Palestra di algoritmi

21/11/2023 - gruppo B

Alcune strutture dati utili

Pair

Struttura dati con due campi: first e second.

```
pair<int, char> coppia;
coppia.first = 2;
coppia.second = ‘a’;
cout << coppia.first << “ “ << coppia.second << “\n”;

vector<pair<int, char>> v;
v.push_back({2,’a’});
v.push_back(make_pair(3, ‘b’));
for(auto elem: v) {
        cout << elem.first << “ “ << elem.second << “\n”;
}
```

Map

Struttura dati che memorizza coppie chiave-valore.

La chiave è come l’indice di un array, il valore è il contenuto in quell’indice.

La chiave è unica.

map<int, char> m;

m[1] = ‘a’;

cout << m[1];

Soluzioni esercizi scorsa volta

Halloween candies

Idea: ordiniamo i team per punteggio crescente e aumentiamo di 1 le caramelle ogni volta

```
sort(vet, vet+N);
long long int caramelleDaDare = 1;
long long int caramelleTotali = 1;
for(int i=1;i<N;i++){
	if(vet[i-1]!=vet[i]){
		caramelleDaDare++;
	}
	caramelleTotali+=caramelleDaDare;
}
out << caramelleTotali;
```

Quadrati

Idea: al posto di controllare se un numero è un quadrato perfetto, partiamo dalla base e eleviamo finché siamo nel range

```
in >> a >> b;
x = sqrt(b);    //troviamo la base di inizio
y = sqrt(a);    //troviamo la base di fine
if (y*y!=a)     //se estremo iniziale escluso
      n = x-y;
else            //se estremo iniziale incluso
      n=x-y+1;
out << n;
```

3x2

Idea: ordino per prezzo e divido 3 a 3

```
int vet[MAX];
int n;
in >> n;

for(int i=0;i<n;i++) { in >> vet[i]; }

sort(vet, vet+n);
int cont = 0; int somma = 0;

for(int i=n-1;i>=0;i--) {       //parto dal fondo
        if(cont < 2) {          //ogni 3, il meno costoso è gratis
                somma += vet[i];
                cont++;
        } else { cont = 0; }
}

out << somma;
```

Gasoline

Idea: ad ogni distributore compro abbastanza benzina da arrivare al prossimo distributore meno caro

prezzi = vettore dei prezzi, gasoline = vettore di quanta benzina serve per il distributore successivo

```
int attuale = prezzi[0];            //parto dal prezzo iniziale ovviamente
long long int ris = attuale*gasoline[0];

for(int i=1;i<n;i++) {
        if(p[i]<attuale)            //aggiorno il prezzo quando mi conviene
                attuale=p[i];
        ris+=attuale*g[i];
}

out << ris;
```

Disuguaglianze

Idea: utilizzo il bubble sort (continuo a fare scambi fin quando non sono tutti in ordine)

```
while(!ok) {
        ok=true;
        for(int i=0;i<n-1;i++) {
                if(vet2[i]=='<') {                  //se la disuguaglianza è minore
                        if(vet[i]>vet[i+1]) {       //e il primo è maggiore del secondo
                                int temp=vet[i];
                                vet[i]=vet[i+1];
                                vet[i+1]=temp;
                                ok=false;
                        }
                } else {
                        if(vet[i]<vet[i+1]) {
                                int temp=vet[i];
                                vet[i]=vet[i+1];
                                vet[i+1]=temp;
                                ok=false;
}}}}
```

Turni

Idea: partiamo dal giorno 0 e prendiamo ogni volta il turno con fine “più in là”

```
map <int,int> m;
int n,k;

in >> k >> n;

for(int i=0;i<n;i++) {
        int da,a;
        in >> da >> a;
        m[da]=max(m[da],a); //se ci sono due turni che iniziano nello stesso giorno, tengo quello più lungo
}

int att=m[0];        //devo partire dal giorno 0
int cont=1;          //un turno sarà fatto per forza
```

```
while(att!=k-1) {    //fin quando non ho coperto tutto il periodo
        int prox=att;
        for(auto &coppia : m) {
                if(coppia.first<=att+1) {
                        if(coppia.second>prox) {
                                prox=coppia.second;
                        }
                } else
                        break;
        }
        att=prox;
        cont++;
}

out << cont;
```

