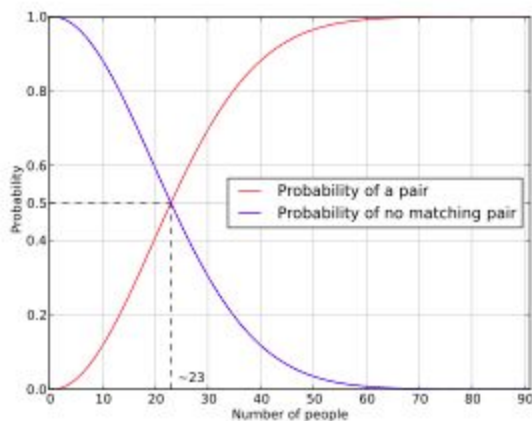


EE4371 Endsem Report

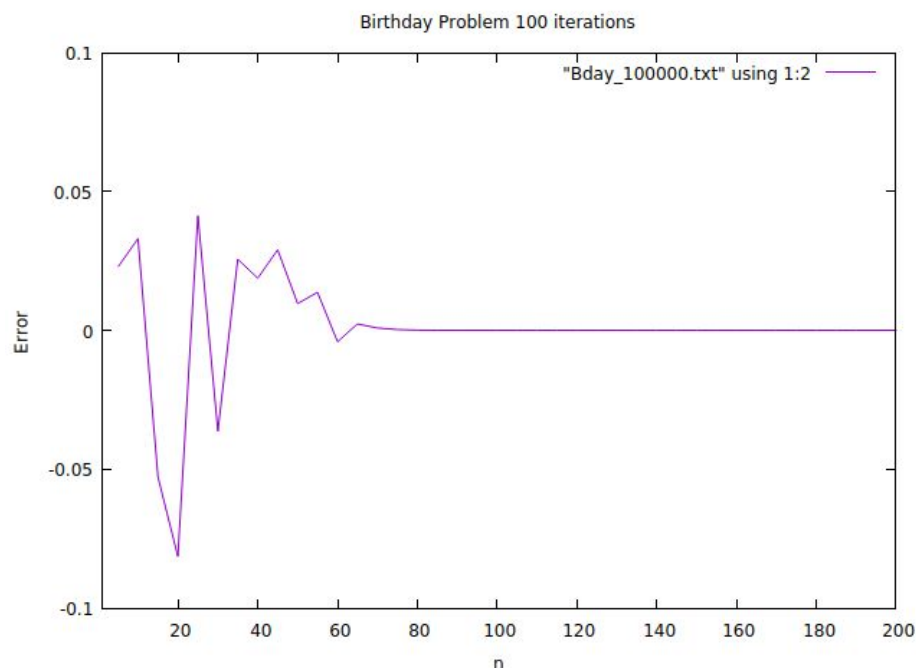
Qs 1) Design a program that can test the Birthday problem, by a series of experiments, on randomly generated birthdays which test this paradox for $n = 5, 10, 15, 20, 25, 30 \dots 200$.

Probability for n people is $1 - (365 \text{ permute } n \text{ divided by } 365^n)$ using complementation method for finding probability since $365 \text{ permute } n \text{ divided by } 365^n$ is probability for all distinct birthdays.

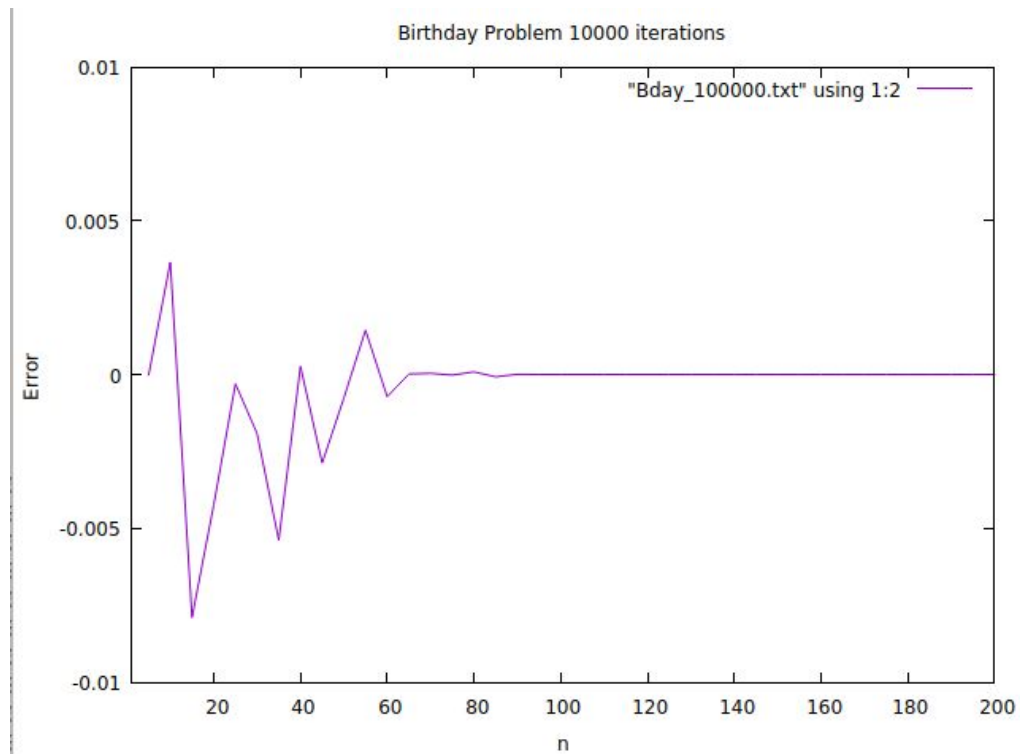


The .py script for the same is also attached in the submission. The output of the code is writing the error values to a .txt file, which I have plotted using gnuplot, also attached the snapshots of the plots.

Error plots: For max_iter 100: (conducting experiment 100 times)



For max_iter 10000: (conducting experiment 10000 times)



As is expected the error value (difference between experimental and theoretical probabilities decreases as we increase the number of times the experiment is conducted)

The error also goes down as n increases since the probability becomes a very high value and experimentally also it gives similar values so error is almost 0 ($< 10^{-7}$ since such small errors cannot be taken in float on python, once error $< 10^{-7}$ it will show error as 0 only. In error trend a maxima is spotted at around $n=15, n=20$ since for very small n again probability is very low and experimental is also low hence making difference small, error is maximum around values where theoretical probability is in the range 0.2 to 0.8 below and over this the error is very less.

The theoretical probability is in this range of 0.2 to 0.8 for n in the range of 15 to 35 as from the first graph which shows probability vs n for all n.

Code for the getting the above is :

```
1 #Endsem Qs 1)
2 #Birthday Problem : Probability that 2 birthdays coincide for randomly chosen n people
3 #Probability for n people is 1 - (365 permute n divided by 365^n) using complementation
4 #method for finding probability since 365 permute n divided by 365^n is probability for all distinct birthdays
5
6 import random
7
8 def recursive(n):                                #Recursion for 365 permute n divided by 365^n
9     if n==1:
10         return 1
11     else:
12         return ((366-n)/365)*recursive(n-1)
13
14 def theoretical_probability(n):                  #Theoretical probability that 2 birthdays coincide
15     return 1-recursive(n)
16
17 def experimental_probability(n,max_iter):
18
19     s=0                                          #Expermintal probability variable
20     c=0                                          #Count variable
21     for x in range(max_iter):                  #Running the experiment 'iteration' number of times
22         date_count = {}                        #Initialsing a empty dictionary for storing dates as key and the number of people having birthday as that date as value
23         for y in range(n):                    #Calculating birthday for each person
24             rand_num = random.randint(1,365)  #Creating a random birthday
25             if rand_num in date_count:         #If new date is in the dictionary, increase count by 1 and break
26                 c += 1
27                 break
28             else:                              #Else add the date in the dictionary with key = date and value = 1 which signifies that the date is in the dictionary
29                 date_count[rand_num]=1
30
31     s+=c                                        #sum of c's for each trial
32     p=s/max_iter                              #Experimental probability is sum of trials divided by number of trials
33     return p
34
35 file = open("Bday_100000.txt", "w")
36 max_iter = 10000
37 for x in range(5,201,5):                      #Iterating the values of n
38     file.write("%d %d\n"%(x,(float(experimental_probability(x,max_iter))-float(theoretical_probability(x)))))
39     #writing theoritical and experimental probability for each n to the file "Bday_100000.txt"
40     #will plot the data in this file by setting max_iter to 100 first then to 10000
41     #Have attached both the plots in the document which is : [experimental_probability(n,max_iter)-theoretical_probability(n)] vs n for each n
```

Time Complexity analysis :

The time complexity of finding the theoretical probability is $O(n)$ since we iterate through the loop of order n once to find 365 permute n and 365^n .

The time complexity of experimental probability : In this we are generating a random integer in $O(1)$ time and checking if it is already present in the dictionary in $O(1)$ time but the total function is looping over max_iter and n in nested form so time complexity of experimental probability is $O(n \cdot \text{max_iter})$. In our case we have taken max_iter as 100 first and then 10000.

Qs 2) In each of the following situations, indicate whether $f = O(g)$, or $f = \Omega(g)$, or both (in which case). Justify your answer.

We know that, a function $f(x)$ is said to be $f(x) = O(g(x))$ if there exists a positive real number 'c' s.t. $f(x) < cg(x)$ for all $x \geq x_0$ OR as $x \rightarrow \infty$.

Also, a function $f(x)$ is said to be $f(x) = \Omega(g(x))$ if there exists a positive real number 'c' s.t. $f(x) > cg(x)$ for all $x \geq x_0$ OR as $x \rightarrow \infty$.

Note that if both of these conditions come true for distinct real numbers c_1 and c_2 then it is said that $f(x) = \Theta(g(x))$.

1. $f(n) = n-100$, $g(n) = n-200$

Take $c_1 = 1 \Rightarrow f(n) > cg(n)$ for all n since $n-100 > n-200$ is always true

Take $c_2 = 2 \Rightarrow f(n) < cg(n)$ for all $n > 300$

Since both the definitions are true, in this case, $f(n) = \Theta(g(n))$.

2. $f(n) = 100n + \log n$, $g(n) = n + (\log n)^2$

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \frac{100n + \log n}{n + (\log n)^2} = \frac{100 + 1/n}{1 + \frac{2 \log n}{n}} = \frac{100n+1}{n+2 \log n} = \frac{100}{1+2/n} = \frac{100n}{n+2} = 100$$

We get the above using L'Hospital's Theorem for limits.

Since we have $f(n) = 100 \cdot g(n)$ as $n \rightarrow \infty$,

Take $c_1 = 99 \Rightarrow f(n) > cg(n)$ as $n \rightarrow \infty$

Take $c_2 = 101 \Rightarrow f(n) < cg(n)$ as $n \rightarrow \infty$

Since both the definitions are true, in this case, $f(n) = \Theta(g(n))$.

3. $f(n) = \log 2n$, $g(n) = \log 3n$

$$\Rightarrow f(n) = \log 2 + \log n, g(n) = \log 3 + \log n$$

Take $c_1 = 1 \Rightarrow f(n) < cg(n)$ for all n since $\log 3 > \log 2$ is always true

Take $c_2 = 2 \Rightarrow f(n) > cg(n)$ for all n since $2 \log 2 > \log 3$ is always true

Since both the definitions are true, in this case, $f(n) = \Theta(g(n))$.

4. $f(n) = n \cdot 2^n$, $g(n) = 3^n$

Take $c = 1 \Rightarrow n \cdot 2^n < 3^n$ i.e $n < (\frac{3}{2})^n$ i.e $\log n < n \cdot \log(1.5)$ which is true for all $n \geq 1$

If we try to show the other inequality, we will get: $c \cdot n \cdot 2^n > 3^n$ which would be true only if there exists a 'c' such that $c > (1.5)^n \div n$ but the function on RHS tends to ∞ as n tends to $\infty \Rightarrow$ No such c exists.

Therefore, we have that $f(n) = O(g(n))$.

Qs 3) Show that the running time of the merge-sort algorithm on n-element sequence is $O(n \log n)$, even when n is not a power of 2.

I will use the method of Mathematical induction to show the same.

Claim : $T(n) \leq c_1 \times n \log n + c_2 \times n$, where c_1, c_2 are arbitrary real constants.

We know from the recursion that : $T(n) = T(\lfloor n/2 \rfloor) + T(n - \lfloor n/2 \rfloor) + k_2 \times n$, where k_2 is some real constant ($k_2 \times n$ term comes from the merge call)

Base Case : $n = 1 \Rightarrow T(1) = c$ (since nothing to sort) $\leq c_1 \times 1 \log 1 + c_2 \times 1$, such constants can be easily found, hence shown for base case.

Assumption : Let the claim be true for all integers from 1 to $n-1$ for some $n > 1$.

To show that : The claim is true for n

Using the recursion; we have $T(n) = T(\lfloor n/2 \rfloor) + T(n - \lfloor n/2 \rfloor) + k_2 \times n$

Case 1: n is even $\Rightarrow T(n) = 2 \times T(n/2) + k_2 \times n$

Since for all integers less than n , the claim is true, it is true for $n/2$ and we can use that equation and substitute the value in the above equation.

$$\Rightarrow T(n) \leq 2 \times \{c_1 \times (n/2) \times \log(n/2) + c_2 \times (n/2)\} + k_2 \times n$$

$$\Rightarrow T(n) \leq c_1 \times n \times (\log n - \log 2) + c_2 \times n + k_2 \times n$$

$$\Rightarrow T(n) \leq c_1 \times n \log n + (c_2 + k_2 - c_1 \log 2) \times n$$

Since constants can be arbitrary real numbers in the inequality claimed for $T(N)$, choose $c_1' = c_1$ and $c_2' = c_2 + k_2 - c_1 \log 2$

$$\Rightarrow T(n) \leq c_1' \times n \log n + c_2' \times n$$

Hence, the claim has been proven for even n .

Case 2 : n is odd $\Rightarrow T(n) = T((n-1)/2) + T((n+1)/2) + k_2 \times n$

Since for all integers less than n , the claim is true, it is true for $(n-1)/2$, $(n+1)/2$ and we can use that equation and substitute the value in the above equation.

$$\Rightarrow T(n) \leq c1 \times ((n-1)/2) \times \log((n-1)/2) + c2 \times ((n-1)/2) + c1 \times ((n+1)/2) \times \log((n+1)/2) + c2 \times ((n+1)/2) + k2 \times n$$

$$\Rightarrow T(n) \leq c1 \times \{(n-1)/2 \times \log(n-1) + (n+1)/2 \times \log(n+1)\} - c1 \times n \log 2 + c2 \times n + k2 \times n$$


To find a constant $k1$ such that :

$$(n-1)/2 \times \log(n-1) + (n+1)/2 \times \log(n+1) \leq k1 \times n \log n$$

$$\Leftrightarrow \log(n-1)^{(n-1)} + \log(n+1)^{(n+1)} \leq \log n^{(2 \times k1 \times n)}$$

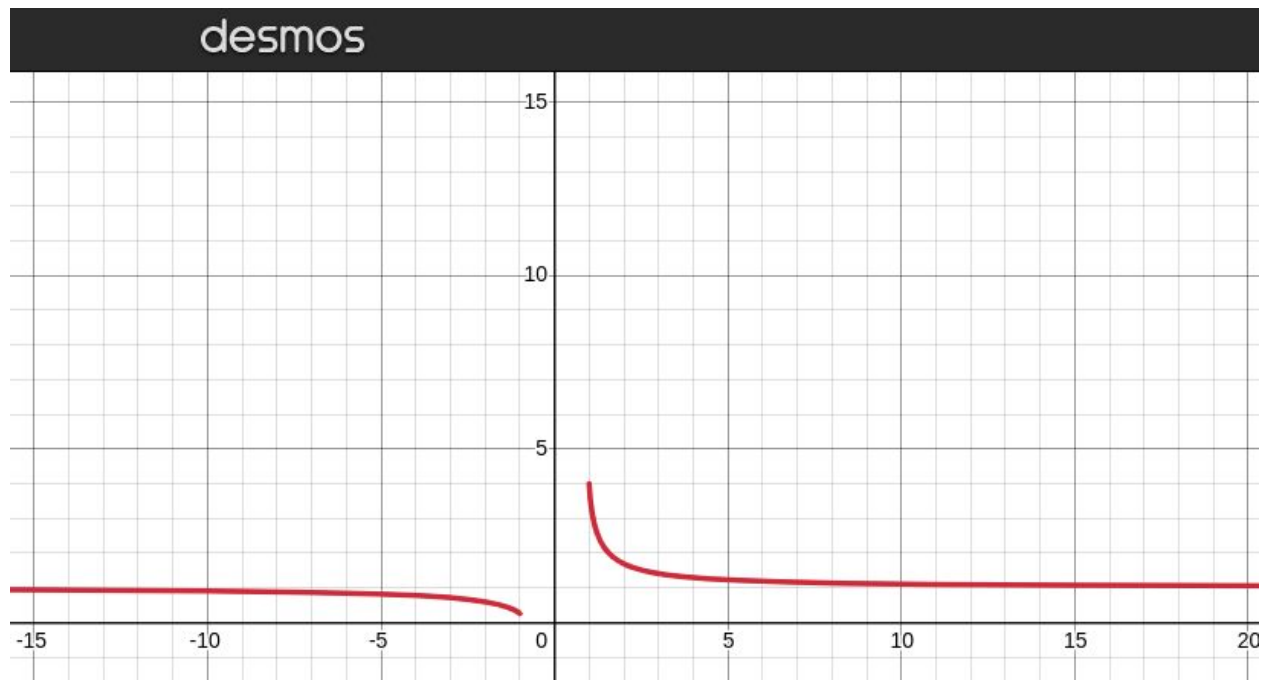
$$\Leftrightarrow (n-1)^{(n-1)} \times (n+1)^{(n+1)} \leq n^{(2 \times k1 \times n)}$$

$$\Leftrightarrow ((n-1)/n)^{(n-1)} \times ((n+1)/n)^{(n+1)} \leq n^{2n(k1-1)}$$



$$\left(\frac{(x-1)}{x} \right)^{(x-1)} \cdot \left(\frac{(x+1)}{x} \right)^{(x+1)}$$

The graph for the above equation is as follows :



Which implies, that $LHS \leq 4$ and equality occurs as n tends to 1.

Since we clearly have $n > 1$; we have to find a $k1$ such that :

$4 \leq n^{2n(k_1-1)}$ and we have $n \geq 2 \Rightarrow k_1 = 2$ will also suffice the equality for every n .

Choose $k_1 = 2$ for this case :

$$\Rightarrow T(n) \leq 2c_1 \times n \log n + (c_2 + k_2 - c_1 \log 2) \times n$$

Since constants can be arbitrarily chosen, take $c_1' = 2c_1$, $c_2' = c_2 + k_2 - c_1 \log 2$

$$\Rightarrow T(n) \leq c_1' \times n \log n + c_2' \times n$$

Hence, the claim has been proven for odd n .

Since it has been shown that the claim is true for all the cases, we can now say by the method of Mathematical Induction that the claim is true for all integers n .

$$\Rightarrow T(n) = O(n \log n) \Leftrightarrow \text{for some } c, T(n) \leq c \times n \log n \text{ for all } n \geq \text{some } m$$

Choose $c = c_1' + c_2' \Rightarrow c_1' \times n \log n + c_2' \times n \leq (c_1' + c_2') \times (n \log n)$

Hence proven that $T(n) = O(n \log n)$ for all integers n

Qs 4) Show how to implement a stack using two queues. Analyze the running time of the stack operations.

A little bit about the 2 data structures.

Queue : One end is always used to insert data (enqueue) and the other is used to remove data (dequeue). Queue follows FIFO (First-In-First-Out) methodology, i.e., the data item stored first will be accessed first.

Basic operations of queue : Basic operations associated with queues –

- enqueue() – add (store) an item to the queue.
- dequeue() – remove (access) an item from the queue.

Few more functions are required to make the above-mentioned queue operation efficient. These are –

- peek() – Gets the element at the front of the queue without removing it.
- isfull() – Checks if the queue is full.
- isempty() – Checks if the queue is empty.

To enqueue a new element, the rear pointer is used and to dequeue the front pointer is used.

Stacks : It is a LIFO (Last-in First-out) data structure. Here, the element which is placed (inserted or added) last, is accessed first. In stack terminology, insertion operation is called PUSH operation and removal operation is called POP operation.

Basic Operations : A stack is used for the following two primary operations –

- push() – Pushing (storing) an element on the stack.
- pop() – Removing (accessing) an element from the stack.

To use a stack efficiently, we need to check the status of stack as well. For the same purpose, the following functionality is added to stacks –

- peek() – get the top data element of the stack, without removing it.
- isFull() – check if stack is full.
- isEmpty() – check if stack is empty.

So as it is clearly visible, the functionality of a stack and queue is not much different other than the fact that a queue is FIFO and a stack is LIFO.

To implement a stack using 2 queues, it can be done using 2 methods.

Method 1: Push implementation is done in $O(n)$ and pop implementation is done in $O(1)$

Method 2: Push implementation is done in $O(1)$ and pop implementation is done in $O(n)$

Our general idea is that while implementing any of the functions, we will use the other queue as a buffer memory to store data temporarily. At the start and end of the functions being called one queue will always be empty.

Method 1:

First dequeue original queue into second queue, add the new element to the original queue, it will automatically become the first element and then enqueue the original queue from the second queue.

By doing this, our newly PUSHed element is now always at the front of the queue (which happens for a stack) and while POPing elements, always dequeue from the original queue since it always has the latest added element at the front of the queue which ensures that our so-formed data structure is LIFO like a stack.

Method 2:

In this, while PUSHing an element, just enqueue it to the original queue. To POP an element, dequeue all but the last element from the original queue and enqueue it into the second queue. Then again dequeue from second queue and enqueue into the original queue thus making the latest added element as the front of the queue and when POP is called it returns this which makes this implementation a LIFO type.

