**NAME: Sneha Sanjeevkumar Baragi**

**ROLLNO: 125**

**DIV:F**

**USN: 01FE23BCI054**

**COURSE: Design and analysis of algorithms**

**-------------------------------------------------------------------------------------------------------------**

**KEY CONCEPTS / THINGS THAT I LEARNED IN THIS COURSE**

1. **TIME COMPLEXITY ANALYSIS:**

**Task 1:**

**Write a recursive C++ program for Towers of Brahma problem and record the time taken for following given input size. Use a spreadsheet application and plot a graph presenting your results.**

**-> Program:**

**#include <iostream>**

**#include <ctime>**

**using namespace std;**

**void towers(int n, string from, string to, string aux) {**

**if (n == 1) {**

**cout << "Move disc 1 from " << from << " to " << to << endl;**

**return;**

**}**

**towers(n-1, from, aux, to);**

**cout << "Move disc " << n << " from " << from << " to " << to << endl;**

**towers(n-1, aux, to, from);**

**}**

**int main() {**

**int n;**

**string A = "A", B = "B", C = "C";**

**cin >> n;**

**// Start measuring time**

**clock\_t start = clock();**

**towers(n, A, C, B);**

**// Stop measuring time**

**clock\_t end = clock();**

**double elapsed = double(end - start) / CLOCKS\_PER\_SEC;**

**cout << "Time taken for execution: " << elapsed << " seconds" << endl;**

**return 0;**

**}**

|  |  |
| --- | --- |
| Table: |  |
| X-axis | Y-axis |
| Input | Time |
| 1 | 0.649 |
| 2 | 0.65 |
| 3 | 0.937 |
| 4 | 1.554 |
| 5 | 1.98 |
| 6 | 3.365 |

**Task 2:**

**Compare and present the orders of growth of linear search and binary search for the best-case, worst-case and average case inputs for the given below input sizes. Use rand() function to generate the random array inputs between size 0 to 1000. Present your results using a spreadsheet application with an appropriate graphs. Do not use any form of print statements while capturing the time for the program.** **The table and graph has to be generated for best case, worst case and average case.**

* **Program**

**#include<iostream>**

**#include<ctime>**

**#include<cstdlib>**

**#include<algorithm> // For std::sort**

**using namespace std;**

**#define SIZE 50000**

**#define ITER 10000 // Number of iterations to average the time**

**int linear\_search(int a[SIZE], int target)**

**{**

**for(int i = 0; i < SIZE; i++)**

**{**

**if(a[i] == target)**

**return i;**

**}**

**return -1;**

**}**

**int Binarysearch(int a[SIZE], int target)**

**{**

**int low = 0;**

**int high = SIZE - 1;**

**int mid;**

**while(low <= high)**

**{**

**mid = (low + high) / 2;**

**if(a[mid] == target)**

**{**

**return mid;**

**}**

**if(target > a[mid])**

**{**

**low = mid + 1;**

**}**

**else**

**{**

**high = mid - 1;**

**}**

**}**

**return -1;**

**}**

**int main()**

**{**

**int a[SIZE];**

**int target;**

**srand(78);**

**for(int i = 0; i < SIZE; i++)**

**{**

**a[i] = rand() % SIZE;**

**cout << a[i] << " ";**

**}**

**cout << "\nEnter the target number to search: ";**

**cin >> target;**

**cout << "\nArray Size: " << SIZE << "\n";**

**// Measure Linear Search time with iterations**

**clock\_t start = clock();**

**for (int i = 0; i < ITER; i++)**

**{**

**linear\_search(a, target);**

**}**

**clock\_t end = clock();**

**double linearTime = double(end - start) / (CLOCKS\_PER\_SEC \* ITER);**

**cout << "Linear Search Time (average per iteration): " << linearTime << " seconds\n";**

**sort(a, a + SIZE);**

**// Measure Binary Search time with iterations**

**start = clock();**

**for (int i = 0; i < ITER; i++)**

**{**

**Binarysearch(a, target);**

**}**

**end = clock();**

**double binaryTime = double(end - start) / (CLOCKS\_PER\_SEC \* ITER);**

**cout << "Binary Search Time (average per iteration): " << binaryTime << " seconds\n";**

**return 0;**

**}**

|  |  |  |  |
| --- | --- | --- | --- |
|  | best |  |  |
| input | linear search | binary search | |
| 100 | 0.001 | 0.001 |  |
| 1000 | 0.001 | 0.001 |  |
| 5000 | 0.001 | 0.001 |  |
| 10000 | 0.001 | 0.001 |  |
| 50000 | 0.001 | 0.001 |  |

|  |  |  |
| --- | --- | --- |
|  | Average |  |
| x axis | yaxis | y axis |
| input size | linear | binary |
| 100 | 0 | 0.135 |
| 500 | 1.87 | 0.45 |
| 1000 | 2.41 | 1.35 |
| 5000 | 2.75 | 1.48 |
| 10000 | 3.85 | 2.51 |
| 50000 | 4.01 | 3.42 |

|  |  |  |
| --- | --- | --- |
|  | worst |  |
| x axis | linear | binary |
| 100 | 0.8 | 0.2 |
| 500 | 1.3 | 0.5 |
| 1000 | 1.9 | 0.89 |
| 5000 | 2.5 | 1.5 |
| 10000 | 3.48 | 1.65 |
| 50000 | 3.91 | 2.5 |

**\*Length of String**

#include <stdio.h>

#include <time.h>

// Function to calculate the length of the string recursively

int getlength(char \*str, int length) {

if(\*str != '\0'){

length++;

str++;

return getlength(str, length);

}

return length;

}

int main() {

char str[100];

int length = 0;

printf("Enter a string: ");

scanf("%s", str);

// Start measuring time

clock\_t start = clock();

// Calculate the length of the string

length = getlength(str, length);

// Stop measuring time

clock\_t end = clock();

double elapsed = ((double)(end - start)) / CLOCKS\_PER\_SEC;

// Print the result

printf("The length is %d\n", length);

printf("Time taken for execution: %f seconds\n", elapsed);

return 0;

}

|  |  |  |
| --- | --- | --- |
|  |  |  |
| length(n) | library | recursion |
| 100 | 3 | 3 |
| 500 | 3 | 3 |
| 1000 | 4 | 4 |
| 5000 | 4 | 4 |
| 10000 | 5 | 5 |
| 50000 | 5 | 5 |

1)What are the challenges in learning ,understanding these concepts?

-> Understanding the time complexity of algorithms like the Tower of Hanoi, linear and binary search, and string length calculations can be challenging due to the abstract nature of theoretical concepts, the need for strong mathematical foundations, and the varying behaviors of different algorithms.

2) What are the challenges in correlating with the real world application?

-> Correlating theoretical concepts with real-world applications presents several challenges, such as bridging the gap between abstract ideas and practical use, adapting algorithms to handle real-world data and constraints, and dealing with the variability and unpredictability of real-life scenarios.

3) How do you determine the most efficient approach /design techniques when solving the complex problems ?

-> To determine the most efficient approach for complex problems, it's crucial to clearly define the problem, break it down into smaller parts, research existing solutions, evaluate different design techniques such as divide and conquer or dynamic programming, and iteratively test and refine the chosen approach based on performance and scalability. This methodical process ensures a well-rounded and efficient solution.

**2. BINARY SEARCH TREE**

#include<iostream>

#include<cstdlib>

using namespace std;

struct tree

{

int data;

struct tree\* left;

struct tree\* right;

};

typedef struct tree TREE;

class Binarysearchtree

{

public:

TREE\* insert\_into\_bst(TREE\*,int);

void inorder(TREE\*);

void preorder(TREE\*);

void postorder(TREE\*);

TREE\* delete\_from\_bst(TREE\*,int);

};

TREE\* Binarysearchtree::insert\_into\_bst(TREE\* root,int data)

{

TREE\* newnode=(TREE\*)malloc(sizeof(TREE));

if(newnode==NULL)

{

cout<<"memory allocation failed"<<endl;

return root;

}

newnode->data=data;

newnode->left=NULL;

newnode->right=NULL;

if(root==NULL)

{

root=newnode;

cout<<"Root node inserted into tree"<<endl;

return root;

}

TREE\* curnode=root;

TREE\* parent=NULL;

while(curnode!=NULL)

{

parent=curnode;

if(newnode->data<curnode->data)

{

curnode=curnode->left;

}

else

{

curnode=curnode->right;

}

}

if(newnode->data<parent->data)

{

parent->left=newnode;

}

else

{

parent->right=newnode;

}

cout<<"Node inserted successfully into the tree"<<endl;

return root;

}

void Binarysearchtree::inorder(TREE\* root)

{

if(root!=NULL)

{

inorder(root->left);

cout<<root->data<<"\t";

inorder(root->right);

}

}

void Binarysearchtree::preorder(TREE\* root)

{

if(root!=NULL)

{

cout<<root->data<<"\t";

preorder(root->left);

preorder(root->right);

}

}

void Binarysearchtree::postorder(TREE\* root)

{

if(root!=NULL)

{

postorder(root->left);

postorder(root->right);

cout<<root->data<<"\t";

}

}

TREE\* Binarysearchtree::delete\_from\_bst(TREE\* root,int data)

{

TREE\* curnode=root;

TREE\* parent=NULL;

TREE\* successor=NULL;

TREE\* p=NULL;

if(root==NULL)

{

cout<<"TREE is empty"<<endl;

return root;

}

while(curnode!=NULL &&curnode->data!=data)

{

parent=curnode;

if(data<curnode->data)

{

curnode=curnode->left;

}

else

{

curnode=curnode->right;

}

}

if(curnode==NULL)

{

cout<<"Item not found"<<endl;

return root;

}

if(curnode->left==NULL)

{

p=curnode->right;

}

else if(curnode->right==NULL)

{

p=curnode->left;

}

else

{

successor=curnode->right;

while(successor->left!=NULL)

{

successor=successor->left;

}

successor->left=curnode->left;

p=curnode->right;

}

if(parent==NULL)

{

free(curnode);

return p;

}

if(curnode==parent->left)

{

parent->left=p;

}

else

{

parent->right=p;

}

free(curnode);

return root;

}

int main()

{

Binarysearchtree bst;

TREE\* root=NULL;

int choice=0,k,c;

int data=0;

TREE\* co;

while(1)

{

cout<<"\n\*MENU\n";

cout<<"1-Insert into BST\n";

cout<<"2-Inorder Traversal\n";

cout<<"3-Preorder Traversal\n";

cout<<"4-Postorder traversal\n";

cout<<"5-Delete from BST\n";

cout<<"Any other option to exit\n";

cout<<"\*\n";

cout<<"Enter your choice: ";

cin>>choice;

switch(choice)

{

case 1:cout<<"Enter the item to insert: ";

cin>>data;

root=bst.insert\_into\_bst(root,data);

break;

case 2: if (root==NULL)

{

cout<<"Tree is empty\n";

}

else

{

cout<<"Inorder traversal is..\n";

bst.inorder(root);

}

break;

case 3: if (root==NULL)

{

cout<<"Tree is empty\n";

}

else

{

cout<<"Preorder traversal is..\n";

bst.preorder(root);

}

break;

case 4: if (root==NULL)

{

cout<<"Tree is empty\n";

}

else

{

cout<<"Postorder traversal is..\n";

bst.postorder(root);

}

break;

case 5:cout<<"Enter the item to be deleted: ";

cin>>data;

root=bst.delete\_from\_bst(root,data);

break;

default:cout<<"Exiting code:\n";

exit(0);

}

}

return 0;

}

1. What are the challenges in learning/understanding above concepts?

Understanding binary search trees (BSTs) can be challenging due to their hierarchical structure and recursive operations, which is difficult for us to grasp. Key challenges include visualizing the tree structure, handling edge cases like duplicate keys or node deletions, and ensuring the tree remains balanced to maintain efficiency. Additionally, understanding the complexity of operations.

2. What are the challenges in correlating with real-world applications?

Relating BSTs to real-world applications can be difficult due to the need to identify appropriate use cases and performance trade-offs. For example, BSTs are ideal for scenarios requiring ordered data, but other structures like hash tables may outperform them for fast lookups. Maintaining an optimal structure in dynamic data scenarios, along with managing memory and understanding when to prefer self-balancing variants, adds complexity to their practical use.

3. How do you determine the most efficient approach/design techniques when solving a problem?

To determine the most efficient approach, analyze the problem requirements (e.g., fast search, insertion, or ordered data) to confirm if a BST is suitable. Compare it with alternatives like hash tables or balanced trees to evaluate performance trade-offs. For static or small datasets, a simple BST may sufficient, but for larger, dynamic datasets, self-balancing trees like AVL or Red-Black Trees are better. Testing with real-world data and evaluating time and space complexity are key to choosing the right design.

**3) DFS AND BFS**

DFS CODE

#include <iostream>

using namespace std;

int v = 5;

int m[10][10] = {{0,1,1,0,0}, {1,0,0,1,1},

{1,0,0,0,1}, {0,1,0,0,0}, {0,1,1,0,0}};

int visited[10];

void dfs(int m[10][10], int v, int source) {

visited[source] = 1;

for (int i = 0; i < v; i++) {

if (m[source][i] == 1 && visited[i] == 0) {

cout << i << "\t";

dfs(m, v, i);

}

}

}

int main() {

int source;

for (int i = 0; i < v; i++)

visited[i] = 0;

cout << "Enter the source vertex: ";

cin >> source;

cout << "The DFS Traversal is... \n";

cout << source << "\t";

dfs(m, v, source);

return 0;

}

BFS CODE

#include <iostream>

using namespace std;

void bfs(int m[10][10], int v, int source) {

int queue[20];

int front = 0, rear = 0, u, i;

int visited[10];

for (i = 0; i < v; i++)

visited[i] = 0;

queue[rear] = source;

visited[source] = 1;

cout << "The BFS Traversal is... \n";

while (front <= rear) {

u = queue[front];

cout << u << "\t";

front++;

for (i = 0; i < v; i++) {

if (m[u][i] == 1 && visited[i] == 0) {

visited[i] = 1;

rear++;

queue[rear] = i;

}

}

}

}

int main() {

int v = 5;

int m[10][10] = {{0,1,1,0,0}, {1,0,0,1,1},

{1,0,0,0,1}, {0,1,0,0,0}, {0,1,1,0,0}};

int source;

cout << "Enter the source vertex: ";

cin >> source;

bfs(m, v, source);

return 0;

}

1) What are the challenges in learning ,understanding the dfs and bfs code?

The challenges we found during learning DFS and BFS can be challenging due to their traversal mechanisms and implementation complexities. DFS explores deeply and backtracks, while BFS traverses level by level, which can be hard to visualize and differentiate without practice. Managing stacks (DFS) or queues (BFS), marking visited nodes, and handling edge cases like cycles and disconnected graphs add to the difficulty.

Recursion in DFS and understanding graph representations (e.g., adjacency list or matrix) often confuse beginners. Debugging issues and applying these algorithms to variants like cycle detection or shortest path require a strong grasp of their fundamentals. BFS’s limitations with weighted graphs and the need to track multiple states can also overwhelm learners

2) What are the challenges in correlating with the real world application?

Correlating DFS and BFS with real-world applications can be challenging due to their abstract **nature and the need to map nodes and edges to tangible entities like cities, roads, or social connections.** Choosing the right graph representation (e.g., adjacency lists or matrices) and simplifying complex, real-world scenarios into manageable graph models can be confusing. Additionally, recognizing when to use DFS (e.g., for backtracking in games) or BFS (e.g., for shortest paths in unweighted networks) requires understanding their specific strengths and limitations in practical contexts

3) How do u determine the most efficient approach /design techniques when solving the complex problems ?

•Use appropriate data structures (adjacency lists, queues, stacks).

•Keep track of visited nodes.

•Stop searching early if possible.

**4) Heap**

#include <iostream>

#include <vector>

using namespace std;

void Heapify(vector<int>& H, int i, int n)

{

int v;

for (int i = n / 2; i >= 1; i--)

{

int v = H[i];

bool heap = false;

while (!heap && 2 \* i <= n) {

int j = 2 \* i;

if (j < n && H[j] < H[j + 1]) {

j=j+1;

}

if (v >= H[j]) {

heap = true;

} else {

H[i] = H[j];

i = j;

}

}

H[i] = v;

}

}

int main() {

vector<int> H = {7, 6, 17, 11, 64, 29, 6, 12, 2};

for (int i = 1; i < H.size(); i++) {

cout << H[i] << " ";

}

cout<<"\n";

int n = H.size();

int i;

Heapify(H, i, n);

cout<<"after heapification"<<endl;

for (int i = 1; i < H.size(); i++) {

cout << H[i] << " ";

}

cout << endl;

return 0;

}

1. What are the challenges in learning/understanding above concepts?

Understanding heaps can be challenging due to their unique properties and implementation. struggle with visualizing the heap's binary tree structure, especially as it is typically implemented using arrays. Grasping the "heap property" (max-heap or min-heap) and how it is maintained during insertion and deletion can be difficult.

2. What are the challenges in correlating with real-world applications?

Relating heaps to real-world applications can be challenging as it requires identifying scenarios where their use is optimal, such as priority queues, shortest path algorithms, or event scheduling. Selecting between different types of heaps based on application requirements like time efficiency or memory constraints adds complexity. Furthermore, implementing a heap in large-scale systems or adapting it for specific use cases like real-time processing requires additional understanding of its strengths and limitations.

3. How do you determine the most efficient approach/design techniques when solving a problem?

To determine the most efficient use of heaps, analyze the specific problem requirements, such as whether fast access to the smallest or largest element is needed. Compare the heap's efficiency with other data structures like balanced trees or arrays based on the operations' time complexity. For general use, binary heaps are simple and effective, while Fibonacci heaps may be preferred in algorithms like Dijkstra's where reduced amortized time for certain operations is advantageous. Testing with the problem's dataset and considering trade-offs like memory overhead ensures the optimal choice.

**5) SORTING**

ALGOROTHMS

Bubble Sort

ALGORITHM BubbleSort(A[0..n-1])

// Sorts a given array using bubble sort

// Input: An array A[0..n-1] of orderable

// elements

// Output: Array A[0...n-1] sorted in

// ascending order

for i <-0 to n - 2 do

for j <- 0 to n - 2 - i do

if A[j+1] < A[j]

swap A[j] and A[j+1]

Selection Sort

ALGORITHM SelectionSort(A[0..n-1])

// Sorts a given array using selection sort

// Input: An array A[0..n-1] of orderable

// elements

// Output: Array A[0...n-1] sorted in

// ascending order

for i<- 0 to n - 2 do

min <-i

for j <- i + 1 to n - 1 do

if A[j] < A[min]

min <- j

swap A[i] and A[min]

Insertion Sort

ALGORITHM InsertionSort(A[0..n-1])

// Sorts a given array using insertion sort

// Input: An array A[0..n-1] of orderable

// elements

// Output: Array A[0...n-1] sorted in

// ascending order

for i<- 1 to n - 1 do

v<- A[i]

j<- i - 1

while j >= 0 and A[j] > v do

A[j + 1] <- A[j]

j<- j - 1

A[j + 1] <- v

Merge Sort

ALGORITHM MergeSort(A[0..n-1])

// Sorts a given A[0..n-1] by recursive mergesort

// Input: An array A[0..n-1] of orderable elements

// Output: Array A[0...n-1] sorted in nondecreasing order

if n > 1

copy A[0...|n/2| - 1 ] to B[0...|n/2| - 1]

copy A[ |n/2| ... n - 1 ] to C[0......| ̄ n/2 ̄| - 1]

MergeSort(B[0...|n/2| - 1] )

MergeSort(C[0......| ̄ n/2 ̄| - 1] )

Merge(B,C, A)

ALGORITHM Merge(B[0...p-1], C[0...q-1], A[0...p+q-1])

// Merges two sorted arrays into one sorted array

// Input: Arrays B[0...p-1] and C[0...q-1] both sorted

// Output: Sorted array A[0...p+q-1]of the elements of B and C

i <- 0

j <- 0

k <- 0

while i < p and j < q do

if B[i] <= C[j]

A[k] <- B[i]

i <- i + 1

else

A[k] <- C[j]

j <- j + 1

k <- k + 1

if i = p

copy C[j...q - 1] to A[k...p + q - 1]

else

copy B[i...p - 1] to A[k...p + q - 1]

Quick Sort

ALGORITHM QuickSort(A[l...r])

// Sorts a subarray by quicksort

// Input: A subarray A[l...r] of A[0...n-1], defined by its left and right indices l and r

// Output: Subarray A[l...r] sorted in nondecreasing order

if l < r

s <-Partition(A[l...r])

QuickSort(A[l...s - 1])

QuickSort(A[s + 1...r])

ALGORITHM Partition(A[l...r])

// Partitions a subarray by using its first element as a pivot

// Input: A subarray A[l...r] of A[0...n-1], defined by its left and right indices l and r (l < r)

// Output: Subarray A[l...r], with split position returned as this functions value

p <-A[l]

i <- l

j <-r + 1

repeat

repeat i <-i + 1 until A[i] >= p

repeat j <- j – 1 until A[j] <= p

swap(A[i] and A[j])

until i >= j

swap (A[i], A[j])

swap (A[l], A[j])

return j

Algorithm for Heap Construction

ALGORITHM HeapBottomUp(H[1..n])

// Constructs a heap from the elements of a given array by the bottom up algorithm

// Input: An array H[1...n] of orderable items

// Output: A heap H[1... n]

for i <-|\_ n/2 \_| downto 1 do

k <- i

v <- H[k]

heap<-false

while not heap and 2 \* k <=n d0

j <- 2 \* k

if j < n

if H[j] < H[j + 1]

j <- j + 1

if v >= H[ j ]

heap<- true

else

H[k] <-H[j]

k <- j

H[k] <- v

C++ CODES

INSERTION SORT

#include <iostream>

#include <vector>

using namespace std;

void read(vector<int>& a) {

int n;

cout << "Enter the size of array: ";

cin >> n;

a.resize(n);

cout << "Reading all array elements...\n";

for (int i = 0; i < n; i++) {

cin >> a[i];

}

}

void display(const vector<int>& a, const string& message) {

cout << message << endl;

for (size\_t i = 0; i < a.size(); i++) {

cout << "a[" << i << "]=" << a[i] << endl;

}

}

void insertionSort(vector<int>& a) {

for (size\_t i = 1; i < a.size(); i++) {

int key = a[i];

int j = i - 1;

while (j >= 0 && key < a[j]) {

a[j + 1] = a[j];

j--;

