EE4371 - End-Sem Assignment

EE19B094

Manvar Nisharg

Question 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...200.

Answer 1

In [ ]:

```
import random   #Importing necessary files

#PURPOSE : Calculating theoritical probability for the Birthday Problem (i.e. two people have same birthdays) with n people
#PARAMETERS : Number of people (n)
#RETURN VALUE : Calculated theorirical probability
#CONSOLE INPUT : Null
#CONSOLE OUTPUT : Null
#NOTE : This function is called by Birthday_Problem() function
def theoretical_probability(n):
        prob=1  #variable to calculate probability that NO two people have same birthdays
        for x in range(n):
                prob *= ((365-x)/365)   #Calculating Probability that NO two people have same birthdays (Refer to PDF attached for further explanations)
        return float(1-prob)    #Return 1-(probabilty of no two poersons have same birthday) i.e. Atleast two persons have same Birthday



#PURPOSE : Calulate experimental probability for the Birthday Problems (i.e. two people have same birthdays) with n people
#PARAMETERS : Number of people (n) , Numbers of times the experiment to be run and taken average of for a particular n (iterations)
#RETURN VALUE : Calulated experimental probability
#CONSOLE INPUT : Null
#CONSOLE OUTPUT ; Null
#NOTE : This function is called by Birthday_Problem() function
def experimental_probability(n,iterations):
        count=0         #Variable to calculate how many cases have two persons with same birthday
        for x in range(iterations):     #Running the experiment 'interation' number of times
                date_count = {}         #Initalsing a empty dictionary for storing dates as key and the number of people having birthday as that date as value
                for y in range(n):      #Calculating birthday for each person
                        rand_num = random.randint(1,365)        #Creating a random birthday
                        if rand_num in date_count:      #If date is already present in date_count dictionary increase count by 1 and break
                                count += 1
                                break
                        else:                                           #Else add the date in the dictionay
                                date_count[rand_num]=1
        return float(count)/float(iterations)   #Return calculated expeimental probability by dividing number of cases having two people with same birthdays with total number of cases



#Q6
#PURPOSE : Main function which calls other function to calculate theoritical and experimental probability for values of n given in the question
#PARAMETERS : Null
#RETURN VALUE : Null
#CONSOLE INPUT : Numbers of times the experiment to be run and taken average of for a particular n (iterations)
#CONSOLE OUTPUT : Printing the experimental and theoritical probability for each n
def Birthday_Problem():
        print("To test the Birthday Problem")
        iterations = int(input("Enter the number of times to run the experiment for a specifc n : "))   #Taking iterations as input from user
        print("iterations = %d"%iterations)     #printing iterations
```

```
        for x in range(5,201,5):        #Iterating the values of n
                print("n=%d experimental=%lf%% theoritical=%lf%%"%(x,float(experimental_probability(x,iterations))*100,theoretical_probability(x)*100)) #Printing theoritical and experiment

print() #Printing Newline

Birthday_Problem()
```

Calculating theoritical probability

Ignoring special cases as a leap year, twins, etc and 365 days are equally likely for birthday. We can calculate the theoretical probability that no two people have the same birthdays using Conditional Probability. I.e. for ith person his/her birthday is not the same to any of the 1 to (i-1)th persons.

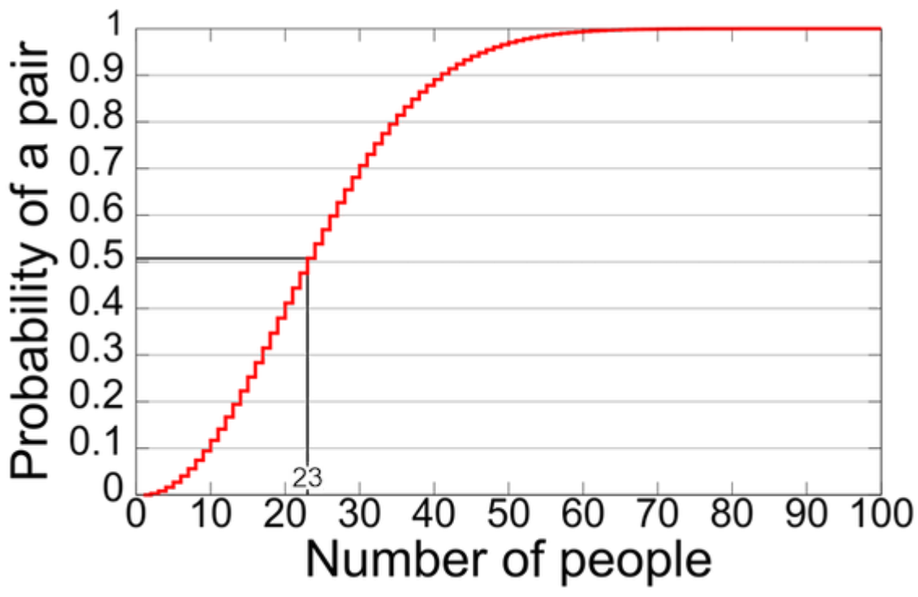
$$\bar{p}(n) = 1 \times \left(1 - \frac{1}{365}\right) \times \left(1 - \frac{2}{365}\right) \times \cdots \times \left(1 - \frac{n-1}{365}\right)$$

$$= \frac{365 \times 364 \times \cdots \times (365 - n + 1)}{365^n}$$

$$= \frac{365!}{365^n (365 - n)!} = \frac{n! \cdot \binom{365}{n}}{365^n} = \frac{{}_{365}P_n}{365^n}$$

Here p is the probability that NO two people have the same birthdays. Therefore (1-p) gives us the probability of there exists at least two people with the same birthday.

Graph of Theoritical Probability vs. No of people



Calculating Experimental probability

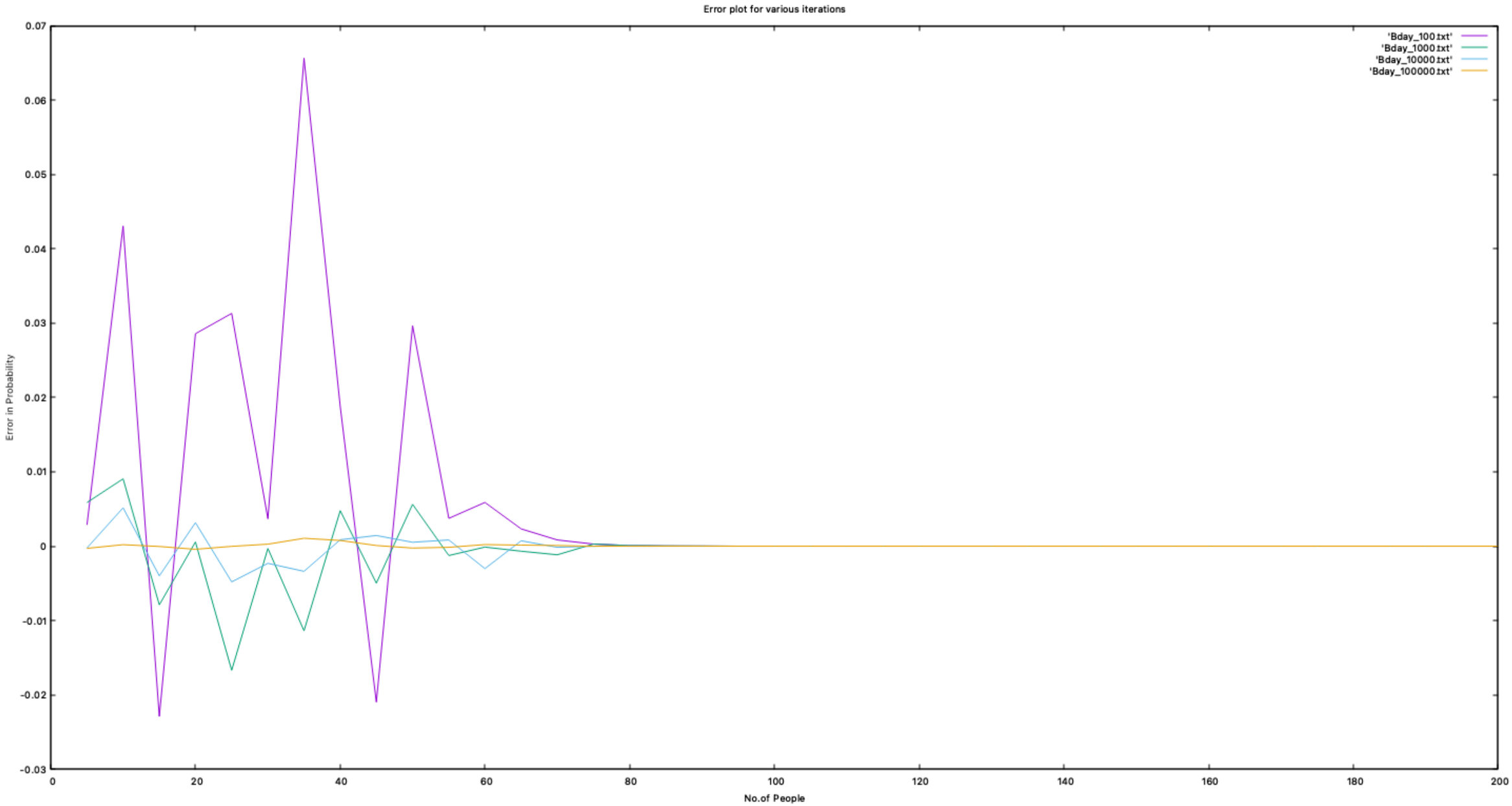
The basic idea is to create random birthdays for n people and check if any two people have the same birthday.

This is implemented using a dictionary which contains the birthday dates of all people encountered so far. The dictionary is updated after every random birthday is generated. If a birthday is already present in the dictionary break the loop and increase the count of experiments where two people have the same birthday (count) by 1.

Repeat the experiment ‘iteration’ number of times where iteration will be taken as input from the user. More the value of ‘iteration’ more accurate will be the result by the code will take more time to execute. Finally, the experimental probability will be calculated as

$$Prob = \frac{count}{iterations}$$

Graphs of error in experimental probability for various iterations :



Pink : Iterations = 100

Green : Iterations = 1000

Blue : Iterations = 10000
Yellow : Iterations = 100000

Time Complexity

1) $\underline{Theoretical\ Probability}$: $O(n)$ as we loop through n numbers and multiply them.

2) $\underline{Experimental\ Probability}$: For a particular experiment, we loop the process of creating a random number ($O(1)$) and checking and updating dictionary ($O(1)$ as a dictionary is basically a hash table) n times. Therefore $O(n)$ for a particular experiment. We then repeat the experiment 'iterations' number of times. Therefore the overall time complexity will be $O(n \cdot iterations)$.

Therefore for a particular n calculating theoretical and experimental probability takes

$$time \ = \ O(n \cdot iterations)$$

Question 2

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

	$f(x)$	$g(x)$
(a)	$n - 100$	$n - 200$
(b)	$100n + logn$	$n + (logn)^2$
(c)	$log2n$	$log3n$
(d)	$n \cdot 2^n$	3^n

Answer 2

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

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

a)

$f(n) = n - 100 \; and \; g(n) = n - 200$

$Taking \; c = 1 \; and \; n_o = 200 \; we \; get,$

$n - 100 \; > \; 1 * (n - 200) \; for \; all \; n > 200$

$\Rightarrow -100 \; > \; -200 \quad which \; is \; true \; for \; given \; conditions.$

$Therefore, \; f = \Omega(g)$

$Similarly \; taking \; c = 2 \; and \; n_o = 300 \; we \; get,$

$n - 100 \; < \; 2 * (n - 200) \; for \; all \; n > 300$

$\Rightarrow 300 \; < \; n \quad which \; is \; true \; for \; given \; conditions.$

$Therefore, \; f = O(g)$

$Therefore, f = \Theta(g)$

b)

$f(n) = 100n + logn \; and \; g(n) = n + (logn)^2 \qquad (Assuming \; base \; to \; be \; 2)$

$Taking \; c = 1 \; and \; n_o = 1 \; we \; get,$

$100n + logn \; > \; n + (logn)^2 \;\; for \;\; all \;\; n > 1$

$\Rightarrow 99n \; > \; (logn)^2 - logn$

$As, \; n \; > \; (logn)^2 \; for \; n > 1,$

$\Rightarrow 99n \; > \; (logn)^2 - logn \quad which \; is \; true \; for \; given \; conditions.$

$Therefore \; f = \Omega(g)$

$Similarly, \; taking \; c = 200 \; and \; n_o = 2 \; we \; get,$

$100n + logn \; < \; 200 * (n + (logn)^2) \; for \; all \; n > 2$

$As, \; 100n \; < \; 200n \; and \; logn \; < \; (logn)^2 \; for \; n > 2$

$\Rightarrow 100n \; < \; 200n + 200(logn)^2 - logn \quad which \; is \; true \; for \; given \; conditions.$

$Therefore \; f = O(g)$

$Therefore \; f = \Theta(g)$

c)

$f(n) = log2n \; and \; g(n) = log3n \qquad (Assuming \; base \; to \; be \; 2)$

$Taking \; c = 0.1 \; and \; n_0 = 1 \; we \; get,$

$log(2n) > 0.1 * log(3n) \; for \; all \; n > 1$

$\Rightarrow 0.9 * logn \; > \; 0.1 * log3 - log2$

$\Rightarrow n > 0.533 \quad which \; is \; true \; for \; given \; conditions.$

$Therefore \; f = \Omega(g)$

$Similarly, \; taking \; c = 1.1 \; and \; n_0 = 1 \; we \; get,$

$log(2n) < 1.1 * log(3n) \; for \; all \; n > 1$

$\Rightarrow log2 - 1.1 * log3 \; < \; 0.1 * logn$

$\Rightarrow n \; > \; 0.0058 \quad which \; is \; true \; for \; given \; conditions.$

$Therefore \; f = O(g)$

$Therefore \; f = \Theta(g)$

d)

$f(x) = n \cdot 2^n \; and \; g(x) = 3^n$

$Taking \; c = 1 \; and \; n_0 = 1 \; we \; get,$

$n \cdot 2^n < 3^n \; for \; all \; n > 1$

$\Rightarrow (3/2)^n \; > \; n$

$\Rightarrow log(3/2) \cdot n > log(n) \quad which \; is \; true \; for \; given \; conditions. \; (Assume \; base \; of \; log \; to \; be \; 2.)$

$Therefore \; f = O(g)$

Question 3

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

Answer 3

We need to prove that $T(n) \; \leq \; k_1 \cdot nlogn \; + \; k_2 \cdot n$

We will use the method of induction.

The recursive formula for merge sort is,

$$T(n) \; = \; T([n/2]) \; + \; T(n - [n/2]) \; + \; k \cdot n \quad where \; k \; is \; a \; computational \; constant.$$

As the base case to induction, let $n = 1 \implies T(1) = c$

Now, consider two cases

Case 1: n is even

$$\implies T(n) = 2 \cdot T(n/2) + k \cdot n$$
$$\implies T(n) \leq 2 \cdot [k_1 \cdot (n/2) \cdot log(n/2) + k_2 \cdot (n/2)] + k \cdot n$$
$$\implies T(n) \leq k_1 \cdot nlogn + (k_2 + k - k_1 log2) \cdot n$$
$$\implies T(n) \leq c_1 \cdot nlogn + c_2 \cdot n$$
$$Hence\ proved.$$

Case 2: n is odd

$$\implies T(n) = T((n-1)/2) + T((n+1)/2) + k \cdot n$$
$$\implies T(n) \leq [k_1 \cdot (n-1/2) \cdot log(n-1/2) + k_2 \cdot (n-1/2)] + [k_1 \cdot (n+1/2) \cdot log(n+1/2) + k_2 \cdot (n+1/2)] + k \cdot n$$
$$\implies T(n) \leq k_1 \cdot [(n-1)/2 \cdot log(n-1) + (n+1)/2 \cdot logn(n+1)] - k_1 \cdot nlog2 + k_2 \cdot n + k \cdot n$$

Thus we need a constant c_1 such that,

$$(n-1)/2 \cdot log(n-1) + (n+1)/2 \cdot log(n+1) \leq c_1 \cdot nlogn$$
$$\implies ((n-1)/n)^{n-1} \cdot ((n+1)/n)^{n+1} \leq n^{2n(c_1-1)}$$

Plotting LHS we see that $LHS \leq 4$. So, in our case where we have $n > 2$,

$$we\ need\ 4 \leq n^{2n(c_1-1)}$$
$$\implies c_1 = 2\ will\ satisfy\ the\ condition.$$
$$\therefore for\ c_1 = 2 \implies T(n) \leq 2 \cdot k_1 \cdot nlogn + (k_2 + k - k_1 log2) \cdot n$$
$$\implies T(n) \leq p_1 \cdot nlogn + p_2 \cdot n$$
$$Hence\ proved$$

Question 4

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

Answer 4

There are two ways to implement a stack using two queues.

Both methods have a cheap operation and a costly operation.

Method 1 : Costly $pop()$

In [ ]:
```
#Importing queue header files
from queue import Queue

#Class to implement stack using queue
class stack_using_queue:

        #Initilastion function for the class
        def __init__(self):
                self.Q1 = Queue()       #Queue number 1 (active queue)
                self.Q2 = Queue()       #Queue number 2

        #Function to push into the stack
        def push(self,value):
                self.Q1.put(value)      #Push number into the active queue

        #Function to pop from the stack
        def pop(self):

                #Return if stack is empty
                if self.Q1.empty():
                        return

                #Transfer all but last element from active queue (Q1) to the other queue (Q2)
                while self.Q1.qsize() != 1:
                        self.Q2.put(self.Q1.get())

                #Dequeue the remaining last element effectively poping the number from the stack
                dequeued = self.Q1.get()
                print("Popped : %d"%dequeued)

                #Swaping the active queue
                self.Q_temp = self.Q1
                self.Q1 = self.Q2
                self.Q2 = self.Q_temp

        #Function to get the top element of the stack
        def get_top(self):

                #Return if stack is empty
                if self.Q1.empty():
                        return

                #Transfer all but one elment from Q1 to Q2
                while self.Q1.qsize() != 1:
                        self.Q2.put(self.Q1.get())

                #Store the last element in a variable and pop it
                top = self.Q1.get()

                #Push it back into Q2
                self.Q2.put(top)

                #Swap Q1 and Q2
```

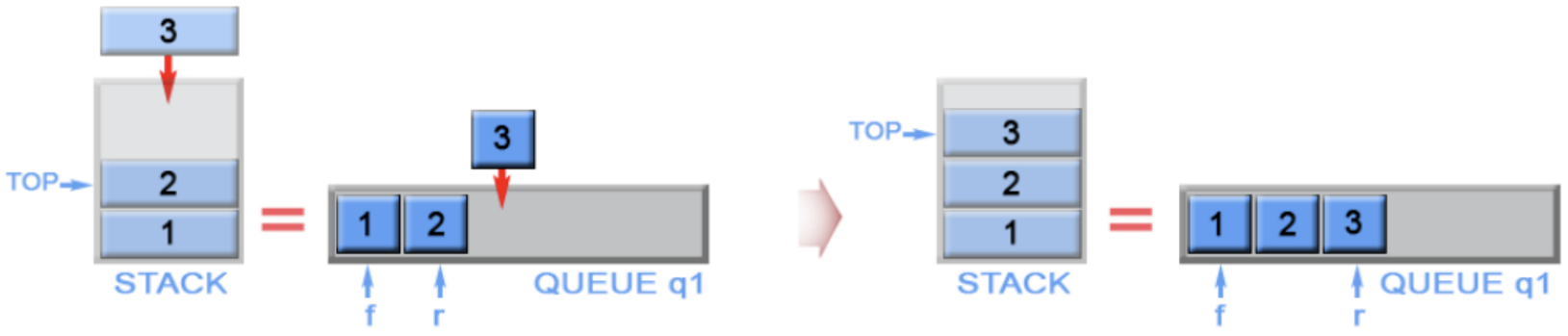
```
                self.Q_temp = self.Q1
                self.Q1 = self.Q2
                self.Q2 = self.Q_temp

                return top


#Testing
test = stack_using_queue()
test.push(1)
test.push(3)
test.push(2)
print(test.get_top())
test.pop()
print(test.get_top())
```

Output for the given code

-
```
3
Popped : 3
2
```

Pushing an element

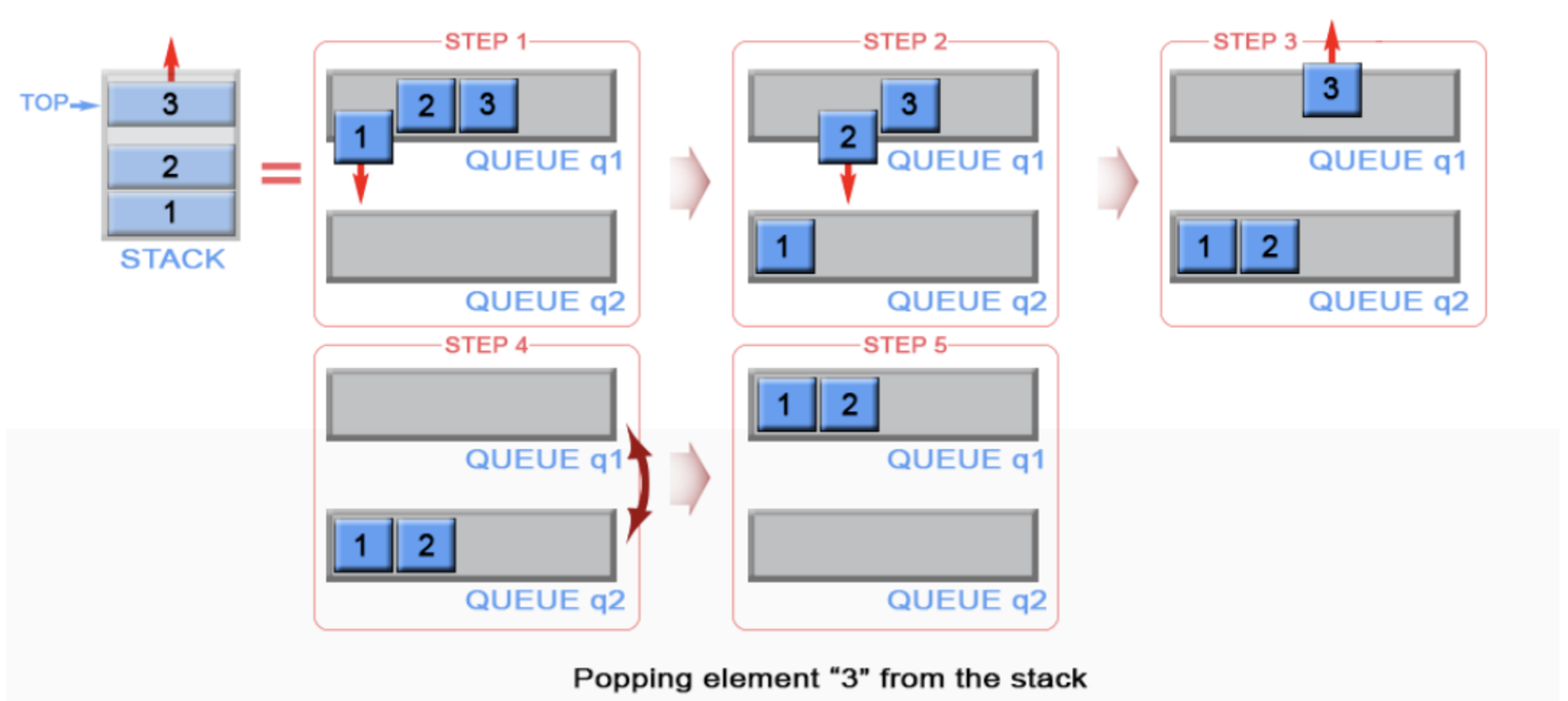
We simply push the element into Q1 (active queue)
Thus, this will take only $O(1)$ time.



Pushing element "3" to the stack

Popping an element

We first transfer all but the last element from Q1 to Q2, then dequeue the last element and then swap the Queues
Thus, clearly this will take $O(n)$ time.



Popping element "3" from the stack

Method 2 : Costly $\mathrm{push}()$

In [ ]:

```
#Importing queue header files
from queue import Queue

#Class to implement stack using two queues
class stack_using_queue:

    #Initilastion function for the class
    def __init__(self):

        self.Q1 = Queue()   #Queue number 1 (active queue)
        self.Q2 = Queue()   #Queue number 2

    #Function to push into the stack
    def push(self,value):

        self.Q2.put(value)  # Push value first in empty Q2

        #Transfer all elements from active queue (Q1) to the other queue (Q2)
        while self.Q1.qsize():
            self.Q2.put(self.Q1.get())

        #Swap the active queue
        self.Q_temp = self.Q1
        self.Q1 = self.Q2
        self.Q2 = self.Q_temp

    def pop(self):

        #Return if stack is empty
        if (self.Q1.empty()):
            return
```

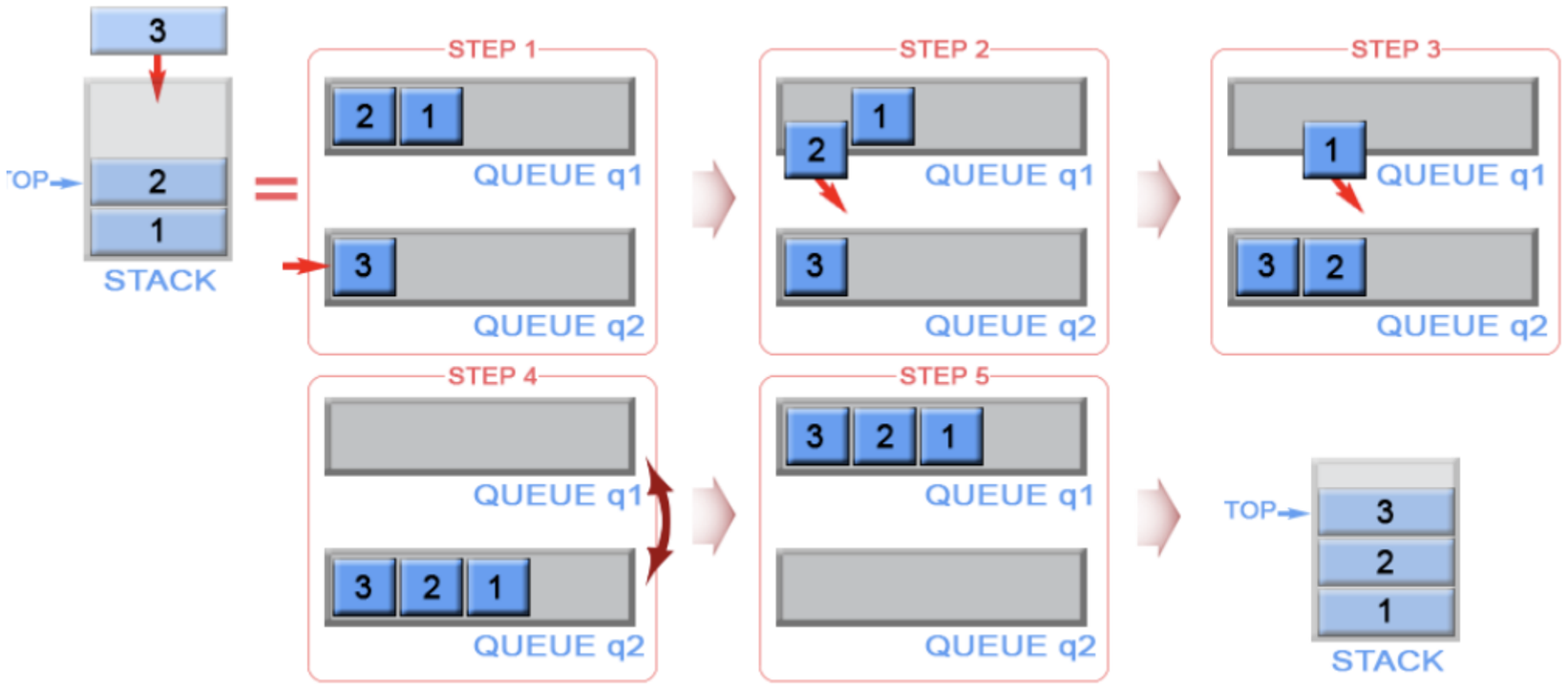
```
        #Dequeue from active queue (Q1)
        dequeued = self.Q1.get()
        print("Popped : %d"%dequeued)

    def get_top(self):

        #Return if stack is empty
        if (self.Q1.empty()):
            return -1

        #Else return the top element
        return self.Q1.queue[0]


#Testing
test = stack_using_queue()
test.push(1)
test.push(3)
test.push(2)
print(test.get_top())
test.pop()
print(test.get_top())
```

Output for the given code

```
3
Popped : 3
2
```

Pushing an element

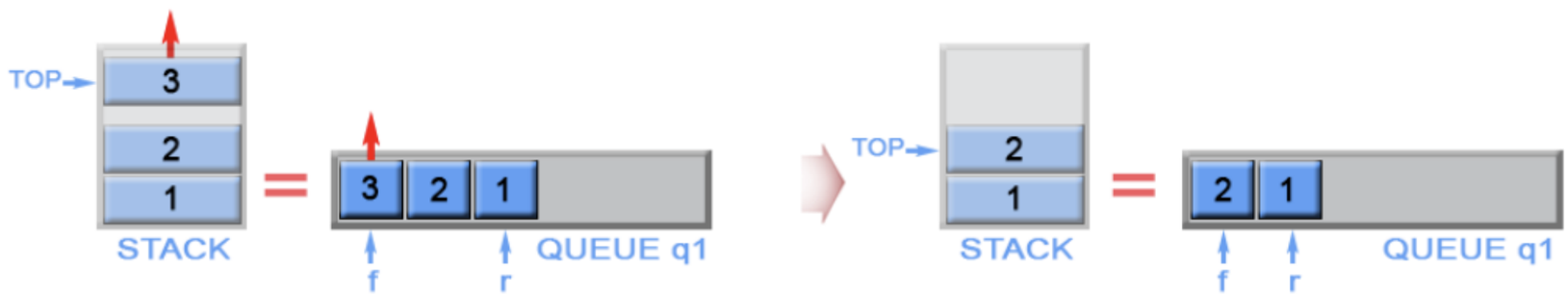
We first push the element into Q2, transfer the elements into Q1 to Q2, and lastly swap the queues.
Thus, this will take $O(n)$ time.



Pushing element "3" to the stack

Popping an element

We simply dequeue from the active queue (Q1)
Thus, clearly this will take only $O(1)$ time.



Popping element "3" from the stack

Question 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).

Answer 5

NOTE: This question can be implemented in O(n) time using python dictionaries as it uses hashing but, as the question specifically asks for O(nlogn) algorithm, it has been implemented in O(nlogn).