Code for method 1:

```
In [1]: from queue import Queue

class Stack:

    def __init__(self):

        # Two inbuilt queues
        self.q1 = Queue()
        self.q2 = Queue()

        self.current_size = 0    # To maintain current number of elements

    def push(self, x):
        self.current_size += 1    # Since an element is added, increase number of elements by 1

        # Push x first in empty q2
        self.q2.put(x)

        # Push all the remaining elements in q1 to q2.
        while (not self.q1.empty()):
            self.q2.put(self.q1.queue[0])
            self.q1.get()

        # swap the names of two queues since q1 is our original queue
        # and q2 is the one we are keeping as an empty buffer
        self.q = self.q1
        self.q1 = self.q2
        self.q2 = self.q

    def pop(self):

        # if no elements are there in q1
        if (self.q1.empty()):
            return "Cannot pop since the queue is empty"
        self.q1.get()
        self.current_size -= 1    # Since we are removing an element, decrease number of elements by 1

    def peek(self):    # Returns the value of element at the top
        if (self.q1.empty()):
            return "The queue is empty"
        return self.q1.queue[0]

    def size(self):
        return self.current_size
```

```
In [2]: s = Stack()
list1 = [1,4,2,8,5,7]

for x in list1:
    s.push(x)    # To make a stack of the elements in the order as above
                # Current size should be 6 and peek should return the last element (i.e 7) since it is LIFO

print("current size: ", s.size())
print("Element at top is:",s.peek())

s.pop()
s.pop()

    # After popping twice, top element should be 8 and size should be 4
print("current size: ", s.size())
print("Element at top is:",s.peek())

s.push(9)

    # After element 9 is pushed into stack, element at top should be 9 and size should be 5
print("current size: ", s.size())
print("Element at top is:",s.peek())
```

```
current size: 6
Element at top is: 7
current size: 4
Element at top is: 8
current size: 5
Element at top is: 9
```

Code for method 2:

```
In [1]: from queue import Queue

class Stack:

    def __init__(self):

        # Two inbuilt queues
        self.q1 = Queue()
        self.q2 = Queue()

        self.current_size = 0    # To maintain current number of elements

    def push(self, x):
        self.current_size += 1    # Since an element is added, increase number of elements by 1

        # Simply add element to original queue
        self.q1.put(x)

    def pop(self):

        # if no elements are there in q1
        if (self.q1.empty()):
            return "Cannot pop since the queue is empty"

        # Push all the remaining elements in q1 to q2.
        while (self.q1.qsize() != 1):
            self.q2.put(self.q1.queue[0])
            self.q1.get()

        # swap the names of two queues since q1 is our original queue
        # and q2 is the one we are keeping as an empty buffer
        self.q = self.q1
        self.q1 = self.q2
        self.q2 = self.q

        self.current_size -= 1    # Since we are removing an element, decrease number of elements by 1

    def peek(self):    # Returns the value of element at the top
        if (self.q1.empty()):
            return "The queue is empty"
        return self.q1.queue[-1]

    def size(self):
        return self.current_size
```

```
In [2]: s = Stack()
list1 = [1,4,2,8,5,7]

for x in list1:
    s.push(x)    #To make a stack of the elements in the order as above
                #Current size should 6 and peek should return the last element (i.e 7) since it is LIFO

print("current size: ", s.size())
print("Element at top is:",s.peek())

s.pop()
s.pop()    #After popping twice, top element should be 8 and size should be 4
print("current size: ", s.size())
print("Element at top is:",s.peek())

s.push(9)    #After element 9 is pushed into stack, element at top should be 9 and size should be 5
print("current size: ", s.size())
print("Element at top is:",s.peek())

current size: 6
Element at top is: 7
current size: 4
Element at top is: 8
current size: 5
Element at top is: 9
```

The 2 methods are very similar, the only difference is that in the first method we dequeue and enqueue to put the newly added element at the top during the PUSH function call and in the second method we dequeue and enqueue to put the newly added element at the top during the POP function call.

Qs 5) You are given an array of n elements, and you notice that some of the elements are duplicates; that is, they appear more than once in array. Show how to remove all duplicates from the array in time $O(n \log n)$.

This can be done using 2 methods:

1st method (Using merge sort):

The assumption for this method is that each element of the array is a float or an integer (since merge sort can be implemented on numbers).

The idea is to sort the array using merge sort in time complexity $O(n \log n)$ and then delete all the duplicates which can be done in time complexity $O(n)$ since we have to iterate through the sorted array only once to delete all duplicates.

So total time complexity is $O(n \log n)$.

Following is the attached code (I have also included the .ipynb file and its downloaded .html form in the submission)

```
In [1]: #Will write a Merge Sort code and implement it to order the sequence
#Then will iterate through the sorted sequence once to remove all the repeated entries
```

```
In [2]: def Merge_Sort(seq):
n=len(seq)
if n<=1:                                #Defining initial condition for recursion
    return seq
if n%2==0:
    n1=n//2
else:
    n1=(n+1)//2
left = (seq[0:n1])                        #Dividing original array into left half approximately
right = (seq[n1:n])                       #Dividing original array into right half approximately
L=Merge_Sort(left)                        #Recursive step on left
R=Merge_Sort(right)                       #Recursive step on right
return Merge(L,R)
```

```
In [3]: def Merge(L,R):
n1=len(L)
n2=len(R)
n = n1+n2
result = [0]*n                            #Creating empty list of total length to store sorted list
i=0                                        #Variable to iterate through L
j=0                                        #Variable to iterate through R
k=0                                        #Variable to iterate through result list

while i < n1 and j < n2 :                  #Merge Step
    if L[i] <= R[j]:                       #To put smaller element before larger element
        result[k] = L[i]
        i += 1
    else:
        result[k] = R[j]
        j += 1
    k += 1

while i < n1:                              # Copy the remaining elements of L[], if there are any
    result[k] = L[i]
    i += 1
    k += 1

while j < n2:                              # Copy the remaining elements of R[], if there are any
    result[k] = R[j]
    j += 1
    k += 1
return result
```

```
In [4]: X = input("Enter the sequence from which you want to delete duplicates:")
Enter the sequence from which you want to delete duplicates:-2 3 4.5 7 9 -6 4.5 -2 3
```

```
In [5]: temp = X.split()
seq = list()
for i in temp:
    if i.isnumeric():
        seq.append(int(i))
    else:
        seq.append(float(i))
#Creating empty list to append to it the int for float values from input
#Since time complexity of function is O(nlogn), this will not affect run time
#much, and since merge sort is written for floats we have to input a seq of
#floats or ints or combined
```

```
In [6]: sorted_seq = Merge_Sort(seq)
#Applying Merge Sort to the sequence
```

```
In [7]: deleter = sorted_seq[0]
result_seq = list()
result_seq.append(deleter)
k = len(sorted_seq)
for j in range(0,k):
    if j!=0:
        if sorted_seq[j] == deleter:
            continue
        else:
            deleter = sorted_seq[j]
            result_seq.append(deleter)
#Creating deleter to check if element is repeating
#Since first element will be there, if repeated again will
#not append it to list as can be seen from the for loop
#Since I have assigned deleter variable to sorted_seq[0]
#and don't want to delete that
#Not adding already occurred element, hence continue
#Appending newly occurred element
print(seq)
print(result_seq)
#printing original and sorted sequence

[-2.0, 3, 4.5, 7, 9, -6.0, 4.5, -2.0, 3]
[-6.0, -2.0, 3, 4.5, 7, 9]
```

2nd method: Using dictionaries which uses the hash table structures.

