Roll No: EE18B067 Name:Abhishek Sekar

Date: December 12, 2021

1. 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 $\mathcal{O}$(nlogn).

Solution:
Code:

```
#Code for merge sort

def merge(L,R): # function to merge the left part and the right part into a sorted array
    Result = [] #initialising the result array
    i = 0
    j = 0
    n1 = len(L)
    n2 = len(R)

    while (i < n1 and j < n2):
        if (L[i] <= R[j]):
            Result.append(L[i]) #append the smaller element
            i += 1              #update i
        else:
            Result.append(R[j])
            j += 1

    #Copy the remaining elements of L, should there be any
    while (i < n1):
        Result.append(L[i])
        i += 1

    #Copy the remaining elements of R, should there be any
    while (j < n2):
        Result.append(R[j])
        j += 1

    return Result

def mergesort(A): #function that performs merge sort on an array A
    n = len(A)
```

```
    if(n <= 1):
        return A

    else:

        #The divide and conquer step, where we're splitting A into L and R portions
        L = mergesort(A[:n//2])
        R = mergesort(A[n//2:])

    return merge(L,R)


def unique(A): #removes all duplicates from array A
    temp = []
    A = mergesort(A) #O(nlogn) step
    ele = A[0]
    temp.append(A[0])
    for i in range(1,len(A)):
        if(A[i] != ele):
            ele = A[i]  #make ele the new element to check in the future steps
            temp.append(A[i]) #append every unique element

    return temp
```

Sample Input and Output:

```
Input Array:[68, 66, 44, 44, 3, 3, 2, 3, 4, 5, 6, 7, 8, 9, 2, 3, 5, 1, 0, -1, -5, -6.7, -8]
Unique Array:[-8, -6.7, -5, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 44, 66, 68]
```

Explanation:

- First, we will have to sort the original sequence A. This can be done using merge sort or quick sort in roughly $\mathcal{O}$(nlog(n)) time.
- Next, we iterate through the sorted array in a sequential fashion and append all the unique elements to another array. This takes $\mathcal{O}$(n) time as we're just iterating throughout the array.
- Lastly, we return this array containing the resulting elements.

We can include an additional step before returning the output should the ordering of the array A be retained. In this case, we'll start off with another array which is just an array with A in it's first row and it's indices in the second. Now, the sorting will take place based on the first row of this new array and hence, in the penultimate step, the original ordering can be preserved by sorting the array again, but now across the row of indices.

Time Complexity:
Thus, from our description of the algorithm, we see that it involves one or two steps comprising a sort and one step which involves traversing the whole array.
Thus, our net time complexity will be $\mathcal{O}$(sort) +$\mathcal{O}$(traversal) = $\mathcal{O}$(nlog(n)) + $\mathcal{O}$(n) as there are n elements in the array.
Thus, the net time complexity will be of the order, $\mathcal{O}$(nlog(n)).

2. Given an array A of n integers in the range [0, n^2 - 1], describe a simple method for sorting A in $\mathcal{O}$(n) time.

Solution:
Code:

```
#radix sort based approach to solve the question

def countsort(arr,n,scale): #hashing based sort n -> base scale -> scale of digit

    output_arr = [0]*n #output array
    count      = [0]*n #count array

    #storing the counts
    for i in range(n):
        count[(arr[i]//scale) % n] += 1

    #making each count the sum of the previous counts
    #this will indicate the actual position of this digit in output[]
    for i in range(1,n):
        count[i] += count[i-1]

    #make the output
    for i in range(n-1,-1,-1):

        output_arr[count[ (arr[i] // scale) % n] - 1] = arr[i]
        count[(arr[i] // scale) % n] -= 1

    arr[:] = output_arr[:] #thus now arr is sorted according to scale


def radixsort(arr,n):

    #first do countsort with scale = 1 with base n on the first digit
    print(f'Input array: {arr}')
    countsort(arr,n,1)
```

```
    #now do countsort with scale = n with base n on the second digit
    countsort(arr,n,n)

    print(f'Sorted array: {arr}')
```

Sample Input and Output:

```
Input array: [40, 12, 45, 32, 33, 1, 22]
Sorted array: [1, 12, 22, 32, 33, 40, 45]
```

Explanation:
Let us discuss the two different sorting techniques used in the above code, count sort and radix sort.
First, starting off with count sort.
Count sort is a hashing based sort technique, where we first find the *count* of each element in the input array, which is nothing but the number of times said element occurs in the input array (for a base greater than the largest element in the array).
Next, we accumulate all the previous counts for each element.
Now, this is indicative of the relative position of that element inside the array.
Once we have this info, we just update the output array which contains the sorted input and as we update it we appropriately decrease the count of the element we're updating.
Shown below is a small example for counting sort.

```
Input array : [1, 4, 3, 3, 2]
Count: [0, 1, 1, 2, 1]
Cumulative count:[0, 1, 2, 4, 5]
Output Arr: [0, 2, 0, 0, 0]
Count : [0, 1, 1, 4, 5]
Output Arr: [0, 2, 0, 3, 0]
Count : [0, 1, 1, 3, 5]
Output Arr: [0, 2, 3, 3, 0]
Count : [0, 1, 1, 2, 5]
Output Arr: [0, 2, 3, 3, 4]
Count : [0, 1, 1, 2, 4]
Output Arr: [1, 2, 3, 3, 4]
Count : [0, 0, 1, 2, 4]
Sorted array: [1, 2, 3, 3, 4]
```

As we can see above, as and when any new element joins the output array, the count is updated and the position at which it joins is dictated is dictated by the cumulative count array. For instance 2 joins at the second index of the output array as the third index of the cumulative count is 2.
Now coming to radix sort, we're implementing it using the count sort algorithm described above.
Radix sort as the name suggests sorts in a digit by digit manner, i.e., starts sorting the elements based on the least significant digit first then moves on upwards till the most significant digit. For each sub routine which comprises of sorting till that location is done using the count sort algorithm.
Thus, we can try using these techniques to get an $\mathcal{O}$(n) sort by exploiting the properties of the input array.
As the range of the array is $[0, n^2 - 1]$, we can employ a radix sort based strategy using a base of n as the whole range can be represented using just two digits in this number system.
We first employ count sort on the first digit and using that output apply a count sort on the second digit to obtain the final sorted array.
Time Complexity:
First let us look at the theoretical time complexities before looking at the code.
Count sort has a time complexity of $\mathcal{O}$(n+k) where n is the size of the input and k is the range or base considered, therefore in our case, count sort has a time complexity of $\mathcal{O}$(n) which is supported by the code too.

Radix sort has a time complexity of $\mathcal{O}(d \cdot (n+b))$ where d is the number of digits in the representation and b the base of the representation.
Therefore, we see that even radix sort is $\mathcal{O}$(n) for our application and this is supported by the code too.
Thus, using a count sort based radix sort we can sort the given array A in $\mathcal{O}$(n) time.

3. Describe a non-recursive algorithm for enumerating all permutations of the numbers $\{1, 2, \ldots, n\}$ using an explicit stack.

Solution:
Code:

```
def permutations(n): #give natural number n and the function generates permutations of {1,2

    stack = [] #explicit stack used
    permutation_list = [] #list used to store the various permutations

    for i in range(1,n+1): #main loop

        if(i == 1):
            permutation_list.append([1])

        else:
            temp_list = []

            for lists in permutation_list: #for each 'sub' permutation


                for j in range(i):

                    list_copy = lists[:] #a copy of the list
                    stack = []


                    for k in range(j):


                        stack.append(list_copy.pop())

                    stack.append(i)  #incorporate the ith digit at the jth location

                    for l in range(len(list_copy)):
```

```
                    stack.append(list_copy.pop())



                temp_list.append(stack) #create a new 'sub' permutation



        permutation_list = temp_list



for i,permutation in enumerate(permutation_list): #print all the permutations

    print(f"Permutation {i+1}:{permutation}")
```

Sample Input and Output:

```
Input : n = 3
Permutation 1:[3, 1, 2]
Permutation 2:[1, 3, 2]
Permutation 3:[1, 2, 3]
Permutation 4:[3, 2, 1]
Permutation 5:[2, 3, 1]
Permutation 6:[2, 1, 3]
```

Explanation:
The above stack based algorithm actually breaks the standard recursive algorithm into an iterative one and thus takes the same computational time.
Before discussing the code, let us look at the approach.
What we wish to do is to compute at iteration i , all permutations of the numbers $\{1, 2, \ldots, i\}$. Thus, as we iterate to the desired n, we obtain all the permutations.
Here a python list is used as a stack, where the append() operation is similar to pushing data into the stack and pop() is similar to removing.
We also have a list to store all the permutations.
At the first iteration, we initialize the permutation list with 1, as 1 is the only permutation one can have.
For the second iteration however, we can have two possible cases, [2,1] and [1,2].
To generate this, we make use of the permutations till the previous iteration.
We include the new number, which in this case is 2, to all possible locations to all the permutations till the previous iteration.
To do this we create a list which holds a copy of the permutation.
We iterate over all the possible locations described above and push data from the list into the stack till that location is reached. Once the location is reached, we push the new number into the stack and the rest of the data is fed into the stack after that.
This procedure is repeated for all the permutations and this successfully generates new permutations for the i^{th} iteration from those in the $(i-1)^{th}$ one.
To demonstrate this, let us consider the example of the second iteration.
Our list of permutations is just [1], so in order to generate a new permutation, we can add the number 2 before 1 or after 2.
Thus, for the former case we first push 2 into the stack before pushing 1 from the list and for the latter we first push 1 then

push 2.
Thus, this approach incorporates an explicit stack to generate all the permutations of the sequence $\{1, 2, \ldots, n\}$ in a non-recursive manner.

4. Suppose Dijkstra's algorithm is run on the following graph, starting at node A.

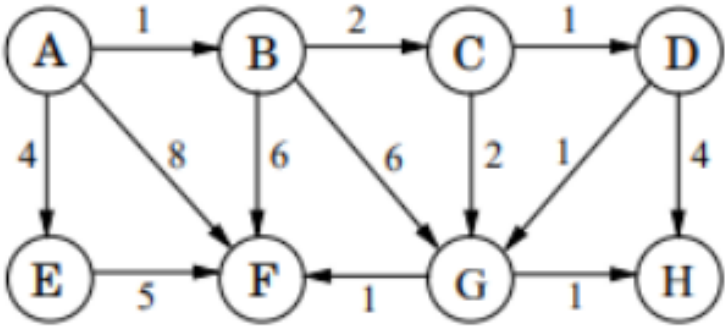

Figure 1: Given Graph

(a) Draw a table showing the intermediate distance values all the nodes at each iteration of the algorithm.

Solution:
Short Note on Djikstra's Algorithm:
We use Djikstra's algorithm to compute the shortest path to the other vertices from A.
This is computed in the following manner repeatedly till all the vertices are exhausted.

- **Initialization:**
  Keep track of the vertices included in the shortest path tree. This set is initially empty.
  Assign a distance to all vertices in the input graph. Initialize these distances with infinity except for the source node.
- Pick a vertex u not in the set such that it has the minimum distance value. Include it to the set and update the distances of all the adjacent vertices of u if the distance of the edge u-v is lesser than the distance v.

