983097527156173584921832386534656714445728342852346790271424206968305063609052245200918600263836004271181751052707113924042434444865789047405249707495056215215299522690963539513872398546252
1982768068809723293241271322211622368190717621214983910771692349995249050529835677012634313201788920388416616701623894782343916968753478363994038106324821093751766114324535287564574428553832
7830398302166568519065958205954652523327707063108086448794007461235285657255006590015956205707464513120297950863440606814394412509456520188162715932607620064553084088058026066420674753318143
260590261881616584322855851097438430581040745435723220661705993171163787465729989110565095259163479241110287737875495584654951870646291313822693964819474736588056654591065710180210786188784
04683114728459935398177975204155642328304935150039468883706868098572160385595490231443579500600052839167559391647880494941960821700265824368980616342863528205700420317901634284994815134969421
0015569177570826453302345537402004454841695870187934680077926419552466951538093768602912375770930474798841335585049410077856739520665426907776794251124665730335810733170631584512271183968603
2925999081637107592024528384098715384787925830412228570859462028051571173534726843418438699265717766888314163913929677126927292940966237216536116681465103457219640674532309865056589534188
3632553422861772232389405358714011654012069081813591861262359067976520817767301869443509065134501649240835017037882661478084259799424867541167518341215658256335070675985658379884740939440852
0770542417525461467763685615894468040317403370585630263716930140365252449661162652059562534902557303180700324290751074627090508481394724166218633087659334977367828420493269108836916286835
6318544063886723961746642620575948356597510116122539677399855846809138143797964680023863497057903056800771481659818079119136025941103981425951887706336041412329151144045651530048115519242
5681624942557308589468342858571933566960171822979146840791071165837453004473371079914766334251364516369396068227849475127867889338958300882734812662835392381996767295662049029560524527824604
0278597356341094712553284357464082368411079486257214392233990898882732485082271200256403915005895485343040216407158038351494812841454679590647704294983799477844083285699378710283623116773435
702382795883448683909583315258535421066032573939907858302491183309270088976475238328161053028795613587021240849368618388548547395336620700750813573285214718443530844432412281593627924008314
414513334653593501077700006708437459006113222359719124455611406339403565481721745922830436283570818129167543884015623772771199720880812630729035020610412660768901729340342591076220755146974036
34610031727146820934751353548562829680001
```

# RSA ALGORITHM (Rivest -Shamir -Adleman Algorithm)

**Definition:** Implementing RSA algorithm using Fermat's little test, Fermat's repeated squaring algorithm, Euclid's algorithm for GCD and Extended Euclid's algorithm, to generate encryption and decryption keys.

## Theory:

RSA algorithm is asymmetric cryptography algorithm. Asymmetric actually means that it works on two different keys i.e., Public Key and Private Key. The idea of RSA is based on the fact that it is difficult to factorize a large integer. The public key consists of two numbers where one number is multiplication of two large prime numbers. And private key is also derived from the same two prime numbers. So, if somebody can factorize the large number, the private key is compromised. Therefore, encryption strength totally lies on the key size and if we double or triple the key size, the strength of encryption increases exponentially. RSA keys can be typically 1024 or 2048 bits long.

The RSA algorithm holds the following features –

- RSA algorithm is a popular exponentiation in a finite field over integers including prime numbers.
- The integers used by this method are sufficiently large making it difficult to solve.
- There are two sets of keys in this algorithm: private key and public key.

## Algorithm:

1. Choose two different random prime numbers  $p$  and  $q$
2. Calculate  $n = p * q$
3. Calculate  $t = (p - 1) * (q - 1)$

4. Choose an integer  $e$  such that  $1 < e < t$  and  $\gcd(e, t) = 1$
5. Choose an integer  $d$  such that  $d = (1 + k.t)/e$  for some integer  $k$

Once the parameters  $n$ ,  $e$ , and  $d$  are calculated, the RSA key pair is defined as:

- Encryption key —  $(n, e)$
- Decryption key —  $(n, d)$

For a given public key  $(n, e)$ , its corresponding private key  $(n, d)$ , and a given message  $m$ :

- Encryption

ciphertext  $c = m^e \bmod n$

- Decryption

plaintext  $m = c^d \bmod n$

Where,  $n$  — Modulus for the public/private keys

$e$  — Public key exponent

$d$  — Private key exponent

### **Program:**

```
import random as r

def gcd_alg(x,y):
    while(y):
```

```

        x,y=y,x%y

    return x

#####

def randprime():

    v=2

    while (v!=1):

        m=1

        N = r.randint(2*2048,2*2050)

        while(m<=100):

            a=r.randint(1,500)

            x = a

            y = N-1

            product = 1

#####

            #Implement (x^y) mod N, modular exponentiation

            while y > 0:

                if (y & 1 == 1):#i.e, if last bit of y is 1

                    product = (product*x)%N

                y = y>>1;

            x=(x*x) %N

```

```

        v=product

        m=m+1

    if (product!=1):

        break

#####

    if(v==1):

        return N

#####

def ExEuc_alg(a,b):

    x0=1

    x1=0

    y0=0

    y1=1

    sign=1

    while (b!=0):

        r=a%b

        q=a//b

        a=b

        b=r

        xx=x1

```

```

        yy=y1

        x1=(q*x1)+x0

        y1=(q*y1)+y0

        x0=xx

        y0=yy

        sign=-sign

    x=(sign*x0)

    y=(-sign*y0)

    return x

#####

p =randprime()

q = randprime()

while (p==q):

    q=randprime()

n = p*q

phi = (p-1) * (q-1)

e=r.randint(1,phi)

while(gcd_alg(e,phi)!=1):

    e=r.randint(1,phi)

a=e

```



b=phi

d=ExEuc\_alg(a,b)

print("(e,n)=", "("+e+","+n+")")

print("(d,n)=", "("+d+","+n+")")

**Result:**