}

a[j + 1] = key;

}

}

int main() {

vector<int> a;

read(a);

display(a, "Displaying all array elements before sorting");

insertionSort(a);

display(a, "Displaying all array elements after insertion sorting");

return 0;

}

BUBBLE SORT

#include <iostream>

#include <vector>

#include <string> // Include for string

using namespace std;

void read(vector<int>& a) {

int n;

cout << "Enter the size of array: ";

cin >> n;

a.resize(n);

cout << "Reading all array elements...\n";

for (int i = 0; i < n; i++) {

cin >> a[i];

}

}

void display(const vector<int>& a, const string& message) {

cout << message << endl;

for (size\_t i = 0; i < a.size(); i++) {

cout << "a[" << i << "]=" << a[i] << endl;

}

}

void bubbleSort(vector<int>& a) {

int n = a.size();

for (int i = 0; i < n - 1; i++) {

for (int j = 0; j < n - i - 1; j++) {

if (a[j] > a[j + 1]) {

swap(a[j], a[j + 1]); // Use std::swap

}

}

}

}

int main() {

vector<int> a;

read(a);

display(a, "Displaying all array elements before sorting");

bubbleSort(a);

display(a, "Displaying all array elements after bubble sorting");

return 0;

}

QUICK SORT

#include <iostream>

using namespace std;

int partitionq(int l, int h, int a[]) {

int pivot = a[l];

int i = l;

int j = h;

while (i < j) {

while ( a[i] <= pivot) {

i++;

}

while (a[j] > pivot) {

j--;

}

if (i < j) {

swap(a[i], a[j]);

}

}

swap(a[l], a[j]);

return j;

}

void quicksort(int l, int h, int a[]) {

if (l < h) {

int j = partitionq(l, h, a);

quicksort(l, j - 1, a);

quicksort(j + 1, h, a);

}

}

int main() {

int n;

int a[10];

cout<<"enter the number of elements in array"<<endl;

cin>>n;

cout<<"enter the array elements"<<endl;

for(int i=0;i<n;i++)

{

cin>>a[i];

}

int l = 0;

int h = n - 1;

quicksort(l, h, a);

cout << "Sorted array is:" << endl;

for (int i = 0;a[i]!='\0'; i++) {

cout << a[i] << "\t";

}

return 0;

}

HEAP SORT

#include <iostream>

#include <vector>

using namespace std;

void Heapify(vector<int>& H, int i, int n)

{

int v;

for (int i = n / 2; i >= 1; i--)

{

int v = H[i];

bool heap = false;

while (!heap && 2 \* i <= n) {

int j = 2 \* i;

if (j < n && H[j] < H[j + 1]) {

j=j+1;

}

if (v >= H[j]) {

heap = true;

} else {

H[i] = H[j];

i = j;

}

}

H[i] = v;

}

}

int main() {

vector<int> H = {7, 6, 17, 11, 64, 29, 6, 12, 2};

for (int i = 1; i < H.size(); i++) {

cout << H[i] << " ";

}

cout<<"\n";

int n = H.size();

int i;

Heapify(H, i, n);

cout<<"after heapification"<<endl;

for (int i = 1; i < H.size(); i++) {

cout << H[i] << " ";

}

cout << endl;

return 0;

}

1)What are the challenges in learning ,understanding these concepts?

Learning how to sort things in a computer program can be tricky. It's hard to imagine how the computer moves things around, and there are lots of different ways to do it. Each way (or "algorithm") has its own rules and works better in some situations than others. It's also tough to figure out how fast or slow each method is. Writing the code to sort things correctly can be complicated, and it's easy to make mistakes. To learn it well, it helps to start with the easy methods, look at pictures or animations that show how they work, try some examples, and write the code.

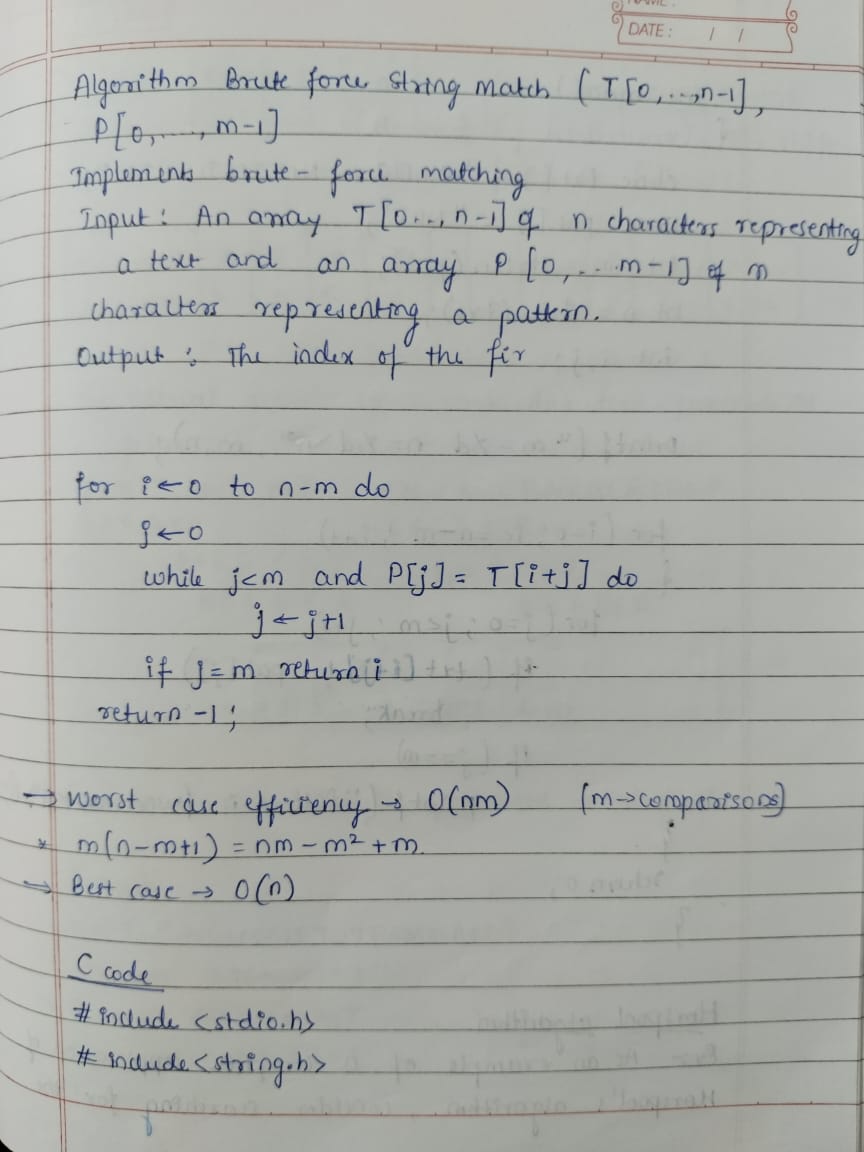
2) What are the challenges in correlating with the real world application?

