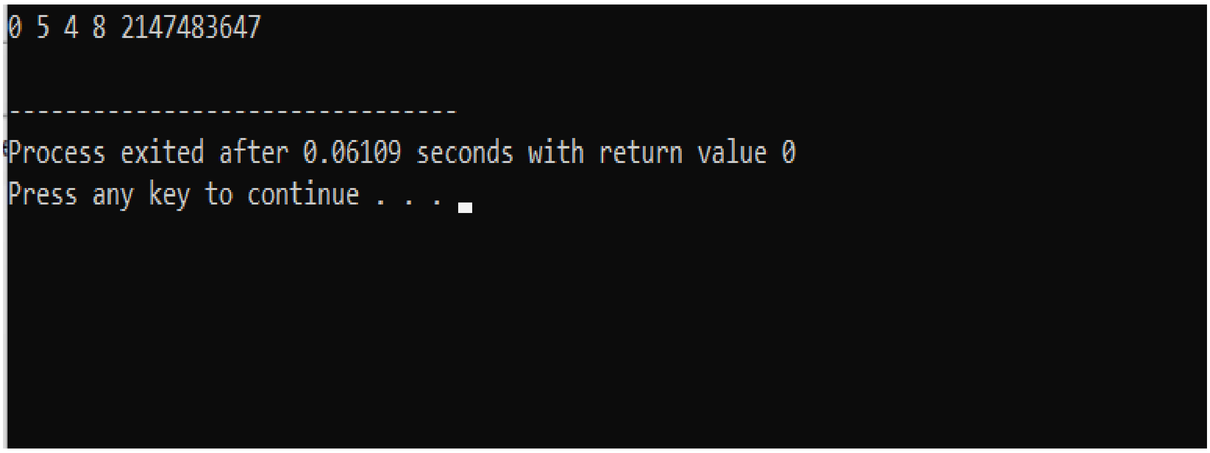
Bellman Ford Algorithm:

```
#include <bits/stdc++.h>
struct Edge {
  int u;
  int v;
  int w;
};
struct Graph {
  int V;
  int E;
  struct Edge* edge;
};
struct Graph* createGraph(int V, int E) {
  struct Graph* graph = new Graph;
  graph->V = V;
  graph->E = E;
  graph->edge = new Edge[E];
  return graph;
}
void printArr(int arr[], int size) {
  int i;
  for (i = 0; i < size; i++) {
    printf("%d ", arr[i]);
  }
  printf("\n");
}
void BellmanFord(struct Graph* graph, int u) {
  int V = graph->V;
  int E = graph->E;
  int dist[V];
  for (int i = 0; i < V; i++)
    dist[i] = INT_MAX;
  dist[u] = 0;
  for (int i = 1; i <= V - 1; i++) {
    for (int j = 0; j < E; j++) {
      int u = graph->edge[j].u;
      int v = graph->edge[j].v;
      int w = graph->edge[j].w;
      if (dist[u] != INT_MAX && dist[u] + w < dist[v])
        dist[v] = dist[u] + w;
    }
  }
  for (int i = 0; i < E; i++) {
    int u = graph->edge[i].u;
    int v = graph->edge[i].v;
    int w = graph->edge[i].w;
    if (dist[u] != INT_MAX && dist[u] + w < dist[v]) {
      printf("Graph contains negative w cycle");
      return;
    }
  }
  printArr(dist, V);
  return;
}
int main() {
  int V = 5;
  int E = 8;
  struct Graph* graph = createGraph(V, E);
  graph->edge[0].u = 0;
  graph->edge[0].v = 1;
  graph->edge[0].w = 5;
  graph->edge[1].u = 0;
  graph->edge[1].v = 2;
  graph->edge[1].w = 4;
  graph->edge[2].u = 1;
  graph->edge[2].v = 3;
  graph->edge[2].w = 3;
  graph->edge[3].u = 2;
  graph->edge[3].v = 1;
  graph->edge[3].w = 6;
  graph->edge[4].u = 3;
  graph->edge[4].v = 2;
  graph->edge[4].w = 2;
  BellmanFord(graph, 0);
  return 0;
}
```

OUTPUT:



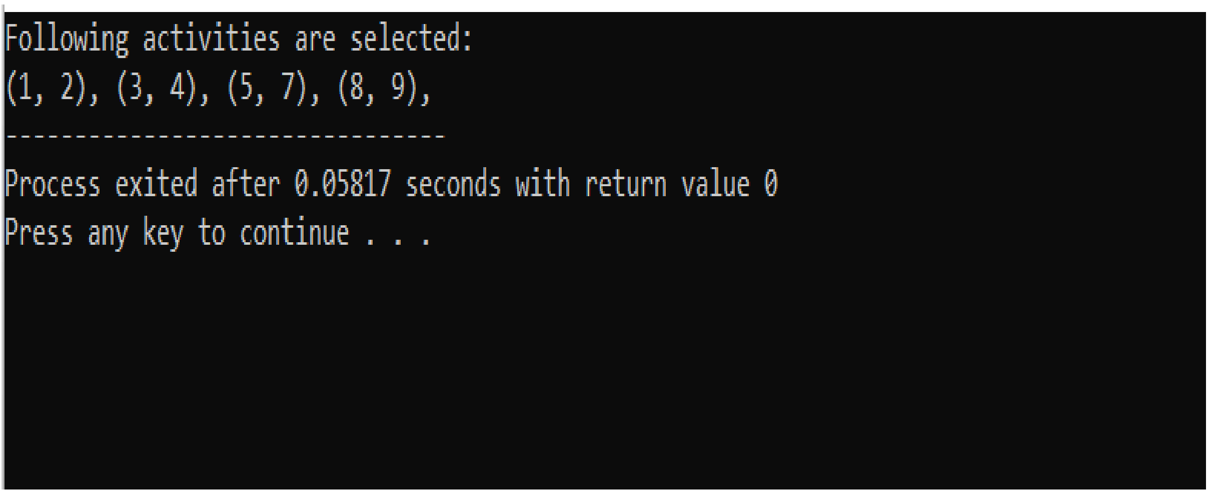
```
0 5 4 8 2147483647


--------------------------------
Process exited after 0.06109 seconds with return value 0
Press any key to continue . . .
```

Activity Selection:

```
#include <bits/stdc++.h>
using namespace std;
struct Activity {
 int start, end;
};
bool compare(Activity a, Activity b) {
 return (a.end < b.end);
}
void printMaxActivities(Activity arr[], int n) {
 sort(arr, arr + n, compare);
 cout << "Following activities are selected: \n";
 int i = 0;
 cout << "(" << arr[i].start << ", " << arr[i].end << "), ";
 for (int j = 1; j < n; j++) {
 if (arr[j].start >= arr[i].end) {
 cout << "(" << arr[j].start << ", " << arr[j].end << "), ";
 i = j;
 }
 }
}
int main() {
 Activity arr[] = {{5, 9}, {1, 2}, {3, 4}, {0, 6}, {5, 7}, {8, 9}};
 int n = sizeof(arr) / sizeof(arr[0]);
 printMaxActivities(arr, n);
 return 0;
}
```

OUTPUT:



```
Following activities are selected:
(1, 2), (3, 4), (5, 7), (8, 9),
--------------------------------
Process exited after 0.05817 seconds with return value 0
Press any key to continue . . .
```

Huffman Code:

```
#include <iostream>
#include<malloc.h>
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;
}
```

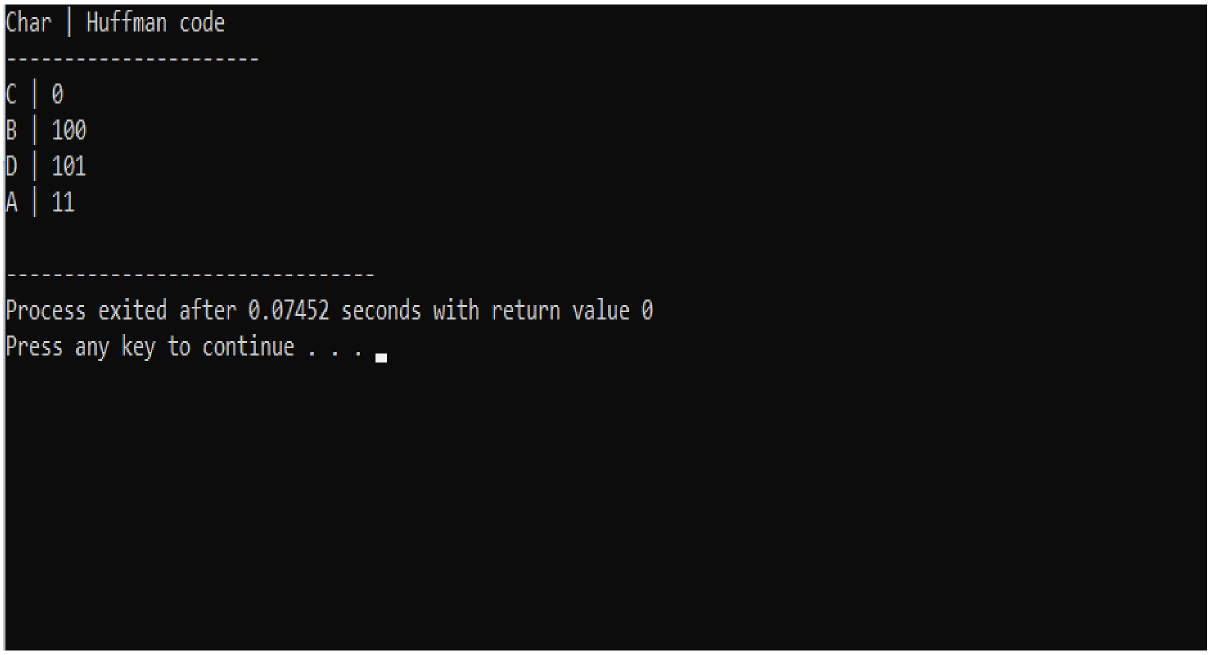
```
void printArray(int arr[], int n)
 {
 int i;
 for (i = 0; i < n; ++i)
 cout << arr[i];
 cout << "\n";
}
void swapMinHNode(struct MinHNode **a, struct MinHNode **b) {
 struct MinHNode *t = *a;
 *a = *b;
 *b = t;
}
void minHeapify(struct MinH *minHeap, int idx) {
 int smallest = idx;
 int left = 2 * 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);
 }
}
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'};
 int freq[] = {5, 1, 6, 3};
 int size = sizeof(arr) / sizeof(arr[0]);
 cout << "Char | Huffman code ";
 cout << "\n----------------------\n";
 HuffmanCodes(arr, freq, size);
 }
```

