**Design & Analysis of Algorithm**

**Lab Assignment No: 01**

Rushikesh Bhivsen Shinde

BBCO19134

**Problem Statement:**

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

**Code:**

//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");

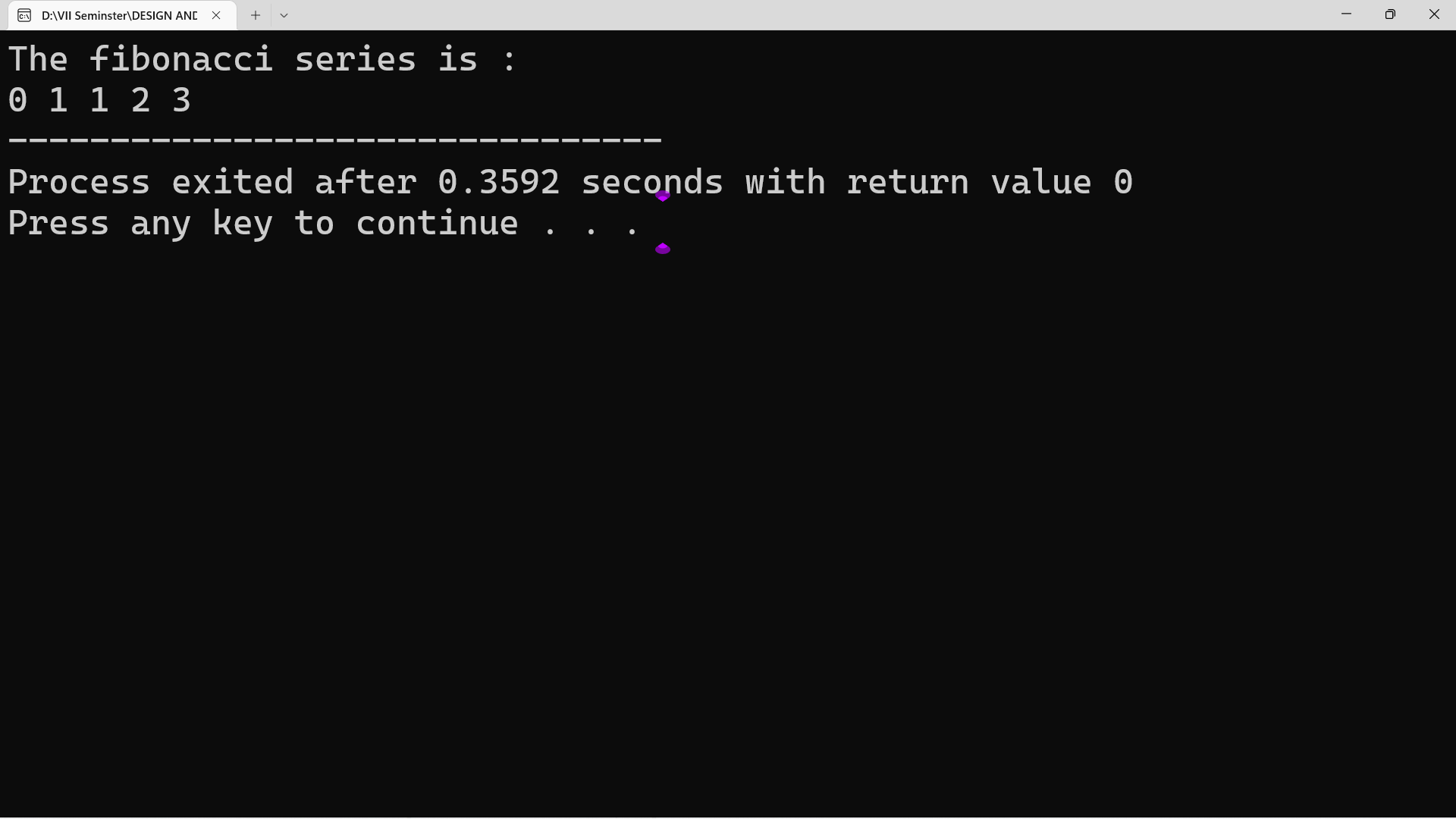
for (i = 1; i <= n; i++) {

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

}

}

