UCS415 – Design and Analysis of Algorithms Lab Assignment 2

**Name: Akshita Pathak**

**Roll No: 102203796**

**Subgroup: 2CO18**

**Write a program to solve the following problems using greedy Greedy approach:**

* **Activity selection**

/\*

1. Activity Selection problem

\*/

#include<iostream>

using namespace std;

void activity\_selector(int s[],int f[],int n){

int k=0;

cout<<"Activities selection order: \n "<<k<<" ";

//traverse from second activity to nth one (index from 0 so start from 1)

for(int m=1;m<n;m++){

//check if start time of new job>=finish time of already added job

if(s[m]>=f[k]){

//add it to soln

cout<<m<<" ";

//set k to new inserted

k=m;

}

}

}

int main()

{

int s[]={0,1,3,4,5};

int f[]={2,3,4,6,7};

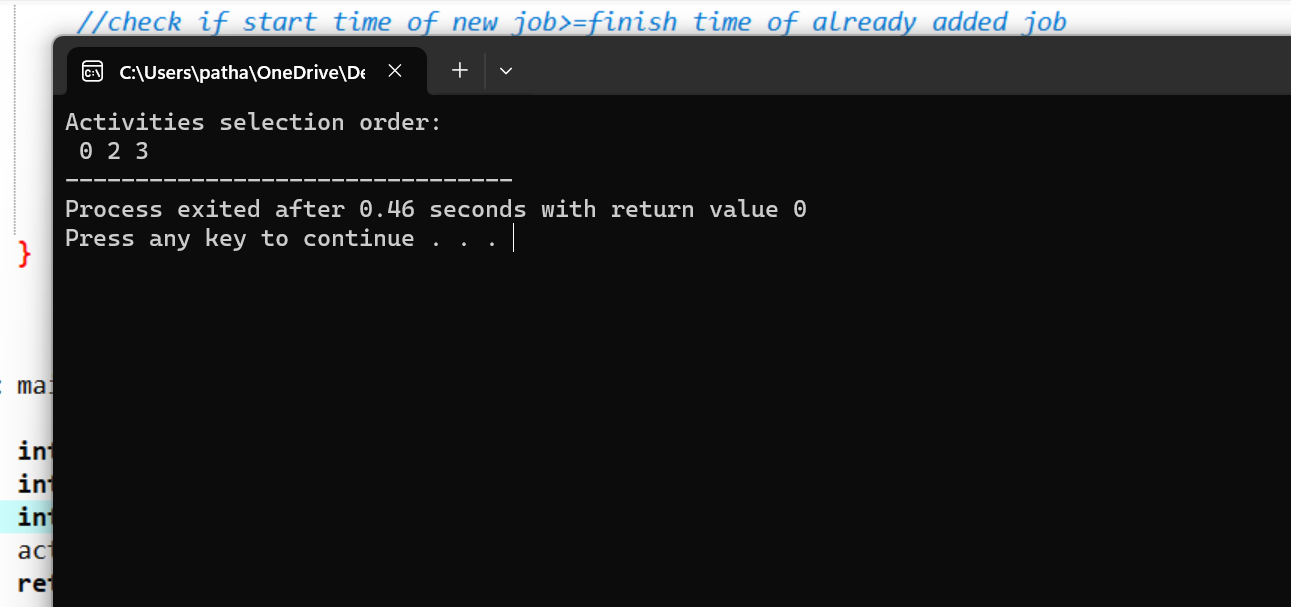
int n=sizeof(s)/sizeof(s[0]); //store number of activities in n

activity\_selector(s,f,n);

return 0;

}

**OUTPUT**:



* **Job sequencing**

/\*

2. Job sequencing with deadlines

\*/

#include<iostream>

using namespace std;

int job\_seq(int d[],int j[],int n){

//initialise first elements of d,j array

d[0]=j[0]=0;

//include 1st job of maxm profit

j[1]=1;

//k represents no of jobs considered

int k=1;

//loop from second to last job

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

//search for position r to insert job i

int r=k;

while(d[j[r]]>d[i] && d[j[r]]!=r){

r=r-1;

}

//check if job can be inserted

if(d[j[r]]<=d[i] && d[i]>r){

//make space for new job by shifting rest downwards

for(int q=k;q>=r+1;q--){

j[q+1]=j[q];

}

//insert

j[r+1]=i;

//increase count of k ie no of jbs considered

k=k+1;

}

}

return k; //ie no of jobs considered

}

int main()