```
(darshanreddych123@darshan) ~  
$ python3 rsa-1.py  
e,n)= ( 29228369319643818030920859393712667620335249300857201124935303074380187340346371679448775320662237838356240378442321141664780112906545182334008335209  
824943605619812164192033723359056735712627569253252480627937966786877569481502236414650099899406484771916592545312268116502813332404774375400391101196332691  
3818765726109175401115225492782145455409198978349368716263948511326690228303607774161903171543338147992002606735214284403408190639529350420985840053459465658  
0794765025328886247875698135433119645205635676260450002975911062118346333087835929739259228633867049457215510944352390672454765420474376900313542347495910626  
7492853422891688486293554481234548751539391551875874377666744403570269397041600484120444758707524430427291781989705048434195797020226694998186861812193732468  
4435750411721877268470361198606626461280189267962825662441191274284620556952369731690388354157612197609043165374937921478896387247922027509871042668909867731  
966247213327771781471503304028725501981074957082008671470214711315507333825816751640627960292695808845087977756067500452413086340740476152050995036338672571  
8966523214651533396410331212030796714369920946319824580442286015584000479598854335705327829132041818141207336715041235561056492266405667 , 789055854800516822  
3210545495276116933967538922200426707779502408271983525262964333870058444183137747382248217218607495707007027015088467123944590370402629999740071014013728350  
8115308741696952704407110104988460541979415498373150193044436984384533588449553861134117658349116806852267132995263273744356636249265292470295628744063261674  
6180918021595447209220167367234794743675317274457535365846164069955789282969008470722549236469941046308785162701490578281213959921659915488684272499372367078  
1956881151363015086214676639765340349997648032725004897886342509656627670422531736024638683691215509150270398473234996960668907616244605405201535172792280829  
7872530377157966520877816301514374680395579456661018737174061947133070051016914988987566693835301618281457949177176093611537720418475059971880716519198622726  
8500492116953665511902328534272133642940523309840359189117463478304626417616216099850319698458061323452376057265995849421328921561353615659278449453503786067  
5976697165197903296453875208946730373783931920315156833405301451373938692999424066537000615275834272564892490579490686592504792920448472586194375230703242605  
867054377015218243439758229132866516596246658712374845393054596050950003957132376917913426825382950612692736 )  
d,n)= ( 22943123943252944487939479317280127851135003618939628032584693841913528148744788046264960784261349007940514564976685560955853258058423334360994694224  
7999240063219065955489617812322175660125773300490599250128424917163922622359359574131052031414196703318514466180949298961980560769324346182185619729954311724  
5230333420954895820403772729415171548034793651453122986943964484296572926390343435130826726433511880719699545780574288802433865414235219848143506702475483374  
6517829810822376798797967759387045046425867838907848365843152616454109881210803087955133500046976062160306061119512153165830689493159214453946244938766373189  
1133763701221375865051758791773261343543158157297862467130934148827106528019152099125915089127668821386195626550899007096301289460445356819037452561059335439  
5089108304679529568677169374457788815099911611223533352980585747993134781038515222761487115895869634801887039857710994512164806255250234368671067848445606609  
0691198674940078650643915830938503271267641136212550513945221647615664889023663260305842182688644434928645411769585181460145289271089070827922201789125958689  
11367100968286756714614200356709584929570383656331977018453132983849998930666980335991142235981941885839664408982469479908919436195343 , 7890558548005168229  
2105454952761169339675389222004267077795024082719835252629643338700584441831377473822482172186074957070070270150884671239445903704026299997400710140137283508  
1153087416969527044071101049884605419794154983731501930444369843845335884495538611341176583491168068522671329952632737443566362492652924702956287440632616744  
180918021595447209220167367234794743675317274457535365846164069955789282969008470722549236469941046308785162701490578281213959921659915488684272499372367078  
9568811513630150862146766397653403499976480327250048978863425096566276704225317360246386836912155091502703984732349969606689076162446054052015351727922808290  
8725303771579665208778163015143746803955794566610187371740619471330700510169149889875666938353016182814579491771760936115377204184750599718807165191986227263  
5004921169536655119023285342721336429405233098403591891174634783046264176162160998503196984580613234523760572659958494213289215613536156592784494535037860676  
9766971651979032964538752089467303737839319203151568334053014513739386929994240665370006152758342725648924905794906865925047929204484725861943752307032426050  
670543770152182434439758229132866516596246658712374845393054596050950003957132376917913426825382950612692736 )
```



## Find The Median Using Divide And Conquer Method

**Definition:** Given a list of elements S, Task is to find the median, in the given list of elements.

**Theory:** The median of a list of numbers is its 50th percentile: half the numbers are bigger than it, and half are smaller. For instance, the median of [45, 1, 10, 30, 25] is 25, since this is the middle element when the numbers are arranged in order. If the list has even length, there are two choices for what the middle element could be, in which case we pick the smaller of the two.

### Algorithm:

1. Divide the list into sublists if size n, assume 5.
2. Initialize an empty array M to store medians we obtain from smaller sublists.
3. Loop through the whole list in sizes of 5, assuming our list is divisible by 5.
4. For  $\frac{n}{5}$  sublists, use select brute-force subroutine to select a median m, which is in the 3rd rank out of 5 elements.
5. Append medians obtained from the sublists to the array M.
6. Use quickSelect subroutine to find the true median from array M, The median obtained is the viable pivot.
7. Terminate the algorithm once the base case is hit, that is, when the sublist becomes small enough. Use Select brute-force subroutine to find the median.

## Program:

```
def median_of_medians(arr):
    if arr is None or len(arr) == 0:
        return None
    return select_pivot(arr, len(arr) // 2)

def select_pivot(arr, k):
    chunks = [arr[i : i+5] for i in range(0, len(arr), 5)]
    sorted_chunks = [sorted(chunk) for chunk in chunks]
    medians = [chunk[len(chunk) // 2] for chunk in sorted_chunks]
    if len(medians) <= 5:
        pivot = sorted(medians)[len(medians) // 2]
    else:
        pivot = select_pivot(len(medians) // 2)
    p = partition(arr, pivot)
    if k == p:
        return pivot
    if k < p:
        return select_pivot(arr[0:p], k)
    else:
        return select_pivot(arr[p+1:len(arr)], k - p - 1)

def partition(arr, pivot):
    left = 0
    right = len(arr) - 1
```

```

i = 0
while i <= right:
    if arr[i] == pivot:
        i += 1
    elif arr[i] < pivot:
        arr[left], arr[i] = arr[i], arr[left]
        left += 1
        i += 1
    else:
        arr[right], arr[i] = arr[i], arr[right]
        right -= 1
return left
arr = [25, 42, 23, 36, 12, 47, 89, 96, 85]
pivot = median_of_medians(arr)
print ("median of the list=",arr,"is",pivot)

```

### Output:

median of the list= [25, 42, 23, 36, 12, 47, 89, 96, 85] is 42

median of the list= [354, 456, 25, 123, 475, 598, 896, 523] is 475

# Depth First Search

**Definition:** Given a graph  $G(V, E)$  (directed or undirected), Task is to find reachability from source  $S$  to destination  $D$ .

**Theory:** Depth-first search is a surprisingly versatile linear-time procedure that reveals a wealth of

information about a graph. The most basic question it addresses is,

What parts of the graph are reachable from a given vertex?

**Undirected graph:** A graph with unordered pairs of vertices, so the edge from vertex  $a$  to  $b$  is identical to the edge from vertex  $b$  to  $a$ .

**Directed graph:** A directed graph is a graph that is made up of a set of vertices connected by edges, where the edges have a direction associated with them.

## Algorithm:

1. Start

2.  $N$  = Total number of vertices;  $A=1, B=2, C=3, \dots, Z=26$ ;  $E$  = Edges entries;  $AL = []$  #adjacency list;  $Stack = []$  #empty stack;  $source = ""$ ;  $destination = ""$ ;

3. Append edges entries to  $AL$

4. Push source to stack

5. Mark source as visited in  $AL$

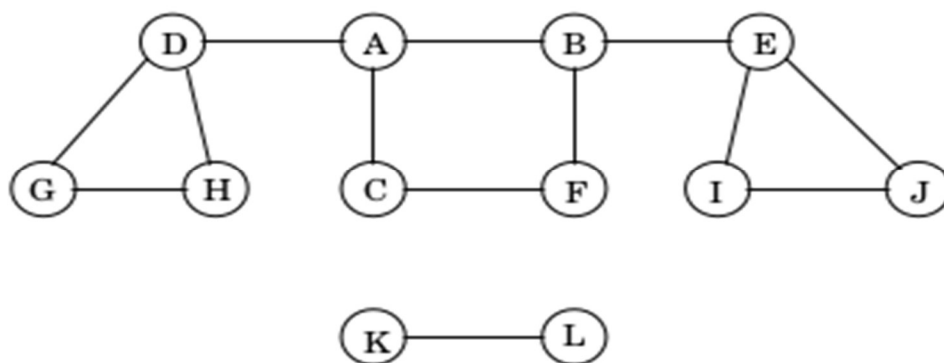
6. if( $source == destination$ ) {print "reachable"; Goto 14;}

7.  $prevsource = source$

8. Check for available vertex from the source vertex which is not yet visited.
9. If found make source=available\_vertex
10. If(source==prevsource){ Pop the source vertex and prevsource from the stack if stack length greater than 1 }
11. Push source vertex to stack
12. If(length of stack is 1){print “not reachable”; Goto 14;}
13. Goto 5
14. Stop

**Python code:**

**Undirected graph:**



**#Python code to find reachability from source to destination in undirected graphs.**

```
n = 12;
```

```
# A=1, B=2, C=3, D=4, E=5, F=6, G=7, H=8, I = 9, J=10, K = 11, L = 1
```

```
E=[[1,2],[1,3],[1,4],[2,1],[2,5],[2,6],[3,1],[3,6],[4,1],[4,8],[4,7],[5,9],[5,10],
,[5,2],[6,3],[6,2],[7,8],[7,4],[8,4],[8,7],[9,10],[9,5],[10,9],[10,5],[11,12],[1
2,11]]
```