Using a Hash table is much more efficient since the time complexity is $O(n)$ for this implementation. Implementing hash table algorithm from scratch for inputs is difficult, but it can be done using the in-built structure of dictionaries which uses the technique of hashing (i.e creating key/value pairs to store elements in it) Each element in a dictionary is given a key which can be used to access the value given to the key in $O(1)$ from the dictionary.

This method does not need to make any assumptions on the input, since we are not sorting, we are just inserting the entries in a dictionary or doing look-up on the dictionary every time a new entry form the list comes.

So we will initiate an empty dictionary and iterate through the list, we will add an element as key = element, value = 1 to the dictionary if it is already not present in it (which can be checked in $O(1)$) and therefore run time of this implementation will be $O(n)$.

How hashing works :

- When adding entries to the table, we start with some slot, that is based on the hash of the key.
- If that slot is empty, the entry is added to the slot.
- If the slot is occupied, Python compares the hash AND the key of the entry in the slot against the hash and key of the current entry to be inserted respectively.
- If both match, then it thinks the entry already exists, gives up and moves on to the next entry to be inserted. If either hash or the key don't match, it starts probing.
- Probing just means it searches the slots by slot to find an empty slot.
- Python uses random probing. In random probing, the next slot is picked in a pseudo random order. The entry is added to the first empty slot. What is important is that the slots are probed until the first empty slot is found. The same thing happens for lookups, just starts with the initial slot i (where i depends on the hash of the key).
- If the hash and the key both don't match the entry in the slot, it starts probing, until it finds a slot with a match. If all slots are exhausted, it reports a fail.
- When a new dict is initialized it starts with 8 slots and the dict will be resized if it is two-thirds full. This avoids slowing down lookups.
- This implementation causes adding and lookup in a dictionary take $O(1)$ in general

Following is attached the code for the same :

```
In [8]: #Faster implementation than in O(nlogn) is possible to get the same result

In [9]: #It can be done by using dictionary which uses the principle of hash tables.

In [10]: def remove_duplicates ():
    print("To remove duplicates from a sequence of elements ")
    print()

    seq2 = input("Enter the sequence of elements seperated by space : ")
    list_of_elements = seq2.split()          #To make a list of the sequence given as input by user

    dict_of_elements = {}                   #Creating an empty dictionary
    sorted_seq2 = []                       #Creating an empty list to store result

    for element in list_of_elements:
        if element in dict_of_elements:    #If element is already present, do nothing
            pass
        else:
            sorted_seq2.append(element)     #If a new element has occurred, add it to the sorted_seq2 list
            dict_of_elements[element] = 1   #If a new element has occurred, add its entry in dictionary
                                            #with the element as key and its value as 1

    return sorted_seq2

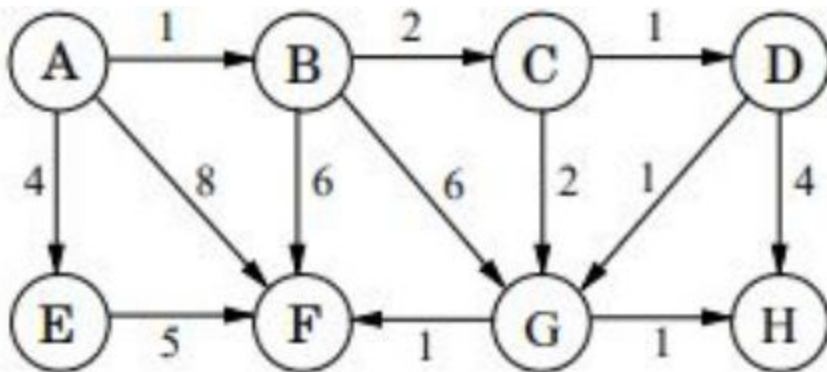
sorted_seq2 = remove_duplicates()
print(sorted_seq2)

To remove duplicates from a sequence of elements

Enter the sequence of elements seperated by space : cd 5 6.7 -8 a cd 6.7 -6 5
['cd', '5', '6.7', '-8', 'a', '-6']
```

Qs 6) Suppose Dijkstra's algorithm is run on the following graph, starting at node A. (a) Draw a table showing the intermediate distance values of all the nodes at each iteration of the algorithm. (b) Show the final shortest-path tree.

Original graph:



Dijkstra's algorithm is used to find the shortest path between any 2 nodes of a graph. The procedure is, that at each iteration of the algorithm, the next node is added to the tree and distances are recalculated : if there occurs a shorter path due to the introduction of that node, then the distances are updates, this procedure is followed till all the nodes are added to the tree. Distances to nodes that are not directly connected to other nodes by edges are taken as ∞ and other distances are taken as the value which it represents, distance to itself is taken as 0. Step by step tabular form of distances as measured from A are :

Step 1) After adding A:

Node	Shortest distance
A	0
B	1
C	∞
D	∞
E	4
F	8
G	∞
H	∞

Step 2) After adding node B to the tree :

Node	Shortest distance
A	0
B	1
C	3
D	∞
E	4
F	7
G	7
H	∞

Step 3) After adding node C to the tree :

Node	Shortest distance
A	0
B	1
C	3
D	4
E	4
F	7
G	5
H	∞

Step 4) After adding node D to the tree :

Node	Shortest distance
A	0
B	1
C	3
D	4
E	4
F	7
G	5
H	8

Step 5) After adding node E to the tree :

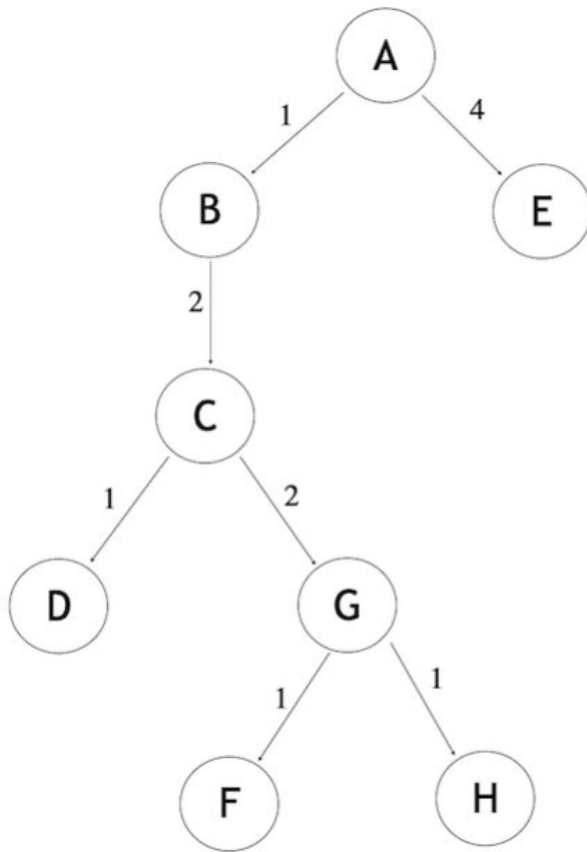
Node	Shortest distance
A	0
B	1
C	3
D	4
E	4
F	7
G	5
H	8

Since node F does not have any outgoing paths, it will be skipped, since it will not give any better results to the current shortest paths.

Step 6) After adding node G to the tree:

Node	Shortest distance
A	0
B	1
C	3
D	4
E	4
F	6
G	5
H	6

These are the final shortest paths from A to all the nodes in the given graph. The final tree looks as follows :



Qs 7) Algorithms for “Transport Protocols” (i) Describe the congestion control algorithms for (a) Cubic TCP, and (b) Compound TCP. Comment on the similarities and the differences between Cubic TCP and Compound TCP. (ii) What does it mean for a TCP to be fair? And how might one evaluate fairness when TCP operates over a large scale network, like the Internet? (iii) How might you design the congestion control algorithms of a TCP, where the efficiency (faster download times) and the fairness attributes of your proposal are better than both Cubic and Compound TCP.

Brief : TCP (Transmission Control Protocol) is a standard that defines how to establish and maintain a network conversation through which application

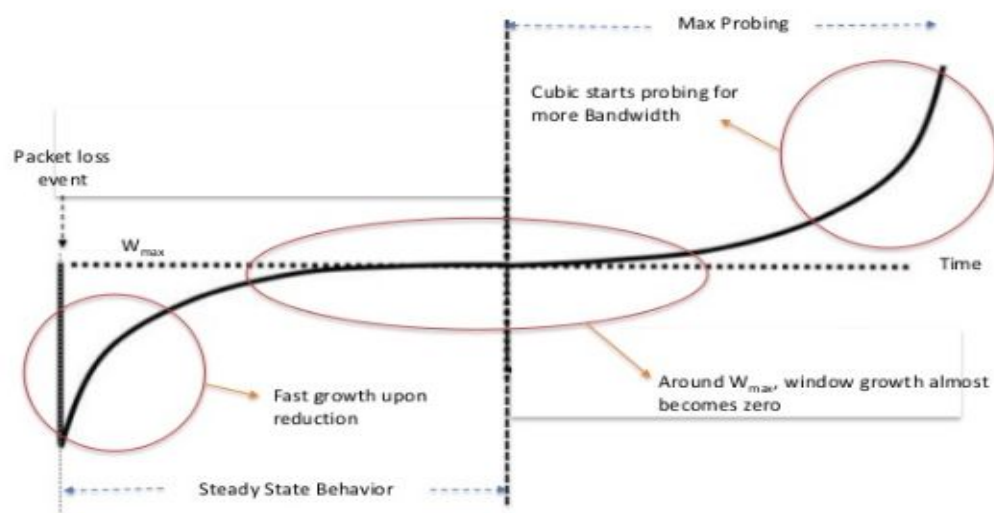
programs can exchange data. TCP works with the Internet Protocol (IP), which defines how computers send packets of data to each other.

TCP is a connection-oriented protocol, which means a connection is established and maintained until the application programs at each end have finished exchanging messages. It determines how to break application data into packets that networks can deliver, sends packets to and accepts packets from the network layer, manages flow control and -- because it is meant to provide error-free data transmission -- handles retransmission of dropped or garbled packets and acknowledges all packets that arrive.

Since it is a network connection between multiple ends, there will be some congestion occurring during data transfer since there can be multiple programs sending and / or receiving data at the same time. To control these congestions, Cubic and Compound TCP are 2 of the methods used.

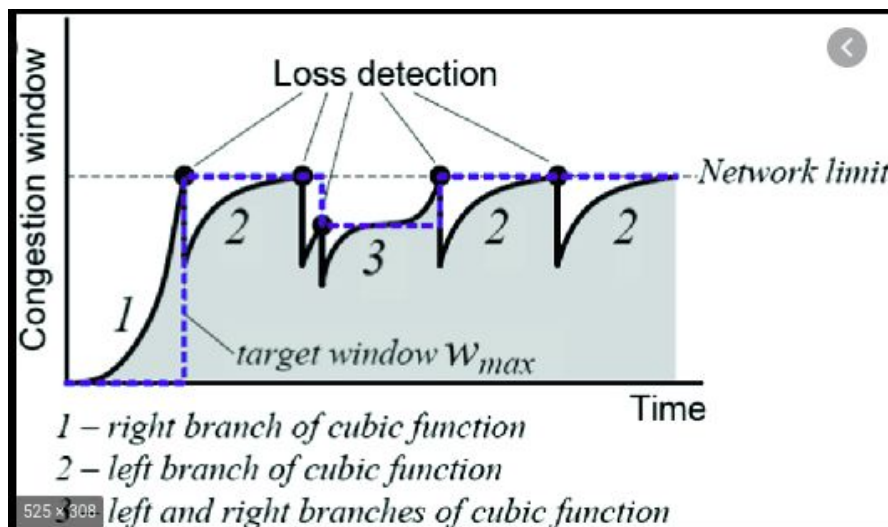
Part (i):

CUBIC TCP : CUBIC is a network congestion avoidance algorithm for TCP which can achieve high bandwidth connections over networks more quickly and reliably in the face of high latency than earlier algorithms. It helps optimize long fat networks.



In Cubic TCP, the window size is a cubic function of time since the last congestion event, with the inflection point set to the window size prior to the

event. Because it is a cubic function, there are two components to window growth. The first is a concave portion where the window size quickly ramps up to the size before the last congestion event. Next is the convex growth where CUBIC probes for more bandwidth, slowly at first then very rapidly. CUBIC spends a lot of time at a plateau between the concave and convex growth region which allows the network to stabilize before CUBIC begins looking for more bandwidth. Such nature of CUBIC improves the algorithm stability while maintaining high network utilization.



After a window reduction in response to a congestion event is detected by duplicate ACKs or Explicit Congestion Notification-Echo (ECN-Echo) ACKs, CUBIC registers the congestion window size where it got the congestion event as W_{max} and performs a multiplicative decrease of congestion window (by a factor of 0.7). After it enters into congestion avoidance, it starts to increase the congestion window using the concave profile of the cubic function. The cubic function is set to have its plateau at W_{max} so that the concave window increase continues until the window size becomes W_{max} . After that, the cubic function turns into a convex profile and the convex window increase begins.

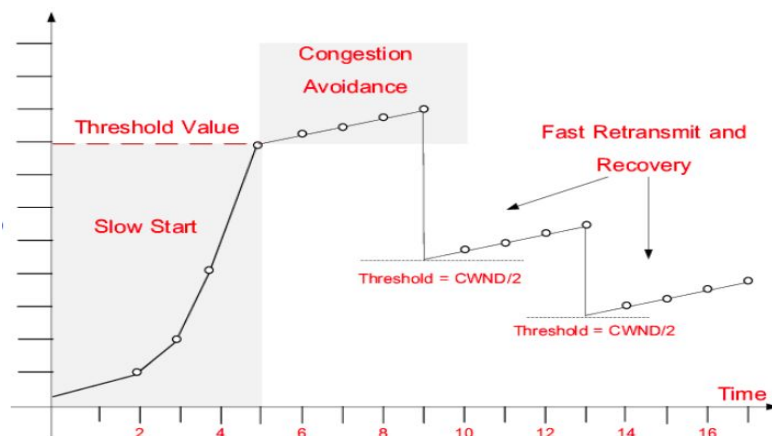
Thus, the aggressiveness of Cubic mainly depends on the maximum window size before a window reduction, which is smaller in small BDP networks than in large BDP networks. Thus, Cubic increases its congestion window less aggressively in small BDP networks than in large BDP networks. Furthermore, in cases when the cubic function of Cubic increases its congestion window less aggressively than

Standard TCP, Cubic simply follows the window size of Standard TCP to ensure that Cubic achieves at least the same throughput as Standard TCP in small BDP networks.

Compound TCP : Compound TCP is designed to aggressively adjust the sender's congestion window to optimise TCP for connections with large bandwidth-delay products while trying not to harm fairness.

Compound TCP uses estimates of queuing delay as a measure of congestion; if the queuing delay is small, it assumes that no links on its path are congested, and rapidly increases its rate, it does not seek to maintain a constant number of packets queued.

Compound TCP maintains two congestion windows: a regular AIMD window and a delay-based window (where AIMD is the additive-increase / multiplicative-decrease algorithm which is a feedback control algorithm best known for its use in TCP congestion control. AIMD combines linear growth of the congestion window with an exponential reduction when congestion is detected. Multiple flows using AIMD congestion control will eventually converge to use equal amounts of a shared link.) The size of the actual sliding window used is the sum of these two windows. If the delay is small, the delay-based window increases rapidly to improve the utilisation of the network. Once queuing is experienced, the delay window gradually decreases to compensate for the increase in the AIMD window. The aim is to keep their sum approximately constant, at what the algorithm estimates is the path's bandwidth-delay product. In particular, when queuing is detected, the delay-based window is reduced by the estimated queue size to avoid the problem of "persistent congestion"



Similarities between CUBIC and Compound TCP :

1. Both were created to have better congestion management and counter the problem of low utilization of TCP in fast long-distance networks.
2. Both are much more aggressive methods than other normally used TCP methods.
3. They can become unfair to cross-traffic due to their highly aggressive nature.

Differences between CUBIC and Compound TCP:

1. Cubic has both concave and convex parts of the graph, while Compound only uses convex and linear.
2. Compound uses the delay component to predict congestion and acts accordingly, whereas no such method is used in Cubic.
3. Both use distinct methods to decide its aggressive nature.

Part (ii) : Fairness of TCP

Fairness measures or metrics are used in network engineering to determine whether users or applications are receiving a fair share of system resources.

There are several mathematical and conceptual definitions of fairness.

Congestion control mechanisms for new network transmission protocols or peer-to-peer applications must interact well with TCP. TCP fairness requires that a new protocol receive no larger share of the network than a comparable TCP flow. This is important as TCP is the dominant transport protocol on the Internet, and if new protocols acquire unfair capacity they tend to cause problems such as congestion collapse. TCP throughput unfairness over WiFi is a critical problem and needs further investigations.

Thus, we consider that the network behavior is fair at the session-level of a given flow when it is friendly with all other competing flows while the totality of slots are separately analyzed. In other words, a session that would obtain much more resources than other flows for some slots but compensate them by getting less resources for some other slots is not fair since this behavior may jeopardize the access to the services for the competing flows. A few fairness indices are defined to decide if a certain TCP is fair or not, one of them is:

Jain's Fairness Index:

$$\mathcal{J}(x_1, x_2, \dots, x_n) = \frac{(\sum_{i=1}^n x_i)^2}{n \cdot \sum_{i=1}^n x_i^2} = \frac{\bar{\mathbf{x}}^2}{\mathbf{x}^2} = \frac{1}{1 + \widehat{c_v}^2}$$

Raj Jain's equation rates the fairness of a set of values where there are n users, x_i is the throughput for the i th connection, and $\widehat{c_v}$ is the sample coefficient of variation. The result ranges from $1/n$ (worst case) to 1 (best case), and it is maximum when all users receive the same allocation. This index is n/k when k users equally share the resource, and the other $n - k$ users receive zero allocation.

Part (iii) Possible suggestions for a better TCP :

1. If one remembers, in compound TCP, the sliding window was a combination of 2 windows, but the aim of the purpose was to always balance out the sum of the 2 windows, the delay window and the AIMD window, now since this is the case and data transfer is more or less continuous and with very less changes except for the sudden changes (for example the data usage when a user is dormant is very low, while using a texting app like whatsapp is medium, while using apps like youtube it is again a more or less constant but much higher value) . Because of this it may be a reasonable idea to fix the size of the sliding window except for these cases of special sudden changes in data transfer when congestion can be handled specifically on encounter.
2. The general idea of the CUBIC was that it has a concave and a convex nature of graph while encountering congestion, using a cubic function for this purpose need not necessarily be the best idea, we can basically use any function or combination of functions which give a similar nature graph in the congestion window, and we can use better, more suitable concave or convex functions to deal with congestion.

Qs 8) Algorithms for “Search” (i) Describe the algorithmic components of PageRank, which is the search algorithm used by Google. (ii) Outline and describe the algorithms used for searching in video sharing sites, like

YouTube. (iii) Describe how you might design a framework for searching on youtube?

I will describe PageRank with mathematical parts too on how it works, for video sharing sites like youtube I will give a basic outline or idea as to how their search engines work with some example.

Part (i) Small brief : PageRank is the first, most oldest algorithm used by the google search engine to rank the web pages when any search is made using the engine, although there are more algorithms that are used too now. It is named after Larry Page, its founder. It is a way of measuring the importance of web pages. According to Google : PageRank works by counting the number and quality of links to a page to determine a rough estimate of how important the website is. The underlying assumption is that more important websites are likely to receive more links from other websites.

Description: PageRank is a link analysis algorithm and it assigns a numerical weighting to each element of a hyperlinked set of documents, such as the World Wide Web, with the purpose of "measuring" its relative importance within the set. The algorithm may be applied to any collection of entities with reciprocal quotations and references. The numerical weight that it assigns to any given element E is referred to as the *PageRank of E* and denoted by $PR(E)$.

A PageRank results from a mathematical algorithm based on the webgraph, created by all World Wide Web pages as nodes and hyperlinks as edges. As one can imagine, a page that has many incoming links (i.e hyperlinked in many other pages) will have a high PageRank, and similarly hyperlinking any link on a page is like a vote of support, it increases the PageRank of the linked page..

Mathematics : PageRank algorithm outputs a probability distribution. Since it is a probability distribution, as one can guess, $\sum PR(u) = 1$. When some query is searched, all pages will be shown in decreasing order of their $PR(u)$ values for the query.

To find the PageRank of any webpage the following approach is used.

Assume a small universe with only 5 webpages, then if no page links to any other page, $PR(E)$ will be initialized at 0.2 for each of the webpages.

Note that:

1. Links to itself are not counted.
2. PageRank of a page **does not depend on how many times a page is linked**, it will count that as **linked once only**. So suppose there are distinct n such outgoing links from a page (i.e it hyperlinks n different pages) then it adds to PageRank of each page by $1/n$ th of its PageRank (since the probability that a random user goes to any link is $1/n$)

If A had 1 outgoing link to D; B had 3 outgoing links to A,C,E ; C had 2 outgoing links to B,E ; D had 0 outgoing links and E had 2 outgoing links to A,B then, each value is initialised at 0.2, and at each iteration, all the values are updated according this formula and we keep iterating till each value almost converges.

$$PR(A) = \frac{PR(B)}{3} + 0 \times PR(C) + 0 \times PR(D) + \frac{PR(E)}{2}$$

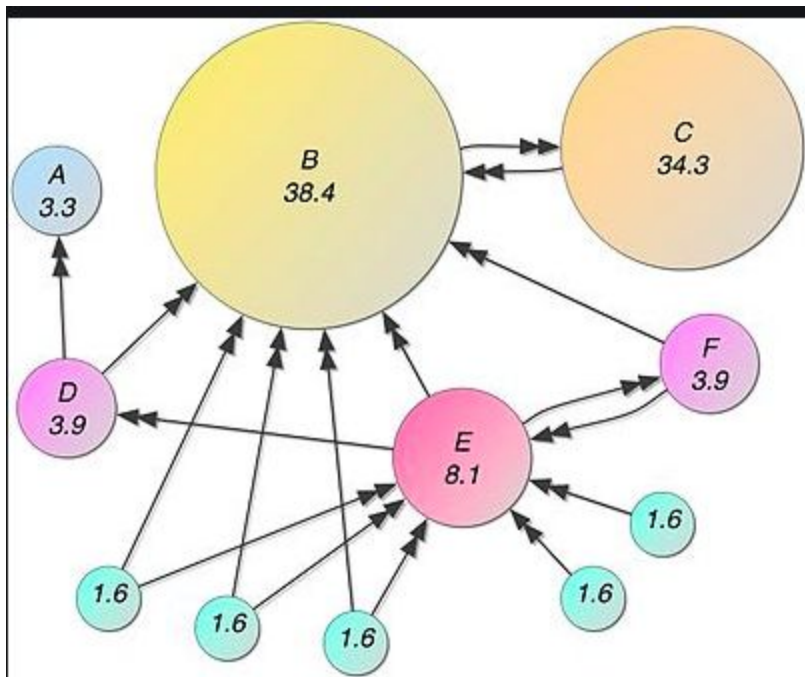
$$PR(B) = 0 \times PR(A) + \frac{PR(C)}{2} + 0 \times PR(D) + \frac{PR(E)}{2}$$

$$PR(C) = \frac{PR(B)}{3} + 0 \times PR(A) + 0 \times PR(D) + 0 \times PR(E)$$

$$PR(D) = 1 \times PR(A) + 0 \times PR(B) + 0 \times PR(C) + 0 \times PR(E)$$

$$PR(E) = 0 \times PR(A) + \frac{PR(B)}{3} + \frac{PR(C)}{2} + 0 \times PR(D)$$

A general illustration of an example with PageRanks denoted in % value is :



The arrows indicate outgoing links from a page to another

In more general terms, if a page u is linked in each of the pages in set Bu then the formula to update $PR(u)$ at each iteration is $\sum_{v \in Bu} \frac{PR(v)}{L(v)}$ where $PR(v)$ is the PageRank of each respective page v in the set Bu as found in previous iteration and $L(v)$ is the total number of outgoing links from page v .

While doing the above, we did not consider the fact that a random user browsing can stop clicking on links at any time, so let us assume that the probability that the user continues clicking on links at any time is d (damping factor). So this will change the above formula slightly, now we will have a term of $\frac{1-d}{N}$ where N is the number of documents or pages in the collection of all pages.

$$PR(u) = \frac{1-d}{N} + d \times \left(\sum_{v \in Bu} \frac{PR(v)}{L(v)} \right), \text{ where the notation is the same as before.}$$

Now, we have to iterate through this formula till we get the final converged values of PageRank for each of the pages.

To do this, we can use an iterative approach where we calculate till each converges within some max error or we can use a matrix approach.

Iterative approach is pretty straight-forward on how to use it, since we just have to decide a max error between $PR(u, t)$ and $PR(u, t+1)$ and once all $PR(u)$ will fall under this condition it will give a pretty good approximation of PageRank values of each page, setting a smaller error will increase the accuracy but decrease the time efficiency, so one has to choose accordingly.

Matrix approach :

$$\mathbf{R} = \begin{bmatrix} PR(p_1) \\ PR(p_2) \\ \vdots \\ PR(p_N) \end{bmatrix}$$

where \mathbf{R} is the solution of the equation

$$\mathbf{R} = \begin{bmatrix} (1-d)/N \\ (1-d)/N \\ \vdots \\ (1-d)/N \end{bmatrix} + d \begin{bmatrix} \ell(p_1, p_1) & \ell(p_1, p_2) & \cdots & \ell(p_1, p_N) \\ \ell(p_2, p_1) & \ddots & & \vdots \\ \vdots & & \ell(p_i, p_j) & \\ \ell(p_N, p_1) & \cdots & & \ell(p_N, p_N) \end{bmatrix} \mathbf{R}$$

where the adjacency function $l(p_i, p_j)$ is the ratio between number of links outbound from page j to page i to the total number of outbound links of page j . The adjacency function is 0 if page p_i does not link to p_j , and normalized such

that, for each j $\sum_{i=1}^N l(p_i, p_j) = 1$

To compute R , we have the following relation :

$$\mathbf{R}(t+1) = d\mathcal{M}\mathbf{R}(t) + \frac{1-d}{N}\mathbf{1},$$

where $\mathbf{R}_i(t) = PR(p_i; t)$ and $\mathbf{1}$ is the column vector of length N containing only ones.

The matrix \mathcal{M} is defined as

$$\mathcal{M}_{ij} = \begin{cases} 1/L(p_j), & \text{if } j \text{ links to } i \\ 0, & \text{otherwise} \end{cases}$$

Now, in the limiting case as $t \rightarrow \infty$, we have $R = d \cdot M \times R + \frac{1-d}{N} \cdot 1$

Therefore, by taking inverse of matrix we can have final value of R as :

$$R = (I - d \cdot M)^{-1} \times \frac{1-d}{N} \cdot 1.$$

Part (ii) : Clearly, Video sharing sites like Youtube cannot use an algorithm exactly like PageRank since very few videos link to other videos although it can be modified and given some weightage in the final algorithm used to give search results. Such sites take the following things into consideration while giving search results :

1. Relevance : This points out factors such as the title, tags, description, and video content itself.
2. Quality : For quality, the systems are designed to identify signals that can help determine which channels demonstrate expertise, authoritativeness, and trustworthiness on a given topic.
3. Engagement : The search algorithm incorporates aggregate engagement signals from users, such as the watch time of a particular video for a particular query. It notes that engagement signals are a valuable way to determine relevance as well. (Relevance as in for some user who watches lots of food cooking videos, when “chicken” is searched will show cooking videos related to chicken dishes, but for a user who watches lot of animal

breeding videos it will show videos related to how to feed chicken or how to keep them healthy)

4. Credibility : When a search related to politics or current affairs, the credibility of the source of the news also matters so it will rank such search also according to the credibility.

Such sites also have recommendations, these are based on the videos a user watches and similar videos watched by other users who also watched the videos in your history, these will be again ranked according to what the user watches most and what other similar users watched most after also having watched the videos in your history.

Part (iii) Framework for searching on Youtube :

There are 2 type of users on Youtube:

While operating Youtube for users who have already used it in the past and / or use Youtube consistently, search results should be given more on the basis of relevance, i.e using the history of the user and what type of videos the user prefers to watch, which includes factors like length of the video, genre of the video, channels to which the user has subscribed, videos of channels which the user frequently watches, etc. If this is taken care of mostly Youtube gives good search results for its old users, because user history is probably the most important factor in deciding what to show up for a user when the user searches something to make sure that the user spends time watching videos on youtube. For example if a user watches complete videos of a certain chess channel but leaves videos unfinished of some other one then on searching chess, it should show up the channel with completed videos on top. It should also use the recommender system, i.e on searching any keyword, give results of videos most watched by similar users on searching similar or same keyword. These are the factors that should be given the most weightage while giving search results for an old user.

For users who are new or have very less search history, it is difficult even for youtube to decide what the user wants to watch to give search results, in such cases it should give more weightage to factors like credibility, total number of views, the type of content (adult rated / child rated for example) i.e the content should be neutral wrt to any watcher, if the user submits a basic form of types in

which the user specifies his likes and dislikes it will make the job of the search engine easier, but it is also a privacy issue, so will not get into that. But essentially for new users, it should give neutral content with high views and good credibility when a keyword is searched.