{

int d[]={3,4,4,2,3,1,2};

int p[]={35,30,25,20,15,12,5};

int n=7;

int j[n+1]; //array to store job sequence

int ans=job\_seq(d,j,n);

cout<<"job sequence: \n";

for(int i=1;i<=ans;i++){

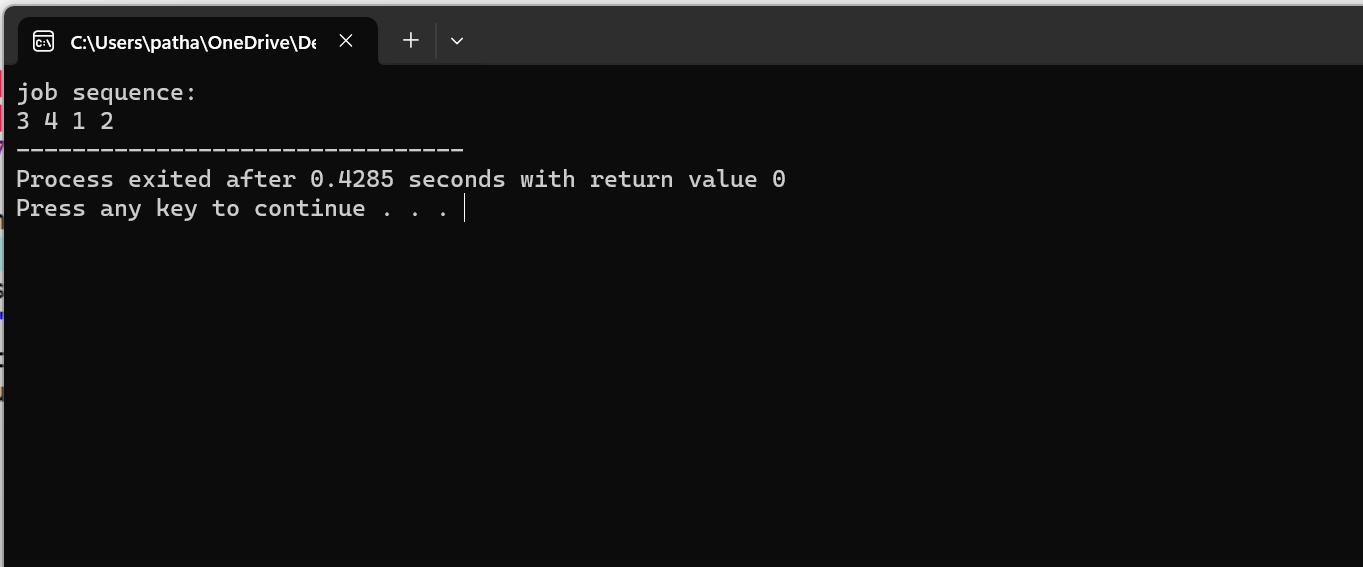
cout<<j[i]<<" ";

}

return 0;

}

**OUTPUT**:

****

* **Fractional knapsack**

/\*

3. Fractional knapsack

\*/

#include<iostream>

#include <algorithm> //to use sort fnc

using namespace std;

struct item{

int profit;

int weight;

};

bool compareItems(const item &a,const item &b) {

return (1.0\*a.profit/a.weight)>(1.0\*b.profit/b.weight);

}

double knapsack(item objects[],int n,int M) {

//to sort items in descending order of profit/weight (per-unit profit)

sort(objects,objects+n,compareItems);

double totalval = 0.0;

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

//curr item weight> knapsack remain capacity then take fractional

if(objects[i].weight>M){

//take fraction

totalval=totalval+M\*(1.0\*objects[i].profit/objects[i].weight);

M=0;

break;

}

else{

//take full item

totalval=totalval+objects[i].profit;

M=M-objects[i].weight;

}

}

return totalval;

}

int main()

{

//no of items

int n=3;

//capacity of knapsack

int M=50;