```
AL = []
```

```
for i in range(n):
```

```
    AL.append([n+1])
```

```
for e in E:
```

```
    AL[e[0]-1].append(e[1])
```

```
# perform DFS at A (vertex 1)
```

```
# n+1 not visited
```

```
# n+2 visited
```

```
s = ""# source vertex
```

```
so = s
```

```
d = "" # destination vertex
```

```
stack = []
```

```
stack.append(s)
```

```
while True:
```

```
    AL[s-1][0] = (n+2) # mark v=1 for s
```

```
    if(s == d):
```

```
        print("stack=",stack, "\t", so, " is reachable by", d)
```

```
        break
```

```
    ss = s; # check the next available vertex for visiting
```

```
    for i in range(1, len(AL[s-1])):
```

```
        if(AL[AL[s-1][i] - 1][0] == (n+1)):
```

```

        s = AL[s-1][i]
        break
    if(s == ss): # if s is completely explored
        if(len(stack) > 1):
            s = stack.pop()
            s = stack.pop()
        stack.append(s) # push the s which
        if(stack[0] == stack[-1] and len(stack) > 1):
            print("Not Reachable")
            break

```

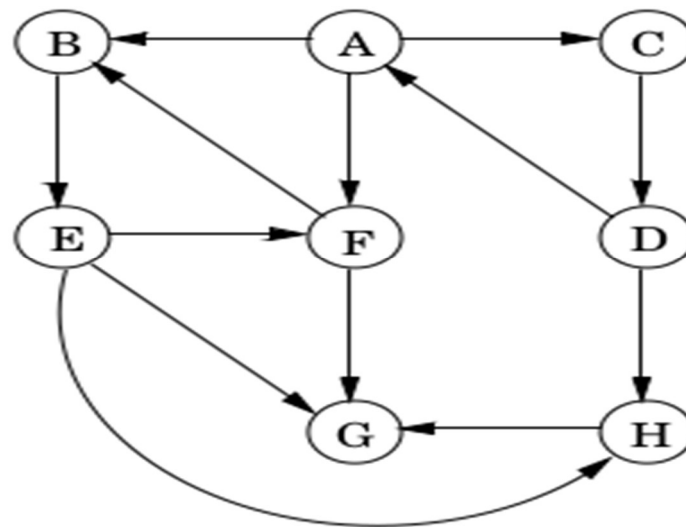
## Output:

```

18
└─(darshanreddych123@ darshan)-[~]
└─$ python dfs.py
stack= [1, 2, 5, 9, 10]          1 is reachable by 10
22
└─(darshanreddych123@ darshan)-[~]
└─$ python dfs.py
stack= [1, 2, 5]          1 is reachable by 5
26
└─(darshanreddych123@ darshan)-[~]
└─$ python dfs.py
Not Reachable
30
└─(darshanreddych123@ darshan)-[~]
└─$ python dfs.py
Not Reachable
34
└─(darshanreddych123@ darshan)-[~]
└─$ python dfs.py
stack= [11, 12]          11 is reachable by 12
38
└─(darshanreddych123@ darshan)-[~]
└─$ python dfs.py
stack= [12, 11]          12 is reachable by 11
42

```

**Directed graph:**



**#Python code to find reachability from source to destination in directed graphs.**

```
n = 12;
# A=1, B=2, C=3, D=4, E=5, F=6, G=7, H=8, I = 9, J=10, K = 11, L = 12
E=[[1,2],[1,6],[1,3],[2,5],[3,4],[4,1],[4,8],[5,6],[5,7],[5,8],[6,2],[6,7],[8,7]]
AL = []
for i in range(n):
    AL.append([n+1])
for e in E:
    AL[e[0]-1].append(e[1])

# Perform DFS at A (vertex 1)
# n+1 not visited
# n+2 visited
```



```

s = "" # source vertex
so = s
d = "" # destination vertex
stack = []
stack.append(s)
while True:
    AL[s-1][0] = (n+2) # mark v=1 for s
    if(s == d):
        print("stack=",stack, "\t", so, " is reachable by", d)
        break
    ss = s; # check the next available vertex for visiting
    for i in range(1, len(AL[s-1])):
        if(AL[AL[s-1][i] - 1][0] == (n+1)):
            s = AL[s-1][i]
            break
    if(s == ss): # if s is completely explored
        if(len(stack) > 1):
            s = stack.pop()
            s = stack.pop()
        stack.append(s) # push the s which
    if(stack[0] == stack[-1] and len(stack) > 1):
        print(d,"is Not Reachable to ",ss)
        break

```

## Output:

```
(darshanreddych123@ darshan)-[~]
$ python dfs.py
stack= [1, 2, 5, 6]      1 is reachable by 6

(darshanreddych123@ darshan)-[~]
$ python dfs.py
stack= [1, 2, 5]        1 is reachable by 5

(darshanreddych123@ darshan)-[~]
$ python dfs.py
stack= [1, 3, 4]        1 is reachable by 4

(darshanreddych123@ darshan)-[~]
$ python dfs.py
stack= [1, 3]          1 is reachable by 3

(darshanreddych123@ darshan)-[~]
$ python dfs.py
stack= [1, 2]          1 is reachable by 2

(darshanreddych123@ darshan)-[~]
$ python dfs.py
stack= [1, 2, 5, 6, 7]  1 is reachable by 7

