LAB MANUAL

Laboratory Practice III

(DESIGN AND ANALYSIS OF ALGORITHMS)

**Year : 2022 - 2023**

**Course Code : 410246**

**Semester : VII**

**Branch : Computer Engineering**

**Prepared by:**

**Ms. Manisha Ambekar**



**Dr. D Y Patil Pratishthan’s**

**Dr. D.Y. Patil Institute of Engineering, Management and Research, Akurdi, Pune**



**1. PROGRAM OUTCOMES:**

|  |  |
| --- | --- |
| **PROGRAM OUTCOMES (POS)** | |
| **PO-1** | Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems**(Engineering knowledge).** |
| **PO-2** | Identify, formulate, review research literature, and analyze complex engineering problems reaching  substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences **(Problem analysis)**. |
| **PO-3** | Design solutions for complex engineering problems and design system components or processes that  meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations **(Design/development of solutions)**. |
| **PO-4** | Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions **(Conduct**  **investigations of complex problems)**. |
| **PO-5** | Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the  limitations **(Modern tool usage)**. |
| **PO-6** | Apply reasoning informed by the contextual knowledge to assesssocietal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice  **(The engineer and society)**. |
| **PO-7** | Understand the impact of the professional engineering solutions in societal and environmental  contexts, and demonstrate the knowledge of, and need for sustainable development **(Environment and sustainability)**. |
| **PO-8** | Apply ethical principles and commit to professional ethics and responsibilities and norms of the  engineering practice **(Ethics)**. |
| **PO-9** | Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings **(Individual and team work)**. |
| **PO-10** | Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions  **(Communication)**. |
| **PO-11** | Demonstrate knowledge and understanding of the engineering and management principles and apply  these to one’s own work, as a member and leader in a team, to manage projects and in multidisciplinary environments **(Project management and finance)**. |
| **PO-12** | Recognize the need for, and have the preparation and ability to engage in independent and life-long  learning in the broadest context of technological change **(Life-long learning)**. |

**2. PROGRAM SPECIFIC OUTCOMES**

|  |  |
| --- | --- |
| **PROGRAM SPECIFIC OUTCOMES (PSO's)** | |
| **PSO-1** | **Professional Skills:** The ability to understand, analyze and develop computer programs in the areas related to algorithms, system software, multimedia, web design, big data analytics, and networking for efficient design of computer-based systems of varying complexity. |
| **PSO-2** | **Problem-Solving Skills:** The ability to apply standard practices and strategies in software project development using open-ended programming environments to deliver a quality product for business success. |
| **PSO-3** | **Successful Career and Entrepreneurship:** The ability to employ modern computer languages, environments, and platforms in creating innovative career paths to be an entrepreneur, and a zest for higher studies. |

*Companion Course: Design and Analysis of Algorithms (410241),*

*Machine Learning (410242),*

*Block chain Technology (410243)*

**3. COURSE OBJECTIVES:**

● Learn effect of data preprocessing on the performance of machine learning algorithms

● Develop in depth understanding for implementation of the regression models.

● Implement and evaluate supervised and unsupervised machine learning algorithms.

● Analyze performance of an algorithm.

● Learn how to implement algorithms that follow algorithm design strategies namely

Divide and Conquer, greedy, dynamic programming, backtracking, branch and bound.

● Understand and explore the working of Block chain technology and its applications

**4. COURSE OUTCOMES:**

After completion of the course, students will be able to

CO1: Apply preprocessing techniques on datasets.

CO2: Implement and evaluate linear regression and random forest regression models.

CO3: Apply and evaluate classification and clustering techniques.

CO4: Analyze performance of an algorithm.

CO5: Implement an algorithm that follows one of the following algorithm design strategies:

Divide and conquer, greedy, dynamic programming, backtracking, branch and bound.

CO6: Interpret the basic concepts in Block chain technology and its applications

**Course Contents**

**Group A: Design and Analysis of Algorithms**

**5. ATTAINMENT OF PROGRAM OUTCOMES AND PROGRAM SPECIFIC OUTCOMES:**

|  |  |  |  |
| --- | --- | --- | --- |
| **S.No** | **Experiment** | **Program Outcomes Attained** | **Program Specific Outcomes**  **Attained** |
| **1** | **Write a program non-recursive and recursive program to calculate Fibonacci numbers and analyze their time and space complexity** | **PO-2, PO-3** | **PSO-1** |
| **2** | **Write a program to implement Huffman Encoding using a greedy strategy** | **PO-3** | **PSO-1, PSO2** |
| **3** | **Program to implement fractional knapsack problem using greedy method** | **PO-3** | **PSO-1, PSO2** |
| **4** | **Implement 0/1 Knapsack problem using Dynamic Programming.** | **PO-3, PO-12** | **PSO-1 , PSO2** |
| **5** | **Implement N Queen's problem using Back Tracking.** | **PO-3** | **PSO-1, PSO2** |
| **6** | **Mini Project - Implement the Naive string matching algorithm and Rabin-Karp algorithm for string matching. Observe difference in working of both the algorithms for the same input.** | **PO-3, PO-12** | **PSO-1 , PSO2** |

**6**. **MAPPING COURSE OBJECTIVES LEADING TO THE ACHIEVEMENT OF PROGRAM OUTCOMES**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Course Objectives** | **Program Outcomes** | | | | | | | | | | | | **Program Specific Outcomes** | | |
| **PO1** | **PO2** | **PO3** | **PO4** | **PO5** | **PO6** | **PO7** | **PO8** | **PO9** | **PO 10** | **PO 11** | **PO 12** | **PSO1** | **PSO2** | **PSO3** |
| **4** | **3** | **2** | **2** | **-** | **1** | **-** | **-** | **1** | **2** | **-** | **2** | **2** | **2** | **2** | **-** |
| **5** | **3** | **2** | **3** | **-** | **1** | **-** | **-** | **1** | **2** | **-** | **-** | **2** | **3** | **3** | **-** |

**7. SOFTWARE AND HARDWARE REQUIREMENTS:**

**HARDWARE:**

Desktop Computer Systems

**SOFTWARE:**

Application Software: C Programming Compiler

**INDEX**

|  |  |  |
| --- | --- | --- |
| **Sl. No** | **Assignment** | **Page No** |
| **1** | **Write a program non-recursive and recursive program to calculate Fibonacci numbers and analyze their time and space complexity** | **1-9** |
| **2** | **Write a program to implement Huffman Encoding using a greedy strategy** | **10-15** |
| **3** | **Program to implement fractional knapsack problem using greedy method** | **16-17** |
| **4** | **Implement 0/1 Knapsack problem using Dynamic Programming.** | **18-19** |
| **5** | **Implement N Queen's problem using Back Tracking.** | **19-20** |
| **6** | **Mini Project - Implement the Naive string matching algorithm and Rabin-Karp algorithm for string matching. Observe difference in working of both the algorithms for the same input** | **20-29** |

**Assignmentt-1**

# TITLE: Write a program non-recursive and recursive program to calculate Fibonacci numbers and analyze their time and space complexity

# OBJECTIVE:

Design a non-recursive and recursive program to calculate Fibonacci numbers and analyze their time

and space complexity

# RESOURCES:

Dev C++

# PROGRAM LOGIC:

The Fibonacci series can be described using the mathematical equation F(n)=F(n-1)+F(n-2)F(n)=F(n−1)+F(n−2), with the condition F(1)=0 and F(2)=1. The sum of the previous two numbers in the sequence is the next number of the sequence. The Fibonacci series can be implemented in C **using recursion or without using recursion**.