**Output:**

****

**//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() {

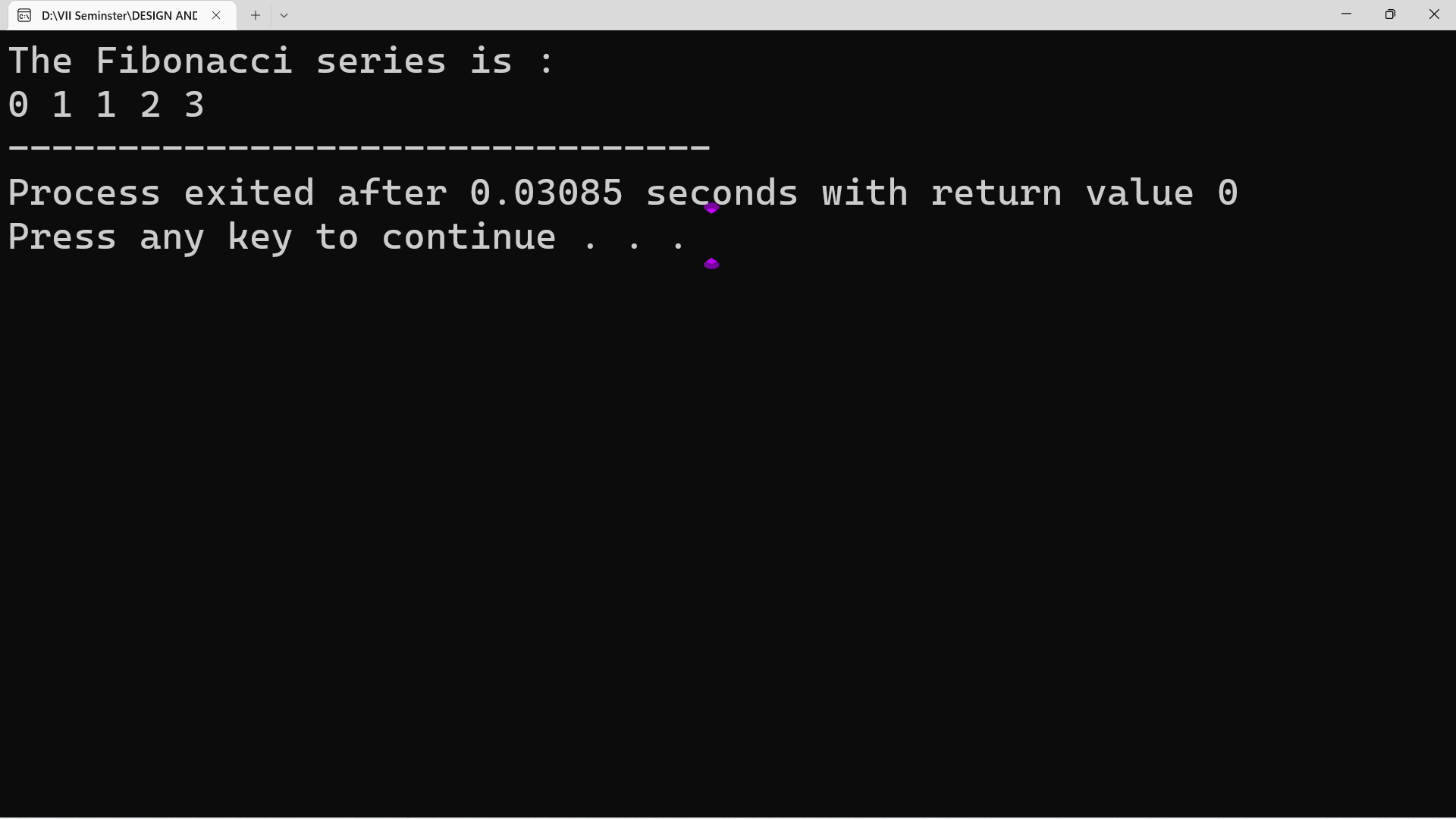
int n = 5;

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

fib(n);

return 0;

}

**Output: **

**Design & Analysis of Algorithm**

**Lab Assignment No: 02**

Rushikesh Bhivsen Shinde

BBCO19134

**Problem Statement:**

Write a program to implement Huffman Encoding using a greedy strategy.