//add profit,weight

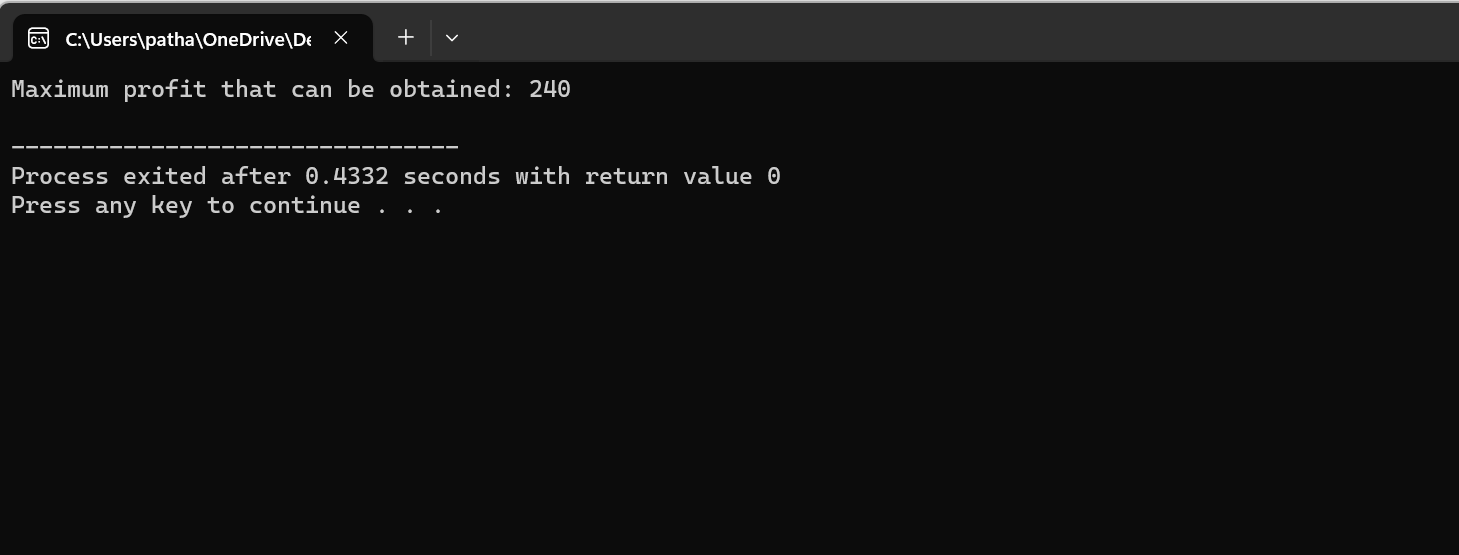
item objects[n]={{60,10},{100,20},{120,30}};

cout<<"Maximum profit that can be obtained: "<<knapsack(objects,n,M)<<endl;

return 0;

}

**OUTPUT**:



* **Huffman Coding**

/\*

4. Huffman Coding

\*/

#include <stdio.h>

#include <stdlib.h>

//maximum height of huffman tree

#define MAX\_TREE\_HT 100

//defining node of huffman tree

struct MinHeapNode{

//characters of msg

char data;

//frequncy/count of character in msg

int freq;

//left n right child

struct MinHeapNode \*left,\*right;

};

//creating a collection of huffman tree nodes

struct MinHeap{

//current size of min heap

int size;

//capacity of min heap

int capacity;

//array of min heap node pointers

struct MinHeapNode\*\* array;

};

//creating node of character n its frequency

struct MinHeapNode\* newNode(char data, int freq){

struct MinHeapNode\* temp=(struct MinHeapNode\*)malloc(sizeof(struct MinHeapNode));

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

temp->data=data;

temp->freq=freq;

return temp;

}

//to create min heap of given capcity

struct MinHeap\* createMinHeap(int 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;

}

//fnc to swap 2 min heap nodes

void swapMinHeapNode(struct MinHeapNode\*\* a, struct MinHeapNode\*\* b){

struct MinHeapNode\* t=\*a;

\*a=\*b;

\*b=t;

}

//min heapify fn

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

int smallest=idx, left=2 \* idx + 1, 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);

}

}

//to check if size of heap is 1 or not

int isSizeOne(struct MinHeap\* minHeap){

return(minHeap->size==1);

}

//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;

}

//to insert new node in 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;

}

//to build min heap

void buildMinHeap(struct MinHeap\* minHeap){

int n=minHeap->size-1;

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

minHeapify(minHeap, i);

}

//to print array of size n

void printArr(int arr[], int n){

for(int i=0; i<n; ++i)printf("%d", arr[i]);

printf("\n");

}

//to check if node is leaf node or not

int isLeaf(struct MinHeapNode\* root){

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

}

//To create a min heap of capacity=size and inserts all ip character in min heap

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;

}

//builds huffman tree

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

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

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

while(!isSizeOne(minHeap)){

left=extractMin(minHeap);

right=extractMin(minHeap);

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

top->left=left;

top->right=right;

insertMinHeap(minHeap, top);

}

return extractMin(minHeap);

}

//print codes from root to leaf

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

if(root->left){

arr[top]=0;

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

}

if(root->right){

arr[top]=1;

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

}

if(isLeaf(root)){

printf("%c: ", root->data);

printArr(arr, top);

}

}

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]);

struct MinHeapNode\* root=buildHuffmanTree(arr,freq,size);

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

printCodes(root,arrCode,top);

return 0;

}

**OUTPUT**:

