INDEX

S.R. NO.	TITLE	DATE	REMARK	SIGN.
1.	Write a program in C to create two sets and perform the Union operation on sets.			
2.	Write a program in C to create two sets and perform the Intersection operation on sets.			
3.	Write a program in C to create two sets and perform the Difference operation on sets.			
4.	Write a program in C to create two sets and perform the Symmetric Difference operation.			
5.	Write a C Program to find Cartesian Product of two sets.			
6.	Write a program in C to perform the Power Set operation on a set.			
7.	Write a program in C to Display the Boolean Truth Table for AND, OR , NOT .			
8.	Write a program in C for minimum cost spanning tree.			
9.	Write a program in C for finding shortest path in a Graph.			
10.	Working of Computation software.			
11.	Discover a closed formula for a given recursive sequence vice-versa.			
12.	Recursion and Induction			
13.	Practice of various set operations			
14.	Counting			
15.	Combinatorial equivalence			

S.R. NO.	TITLE	DATE	REMARK	SIG
16.	Permutations and combinations			
17.	Difference between structures, permutations, and sets			
18.	Implementation of a recursive counting technique			
19.	The Birthday problem			
20.	Poker Hands problem			
21.	Baseball best-of-5 series: Experimental probabilities			
22.	Baseball: Binomial Probability			
23.	Expected Value Problems			
24.	Basketball: One and One			
25.	Binary Relations: Influence			

Program 10:- Working of Computation software.

Theory:-

SageMath (previously Sage or SAGE) is a free open-source mathematical software system based on the Python programming language. Originally created for research into Mathematics, it has been evolving into a powerful tool for

Math education. It combines numerous other mathematical software packages with a single interface, using Python.

By learning SageMath, you are also learning a lot about Python.

As an open source project, SageMath invites contributions from all of its users. This tutorial is one of many sources.
of information for learning about how to use SageMath. For more information see the SageMath project's website.
This tutorial assumes that the reader has access to a running copy of SageMath. On most operating systems, installing
SageMath usually consists of downloading the proper package from the project's main website, unwrapping it, and
executing sage from within. For more information on the process of installing sage see SageMath's Installation Guide.
A good alternative is to run SageMath in the cloud using Cocalc. All you need to do is either sign up for a free account
or sign in through a Google/Github/Facebook/Twitter account. Once you are signed up, you can start a project using
SageMath, and also share it with other users. For more information about Cocalc and its features, visit Cocalc Tutorial.
If you opted for the physical installation and started SageMath, you should know that there are two ways to enter
commands: either from the command line or by using the web-based notebook. The notebook interface is similar in
design to the interface of Matlab, Mathematica, or Maple and is a popular choice.
Everything that follows the sage: prompt is a command that we encourage the reader to type in on their own.
For example, if we wanted to factor the integer 1438880 we would give the following example using SageMath's
factor() command.

```
sage: factor(1438880)
2^5 * 5 * 1...
```

Program 11:- Discover a closed formula for a given recursive sequence vice-versa.

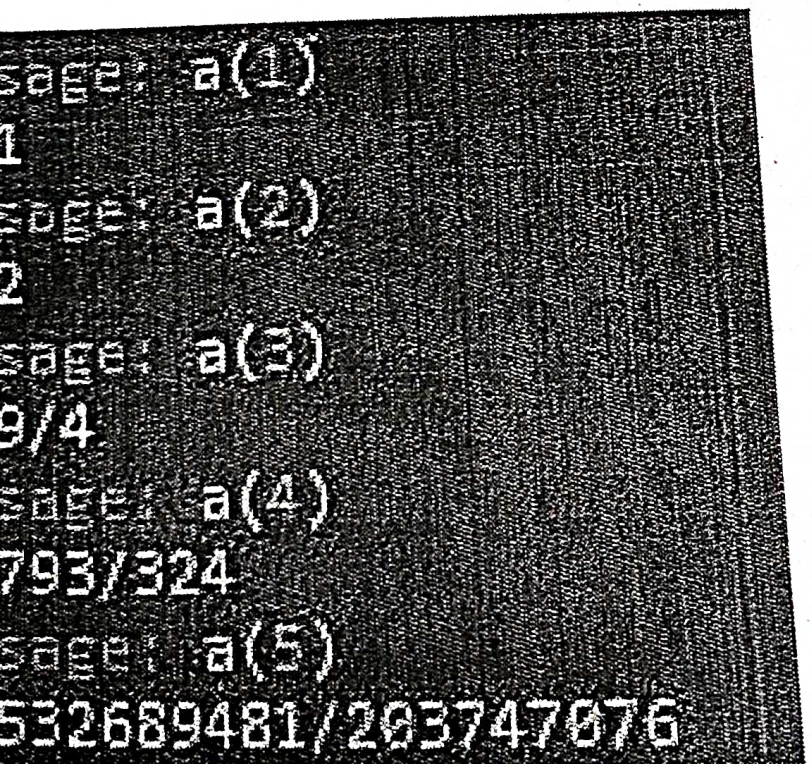
Theory:- A recursive sequence is a sequence in which terms are defined using one or more previous terms which are given. If you know the nth term of an arithmetic sequence and you know the common difference , d , you can find the (n+1)th term using the recursive formula an+1=an+d .