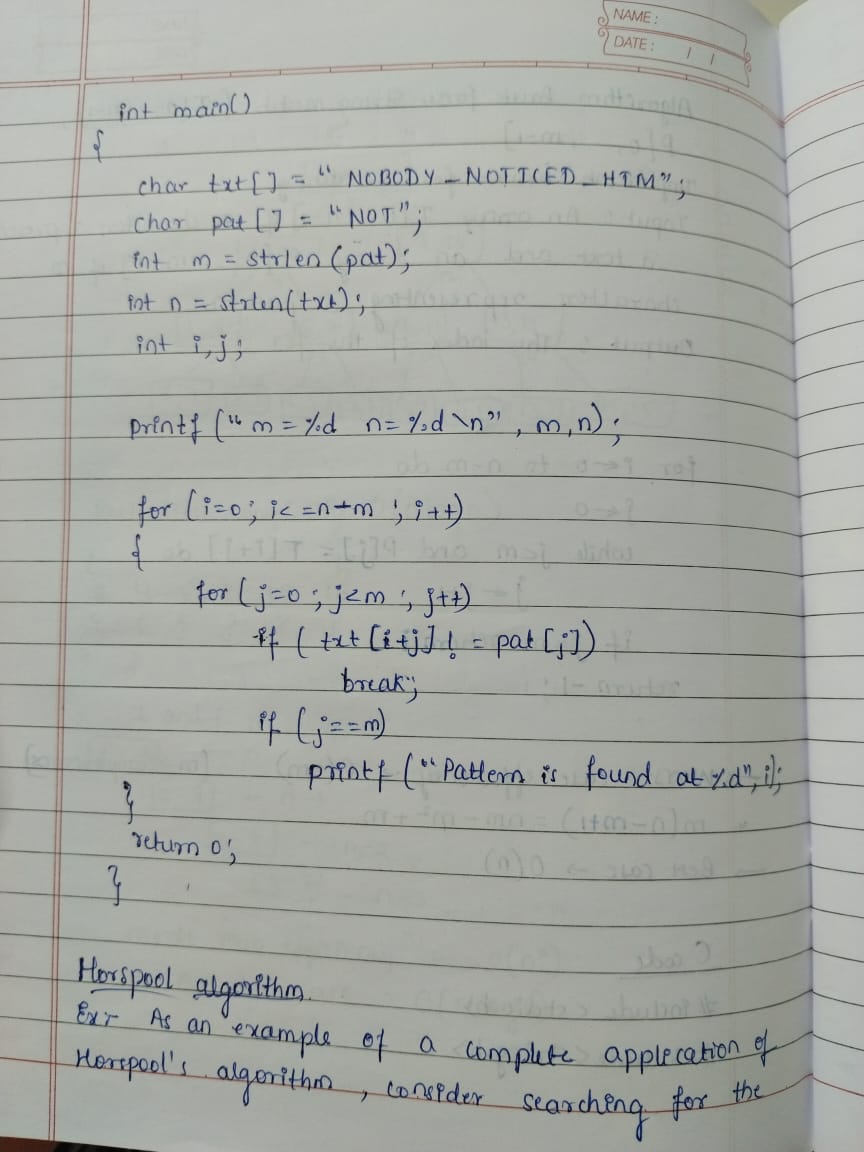
Using sorting in real life is harder than just learning about it in class. Real-world data is messy – it's huge, has mistakes, and keeps changing. Different jobs need different things from sorting – some need to keep things in the same order if they're the same, some don't have much memory to work with, and some need to be super fast. Picking the right sorting method is tricky because there's no perfect one. Plus, sorting is usually just one part of a bigger program, so it has to work with other software. And to make it run as fast as possible, you often have to tweak it

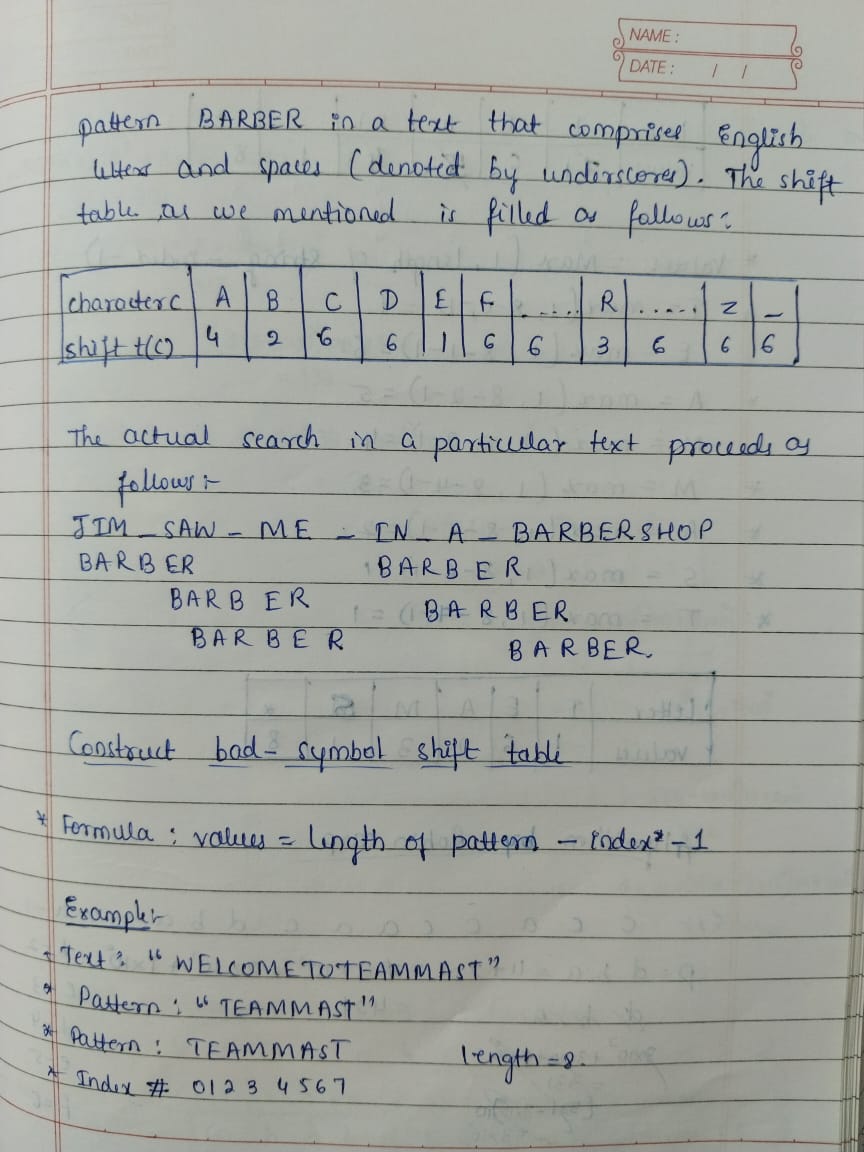
3) How do you determine the most efficient approach /design techniques when solving the complex problems ?

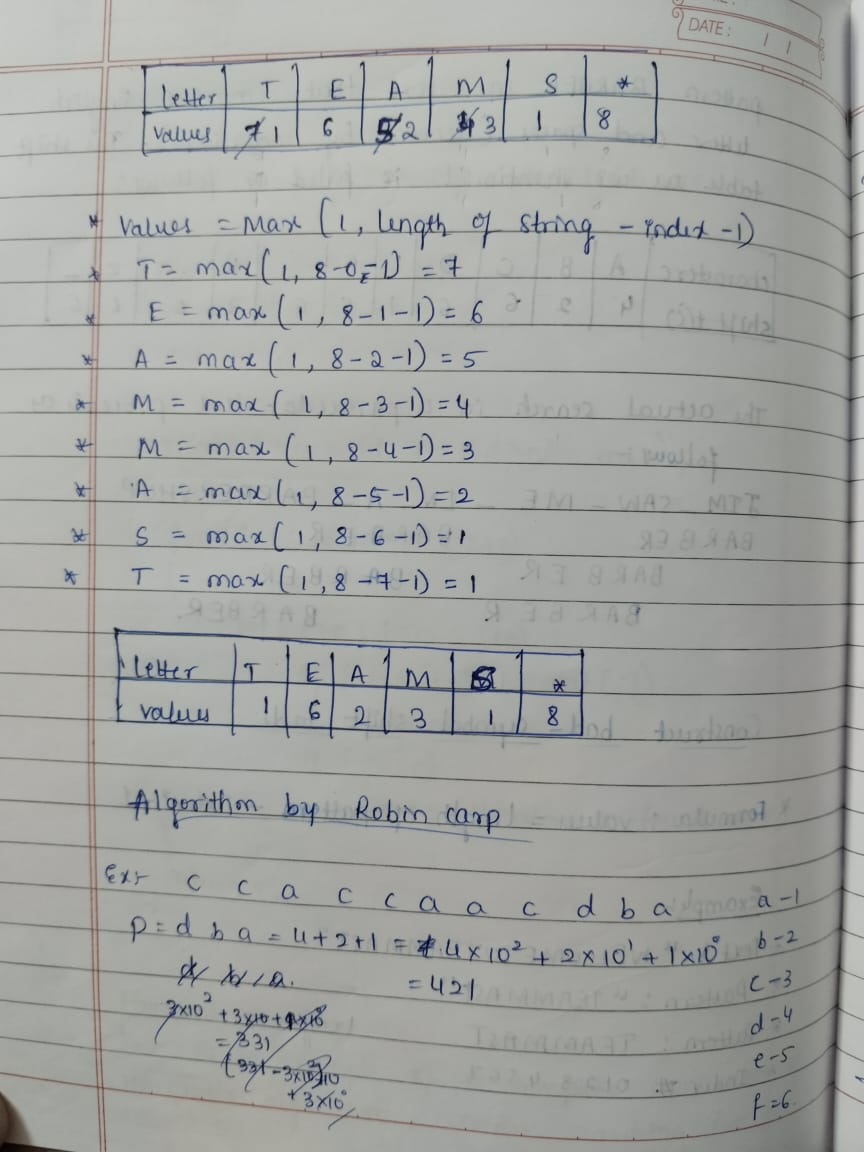
To solve complex problems efficiently, start by thoroughly understanding the problem and breaking it down. Explore different solution approaches, considering various problem-solving paradigms. Analyze these approaches based on time and space complexity, correctness, and ease of implementation, weighing any trade-offs. Employ efficient design techniques like choosing appropriate data structures and algorithms, and applying optimization strategies. Always consider constraints like time, memory, and hardware limitations. Finally, use iterative refinement, including prototyping, testing, and debugging, to continuously improve the solution

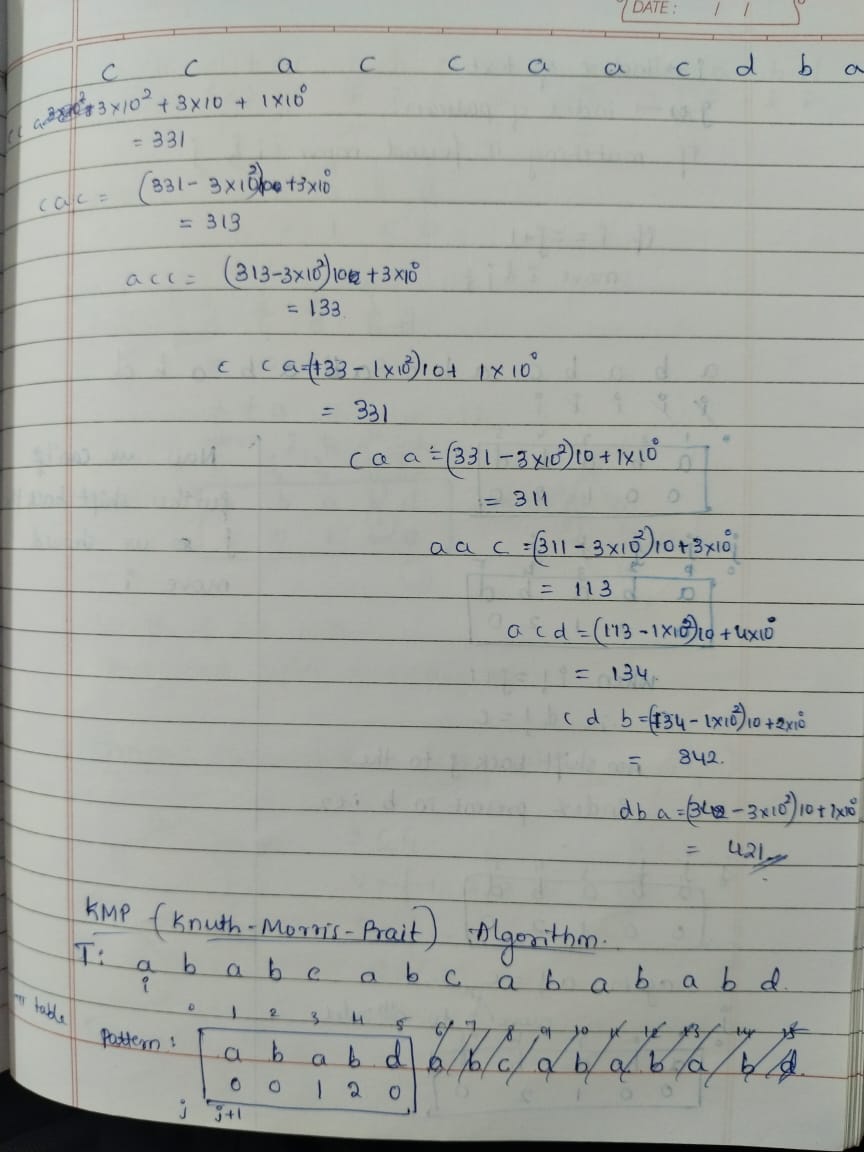
**6) Pattern Searching:**

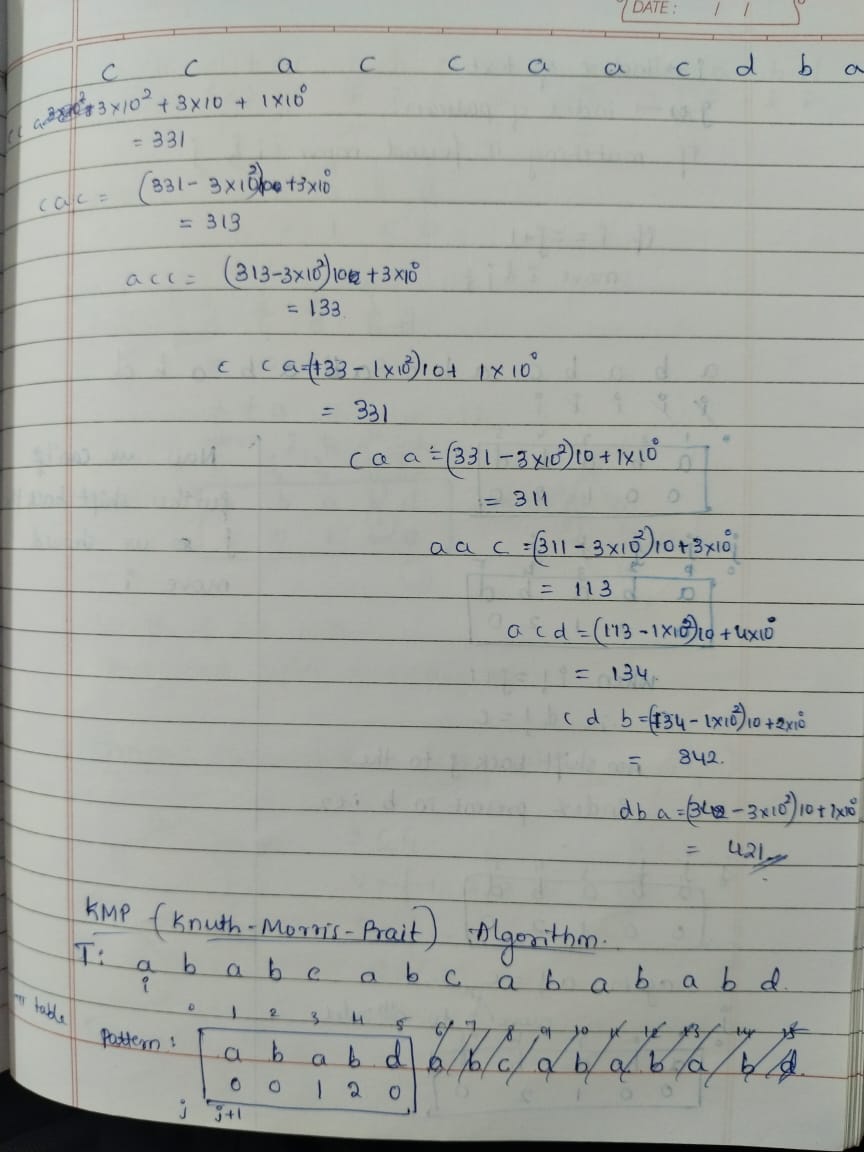


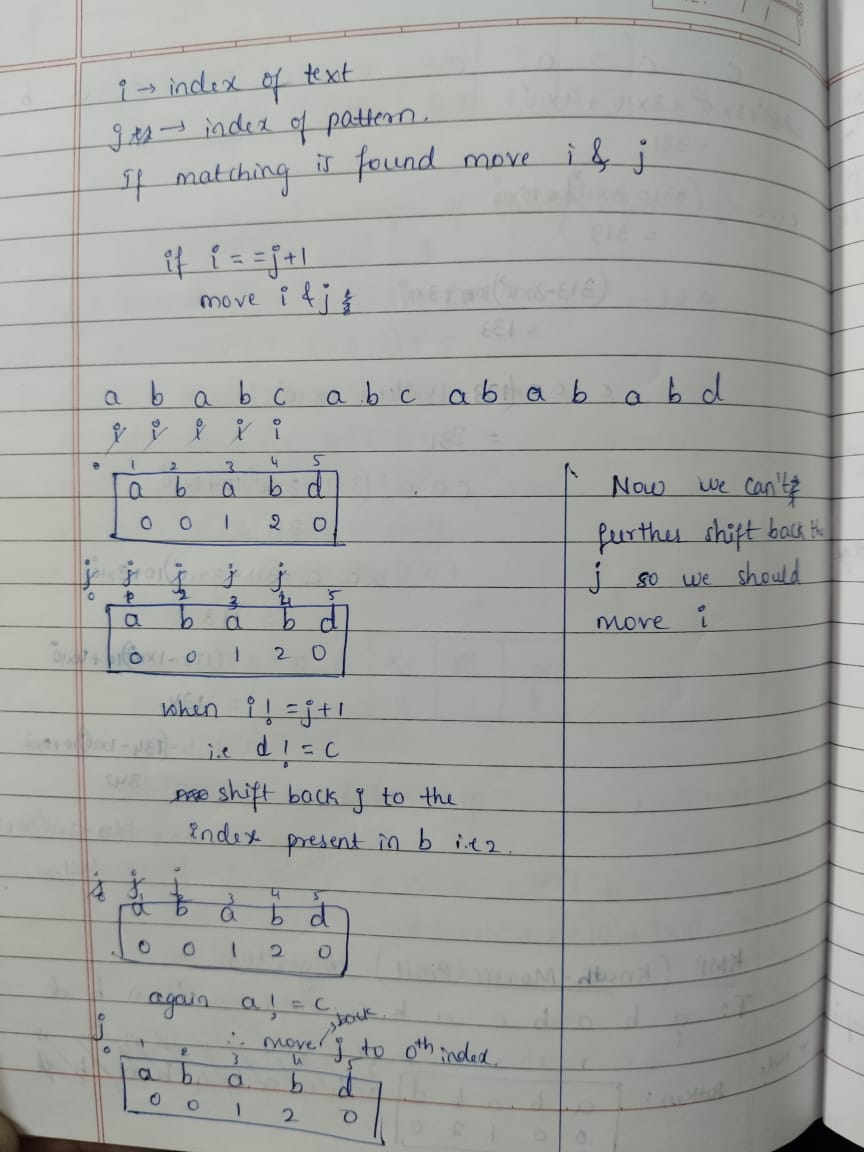


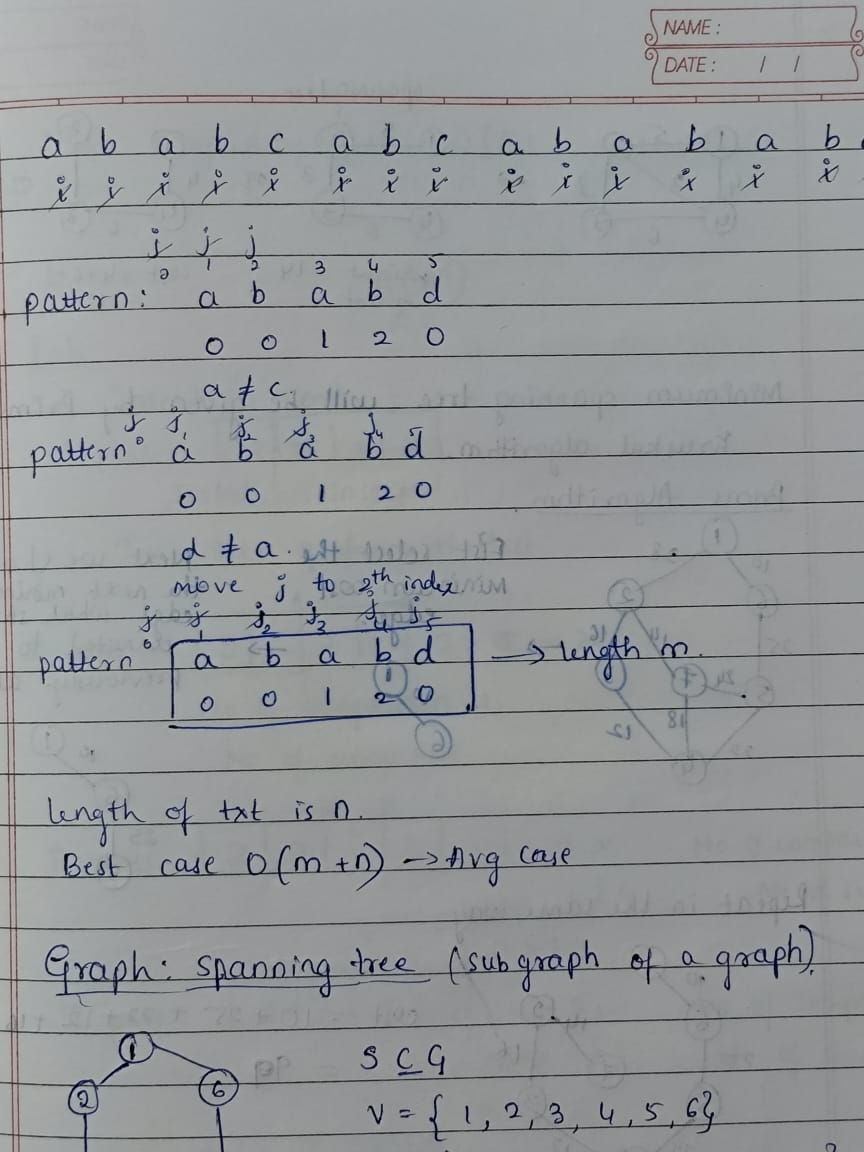












1)What are the challenges in learning ,understanding pattern searching methods like robin carp,boyer moore,brute force string match,kmp?

