EE4371 - Final Exam

Om Shri Prasath, EE17B113

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 $O(n \log n)$

Solution: $O(n \log n)$ Time Complexity

The solution to remove the duplicates in $O(n \log n)$ time complexity is as follows :

- We sort the array `arr`
- We start two iterators `i` which starts from the beginning of the array and `j` which starts from the second element of the array (assuming that the array has more than one element).
- Until `j` reaches the end of the array we do the following :
  - We compare `arr[i]` and `arr[j]`
  - If they are not same, we give an increment of `1` to `i` and assign `a[j]` to `a[i]`
  - If they are the same, we just continue
  - We give an increment of `1` to `j` irrespective of the above condition
- We return `arr[:i+1]` which contains all the unique elements in the array

In [1]:
```python
"""
Function to remove duplicates from array

Input  : Array
Output : Array with duplicates removed
"""


def remove_duplicates(arr):
    # Get the length of the array
    n = len(arr)
    # Sort the array
    arr.sort()
    # Initialize the iterators
    i = 0
    j = 1
    # Loop to overwrite the duplicate
    while j < n:
        # Overwrite a[i] with a[j]
        if arr[i] != arr[j]:
            i += 1
            arr[i] = arr[j]
        j += 1
    # Return array without duplicates
    return arr[: i + 1]


# Test the function

arr = [1, 2, 4, 2, 2, 4, 2, 1, 3, 4, 6, 3, 2, 2, 5, 5, 6]
print("Original array with duplicates :", arr)
print("Array with duplicates removed  :", remove_duplicates(arr[:]))
```

```
Original array with duplicates : [1, 2, 4, 2, 2, 4, 2, 1, 3, 4, 6, 3, 2, 2, 5, 5, 6]
Array with duplicates removed  : [1, 2, 3, 4, 5, 6]
```

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

Solution : $O(n)$ Time Complexity

Here since the range of the numbers are fixed between 0 and $n^2 - 1$, we know that in a particular base the maximum number of digits of the given numbers will be fixed. Number of digits being fixed means that radix sort will be viable. The radix sort works as follows :

- For the given number we represent it in a base, and do the following starting from the least significant digit to the most significant digit
  - Let the digit to be used in the iteration be the i^{th} digit of the number
  - Set the key of all the numbers to be the i^{th} digit of the number
  - Sort using a **stable** sort algorithm like the count sort, which sorts the numbers such that the order is preserved for equal numbers
  - We then move to the next digit

The time complexity of radix sort is of the order $O((n+b)\log_b k)$ where b is the base used for the representation of the numbers and k is the largest possible number. In this case, $k = n^2$ and we can use $b = n$, this gives us time complexity of $O(2n \log_n n^2) = O(4n \log_n n) = O(n)$, which is the time complexity required for our case.

In [2]:
```python
"""
Function to count sort a given array using the p'th digit as key

Input  : Array of numbers
Output : Array sorted with the p'th digit
"""


def count_sort(arr, p, n):
    # Output array
    output = [0] * n
    # Counts array
    counts = [0] * n

    # Increase counts for the p'th digit
    for i in arr:
        counts[(i // p) % n] += 1

    # Feed forward the counts
    for i in range(1, n):
        counts[i] += counts[i - 1]

    # Fill the output array based on the counts
    for i in arr[::-1]:
        output[counts[(i // p) % n] - 1] = i
        counts[(i // p) % n] -= 1

    # Return the sorted array
    return output


"""
Function to apply radix sort on array
Input  : Array of numbers
Output :
"""


def radix_sort(arr):
    n = len(arr)

    """
    Number of digits to be taken will be 2
    Since n^2-1 is the largest number, the number
    of digits will be 2 in base n, since n^2-1 is
    just [n-1,n-1] in base n
    """

    # Sort for first digit
    arr = count_sort(arr, 1, n)

    # Sort for second digit
    arr = count_sort(arr, n, n)

    # Return the sorted array
    return arr


# Testing the code

arr = [24, 3, 8, 10, 19]
print("Original Array : ", arr)
print("Sorted array   : ", radix_sort(arr[:]))
```

```
Original Array :  [24, 3, 8, 10, 19]
Sorted array   :  [3, 8, 10, 19, 24]
```

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

Solution : $O(n!)$ Time Complexity

We use the non-recursive Heap Algorithm for solving the given problem by simulating a stack:

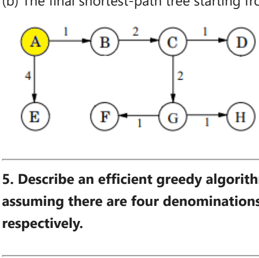
- We will initialize array `state` of size n with 0 which is used to encode the for-loop counter of a recursive function call
- We will create the array `arr` of numbers from 1 to n and print it as one of the permutations.
- We will initialize a iterator `i` =0
- Until `i` is less than n we do the following :
  - If `state[i]<i`, which means that we are inside a recursive call stack
    - If i is odd, we swap `arr[i]` with `arr[state[i]]`
    - If i is even, we swap `arr[i]` with `arr[0]`
    - We print `arr` as one of the elements
    - We increase `state[i]` by 1, which indicates that we have done one iteration of swapping
    - We set `i` to 0, which simulates a recursive call of the function
  - If `state[i]>=i`, means that the stack is completed and we pass back the control to the previou stack
    - We set `state[i]` = 0, indicating the completion of the stack
    - We increment `i` by 1

The above algorithm is of the order $O(n!)$ time complexity since we reach all the permutations once, and there are $n!$ permutations

In [3]:
```python
"""
Function to print permutations of an array of numbers from 1 to n

Input  : n - Number of elements in the array
Output : Permutations of the array
"""


def generate_permutations(n):

    # Initialize the array for permutation
    arr = list(range(1, n + 1))

    # Initialize the state variable
    state = [0] * n

    # Print the first permutation of the array
    print(arr)

    # Initialize the pointer
    i = 0

    # Loop which simulates the recursive algorithm
    while i < n:
        # Inside a recursive call
        if state[i] < i:
            # Swap elements based on i
            if i & 1:
                arr[state[i]], arr[i] = arr[i], arr[state[i]]
            else:
                arr[0], arr[i] = arr[i], arr[0]

            # Print the permutation
            print(arr)

            # Simulate calling function recursively
            state[i] += 1
            i = 0
        # Completed a recursize vall
        else:
            state[i] = 0
            i += 1


# Check the function

generate_permutations(4)
```

```
[1, 2, 3, 4]
[2, 1, 3, 4]
[3, 1, 2, 4]
[1, 3, 2, 4]
[2, 3, 1, 4]
[3, 2, 1, 4]
[4, 2, 1, 3]
[2, 4, 1, 3]
[1, 4, 2, 3]
[4, 1, 2, 3]
[2, 1, 4, 3]
[1, 2, 4, 3]
[1, 3, 4, 2]
[3, 1, 4, 2]
[4, 1, 3, 2]
[1, 4, 3, 2]
[3, 4, 1, 2]
[4, 3, 1, 2]
[4, 3, 2, 1]
[3, 4, 2, 1]
[2, 4, 3, 1]
[4, 2, 3, 1]
[3, 2, 4, 1]
[2, 3, 4, 1]
```

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



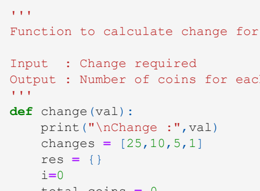
(a) Draw a table showing the intermediate distance values all the nodes at each iteration of the algorithm.

(b) Show the final shortest-path tree.