Iterations of Djikstra's on the given graph:

Iteration 1:

- Initialization: Distance of A is 0, all other distances are infinity.
- Set = $\{A\}$

Distances at iteration 1

A	B	C	D	E	F	G	H
0	∞	∞	∞	∞	∞	∞	∞

Iteration 2:

- Compute distances from A to adjacent vertices and update them if the distances are lower.
  Therefore, the new distance of B will be 1, new distance of E = 4 and F = 8.
- New node not in set with minimum distance = B.
  Therefore set = $\{A, B\}$

Distances at iteration 2

A	B	C	D	E	F	G	H
0	1	∞	∞	4	8	∞	∞

Iteration 3:

- Compute distances from B to adjacent vertices and update them if the distances are lower.
  Therefore, the new distance of C will be 2+1 = 3, new distance of F = 6+1 = 7 and G = 6+1 = 7.
- New min cost vertex = C.
  Therefore set = $\{A, B, C\}$

Distances at iteration 3

A	B	C	D	E	F	G	H
0	1	3	∞	4	7	7	∞

Iteration 4:

- Compute distances from C to adjacent vertices and update them if the distances are lower.
  Therefore, the distance of D = 4, G = 5.
- New min cost vertex = D.
  Therefore set = $\{A, B, C, D\}$

Distances at iteration 4

A	B	C	D	E	F	G	H
0	1	3	4	4	7	5	∞

Iteration 5:

- Compute distances from D to adjacent vertices and update them if the distances are lower.
  Therefore, the distance of H = 8. Distance of G remains un-updated.
- New min cost vertex = E.
  Therefore set = $\{A, B, C, D, E\}$

Distances at iteration 5

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

Iteration 6:

- Compute distances from E to adjacent vertices and update them if the distances are lower.
  All distances remain un-updated.
- New min cost vertex = G.
  Therefore set = $\{A, B, C, D, E, G\}$

Distances at iteration 6

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

Iteration 7:

- Compute distances from G to adjacent vertices and update them if the distances are lower.
  Therefore, the distance of H = 6, F = 6.
- New min cost vertex = H,F.
  Therefore set = $\{A, B, C, D, E, G, H, F\}$
- All vertices in set, so Djikstra's terminates.(As running it again with H and F won't change result again.)

Distances at iteration 7

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

Final Table iteration wise:

	A	B	C	D	E	F	G	H
iter 1	0	∞	∞	∞	∞	∞	∞	∞
iter 2	0	1	∞	∞	4	8	∞	∞
iter 3	0	1	3	∞	4	7	7	∞
iter 4	0	1	3	4	4	7	5	∞
iter 5	0	1	3	4	4	7	5	8
iter 6	0	1	3	4	4	7	5	8
iter 7	0	1	3	4	4	6	5	6
iter 8	0	1	3	4	4	6	5	6

(b) Show the final shortest-path tree.

Solution:
From the computed distances, we can eliminate some edges to come up with the final shortest path tree shown in the below figure.

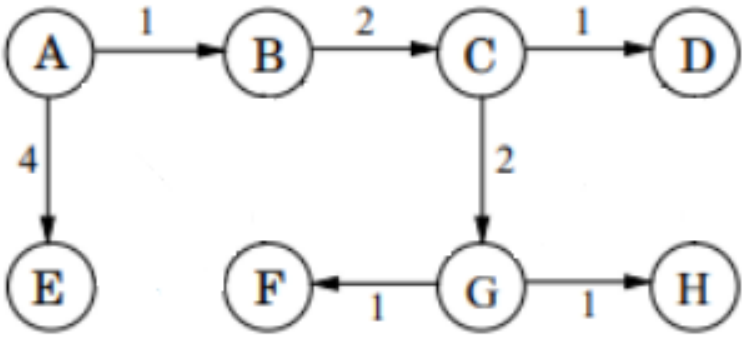

Figure 2: Shortest Path Tree

5. Describe an efficient greedy algorithm in python for making change for a specified value using a minimum number of coins assuming there are four denominations of coins (called quarters, dimes, nickels, and pennies) with values 25, 10, 5, and 1, respectively.