Learning pattern searching algorithms like brute-force, KMP, Boyer-Moore, and Rabin-Karp presents several challenges. These include grasping the core logic of each algorithm, understanding the pre-processing steps involved in KMP and Boyer-Moore (like building prefix tables or bad character tables), and analyzing their time and space complexities. Implementing these algorithms correctly, especially handling edge cases and boundary conditions, can also be tricky. Furthermore, understanding the trade-offs between these algorithms and choosing the most suitable one for a given scenario requires careful consideration of factors like pattern and text length, and frequency of searches.

In short, the challenges are: understanding each algorithm's logic, mastering pre-processing steps (KMP, Boyer-Moore), analyzing performance, correct implementation (especially edge cases), and choosing the right algorithm for a specific use case.

2) What are the challenges in correlating with the real world application?

Applying pattern searching algorithms like Rabin-Karp, Boyer-Moore, brute-force, and KMP to real-world scenarios presents several challenges. These include handling the sheer scale and diverse nature of real-world data, which can be massive and contain various data types. Real-world patterns themselves can be complex, involving variable lengths, wildcards, approximate matches, or the need to search for multiple patterns simultaneously. Meeting stringent performance requirements, especially in applications like search engines or network security, is crucial. The preprocessing overhead of algorithms like KMP and Boyer-Moore, while improving search speed, must be considered, especially for infrequent searches. Memory usage can also be a constraint, particularly in resource-limited environments. Finally, integrating these algorithms with existing systems adds further complexity.

In short, the challenges are: large and diverse data, complex patterns, performance needs, preprocessing cost, memory limits, and system integration.

3) How do you determine the most efficient approach /design techniques when solving the complex problems?

To determine the most efficient approach for pattern searching with algorithms like Rabin-Karp, Boyer-Moore, brute-force, and KMP, consider these factors:

* **Pattern and text length:** For short patterns and texts, brute-force might be sufficient. For longer ones, KMP, Boyer-Moore, or Rabin-Karp are more efficient.
* **Frequency of searches:** If you're searching for the same pattern many times in different texts, the preprocessing overhead of KMP or Boyer-Moore becomes worthwhile.
* **Alphabet size:** Boyer-Moore performs well with larger alphabets.
* **Expected matches:** If matches are rare, Boyer-Moore tends to be very fast.
* **Memory constraints:** Brute-force uses minimal memory, while KMP requires some extra space for the prefix table.

**7) Graph Algorithm**

**1.Dijkstra’s Algorithm:**

ALGORITHM Dijkstra(G, s)

// Dijkstra’s algorithm for single source shortest path

// Input: A weighted connected graph G(V, E) with non-negative weights and its vertex s

// Output: the length dv of a shortest path from s to v and its penultimate vertex pv for

// every vertex v in V

Initialize(Q) // Initialize vertex priority queue to empty

for every vertex v in V do

dv <- ∞

pv <- null

Insert (Q, v, dv) // Initialize vertex priority in priority queue

ds <- 0

Decrease (Q, s, ds) // Update priority of s with ds

VT<-Ø

for i<-0 to |V| - 1 do

u\*<-DeleteMin(Q)

VT = VT U {u\*}

for every vertex u in V – VT that is adjacent to u\* do

if du\* + w(u\*, u) < du

du<-du\* + w(u\*, u)

pu<-u\*

Decrease (Q, u, du)

**2. Floyd’s Algorithm**

ALGORITHM Floyd (W[1..n,1..n])

// Implements Floyd’s algorithm for all pair shortest path problem

// Input: The weight matrix W of the graph with no negative length cycle

// Output: The distance matrix of the shortest path’s lengths

D W

for k <- 1 to n do

for i <- 1 to n do

for j<-1 to n do

D [i, j] <-min {D[i, j], D[i, k] +D[k, j]}

return D

**3. Warshall’s Algorithm**

ALGORITHM Warshall (A[1..n,1..n])

// Implements Warshall’s algorithm for computing transitive closure

// Input: The adjacency matrix A of a diagraph with n vertices

// Output: The transitive closure of the diagraph

R(0) <-A

for k <-1 to n do

for i <- 1 to n do

for j <-1 to n do

R(k) [i, j] <-R(k-1) [i, j] or (R(k-1) [i, k] and R(k-1) [k, j])

return R(n)

**4. Kruskal’s Algorithm**

ALGORITHM Kruskal(G)

// Kruskal’s algorithm to construct a minimum spanning tree

// Input: A weighted connected graph G(V, E)

// Output: ET, the set of edges composing of MST of G

sort E in nondecreasing order of the edge weights w(ei1) <= … <= w(ei|E|)

ET <- Ø

ecounter <-0

k <-0

while ecounter < |V| - 1 do

k <-k + 1

if ET U {eik} is acyclic

ET <-ET U {eik}

ecounter <-ecounter + 1

return ET

**5. Prim’s Algorithm**

ALGORITHM Prim(G)

// Prim’s algorithm to construct a minimum spanning tree

// Input: A weighted connected graph G(V, E)

// Output: ET, the set of edges composing of MST of G

VT <- {vo}

ET <-Ø

for i <-1 to |V| - 1 do

find a minimum weight edge e\* = (v\*,u\*) along all the edges (v, u) such that

v is in VT and u is in V – VT

VT <-VT U {u\*}

ET <-ET U {e\*}

return ET

**6. Bellman-Ford Algorithm**

d[s] ← 0

for each v ∈ V − {s}

do d[v] ← ∞ // initialization

for i ← 1 to |V| − 1

do for each edge (u, v) ∈ E

do if d[v] > d[u] + w(u, v)

then d[v] ← d[u] + w(u, v) // relaxation step

for each edge (u, v) ∈ E

do if d[v] > d[u] + w(u, v)

then report that a negative-weight cycle exists

At the end, d[v] = δ(s, v), if no negative-weight cycles.

1)What are the challenges in learning ,understanding GRAPH ALGORITHMS concepts?

->Understanding graph algorithms poses challenges due to their complex structures and the need for a strong foundation in discrete mathematics and graph theory. The diverse range of algorithms, like Dijkstra's, Bellman-Ford, and Kruskal's, each come with unique applications and complexities. Additionally, analyzing their efficiency and translating theoretical concepts into real-world applications can be daunting. Handling edge cases, such as disconnected graphs or negative weights, further adds to the complexity.

2) What are the challenges in correlating with the real world application?

->Correlating graph algorithms with real-world applications can be challenging due to the complexity and variability of real-world data, which often differs significantly from the clean, structured data used in academic examples. Additionally, real-world scenarios may involve dynamic and large-scale graphs, requiring efficient handling of data updates and optimizations that aren't as straightforward. Moreover, practical constraints such as computational resources, time limits, and data quality can further complicate the direct application of theoretical algorithms, necessitating customized solutions and adaptations.

3) How do you determine the most efficient approach /design techniques when solving the complex problems ?

->Determining the most efficient approach for solving complex problems involving graph algorithms requires a systematic and iterative process. First, clearly define the problem and its constraints. Then, break the problem down into smaller, manageable parts and understand the underlying graph structure. Research and evaluate different algorithms, considering factors like time and space complexity, scalability, and ease of implementation. Use techniques such as divide and conquer, dynamic programming, or greedy algorithms based on the problem requirements. Prototype and test the chosen approach, analyze its performance, and iteratively refine it. Combining theoretical knowledge with practical experimentation ensures an efficient and effective solution.