(a) The Dijktra's algorithm with intermediate distance values is as follows (represented as intermediate distance, parent node):

From A →	B	C	D	E	F	G	H
A	1,A	∞,-	∞,-	4,A	8,A	∞,-	∞,-
B	1,A	3,B	∞,-	4,A	7,B	7,B	∞,-
C	1,A	3,B	4,C	4,A	7,B	5,C	∞,-
D	1,A	3,B	4,C	4,A	7,B	5,C	8,D
E	1,A	3,B	4,C	4,A	7,B	5,C	8,D
G	1,A	3,B	4,C	4,A	6,G	5,C	6,G
F	1,A	3,B	4,C	4,A	6,G	5,C	6,G
H	1,A	3,B	4,C	4,A	6,G	5,C	6,G

(b) The final shortest-path tree starting from A is given as is :



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 : $O(1)$ Time Complexity*

The greedy solution hint tells us that to minimize the number of coins involved, we need to utilize the maximum value coins as much as possible. The algorithm is as follows :

- Start from the highest denomination
- If the denomination is less than the change required, we subtract the highest amount we can make using the denomination which is less than the change from the change. The result now becomes the new change.
- We move to the next largest denomination and repeat for all the denominations.

Since the loop runs for how much denominations are present, the time complexity is $O(1)$, since the number of denominations are fixed in this case.

* Although in this case the solution is of constant time complexity, if the denominations of the coins is also given as input, then the time complexity will be of the order $O(n)$ where n is the number of denominations involved.

In [4]:
```python
'''
Function to calculate change for 25,10,5,1 denomination

Input  : Change required
Output : Number of coins for each denomination
'''
def change(val):
    print("\nChange :",val)
    changes = [25,10,5,1]
    res = {}
    i=0
    total_coins = 0
    for i in range(4):
        res[changes[i]] = val//changes[i]
        total_coins+=res[changes[i]]
        val = val - changes[i]*(val//changes[i])
        i-=1
    print('Number of coins required :',total_coins)
    for i in res:
        print(i,"-",res[i],"coins")

# Testing the code

change(25)
change(100)
change(1)
change(88)
```

```
Change : 25
Number of coins required : 1
25 - 1 coins
10 - 0 coins
5 - 0 coins
1 - 0 coins

Change : 100
Number of coins required : 4
25 - 4 coins
10 - 0 coins
5 - 0 coins
1 - 0 coins

Change : 1
Number of coins required : 1
25 - 0 coins
10 - 0 coins
5 - 0 coins
1 - 1 coins

Change : 88
Number of coins required : 7
25 - 3 coins
10 - 1 coins
5 - 0 coins
1 - 3 coins
```

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 : $O(n^3)$ Time Complexity

We can try to fit parantheses over all possible combinations which will be tedious. Instead we can try to approach this in a dynamic programming way.

Let `shapes` denote the array of shapes of the matrix and `dp[i][j]` denote the cost of computing the product of matrices from position `i` to `j`. To compute the best way to calculate the product from `i` to `k`, we try to first compute the product from `i` to `k` and `k+1` to `j`, and then find the product between the results. The cost of computing product `i` to `k` is `dp[i][k]` and the cost of computing product from `k+1` to `j` is `dp[k+1][j]`. The cost of computing the resulting product will be `shapes[i-1]*shapes[k]*shapes[j]`. Thus, the dynamic programming subproblem is given as :

$$dp[i][j] = \min_{i \leq k < j} dp[i][k] + dp[k+1][j] + shapes[i-1] * shapes[k] * shapes[j]$$

Thus, we can solve the problem using dynamic programming as follows :

- We have array `shapes` of size `n` where `shapes[i],shapes[i+1]` represents the size of the i'th matrix
- Initialize an array `dp[n][n]` with infinity
- Initialize the diagonal elements as 0, this means that the cost of multiplying only one matrix is zero
- Initialize an iterator `l` which indicates what is the length of multiplication we are calculating
- While `l` goes from 2 to n
  - Initialize iterator `i` which represents the starting point from which we are going to check the multiplication cost
  - While `i` runs from 1 till n-l+1
    - Initialize `j` as i+l-1 which indicates the ending point till which we are going to check the multiplication cost
    - Initialize iterator `k` which is the point where we split the expression by paranthesization
      - While `k` runs from `i` to `j`
        - We calculate the cost of multiplication as `dp[i][k] + dp[k+1][j] + shapes[i-1]\*shapes[k]\*shapes[j]`
        - If the cost is less than `dp[i][j]`, we set `dp[i][j]` as the above costs
- The resulting lowest operations matrix multiplication will be present in `dp[1][n-1]`