In [ ]:

```
"""*****************************************************************************************************
PURPOSE : Remove duplicates from a sequence of numbers (Assign-2 Q6)
AUTHOR : Manvar Nisharg (EE19B094)
INPUT : Sequence of numbers
OUTPUT : Sequence of numbers with duplicates removed in sorted and original form
NOTE : Please use python 3.x as several functions used as only compatible with python 3 or later versions
*****************************************************************************************************"""

from array import array #Importing necessary files

"""
Function to merge two sorted subarrays into one

:param arr: The array whose two subarrays are to be merged
:param type: list
:param start: The starting index in arr from where elements are to be merged(i.e. starting index of first subarray)
:param type: int
:param mid: The midpoint index in arr (i.e. index where two subarrays are to be divided)
:param type: int
:param end: The endpoint index in arr till where elements are to be merged(i.e. ending index of second subarray)
:param type: int
"""
def merge(arr, start, mid, end):
    #size of arrays to be merged
    size_left = mid - start + 1
    size_right = end - mid


    ##Initalize temporary arrays and copy data from original arrays to temporary arrays
    temp_left = arr[start : start + size_left]
    temp_right = arr[start + size_left : end+1]


    #Pointers to current elements in all three arays
    i = 0     # Initial index of first temporary array
    j = 0     # Initial index of second temporaary array
    k = start     # Initial index of merged array

    #Comparing elements and adding them into the original array until end of any one array is reached
    while i < size_left and j < size_right :
        if temp_left[i] <= temp_right[j]:
            arr[k] = temp_left[i]
            i += 1
        else:
            arr[k] = temp_right[j]
            j += 1
        k += 1

    # Copy the remaining elements of temp_left array if any, into the original array
    while i < size_left:
        arr[k] = temp_left[i]
        i += 1
        k += 1

    # Copy the remaining elements of temp_right array if any, into the original array
    while j < size_right:
        arr[k] = temp_right[j]
        j += 1
        k += 1
```

```
"""
Function to sort a array using merger sort algorithm

:param arr: The array which is to be merged
:param type: list
:param start: The index of first element to be sorted
:param type: int
:param end: The index of last element to be sorted
:param type: int
"""
def mergeSort(arr,start,end):
    if start >= end:
        return

    mid = (start+(end-1))//2

    # Dividing array into two halves and sorting both the halves first
    mergeSort(arr, start, mid)
    mergeSort(arr, mid+1, end)
    # Merging the already sorted arrays into one
    merge(arr, start, mid, end)



"""
Function to return index of a number in a sorted 2D array using binary search algorithm (w.r.t. first coloumn)

:param arr: The arr in which number is to be searched
:param type: list
:param number: The number to be searched
:param type: float

:return: index of number in array
"""
def binary_search(arr,number):
    #Base Cases
    if(len(arr)==0):
        return
    elif(len(arr)==1):
        return 0

    #Initalize pointers to beginning and end of array
    start = 0
    end = int(len(arr)-1)

    while start <= end:

        mid = int((start + end)/2)  #Middle index of active window of search

        if(arr[mid][0] == number):     #If number at mid is equal to 'number' then return mid
            return mid
        elif(arr[mid][0] > number):  #Else if number at mid is greater than 'number' then change end pointer to mid-1
            end = mid - 1
        else:    #Else if number at mid is less than 'number' then change start pointer to mid+1
            start = mid + 1

    return -1   #If number is not found return -1


"""
Call function to remove duplicates from a sequence of numbers

:input: Sequence of numbers
:output: Sequence of numbers with duplicates removed in sorted and original form
"""
def remove_duplicates():
    print("\nTo remove duplicates from a sequence of numbers :")
```

```
sequence = input("Enter sequence of numbers seperated by space : ")     #Storing sequence of numbers as string (sequence)
list_of_numbers_original = sequence.split()  #Splitting sequence string into individual numbers with space as delimitter in a list (list_of_numbers_original)

if len(list_of_numbers_original)==0:
    print()
    print("No sequence was given as input.")
    return

list_of_numbers_original = list(map(float, list_of_numbers_original))   #Converting numbers stored as string to float
list_of_numbers = list_of_numbers_original.copy()   # Making a copy of original list for processing

mergeSort(list_of_numbers,0,len(list_of_numbers)-1)     # Sorting the list of numbers using merge-sort

#Initalize an empty 2D array whose first coloumn will contain all the sorted numbers once and
#second coloumn contains if that number has already been printed or not (0 for NO and 1 for YES)
#so initially 0 for all
#This array will be used later to print sequence in original order
binary_array_for_numbers = []



#Prining sorted list so that one number is printed only once

print("The sequence with duplicates removed in sorted order is :")
current = None   #Initalize variable which stores value of active number


#Looping through list from index 1 to end
for i in range(0,len(list_of_numbers)):
    if list_of_numbers[i]==current:    #If number is already encountered, skip
        continue
    #else the number is encountered for the first time print it and change the value of current to that value
    print(list_of_numbers[i],end = " ")
    current = list_of_numbers[i]
    #Adding number into binary_array_for_numbers array as it is encountered for first time
        temp = []
        temp.append(list_of_numbers[i])
        temp.append(0)
        binary_array_for_numbers.append(temp)

    print()
    print()


    #Printing sequence with duplicates removed in original order

    print("The sequence with duplicates removed in original order is :")

    #Looping through the numbers in the original list given by user
    for i in range(len(list_of_numbers_original)):
        index = binary_search(binary_array_for_numbers,list_of_numbers_original[i])  #Finding index of that number in binary_array_of_numbers using binary search
        if(binary_array_for_numbers[index][1] == 1):    #If the value of second coloumn is 1 for that index that means number has alrady been printed
            continue
        else:
            print(binary_array_for_numbers[index][0],end = " ")     #Else print the number and change value of second coloumn to 1
            binary_array_for_numbers[index][1] = 1


remove_duplicates() #Calling function to remove duplicates
```

Output for some test case

- To remove duplicates from a sequence of numbers :

```
Enter sequence of numbers seperated by space : 2.1 -4.5 1.0 2.1 8.9 8.9 7.7

The sequence with duplicates removed in sorted order is :
-4.5 1.0 2.1 7.7 8.9

The sequence with duplicates removed in original order is :
2.1 -4.5 1.0 8.9 7.7
```

Algorithm

The algorithm can be divided into two parts. One is implemeted using sort by using merge sort to print the sequence in sorted order. And another is binary search to print the sequence in original order. In this we take advantage of the fact that we sorted the sequence to implement the first part.
Assumption : All the elements are floating point numbers.

$\underline{Sorted\ Order}$:

In this we use the conventional merge sort algorithm to sort the numbers first. Functions in the code to implement this are :

- **mergeSort()**: Recursive function to implement merge sort. This function divides the input array into two parts, sorts them using recursion and then finally join them using the merge() function.
- **merge()**: Function that merges two sorted arrays into one. It uses two pointer method where one pointer points to one element in each array and compare them and add the smaller number to the result array and increment that pointer untill all elements are added.

So, we first sort the list of numbers using merge sort. Now, we traverse the sorted array and print the first appearance of each number and, as the array is sorted we need to traverse only once.
Now, we already know that merge sort takes $O(nlogn)$ complexity. And then traversing the sorted array once takes $O(n)$ time. Thus, overall complexity of this part is $O(nlogn) + O(n) = \underline{O(nlogn)}$.
Also, we know merge sort takes up $\underline{O(n)}$ space complexity.

$\underline{Original\ Order}$:

In this we use binary search on the already sorted array in the first part to implement the algorithm. Only function for this part is :

- **binary_search()**: This function takes a 2D array and the number to be searched (x). It then uses conventional binary search algorithm to find the number in the first coloumn of the 2D array.
  Overview of the binary search algorithm is :
  1. Compare x with the middle element.
  2. If x matches with the middle element, we return the mid index.
  3. Else if x is greater than the mid element, then x can only lie in the right (greater) half subarray after the mid element. Then we apply the algorithm again for the right half.
  4. Else if x is smaller, the target x must lie in the left (lower) half. So we apply the algorithm for the left half.

  As we can see we reduce the size of array by half each time so binary search takes O(nlogn) time to execute.

Now to implement the algorithm we traverse the original sequence and print a number we encounter that number for the first time else skip.
To implement that we keep a 2D array (binary_array_for_numbers) whose first numbers contain all the numbers once in sorted order and second coloumn contains 0 if we haven't encountered the number yet and 1 if we have. We create this array while we traverse the sorted array in the previous part, every time we print a number in that part we also add the number in binary_array_for_numbers with initial encounter value as 0. Now, as we traverse the original sequence we use binary search to find that number in

binary_array_for_numbers and check if we have encounter it before or not. If no print it, change it's encounter value to 1 else skip the number.
As we use binary search for every number in the original sequence the algorithm takes up $\underline{O(nlogn)}$ time as well.

Also, as we use a 2D array, space complexity of this part is $\underline{O(n)}$.

So, both the part of the codes take up $O(nlogn)$ time therefore,

$$the\ overall\ time\ complexity\ of\ the\ code\ is\ \underline{O(nlogn)}$$

Also, both the part of code takes up $O(n)$ space complexity therefore,

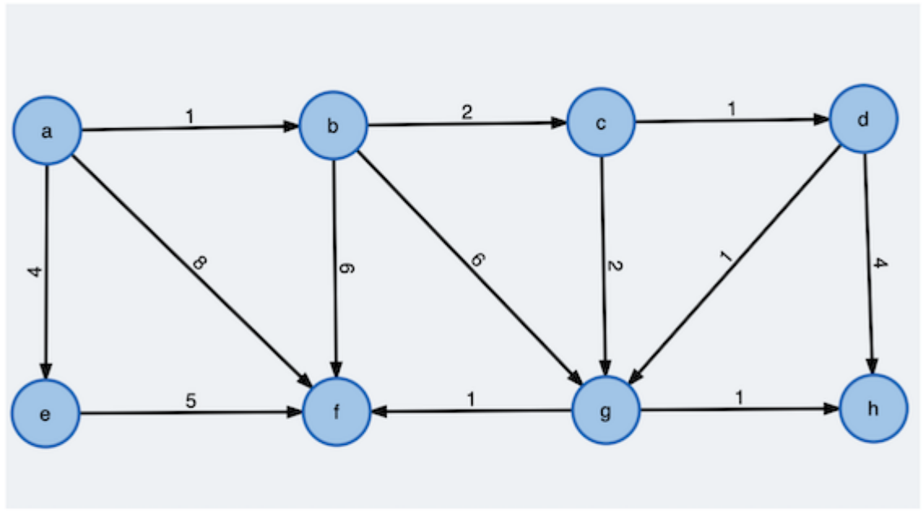
$$the\ overall\ space\ complexity\ of the\ code\ is\ \underline{O(n)}$$

Question 6

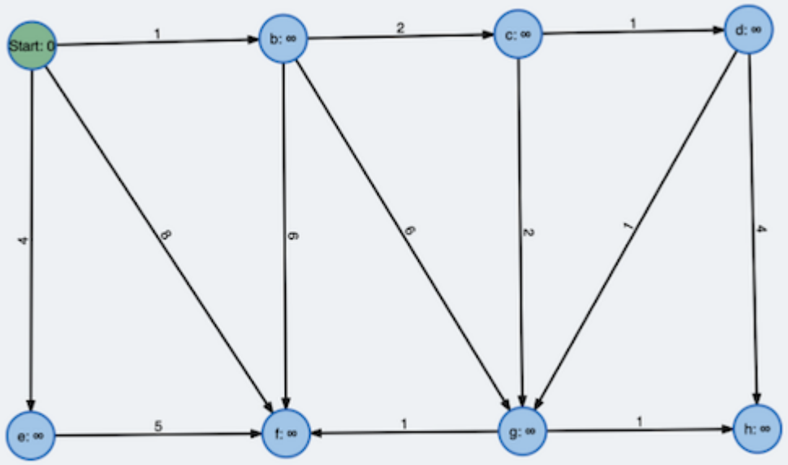
Suppose Dijkstra's algorithm is run on the following graph, starting at node A.
(a) Draw a table showing the intermediate distance values all the nodes at each iteration of the algorithm.
(b) Show the final shortest-path tree.

