NB: la maggior parte degli algoritmi necessita di creazione file di input.

ORDINAMENTI PER CONFRONTI:

```
//ordinamento

#include <iostream>
#include <vector>
#include <fstream>

using namespace std;

class Array{
	public:
		vector<int> A;
		Array(vector<int> Arr):A(Arr){}

	void swap(int& i, int& j){
		int temp = i;
		i = j;
		j = temp;
	}

	void compswap(int& i, int& j){
		if(j < i){
			swap(i, j);
		}
	}

	void BubbleSort(vector<int>&A){
		for(int i=0; i<A.size()-1; i++){
			for(int j=1; j<A.size()-i; j++){
				compswap(A[j-1], A[j]);
			}
		}
	}

	void InsertionSort(vector<int> &A){
		for(int i=1; i<A.size(); i++){
			int key = A[i];
			int j = i-1;
			while(j>=0 && A[j]>key){
				A[j+1]=A[j];
				j--;
			}
			A[j+1]=key;
		}
	}

	void Merge(vector<int>& A, int l, int m, int r) {
    int i, j, k;
```

```
    int n1 = m - l + 1;
    int n2 = r - m;
    vector<int> L, R;

    for (i = 0; i < n1; i++) {
        L.push_back(A[l + i]);
    }

    for (j = 0; j < n2; j++) {
        R.push_back(A[m + 1 + j]);
    }

    i = 0;
    j = 0;
    k = l;

    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            A[k] = L[i];
            i++;
        } else {
            A[k] = R[j];
            j++;
        }
        k++;
    }

    while (i < n1) {
        A[k] = L[i];
        i++;
        k++;
    }

    while (j < n2) {
        A[k] = R[j];
        j++;
        k++;
    }
}

void MergeSort(vector<int>& A, int p, int q) {
    if (p < q) {
        int r = (p + q) / 2;
        MergeSort(A, p, r);
        MergeSort(A, r + 1, q);
        Merge(A, p, r, q);
    }
}

};

int main(){
vector<int> A;
```

```
ifstream fileinput("numeri.txt");
if(!fileinput.is_open()){
      cout<<"errore"<<endl;
      return 1;
}
int num;
while(fileinput>>num){
      A.push_back(num);
}
fileinput.close();

Array myArray(A);
myArray.MergeSort(A, 0, A.size() - 1);


ofstream fileoutput("o.txt");
if(!fileoutput.is_open()){
      cout<<"errore"<<endl;
      return 1;
}

for(auto el: A){
    fileoutput << el << " ";
}
fileoutput<< endl;
fileoutput.close();

return 0;
}
```

ORDINAMENTI LINEARI:

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <cmath>

using namespace std;

template<typename T>
void countingSort(vector<T>& numbers){
    T maxVal = *max_element(numbers.begin(), numbers.end()); // corretta inizializzazione di maxVal
    vector<int> cont(maxVal+1);

    for(T& n : numbers){
        cont[n]++;
    }

    int index = 0;
    for(int i=0; i<=maxVal; i++){ // ciclo fino a maxVal incluso
        for(int j = 0; j<cont[i]; j++){
            numbers[index++]=i;
```

```
        }
    }
}

template <typename T>
void radixSort(vector<T>& numbers){
     T maxVal = *max_element(numbers.begin(), numbers.end());
     int numDigits = (int)log10(maxVal)+1;
     for(int i=0; i<numDigits; i++){
          vector<vector<int>> buckets(10);
          int divisor = pow(10, i);
          for(T& num : numbers){
               int digit = (num/divisor)%10;
               buckets[digit].push_back(num);
          }
          numbers.clear();

          for(auto& bucket : buckets){
               for(int& num:bucket){
                    numbers.push_back(num);
               }
          }
     }
}




int main(){
    vector<int> numbers = { 0, 5, 4, 9, 22, 44, 7};
    radixSort(numbers);

    for(int i : numbers){ // stampa degli elementi con il for-each loop
        cout << i << " ";
    }

    return 0;
}
```

HEAP:

```
#include <iostream>
#include <vector>
#include <fstream>

using namespace std;

template <typename T>
class Node{
     public:
          T k;
```

```
        Node(T k): k(k){}
};

template <typename T>
class Heap{
    private:
        vector<Node<T>*> data;
        int heapsize;


        void max_heapify(int i){
            int l = i*2+1;
            int r = i*2+2;
            int max = i;

            if(l<heapsize && data[l]->k > data[max]->k){
                max = l;
            }
            if(r<heapsize && data[r]->k > data[max]->k){
                max = r;
            }
            if(max!= i){
                swap(data[i], data[max]);
                max_heapify(max);
            }
        }

        void build_max_heap(){
            for(int i =heapsize/2; i>=0; i--){
                max_heapify(i);
            }
        }

    public:
        Heap(vector<Node<T>*> &values) : data(values),
heapsize(values.size()){
            build_max_heap();
        }

        T getmax() const{
            return data[0]->k;
        }

        vector<Node<T>*> getNodes(){
            return this->data;
        }

        T extractmax(){

            if(heapsize == 0){
                throw out_of_range("heap is empty");
            }
            T max = getmax();
```

```
				data[0] = data[heapsize-1];
				heapsize--;
				data.pop_back();
				max_heapify(0);
				return max;
			}


			void insert(T key){
				heapsize++;
				data.push_back(new Node<T>(key));

				int i = heapsize-1;
				while(i>=0 && data[(i-1)/2]->k<data[i]->k){
					swap(data[i], data[(i-1)/2]);
					i=(i-1)/2;
				}
			}

			void heapsort(){
				build_max_heap();
				for(int j = heapsize-1; j>=1; j--){
					swap(data[0], data[j]);
					heapsize--;
					max_heapify(0);
				}
			}
};

int main(){
	ifstream in("nodiheap.txt");
	//controlliamo apertura

	int val;
	vector<Node<int>*> nodes;

	while(in>>val){
		Node<int>* nodo = new Node<int>(val);
		nodes.push_back(nodo);
	}


	Heap<int> H(nodes);

	for(auto h : H.getNodes()){
		cout<<h->k<<endl;
	}

		cout<<endl;

	cout<<"inseriamo 77"<<endl;
	int k = 77;
```

```
    H.insert(k);


    for(auto h : H.getNodes()){
        cout<<h->k<<endl;
    }

        cout<<endl;


    cout<<"inseriamo 9"<<endl;
    int kk = 9;
    H.insert(kk);

    for(auto h : H.getNodes()){
        cout<<h->k<<endl;
    }


    cout<<endl;
    cout<<"estraiamo il massimo : "<<H.extractmax()<<endl;


    for(auto h : H.getNodes()){
        cout<<h->k<<endl;
    }

    cout<<endl;
    cout<<"facciamo l'heapsort:"<<endl;
    H.heapsort();
        for(auto h : H.getNodes()){
        cout<<h->k<<endl;
    }
    return 0;
}
```

ALGORITMI DIVIDE ET IMPERA:

```
//algoritmi d.e.i

#include <iostream>
#include <vector>

using namespace std;

int maxCrossingSum(vector<int> a, int l, int m, int h){
    int sum = 0;
    int left_sum = INT_MIN;
    for(int i = m; i>=l; i--){
        sum = sum+a[i];

        if(sum>left_sum){
            left_sum=sum;
```

```
        }

    }
    sum=0;
    int right_sum=INT_MIN;
    for(int i=m+1; i<=h; i++){
        sum=sum+a[i];

        if (sum>right_sum){
            right_sum=sum;
        }
    }

    return left_sum+right_sum;
}

int maxSubArray(vector<int> a, int l, int h){
    if(l==h){
        return a[l];
    }
    int m = (l+h)/2;
    int left_sum=maxSubArray(a, l, m);
    int right_sum=maxSubArray(a, m+1, h);
    int cross_sum=maxCrossingSum(a, l, m, h);


    return max(max(left_sum, right_sum), cross_sum);

}


bool RicercaBinaria(vector<int>a, int x, int low, int high){
    int mid = (low+high)/2;
    if (a[mid]==x){
        return true;
    } else if(high<=low){
        return false;
    }
    else if(x<a[mid]){
            return RicercaBinaria(a, x, low, mid-1);
        }
    else{
            return RicercaBinaria(a,x, mid+1, high);
        }
    }

int main(){
    vector<int>a;
    for(int i=0; i<10;i++){
        a.push_back(i);
    }

    int sum;
```

```
	int n = a.size();
	sum = maxSubArray(a, 0, n-1);
	cout<<sum;
	cout<<endl;
	int x = 4;
	bool result = RicercaBinaria(a,x,0, n);
	cout<<result;


	return 0;
}
```

ALBERO:

```
#ifndef Albero_HPP
#define Albero_HPP

#include <iostream>
#include <vector>
#include <fstream>

using namespace std;

template<typename T>
class Node{
	public:
		T val;
		Node* parent;
		Node* left;
		Node* right;

		Node (T val){
			this->val = val;
			left = right = parent = nullptr;

		}


};

template<typename T>
class ABR{
	public:
		Node<T>* root;

		ABR(){
			root = nullptr;
		}

		void insertNode(Node<T>* NodeToInsert){
			Node<T>* parentNode = nullptr;
			Node<T>* currentNode = root;
```

```
		while(currentNode != nullptr){
			parentNode = currentNode;

			if(NodeToInsert->val<currentNode->val){
				currentNode=currentNode->left;
			}
			else{
				currentNode=currentNode->right;
			}
		}
			NodeToInsert->parent = parentNode;

			if(parentNode == nullptr){
				root = NodeToInsert;

			}else if(NodeToInsert->val < parentNode->val){
				parentNode->left = NodeToInsert;

			}else{
				parentNode->right = NodeToInsert;
			}

	}

	void transplant(Node<T>* u, Node<T>* v){
		if(u->parent == nullptr){
			root = v;
		}
		else if(u==u->parent->left){
			u->parent->left=v;
		}else{
			u->parent->right=v;
		}

		if(v!=nullptr){
			v->parent = u->parent;
		}
	}

	void deleteNode(T val){
		Node<T>* nodeToDelete = searchNode(val);

		if(nodeToDelete == nullptr){
			return;
		}
		if(nodeToDelete->left == nullptr){
			transplant(nodeToDelete, nodeToDelete->right);
		}else if(nodeToDelete->right == nullptr){
			transplant(nodeToDelete, nodeToDelete->left);
		}else{
			Node<T>* minimum = getMinimum(nodeToDelete->right);
			if(minimum->parent->val != nodeToDelete->val){
				transplant(minimum, minimum->right);
```

```
                    minimum->right = nodeToDelete->right;
                    minimum->right->parent=minimum;
                }
                transplant(nodeToDelete, minimum);
                minimum->left=nodeToDelete->left;
                minimum->left->parent=minimum;
            }
            delete nodeToDelete;
        }

        Node<T>* getMinimum(Node<T>* node){

            while(node->left != nullptr){
                node=node->left;

            }
            return node;
        }

        Node<T>* getMaximum(Node<T>* node){

            while(node->right != nullptr){
                node=node->right;

            }
            return node;
        }

        Node<T>* getPredecessor(Node<T>* node){
            if(node->left != nullptr){
                return getMaximum(node->left);
            }
            Node<T>* parentNode = node->parent;
            while(parentNode != nullptr && node==parentNode->left){
                node = parentNode;
                parentNode = parentNode->parent;
            }
        return parentNode;
        }

        Node<T>* getSuccessor(Node<T>* node){
            if(node->right != nullptr){
                return getMinimum(node->right);
            }
            Node<T>* parentNode = node->parent;
            while(parentNode != nullptr && node==parentNode->right){
                node = parentNode;
                parentNode = parentNode->parent;
            }
        return parentNode;
        }

        Node<T>* searchNode(T val){
```

```
                Node<T>* current = root;
                while(current != nullptr && val != current->val){
                     if(val<current->val){
                          current = current->left;
                     }
                     else{
                          current = current->right;
                     }
                }
                return current;
           }


           void preorderVisit(Node<T>* node, vector<T>& v){
                if(node){
                     v.push_back(node->val);
                     preorderVisit(node->left, v);
                     preorderVisit(node->right, v);
                }
           }

           void inorderVisit(Node<T>* node, vector<T>& v){
                if(node){
                     inorderVisit(node->left, v);
                     v.push_back(node->val);
                     inorderVisit(node->right, v);
                }
           }

           void postorderVisit(Node<T>* node, vector<T>& v){
                if(node){
                     postorderVisit(node->left, v);
                     postorderVisit(node->right, v);
                     v.push_back(node->val);
                }
           }

           int getHeight(Node<T>* node){
                if(node==nullptr){
                     return 0;
                }
                return 1 + max(getHeight(node->left),getHeight(node-
>right));
           }

};

#endif
#include "Albero.hpp"

int main(){

     ifstream fileread("input.txt");
```

```
    if(!fileread.is_open())    {
        cout<<"errore";
        return 1;
    }

    ABR<int>* albero = new ABR<int>();
    int element;

    while(fileread>>element){
        Node<int>* newNode = new Node<int>(element);
        albero->insertNode(newNode);
    }

    fileread.close();

    ofstream filewrite("output.txt");
    if(!filewrite.is_open())   {
        cout<<"errore";
        return 1;
    }

    //cerchiamo il nodo 3
    int val = 3;
    Node<int>* searched_node = albero->searchNode(val);

    if(searched_node == nullptr){
        filewrite<<"il nodo non c'è "<<endl;
    }else{
            filewrite<<"il nodo c'è "<<endl;
    }

    albero->deleteNode(3);

    Node<int>* searched_node3 = albero->searchNode(val);

    if(searched_node3 == nullptr){
        filewrite<<"il nodo non c'è "<<endl;
    }else{
            filewrite<<"il nodo c'è "<<endl;
    }



    filewrite.close();

    cout<<"creato file output"<<endl;
    return 0;
}
```

Esempio file input

```
5 7
0 0
1 2
2 4
```

```
3 6
4 8
0 1 5
0 2 2
1 2 3
1 3 4
2 3 2
2 4 1
3 4 3
```

ZAINO 01-FRAZIONARIO:

```
#include <iostream>
#include <vector>
#include <fstream>


using namespace std;

template<typename T>
class Item{
	public:
		T value;
		T weight;
		Item(T value, T weight): value(value), weight(weight){}

};

template<typename T>
class Knapsack{
	public:
		vector<Item<T>*> items;
		T value;
		T capacity;

		Knapsack(vector<Item<T>*> items, T capacity) : items(items),
capacity(capacity){}

		vector<Item<T>> solve01(){
			int n = items.size();
			vector<Item<T>> inTheKnapsack;
			vector<vector<T>> maxValues(n+1,
vector<T>(capacity+1,0));

			for(int i=1; i<=n; i++){
				for(int j=1; j<=capacity; j++){
					if(items[i-1]->weight >j){
						maxValues[i][j]=maxValues[i-1][j];
					}
					else{
						maxValues[i][j]=max(maxValues[i-
1][j],items[i-1]->value + maxValues[i-1][j-items[i-1]->weight]);
```

```
                    }
                }
            }

            int i = n;
           int j = capacity;
           while (i > 0 && j > 0){
               if (maxValues[i][j] != maxValues[i-1][j]){
                   inTheKnapsack.push_back(*items[i-1]);
                   j -= items[i-1]->weight;
               }
               i--;
           }


            return inTheKnapsack;

       }

       vector<pair<Item<T>, double>> solvefractional(){
           vector<pair<Item<T>, double>> x;
           int n = items.size();
           int k=1;
           T value = 0.0;

           while(k<=n && capacity>0.0){
               if(items[k-1]->weight <= capacity){
                   x.push_back(make_pair(*items[k-1], 1.0));
                   value += items[k-1]->value;
                   capacity -= items[k-1]->weight;
               }
               else{
                   double fraction = capacity/items[k-1]->weight;
                   x.push_back(make_pair(*items[k-1], fraction));
                   value += fraction*items[k-1]->value;
                   capacity = 0.0;
               }
               k++;
           }
           return x;
       }



};

int main() {
    // Creazione di una lista di oggetti
    Item<int>* item1 = new Item<int>(4, 2);
    Item<int>* item2 = new Item<int>(3, 1);
    Item<int>* item3 = new Item<int>(6, 3);
    Item<int>* item4 = new Item<int>(7, 2);
```

```
    vector<Item<int>*> items {item1, item2, item3, item4};

    // Creazione dello zaino con una capacità massima di 7
    Knapsack<int> knapsack(items, 7);

    // Risoluzione del problema con il metodo solve01
    vector<Item<int>> solution = knapsack.solve01();

    // Stampa degli oggetti all'interno dello zaino
    cout << "Gli oggetti all'interno dello zaino sono:" << endl;
    for (auto item : solution) {
        cout << "Valore: " << item.value << " - Peso: " << item.weight <<
endl;
    }

     vector<pair<Item<int>, double>> solutionF =
knapsack.solvefractional();
     cout << "Gli oggetti all'interno dello zaino sono:" << endl;
     for (auto item : solutionF) {
    cout << "Valore: " << item.first.value << " - Peso: " <<
item.first.weight << " - Frazione: " << item.second << endl;
     }
    return 0;
}
```

FASTEST-WAY:

```
#include <iostream>
#include <vector>
using namespace std;

class CatenaMontaggio {
private:
    vector<int> tempi_elaborazione;
    vector<int> tempi_trasferimento;
    int tempo_avvio;
    int tempo_fermo;
public:
    // Costruttore
    CatenaMontaggio(vector<int> elab, vector<int> trasf, int avvio, int
fermo) {
        tempi_elaborazione = elab;
        tempi_trasferimento = trasf;
        tempo_avvio = avvio;
        tempo_fermo = fermo;
    }

    // Metodo per calcolare il percorso più veloce
    int percorsoVeloce(int e1, int e2, int x1, int x2, vector<int>& L) {
        int n = tempi_elaborazione.size();
        vector<int> f1(n), f2(n);
        f1[0] = tempo_avvio + tempi_elaborazione[0];
```

```
        f2[0] = tempo_avvio + tempi_elaborazione[1] +
tempi_trasferimento[0];
        for (int j = 1; j < n; j++) {
            if (f1[j-1] + tempi_elaborazione[j] + e1 <= f2[j-1] +
tempi_trasferimento[j-1] + tempi_elaborazione[j] + e2) {
                f1[j] = f1[j-1] + tempi_elaborazione[j];
                L[j] = 1;
            }
            else {
                f1[j] = f2[j-1] + tempi_trasferimento[j-1] +
tempi_elaborazione[j];
                L[j] = 2;
            }
            if (f2[j-1] + tempi_elaborazione[j] + e2 <= f1[j-1] +
tempi_trasferimento[j-1] + tempi_elaborazione[j+1] + e1) {
                f2[j] = f2[j-1] + tempi_elaborazione[j];
                L[j] = 2;
            }
            else {
                f2[j] = f1[j-1] + tempi_trasferimento[j-1] +
tempi_elaborazione[j+1];
                L[j] = 1;
            }
        }
        int f_star = min(f1[n-1]+x1, f2[n-1]+x2);
        return f_star;
    }
};

int main() {
    // Definizione delle due catene di montaggio
    vector<int> elab1 = {4, 5, 3, 2};
    vector<int> trasf1 = {2, 1, 3};
    int avvio1 = 2, fermo1 = 1;
    CatenaMontaggio catena1(elab1, trasf1, avvio1, fermo1);

    vector<int> elab2 = {2, 10, 1, 4};
vector<int> trasf2 = {3, 2, 1};
int avvio2 = 4, fermo2 = 2;
CatenaMontaggio catena2(elab2, trasf2, avvio2, fermo2);

// Calcolo del percorso più veloce
int e1 = elab1[0], e2 = elab2[0];
int x1 = fermo1, x2 = fermo2;
vector<int> L1(elab1.size()), L2(elab2.size());
int f_star = min(catena1.percorsoVeloce(e1, e2, x1, x2, L1),
catena2.percorsoVeloce(e2, e1, x2, x1, L2));