**Code:**

// C++ program for Huffman Coding

#include <iostream>

#include <cstdlib>

using namespace std;

// This constant can be avoided by explicitly

// calculating height of Huffman Tree

#define MAX\_TREE\_HT 100

// A Huffman tree node

struct MinHeapNode {

// One of the input characters

char data;

// Frequency of the character

unsigned freq;

// Left and right child of this node

struct MinHeapNode \*left, \*right;

};

// A Min Heap: Collection of

// min-heap (or Huffman tree) nodes

struct MinHeap {

// Current size of min heap

unsigned size;

// capacity of min heap

unsigned capacity;

// Array of minheap node pointers

struct MinHeapNode\*\* array;

};

// A utility function allocate a new

// min heap node with given character

// and frequency of the character

struct MinHeapNode\* newNode(char data, unsigned freq)

{

struct MinHeapNode\* temp

= (struct MinHeapNode\*)malloc

(sizeof(struct MinHeapNode));

temp->left = temp->right = NULL;

temp->data = data;

temp->freq = freq;

return temp;

}

// A utility function to create

// a min heap of given capacity

struct MinHeap\* createMinHeap(unsigned capacity)

{

struct MinHeap\* minHeap

= (struct MinHeap\*)malloc(sizeof(struct MinHeap));

// current size is 0

minHeap->size = 0;

minHeap->capacity = capacity;

minHeap->array

= (struct MinHeapNode\*\*)malloc(minHeap->

capacity \* sizeof(struct MinHeapNode\*));

return minHeap;

}

// A utility function to

// swap two min heap nodes

void swapMinHeapNode(struct MinHeapNode\*\* a,

struct MinHeapNode\*\* b)

{

struct MinHeapNode\* t = \*a;

\*a = \*b;

\*b = t;

}

// The standard minHeapify function.

void minHeapify(struct MinHeap\* minHeap, int idx)

{

int smallest = idx;

int left = 2 \* idx + 1;

int right = 2 \* idx + 2;

if (left < minHeap->size && minHeap->array[left]->

freq < minHeap->array[smallest]->freq)

smallest = left;

if (right < minHeap->size && minHeap->array[right]->

freq < minHeap->array[smallest]->freq)

smallest = right;

if (smallest != idx) {

swapMinHeapNode(&minHeap->array[smallest],

&minHeap->array[idx]);

minHeapify(minHeap, smallest);

}

}

// A utility function to check

// if size of heap is 1 or not

int isSizeOne(struct MinHeap\* minHeap)

{

return (minHeap->size == 1);

}

// A standard function to extract

// minimum value node from heap

struct MinHeapNode\* extractMin(struct MinHeap\* minHeap)

{

struct MinHeapNode\* temp = minHeap->array[0];

minHeap->array[0]

= minHeap->array[minHeap->size - 1];

--minHeap->size;

minHeapify(minHeap, 0);

return temp;

}

// A utility function to insert

// a new node to Min Heap

void insertMinHeap(struct MinHeap\* minHeap,

struct MinHeapNode\* minHeapNode)

{

++minHeap->size;

int i = minHeap->size - 1;

while (i && minHeapNode->freq < minHeap->array[(i - 1) / 2]->freq) {

minHeap->array[i] = minHeap->array[(i - 1) / 2];

i = (i - 1) / 2;

}

minHeap->array[i] = minHeapNode;

}

// A standard function to build min heap

void buildMinHeap(struct MinHeap\* minHeap)

{

int n = minHeap->size - 1;

int i;

for (i = (n - 1) / 2; i >= 0; --i)

minHeapify(minHeap, i);

}

// A utility function to print an array of size n

void printArr(int arr[], int n)

{

int i;

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

cout<< arr[i];

cout<<"\n";