OUTPUT:



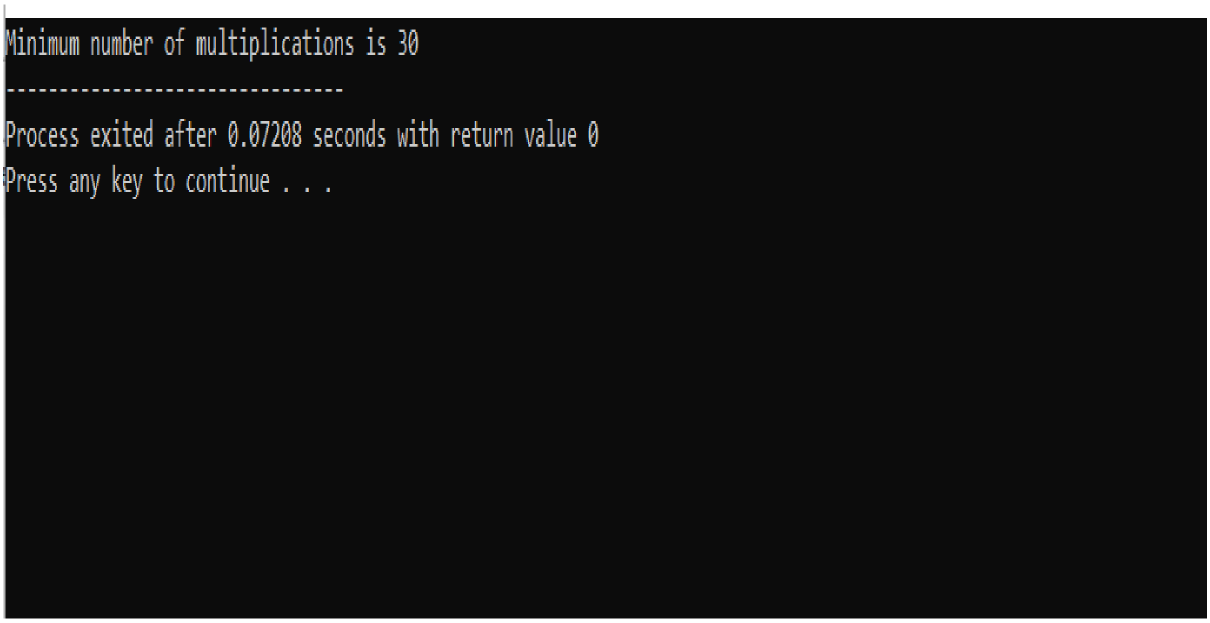
```
Char | Huffman code
---------------------
C | 0
B | 100
D | 101
A | 11

--------------------------------
Process exited after 0.07452 seconds with return value 0
Press any key to continue . . .
```

Matrix Chain Multiplication:

```
#include <bits/stdc++.h>
using namespace std;
int MatrixChainOrder(int p[], int i, int j)
{
 if (i == j)
 return 0;
 int k;
 int mini = INT_MAX;
 int count;
 for (k = i; k < j; k++)
 {
 count = MatrixChainOrder(p, i, k) + MatrixChainOrder(p, k + 1, j) + p[i - 1] * p[k] * p[j];
 mini = min(count, mini);
 }
 return mini;
}
int main()
{
 int arr[] = { 1, 2, 3, 4, 3 };
 int N = sizeof(arr) / sizeof(arr[0]);
 cout << "Minimum number of multiplications is "<<MatrixChainOrder(arr,1,N-1);
 return 0;
}
```

OUTPUT:



```
Minimum number of multiplications is 30
--------------------------------
Process exited after 0.07208 seconds with return value 0
Press any key to continue . . .
```