Solution:
Code:

```
def Min_change(val): #compute change using minimum number of coins for the given value usin

    denominations = [1,5,10,25]

    i = len(denominations) - 1

    coins = [] #list to store the change

    while(i >= 0): #iterate through different denominations
```

```
        #find coins using greedy approach
        while(val >= denominations[i]): #exhaust the use of that coin
            val -= denominations[i]
            coins.append(denominations[i])

        i -= 1


    print(f"Recommended change coins of the following denomination: {coins}")
```

Sample Input and Output:

```
Input : 37
Recommended change coins of the following denomination: [25, 10, 1, 1]
```

Explanation:
As this is a greedy approach we try making the change using coins of the highest denomination first, i.e., quarters followed by dimes, nickels and pennies in that order.
At each step, we try using the maximum number of coins at a particular denomination and only proceed to using coins of a lower denomination if coins from the higher denomination can't fulfil the change.
This constraint is checked by a while loop where the iteration terminates if the remaining change turns out to be lower than the denomination we are looking at.
We repeat this procedure till no more change is to be imparted.
At each step we append the coins used to impart the change to a list so as to keep track on the coins we are using.
Time Complexity:
As we have a while loop over the value,v, we have a worst case time complexity of $\mathcal{O}$(v).

6. Design an efficient algorithm in python for the matrix chain multiplication problem that outputs a fully parenthesized expression for how to multiply the matrices in the chain using the minimum number of operations.

Solution:
Code:

```
import sys
def print_parentheses(i,j,n,bracket):

    if(i == j): #if only one matrix is left in the current segment
        print(f'[M{i}]',end = " ")
        return

    print('(',end = " ")


    print_parantheses(i,bracket[i][j],n,bracket)
```

```
    print_parantheses(bracket[i][j]+1,j,n,bracket)

    print(')', end = " ")

def matrix_chain(p,n):

    dp = [[0 for x in range(n)] for x in range(n)]
    bracket = [[0 for x in range(n)] for x in range(n)]

    for l in range(2,n): #l chain length

        for i in range(1,n-l+1):

            j = i+l-1
            dp[i][j] = sys.maxsize #some large number
            for k in range(i,j):

                q = dp[i][k] + dp[k + 1][j] + p[i - 1]*p[k]*p[j]

                if(q < dp[i][j]):
                    dp[i][j] = q
                    bracket[i][j] = k




    print('Optimal Parentheses:')
    print_parantheses(1,n-1,n,bracket)
    print(" ")
    print('Optimal Multiplications')
    print(dp[1][n-1])
```

Sample Input and Output: Input: $M_1 \cdot M_2 \cdot M_3$
Where M_1 is a 1x2 matrix,M_2 is a 2x3 matrix and M_3 is a 3x4 matrix.
Output:
Optimal Parentheses:
(([M1] [M2]) [M3])
Optimal Multiplications
18
Explanation:
We incorporate a tabulated dynamic programming based approach to solve this problem.
At its core, the problem can be thought of as a dynamic programming problem where we wish to minimize the computational cost and the places where the brackets are to be inserted can be thought of as tracing the optimal path obtained from the

table.
The main part of this algorithm is present in the function matrix chain where we wish to tabulate this dynamic programming matrix.
The function takes an input p, which denotes the sizes of the matrix in the following fashion.
The dimensions of matrix i is given by, p[i-1]x p[i].
n refers to the length of the vector p.
dp[i][j] has the interpretation of the number of computations to be made from matrix i to matrix j.
Our approach starts by looking at matrix sub chains l at a time, where we try computing the computational cost for multiplying matrices for all such sub chains.
The DP update equation is given below.