Quadri

Idea: utilizzo una “finestra” di quadri che aumenta a destra e diminuisce a sinistra

Parto utilizzando come “finestra” solo il primo quadro

Se posso aggiungere il quadro a destra senza sforare, aumento la finestra

Quando sforo tolgo quadri a sinistra fin quando non torno nel valore previsto

La finestra più grande che otteniamo è il nostro risultato

```
int b = 0;        //Dimensione della finestra
int i = 0;        //Indice per scorrere l'array
long long somma = 0; //Somma dei valori nella finestra

//Scorrimento iniziale per formare la prima finestra (fin quando non sforerei)
while (((somma + V[i]) <= m) && (i < n)) {
        somma += (long long)V[i];
        ++i;
        ++b;
}
```

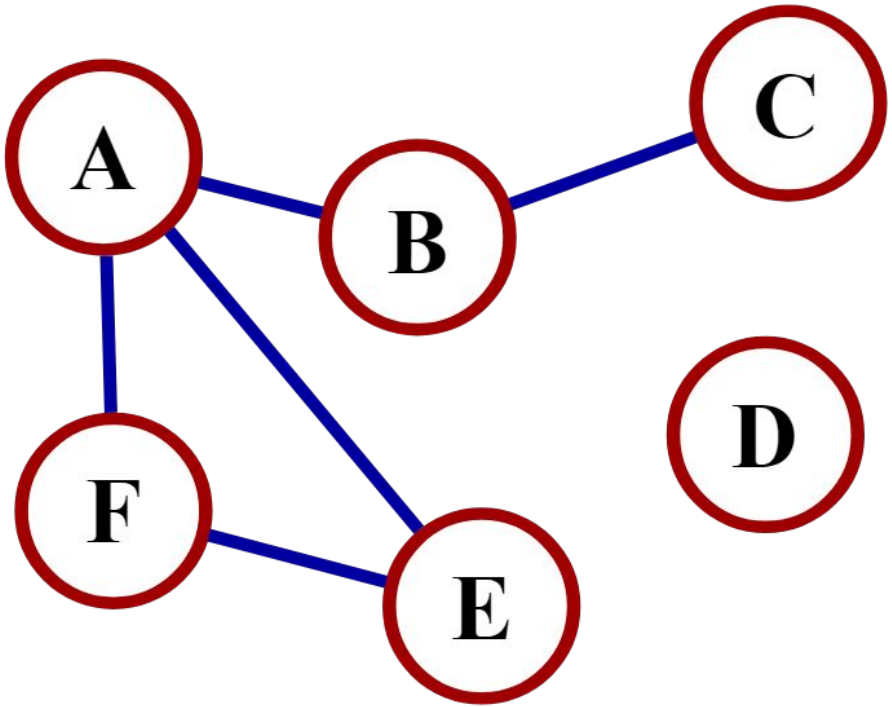
```
//Scorrimento attraverso gli elementi rimanenti dell'array
for (; i < n; ++i) {
        //Verifica se il valore corrente supera il massimale M
        if (V[i] > m)
            return 0;
        //Aggiorna la somma della finestra
        somma += (long long)V[i] - V[i - b];

        //Riduci la dimensione della finestra se la somma supera il massimale M
        while (somma > m) {
            --b;
            somma -= (long long)V[i - b];
        }
    }

    return b;
```

Grafi

Cos’è un grafo?

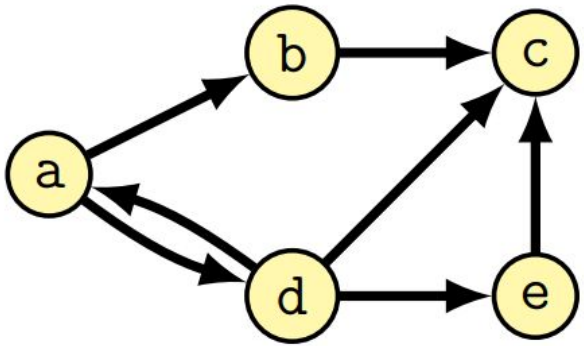


Insieme di nodi uniti da archi.