// Stampa dei risultati
cout << "Il percorso piu veloce richiede " << f_star << " unita di
tempo." << endl;
cout << "Sequenza di svolgimento delle attivita sulla prima catena: ";
for (int i = 0; i < L1.size(); i++) {
```

```
    if (i > 0) {
        if (L1[i] != L1[i-1]) {
            cout << " | ";
        }
    }
    cout << "L" << L1[i];
}
cout << endl;
cout << "Sequenza di svolgimento delle attivita sulla seconda catena: ";
for (int i = 0; i < L2.size(); i++) {
    if (i > 0) {
        if (L2[i] != L2[i-1]) {
            cout << " | ";
        }
    }
    cout << "L" << L2[i];
}
cout << endl;
     return 0;
}
```

DISTANZA DI EDITING:

```
#include <iostream>
#include <vector>
#include <fstream>
#include <string>

using namespace std;

class Node{
     public:
          string word;
          Node* left;
          Node* right;

          Node(string word){
               this->word = word;
               left = nullptr;
               right = nullptr;
          }

          int editDistance(string s1, string s2){
               int len1 = s1.length();
               int len2 = s2.length();

               vector<vector<int>> dp(len1+1, vector<int>(len2+1));

               for(int i=0; i<= len1; i++){
                    for(int j=0; j<=len2; j++){
                         if(i==0){
                              dp[i][j]=j;
                         }else if(j==0){
```

```
                        dp[i][j]=i;
                    }else if(s1[i-1]==s2[j-1]){
                        dp[i][j]=dp[i-1][j-1];
                    }else{
                        dp[i][j]= 1 + min(dp[i-1][j],
min(dp[i][j-1], dp[i-1][j-1]));

                    }
                }
            }
            return dp[len1][len2];
        }

};

class Tree {
    public:
        vector<Node*> nodes;
        Node* root;
        //Costruttore che crea un albero a partire da un vettore di
parole
        Tree(vector<string>& words) {
            // Inserimento della radice
            root = new Node(words[0]);
            nodes.push_back(root);
            // Inserimento degli altri nodi
            for (int i = 1; i < words.size(); i++) {
                insert(words[i]);
            }
        }

        // Funzione per inserire un nuovo nodo nell'albero
        void insert(string word) {
            Node* current = root;
            Node* parent = NULL;
            while (current != NULL) {
                parent = current;
                if (word < current->word) {
                    current = current->left;
                } else if (word > current->word) {
                    current = current->right;
                } else {
                    return;
                }
            }
            Node* newNode = new Node(word);
            if (word < parent->word) {
                parent->left = newNode;
            } else {
                parent->right = newNode;
            }
            nodes.push_back(newNode);
        }
```

```
          // Funzione per cercare tutte le parole nell'albero che
distano esattamente k operazioni dalla parola di riferimento
          vector<string> findWords(string word, int k) {
               vector<string> result;
               for (Node* node : nodes) {
                    if (node->editDistance(node->word, word) == k) {
                         result.push_back(node->word);
                    }
               }
               return result;
          }

};

int main(){

     vector<string> nomi;
     nomi.push_back("ilaria");
     nomi.push_back("gianfranco");
     nomi.push_back("sara");
     nomi.push_back("rosa");
     nomi.push_back("ernesto");
     nomi.push_back("diacono");
     nomi.push_back("braso");

     Tree tree(nomi);

     string word = "vaso";
     int k = 2;
     vector<string> result = tree.findWords(word, k);

     cout << "Parole a distanza " << k << " dalla parola " << word << ":
";
     for (string s : result) {
          cout << s << " ";
     }

     return 0;
}
```

GREEDY ACTIVITY SELECTOR:

```
//GREEDY ACTIVITY SELECTOR

#include <iostream>
#include <list>
#include <vector>
#include <map>


using namespace std;
```

```
void GreedyActivitySelector(list<int> A, vector<int> start, vector <int>
finish){

      int n;
            cout<<"inserire numero di attivita': ";
      cin>>n;
      int tempo_inizio;
      int tempo_fine;
      for(int i=0; i<n; i++){
            cout<<"tempo di inizio dell'attivita' "<<i<<" :";
            cin>>tempo_inizio;
            start.push_back(tempo_inizio);
            }
      cout<<endl;
      for(int x=0; x<n; x++){
            cout<<"tempo di fine dell'attivita' "<<x<<" :";
            cin>>tempo_fine;
            finish.push_back(tempo_fine);
      }

      cout<<endl;
      A.push_back(0);
      int prev=0;

      for(int j=1;j<n;j++){
            if(start[j]>=finish[prev]){
                  A.push_back(j);
                  prev=j;
            }
      }

            list<int>::iterator lii;
      for(lii=A.begin();lii!=A.end();lii++){
            cout<<"Attivita schedulate: "<<*lii<<endl;
      }


}
int main(){
      list<int> A ;
      vector<int> start;
      vector<int> finish;

      GreedyActivitySelector(A, start, finish);



return 0;
}
```

CODIFICA E DECODIFICA DI HUFFMAN:

```
#ifndef HUFF_HPP
```

```
#define HUF_HPP

#include <fstream>
#include <iostream>
#include <vector>
#include <queue>
#include <string>
#include <unordered_map>

using namespace std;

class Node{
	public:
		char data;
		int freq;
		Node* left;
		Node* right;

		Node(char data, int freq){
			this->data = data;
			this->freq = freq;
			this->left = this->right = nullptr;
		}
};

class Compare{
	public:
		bool operator()(Node* l, Node* r){
			return l->freq > r->freq;
		}
};

class HuffmanTree{
	public:
		Node* root;
		unordered_map<char, string> codes; // ad ogni carattere
corrisponderà una stringa codificata

		HuffmanTree(string data){
			unordered_map<char, int> freq; //ad ogni carattere
corrisponderà una frequenza
			for(char c: data){
				freq[c]++;
			}

			priority_queue<Node*, vector<Node*>, Compare> pq; //creo
la coda di priorità, e metto tutti i caratteri con le loro frequenze
			for(auto pair:freq){
				pq.push(new Node(pair.first, pair.second));
			}

			while(pq.size()>1){
				Node* left = pq.top();
```

```
                    pq.pop();
                    Node* right = pq.top();
                    pq.pop();

                    Node* parent = new Node('$', left->freq+right-
>freq); //il nostro nodo zeta che ha come frequenza la somma delle
frequenze di x e y ovvero dx e sx
                    parent->left = left;
                    parent->right = right;
                    pq.push(parent); //lo imposto come padre e lo metto
nella coda
               }

               root = pq.top();
               pq.pop();

               generateCode(root, " ");

          }

          void generateCode(Node* node, string code){
               if(node->left == nullptr && node->right == nullptr){
                    codes[node->data]=code;
                    return;
               }

               generateCode(node->left, code+"0");
               generateCode(node->right, code+"1");
          }

          string encode(string data){ //metodo che concatena tutte le
codifiche in un unica stringa
               string encoded = "";
               for(char c : data){
                    encoded += codes[c];
               }
               return encoded;
          }

          string decode(string data){
               string decoded = "";
               Node* current = root;
               for(char c : data){
                    if(c=='0'){
                         current = current->left;
                    }else{
                         current = current->right;
                    }

                    if(current->left == nullptr && current->right ==
nullptr){
                         decoded += current->data;
                         current = root;
```

```
                    }
                }
                return decoded;
            }

};

#endif
#include "HUFF.hpp"


int main(){

	ifstream inputFile("stringa.txt");
	//controllo file aperto corettamente

	string data;
	inputFile>>data;
	HuffmanTree tree(data);

	string encoded = tree.encode(data);
	string decoded = tree.decode(encoded);

	inputFile.close();

	ofstream outputFile("huffman.txt");
	outputFile<<"la stringa di partenza: "<<data<<endl;
	outputFile<<"la stringa codificata: "<<encoded<<endl;
	outputFile<<"la stringa decodificata: "<<decoded<<endl;

	outputFile.close();

	cout<<"creato file di output";


	return 0;
}
```

Esempio file input

parthenope

LCS:

```
#include <iostream>
#include <cstring>
using namespace std;

const int N = 1005;
string X, Y;
int n, m, c[N][N];
char b[N][N];

void printLCS(int i, int j) {
    if (i == 0 || j == 0)
```

```
        return;
    if (b[i][j] == '\\') {
        printLCS(i - 1, j - 1);
        cout << X[i - 1];
    } else if (b[i][j] == '|')
        printLCS(i - 1, j);
    else
        printLCS(i, j - 1);
}

void LCS() {
    n = X.length();
    m = Y.length();

    memset(c, 0, sizeof(c));
    memset(b, 0, sizeof(b));

    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            if (X[i - 1] == Y[j - 1]) {
                c[i][j] = c[i - 1][j - 1] + 1;
                b[i][j] = '\\';
            } else if (c[i - 1][j] >= c[i][j - 1]) {
                c[i][j] = c[i - 1][j];
                b[i][j] = '|';
            } else {
                c[i][j] = c[i][j - 1];
                b[i][j] = '<';
            }
        }
    }
}

void printB() {
    cout << "Matrice delle soluzioni B:" << endl;
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            cout << b[i][j] << " ";
        }
        cout << endl;
    }
}

int main() {
    cout << "Inserire la prima stringa: ";
    cin >> X;
    cout << "Inserire la seconda stringa: ";
    cin >> Y;

    LCS();
    cout << "LCS lunghezza: " << c[n][m] << endl;
    cout << "LCS: ";
    printLCS(n, m);
```

```
    cout << endl;

    printB();

    return 0;
}
```

GRAFO CON BFS, DFS, CONNECTED COMPONENTS, OPERAZIONI INSIEMI DISGIUNTI:

```
//Grafo

#ifndef Grafo_HPP
#define Grafo_HPP

#include <iostream>
#include <fstream>
#include <vector>
#include <map>
#include <functional>
#include <stack>
#include <queue>
#include <algorithm>
#include <unordered_map>
using namespace std;

const int WHITE = 0;
const int GREY = 1;
const int BLACK = 2;
const int INFINITY = 9999;

template<typename T>
class Node{
    public:
        T val, key;
        int rank;
        int color = WHITE;
        T discovery_time = 0;
        T finish_time = 0;
        T distance = INFINITY;

        Node<T>* parent;

        Node(T val) : val(val){}
};

template<typename T>
class Edge{
    public:
        T weight;
        Node<T>* source;
        Node<T>* destination;
```

```
        Edge(T weight, Node<T>* source, Node<T>* destination) :
weight(weight), source(source), destination(destination){}
};

template<typename T>
class minCompare {
public:
    bool operator()(Node<T>* a, Node<T>* b) {
        return a->key > b->key;
    }
};

template<typename T>
class Graph{
    public:
     T time = 0;
        vector<Node<T>*> nodes;
        vector<Edge<T>*> edges;
        map<T, vector<pair<T,T>>> adjacencyList;

        void addNode(Node<T>* node) {
            nodes.push_back(node);
        }

        void addEdge(Node<T>* source, Node<T>* destination, T weight) {
            Edge<T>* edge = new Edge<T>(weight, source, destination);
            edges.push_back(edge);
            adjacencyList[source->val].push_back(make_pair(destination-
>val, weight));
            // aggiungo anche l'arco inverso se il grafo non è orientato
            adjacencyList[destination->val].push_back(make_pair(source-
>val, weight));
        }

        void bfs(Node<T>* startNode) {
            for (Node<T>* node : nodes) {
                node->color = WHITE;
                node->distance = INFINITY;
                node->parent = nullptr;
            }
            startNode->color = GREY;
            startNode->distance = 0;
            startNode->parent = nullptr;

            queue<Node<T>*> q;
            q.push(startNode);

            while (!q.empty()) {
                Node<T>* u = q.front();
                q.pop();
                for (auto v : adjacencyList[u->val]) {
                    Node<T>* node = nullptr;
                    for (Node<T>* n : nodes) {
```

```
                if (n->val == v.first) {
                    node = n;
                    break;
                }
            }
            if (node == nullptr) {
                continue;
            }
            if (node->color == WHITE) {
                node->color = GREY;
                node->distance = u->distance + v.second;
                node->parent = u;
                q.push(node);
            }
        }
        u->color = BLACK;
    }
}

void dfs(){
   for(auto u: nodes){
        u->color = WHITE;
        u->parent = nullptr;
        }
        for(auto u: nodes){
             if(u->color == WHITE){
                  dfs_visit(u);
             }
        }
   }

   void dfs_visit(Node<T>* u){
        u->color = GREY;
        u->discovery_time=++time;
         for (auto v : adjacencyList[u->val]) {
            Node<T>* node = nullptr;
            for (Node<T>* n : nodes) {
                if (n->val == v.first) {
                    node = n;
                    break;
                }
            }
            if (node == nullptr) {
                continue;
            }
            if(node->color==WHITE){
             node->parent= u;
             dfs_visit(node);
                  }
        }
        u->color=BLACK;
        u->finish_time=++time;
   }
```

```
	void unionSet(Node<T>* x, Node<T>* y){
		linkSet(findSet(x), findSet(y));
	}

	void linkSet(Node<T>*x, Node<T>* y){
		if(x->rank > y->rank){
			y->parent = x;
		}else{
			x->parent = y;
		}

		if(x->rank == y->rank){
			y->rank++;
		}
	}

	Node<T>* findSet(Node<T>* x){
		if(x!=x->parent){
			x->parent = findSet(x->parent);
		}
	return x->parent;
	}

	void makeSet(Node<T>* x){
		x->parent = x;
		x->rank = 0;
	}

	void connected_components(){
		for(auto v : nodes){
			makeSet(v);
		}
		for(auto edge : edges){
			if(findSet(edge->source) != findSet(edge->destination)){
				unionSet(edge->source, edge->destination);
			}
		}
	}


};

#endif

#include "Grafo.hpp"


int main() {
    ifstream input("inputG.txt");
    if (!input) {
        cerr << "Errore durante la lettura del file di input" << endl;
```

```
        return 1;
    }

    int numNodes, numEdges;
    input >> numNodes >> numEdges;

    // Mappa che tiene traccia dei nodi già creati e dei loro valori
    unordered_map<int, Node<int>*> nodesMap;

    Graph<int> g;
    for (int i = 0; i < numEdges; i++) {
        int sourceVal, destVal, weight;
        input >> sourceVal >> destVal >> weight;

        // Cerca il nodo di partenza nella mappa, creandolo se non esiste
        Node<int>* sourceNode = nullptr;
        auto itSource = nodesMap.find(sourceVal);
        if (itSource == nodesMap.end()) {
            sourceNode = new Node<int>(sourceVal);
            nodesMap[sourceVal] = sourceNode;
            g.addNode(sourceNode);
        } else {
            sourceNode = itSource->second;
        }

        // Cerca il nodo di destinazione nella mappa, creandolo se non
esiste
        Node<int>* destNode = nullptr;
        auto itDest = nodesMap.find(destVal);
        if (itDest == nodesMap.end()) {
            destNode = new Node<int>(destVal);
            nodesMap[destVal] = destNode;
            g.addNode(destNode);
        } else {
            destNode = itDest->second;
        }

        g.addEdge(sourceNode, destNode, weight);
    }
     input.close();

    // Esegui BFS e stampa le distanze dai nodi sorgente
    g.bfs(nodesMap[1]);
    for (Node<int>* node : g.nodes) {
        cout << "Distanza dal nodo " << nodesMap[1]->val << " al nodo "
<< node->val << ": " << node->distance << endl;
    }

    // Esegui DFS e stampa i tempi di scoperta e completamento dei nodi
    g.dfs();
    for (Node<int>* node : g.nodes) {
```

```
        cout << "Nodo " << node->val << ": tempo di scoperta=" << node-
>discovery_time << ", tempo di completamento=" << node->finish_time <<
endl;
    }

    return 0;
}
```

Esempio file input:

```
5 7
1 2 3
1 3 5
2 3 2
2 4 1
3 4 2
3 5 7
4 5 4
```

GRAFO CON DIJKSTRA, BELLMAN FORD, KRUSKAL:

```
#include <iostream>
#include <vector>
#include <fstream>
#include <string>
#include <stack>
#include <queue>
#include <algorithm>
#include <map>
#include <limits>


using namespace std;

class Node {
public:
    int id, val, dist, rank, key;
    Node* parent;
    Node(int id, int val) : id(id), val(val),
dist(numeric_limits<int>::max()) {}
};

class Edge {
public:
    int weight;
    Node* source;
    Node* destination;
    Edge(int weight, Node* source, Node* destination) : weight(weight),
source(source), destination(destination) {}
};

struct CompareNode {
  bool operator()(const Node* a, const Node* b) const {
    return a->key > b->key;
  }
```

```
};


class Graph {
public:
    vector<Node*> nodes;
    vector<Edge*> edges;
    vector<int> parent;
    vector<int> distance;
    map<int, vector<pair<int, int>>> adjacencyList; // nuova mappa


    Graph(int n) {
        parent.resize(n);
        for (int i = 0; i < n; i++) {
            parent[i] = i;
        }
        distance.resize(n, numeric_limits<int>::max());
    }

    void addNode(Node* node) {
        nodes.push_back(node);
    }

    void addEdge(Edge* edge) {
        edges.push_back(edge);
        adjacencyList[edge->source->id].push_back({edge->destination->id,
edge->weight}); // nuova riga

    }

    void initializeSingleSource(Node* source) {
        for (auto node : nodes) {
            node->dist = numeric_limits<int>::max();
        }
        source->dist = 0;
        distance[source->id] = 0;
    }

    void relax(Edge* edge) {
        int newDist = edge->source->dist + edge->weight;
        if (newDist < edge->destination->dist) {
            edge->destination->dist = newDist;
            distance[edge->destination->id] = newDist;
            parent[edge->destination->id] = edge->source->id;
        }
    }

    void dijkstra(Node* source) {
        initializeSingleSource(source);
        priority_queue<pair<int, Node*>, vector<pair<int, Node*>>,
greater<pair<int, Node*>>> pq;
            pq.push({ source->dist, source });
```

```
        while (!pq.empty()) {
            Node* curr = pq.top().second;
            pq.pop();
            for (auto edge : edges) {
                if (edge->source == curr) {
                    relax(edge);
                    pq.push({ edge->destination->dist, edge->destination
});
                }
            }
        }
    }

    string bellman_ford(Node* source){
     initializeSingleSource(source);
     int i=0;
     while(i<nodes.size()){

     for(auto edge : edges){
           relax(edge);
           }
           for(auto edge : edges){
           if(edge->destination->dist > edge->source->dist + edge-
>weight){
                 return "Trovato ciclo negativo \n";
                 }
           }
           i++;
           }
     return "Non trovato ciclo negativo \n";
     }

     void make_set(Node* x){
           x->parent = x;
           x->rank = 0;
     }

     Node* findset(Node* x){
           if (x != x->parent){
                 x->parent = findset(x->parent);
           }
     return x->parent;
     }

     void link(Node* x, Node* y){
           if(x->rank>y->rank){
                 y->parent = x;
           }else{
                 x->parent = y;
           }
           if(x->rank == y->rank){
                 y->rank++;
           }
```

```
    }

    void union_set(Node* x, Node* y){
        link(findset(x), findset(y));
    }

    vector <Edge*> Kruskal(){
        vector<Edge*> A;
        for(auto node:nodes){
            make_set(node);
        }
        sort(edges.begin(), edges.end(),[](Edge* a, Edge* b){
         return a->weight<b->weight;
      });

        for(auto edge: edges){
            if( findset(edge->source) != findset(edge-
>destination)){
                A.push_back(edge);
                union_set(edge->source, edge->destination);
            }
        }
    return A;
    }

    void ShowAdjacentList(Node* u) {
   cout << "Lista di adiacenza del nodo " << u->id << ":\n";
   for (auto it = adjacencyList[u->id].begin(); it != adjacencyList[u-
>id].end(); it++) {
       cout << "  " << u->id << " -> " << it->first << " (peso " << it-
>second << ")\n";
   }
}

};

int main() {

    bool pesiNegativi;

    ifstream file("input.txt");
    if(!file.is_open()){
        cout<<"errore";
        return 1;
    }
    int numNodes, numEdges;
    file>>numNodes>>numEdges;

    Graph graph(numNodes);
    // Create nodes
    for(int i=0; i<numNodes; i++){
        int id, val;
        file>>id>>val;
```

```
            Node* node = new Node(id, val);
            graph.addNode(node);
        }


    // Create edges

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

            int sourceId, destinationId, weight;
            file>>sourceId>>destinationId>>weight;

            Node* sourceNode = graph.nodes[sourceId];
            Node* destinationNode = graph.nodes[destinationId];

            if(weight<0){
                  pesiNegativi = true;
            }

            Edge* edge = new Edge(weight, sourceNode, destinationNode);
            graph.addEdge(edge);
        }

        file.close();
        Node* source = graph.nodes[0];

        //creiamo file output
        ofstream file2("output.txt");
        if(!file2.is_open()){
    cout << "Errore nella creazione del file di output." << endl;
    return 1;
        }


        //Run algoritmo di bellman ford
        string result;
        result = graph.bellman_ford(source);
        file2<<result;

        file2<<endl;
        //Run Dijkstra
        //DISCLAIMER: DIJKSTRA NON FUNZIONERA' SE NEL FILE VI SONO INPUT
NEGATIVI

        if (pesiNegativi){
              file2<<"Dijkstra non puo' essere effettuato in quanto vi sono
pesi negativi"<<endl;
        }
        else{
              graph.dijkstra(source);
        for (auto node : graph.nodes) {
```

```
        file2 << "Node " << node->id << ": distance=" << node->dist << ", 
parent=" << graph.parent[node->id] << endl;
            }
           }
     file2<<endl;

     vector<Edge*> mst_K;
     vector<Edge*> mst_P;

     file2<<"MST KRUSKAL:"<<endl;
     mst_K = graph.Kruskal();
     for(Edge* edge: mst_K){

        int sourceID = edge->source->id;
        int destinationID = edge->destination->id;
        int weight = edge->weight;

        file2<<"source: "<<sourceID<<" destination: "<<destinationID<<"
weight: "<<weight<<endl;

    }

    file2<<endl;

     file2.close();

     cout<<"creato file output"<<endl;
    return 0;
}
```

Esempio file input:

```
5 7
0 0
1 2
2 4
3 6
4 8
0 1 5
0 2 2
1 2 3
1 3 4
2 3 2
2 4 1
3 4 3
```

GRAFO CON BFS RICORSIVA:

```
#include <iostream>
#include <vector>
#include <queue>

using namespace std;

const int WHITE = 0;
```

```
const int GREY = 1;
const int BLACK = 2;

const int INF = 999999;

template <typename T>
class Node{
	public:
		T val;
		int color = WHITE;
		T dist;
		Node<T>* parent;
		vector<Node<T>*> adj;

		Node(T val): val(val){}


};


template <typename T>
class Edge{
	public:
		Node<T>* source;
		Node<T>* destination;

		Edge(Node<T>* source, Node<T>* destination): source(source),
destination(destination){
		}
};

template <typename T>
class Graph{
	public:
		vector<Node<T>*> nodes;
		vector<Edge<T>*> edges;

		Graph(){}

		void addNode(Node<T>* node){
			nodes.push_back(node);
		}

		void addEdge(Node<T>* source, Node<T>* destination){
			Edge<T>* edge = new Edge<T>(source, destination);

			edges.push_back(edge);
			source->adj.push_back(destination);
		}

		void BFS(Node<T>* s){
			for(auto u : nodes){
				if(u!=s){
```

```
                    u->color = WHITE;
                    u->dist = INF;
                    u->parent = nullptr;
                }
            }
            s->color = WHITE;
            s->dist = 0;
            s->parent =nullptr;
            queue<Node<T>*> Q;
            Q.push(s);
            while(!Q.empty()){
                Node<T>* u = Q.front();
                Q.pop();
                for(auto v : u->adj){
                    if(v->color == WHITE){
                        v->color = GREY;
                        v->dist = u->dist + 1;
                        v->parent = u;
                        Q.push(v);
                    }
                }
                u->color = BLACK;
            }
        }

        void BFS_recursive(Node<T>* u){
            u->color = GREY;
            for(auto v : u->adj){
                if(v->color == WHITE ){
                    v->parent = u;
                    v->dist = u->dist + 1;
                    BFS_recursive(v);
                }
            }
            u->color = BLACK;
        }
};

int main(){
    Graph<int> G;

    // Creazione nodi
    Node<int>* A = new Node<int>(0);
    Node<int>* B = new Node<int>(1);
    Node<int>* C = new Node<int>(4);
    Node<int>* D = new Node<int>(5);
    Node<int>* E = new Node<int>(6);
    Node<int>* F = new Node<int>(3);

    // Aggiunta nodi al grafo
    G.addNode(A);
    G.addNode(B);
    G.addNode(C);
```

```
    G.addNode(D);
    G.addNode(E);
    G.addNode(F);

    // Aggiunta archi al grafo
    G.addEdge(A, B);
    G.addEdge(A, C);
    G.addEdge(B, D);
    G.addEdge(C, D);
    G.addEdge(C, E);
    G.addEdge(D, E);
    G.addEdge(D, F);

    // Esecuzione BFS iterativa a partire da A
    cout << "BFS iterativa a partire da A: ";
    G.BFS(A);
    for(auto u : G.nodes){
        cout << u->val << "(" << u->dist << ") ";
    }
    cout << endl;

    // Reset colori e distanze dei nodi
    for(auto u : G.nodes){
        u->color = WHITE;
        u->dist = INF;
    }

    // Esecuzione BFS ricorsiva a partire da A
    cout << "BFS ricorsiva a partire da A: ";
    A->dist = 0;
    G.BFS_recursive(A);
    for(auto u : G.nodes){
        cout << u->val << "(" << u->dist << ") ";
    }
    cout << endl;

return 0;
}
```

GRAFO CON PRIM:

```
#include <iostream>
#include <vector>
#include <queue>

using namespace std;

// Definizione della classe Grafo
class Grafo {
private:
    int V; // Numero di vertici del grafo
    vector<vector<pair<int, int>>> adj; // Lista di adiacenza dei vertici
```

```
public:
    // Costruttore della classe Grafo
    Grafo(int V) {
        this->V = V;
        adj.resize(V);
    }

    // Funzione per aggiungere un arco al grafo
    void aggiungiArco(int u, int v, int peso) {
        adj[u].push_back(make_pair(v, peso));
        adj[v].push_back(make_pair(u, peso));
    }