# PROCEDURE:

* + 1. Create: Open Dev C++, write a program after that save the program with .c extension.
    2. Compile: Alt + F9
    3. Execute: Ctrl + F10

# SOURCE CODE:

## 1. Using Recursion

include<stdio.h>

int fib(int n) {

if (n == 1)

return 0; //First digit in the series is 0

else if (n == 2)

return 1; //Second digit in the series is 1

else

return (fib(n - 1) + fib(n - 2)); //Sum of previous two numbers in the series gives the next number in the series

}

int main() {

int n = 5;

int i;

printf("The fibonacci series is :\n");

for (i = 1; i <= n; i++) {

printf("%d ", fib(i));

}

}

**Explanation:**

In the above code, we created a function named fib(), which has an integer as the parameter and return value data type. In the main function, we iterate over a loop from 1 to n times, and at each iteration called the fib() function. In fib() function, if the parameter passed is 0 or 1, we will return the same value otherwise, we will return the sum of recursive calls with parameter values one and two less than our current parameter, which is **fib(i-1)+fib(i-2)**.

## Time and Space Complexity of Recursive Method

* The time complexity of the above code is **T(2^N), i.e., exponential**.
* The Space complexity of the above code is **O(N) for a recursive series**.

## **2. without Recursion**

#include<stdio.h>

int fib(int n) {

int arr[5];

int i;

arr[0] = 0; // First term is zero

arr[1] = 1; // Second term is one

for (i = 2; i <= n; i++) {

arr[i] = arr[i - 1] + arr[i - 2]; //Calculating the sum of previous two fibonacci numbers

}

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

printf("%d ", arr[i]);

}

}

int main() {

int n = 5;

printf("The Fibonacci series is : \n");

fib(n);

return 0;

}

**Explanation** In the above code, In the main() function, We called the fib() function with n as a parameter. The value of n is initialized to 5.

In the fib() function, We create an array of size n, in our case an array of size 5 to hold the Fibonacci numbers. The first and second element in the array is initialized to 0 and 1, respectively. Later we used for loop to find the other elements of the array i.e the Fibonacci numbers using the formula **arr[i] = arr[i-1] + arr[i-2]**. Later we will print all the elements in the array.

## Time Complexity and Space Complexity of Dynamic Programming

The time complexity of the above code is T(N), i.e., linear. We have to find the sum of two terms, and it is repeated n times depending on the value of n.

The space complexity of the above code is O(N).

**INPUT/ OUTPUT:**

## 1. Using Recursion

# The fibonacci series is :

# 0 1 1 2 3

## 2. without using Recursion

# Output

# The Fibonacci series is :

# 0 1 1 2 3

# Conclusion: Students successfully executed non-recursive and recursive program and also calculated and analyzed time and space complexity

**Assignment -2**

**TITLE:** Write a program to implement Huffman Encoding using a greedy strategy

# OBJECTIVE:

Design a program to implement Huffman Encoding using a greedy strategy

# RESOURCES:

Dev C++

# PROGRAM LOGIC:

Huffman coding is a lossless data compression algorithm. The idea is to assign variable-length codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding characters. The most frequent character gets the smallest code and the least frequent character gets the largest code Huffman Coding is a technique of compressing data to reduce its size without losing any of the details

# PROCEDURE:

* + - Create: Open Dev C++, write a program after that save the program with .c extension.
    - Compile: Alt + F9
    - Execute: Ctrl + F10

# SOURCE CODE:

#include <stdio.h>

#include <stdlib.h>

// node

/\*

\_\_\_\_\_

| data|

| freq|

|\_\_\_\_\_|

left ch./ \right chlid address

\_\_\_\_\_/ \ \_\_\_\_\_

| data| | data|

| freq| | freq|

|\_\_\_\_\_| |\_\_\_\_\_|

\*/

typedef struct node

{

int frequency;

char data;

struct node \*left;

struct node \*right;

}node;

int heap\_array\_size = 100; // size of array storing heap

int heap\_size = 0;

const int INF = 100000;

//function to swap nodes

void swap( node \*a, node \*b ) {

node t;

t = \*a;

\*a = \*b;

\*b = t;

}

/\*

function to print tree

https://www.codesdope.com/blog/article/binary-tree-in-c-linked-representation-traversals/

\*/

void inorder(struct node \*root)