In [5]:
```python
"""
Code to calculate the minimum number of operation matrix multiplication chain

Input  : Shapes of matrix to multiply
Output : Paranthesized expression and number of operations
"""


def matrix_chain_order(shapes):

    # List to store matrix names
    mat = []

    # Number of matrices
    n = len(shapes) - 1

    # Create the name of matrices with given shapes
    for i in range(n):
        mat.append(chr(ord("A") + i) + str(shapes[i]) + "x" + str(shapes[i + 1]))

    # Initialize the dp matrix, defining for size n + 1 for easier coding
    dp = [[float("inf")] * (n + 1) for _ in range(n + 1)]

    # Initialize the dp matrix for storing the paranthesized expressions
    dp_mat = [[""] * (n + 1) for _ in range(n + 1)]

    # Initialize the diagonals as zeros
    for i in range(1, n + 1):
        dp[i][i] = 0
        dp_mat[i][i] = mat[i - 1]

    # Loop to find the minimum operation paranthesization
    for l in range(2, n + 1):
        for i in range(1, n - l + 2):
            j = i + l - 1
            for k in range(i, j):
                # Calculate the number of operations using the i-k and k-j matrices
                q = dp[i][k] + dp[k + 1][j] + shapes[i - 1] * shapes[k] * shapes[j]

                # Find the paranthezised expression
                s = "(" + dp_mat[i][k] + dp_mat[k + 1][j] + ")"

                # Check if the current value is the minimum
                if q < dp[i][j]:
                    dp[i][j] = q
                    dp_mat[i][j] = s

    # Return the answer
    return (dp[1][n], dp_mat[1][n])


# Testing the code

SUB = str.maketrans("x0123456789", "ₓ₀₁₂₃₄₅₆₇₈₉")
operations_len, paranthezised_expr = matrix_chain_order([40, 20, 30, 10, 30, 10, 20, 60])
print(paranthezised_expr.translate(SUB), "-", operations_len, "operations")
```

```
((A₄₀ₓ₂₀(B₂₀ₓ₃₀C₃₀ₓ₁₀))(((D₁₀ₓ₃₀E₃₀ₓ₁₀)F₁₀ₓ₂₀)G₂₀ₓ₆₀)) - 55000 operations
```

7. Algorithms for "Transport Protocols"

(i) CUBIC TCP vs Compound TCP

Congestion control algorithms for TCP are used to set the transmission rate of information packets over the network. These algorithms try to reduce congestion, i.e. situations where the network receives more packets than it can handle, which leads to loss of some packets while trying to use the available resources of the network efficiently. These algorithms control packets' transmission rate by setting a **congestion window**($cwnd$), which is the number of packets to be sent at a time. Another critical parameter in congestion control is the round trip time (RTT), the time interval between sending a packet and receiving acknowledgement of receiving a packet. It is used both to measure congestion and usually the time interval used to update the congestion window.

CUBIC TCP

CUBIC TCP algorithm is a loss-based congestion control algorithm. The algorithm is given as follows :

Variables :

- β : Multiplicative decrease factor (set to 0.7)
- W_{max} : Window size just before the last reduction (initially set via guess)
- T : Time elapsed since the last window reduction
- C : A scaling constant (set to 0.4)
- $cwnd$: The congestion window at the current time

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

- Before updating $cwmd$ we have to check if we are in TCP Friendly region. In this, we check if we can acheive same throughput as Standard TCP, whose window update adjusted to CUBIC given as $W_{tcp}(T) = W_{max}\beta + 3\frac{1-\beta}{1+\beta}\frac{t}{RTT}$. If W_{tcp} is larger than W_{cubic}, we are in TCP friendly region and thus $cwmd$ is updated to W_{tcp}, else we update $cwmd$ to be W_{cubic}.
- On encountering a loss, we update $cwmd$ as $\beta \times cwmd$. If the value of W_{max} is less than $cwmd$ during loss, then it remains unchanged, else W_{max} is updated as $\beta \times W_{max}$

Compound TCP

(Note: Here $cwmd$ is different from what it meant in the previous section, win here means what $cwmd$ meant in the previous section)

Compound TCP algorith is a hybrid algorithm which uses both delay-based and loss-based information to adjust win (the congestion window in this case). The algorithm sets win as $win = \min(cwmd + dwmd, awmd)$, where $cwmd$ is the loss-based component, $dwmd$ is the delay-based component and $awmd$ is the advertised window from the receiver.

- The loss-based component $cwmd$ is updated similar to normal TCP by Additive Increase Multiplicative Decrease (AIMD) as follows :

$$cwmd(t+1) = \begin{cases} cwmd(t) + \frac{1}{cwmd(t)+dwmd(t)} & \text{No loss occurs} \\ 0.5 \times cwmd(t) & \text{Loss occurs} \end{cases}$$

- The delay-based component $dwmd$ is updated as by the following rule :
  - We set two parameters $baseRTT$, which is the minimal RTT observed after the connection is started, and $sRTT$, which is the current RTT observed, which could be greater due to queueing.
  - We calculate $Expected\_Throughput = \frac{win}{baseRTT}$ and $Actual\_Throughput = \frac{w}{sRTT}$ and $diff = (Expected\_Throughput - Actual\_Throughput) \times baseRTT$ which denotes the data which has been backlogged in the queue. Thus if the $diff$ reaches a threshold γ, we conclude that the network has reached congestion and updated $dwmd$ as follows :

$$dwmd(t+1) = \begin{cases} dwmd(t) + \max(0, \alpha \times win(t)^k - 1) & diff < \gamma \\ \max(0, dwmd(t) - \zeta \times diff) & diff \geq \gamma \\ \max(0, win(t) \times (1-\beta) - cwmd/2) & \text{Loss occurs} \end{cases}$$

- The delay-based component helps in loss reduction by predicting possible congestion and reducing the window appropriately.

Similarities

- Both algorithms use packet losses information for adjusting the congestion window size.
- Both algorithms have a non-linear update of the congestion window, CUBIC via its cubic function update and Compound via its delay-based algorithm.
- Both algorithms try to maximize the network efficiency, CUBIC via aggressive increase of window size when far away from W_{max} and Compound via reducing packet losses through prediction of packet losses in the delay-based window.

Differences

- Compound has a delay-based component, unlike CUBIC, which solely relies on packet loss for update
- Packet loss in Compound is much lower than CUBIC due to the aforementioned predictive reduction of packet losses via the delay-based window.
- CUBIC is RTT independent, whereas Compound is RTT dependent due to its delay-based component

(ii) TCP Fairness

The fairness of a TCP control protocol is defined by how the protocol provides equal bandwidth to competing flows of packets. If N flows compete for the same bottleneck bandwidth, then each should receive $1/N$ capacity of the bandwidth for the flow to be fair.

For an extensive network like the internet where multiple congestion algorithms are running parallelly, to ensure overall fairness, we should check for fairness on three dimensions :

- Intra Protocol Fairness: Fairness among flows with the same congestion algorithms
- Inter Protocol Fairness: Fairness among flows with different congestion algorithms
- RTT Fairness: Fairness among flows with different RTT values.

We need metrics to measure the fairness of the algorithm. Let there be n competing flows with each flow i receiving T_i data. One such metric which can be used is Jain Index. It is given as follows :

- **Jain Index :** Let O_i be max-min optimal bit-rate, then $J(x_1, x_2, x_3, \ldots, x_n) = \frac{\left(\sum_{i=1}^n x_i\right)^2}{n\sum_{i=1}^n x_i^2}$ where $x_i = T_i/O_i$. $J \in [\frac{1}{n}, 1]$. This metric is used to compare different configurations regardless of bottleneck bandwidth. To get the fairness metrics we can run simulations for the algorithm in a testbed and calculate fairness from this result.

CUBIC has good RTT fairness because it is independent of RTT, while Compound does not have good RTT fairness due to its dependence on RTT. CUBIC and Compound both have good intra-fairness due to CUBIC rapid changes to W_{max} for different flows, and Compound enables it via lowering its delay-based window on high queueing.

(iii) Design for More Efficient and Fairer TCP Congestion Control Algorithm:

To increase the efficiency and fairness of TCP Congestion Control, we have to keep in mind the following points :

- Using delay-based methods will cause RTT unfairness due to the dependence on RTT; thus, we have to stick with loss-based algorithms
- We need to reduce packet losses which will help improve the efficiency of the congestion algorithm

The algorithm proposed is as follows :

- The $cwmd$ will increase as follows :

$$cwmd(t) = c\sqrt{t}$$

where t is the time since the last window reduction, and c is the factor of increase of the window. Initially, we start with a high value of c to aggressively search for good window size, and then when we start detecting losses, we cut both the window size by a factor of β, i.e. $cwmd_{new} = \beta cwmd$, where β is the reduction factor. We also reduce c by a scaled value of $\beta \implies c_{new} = k\beta c$ where k is the scaling factor. We will increase the value of β if $t > T_o$, i.e. the time taken for reaching a loss is greater than a given constant T_o, and decrease the value of β if $t < T_o$, i.e. the time taken for reaching a loss is lesser than T_o.

Initially, the algorithms will aggressively compete for bandwidth, but over time, the scaling factor and the square root factor will mellow the aggressiveness, and the TCP algorithms will slowly grow overtime at the end, which helps with inter and intra-fairness. Moreover, since it is a real-time algorithm independent of RTT, it is RTT fair too.

Due to the square root function, we start aggressively at the beginning along with the high c value, which means that although initially there will be some losses, the bandwidth usage will be very high, unlike CUBIC, which means that efficiency is higher compared to CUBIC. Also, after the algorithm has settled, the losses will be much less than CUBIC since the increase is not very aggressive like it is done in CUBIC after crossing the W_max, so loss efficiency will also be higher. This algorithm will work very well in high bandwidth networks and utilize them as much as possible.

8. Algorithms for "Transport Protocols"

(i) PageRank :

PageRank is a link analysis algorithm that analyses a hyperlinked set of documents and assigns numerical weightage, indicating the documents' relative importance to the set. The metric used for assigning the weightage to a particular document is the number and importance of the pages linked to it and the likeliness of a user visiting the page.

The PageRank algorithm defines the metric as the probability distribution of the likelihood of a random surfer selecting a given page. Let there be a hyperlink of N documents. Let us define $PR(u)$ as the Page Rank value of a page u, and let B_u be set of pages linking to u. Let $L(u)$ be the number of outbound links from u. There is also a damping factor d, which denotes the user's probability to continue surfing through the web pages. Thus, $1-d$ represents the user's probability of stopping surfing at the given page. The PageRank formula is recursively given as :

$$PR(u) = \frac{1-d}{N} + d\sum_{v \in B_u} \frac{PR(v)}{L(v)}$$

The above equation is recursively defined; thus, we need an iterative solution to reach the final answer.

Let $\mathbf{R} = [PR(p_1) \quad PR(p_2) \quad PR(p_3) \quad \ldots \quad PR(p_N)]^T$ be the final PageRanks of the hyperlink, then $\mathbf{R}$ is the solution of the following equation :

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

Here, the matrix on the RHS is the adjacency matrix, and $\ell(p_i, p_j)$ is the ratio between several links outbound from page j to i to the total outbound links from j. ℓ is defined such that $\sum_{i=1}^N \ell(p_i, p_j) = 1$. Thus, the matrix is a stochastic matrix, which means we can start with random initialization of $\mathbf{R}$ and iteratively substitute $\mathbf{R}$ in the above equation until the value of $\mathbf{R}$ converges to the required solution. Moreover, since the adjacency matrix has a large eigengap, we can solve $\mathbf{R}$ with high accuracy on a few iterations itself.

