**EXPERIMENT 4**

**TITLE:** GREEDY METHOD

1. **Implement the activity-selection problem (You are given n activities with their start and finish times. Select the maximum number of activities that can be performed by a single person, assuming that a person can only work on a single activity at a time.**

**Example: Consider the following 6 activities.**

**Start [] = {1, 3, 0, 5, 8, 5}; finish [] = {2, 4, 6, 7, 9, 9};**

**The maximum set of activities that can be executed by a single person is {0, 1, 3, 4}).**

**Code:**

#include<stdio.h>

#include<time.h>

int main()

{

int n,i,j,temp1,temp2,temp3,lastindex;

clock\_t s,end;

float t;

printf("Enter the number of activities to be performed: ");

scanf("%d",&n);

int anum[n],start[n],final[n];

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

{

printf("Enter the Activity number, start time and final time for %d process\n",i+1);

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

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

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

}

s=clock();

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

{

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

{

if(final[j]<final[i])

{

temp1 = final[i];

final[i] = final[j];

final[j] = temp1;

temp2= anum[i];

anum[i]=anum[j];

anum[j]=temp2;

temp3=start[i];

start[i]=start[j];

start[j]=temp3;

}

}

}

//sorted successfully

printf("Activities sorted by final time are:\n");

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

{

printf("%d\t%d\t%d\n",anum[i],start[i],final[i]);

}

printf("%d Activity is performed\n",anum[0]);

lastindex=0;

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

{

if(start[i]>=final[lastindex])

{

printf("%d Activity is performed\n",anum[i]);

lastindex=i;

}

}

end=clock();

t=(float)end-s/CLOCKS\_PER\_SEC;

printf("Time taken for processing is: %f\n",t);

return 0;

}

1. **Consider the following scheduling problem. You are given n jobs. Job i is specified by an earliest start time si, and a processing time pi. We consider a preemptive version of the problem where a job's execution can be suspended at any time and then completed later. For example if n = 2 and the input is s1 = 2, p1 = 5 and s2 = 0, p2 = 3, then a legal preemptive schedule is one in which job 2 runs from time 0 to 2 and is then suspended. Then job 1 runs from time 2 to 7 and secondly, job 2 is completed from time 7 to 8. The goal is to output a schedule that minimizes ΣCj = 1, where Cj is the time when job j is completed and j runs from 1 to n. In the example schedule given above, C1 =7 and C2=8.**

**Code:**

#include<stdio.h>

int main()

{

int n,i,j,temp,t=0;

printf("Enter the number of processes you want: ");

scanf("%d",&n);

int p[n],s[n],pt[n],c[n];

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

{

printf("Enter details for %d process:\n",i+1);

printf("Enter the process number: ");

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

printf("Enter the Start Time: ");

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

printf("Enter the Processing Time: ");

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

}

printf("PNum Start Burst\n");

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

{

printf("%d\t%d\t%d\n",p[i],s[i],pt[i]);

}

//

///////////////

printf("Sorting on the basis of Start Time\n");

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

{

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

{

if(s[j]>s[j+1])

{

temp=s[j];

s[j]=s[j+1];

s[j+1]=temp;

//

temp=p[j];

p[j]=p[j+1];

p[j+1]=temp;

//

temp=pt[j+1];

pt[j]=pt[j+1];

pt[j+1]=temp;

//

}

}

}

///////////////

printf("PNum Start Burst\n");

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

{

printf("%d\t%d\t%d\n",p[i],s[i],pt[i]);

}

//

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

{

if((s[i]+pt[i])<=s[i+1])

{

temp=s[i]+pt[i];

t=t+temp;

c[i]=t;

}

else

{

c[i]=0;

temp=s[i+1]-s[i];

t=t+temp;

pt[i]=pt[i]-temp;

}

}

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

{

if(c[i]==0)

{

c[i]=t+pt[i];

t=t+pt[i];

}

}

printf("PNum Start Burst CompletionTime\n");

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

{

printf("%d\t%d\t%d\t%d\n",p[i],s[i],pt[i],c[i]);

}

return 0;

}

1. **To find Optimal solution for a Knapsack Problem using Greedy Method**

**Code:**

#include<stdio.h>

int main()

{

int n,i,bag,itemp,j;

printf("Enter the number of items: ");

scanf("%d",&n);

int profit[n],weight[n],item[n];

float ppw[n],ftemp,p=0;

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

{

printf("\n\nDetails of item number: %d\n",i+1);

printf("Item name: ");

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

printf("Profit: ");

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

printf("Weight: ");

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

ppw[i]=(float)(profit[i]/weight[i]);

}

printf("\n\n");

printf("Enter the weight of the bag: ");

scanf("%d",&bag);

printf("Sorting the process by profit per weight\n");

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

{

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

{

if(ppw[i]<ppw[j])

{

ftemp=ppw[i];

ppw[i]=ppw[j];

ppw[j]=ftemp;

itemp=profit[i];

profit[i]=profit[j];

profit[j]=itemp;

itemp=weight[i];

weight[i]=weight[j];

weight[j]=itemp;

itemp=item[i];

item[i]=item[j];

item[j]=itemp;

}

}

}

printf("\n\n");

printf("Item name Profit Weight Profit per weight\n");

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

{

printf("%d %d %d %f\n",item[i],profit[i],weight[i],ppw[i]);

}

while(bag>0)

{

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

{

///////

if (bag==0)

break;

//////

else if(bag>weight[i])

{

bag=bag-weight[i];

p=p+profit[i];

}

else

{

p=p+ppw[i]\*bag;

bag=0;

}

}

}

printf("Total Profit by Knapsack problem is: %f\n",p);

return 0;

}

1. **Implement the file or code compression using Huffman’s algorithm.**

// C program for Huffman Coding

#include <stdio.h>

#include <stdlib.h>

// 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 Hufmman tree) nodes

struct MinHeap {

// Current size of min heap

unsigned size;

// capacity of min heap

unsigned capacity;

// Attay 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 funvtion 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)

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

printf("\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)) {

printf("%c: ", 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 program to test above functions

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;

}