{

if(root!=NULL) // checking if the root is not null

{

inorder(root->left); // visiting left child

printf(" %d ", root->frequency); // printing data at root

inorder(root->right);// visiting right child

}

}

/\*

function for new node

\*/

node\* new\_node(char data, int freq) {

node \*p;

p = malloc(sizeof(struct node));

p->data = data;

p->frequency = freq;

p->left = NULL;

p->right = NULL;

return p;

}

//function to get right child of a node of a tree

int get\_right\_child(int index) {

if((((2\*index)+1) <= heap\_size) && (index >= 1))

return (2\*index)+1;

return -1;

}

//function to get left child of a node of a tree

int get\_left\_child(int index) {

if(((2\*index) <= heap\_size) && (index >= 1))

return 2\*index;

return -1;

}

//function to get the parent of a node of a tree

int get\_parent(int index) {

if ((index > 1) && (index <= heap\_size)) {

return index/2;

}

return -1;

}

/\* Functions taken from minimum priority queue

https://www.codesdope.com/blog/article/priority-queue-using-heap/

https://www.codesdope.com/blog/article/heap-binary-heap/\*/

void insert(node A[], node\* a, int key) {

heap\_size++;

A[heap\_size] = \*a;

int index = heap\_size;

while((index>1) && (A[get\_parent(index)].frequency > a->frequency)) {

swap(&A[index], &A[get\_parent(index)]);

index = get\_parent(index);

}

}

node\* build\_queue(node c[], int size) {

node\* a = malloc(sizeof(node)\*heap\_array\_size); // a is the array to store heap

int i;

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

insert(a, &c[i], c[i].frequency); // inserting node in array a(min-queue)

}

return a;

}

void min\_heapify(node A[], int index) {

int left\_child\_index = get\_left\_child(index);

int right\_child\_index = get\_right\_child(index);

// finding smallest among index, left child and right child

int smallest = index;

if ((left\_child\_index <= heap\_size) && (left\_child\_index>0)) {

if (A[left\_child\_index].frequency < A[smallest].frequency) {

smallest = left\_child\_index;

}

}

if ((right\_child\_index <= heap\_size && (right\_child\_index>0))) {

if (A[right\_child\_index].frequency < A[smallest].frequency) {

smallest = right\_child\_index;

}

}

// smallest is not the node, node is not a heap

if (smallest != index) {

swap(&A[index], &A[smallest]);

min\_heapify(A, smallest);

}

}

node\* extract\_min(node A[]) {

node minm = A[1];

A[1] = A[heap\_size];

heap\_size--;

min\_heapify(A, 1);

node \*z;

// copying minimum element

z = malloc(sizeof(struct node));

z->data = minm.data;

z->frequency = minm.frequency;

z->left = minm.left;

z->right = minm.right;

return z; //returning minimum element

}

// Huffman code

node\* greedy\_huffman\_code(node C[]) {

node \*min\_queue = build\_queue(C, 6); // making min-queue

while(heap\_size > 1) {

node \*h = extract\_min(min\_queue);

node \*i = extract\_min(min\_queue);

node \*z;

z = malloc(sizeof(node));

z->data = '\0';

z->left = h;

z->right = i;

z->frequency = z->left->frequency + z->right->frequency;

insert(min\_queue, z, z->frequency);

}

return extract\_min(min\_queue);

}

int main() {

node \*a = new\_node('a', 42);

node \*b = new\_node('b', 20);

node \*c = new\_node('c', 5);

node \*d = new\_node('d', 10);

node \*e = new\_node('e', 11);

node \*f = new\_node('f', 12);

node C[] = {\*a, \*b, \*c, \*d, \*e , \*f};

node\* z = greedy\_huffman\_code(C);

inorder(z); //printing tree

printf("\n");

return 0;

}

**Output:**

42 100 11 23 12 58 5 15 10 35 20

...Program finished with exit code 0

Press ENTER to exit console.