}

// Utility function to check if this node is leaf

int isLeaf(struct MinHeapNode\* root)

{

return !(root->left) && !(root->right);

}

// Creates a min heap of capacity

// equal to size and inserts all character of

// data[] in min heap. Initially size of

// min heap is equal to capacity

struct MinHeap\* createAndBuildMinHeap(char data[], int freq[], int size)

{

struct MinHeap\* minHeap = createMinHeap(size);

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

minHeap->array[i] = newNode(data[i], freq[i]);

minHeap->size = size;

buildMinHeap(minHeap);

return minHeap;

}

// The main function that builds Huffman tree

struct MinHeapNode\* buildHuffmanTree(char data[], int freq[], int size)

{

struct MinHeapNode \*left, \*right, \*top;

// Step 1: Create a min heap of capacity

// equal to size. Initially, there are

// modes equal to size.

struct MinHeap\* minHeap = createAndBuildMinHeap(data, freq, size);

// Iterate while size of heap doesn't become 1

while (!isSizeOne(minHeap)) {

// Step 2: Extract the two minimum

// freq items from min heap

left = extractMin(minHeap);

right = extractMin(minHeap);

// Step 3: Create a new internal

// node with frequency equal to the

// sum of the two nodes frequencies.

// Make the two extracted node as

// left and right children of this new node.

// Add this node to the min heap

// '$' is a special value for internal nodes, not used

top = newNode('$', left->freq + right->freq);

top->left = left;

top->right = right;

insertMinHeap(minHeap, top);

}

// Step 4: The remaining node is the

// root node and the tree is complete.

return extractMin(minHeap);

}

// Prints huffman codes from the root of Huffman Tree.

// It uses arr[] to store codes

void printCodes(struct MinHeapNode\* root, int arr[], int top)

{

// Assign 0 to left edge and recur

if (root->left) {

arr[top] = 0;

printCodes(root->left, arr, top + 1);

}

// Assign 1 to right edge and recur

if (root->right) {

arr[top] = 1;

printCodes(root->right, arr, top + 1);

}

// If this is a leaf node, then

// it contains one of the input

// characters, print the character

// and its code from arr[]

if (isLeaf(root)) {

cout<< root->data <<": ";

printArr(arr, top);

}

}

// The main function that builds a

// Huffman Tree and print codes by traversing

// the built Huffman Tree

void HuffmanCodes(char data[], int freq[], int size)

{

// Construct Huffman Tree

struct MinHeapNode\* root

= buildHuffmanTree(data, freq, size);

// Print Huffman codes using

// the Huffman tree built above

int arr[MAX\_TREE\_HT], top = 0;

printCodes(root, arr, top);

}

// Driver code

int main()

{

char arr[] = { 'a', 'b', 'c', 'd', 'e', 'f' };

int freq[] = { 5, 9, 12, 13, 16, 45 };

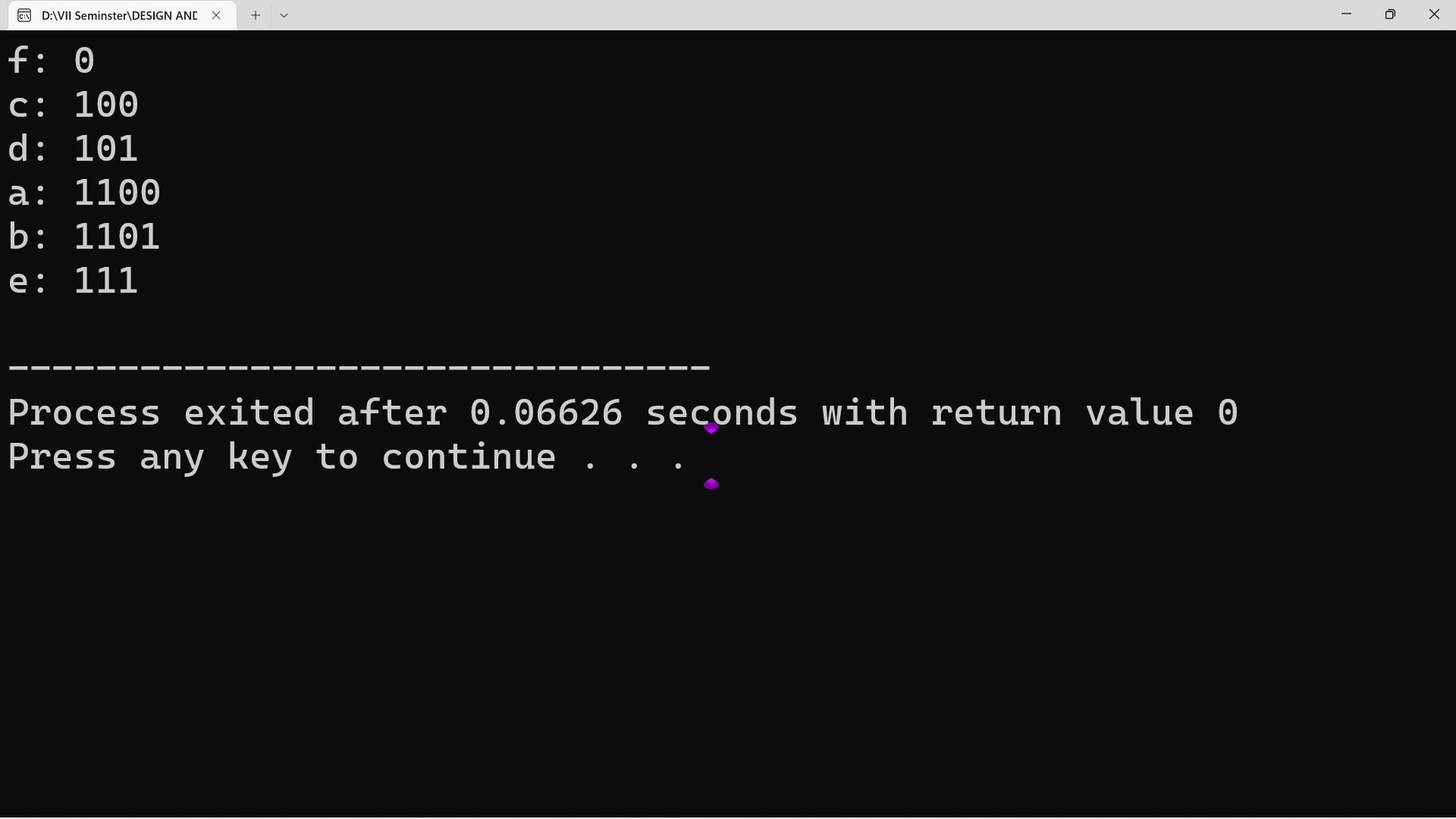
int size = sizeof(arr) / sizeof(arr[0]);

HuffmanCodes(arr, freq, size);

return 0;

}

**Output:**

****

**Design & Analysis of Algorithm**

**Lab Assignment No: 03**

Rushikesh Bhivsen Shinde

BBCO19134

**Problem Statement:**

Write a program to solve a fractional Knapsack problem using a greedy method.

**Code:**

# 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;

}

}

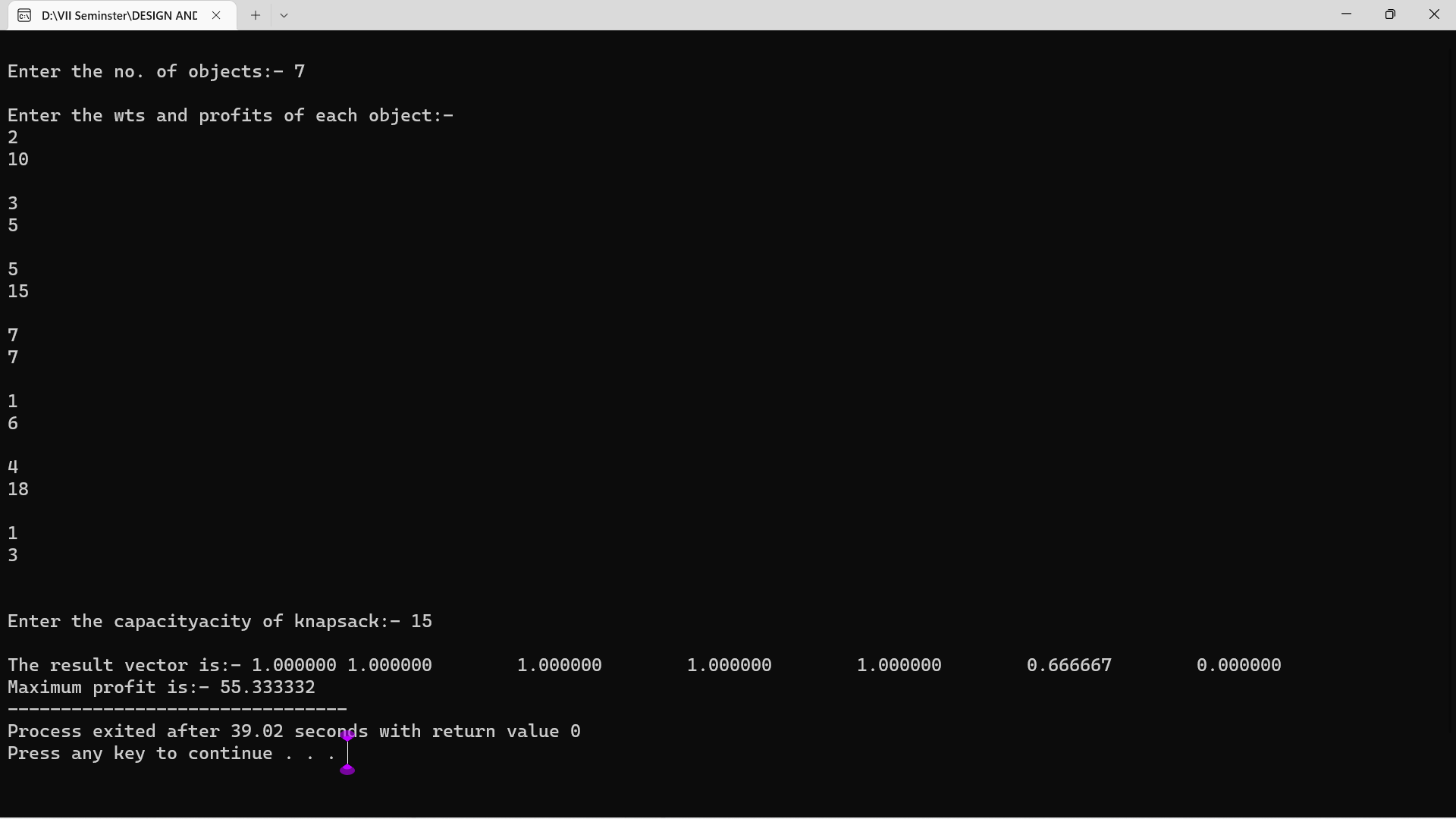
}

knapsack(num, weight, profit, capacity);

return(0);

}

**Output :**



**Design & Analysis of Algorithm**

**Lab Assignment No: 04**

Rushikesh Bhivsen Shinde

BBCO19134

**Problem Statement:**

Write a program to solve a 0-1 Knapsack problem using dynamic programming or branch and bound strategy.

**Code:**

#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]);

}

int main()

{

int profit,count=0;

//clrscr();

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

scanf("%d",&n);

printf("Enter the profit and weights of the elementsn");

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

{

printf("\nFor 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 aren");

printf("Sl.notweighttprofit\n");

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

if(x[i])