(darshanreddych123@ darshan)-[~]
$ python dfs.py
Not Reachable

(darshanreddych123@ darshan)-[~]
$ python dfs.py
7 is Not Reachable by 1

(darshanreddych123@ darshan)-[~]
$ python dfs.py
8 is Not Reachable to 7
```

# Breadth-First Search

**Definition:** Given a graph  $G(V, E)$  (directed or undirected), Task is to find the shortest distance from source  $S$  to destination  $D$ .

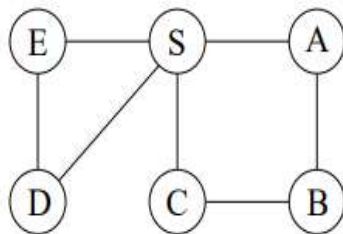
## Theory:

Breadth-first search finds shortest paths in any graph whose edges have unit length. We adapt it to a more general graph  $G = (V, E)$  whose edge lengths  $l_e$  are positive integers

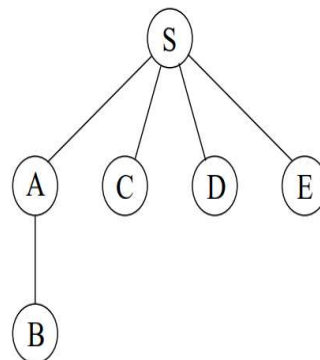
**Undirected graph:** A graph with unordered pairs of vertices, so the edge from vertex  $a$  to  $b$  is identical to the edge from vertex  $b$  to  $a$ .

**Directed graph:** A directed graph is a graph that is made up of a set of vertices connected by edges, where the edges have a direction associated with them.

(a)



Order of visitation	Queue contents after processing node
	[S]
S	[A C D E]
A	[C D E B]
C	[D E B]
D	[E B]
E	[B]
B	[]



### Procedure bfs( $G, s$ )

Input:                Graph  $G = (V, E)$ , directed or undirected; vertex  $s \in V$

Output:     For all vertices  $u$  reachable from  $s$ ,  $\text{dist}(u)$  is set to the distance from  $s$  to  $u$ .

for all  $u \in V$  :

$\text{dist}(u) = \infty$

$\text{dist}(s) = 0$

$Q = [s]$  (queue containing just  $s$ )

while  $Q$  is not empty:

$u = \text{eject}(Q)$

    for all edges  $(u, v) \in E$ :

        if  $\text{dist}(v) = \infty$ :

$\text{inject}(Q, v)$

$\text{dist}(v) = \text{dist}(u) + 1$

Initially the queue  $Q$  consists only of  $s$ , the one node at distance 0.

And for each subsequent distance  $d = 1, 2, 3, \dots$ , there is a point in time at which  $Q$  contains all the nodes at distance  $d$  and nothing else.

As these nodes are processed (ejected off the front of the queue), their as-yet-unseen neighbours are injected into the end of the queue.

### **Algorithm:**

1. Pick any node, visit the adjacent unvisited vertex, mark it as visited, display it, and insert it in a queue.
2. If there are no remaining adjacent vertices left, remove the first vertex from the queue.
3. Repeat step 1 and step 2 until the queue is empty or the desired node is found

## Program:

n=7

```
E = [  
    [1,2],[1,6],  
    [2,3],[2,1],  
    [3,2],[3,6],[3,7],  
    [4,5],[4,6],  
    [5,4],[5,6],  
    [6,1],[6,3],[6,4],[6,5]  
]
```

```
AL = []
```

```
for i in range(n):  
    AL.append([n+1])
```

```
for e in E:  
    AL[ e[0]-1 ].append(e[1])
```

```
for e in AL:  
    print(e)
```

```
s = 6
```

```
Q = [s]
```

```
AL[s-1][0] = 0
```

```
while ( Q != [] ):
```

```
    u = Q.pop(0)
```

```
    print(u, AL[u-1][0])
```

```
    for v in AL[u-1][1:]:  
        if( AL[ v-1 ][0] == n+1 ):  
            Q.append(v)
```

$$AL[v-1][0] = AL[u-1][0] + 1$$

**Output:**

```
[8, 2, 6]
[8, 3, 1]
[8, 2, 6, 7]
[8, 5, 6]
[8, 4, 6]
[8, 1, 3, 4, 5]
[8]
6 0
1 1
3 1
4 1
5 1
2 2
7 2
```

# Dijkstra's Algorithm

**Definition:** Given a graph  $G(V, E)$  (directed or undirected) with positive edge lengths  $\{l_e: e \in E\}$ ; vertex  $s \in V$ , task is to find the shortest path between two vertices in graph.

## Theory:

Breadth-first search finds shortest paths in any graph whose edges have unit length. Can we

adapt it to a more general graph  $G = (V, E)$  whose edge lengths  $l_e$  are positive integers?

Dijkstra's algorithm. The alarm clock algorithm computes distances in any graph with positive integral edge lengths. It is almost ready for use, except that we need to somehow implement the system of alarms. The right data structure for this job is a priority queue (usually implemented via a heap), which maintains a set of elements (nodes) with associated numeric key values (alarm times) and supports the following operations:

Insert. Add a new element to the set.

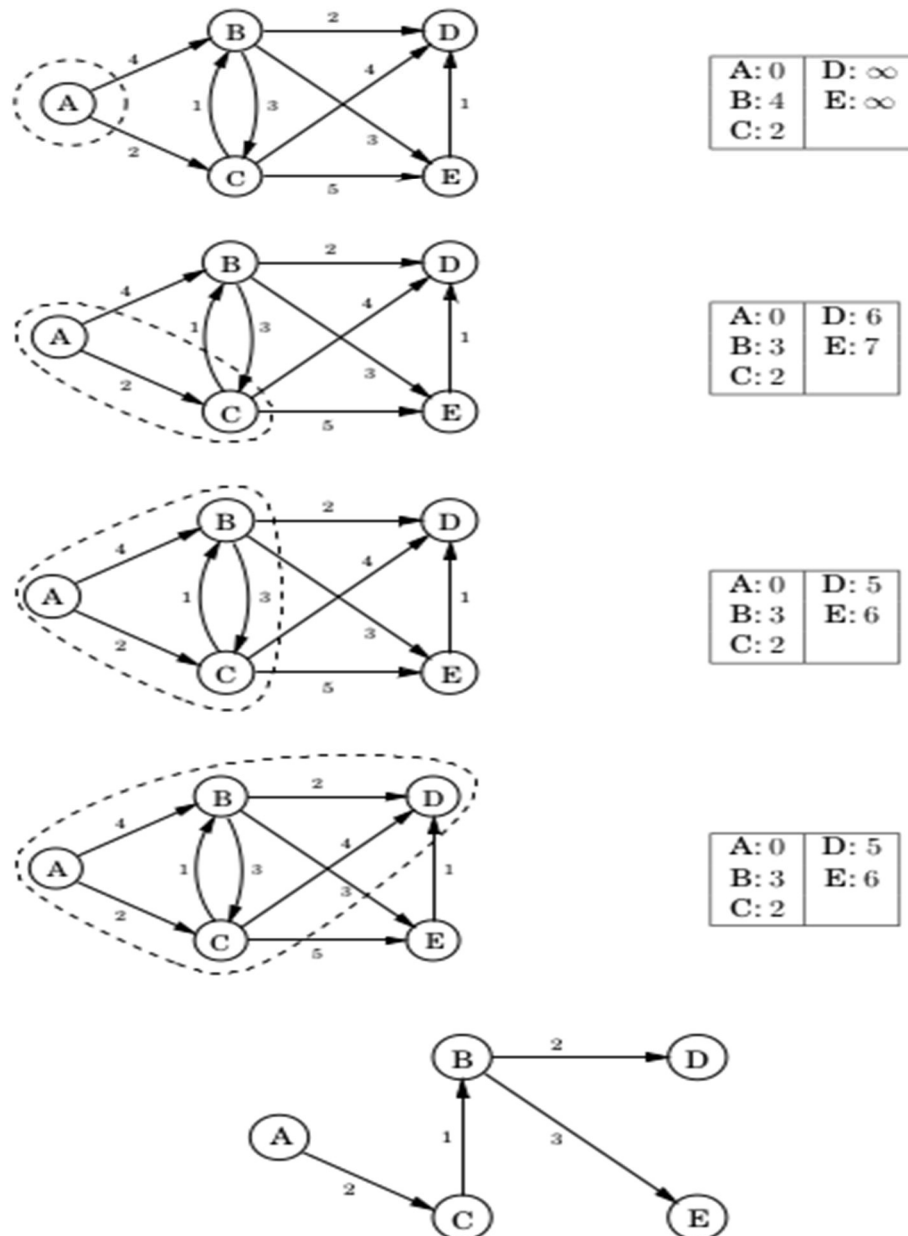
Decrease-key. Accommodate the decrease in key value of a particular element.

Delete-min. Return the element with the smallest key, and remove it from the set.

Make-queue. Build a priority queue out of the given elements, with the given key values. (In many implementations, this is significantly faster than inserting the elements one by one.)

The first two let us set alarms, and the third tells us which alarm is next to go off. Putting this all together, we get Dijkstra's algorithm.

**Figure 4.9** A complete run of Dijkstra's algorithm, with node *A* as the starting point. Also shown are the associated *dist* values and the final shortest-path tree.





### **Algorithm:**

1. start
2.  $n$ = num of vertices, queue=[], AL=[], E= edges entries with their cost, TSPL=[];
3. for  $e$  in E  
append edges entries  $e$  to AL
4. for  $i=0$  to  $i=n-1$   
append [inf,0] to TSPL
5. enqueue source  $s$  to queue
6. make cost 0 and visited 1 in TSPL
7. poll the minimum cost vertex from queue, while queue not empty(dequeue)
8. the minimum vertex is the destination vertex where we want to make our survey
9. visit all the neighbours of destination
10. find tentative shortest distance from the source to neighbour of destination reached
11. update in the TSPL if previous cost is greater than present cost
12. enqueue the unvisited neighbours to queue
13. repeat step 9 to step 12 until all Neighbors are visited
14. if queue is empty goto step 15 else make source =destination  
goto step 5
15. stop

## Python code to implement the Dijkstra's algorithm:

```
#####
```

```
def poll(PQ):  
    minimum = PQ[0][1]  
    index = 0  
    for i,e in enumerate(PQ[1:]):  
        if( minimum > e[1] ):  
            minimum = e[1]  
            index = i+1  
    e = PQ.pop(index)  
    return( e )
```

```
#####
```

```
n = 5;
```

```
E = [  
    [1,2,4],[1,3,2],  
    [2,3,3],[2,4,2],[2,5,3],  
    [3,2,1],[3,4,4],[3,5,5],  
    [5,4,1]  
]
```

```
AL = []  
for i in range(n):
```

```
AL.append([])
```

```
for e in E:
```

```
    AL[e[0]-1].append(e[1:])
```

```
TSPL = []
```

```
for i in range(n):
```

```
    TSPL.append([n+100,0]) # n+1 indicates infinity
```

```
        # 0 indicates not visited
```

```
print(TSPL)
```

```
#[[2, 3], [4, 6], [5, 7]]
```

```
PQ = []
```

```
# implement Dijkstra's algorithm
```

```
source = [1,0]
```

```
PQ.append(source)
```

```
TSPL[source[0]-1][0] = 0
```

```
TSPL[source[0]-1][1] = 1
```

```
while ( PQ != [] ): #and source[0] != destination ):
```

```
    dest = poll(PQ) # visit the new destination from source, which is the  
    nesrest neighbour
```

```
        # to the destination previosly visited
```

```

dest_index = dest[0]-1
dest_cost = TSPL[dest_index][0]
TSPL[dest_index][1] = 1 # update the visit to dest

for neighbor in AL[dest_index]: # visit all the neighbours of dest

    TSPL_neigh_index = neighbor[0] - 1

    # calculate the tentative shortest distance ( cost ) from source the
    # neighbor of dest reached
    cost = dest_cost + neighbor[1]

    if( cost < TSPL[TSPL_neigh_index][0] ):
        TSPL[TSPL_neigh_index][0] = cost

    found = 0;
    for pq in PQ:
        if( pq[0] == neighbor[0] ):
            pq[1] = cost
            found = 1
            break
    if(found == 0):
        PQ.append( [neighbor[0] , cost] )

print(TSPL, "\n\n", PQ)

```

## Output:

```
(darshanreddych123@darshan)-[~/Downloads]
$ python3 graph013.py
[[0, 1], [3, 1], [2, 1], [5, 1], [6, 1]]
[]
```