If you know the nth term and the common ratio , r , of a geometric sequence , you can find the (n+1)th term using the recursive formula. an+1=an·r

Source Code:-

```
def a(n):
  if n==1:
    return 1
  else:
    return a(n-1)+1/a(n-1)^2
```

Output:-



Program 12:- Recursion and Induction.

Theory:-

Recursion:-

We can use recursion to define:

- functions,
- sequences,
- sets.

Mathematical induction and strong induction can be used to prove results about recursively defined sequences and functions.

Structural induction is used to prove results about recursively defined sets.

Induction:-

Suppose you want to prove that a statement about an integer n is true for every positive integer n.

Define a propositional function P(n) that describes the statement to be proven about n.

To prove that P(n) is true for all $n \geq 1$, do the following two steps: I Basis Step:

- Prove that P(1) is true.
- I Inductive Step: Let $k \geq 1$. Assume P(k) is true, and prove that P(k + 1) is true.

Source Code:-

Recursion:-

```
def power(a,n):
  if (n==0):
    return 1
  else:
    return a*(power(a,n-1))
```

Induction:-

```
def Fac(n):
    if(n==0):
        return 1
    else:
        return n*Fac(n-1)
```

```
sage: def power(a,n):
....:     if(n==0):
....:         return 1
....:     else:
....:         return a*(power(a,n-1))
....:
sage: power(2,5)
32
```

```
sage: def Fac(n):
....:     if(n==0):
....:         return 1
....:     else:
....:         return n*Fac(n-1)
....:
sage: Fac(5)
120
```

```
sage: def power(a,n):
....:     if(n==0):
....:         return 1
....:     else:
....:         return a*(power(a,n-1))
....:
sage: power(2,5)
32
```

```
sage: def Fac(n):
....:     if(n==0):
....:         return 1
....:     else:
....:         return n*Fac(n-1)
....:
sage: Fac(5)
120
```

Program 13:- Practice of various set Operations.

Theory:-

The cardinality of a set: The cardinality of a set is the total number of unique elements in a set.

Example: A=1,6,7,8,9

The cardinality of a set A is: n(A)=5

Set Union:- The union of set A and B(denoted by AUB) is the set of all elements of A and B and the common both A and B.

$A \cup B=\{x : x \in A \text{ or } x \in B\}$

Ex.- A={1,2,3}

B={3,4,5}

AUB={1,2,3,4,5}

Set Intersection:- The intersection of sets A and B (denoted by $A \cap B$) is the set of elements which are in both A and B.

$A \cap B=\{x : x \in A \text{ and } x \in B\}$

Ex.- A={1,2,3}

B={2,3,4}

$A \cap B=\{2,3\}$

Set Difference:- The difference between two sets A and B is written A-B and means the set that consists of the elements of A which are not elements of B.

$A-B=\{x: x \in A \text{ and } x \notin B\}$

Ex:- A={1,2,3,4}

B={3,4,5,6}

A-B={1,2}

Symmetric Difference:- If A and B are the two sets be defined there symmetric difference as the set belongs to A or the set to B but not both A and B. (Denoted as $A \Delta B$)

$A \Delta B=\{x: (x \in A \text{ and } x \notin B) \text{ or } (x \in B \text{ and } x \notin A)\}$

Ex:- A={1,2,3,4}

B={2,3,5,6}

$A \Delta B=\{1,4,5,6\}$

Power Set:- Power set of a set is the set of all positive subset of 's' including the empty set.

The cardinality of the power set is 2^n.

Power set is denoted by P(s) where n, is the no. of elements of the set 's' .

Ex:- s={a,b,c}

Power set, P(s)={ {a} , {b} , {c} , {a,b} ,{a,c} , {b,c} , {a,b,c,} , {} }

- y = [2,3,3,...]
- A = Set(y)
- A
- A.cardinality()
- 8 in A
- 10 in A
- B = Set([8,6,17,-4,20,-2])
- B
- A.union(B)
- A.intersection(B)
- A.difference(B)
- B.difference(A)
- A.symmetric_difference(B)
- A = Set([1,2,3]); A
- powA = A.subsets(); powA
- pairsA = A.subsets(2); pairsA
- powA.list()
- pairsA.list()

Output:-

```
sage: Y [2,3,3,6,2,1,8,6,3]
sage: A Set(Y)
sage: A
{1, 2, 3, 6, 8}
sage: A.cardinality()
5

sage: 8 in A
True
sage: 10 in A
False
sage: B Set([8,6,17,-4,20,-2])
sage: B
{17, 20, 6, 8, -4, -2}
sage: A.union(B)
{1, 2, 3, 6, 8, 17, 20, -4, -2}
sage: A.difference(B)
{1, 2, 3}
sage: B.difference(A)
{17, 20, -4, -2}
sage: A.symmetric_difference(B)
{1, 2, 3, 17, 20, -4, -2}
sage: A Set([1,2,3])
sage: A
{1, 2, 3}
sage: powA A.subsets();
sage: powA
Subsets of {1, 2, 3}
sage: pairsA A.subsets(2);
sage: pairsA
Subsets of {1, 2, 3} of size 2
```

Output:-

```
sage: Y [2,3,3,3,2,1,8,6,3]
sage: A Set(Y)
sage: A
{1, 2, 3, 6, 8}
sage: A.cardinality()
5
sage: 8 in A
True
sage: 10 in A
False
sage: B Set([8,6,17, 4,20, 2])
sage: B
{17, 20, 6, 8, -4, -2}
sage: A.union(B)
{1, 2, 3, 6, 8, 17, 20, -4, -2}
sage: A.difference(B)
{1, 2, 3}
sage: B.difference(A)
{17, 20, -4, -2}
sage: A.symmetric_difference(B)
{1, 2, 3, 17, 20, -4, -2}
sage: A Set([1,2,3])
sage: A
{1, 2, 3}
sage: powA A.subsets();
sage: powA
Subsets of {1, 2, 3}
sage: pairsA A.subsets(2);
sage: pairsA
Subsets of {1, 2, 3} of size 2
```

Program 14:- Counting

Theory:- -

Counting seems easy enough: 1, 2, 3, 4, etc. This direct approach works well for counting simple things- like your toes-and may be the only approach for extremely complicated things with no identifiable structure. However, subtler methods can help you count many things in the vast middle ground, such as:

- The number of different ways to select a dozen doughnuts when there are five varieties available.
- The number of 16-bit numbers with exactly 4 ones

Perhaps surprisingly, but not coincidentally, these two numbers are the same: 1820

Counting is the study of arrangements of objects, it is an important part of discrete mathematics. We must count objects to solve many different types of problems, like the determining whether there are enough telephone numbers or internet protocol (IP) addresses to meet demand. Counting techniques are also used when probabilities of events are computed.

Source Code:-

```
Z= sage.functions.prime_pi.PrimePi()
loads(dumps(Z))
loads(dumps(Z))== Z
```

Program 15:- Combinatorialequivalence

Theory:- An equivalence relation on transversely measured train tracks is described which is generated by isotopy and two combinatorial moves called splitting and shifting. Equivalent measured tracks give rise to the same measured geodesic lamination (assuming the tracks are transversely recurrent), and the converse of this result is proved later in this chapter. There is a natural IR^-action on the set of measures supported on a track, and the resulting classes are called projective measures on the track. The collection of all projective measures on a fixed train track is found to have the natural structure of a polyhedron whose faces correspond exactly to the recurrent sub tracks. We then turn to certain useful facts, as follows. The equivalence relation generated by splitting and isotopy alone is shown to coincide with the one considered above. It is easy to see that if a track splits and shifts to another track, then the former track carries the latter one, and a converse to this result is given. Given a fixed multiple curve, we show how to split a measured train track until it hits the multiple curve efficiently. We finally ntroduce standard models for measured train tracks (from [PI]) proving that any measured train track is equivalent to a unique standard model. These standard models are used to prove that if two measured train tracks give rise to the same measured geodesic lamination, then the two train tracks are equivalent. The construction of a measured geodesic lamination from a measured train rack is extended to the setting where the train track is not necessarily ransversely recurrent.

ource Code:-

```
mset=["H","H","H","T","T"]
sage: Arrangements(mset,5).cardinality()
sage: S=Subsets([1,2,3,4,5,6],2,submultiset=True)
sage: S.list()
S=Subsets([1,2,3,4,5,6],4,submultiset=True)
S.list()
S=Subsets([1,2,2,3],submultiset=True)
S.cardinality()
sage: S.list()
```

```
sage: mset = [1,2,3,4,5,6]
sage: Arrangements(mset,5).cardinality()
10
sage: S = Subsets([1,2,3,4,5,6],2,submultiset=True)
sage: S.list()
[[1, 2],
 [1, 3],
 [1, 4],
 [1, 5],
 [1, 6],
 [2, 3],
 [2, 4],
 [2, 5],
 [2, 6],
 [3, 4],
 [3, 5],
 [3, 6],
 [4, 5],
 [4, 6],
 [5, 6]]
sage: S = Subsets([1,2,3,4,5,6],4,submultiset=True)
sage: S.list()
[[1, 2, 3, 4],
 [1, 2, 3, 5],
 [1, 2, 3, 6],
 [1, 2, 4, 5],
 [1, 2, 4, 6],
 [1, 2, 5, 6],
 [1, 3, 4, 5],
 [1, 3, 4, 6],
 [1, 3, 5, 6],
 [1, 4, 5, 6],
 [2, 3, 4, 5],
 [2, 3, 4, 6],
 [2, 3, 5, 6],
 [2, 4, 5, 6],
 [3, 4, 5, 6]]
```

```
sage: S = Subsets([1,2,2,3],submultiset=True)
sage: S.cardinality()
12
sage: S.list()
[[],
 [1],
 [2],
 [3],
 [1, 2],
 [1, 3],
 [2, 2],
 [2, 3],
 [1, 2, 2],
 [1, 2, 3],
 [2, 2, 3],
 [1, 2, 2, 3]]
```

Program 16:- Permutations and Combinations

Theory:-

Permutations:- A permutation is an arrangement of objects in a definite order. The members or elements of sets are arranged here in a sequence or linear order. For example, the permutation of set A={1,6} is 2, such as {1,6}, {6,1}. As you can see, there are no other ways to arrange the elements of set A.

Combination:- A combination is a way of selecting items from a collection where the order of selection does not matter. Suppose we have a set of three numbers P, Q and R. Then in how many ways we can select two numbers from each set, is defined by combination.

Source Code:-

Permutations:-

- Mset=[1,2,3,4,5,6]
- Permutations(Mset,2).list()

Combinations:-

- Mset=[1,2,3,4,5,6]
- Combinations(Mset,4).list()

Output:-

```
sage: Mset [1,2,3,4,5,6]
sage: Permutations(Mset,2) list()
[[1, 2],
 [1, 3],
 [1, 4],
 [1, 5],
 [1, 6],
 [2, 1],
 [2, 3],
 [2, 4],
 [2, 5],
 [2, 6],
 [3, 1],
 [3, 2],
 [3, 4],
 [3, 5],
 [3, 6],
 [4, 1],
 [4, 2],
 [4, 3],
 [4, 5],
 [4, 6],
 [5, 1],
 [5, 2],
 [5, 3],
 [5, 4],
 [5, 6],
 [6, 1],
 [6, 2],
 [6, 3],
 [6, 4],
 [6, 5]]
```