**Assignment 3**

## **TITLE**: Program to implement fractional knapsack problem using greedy method

**OBJECTIVE**: Develop and implement fractional knapsack problem using greedy method

**THEORY:**

Given the weights and values of**N** items, in the form of **{value, weight}** put these items in a knapsack of capacity **W** to get the maximum total value in the knapsack. In **Fractional Knapsack**, we can break items for maximizing the total value of the knapsack

**Input:**arr[] = {{60, 10}, {100, 20}, {120, 30}}, W = 50  
**Output:** 240   
**Explanation:** By taking items of weight 10 and 20 kg and 2/3 fraction of 30 kg.   
Hence total price will be 60+100+(2/3)(120) = 240

**Input:** arr[] = {{500, 30}}, W = 10  
**Output:**166.667

The basic idea of the greedy approach is to calculate the ratio **value/weight** for each item and sort the item on the basis of this ratio. Then take the item with the highest ratio and add them until we can’t add the next item as a whole and at the end add the next item as much as we can. Which will always be the optimal solution to this problem.

Follow the given steps to solve the problem using the above approach:

* + Calculate the ratio(value/weight) for each item.
  + Sort all the items in decreasing order of the ratio.
  + Initialize res =0, curr\_cap = given\_cap.
  + Do the following for every item “i” in the sorted order:
  + If the weight of the current item is less than or equal to the remaining capacity then add the value of that item into the result
  + Else add the current item as much as we can and break out of the loop.
  + Return res.

# include<stdio.h>

void knapsack(int n, float weight[], float profit[], float capacity) {

   float x[20], tp = 0;

   int i, j, u;

   u = capacity;

    for (i = 0; i < n; i++)

      x[i] = 0.0;

    for (i = 0; i < n; i++) {

      if (weight[i] > u)

         break;

      else {

         x[i] = 1.0;

         tp = tp + profit[i];

         u = u - weight[i];

      }

   }

   if (i < n)

      x[i] = u / weight[i];

    tp = tp + (x[i] \* profit[i]);

    printf("\nThe result vector is:- ");

   for (i = 0; i < n; i++)

      printf("%f\t", x[i]);

   printf("\nMaximum profit is:- %f", tp);

 }

int main() {

   float weight[20], profit[20], capacity;

   int num, i, j;

   float ratio[20], temp;

    printf("\nEnter the no. of objects:- ");

   scanf("%d", &num);

    printf("\nEnter the wts and profits of each object:- ");

   for (i = 0; i < num; i++) {

      scanf("%f %f", &weight[i], &profit[i]);

   }

    printf("\nEnter the capacityacity of knapsack:- ");

   scanf("%f", &capacity);

    for (i = 0; i < num; i++) {

      ratio[i] = profit[i] / weight[i];

   }

    for (i = 0; i < num; i++) {

      for (j = i + 1; j < num; j++) {

         if (ratio[i] < ratio[j]) {

            temp = ratio[j];

            ratio[j] = ratio[i];

            ratio[i] = temp;

             temp = weight[j];

            weight[j] = weight[i];

            weight[i] = temp;

             temp = profit[j];

            profit[j] = profit[i];

            profit[i] = temp;

         }

      }

   }

    knapsack(num, weight, profit, capacity);

   return(0);

}

**Output :**

Enter the no. of objects:- 7

 Enter the wts and profits of each object:-

2 10

3 5

5 15

7 7

1 6

4 18

1 3

 Enter the capacity of knapsack:- 15

 The result vector is:- 1.000000        1.000000        1.000000        1.000000

    1.000000        0.666667        0.000000

 Maximum profit is:- 55.333332

**Assignment 4**

**Implement 0/1 Knapsack problem using Dynamic Programming.**

#include<stdio.h>

#include<conio.h>

int w[10],p[10],v[10][10],n,i,j,cap,x[10]={0};

int max(int i,int j)

{

return ((i>j)?i:j);

}

int knap(int i,int j)

{

int value;

if(v[i][j]<0)

{

if(j<w[i])

value=knap(i-1,j);

else

value=max(knap(i-1,j),p[i]+knap(i-1,j-w[i]));

v[i][j]=value;

}

return(v[i][j]);

}

void main()

{

int profit,count=0;

clrscr();

printf("\nEnter the number of elements\n");

scanf("%d",&n);

printf("Enter the profit and weights of the elements\n");

for(i=1;i<=n;i++)

{

printf("For item no %d\n",i);

scanf("%d%d",&p[i],&w[i]);

}

printf("\nEnter the capacity \n");

scanf("%d",&cap);

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

for(j=0;j<=cap;j++)

if((i==0)||(j==0))

v[i][j]=0;

else

v[i][j]=-1;

profit=knap(n,cap);

i=n;

j=cap;

while(j!=0&&i!=0)

{

if(v[i][j]!=v[i-1][j])

{

x[i]=1;

j=j-w[i];

i--;

}

else

i--;

}

printf("Items included are\n");

printf("Sl.no\tweight\tprofit\n");

for(i=1;i<=n;i++)

if(x[i])

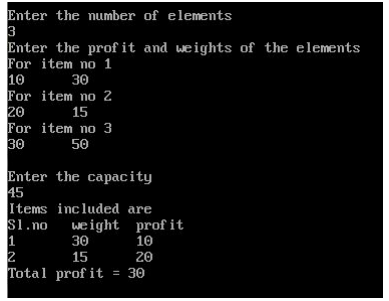
printf("%d\t%d\t%d\n",++count,w[i],p[i]);

printf("Total profit = %d\n",profit);

getch();

}

**Output**



**Assignment 5**

**Implement N Queen's problem using Back Tracking.**

#include<stdio.h>

#include<conio.h>

#include<math.h>

int a[30],count=0;

int place(int pos)

{

int i;

for(i=1;i<pos;i++)

{

if((a[i]==a[pos])||((abs(a[i]-a[pos])==abs(i-pos))))

return 0;

}

return 1;

}

void print\_sol(int n)

{

int i,j;

count++;

printf("\n\nSolution #%d:\n",count);

for(i=1;i<=n;i++)

{

for(j=1;j<=n;j++)

{

if(a[i]==j)

printf("Q\t");

else

printf("\*\t");

}

printf("\n");

}

}

void queen(int n)

{

int k=1;

a[k]=0;

while(k!=0)

{

a[k]=a[k]+1;

while((a[k]<=n)&&!place(k))

a[k]++;

if(a[k]<=n)

{

if(k==n)

print\_sol(n);

else

{

k++;

a[k]=0;

}

}

else

k--;

}

}

void main()

{

int i,n;

clrscr();

printf("Enter the number of Queens\n");

scanf("%d",&n);

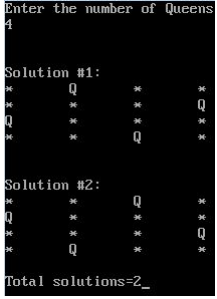
queen(n);

printf("\nTotal solutions=%d",count);

getch();

}

**Output:**



**Mini Project - Implement the Naive string matching algorithm and Rabin-Karp algorithm for string matching. Observe difference in working of both the algorithms for the same input.**

**String Matching**

Pattern matching is the process of checking a perceived sequence of string for the presence of the constituents of some pattern. In contrast to pattern recognition, the match usually has to be exact. The patterns generally have the form sequences of pattern matching include outputting the locations of a pattern within a string sequence, to output some component of the matched pattern, and to substitute the matching pattern with some other string sequence (i.e., search and replace). Pattern matching concept is used in many applications Following figure shows the different applications.

**Algorithms used for pattern searching**

Naive Pattern Searching

Rabin Karp String Search Algorithm

Knuth–Morris–Pratt algorithm

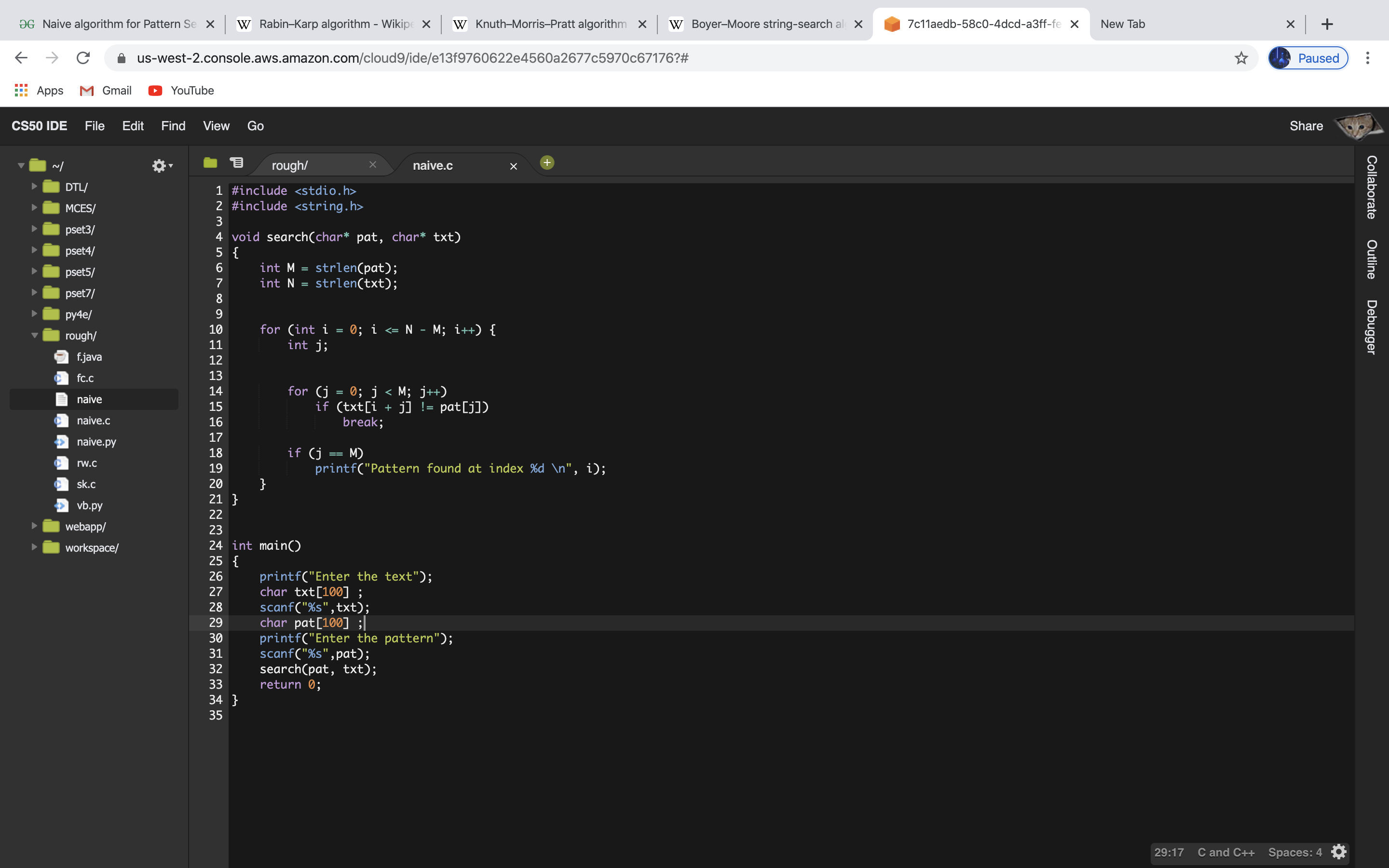
Boyer–Moore string search algorithm

##### **Naive Pattern Searching**

A naive string matching algorithm compares the given pattern against all positions in the given text. Each comparison takes time proportional to the length of the pattern, and the number of positions is proportional to the length of the text. Therefore, the worst-case time for such a method is proportional to the product of the two lengths. In many practical cases, this time can be significantly reduced by cutting short the comparison at each position as soon as a mismatch is found, but this idea cannot guarantee any speedup.