    // Funzione per l'algoritmo di Prim
    int prim() {
        // Creazione della coda di priorità per gli archi del grafo
        priority_queue<pair<int, int>, vector<pair<int, int>>,
greater<pair<int, int>>> pq;

        int src = 0; // Sorgente
        vector<int> key(V, INT_MAX); // Inizializzazione delle chiavi
        vector<bool> inMST(V, false); // Inizializzazione del MST
        vector<int> parent(V, -1); // Inizializzazione dei genitori

        // Inserimento del primo vertice nella coda di priorità
        pq.push(make_pair(0, src));
        key[src] = 0;

        while (!pq.empty()) {
            int u = pq.top().second;
            pq.pop();

            inMST[u] = true;

            // Scorrimento dei vertici adiacenti al vertice corrente
            for (auto i = adj[u].begin(); i != adj[u].end(); ++i) {
                int v = i->first;
                int peso = i->second;

                // Se v non è presente nel MST e il peso dell'arco u-v è
minore della chiave di v
                if (inMST[v] == false && peso < key[v]) {
                    key[v] = peso;
                    pq.push(make_pair(key[v], v));
                    parent[v] = u;
                }
            }
        }

        int costoTotale = 0;

        // Stampa degli archi del MST e calcolo del costo totale
        for (int i = 1; i < V; ++i) {
```

```
            cout << parent[i] << " - " << i << "    " << key[i] << endl;
            costoTotale += key[i];
        }

        return costoTotale;
    }
};

int main() {
    int V = 5; // Numero di vertici

    // Creazione del grafo
    Grafo g(V);

    // Aggiunta degli archi del grafo
    g.aggiungiArco(0, 1, 2);
    g.aggiungiArco(0, 3, 6);
    g.aggiungiArco(1, 2, 3);
    g.aggiungiArco(1, 3, 8);
    g.aggiungiArco(1, 4, 5);
    g.aggiungiArco(2, 4, 7);
    g.aggiungiArco(3, 4, 9);

    // Applicazione dell'algoritmo di Prim e stampa del costo totale
    cout << "Costo totale del mst "<<endl;
     cout<<g.prim()<<endl;
    return 0;
}
```

GRAFO CON COMPONENTI FORTEMENTE CONNESSE:

```
#include <iostream>
#include <vector>
#include <stack>

using namespace std;

// Classe che rappresenta un grafo diretto mediante liste di adiacenza
class Graph {
    int V;
    vector<int> *adj;

public:
    Graph(int V);
    void addEdge(int u, int v);
    void printSCCs();
    void DFSUtil(int v, bool visited[], stack<int> &stack);
    void DFSUtil2(int v, bool visited[]);
    Graph transpose();
};

Graph::Graph(int V) {
    this->V = V;
```

```
    adj = new vector<int>[V];
}

void Graph::addEdge(int u, int v) {
    adj[u].push_back(v);
}

void Graph::DFSUtil(int v, bool visited[], stack<int> &stack) {
    visited[v] = true;

    for (auto i = adj[v].begin(); i != adj[v].end(); i++) {
        if (!visited[*i]) {
            DFSUtil(*i, visited, stack);
        }
    }

    stack.push(v);
}

Graph Graph::transpose() {
    Graph g(V);

    for (int v = 0; v < V; v++) {
        for (auto i = adj[v].begin(); i != adj[v].end(); i++) {
            g.adj[*i].push_back(v);
        }
    }

    return g;
}

void Graph::DFSUtil2(int v, bool visited[]) {
    visited[v] = true;

    cout << v << " ";

    for (auto i = adj[v].begin(); i != adj[v].end(); i++) {
        if (!visited[*i]) {
            DFSUtil2(*i, visited);
        }
    }
}

void Graph::printSCCs() {
    stack<int> stack;

    bool *visited = new bool[V];
    for (int i = 0; i < V; i++) {
        visited[i] = false;
    }

    for (int i = 0; i < V; i++) {
        if (!visited[i]) {
```

```
            DFSUtil(i, visited, stack);
        }
    }

    Graph g = transpose();

    for (int i = 0; i < V; i++) {
        visited[i] = false;
    }

    while (!stack.empty()) {
        int v = stack.top();
        stack.pop();

        if (!visited[v]) {
            g.DFSUtil2(v, visited);
            cout << endl;
        }
    }
}

int main() {
    Graph g(5);

    g.addEdge(1, 0);
    g.addEdge(0, 2);
    g.addEdge(2, 1);
    g.addEdge(0, 3);
    g.addEdge(3, 4);

    cout << "Componenti fortemente connesse:" << endl;

    g.printSCCs();

    return 0;
}
```

HASHTABLE:

```
//hashtable

#include <iostream>
#include <vector>
#include <fstream>

using namespace std;


template <typename T>
class item{
     public:
          T key;
          T value;
```

```
          item(T key, T value): key(key), value(value ) {}

};

template <typename T>
class hashtable{
     public:
          vector<item<T>*> table;
          int m;

          hashtable(int m):m(m){}

          int hash(T key, int i){ //ispezione lineare
               return(key+i)%m;
          }

          int hash_quadratica(int key, int i) {
         // hash con ispezione quadratica
            return (key + i + i*i) % m;
          }

         int doppio_hashing(int key, int i){
         // hash con doppio hashing
         int hash1 = key % m;
         int hash2 = 1 + (key % (m - 1));
         return (hash1 + i * hash2) % m;
          }

          void insert(item<T>* item){
              int i = 0;
              while(i != m){
                  int j = hash(item->key, i);
                  if(table.size() <= j){
                      table.resize(j+1, nullptr);
                  }
                  if(table[j]==nullptr){
                      table[j] = item;
                      return;
                  }else{
                      i++;
                  }
              }
              cout<<"errore overflow";
          }


          item<T>* search(T key){
                    int i = 0;
                    int j = hash(key, 0);
                    while(table[j]!=nullptr && i!= m){
                         if(table[j]->key==key){
                              return table[j];
                         }
```

```
                        i++;
                        j = hash(key, i);
                    }
                    return nullptr;
                }

};

int main(){

    ifstream in("hash.txt");
    //controllo apertura
    int hashsize;
    in>>hashsize;
    hashtable<int> H(hashsize);

    //riempimento hashtable
    for(int i=0; i<hashsize; i++){
        int key, value;
        in>>key>>value;
        item<int>* itemx = new item<int>(key, value);
        H.insert(itemx);
    }

    in.close();

    item<int>* result = H.search(2);

    ofstream out("hashoutput.txt");

    if(result != nullptr){
        out<<"element found, its key: "<<result->key<<", its value: 
"<<result->value<<endl;
    }else if(result == nullptr){
        out<<"element not found"<<endl;
    }
    out.close();

    cout<<"creato file output"<<endl;
    return 0;
}
```

Esempio file input:

```
10
3 5
2 7
1 5
2 6
6 4
4 9
4 7
5 9
0 9
8 7
```