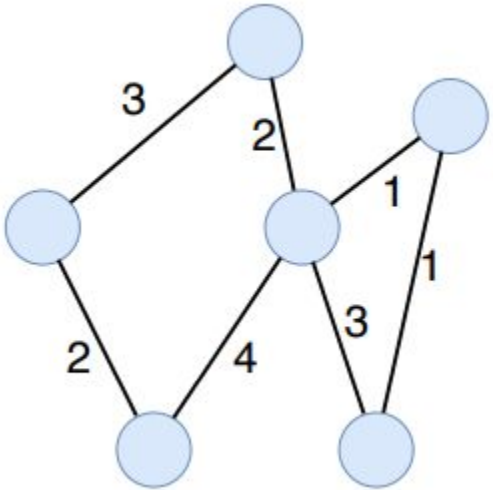
Nodi = {A, B, C, D, E, F}

Archi = {{A,B}, {A,E}, {A,F}, {F,E}, {B,C}}

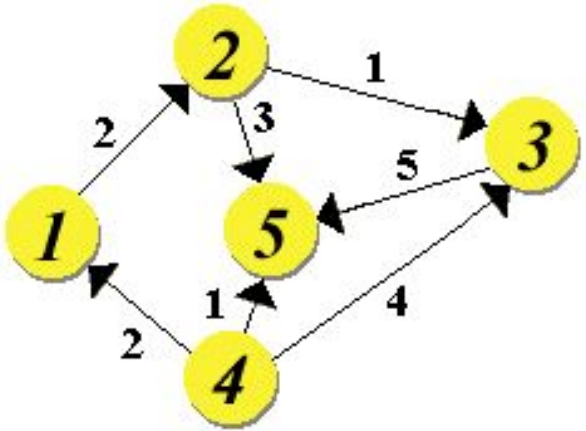
Grafo orientato



Grafo pesato

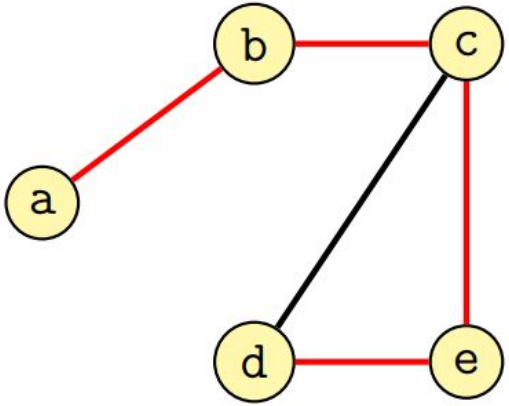


Grafo orientato e pesato

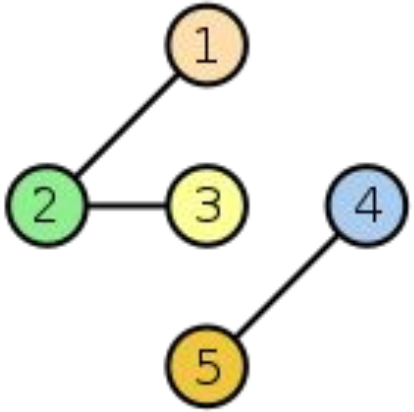


Problemi sui grafi

- cercare se esiste un cammino da un nodo a un altro
- trovare il cammino più breve da un nodo all'altro
- trovare le componenti connesse
- trovare cicli