$$q = dp[i][k] + dp[k+1][j] + p[i-1] \cdot p[k] \cdot p[j] \tag{1}$$

We're essentially trying to search for a 'k' matrix in between i and j such that inserting a bracket at k results in the least number of multiplications.
dp[i][k] represent the number of computations required as we go from the i^{th} matrix to the k^{th} matrix.
Likewise, dp[k+1][j] represent the number of computations required as we go from the $(k+1)^{th}$ matrix to the j^{th} matrix.
However, if we were to stop at k (i.e., insert a bracket there), we'll have to perform some additional computations, as multiplying the matrices from i to k, results in the final matrix having a size of p[i-1] x p[k] from the rules of matrix multiplication. Likewise, multiplying from k+1 through to j results in a matrix of size p[k] x p[j]. Therefore, the number of computations required to multiply these two matrices is $p[i-1] \cdot p[k] \cdot p[j]$.
Thus if this stop at k seems to be lower than what the cost was before, we update the DP matrix and save the location of the k^{th} matrix in the bracket matrix.
We then call a function which prints the parentheses for the matrix in a divide and conquer fashion which lets us know how the multiplication is to be performed.
Time Complexity:
The time complexity of this approach is $\mathcal{O}(n^3)$ as we use three nested loops for computing the DP matrix.

7. [Algorithms for "Transport Protocols"]
   (a) 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.

Solution:
Firstly, let us start off with some definitions.
Some Definitions:

- W_i= Window size (This determines the number of packets which can be transported to or from the user)
- r_i =Throughput/ Rate
- τ_i = Propagation delay
- T_i= Propagation delay + Queuing delay
- RTT (Round Trip delay time)(ms) = time duration taken by a round trip, i.e., the time a request takes in a network to travel, starting from the starting point, going to the destination and then coming back to the starting point.

A general practice for the TCP congestion control algorithms is to slowly increase W_i until the number of packets start to drop. The increment is made after every RTT.(This could be because there are too many users sending and receiving data at the same time, i.e., congestion).
As we wouldn't like to crash or overload, the algorithms drastically drop W_i in the case of congestion.
A new connection is later started by algorithms such as *slow start*.
As far as this answer and our analysis is concerned, we assume this connection has already been started and currently everything is in steady state.
Cubic TCP

- CUBIC TCP is a network congestion avoidance algorithm for TCP which makes use of the fact that most communication links tend to have high bandwidth now.
- It achieves high bandwidth connections over networks much faster and in a more reliable manner when compared to the earlier algorithms.
- It makes full use of these wide fat bandwidth links by aggressively increments the congestion windows but are restricted to not overload the network.
- In order to achieve this, the scheme proposed for increasing and decreasing the transmission ratio is of the form of a cubic function (hence the name).

The Algorithm
The algorithm roughly follows these steps.

- The window size while experiencing congestion is noted as W_{max} and this is kept as the maximum window size.
- This W_{max} value is set as the inflection point of the cubic function that governs the growth of the congestion window.
- Thus, when the transmission is restarted with a smaller window, it keeps increasing as per the concave portion of the cubic function if congestion is not experienced.
- The increments slow down as the window size approaches W_{max}. Once, the inflexion point,W_{max} has been reached, the window size increases but in a discreet manner.
- Lastly, if congestion is still not experienced, the window size continues to increase, but now based on the convex portion of the cubic function.

Thus, to summarize, the window size increases in large steps at first then slows down at the region that caused congestion last time around and once this region has crossed, continues to increase in large steps.
Mathematically, this is represented as,

$$W_i = C\left(T - \left(\frac{W_{max}(1-\beta)}{C}\right)^{\frac{1}{3}}\right)^3 + W_{max} \tag{2}$$

where,

- β is a multiplicative decreasing factor.
- T is the time elapsed since the previous congestion.
- C is a scaling constant

The demonstration of the algorithm is shown in the below image provided by noction.com.