(ii) Video Sites Search Algorithm

Search algorithms for videos are unique because since they do not represent textual context, we need to extract some **metadata** so that we can use it to check if the video matches the user search query. Some of the standard metadata used for the video is:

- **Internal Metadata:** These are details that are embedded in the video itself, like the coding quality of the video, author of the video, date created and other similar information. _ **External Metadata:** There are metadata extracted from the page in which the video is displayed, like title and description of the video, filename of the video, and tags associated with the video. One more piece of metadata which is commonly available is the transcript/subtitles of the video, which can be used to extract useful information from the video.

Apart from the above metadata, we can extract information from the content of the video to be used for searching. Some of the examples are :

- **Video Based Extraction:** From frames in the video, we can try to extract text present in frames (chryons) or some valuable descriptors like colour, texture, shape, motion, and many more using machine learning methods or classical image processing techniques. _ **Audio Based Extraction:** For videos that do not have transcripts/subtitles available, transcript generation algorithms can be used to extract the video's speech information, which can be later used in the search context.

Combining all the above features gives us a textual representation of the video, which can later be used in matching a query to a relevant video. This can be done using many different algorithms. Next, the resulting videos are ranked based on a combination of different metrics as given below :

- **Relevance:** How relevant is the video to the search query
- **Number of Views & Engagement:** How popular the video is which can be tracked by a combination of the number of views and engagements like likes and comments
- **Length:** Length of the video is sometimes used to rank the videos, especially in the YouTube algorithm

(iii) Search Framework For YouTube:

We aim to provide a search framework for a video hosting platform like YouTube. The search algorithm aims to retrieve the most relevant video for the user. For this, we first try to extract all possible videos related to the query given by the user. For this, we can use the metadata and extraction methods described in the previous section. For retrieval, we can use keyword extraction from both queries and video information and use the **reverse indexing** method, which matches various keywords to different videos. We can apply reverse indexing to retrieve all videos remotely relevant to the user query using the keywords from the queries.

The next step consists of ranking these videos based on their relevance to the user queries. Some basic engagement metrics like the number of views, likes, dislikes, the number of subscribers of the video creator can be used, but simply using the engagement information for scoring will not lead to good results due to the following :

- Many videos contain identical content or content derived from other content, e.g. reaction videos, parody videos, song remixes. Sometimes these videos have more engagement with the audience than the intended video, which can prove problematic to use Many videos contain 'clickbait', i.e., misleading thumbnails that make people expect something from the video and make them click it but ultimately fail to deliver it. These videos also contain high engagement, but they are usually rarely the intended target of most search queries.

To combat the above issue, we will use two more essential metrics :

- **Video Originality:** We try to tag videos based on the originality of the video. Original videos of music shows and many more will get the original tag, parody, remixes, and others will get their tag. Thus, unless the user specifically asks for the remix, parody or other tags, the videos with the original tag will occur higher on our search list.
- **Channel Originality:** We try to score the originality of the channels too based on the number of original videos they have created. Videos created by original channels will have a higher ranking. Some important channels like official channels of companies and government organizations can also be given high originality scores.
- **User Retention:** User retention is a very important metric to identify clickbait videos. Usually, one would expect clickbait videos to have many dislikes, but people usually tend to ignore clickbait videos and move on to other videos. Thus, we need to have the metric of user retention to identify clickbait. Usually, when people realize the video is clickbait, their stay time is meagre. Thus, we measure the percentage of viewers who have clicked away before reaching x% of the video and use this metric for ranking the videos. Another method can be used to analyze the thumbnails of the video using deep learning methods and classify whether the video is a clickbait video.

Using a linear combination of the above metrics for relevance scoring where the coefficients can be tuned using A/B testing can effectively improve the search engine.

References :

- CUBIC: A New TCP-Friendly High-Speed TCP Variant
- A Compound TCP Approach for High-speed and Long Distance Networks
- Comparative Study of TCP Protocols: A Survey
- The Anatomy of a Large-Scale Hypertextual Web Search Engine
- Video search engine