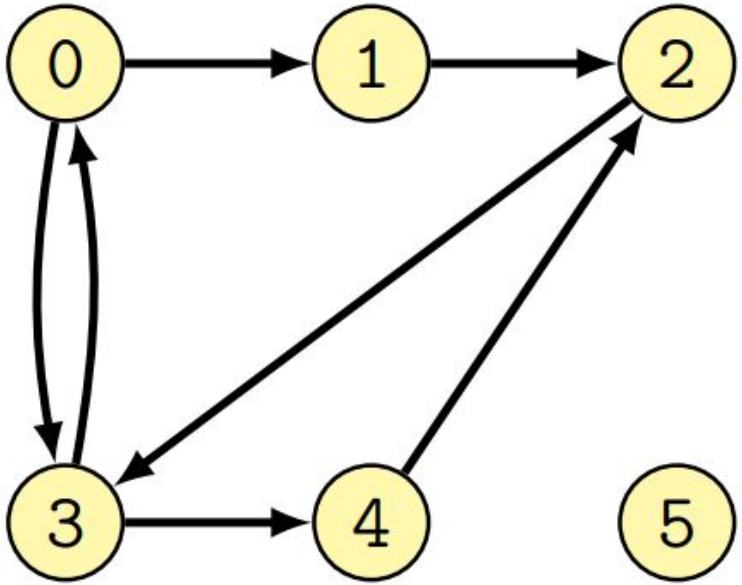


Esiste un cammino dal nodo **a** al nodo **e**



{1, 2, 3} è una componente connessa, {4, 5} è un'altra

Implementare un grafo: matrici di adiacenza

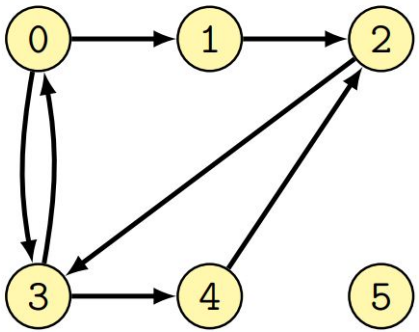


	0	**1**	**2**	**3**	**4**	**5**
0	0	1	0	1	0	0
1	0	0	1	0	0	0
2	0	0	0	1	0	0
3	1	0	0	0	1	0
4	0	0	1	0	0	0
5	0	0	0	0	0	0

Esempio pratico

Solitamente il problema vi darà in input un grafo in questo modo:

- prima riga: numero di nodi (N) e di archi (M)
- successive M righe: due numeri, che rappresentano due nodi. Significa che esiste un arco dal primo nodo al secondo. Nel caso di arco pesato c'è un terzo numero: il peso




```
6 7

0 1
0 3
1 2
2 3
3 0
3 4
4 2
```

Matrice di adiacenza in codice

```
in >> N >> M;
int grafo[MAX][MAX]; //va inizializzata a tutti zeri
for(int i=0;i<M;i++) {
    in >> da >> a;
    grafo[da][a] = 1; //se il grafo è pesato, si mette il peso
    grafo[a][da] = 1; //se il grafo non è orientato, l'arco è bidirezionale
}
```

Se abbiamo pochi archi e tanti nodi, una matrice di adiacenza spreca un sacco di spazio!!!!

Vector

Un vector è un array che può cambiare dimensione dinamicamente. Non dobbiamo decidere in anticipo quanto è grande.

```
vector<int> v;          //dichiaro un vector di interi chiamato v

v.push_back(3);         //aggiungo un 3 al vector

sort(v.begin(), v.end()); //ordino il vector

cout << v.at(0);        //stampo l'elemento in posizione 0

cout << v[0];           //idem
```

Scorrere tutti gli elementi di un vector

```
for(auto elem: v) {
        elem++;
        cout << elem;
}

for(int i=0;i<n;i++){
        elem = vet[i];
        elem++;
}
```

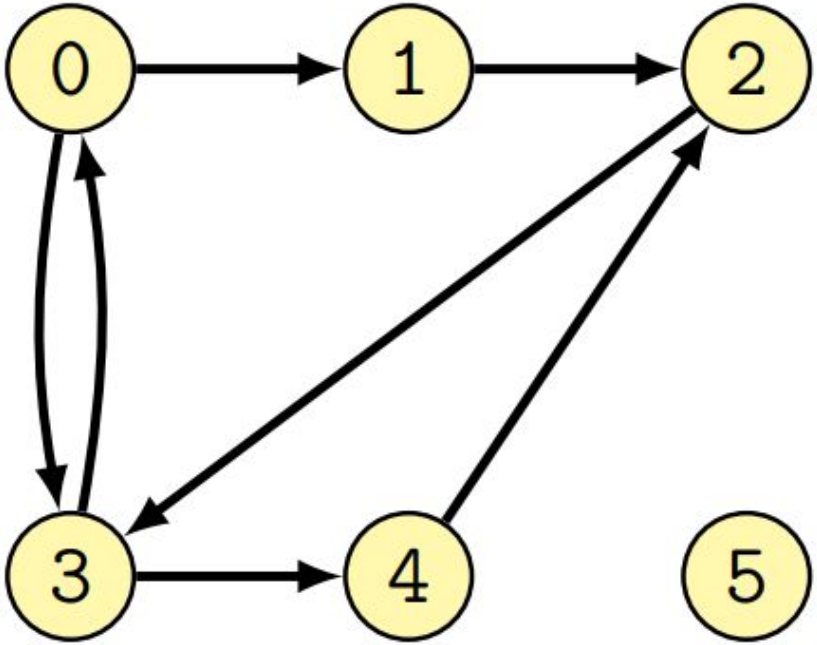
Le modifiche non vengono propagate al vector (elem è una copia).

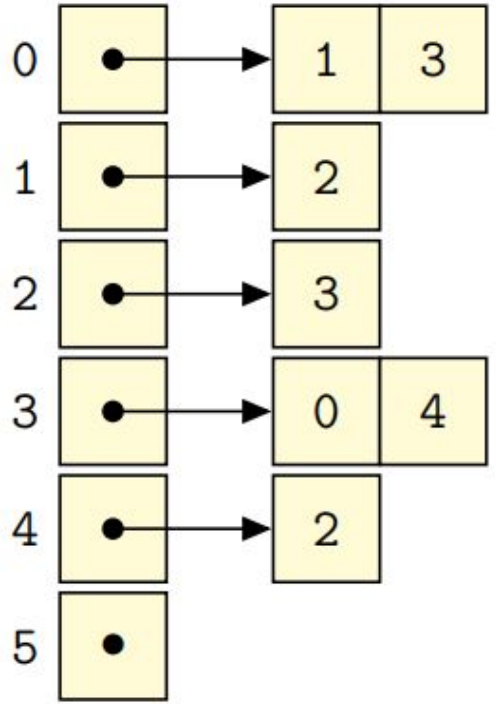
```
for(auto &elem: v) {
        elem++;
        cout << elem;
}

for(int i=0;i<n;i++){
        vet[i]++;
}
```

Le modifiche vengono propagate al vector (elem è l'elemento dell'array).

Implementare un grafo: vettori di adiacenza





Vettori di adiacenza in codice (grafo non pesato)

```
vector<vector<int>> grafo;
int N, M;

in >> N >> M;

grafo.assign(N, vector<int>());

for(int i=0;i<M;i++) {
        int da, a;
        in >> da >> a;
        grafo[da].push_back(a);
        grafo[a].push_back(da);        //nel caso di grafo non orientato
}
```

Vettori di adiacenza in codice (grafo pesato)

```
vector<vector<pair<int, int>>> grafo;    //NB: pair = struct con campi first e second

in >> N >> M;

grafo.assign(N, vector<pair<int,int>>());

for(int i=0;i<M;i++) {
        int da, a, peso;
        in >> da >> a >> peso;
        grafo[da].push_back({a, peso});
        grafo[a].push_back({da, peso});    //nel caso di grafo non orientato
}
```