**Technique** : Each character of the pattern is compared to a substring of the text which is the length of the pattern, until there is a mismatch or a match.

##### **Code / Algorithm**

****

**Time Complexity**

The **best case** occurs when the first character of the pattern is not present in text at all.

txt[] = "AABCCAADDEE";

pat[] = “FAA”;

The number of comparisons in best case is O(n).

The **worst case** of Naive Pattern Searching occurs in when all characters of the text and pattern are same or hen only the last character is different.

txt[] = "AAAAAAAAAAAAAAAAAA";

pat[] = “AAAAA";

txt[] = "AAAAAAAAAAAAAAAAAB";

pat[] = "AAAAB";

The number of comparisons in the worst case is O(m\*(n-m+1)). Although strings which have repeated characters are not likely to appear in English text, they may well occur in other applications (for example, in binary texts).

**Space Complexity**

This is an in-place algorithm. So O(1) auxiliary space is needed.

##### **Rabin–Karp string search algorithm**

Rabin-Karp algorithm slides the pattern one by one. But unlike the Naive algorithm, Rabin Karp algorithm matches the hash value of the pattern with the hash value of current substring of text, and if the hash values match then only it starts matching individual characters.

**Technique** : Hashing! So Rabin Karp algorithm needs to calculate hash values for following strings.

1. Pattern itself.
2. All the substrings of text of length m.

Since we need to efficiently calculate hash values for all the substrings of size m of text, we must have a hash function which has following property.

Hash at the next shift must be efficiently computable from the current hash value and next character in text or we can say hash(txt[s+1 .. s+m]) must be efficiently computable from hash(txt[s .. s+m-1]) and txt[s+m] i.e., hash(txt[s+1 .. s+m])= rehash(txt[s+m], hash(txt[s .. s+m-1])) and rehash must be O(1) operation.

To do rehashing, we need to take off the most significant digit and add the new least significant digit for in hash value.

Rehashing is done using the following formula.

hash( txt[s+1 .. s+m] ) = ( d ( hash( txt[s .. s+m-1]) – txt[s]\*h ) + txt[s + m] ) mod q

hash( txt[s .. s+m-1] ) : Hash value at shift s.

hash( txt[s+1 .. s+m] ) : Hash value at next shift (or shift s+1)

d: Number of characters in the alphabet

q: A prime number

h: d^(m-1)

##### **Code / Algorithm**

**Time Complexity**

The **average** case running time of the Rabin-Karp algorithm is O(n+m),

but its **worst-case** time is O((n-m+1) m).

**Best** case is O(m).

One of the Worst case of Rabin-Karp algorithm O(mn) occurs when all characters of pattern and text are same as the hash values of all the substrings of txt[] match with hash value of pat[].

For example: pat[] = “AAA” and txt[] = “AAAAAAA”.

**Space Complexity**

Space Complexity of Rabin-Karp algorithm is O(m)

**Comparison Tables for Algorithms**

*Different Techniques used by Different Algorithms*

| **Algorithm** | **Techniques** |
| --- | --- |
| Naive string search algorithm | Each character of the pattern is compared to a substring of the text which is the length of the pattern, until there is a mismatch or a match. |
| Rabin–Karp string search algorithm | Hashing |

##### *Comparisons of Worst case Complexity of different algorithms*

| **Algorithm** | **Preprocessing**  **time** | | **Matching time/Searching Phase/Running Time** |
| --- | --- | --- | --- |
|  | **Time complexity** | **Space complexity** |  |
| Naive string search algorithm | O(n) | O(1) | O((n-m+1) m) |
| Rabin–Karp string search algorithm | O(m) | O(m) | Average O(n+m), worst O((n-m+1) m) |
| Knuth–Morris–Pratt algorithm | O(m) | O(m) | O(m+n) |
| Boyer–Moore string search algorithm | O(m + |Σ|) | O(m + |Σ|) | Ω(n/m), O(n) |