Answer 6

Below are a sequence of graphs following the algorithm. The values inside the node show the current value of that node.



Original Graph



Initialisation

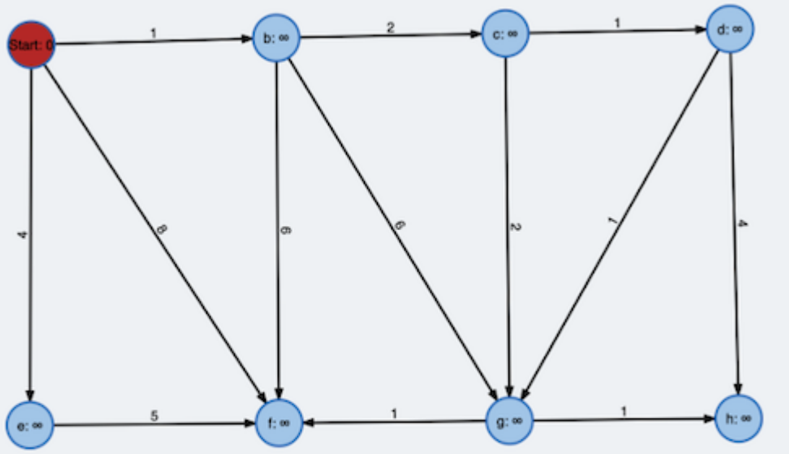
The distance from the starting node to itself is set to 0, all other nodes obtain the maximum distance infinity. The algorithm will try to reduce these distance.

The predecessor of the starting node is the node itself, it is "null", an unknown value.

The starting node is added to a priority queue. This is an ordered list, where nodes closer to the starting node appear first.

Priority Queue:

a

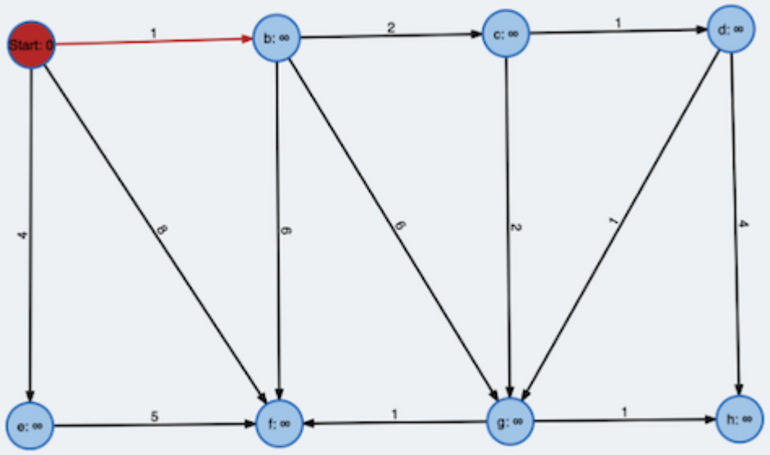


Process next node

The node with minimal distance has been extracted from the priority queue. It will be processed next and is marked in red.

Priority Queue:

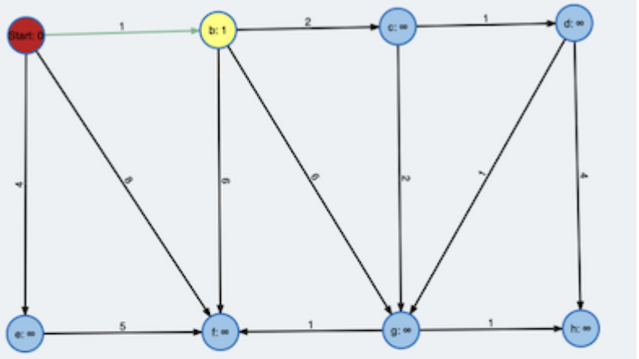
Start at node A



Inspect the neighbors of the current node.

For every neighbor, the distance to the starting node is calculated, if the path goes through the current edge.

Priority Queue:



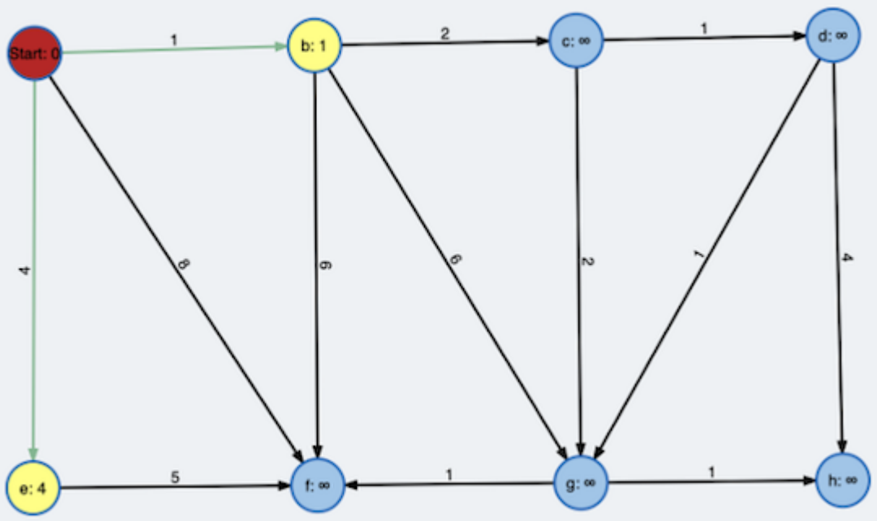
Neighboring nodes not visited

Therefore it is added to the priority queue. The new distance to the starting node is the distance of the parent plus the edge cost.

Priority Queue:

b

One neighbour visited..
Similarly, for all neighbours

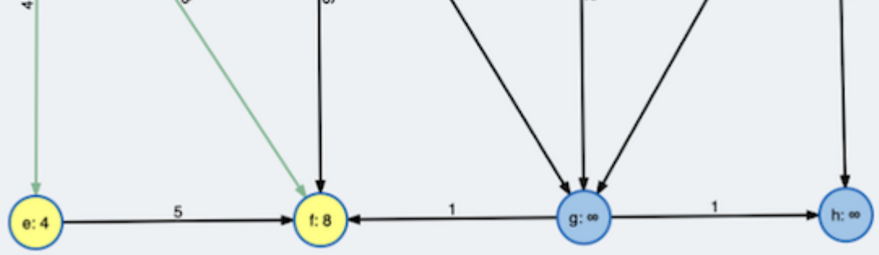


Neighboring nodes not visited

Therefore it is added to the priority queue. The new distance to the starting node is the distance of the parent plus the edge cost.

Priority Queue:

b	e

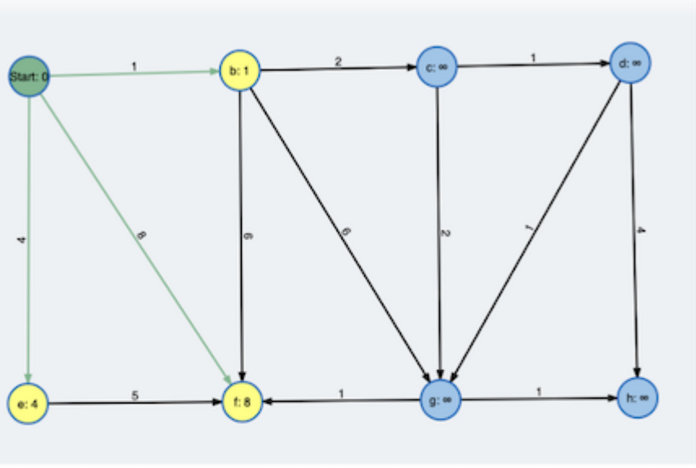


Neighboring nodes not visited

Therefore it is added to the priority queue. The new distance to the starting node is the distance of the parent plus the edge cost.

Priority Queue:

b	e	f

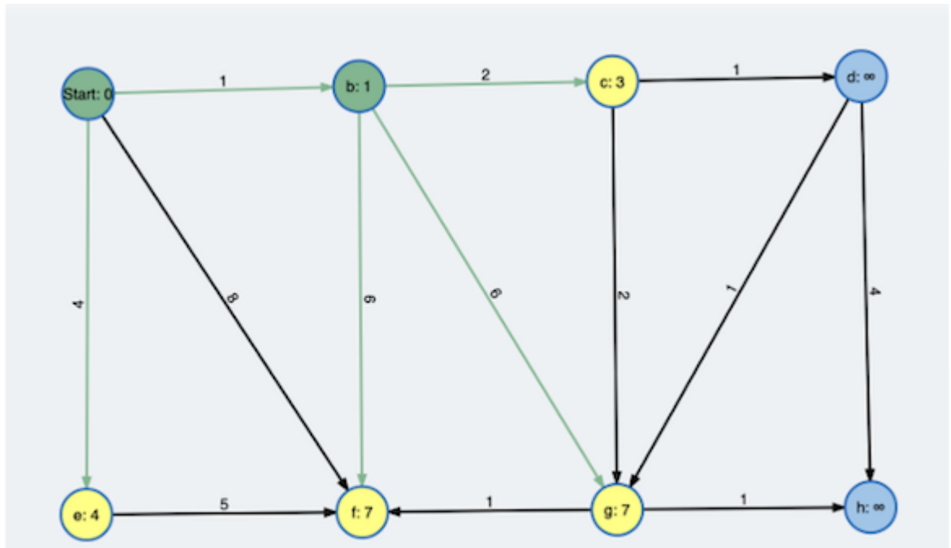


All outgoing nodes have been processed.

Priority Queue:

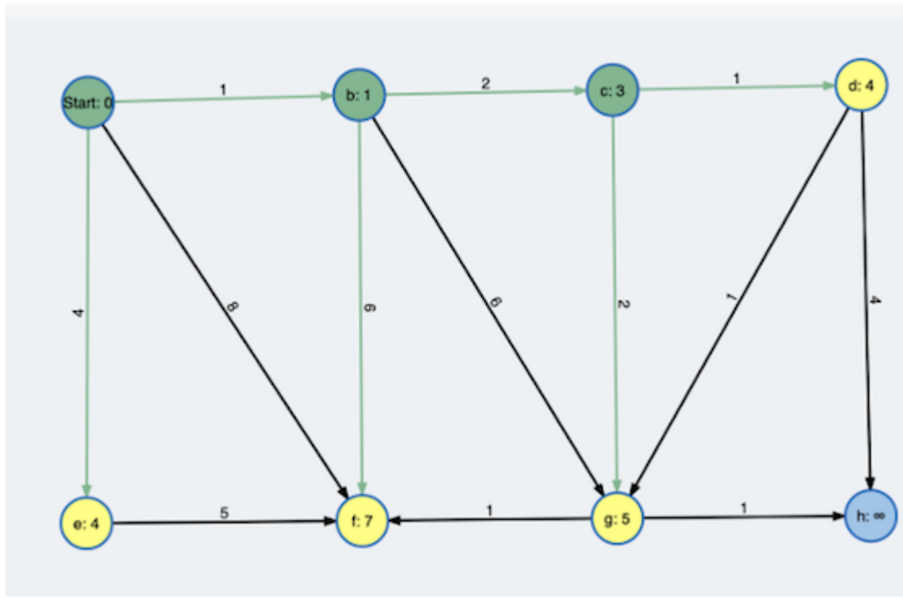
b	e	f

All neighbours of A visited
Similarly, for all nodes



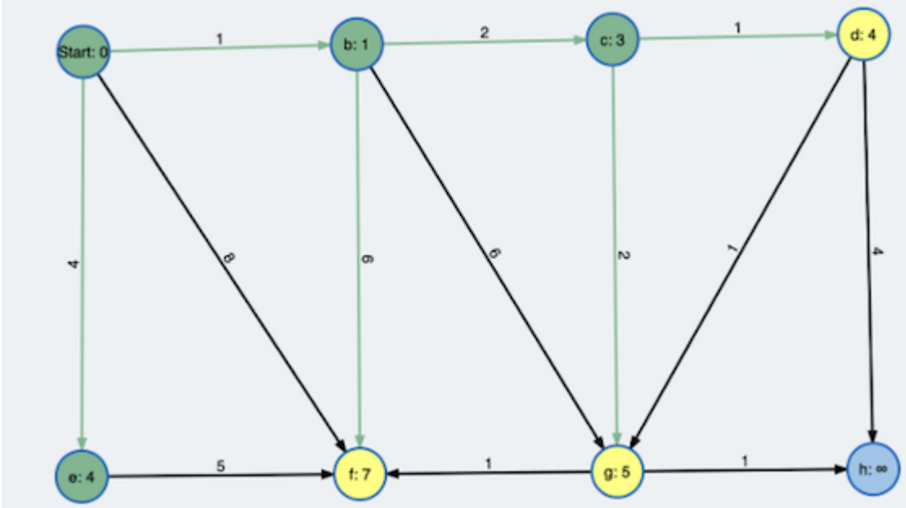
Priority Queue:

c	e	f	g



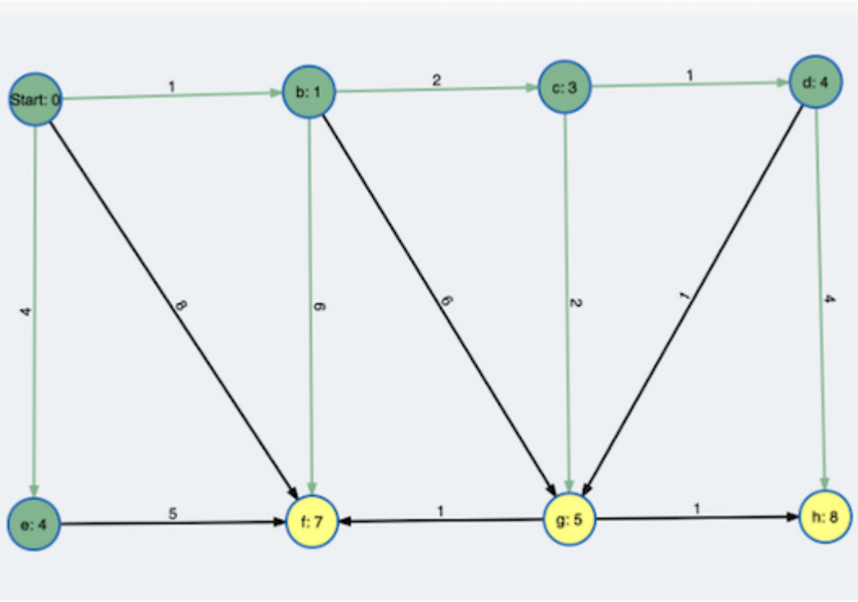
Priority Queue:

e	d	g	f



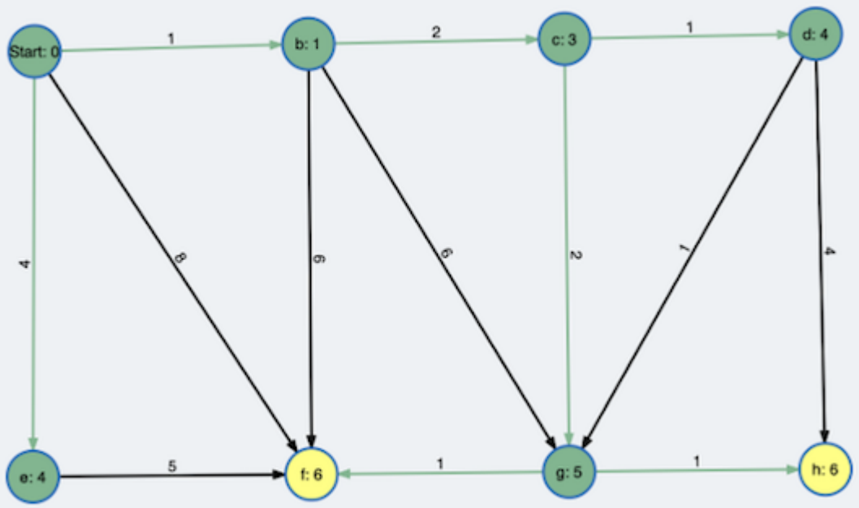
Priority Queue:

d	g	f



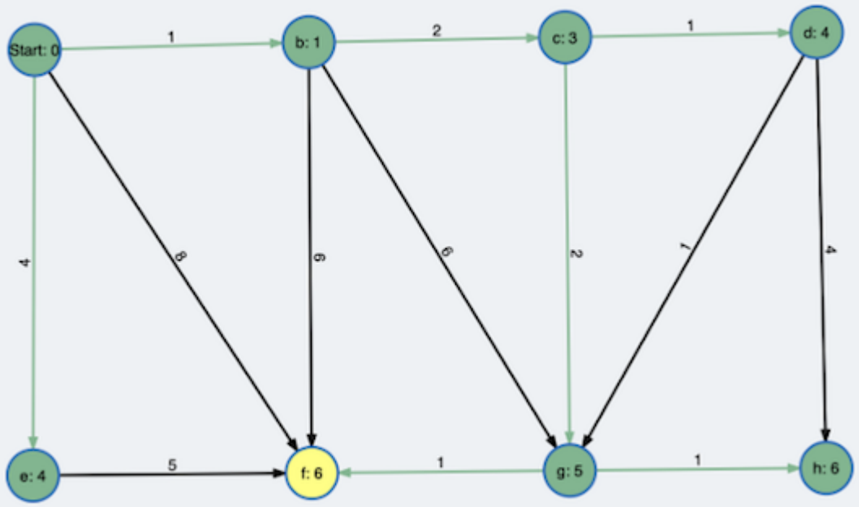
Priority Queue:

g	f	h



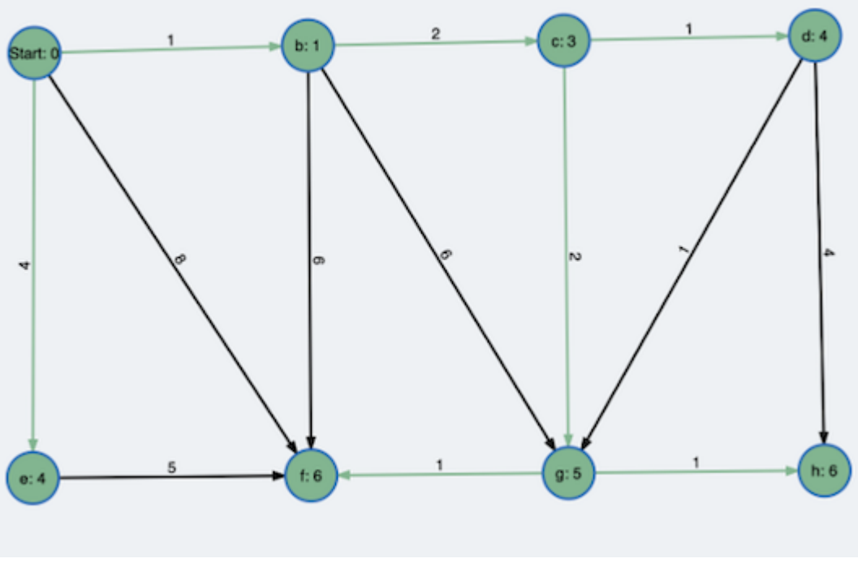
Priority Queue:

h	f



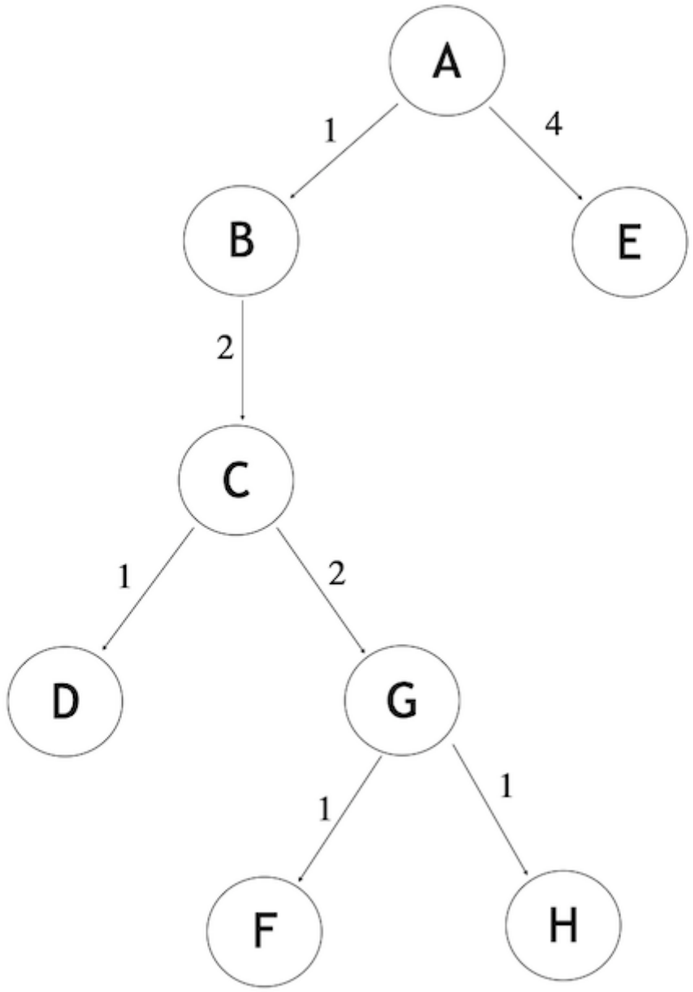
Priority Queue:

f



Priority Queue:

Thus, the shortest path tree is as follows



Question 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.

Answer 7

TCP

Network congestion may occur when a sender overflows the network with too many packets. At the time of congestion, the network cannot handle this traffic properly, which results in a degraded quality of service (QoS). The typical symptoms of a congestion are: excessive packet delay, packet loss and retransmission.

The function of TCP (Transmission Control Protocol) is to control the transfer of data so that it is reliable. The main TCP features are connection management, reliability, flow control and congestion control.

Transmission Control Protocol (TCP) uses a network congestion-avoidance algorithm that includes various aspects of an additive increase/multiplicative decrease (AIMD) scheme, along with other schemes including slow start and congestion window, to achieve congestion avoidance. The TCP congestion-avoidance algorithm is the primary basis for congestion control in the Internet. Per the end-to-end principle, congestion control is largely a function of internet hosts, not the network itself. There are several variations and versions of the algorithm implemented in protocol stacks of operating systems of computers that connect to the Internet.

We will see two of such algorithms.

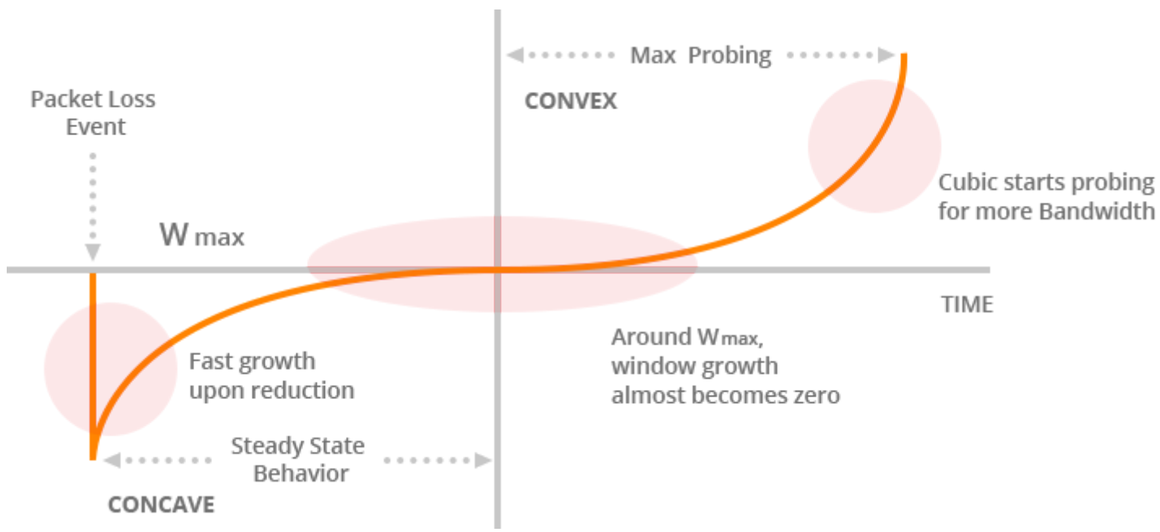
Cubic TCP

The low utilization problem of TCP in fast long-distance networks is well known. This problem arises from a slow increase of the congestion window following a congestion event in a network with a large bandwidth-delay product (BDP).

$Cubic$ is a modification to the congestion control algorithm of Standard TCP to remedy this problem. This document describes the most recent specification of $Cubic$. Specifically, $Cubic$ uses a cubic function instead of a linear window increase function of Standard TCP to improve scalability and stability under fast and long-distance networks.

Basic design and properties of $Cubic$ are :

For better network utilization and stability, $Cubic$ uses a cubic window increase function in terms of the elapsed time from the last congestion event.



While most alternative congestion control algorithms to Standard TCP increase the congestion window using convex functions, $Cubic$ uses both the concave and convex

profiles of a cubic function for window growth. 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. This style of window adjustment (concave and then convex) improves the algorithm stability while maintaining high network utilization. This is because the window size remains almost constant, forming a plateau around W_max where network utilization is deemed highest. Under steady state, most window size samples of Cubic are close to W_max, thus promoting high network utilization and stability. Note that those congestion control algorithms using only convex functions to increase the congestion window size have the maximum increments around W_max, and thus introduce a large number of packet bursts around the saturation point of the network, likely causing frequent global loss synchronizations.

W_i is calculated using the following function:

$$W_i = C(T - K)^3 + w_{max}$$

where $K = (\frac{w_{max}(1-\beta)}{C})^{\frac{1}{3}}$

and

β: Multiplicative decrease factor

w_{max}: Window size just before the last reduction

T: Time elapsed since the last congestion event

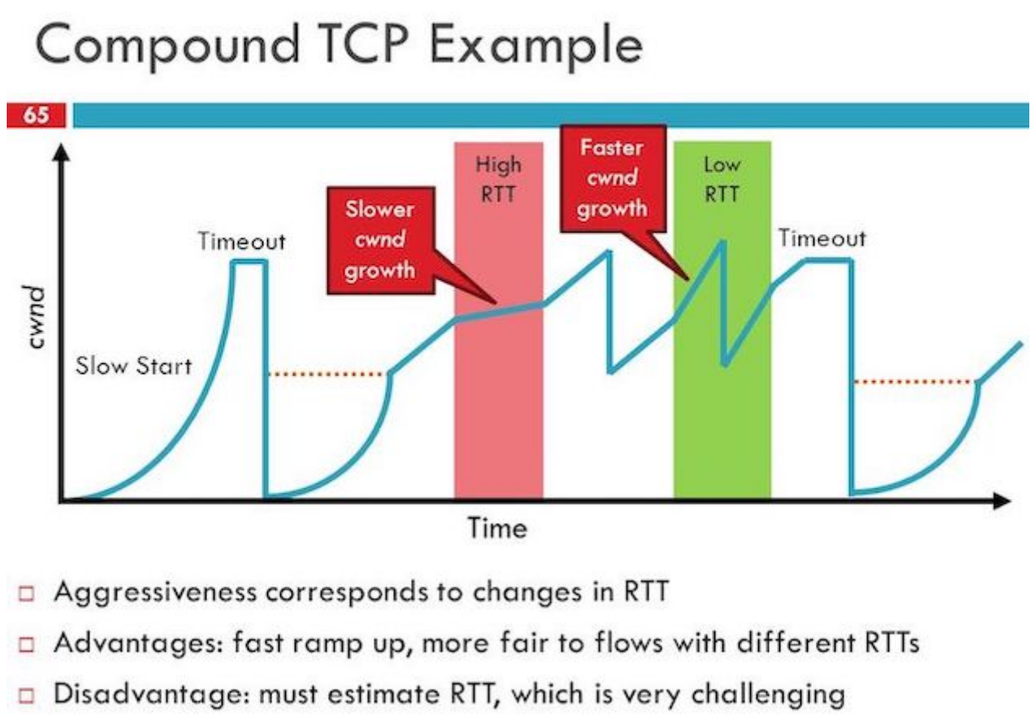
C: A scaling constant

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

As mentioned in $Cubic$ TCP, for a high- speed and long delay network, it takes standard TCP an unreasonably long time to recover the sending rate after a single loss event. Moreover, it is well-known now that in a steady-state environment, with a packet loss rate of p, the current standard TCP's average congestion window is inversely proportional to the square root of the packet loss rate. Therefore, it requires an extremely small packet loss rate to sustain a large window.

The key idea is that if the link is under-utilized, the high-speed protocol should be aggressive and increase the sending rate quickly. However, once the link is fully utilized, being aggressive will not only adversely affect standard TCP flows but will also cause instability. $Compound$ incorporates a scalable delay-based component into the standard TCP's congestion avoidance algorithm. Using the delay component as an automatic tuning knob, $Compound$ is scalable yet TCP friendly.



$Compound$ modifies Standard TCP's loss-based control law with a scalable delay-based component. To do so, a new state variable is introduced in the current TCP Control Block (TCB), namely dwnd (Delay Window), which controls the delay-based component in $Compound$. The conventional congestion window, cwnd, remains untouched, which controls the loss-based component in $Compound$. Thus, the $Compound$ sending window now is controlled by both cwnd and dwnd. As stated above, $Compound$ retains the same behavior during slow start. When a connection starts up, dwnd is initialized to zero while the connection is in slow start phase. Thus the delay component is only activated when the connection enters congestion avoidance. The delay-based algorithm has the following properties. It uses a scalable increase rule when it infers that the network is under-utilized. It also reduces the sending rate when it senses incipient congestion. By reducing its sending rate, the delay-based component yields to competing TCP flows and ensures TCP fairness. It reacts to packet losses, again by reducing its sending rate, which is necessary to avoid congestion collapse.

Some similarities between $Cubic$ and $Compound$:

1. Both were aimed to counter the problem of low utilization of TCP in fast long-distance networks.
2. Both methods are more aggressive than normal TCP methods.
3. Sometimes they are unfair to cross-traffic due to their aggressiveness.

Some differences between $Cubic$ and $Compound$:

1. $Compound$ uses the delay component to predict congestion and acts accordingly, whereas no such method is used in $Cubic$.
2. $Cubic$ changes its aggressiveness using the congestion window size where it got the congestion event whereas, $Compound$ uses the RTT values to change its aggressiveness.
3. $Cubic$ has both concave and convex parts of the graph and thus it remains most of the time plateau between the concave and convex growth region which allows the network to stabilize before it begins looking for more bandwidth, On the other hand $compound$ has a exponential probing for setting initial size of congestion window, followed by the sawtooth pattern for AIMD.