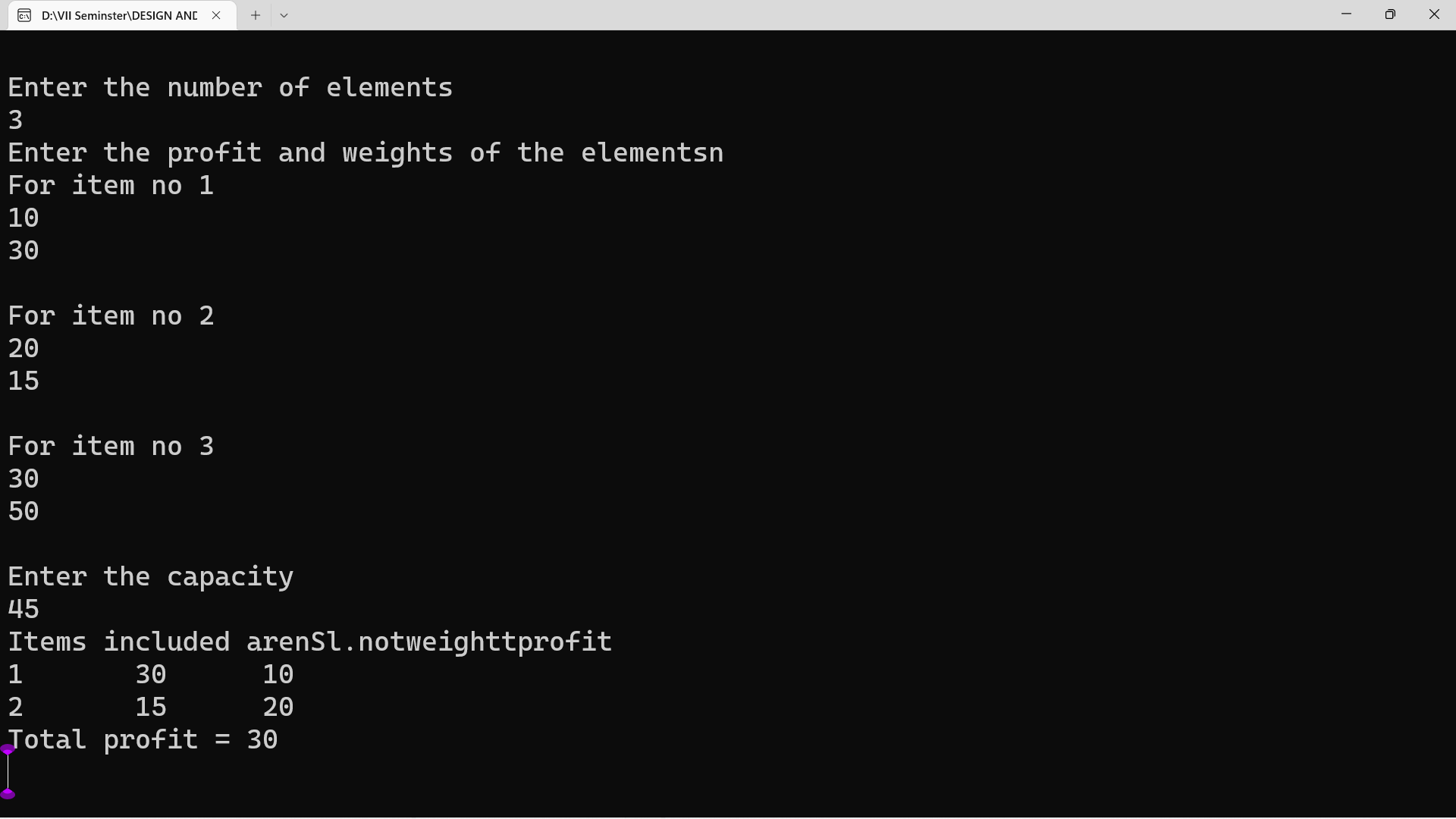
printf("%d\t%d\t%d\n",++count,w[i],p[i]);

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

getch();

}

**Output:**



**Design & Analysis of Algorithm**

**Lab Assignment No: 05**

Rushikesh Bhivsen Shinde

BBCO19134

**Problem Statement:**

Design n-Queens matrix having first Queen placed. Use backtracking to place remaining Queens to generate the final n-queen’s matrix.

**Code:**

#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--;

}

}

int main() {

int i,n;

//clrscr();

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

scanf("%d",&n);

queen(n);

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

getch();

}

**Output:**