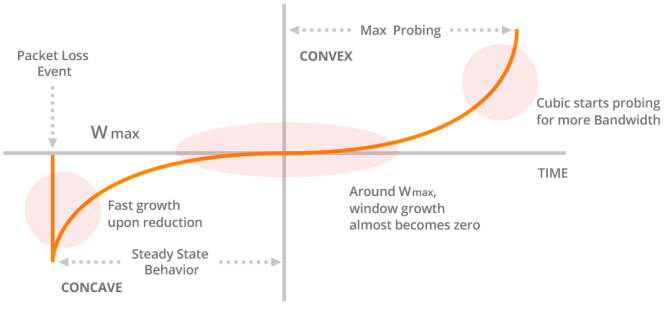

Figure 3: CUBIC TCP Congestion window

Compound TCP

- Compound TCP was an algorithm introduced by Microsoft offered alongside Windows Vista.
- It aggressively adjusts the sender's congestion window optimising TCP for large bandwidth connections as in the previous CUBIC TCP algorithm but tries not to harm the fairness.
- It estimates queuing delay in order to measure congestion.
- If the queuing delay is small, then compound TCP assumes that the path is less congested, which is a reasonable assumption to make.

Compound TCP maintains two different congestion windows.

- Additive increase/Multiplicative decrease (AIMD) window:
  We make a few assumptions here,
  - RTT is constant
  - The sender always transmits data
  - TCP will avoid retransmission timeouts

  In this case, W_i is increased (probed) linearly untill packet loss is detected then it drops the window size by dividing by a constant factor hence the name ' multiplicative decrease'.
  A common choice of the constant factor is 2.
  Therefore, mathematically,

$$W_i(t+1) = W_i(t) + a \qquad \text{(No congestion)}$$
$$W_i(t+1) = \frac{W_i(t)}{b} \qquad \text{(Congestion)}$$

- A delay based window.
  If the delay is small the 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 goal is to keep the sum of both the windows almost constant.
  Thus both the windows try maintaining a constant data transmission speed for a given link capacity.

(b) 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?

Solution:
Fairness of a TCP algorithm is a measure of whether users of an application are receiving a fair share of system resources.
There are several mathematical definitions of fairness.
Some of them are shown below.
Max-Min Fairness:
This type of fairness is said to be achieved by an allocation iff the allocation is feasible and is such that any attempt to increase the allocation of one connection results in the decrease in allocation of another.
A max-min fair allocation is achieved when bandwidth is allocated equally and in infinitesimal increments to all flows until one is satisfied. Then, the same thing is done in a recursive manner until all flows are satisfied or it runs out of bandwidth.
Jain's fairness index:
The mathematical definition of Jain's fairness index is shown below.

$$\mathcal{J}(x_1, x_2, \ldots, x_n) = \frac{(\sum_{i=1}^{n} x_i)^2}{n \cdot (\sum_{i=1}^{n} x_i^2)} = \frac{\overline{x}^2}{\overline{x^2}} = \frac{1}{1 + \hat{c_v}^2} \tag{3}$$

This rates the fairness through n users and x_i is the throughput for the i^{th} connection and $\hat{c_v}$ is the sample coefficient of variation.

This can be developed as a maximization problem as the result is the maximum when all users have the same allocation.
Thus, this is one of the most frequently used measures of fairness.
QoE
This is called the Quality of Experience fairness and it considers the quality of the user end experience.
This measure is popularly used for video and audio streaming.
The above methods are by no means exhaustive and more fairness methods exist.
Fairness for large scale networks:
For large scale networks, fairness of the TCP algorithm can be evaluated by simulating the network conditions.
Using simulators gives us detailed data and helps with system analysis.
It is also very cost efficient as we don't need to build the whole setup for testing TCP.

(c) 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.

Solution:
Current issues with TCP algorithms

- The main problem is that algorithms resolve congestion after it occurs. It does not proactively try to reduce congestion.
- There are some issues in the procedure to identify packet losses in conventional TCP algorithms.
- Each packet has a unique ID. When the receiver gets the packet there is an acknowledgement packet sent back. The receiver then looks for the next packet of data. If the receiver hasn't received it, it sends requests to the sender.If even after a fixed number of requests the packet hasn't been received, then the sender assumes the packet has been lost in transit and the packet is resent. Congestion is reported when a packet is lost in transit or takes longer than usual to arrive.
- The problem with the above scheme is that as the router's buffers fill up, there are high chances for congestion to happen even when the packets aren't lost in transit.

Some Ideas to resolve these issues:

- TCP can try looking into only the lost/sent packets.
- Some mechanism hsa to be employed to tell the sender the buffers are filling up without utilising the channel.
- We can try building a mathematical model based on past history of the user using Deep learning techniques.
- This will go a long way in resolving the congestion issues and predict them before they occur.
- These techniques are implemented in reality as well.
- Tuning the model routinely is also suggested.
- Fairness can also be enforced through this model by repetitive tuning.
- This model thus can prevent congestion before it even happens.

8. [Algorithms for "Search"]

(a) Describe the algorithmic components of PageRank, which is the search algorithm used by Google.

Solution:
Page rank algorithm was more than a million dollar algorithm invented by Larry Page the founder of Google.
The basic reduced aspects of the algorithm are described below.
Page rank is sort of like a graph based model for the world wide web where each web page is a node and there exists an edge from node i to node j if web page i has a link to web page j.

The out degree of the node is the number of outgoing links from a web page while the in degree is the number of links that are directed to the web page.
Thus, a high in degree could indicate importance or popularity.
The degree alone is not a good measure of importance.
The algorithm allocates an importance measure x_i as follows.

$$x_i = \sum_{j \in L_i} \frac{x_j}{n_j} \tag{4}$$

where L_i is the set of all nodes that link to node i and n_i is the out degree of the node i.
It is like summing up the importance node j is giving to node i but for all j.
The page rank matrix is thus represented using this partial importances.
The element $M_{ij} = \frac{1}{n_j}$ if there is a node coming into i from j and is 0 otherwise.
The importance vector is found using the equation x = Mx.
Thus the importance vector is an eigenvector of M with eigenvalue 1.
Thus, using a linear algebra based algorithm we can keep iterating and fine tuning till we obtain the importance vector x which corresponds to an eigenvalue of 1.
The importance vector will then tell us which sites are important and has a quantitative measure of importance.

(b) Outline and describe the algorithms used for searching in video sharing sites like YouTube.

Solution:
The YouTube algorithm is one of the worlds biggest trade secrets!
However, we can make some inferences based on public releases made by Youtube and their goals.
Youtube wishes to retain people on their platform for longer and longer and give the best personalized recommendations to the users.
The recommendation algorithm/search algorithm commonly used by Youtube is usually user by user collaborative filtering, where each user gets recommended videos based on how similar users have rated or liked the video. The similarity of users can be calculated by a plethora of similarity metrics. However, pearson score is the most commonly used one.
The following data can be fed to the above algorithm,

- Number of impressions on the video.
- Number of times the video has been suggested.
- Time spent watching the video.
- Retention time.
- Video tags.
- Geographical tags,i.e., location of content creator etc.
- Impressions of the channel.
- Recency bias. New popular videos are preferred over older ones.
- Users search history and watch history.
- User data is sadly also mined to obtain information regarding his/her other activities to provide better recommendation.
- Of late, more quirky or polarising videos are recommended to users as that helps the platform with retention time.

Thus, these parameters can be used to train a collaborative filtering algorithm which can then find whether or not a user will like a specific video and also find the popularity of a video in an objective manner.

(c) Describe how you might design a framework for searching on youtube?

Solution:
We can devise a framework for searching on YouTube based on the list of parameters mentioned in the previous question.

- We can compare text keywords and the labels a content creator has attached to the video. These labels may however be exploited by users and hence we can additionally develop a machine learning model to flag these *click bait* videos based on a parameter like the negative impressions or the retention time of the video.
- We can use a collaborative filtering based algorithm as it compartmentalizes video into categories and sub categories which eases the workload on other sibling algorithms.
- We can try providing the best choice for a user by checking the browsing and the watch history of the user as they're more likely to watch the content they like.
- We can try providing videos that other similar users have liked in the past by training a collaborative filtering algorithm.
- Compartmentalizing the various parts of a video and analyzing the retention of those parts and the content overlap in those parts with what the user is searching for.

These are some techniques we can employ to come up with the YouTube search algorithm.