Fairness in TCP

One of the main objectives of TCP is to control the congestion in the Internet. This control is not efficient if it does not ensure a fair sharing of network resources. A major problem of TCP is its bias against connections with long round-trip times (RTT). These connections are not able to achieve the same throughput as the other connections sharing the same path and having a smaller RTT. The connections with small RTT increase quickly their windows and grab most of the available bandwidth.

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. However, there are a wide range of Congestion Control Algorithms(CCAs) available or are being researched.When the network deals with a wide variety of transport protocols and notably CCAs, the risk is that the fair sharing of network resources is jeopardized due to insufficiently tested CCAs.When implementing new CCAs, the best practice is to check whether new algorithms compete fairly with traditional standardized CCAs. The challenge is that there exists a wide range of Internet configurations which need to be taken into account during evaluations.Results show that the performance of CCAs can dramatically vary as a function of the network configuration, i.e., network path, bottleneck buffers sizes, and traffic variations.

In the interest of validating new CCAs a first estimation of their fairness needs to be obtained by means of local test platforms. However, these methods many times are not able to actually simulate the real complexity of the large scale internet. Along with this, there are some fairness matrices to check the fairness such as

1. Jain Index

$$\mathcal{J}(x_1, x_2, \ldots, x_n) = \frac{(\sum_{i=1}^{n} x_i)^2}{n \cdot \sum_{i=1}^{n} {x_i}^2} = \frac{\overline{\mathbf{x}}^2}{\overline{\mathbf{x^2}}} = \frac{1}{1 + \widehat{c_\mathrm{v}}^2}$$

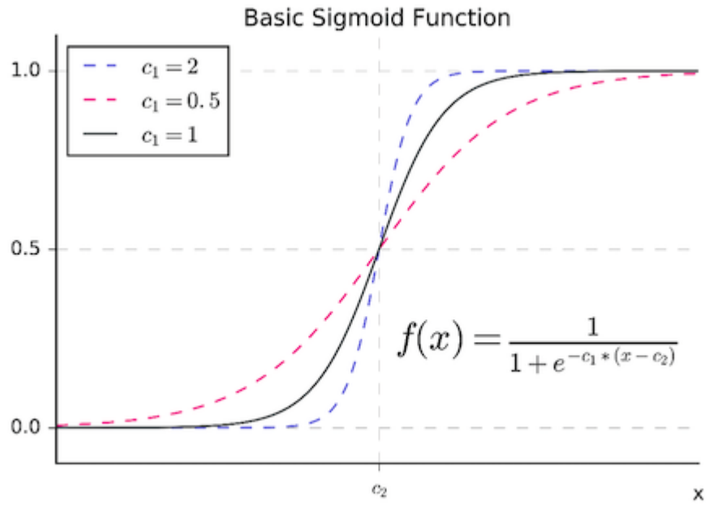
This equation,rates the fairness of a set of values where there are n users, x_i is the throughput for the i^{th} connection, and $\widehat{c}_v$ is the sample coefficient of variation . The result ranges from $\frac{1}{n}$ (worst case) to 1 (best case), and it is maximum when all users receive the same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

2. Product measure
3. Epsilon fairness
4. Bossaer's fairness index

Some possible ways to better the TCP congestion control algorithms :

1. Instead of reacting to a congestion, we can try and react before a congestion occurs. We can send data from routers about the current situtaion of congestion. Using past data/statistics we can apply an algorithm something like a machine learning model to preditct the congestion beforehand. Then we can reduce our congestion windows so as to prevent congestion.

2. As opposed to the $Cubic$ we can exponentially increase when capacity is available and then avoid congestion as long as possible once the load is heavy. I.e. a curve shape similar to a sigmoid. Then, depending on the previous cycles we can change the aggressiveness of the next cycle by changing the values of c_1 and c_2 in the graph.



Question 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?

Answer 8

Description

PageRank is the most basic, 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. The basic idea is that importance of each page is calculated by counting the number of links/references of that website in other websites. Thus the algorithm assusmes that more popular websites are more likely to receive links from other websites.

All webpages are assigned a numerical weight. It denotes the importance of that page relative to the other in the group. The group can be any collection of pages which has references and cross citations which can even be as big as the World Wide Web (WWW). The numerical weight assigned to a entity E is refered to as $PageRank(E)$ and denoted by $PR(E)$. The group of web-pages can be visulasied as a big directed graph where each node denotes a webpage and all edges denotes a link/reference on one website to another. Higher the number of incoming edges into a node higher will be it's $PageRank()$ value.

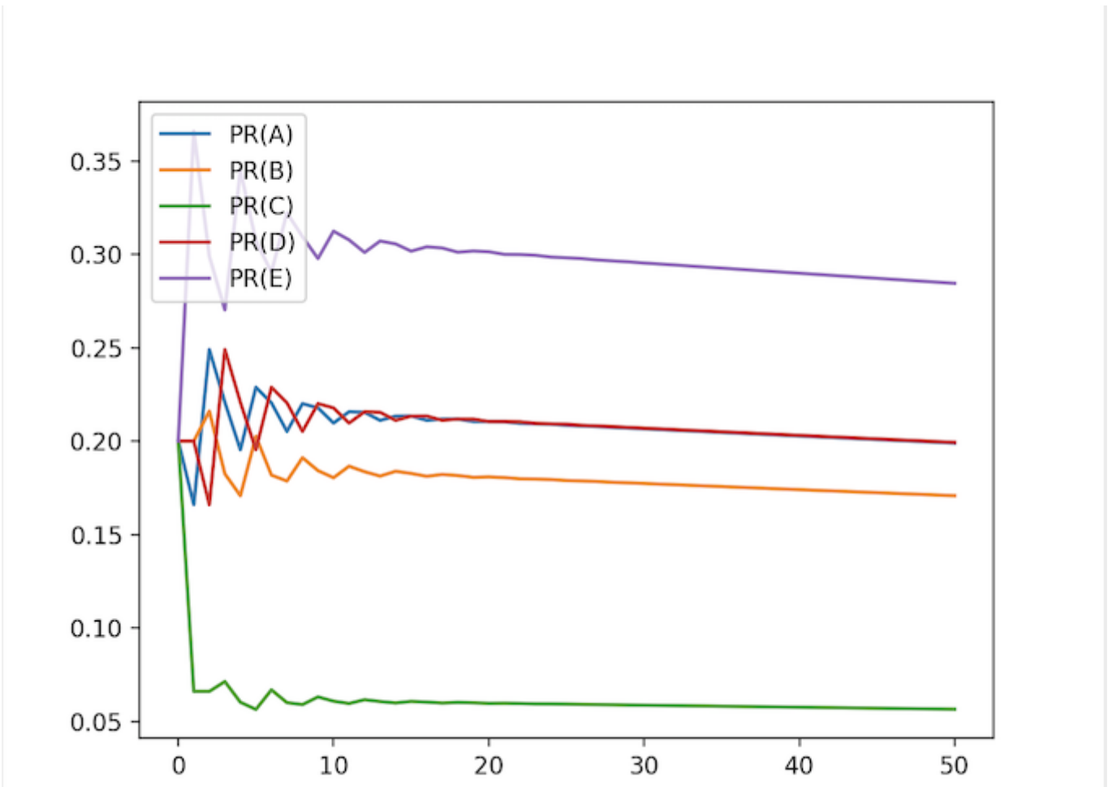
Basic calculations

As Pagerank uses a probability distribution, $\sum PR(u) = 1$. The results of as search will show the webpages in decresing order of their pageranks.
To find the pageranks of webpages following steps are used.

Consider an example where there are only 5 webpages in our consideration. Then, initial values of pagerank of all the webpages will be equal i.e. 0.2 in our case.
Now, let us suppose 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 1 outgoing links to E 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.

$$PR(A) = \frac{PR(B)}{3} + 0 \times PR(C) + 0 \times PR(D) + \frac{PR(E)}{2}$$
$$PR(B) = 0 \times PR(A) + \frac{PR(C)}{2} + 0 \times PR(D) + \frac{PR(E)}{2}$$
$$PR(C) = 0 \times PR(A) + \frac{PR(B)}{3} + 0 \times PR(D) + 0 \times PR(E)$$
$$PR(D) = \frac{PR(A)}{1} + 0 \times PR(B) + 0 \times PR(C) + 0 \times PR(E)$$
$$PR(E) = 0 \times PR(A) + \frac{PR(B)}{3} + \frac{PR(C)}{2} + \frac{PR(D)}{1}$$

Evolution of PR values of pages as we iterate



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

However in general an user might stop browsing any time. So, we need to factor in that case. Thus, let us assume a probability factor of d which denotes the probability that the user continues to browse and clicking links. This, will slightly change the equations now.
$PR(x) = \frac{1-d}{N} + d \cdot (\sum_{y \in X} \frac{PR(y)}{L(y)})$ where N is the total number of webpages

We thus keep iterating these set of equations untill the values converge enough to fall in our max error bracket.

Matrix Approach

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

where **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 $l(pi, pj)$ is the ratio between number of links outgoing from page j to page i to the total number of outbound links of page j, and normalized such that, for each j $\sum_{i=1}^{N} l(pi, pj) = 1$

Thus,

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

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 $t \to \infty$, 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 :
$R = (I - d \cdot M)^{-1} \times \frac{1-d}{N} \cdot 1$.

Now, clearly in websites like YouTube there are no/few links from one videos to another. In such cases, some other set of factors in taken into consideration such as

- All videos have a certain topic as their focal point. Thus based on factors such as the title, tags, description, and video content itself they are ranked.
- The search algorithm incorporates aggregate engagement signals from users, such as the watch time of a particular video for a particular query. Thus, it shows the videos which it thinks might be more interesting to a user based on their past history. They also consider the past history of other but similar users and the videos they liked.
- Videos which are famous and liked by many users in general is expected to have better expertise, authoritativeness, and trustworthiness on a given topic and thus are of better quality and are given higher preferrence.
- When a search related to sensitive topics like 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.
- If a video is very popular is getting more and more views in less time. They are given a higher preference.

Framework for searching on Youtube

In general, for users which regurarly use youtube and have a long history of videos. The factor considering the history of the users should be given more priority. Thus, the factors like genre of videos user likes, channels subscribed to, etc. are important. However, there must also be atleast some new recommendations so that user is able to explore more videos and is not limited to specific channels. For this, there can be filters which allow the user to change the amount of "new" recommendations they get for each search.

Furthermore, the tags which each videos has can be verified more extensively using computer vision so as to reduce the number of fake results. We can even divide them into sub-categories.

There can also be filters which gives more importance to professionally made videos.

We can divide videos based on the age of user it caters to.