Visitare un grafo

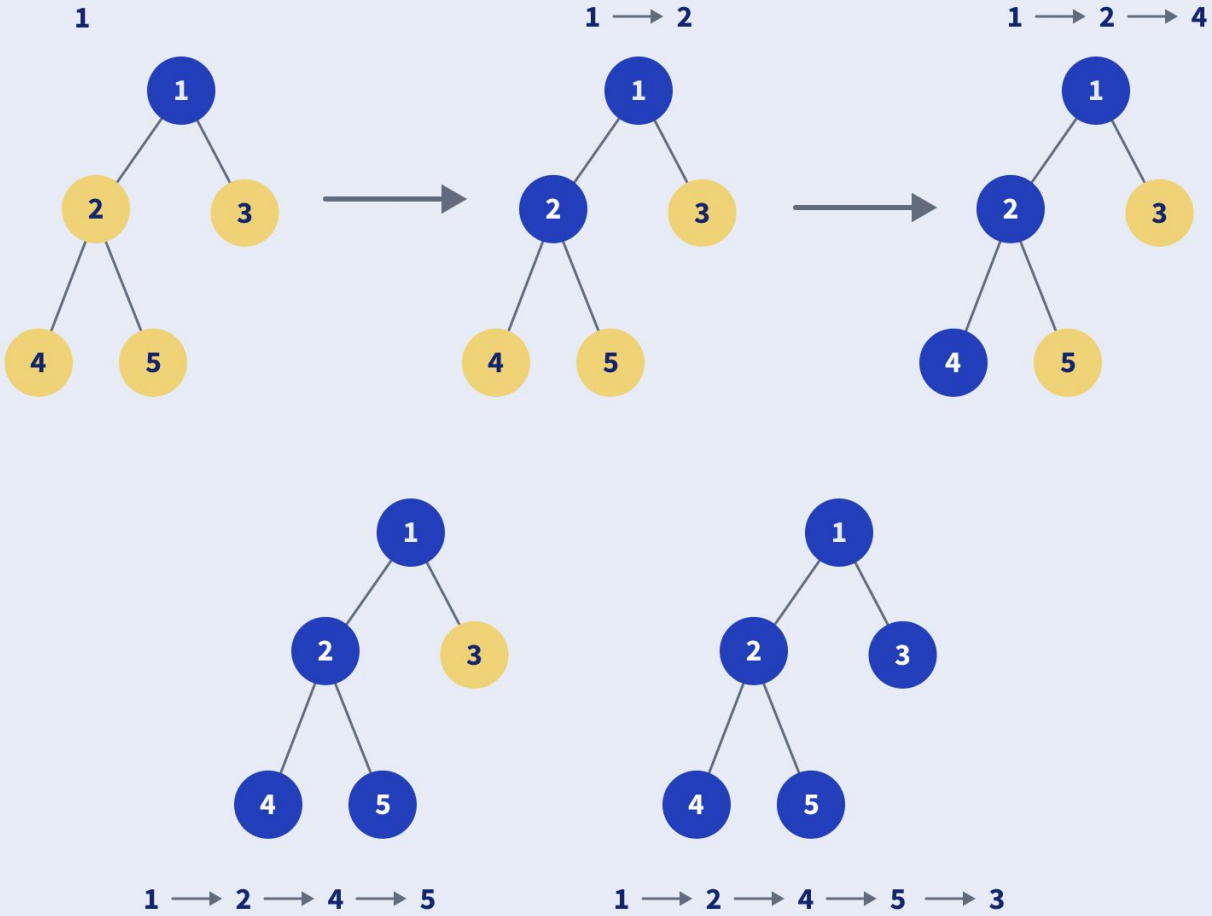
Visita di un grafo: DFS

DFS = Depth First Search (visita in profondità)

Un metodo per visitare un grafo

Si tratta di una visita ricorsiva: per ogni nodo adiacente, si visita ricorsivamente tale nodo, visitando ricorsivamenti i suoi nodi adiacenti, etc.

DFS



DFS: pseudocodice

```
void DFS(vector<vector<int>> g, int nodo, bool[] visitato) {
	visitato[nodo] = true;
	for(int i=0;i<g[nodo].size();i++){
		int vicino = g[nodo][i];
		if(!visitato[vicino])
			DFS(g, vicino, visitato);
	}
}
```

Esercizi

Sunnydale (https://training.olinfo.it/#/task/sunny/statement)

Grasshopper(https://training.olinfo.it/#/task/ois_grasshopper/statement)