DESIGN AND ANANLYSIS OF ALGORITHMS – GREEDY ALGORITHM

Name - ANIKET SHARMA

Reg No. - RA1911003030436

Branch- CSE – I

IMPLEMENT HUFFMAN CODING USING GREEDY ALGORITHM:

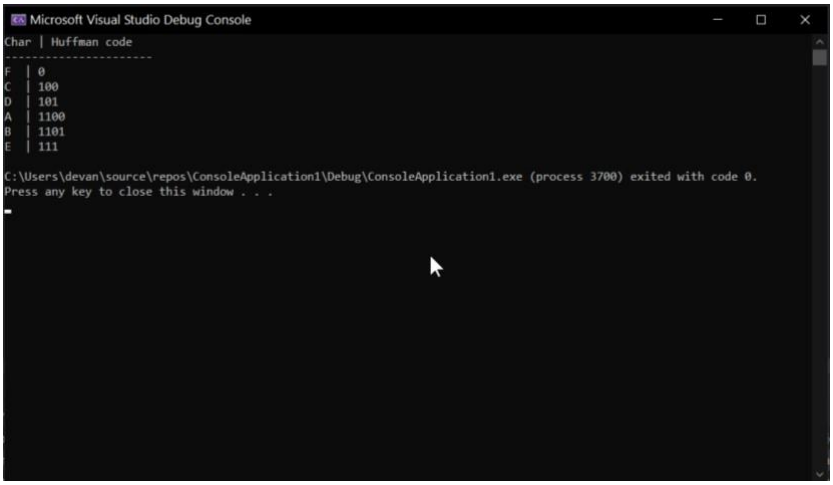
```cpp
#include <iostream>
using namespace std;
#define MAX_TREE_HT 50

struct MinHNode {
    unsigned freq;
    char item;
    struct MinHNode* left, * right;
};
struct MinH {
    unsigned size;
    unsigned capacity;
    struct MinHNode** array;
};

struct MinHNode* newNode(char item, unsigned freq) {
    struct MinHNode* temp = (struct MinHNode*)malloc(sizeof(struct MinHNode));
    temp->left = temp->right = NULL;
    temp->item = item;
    temp->freq = freq;

    return temp;
}

struct MinH* createMinH(unsigned capacity) {
    struct MinH* minHeap = (struct MinH*)malloc(sizeof(struct MinH));
    minHeap->size = 0;
    minHeap->capacity = capacity;
    minHeap->array = (struct MinHNode**)malloc(minHeap->capacity * sizeof(struct MinHNode*));

    return minHeap;
}
```

```cpp

void printArray(int arr[], int n) {
    int i;
    for (i = 0; i < n; ++i)
    {
        cout << arr[i];
    }
    cout << "\n";
}

void swapMinHNode(struct MinHNode** a, struct MinHNode** b) {
    struct MinHNode* t = *a;
    *a = *b;
    *b = t;
}

void minHeapify(struct MinH* 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) {
        swapMinHNode(&minHeap->array[smallest],
            &minHeap->array[idx]);
        minHeapify(minHeap, smallest);
    }
```

```cpp
}

int checkSizeOne(struct MinH* minHeap) {
    return (minHeap->size == 1);
}

struct MinHNode* extractMin(struct MinH* minHeap) {
    struct MinHNode* temp = minHeap->array[0];
    minHeap->array[0] = minHeap->array[minHeap->size - 1];
    --minHeap->size;
    minHeapify(minHeap, 0);

    return temp;
}

void insertMinHeap(struct MinH* minHeap, struct MinHNode* 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;
}

void buildMinHeap(struct MinH* minHeap) {
    int n = minHeap->size - 1;
    int i;
    for (i = (n - 1) / 2; i >= 0; --i) {
        minHeapify(minHeap, i);
    }
}
```

```
int isLeaf(struct MinHNode* root) {
    return !(root->left) && !(root->right);
}

struct MinH* createAndBuildMinHeap(char item[], int freq[], int size) {
    struct MinH* minHeap = createMinH(size);
    for (int i = 0; i < size; ++i) {
        minHeap->array[i] = newNode(item[i], freq[i]);
    }
    minHeap->size = size;
    buildMinHeap(minHeap);

    return minHeap;
}

struct MinHNode* buildHfTree(char item[], int freq[], int size) {
    struct MinHNode* left, * right, * top;
    struct MinH* minHeap = createAndBuildMinHeap(item, freq, size);

    while (!checkSizeOne(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);
}
void printHCodes(struct MinHNode* root, int arr[], int top) {
    if (root->left) {
        arr[top] = 0;
        printHCodes(root->left, arr, top + 1);
    }
    if (root->right) {
        arr[top] = 1;
        printHCodes(root->right, arr, top + 1);
    }
    if (isLeaf(root)) {
        cout << root->item << "  | ";
        printArray(arr, top);
    }
}

void HuffmanCodes(char item[], int freq[], int size) {
    struct MinHNode* root = buildHfTree(item, freq, size);
    int arr[MAX_TREE_HT], top = 0;
    printHCodes(root, 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]);
    cout << "Char | Huffman code ";
    cout << "\n--------------------\n";
    HuffmanCodes(arr, freq, size);
    return 0;
}
```

OUTPUT:



IMPLEMENT FRACTIONAL KNAPSACK USING GREEDY ALGORITHM:

```cpp
#include <iostream>
#include<math.h>
#include<algorithm>
using namespace std;
class item
{
public: int value, weight; item(int value, int weight) { this->value=value; this->weight=weight; }
};
bool cmp(item a,item b)
{
    double r1 = (double)a.value / (double)a.weight;
double r2 = (double)b.value / (double)b.weight;
return r1 > r2;
}
double fractionalKnapsack(int w,item arr[], int n)
{
    sort(arr, arr + n, cmp);
    int curweight = 0;
    double finalvalue = 0.0;
    for (int i = 0; i < n; i++)
    {
        if (curweight + arr[i].weight <= w)
        {
            curweight += arr[i].weight;
            finalvalue += arr[i].value;
        }
        else
        {
            int remain = w - curweight;
            finalvalue += arr[i].value * ((double)remain / (double)arr[i].weight);
            break;
        }
    }
    return finalvalue;
} int main()
{
    int w = 50;
    item arr[] = { {60,10}, {100,20}, {120,30} };
    int n = sizeof(arr) / sizeof(arr[0]);
    cout << "Maximum value we can obtain = "<< fractionalKnapsack(w,arr,n);
    return 0;
}
```

OUTPUT: