

## Introduzione

Ogni programma deve avere la funzione main:

```
int main() {  
    return 0;  
}
```

→ return serve a comunicare col SO (noi restituiremo sempre 0)  
→ ogni funzione deve eseguire un return

Compilatore C++ standard C++11

L> riga di comando

PER ESEGUIRE → g++-std=c++11 -o programma1 programma1.cpp

Input e Output → ci serve la libreria <iostream>

L> le librerie si includono con #include (es. #include <iostream>)

## Esempio

```
#include <iostream>  
int main()  
{ std::cout << "Inserisci 2 numeri:" << std::endl; >> v1 <<  
    OUTPUT  
    int v1 = 0, v2 = 0;  
    PER LEGGERE (INPUT)  
    std::cin << v1 >> v2;  
    // legge istruzione e controlla stream input, apprezzza qualcosa che puo' riempire con int sempre v1, fa lo stesso con v2, se effettua altri dati non importa.  
    SOMMA DI V1 E V2  
    std::cout << "La somma di " << v1 << " e " << v2 << " e " << v1 + v2 << std::endl;  
    return 0; >> OBBLIGATORIO
```

Per eliminare std dopo #include inseriamo using namespace ::std; SOLO SE NON USERAI ALTRI NAMESPACE X

## Commenti

→ II UNA SOLA RIGA  
→ /\* APRE E CHIUSA COMMENTO CON DUE \*/ ] NON SI POSSONO INNESTARE

## de variabili

C++ → fortemente tipizzato <tipo> <nome>

Tipi di dato

bool → 0 o 1  
char → 8 bit  
short → intero corto 16  
int → 32 bit  
long → intero lungo 64  
float → virgola ; precisione singola  
double → precisione doppia

Inizializzazione e definizione

int i1; → devo scrivere prima di usarla  
int i2 = 0;  
bool b = false;

## de liste

int x[10]; //inizializzazione lista  
int x[0]; //esempio  
int x[0]; //inizializzazione oggetto  
com tipi base non hanno senso

i nomi var sono case-sensitive

Cosa succede se metto un valore sbagliato in un tipo dato?

bool b = 42; → in C++ 0 = false, tutti gli altri valori ④  
L> altre b = 1 → true  
int i = b → i = 1

int i = 3,14 || i = 3  
float pi = i => pi = 3

## La visibilità delle variabili (scope)

### • Scope globale (scopagnato)

```
ES.  
#include <iostream>  
using namespace std;  
int reuse=42;  
int main() {  
    int unique=0;  
    cout<<reuse<<" " <<unique; // 42 0  
    int reuse=0; // copre visibilità di reuse globale  
    cout<<reuse<<" " <<unique; // 0 0  
    cout<<reuse.... // 42  
    return 0;  
}
```

### de costanti

const int j=42; → non può essere cambiato

### I caratteri speciali

\n → vai a capo  
\t → tabulazione  
\r → per stampare \r

### Il tipo speciale auto

auto i=0; // il compilatore effettua al  
auto item= val1+val2; // l'elaborazione il tipo più comune.

Gli operatori hanno significati diversi in base al tipo

con le classi si possono definire degli operatori

### Operatori

int i=0, j=1;      j++ ] incrementa di 1  
i=j++;

j++ → prima veluto; poi incremento i=1 j=2

++j → prima incremento; poi veluto. i=3 j=3

cout<<++i <<i++ <<.....

USA versione prefissa (++i)

### Operatori booleani

== = uguale      != = VACUUM AND HA CONTINUA AVVISTATE  
!= = DIVERSO      == SE FACCIO SISTEMA SUBITO  
! = OR VACUA E NON SI TERRA  
|| =

## Tipi composti

1) RIFERIMENTI &

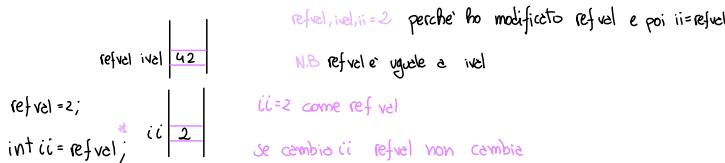
2) PUNTATORI \*

### Riferimenti

int &ref val = 42; variabile intre

int &ref val = i; una variabile riferimento deve essere sempre inizializzata.

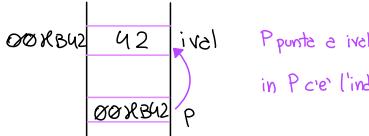
E' come un alias che si riferisce ad un'altra variabile



### Puntatori:

int \*p1; Indirizzo e variabile di tipo int

int \*p1, p2; double d1, \*d2;  
double d1, \*d2;  
int val=42;



int \*p=&val voglio che p punti a val con & come operatore per ottenere l'indirizzo di una variabile  
cout << p; //stampa l'indirizzo di val contenuto in p 0018B42  
cout << \*p; //stampa il contenuto di val) dereferenziando p. 42

DEREFERENZIARE  
Accesso al contenuto delle celle di memoria a cui si punta.  
cout << \*p;

STATO PUNTATORE -> ad una celle immediatamente successive; nullo; ad una celle di memoria.

INIZIALIZZARE UN PUNTATORE A NULLA

1) int \*p1=nullptr;

2) int \*p2=&0;

3) int \*p3=NULL //richiede la libreria <cstdlib>

4) PUNTATORE A CARATTERI char \*c;  posso leggere le celle successive e gestire le stringhe con questi

### Tipi di dato definiti dall'utente

#include <iostream>

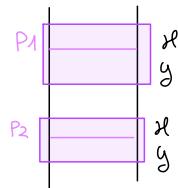
using namespace::std;

#include <cmath>

**STRUCT** per definire una struttura dati.

Struct Punto

```
{ int x;  
    int y; }  
  
int main()  
{ Punto p1, p2;  
  
    double Distanza;  
  
    Cin >> p1.x >> p1.y  
  
    Cin >> p2.x >> p2.y  
  
    Distanza = sqrt((p1.x - p2.x) * (p1.x - p2.x) + (p1.y - p2.y) * (p1.y - p2.y));  
  
    cout << Distanza;  
  
    return 0; }
```



## PUNTATORI A STRUCT

```
int main(){  
  
    punto p1;  
  
    p1.x=3, p1.y=4;  
  
    punto * pp1 = &p1 // pp1 = contiene indirizzo di p1  
  
    cout << pp1.x X Bisogna DIFERENZIARE  
  
    cout << *pp1.x X Bisogna definire ordine di accesso  
  
    cout << (*pp1).x ✓  
  
    cout << pp1->x ✓ // -> permette di DEREFERENZIARE e punta alla proprietà da accedere (ora x)
```

## ISTRUZIONI COMPOSTE

```
while while (condizione) {blocco}  
do while do {blocco} while (condizione)  
for for (condizione finita) {blocco} (init;check;update)  
if if (condizione) {blocco} else {blocco}
```

Una variabile definita in un blocco ha visibilità solo nel blocco. (se dichiaro in while le vedo solo lì).

## LEGGERE M NUMERI E SOMMARLI

- 1) int main()

```
{ int sum=0, value=0;  
  
    while (cin >> value) // ci restituisce true o false se e' riuscito a leggere qualcosa CTRL-Z STOPPA  
        sum+= value;
```

## DETERMINARE IL MASSIMO IN UNA SEQUENZA LETTA DA INPUT

```
int main()
{
    int curvel, maxval;
    if (cin >> curvel)
    {
        maxval = curvel;
        while (cin >> curvel)
            if (curvel > maxval)
                maxval = curvel;
        cout << maxval;
    }
    else
        cout << "Nessun dato inserito" << endl;
    return 0;
}
```

## ESERCIZIO STAMPARE LA TABELLINA DEL 10 BEN FORNATTATA (VEDI PG)

```
#include <iostream>
using namespace
```

de funzioni:

Tipo nome (parametri)

```
{ blocco;
  return ...; // Restituisce il tipo definito su NELLA DICHIARAZIONE
}
```

ESEMPIO FATTORIALE DI UN NUMERO

```
int fact (int val) PARAMETRO FORMALE
{
  int ret=1; // VISIBILE IN FACT
  while (val>=1)
    ret *= val--;
  return ret;
}
```

## COME SI USANO LE FUNZIONI

```
int main ()
{
  int j=5;
  cout << j << "!" << fact(j) << endl; // PARAMETRO ATTUALE (EFFETTIVO)
  return 0;
}
```

Quando viene richiamata una funzione in memoria viene generato un RECORD di ATTIVAZIONE contenente le variabili utili.



Quando viene richiamata una funzione l'esecuzione si blocca e riprende quando si ha un risultato per la funzione.

LE FUNZIONI VOID non restituiscono niente.

ESEMPIO

```
void StampaStelle (int m)
{
  for (int i=0; i<m; ++i)
    cout << "*";
  cout << endl;
  return;
}
```

L'UNICO PONTE DI COMUNICAZIONE TRA CHIAMANTE E CHIAMATO SONO I PARAMETRI O LE VARIABILI GLOBALE.

Si mungeggia di parametri nelle funzioni

- 1) `int f2() { ... }`
- 2) `int f3 (int v1, int v2){....}`

PROTOTIPO FUNZIONE      inizice

Si mette prima del main.    `int fact(int);`

## Passaggio dei parametri

- 1) PER VALORE prendo il valore delle var attuali e le copio in quelle formate; il parametro formale non cambia MAI. PASSO COPIA.
- 2) PER RIFERIMENTO int fact(int &val) in questo caso uso val come Alias del parametro effettivo. Se modifco il parametro formale (con riferimento) modifco anche quello effettivo. In questo modo la funzione puo' restituire piu' risultati (tutte le var con ref che vengono modificate). Passo indirizzo, NO COPIA!
- 3) PER RIFERIMENTO COSTANTE int fact(const int &val) ERRORE se cerco di modificare val in esecuzione.

## ESEMPIO

void reset (int &); // PROTOTIPO FUNZIONE se definisco dopo la funzione

```
int main()
{
    int j = 42;
    reset(j);
    cout << j;
    return 0;
}
```

NOTA BENE j viene modificato in funzione di i  
perche' i e' un riferimento a j.

```
void reset (int &i)
{
    i = 0;
    return;
}
```

Oggetti di tipo struct e passaggio di parametri // riferimento const utile su oggetti definiti dall'utente

```
struct punto
{
    int x;
    int y;
};

double distanza(const punto &p1, const punto &p2)
{
    return sqrt((p1.x - p2.x) * (p1.x - p2.x) + (p1.y - p2.y) * (p1.y - p2.y));
}

int main()
{
    punto p1, p2;
    double d;

    cin >> p1.x >> p1.y;
    cin >> p2.x >> p2.y;
    d = distanza(p1, p2);
    cout << d;
    return 0;
}
```

Il return

ESEMPIO Funzione che restituisce 1 se N e' primo, 0 se non lo e'.

```
bool Primo(int n)
{
    for (int k = m12; k > 1; k--) // conversione implicita in int anche se double/float.
        if (m1.K == 0) return false; // OGNI RAMO DEVE FARE RETURN.
    else return true;
}
```

```

OPPURE: bool Primo(int n)
{
    bool primo=True; //CONDIZIONE PIÙ DIFFICILE DA VERIFICARE
    for (int k=m/2; k>1 && primo; k--) //CONTINUA FINCHE' K>1 e PRIMO=TRUE
        if (n%k==0)
            primo=false;
    return primo;
}

```

**Overloading di funzioni** Posso avere lo stesso nome di funzione nello stesso programma con diverse firme (num di parametri e tipo);

```

void f(int);           NON C'E' AMBIGUITA' SE HANNO FIRME DIVERSE
void f(int,int);       void f(auto); //CORPO SENZA TENERE CONTO DEL TIPO MA DEVO TENERE CONTO DI TUTTI I CASI.
void f(float);         se ci sono tutti e 3 e' ambiguo; o uso auto o int e float etc.
int f(int);            NON VA BENE! IL SISTEMA NON E' IN GRADO DI DISTINGUERE DA void f(int);

```

03/10/2022

## de funzioni ricorsive

Una funzione ricorsiva è una funzione che direttamente o indirettamente richiama se stessa.

### IL FATTORIALE RICORSIVO

```
int Fattoriale(int val)
{
    if (val <= 1) //caso base
        return 1;
    else
        return val * Fattoriale(val-1);
}
```

Genera tanti RECORD DI ATTIVAZIONE molti, meglio se iterative.

### FIBONACCI RICORSIVO

```
int Fibonacci(int m) //ITERATIVO
{
    int f, f1=0, f2=1;
    if ((m==0) || (m==1))
        return m;
    else
    {
        for (int K=2; K<=m; K++)
        {
            f = f1+f2;
            f2 = f1;
            f1 = f;
        }
        return f;
    }
}
```

```
int Fibonacci (int m) //RICORSIVO
{
    if ((m==0) || (m==1))
        return m;
    else
        return Fibonacci(m-1)+Fibonacci(m-2);
}
```

NON È EFFICIENTE

Per fare in modo che chiamente e chiaramente comunicino abbiamo bisogno di parametri per riferimento.

### ESERCIZIO Calcola mcd in modo ricorsivo.

$$\text{mcd}(x,y) = x \text{ se } y=0$$

$$\text{mcd}(x,y) = \text{mcd}(y,r) \text{ se } y>0$$

$$x = q \cdot y + r \quad 0 \leq r < y$$

```
int mcd (int x, int y)
```

```
{
    int resto;
```

```
    resto = x % y;
```

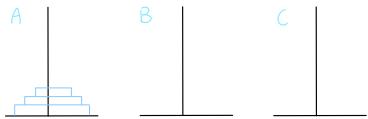
```
    if (resto == 0)
```

```

    return g;
}
else
    return mcd(g, resto);
}

```

## ESERCIZIO - HANOI TOWERS



Possso spostare 1 disco alla volta e solo dalla cima; solo dischi in ordine grande-piccolo. DA A a C.

## SOLUZIONE RICORSIVA

```

Void TowerofHanoi (int m, char from-rod, char to-rod, char aux-rod) //m= numero dischi, char->pila
{
    if (m==1)
    {
        cout<< "Move disk 1 from " << from-rod << "to " << To-rod << endl; //se m=1 sposta disco 1 (piccolo) alla pila To
    }
    else
    {
        TowerofHanoi (m-1, from-rod, aux-rod, to-rod);
        cout<< "Move disk " << m << " from " << from-rod << "to " << to-rod << endl;
        TowerofHanoi (m-1, aux-rod, to-rod, from-rod );
    }
}

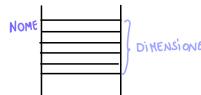
int main()
{
    m=4;
    TowerofHanoi (m,'A','B','C');
    return 0;
}

```

05/10/2022

## Gli array Built-in

Tipo nome [dimensione],



ESEMPIO int a[10];

Il nome dell'array indica il primo elemento.

Le dimensioni sono definite in fase di creazione.

const int dim = valore; // per dimensione - dim non modificabile.

int b[dim];

E' POSSIBILE: int dim;

cin >> dim; // dim non è per forza costante nelle versioni più recenti del linguaggio.

int c[dim];

## Inizializzare gli array

const unsigned sz = 3;

int a1[sz] = {0, 1, 2};

int a2[] = {0, 1, 2}; // senza dim se gli do i valori

int a3[5] = {0, 1, 2} // le ultime due pos. sono inizializzate da default

int a4[2] = {0, 1, 2} // ERRORE!

Bisogna inserire e leggere valore per valore.

Cin >> a1; NO!

a2 = a1; NO!

## STRUTTURA DATI AD ACCESSO DIRETTO (SIANO IN BASE 0)

cout << a1[7]; // accesso alle celle di indice 1

ESEMPIO Leggi un array e stampalo

```
#include <iostream>

using namespace std;
const int dim = 10;
int main()
{
    int a[dim];
    for (int k=0; k<=dim-1; k++) // leggo
        cin >> a[k];
    for (int k=0; k<=dim-1; k++) // stampo
        cout << a[k] << endl;
    return 0;
}
```

OPPURE per accedere agli elementi:

```
for (auto elem: a) // STAMPA
    cout << elem;
```

Non posso accedere solo a determinate pos. e non ho posso tenere conto (di pos.).

## ARRAY E FUNZIONI

Modo CLASSICO

```
void print(int v[dim], int dim) //dim non e' obbligatorio
OPPURE
void print(int *v, int dim)
```

```
int main()
{ int vect[2]={0,1};
  print(vect, 2);
}
```

ESERCIZIO Leggere un array di int di 10 elementi e x e determinare se e' presente nel vettore

```
#include <iostream>
using namespace std;
const int dim= 10;

bool cerca(const int v[], const int dim, const int x)
{
    bool Ris=False;
    for(int K=0; K<= dim-1 && !Ris; K++)
        if (v[K] == x)
            Ris=True;
    return Ris;
}
```

```
int main()
{ int v[dim], x;
  for(int K=0; K<= dim-1; K++)
    cin>> v[K];
  cin>> x;
  if (cerca(v, dim, x))
    cout<< x << " presente " << endl;
  else
    cout<< x << " NON presente " << endl;
}
```

## Gli array multidimensionali (matrici)

int m[3][4]; // mat 3x4

PER ACCEDERE: m[i][j] (righe i, colonne j)

PER INIZIALIZZARE\*: int m[3][4] = {{0,1,2,3},{4,5,6,7},{8,9,10,11}}

POSSO DEFINIRE ANCHE: int a[10][20][30];

## LE FUNZIONI E LE MATRICI

const int Rowsize.... e Colsize...

void print(const int m[][colszie], int Rowsize, int colszie)  
      <sup>se necessario</sup>

// posso evitare le dim. righe ma non col.

OPPURE

void print(const int \*\*m)

ESERCIZIO Leggere un array di dim elementi, trovare il max e stampare l'elenco dei numeri in array e il relativo scarto del max (differenza

PER CASA tra il massimo e il valore di tutti gli elementi -> max, v[], etc... )

ESERCIZIO Leggere da input una parola terminata da un punto e determinare se la parola e' palindroma.

```
#include <iostream>
using namespace std;
const int maxdim=100;
```

```
void LeggiParole (char v[], int &dim)
{ char c;
  dim=0;
  cin>>c;
  while ((c != '.') && (dim < maxdim))
  {   v[dim]=c;
      dim++;
      cin>>c;
  }
}
```

```
bool Palindroma (char v[], int dim)
{
    bool TrovatoDiverso = False;
    for (int k=0; K< dim/2 && !TrovatoDiverso; K++)
        if (v[k] != v[dim-1-k])
            TrovatoDiverso = True;
    return !TrovatoDiverso;
}
```

```
int main()
{
    char v[maxdim]; int d=0;
    LeggiParole(v,d);
    cout(Palindroma(v,d));
    return 0;
}
```

## ESERCIZIO Soluzione completa su TEAMS

5	7	5	9	2	7	5	1
---	---	---	---	---	---	---	---

Vogliamo leggere una seq. di int (men 100) terminata da -1. Mettere in vett ed eliminare i duplicati (comparare).

### PROTOTIPI FUNZIONI

```
void LeggiArray (int v[], int &dim);
```

```
void Stampa (int v[], int dim);
```

```
void Compete (int v[], int &dim)
{
    int k=0;
    while (k<=dim-1)
    {
        if (Trova(v[k], 0, k-1))
            Compete(v, k, dim);
        else
            k++;
    }
}
```

```
void Cancelli (int v[], int pos, int &dim)
{
    for (int k=pos; k<=dim-2; k++)
        v[k]=v[k+1];
    dim--;
}
```

## ESERCIZIO Scrivete una funzione che data una mat. determini se esistono due righe identiche.

```
const int rowsize = .... e col size ...
bool verifica (int m[] [colszie], int rowsize, int colsize)
{
    bool Trovato = false;
    for (int r1=0; r1 <= rowsize-2 && !Trovato; r1++)
        for (int r2=r1+1; r2 <= rowsize-1 && !Trovato; r2++)
        {
            bool identiche = true;
            for (int c=0; c <= colsize-1 && identiche; c++)
                if (m[r1][c] != m[r2][c])
                    identiche = false;
            if (identiche)
                Trovato = true;
        }
    return Trovato;
}
```

## ESERCIZIO Scrivere una funz. che data una mat. determini se tutte le coppie di righe hanno almeno un elemento in comune anche se in colonne diverse.

06/10/2022

## Array e puntatori

```
int a[] = {0,1,2,3,4,5,6,7,8,9}; // a = indirizzo primo elemento  
auto iaz=a; // punta a un intero (int * = tipo)  
  
int *p=a; // come auto iaz=a  
  
iaz=35; // ERRORE perché iaz è un puntatore ad un int  
  
cout << iaz << endl; // Stampa indirizzo iaz  
  
cout << *iaz << endl; // Stampa il primo elemento di a => 0  
  
cout << iaz[1] << endl; // Stampa il valore del secondo elemento; usando [ ] dereferenzio implicitamente OFFSET rispetto all'inizio  
iaz++; // incremento il valore del puntatore; iaz punta alle celle di memoria successive  
  
cout << *iaz << endl; // iaz è incrementato; stampa il secondo elemento => 1  
  
cout << iaz[1] << endl; // stampa 2; offset incrementato di 1 (era in pos 2, ora in pos 3)
```

begin end se ricevono un array BEGIN restituisce l'ind. delle prime celle dell'array. End restituisce l'ind. delle ultime celle +1.

auto m = end(a) - begin(a); // lunghezza di a. NON FUNZIONA NELLE FUNZIONI (restituiscono degli iteratori)

```
int *beg = begin(a);  
int *last = end(a);  
  
while (beg != last)  
{ cout << *beg << " ";  
    beg++;  
}
```

BEG LAST  
0 1 2 3 4 5 6 7 8 9  
LUNGHEZZA a +1

STAMPA IL CONTENUTO DI A  
mi ferma quando beg=last; quando ho ispezionato tutte le celle di memoria di a.

## Puntatura e gestione dinamica della memoria

La gestione dinamica consiste nel poter allocare - liberare memoria durante l'esecuzione del programma all'occorrenza.

ISTRUZIONI FONDAMENTALI new delete Posso usare solo con i puntatori

ESEMPIO new

```
int *pi = new int; // pi non punta a qualcosa che esiste ma ad una celle int senza nome specifico accessibile solo da pi
```

```
int *pi = new int(1024);  

```

```
auto p1 = new auto(obj); // p1 sare' un puntatore al tipo di oggetto di obj
```

```
auto p2 = new auto{a,b,c}; // ERRORE non posso usare l'initializzazione con le graffe
```

## ESEMPIO delete

`delete pi;` // l'indirizzo a cui punta `pi` diventa indefinito; cancella l'area di memoria a cui puntava

`int i = ...;`

`delete i;` // ERRORE, `i` non è un puntatore

## Creazione di memoria dinamica sugli array

`const int capacity = 10;`

`int *elenco = new int[capacity];` // elenco punta ad un array di int con lunghezza 10

`RESIZE` (se mi serve più spazio):

elenco  Temp  } Ricopio elenco in temp, che ha maggiore capacità (perché lo abbiamo definito noi). Ogni resize ha bisogno di copiare l'array ogni volta.

`DELETE` di un array:

`delete [] elenco;` // cancella la porzione di memoria a cui punta elenco. Il puntatore diventa indefinito meglio = `null *`

`delete elenco;` // elimina solo la prima cella ERRORE così la memoria rimane riservata ma non utilizzabile.

`elenco = NULL;` // per `NULL *`

## ESEMPIO array di dimensione non specifica, le addiamo men meno che leggiamo i dati

`const int DimBase = 10;`

`void resize(int *, int &, int);` // PROTOTIPO

`int main()`

{ `int *elenco = new int[DimBase];`

`int size = 0, elem, capacity = DimBase;` // capacity = quanti elem ci metti; size = numero di elementi significativi nell'array

`while (cin >> elem)`

{ `if (size <= capacity - 1)`

`elenco[size++] = elem;`

`else`

{ `resize(elenco, capacity, DimBase);` // aggiungo DimBase posizioni nell'elenco

`elenco[size++] = elem;`

}

}

`return 0;`

}

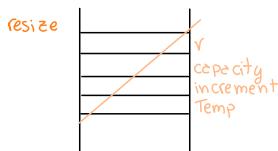
Per reduplicare gli stesso  
capacity invece di DimBase

↳ Riduco il numero di resize

```

void resize(int *v, int &capacity, int increment) // v deve essere passato per riferimento o fuori da resize v non cambia
{
    int *Temp = new int [capacity + increment]; // creo vettore di dimensione capacity+increment
    for (int k=0; k<=capacity-1; k++)
        Temp[k] = v[k]; // copio v in Temp
    delete []v; // cancello l'array piccolo
    v = Temp; // voglio che v punti a Temp (nuovo vettore più grande)
    capacity += increment;
    return;
}

```



Alle fine dell'esecuzione Temp viene cancellato ma lo spazio di memoria no; verrà puntato da v.

NOTA Se perdo l'indirizzo che punta alla memoria del vettore non posso più accedervi

Per una resize di riduzione posso ridurre la capacity.

de stringhe come array di char

```

char str[10]={'C','+', '+', '\0'}; // \0 per fine stringa
char *str2 = new char[10];
cin >> str2; // Posso leggere l'intero vettore di char "\0" viene messo automaticamente

```

Posso usare la libreria <cstring> :

```

strlen(str); // restituisce la lunghezza di str ricercando \0; se non c'è ho un risultato sbagliato.
strcmp(str1, str2); // confronta str1 e str2

```

### ESEMPIO

```

char ca1[]="A string example";
char ca2[]="Another string";
if (ca1 < ca2) // ERRORE
    strcmp(ca1, ca2) // 0 se le stringhe sono uguali; >0 se ca1 è maggiore, <0 se ca1 è minore lessico-graficamente
strcpy(str1, str2); // copia str2 in str1
strcat(str1, str2); // concatena str1 e str2 e mette il risultato in str1

```

10/10/2022

## da complessità computazionale

### 1) MODELLO DI COSTO

- Dimensione dell'input; ①
- Istruzioni a costo unitario; ②

## Tipi di complessità

- 1) Complessità temporale; (tempo).
- 2) Complessità spaziale; (spazio occupato aggiuntivo), spesso più spazio aggiuntivo = tempo.
- 3) Complessità I/O; (quanti trasferimenti di I/O deve fare il programma).

## da complessità temporale

Supponiamo di avere una CPU che esegue un'istruzione in  $10^{-6}$  sec e 3 algoritmi (input m) con complessità diverse (Ricerca in array con array lungo m).

	m=10	m=10.000	m=10 <sup>6</sup>
m	$10^{-5}$ sec	$10^{-2}$ sec	1 sec
$m^2$	0,1 msec	100 sec	$10^6$ sec ~ 12 giorni
m log m	0,00003 sec	0,13 sec	19 sec

### ① DIMENSIONE DELL'INPUT

Parametri che determinano il numero di istruzioni da eseguire (num. elementi array, num. di bit da leggere, valore etc.).

### ② COSTO UNITARIO Assumiamo che hanno costo 1:

- Lettura/scrittura (cin/cout); le letture da file esterni è più lente di quelli interni.
- Assegnamento;
- Arithmetica predefinita; } e combinazioni  
es.  $y = x * 3 + 1$
- Return;
- Accesso ad elementi di Array (Acc. diretto in memoria);
- Valutazione di espressioni booleane di base predefinite (es. (a||b) && (!c) && (c == b))  $\rightarrow$  Approssimato = valle 1)

Supponiamo di voler calcolare il numero di istr. con costo unitario che dobbiamo eseguire

1) i = 1;  $\rightarrow$  assegnamento = 1

while (i <= m)  $\rightarrow$  m+1

i++;  $\rightarrow$  arithmetico di base = 1 · m (m volte esecuzione)

Somma =  $2m + 2$  (Complessità)

2)  $i = 1;$       1  
 $\text{while } (i \leq m)$        $m+1$   
 $\{$        $i = i + 1;$        $\rightarrow 2 \cdot m$   
 $\} \quad j = j * 3 + 37;$        $3m+2$

3)  $i = 1;$        $\star$       1  
 $\text{while } (i \leq 2 \cdot m)$        $2m+2$   
 $\{$        $i = i + 1;$        $2 \cdot 2m$   
 $\} \quad j = j * 3 + 35;$

4)  $i = 1;$       1      Supponiamo che  $m = 9$   
 $\text{while } (i * i \leq m)$        $\sqrt{m}$        $i = 1 \quad i * i = 1 \quad 1 \leq 9 \quad \checkmark$   
 $\{ \quad i = i + 1;$        $2 \cdot \sqrt{m}$        $i = 2 \quad i * i = 4 \quad 4 \leq 9 \quad \checkmark$   
 $\} \quad j = j * 3 + 35;$        $i = 3 \quad i * i = 9 \quad 9 \leq 9 \quad \checkmark$   
 $\} \quad i = 4 \quad i * i = 16 \quad 16 \not\leq 9 \quad \times$

## DETERMINARE SE $x$ E' CONTENUTO IN $V[Dim]$



- **CASO MIGLIORE** al minimo quanto spendo ( $x$  e' alla prima posizione)
  - **CASO PEGGIORE** ( $x$  e' all'ultima pos o non c'e').
- Esiste anche un **CASO MEDIO**.

## NOTAZIONE ASINTOTICA

Vogliamo caratterizzare il comportamento dell'algoritmo quando l'input e' troppo grande. La notazione asintotica veluta con  $m \rightarrow \infty$  ( $m$  = input).

Nell'esempio di prima (\*)  $f(m) = 2m + 2$

Vogliamo capire se l'algoritmo si comporta come:

- 1) una funzione lineare ( $m$ );
  - 2) una funzione polinomiale ( $m^2$ );
  - 3) una funzione logaritmica ( $\log m$ );
  - 4) una funzione esponenziale ( $2^m$ ).
- $\left. \begin{array}{l} g(m) \text{ generale} \\ f(m) \text{ specifica funzione} \end{array} \right\}$

Vogliamo caratterizzare l'algoritmo; per farlo useremo la **NOTAZIONE O**.

$f(m) \in O(g(m))$  se  $\exists c, m_0 > 0 \mid \forall m > m_0 \quad 0 \leq f(m) \leq c \cdot g(m)$        $c = \text{costante qualunque}$



Mi interessa dire se l'algoritmo ha complessità asintotica  $O(m)$ ,  $O(m^2)$  etc.

## ANALISI IN TERMINI DI COSTANTI

Quando  $m$  è piccolo alcuni algoritmi poco efficienti sono migliori.

### LE NOTAZIONI

① complessità minima

② complessità esatta

## BLOCCO DI ISTRUZIONI A COSTO UNITARIO

Il costo è costante, non dipende dell'input, ha complessità  $O(1)$ .

① blocco 1      }       $O(1)$   
                      sei blocchi sono in sequenza  $\Rightarrow O(1) + O(1) = O(1) \Rightarrow$  la complessità dei due è  $O(1)$ .  
blocco 2      }       $O(1)$

② if (cond)      devo valutare il costo delle valutazione della condizione  $f_{\text{cond}}$  la valuto sempre  
    {              }       $f_{\text{if}}$   
else {              }       $f_{\text{else}}$   
                      NEL CASO PEGGIORI  $f_{\text{cond}} + \max(f_{\text{if}}, f_{\text{else}})$   
                      NEL CASO MIGLIORE  $f_{\text{cond}} + \min(f_{\text{if}}, f_{\text{else}})$

③ while (cond)      con  $K$  iterazioni  
    {              }       $K(f_{\text{cond}} + f_{\text{while}})$  se spendo lo stesso nel while è prescindere  
                      Se spendo in base a dei valori devo sommare ogni istruzione con un'analisi.

④ funzioni  $f(\text{parametri})$       se i parametri sono semplici  $\rightarrow$  costo costante  
    {              }      se il percorso è per veloce  $\rightarrow$  quanto costa copiare?  
                      per riferimento = costante  
                      il blocco dipende da cose c'è dentro

### ESEMPIO

void StampaStelle(int K)      PARAMETRI  $\Rightarrow$  1 const.  
{      for (int i=0; i<=K;++)      eseguo  $K$  volte  $O(1)$   
            cout << '\*';      ogni blocco vale 1  $\rightarrow$  è costante  
            cout << endl;  
}

ORDINE DI GRANDEZZA

$O(K)$  lineare in  $K$

```

int main()
{
    int mn, m;      costante
    Cim>> M >> m;  costante
    for (int i=0; i<m; i++)  entriamo nel for m volte
        StampaStelle(m);  COSTO: O(m)
    return Ø;
}

```

]  $m$  volte  
 $O(m)$   $\Rightarrow$  COMPLESSITÀ  $O(m \cdot m)$  E. POLINOMIALE

Supponiamo: (TRIANGOLO)

```

int main()
{
    int mn, m;          i = 0
    Cim>> M >> m;      i = 1   Valore di StampaStelle, cambia in base a i
    for (int i=1; i<m; i++)
        StampaStelle(i);
    return Ø;
}

```

i = 1	1	Valore di StampaStelle, cambia in base a i
i = 2	2	
i = 3	3	$\Rightarrow$ COMPLESSITÀ = $\sum_{i=1}^m i = \frac{m(m+1)}{2} \rightarrow \frac{m^2}{2} + \frac{m}{2} =$
⋮	⋮	
m - 1	m - 1	
m	m	= $O(m^2)$ Ho una complessità quadratica.

12/10/2022

## Il problema della Ricerca

Ricerca di un elemento in un array disordinato.

### RICERCA LINEARE

① bool RicercaLineare (int insieme[], int  $x$ , int  $m$ )  
{     bool Trovato=false;  
    for (int  $K=0$ ;  $K \leq m-1$ ;  $K++$ )  
        if (insieme[ $K$ ] ==  $x$ )  
            Trovato=true;  
    return Trovato;  
}

$m$  volte 1      $C_M = O(m)$       $C_P = O(m)$

NON ESISTONO CASI MIGLIORI O PEGGIORI

② bool RicercaLineare (int insieme[], int  $x$ , int  $m$ )  
{     bool Trovato=false; 1  
    for (int  $K=0$ ;  $K \leq m-1 \&& !Trovato$ ;  $K++$ )  
        if (insieme[ $K$ ] ==  $x$ )  
            Trovato=true;  
    return Trovato;  
}

$2 \leq K \leq m-1 \&& !Trovato$      CASO MIGLIORE ( $x$  e' in pos. 0)     complessità costante      $O(1)$

if (insieme[ $K$ ] ==  $x$ )     CASO PEGGIORE (if mai soddisfatto)      $O(m)$

Trovato=true;

Ricerca di un elemento in un array ordinato

① bool RicercaLineare (int insieme[], int  $x$ , int  $m$ )  
{     bool Trovato=false;  
    for (int  $K=0$ ;  $K \leq m-1 \&& !Trovato \&& insieme[K] \leq x$ ;  $K++$ ) //entro solo se insieme[ $K$ ] <=  $x$   
        if (insieme[ $K$ ] ==  $x$ )  
            Trovato=true;  
    return Trovato;  
}

CASO MIGLIORE:  $O(1)$

CASO PEGGIORE ( $x$  non c'e' ed e' più grande di insieme[m-1]):  $O(m)$

### TECNICA DIVIDE ET IMPARA (BINARIA o DICOTOMICA)

Queste tecniche e' applicabile quando l'insieme puo' essere diviso in sottoinsiemi con le stesse caratteristiche.

- 1) Dividi in sotto problemi;
- 2) Risolvi i sotto problemi;
- 3) Ricombina le soluzioni trovate.

1	3	5	6	7	9	11	18	27	30	31
meta'										11

$\delta\ell = 8$

1) Dividiamo l'array a metà, se 9 e' < 8 controllo SOLO nelle prime metà (Posso DIVIDERE PIÙ VOLTE).



L'ultimo passo e' un vettore contenente solo  $\delta\ell$  o un vettore vuoto.

Devo fare  $\log_2 m$  divisioni  $\Rightarrow O(\log m)$  CASO PEGGIORE  
 $O(1)$  CASO MIGLIORE

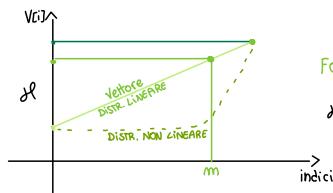
```
bool RicercaBinaria (int v[], int xe, int in, int fin)
```

```
{
    if (in >= fin)
        return ((in == fin) && (xe == v[in]));
    else
    {
        int medio = (in+fin)/2;
        if (v[medio] < xe)
            in = medio + 1;
        else if (v[medio] > xe)
            fin = medio - 1
        else
            in = fin = medio; // o return true;
    }
    return RicercaBinaria (v, xe, in, fin);
}
```

## RICERCA UNIFORME

STIMA DI POS  $\delta\ell$  IN V NON AVENDO INFORMAZIONI SU V

88



FORMULA INVERSA SULLE RETTE

$\delta\ell \rightarrow V[m]$  se la distribuzione degli elementi e' perfetta nello retta, posso stimare la posizione di  $\delta\ell$  (sara'  $V[m]$ ) in un numero costante di passi

## DISTRIBUZIONE NON LINEARE

CASO PEGGIORE: m divisioni, potrei dividere in base a una stima di distribuzione sbagliata (PARTE TROPPO PICCOLA, PARTE TROPPO GRANDE)  $O(m)$

CASO MIGLIORE:  $O(1)$

## Il problema dell'ordinamento - ALGORITMI DI ORDINAMENTO IN LOCO QUADRATICI (non allocano ulteriore memoria)

### ① BUBBLE SORT (Scambio elementi uno di seguito all'altro, scorre più volte il vettore)

```

void BubbleSort (int a[], int N)
{
    bool scambi;
    int &R=N, lastswap, temp;

    do
    {
        scambi=false;
        for (int K=0; K<=R-1; K++)
            if (a[K] > a[K+1])
            {
                Temp=a[K];
                a[K]=a[K+1];
                a[K+1]=Temp;
                scambi=True;
                lastswap=K;
            }
        &R=lastswap;
    } while (scambi);
}

```

#### COMPLESSITÀ

CASO MIGLIORE: a è già ordinato  $O(n)$

CASO PEGGIORE: ho scambiato fino all'ultimo K ( $m-1, m-2, m-3 \dots$  fino a 1)  
 $\Rightarrow O(m^2)$

### ② SELECTION SORT

La parte pesante dell'algoritmo è la ricerca del minimo (spendo  $m, m-1, m-2 \dots 1 \Rightarrow O(m^2)$ )

CASO MIGLIORE:  $O(m^2)$  array già ordinato il numero di istruzioni elementari è minore del Bubble Sort

CASO PEGGIORE:  $O(m^2)$

```

void SelectionSort (int a[], int m)
{
    for (int K=0; K<m-2; K++)
    {
        int min=K; //pos min
        for (int K1=K+1; K1<m-1; K1++)
            if (a[K1]<a[min])
                min=K1;
        int Temp=a[min]; //unico scambio for esterno
        a[min]=a[K];
        a[K]=Temp;
    }
}

```

### ③ INSERTION SORT

```

void insertionSort(int a[], int m)
{
    for (int k=1; k<=m-1; k++)
    {
        int Temp = a[k];
        j = k-1; //cerco pos giuste
        while ((j>=0) && (a[j]>Temp))
        {
            a[j+1] = a[j];
            j--;
        }
        a[j+1] = Temp;
    }
}

```

ALGORITMO MIGLIORE non faccio nessuno scambio (vedi ②), ma solo con assegnazione.

### COMPLESSITA'

CASO MIGLIORE:  $O(m)$  eseguo soltanto 1 volta il controllo del while

CASO PEGGIORI:  $(m, m-1 \dots 1) \Rightarrow O(m^2)$

### INFO AGGIUNTIVA - ORDINAMENTO

$0 \leq V[i] \leq K$  gli elementi in  $V$  si trovano tra 0 e  $K$  ( $mex = k$ )

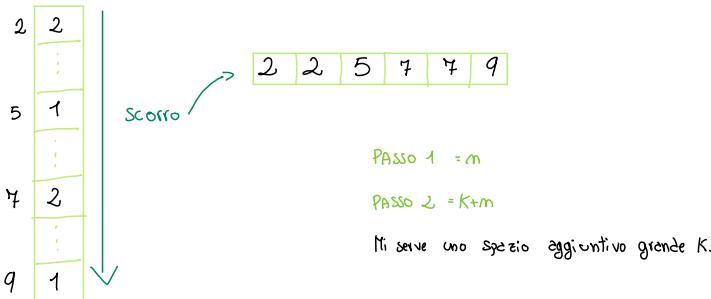
$V$  [ ] cerco il min e lo metto in  $V_2$

$V_2$  [ ] No! complessità =  $O(m^2)$

### ④ COUNTING SORT

Penso sfruttare  $0 < V[i] \leq K$  creando un array di  $K+1$  elementi (cupo intervallo valori). Per ogni  $V[i]$  sfrutto il valore delle celle in  $V_2$  come indice e ogni volta che cerco  $V[i]$  in  $V_2[V_2[i]]$  aggiungo 1.

In  $V_2$  ci sono  $K$  elementi, sfrutto  $V[i]$  come indice in  $V_2$  e in  $V_2[V[i]]$  aggiungo 1.



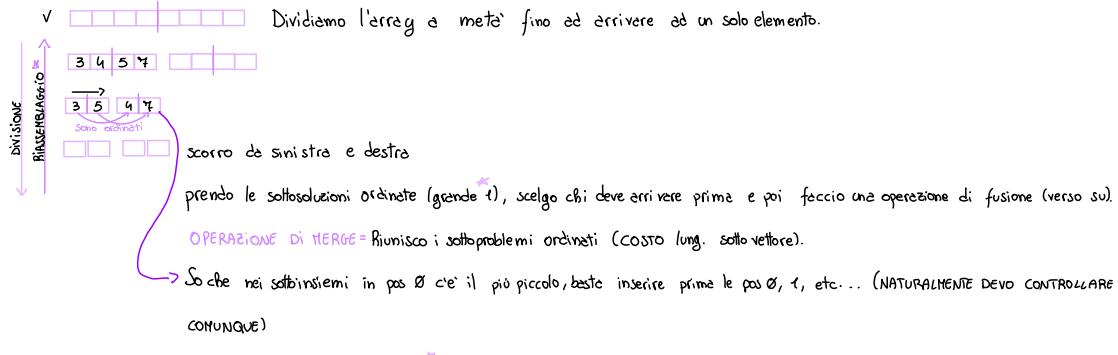
13/10/2022

## Considerazione sul metodo Divide et impazza - TOP DOWN

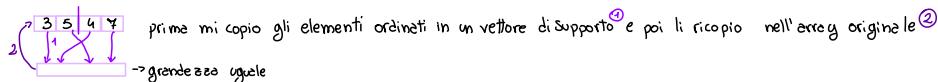
Prendiamo in considerazione il problema di Fibonacci. La soluzione ricorsiva che applica  $\text{fib}(m-1) + \text{fib}(m-2)$  applica già per natura questa tecnica, tuttavia non funziona bene perché richiede il ricalcolo di più istruzioni. NON È EFFICIENTE PER FIBONACCI, NON C'È UN GUADAGNO NELLO SCOMPORRE IN PIÙ SOTTOPROBLEMI.

Algoritmi di ordinamento con Divide et impazza

### 1) MERGE SORT



PER L'OPERAZIONE DI MERGE HO BISOGNO DI UNA VARIABILE D'APPoggio



### IL CODICE

```
void mergeSort(int V[], int in, int fin)
{
    if((fin-in)<=20)
        insertionSort(V,in,fin)
    else
    {
        int medio = (fin+in)/2;

        mergeSort(V,in,medio); // in è medio ordinato
        mergeSort(V,medio+1,fin); // medio+1 è fin ordinato
        merge(V,in,fin,medio); // metto insieme i due sottovettori
    }
}
```

```

void merge (int v[],int in,int fin,int medio)
{
    int A[fin-in];
    int i1=in ,i2=medio+1 ,i3=0; //i1 per metà destra, i2 metà di sinistra, i3 nuovo vettore di supporto
    while((i1<=medio)&&(i2<=fin))
    {
        if(v[i1]< v[i2])
        {
            A[i3]=v[i1];
            i1++;
            i3++;
        }
        else
        {
            A[i3]=v[i2];
            i2++;
            i3++;
        }
    }
    while(i1<=medio) //esco dal while d'iprime quando non ho ancora finito di copiare una parte;
    {
        A[i3]=V[i1]; eseguo questi due (*) per copiare gli elementi rimanenti (ne eseguo solo 1 dei 2)
        i1++;
        i3++;
    }
    while(i2<=fin)
    {
        A[i3]=V[i2];
        i2++;
        i3++;
    }
    for (i3=0 ,i1=in ;i1<=fin ,i3++ ;i1++) //Ricopio da A (vet. supporto) a V (vet. originale).
        V[i1]=A[i3];
}

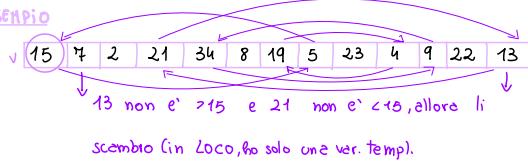
```

## 2) QUICK SORT

$\leq x$  |  $> x$   $\Rightarrow$  ELEMENTO DI PARTIZIONE (SAREBBERE OTTIMALE CHE X FOSSE IL MEDIANO MA AVREI  $O(n^2)$ )

Mi piezzo da qualche parte e inserisco un elemento di partizione  $x$ .

ESEMPPIO



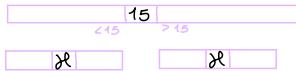
ottimale trovare un elemento che sia al centro, non ho una regola e visto che gli elementi sono a caso lo prendo a caso

PER CONVENZIONE PRENDO IL PRIMO V[0]

alla fine trovo 5 e lo scambio con il 15.

Alla fine ho 15 in mezzo con  $< 15 \quad 15 \quad > 15$ . Si chiama OPERAZIONE di PARTIZIONE (costo m)

Dopo prendo le parte di destra e di sinistra e faccio lo stesso (n volte)



CASO PEGGIORE  $\lambda=1$ , parte destra grande 1, parte sinistra  $m-1$ . SOLUZIONE POSSIBILE: prendo  $\lambda$  e caso oppure metto in disordine l'array.  
L'ARRAY E' GIÀ ORDINATO.  $O(m^2)$

CASO MIGLIORE Divido sempre esattamente a metà  $O(m \log m)$

CASO MEDIO  $O(m \log m)$

Più l'array è disordinato meglio è. (Disordinare l'array ha costo m (lineare)).

## IL CODICE

```
void quickSort(int v[], int in, int fin)
{
    if ((fin - in) <= 20)
        insertionSort(v, in, fin);
    else
    {
        int PosOrd = partizione(v, in, fin);
        quickSort(v, in, PosOrd - 1);
        quickSort(v, PosOrd + 1, fin);
    }
}
```

```
void Partizione(int v[], int in, int fin) // fa meno istruzioni di merge e lavora in loco. Mediamente in termini di costanti
{
    int i = in, j = fin + 1;
    while (i < j)
        do
```

ESISTE UN ALTRO ALGORITMO: HEAP SORT CHE SI BASA SUGLI ALBERI

```

        j--;
        while (v[in] < v[j]);
        do
            i++;
        while (v[in] >= v[i] && i < j)
        if (i < j)
            scambia(v[i], v[j]);
    }
    scambia(v[in], v[j]);
    return j;
}
```

## Esercitazione sulla complessità:

```

1) int f(int x)
{
    return x*x+u;
}

int main()
{
    int n;

    cin >> n;

    int p=1;           CASO MIGLIORE: O(n)
    for (int i=1; i<n; i++) CASO PEGGIORE: O(n)

        p=p*i;

    cout << f(p) << endl;

    return 0;
}

```

2) bool ete5(int a[], int n)  $O(m)$

```

{
    for (int i=0; i<=n-1; i++)
        if (a[i] <= 5)
            return true;
    return false;
}

```

int main()

```

{
    const int N=5;

    int a[N];
    for (int i=0; i<N; i++) O(n)
        cin >> a[i];

    for (int i=0; i<N; i++) O(1)
        if (ete5(a,N) or gt5(a,N))
            return 0;
}

```

bool gt5(int a[], int n)  $O(m)$

```

{
    for (int i=0; i<n; i++)
        if (a[i]>5)
            return true;
    return false;
}

```

CASO MIGLIORE:  $O(2n)$  le costanti non valgono  $O(n)$

CASO PEGGIORE:  $O(3n)$  le costanti non valgono  $O(n)$

```

3) int main()
{
    int n;
    cin >> n;
    int *a = new int[n]; // puntatore ad un vettore di dimensione n
    for (int i=0; i<=n-1; i++) // metto tutto a 0 O(n)
        a[i]=0;
    int i=0;
    bool done=false;
    while (!done)
    {
        for (int k=0; k<=n-1; k++) // stampa O(n)
            cout << a[k] << " ";
        CASO MIGLIORE: O(n)
        CASO PEGGIORI: O(n)
        if (i>=n) // CASO MIGLIORE ESCO SUBITO
            done=true; // PEGGIORI O(m-i)*4
        else
        {
            if (a[i]<4)
                a[i]++;
            else
                i++;
        }
    }
    delete a;
    return 0;
}

```

4) const int N = ...;

```

bool somma (int M[3][N])
{
    int s=0;
    for (int i=0; i<N; i++) // O(m^2) peggiore
        if (M[0][i]==0) // migliore -> escoune
            continue;
    for (int j=0; j<N; j++)
        s+= M[i][j];
    return s>0;
}

```

```

bool Eleboore (int M[3][N], int V[N])
{
    bool b=false;
    int i=1;
    while (i<N && !b) // O(m/2)
    {
        if (somma(M) || V[i-1]==0) // CASO MIGLIORE: O(m)
            b=true; // se il primo elemento di V e' 0 o se le somme e' positivo O(1)
        i=i+2;
    }
    return b;
}

```

```

5) const int N = ...;
   bool g(int vel, int &V[N])
   {
      bool b=false;
      for (int i=0; i<N && !b; i++)
         if (vel == V[i]) CH: vel=V[i] O(1)
            b=true;    CP: O(m)
      return b;
   }

int f(int V1[N], int V2[N])
{
   bool b=false;
   int i=1;
   while (!b && i<N)
   {
      b=g(V1[i], V2) && g(V2[i], V1);
      i=i*2;    CP: log m
   }
   return i;
}

6) void elabora (int M[][][N], int V[N])
{
   bool b=false;
   int j, i=0;
   char car;
   while (i<N && !b)
   {
      cin>>car;
      switch (car)
      {
         case 'a':
            if (f2(N)) //CASO PEGGIORE O(m)
               for (j=0; j<N; j++)
                  cout << M[i][j];
            break;
            f1 O(n)
            f2 O(m^2)
            ..
            CASO MIGLIORE: O(m)
            CASO PEGGIORE: O(m^3)
         case 'b': //CASO MIGLIORE O(1)
            if (f1(M) && f2(V))
               b=true;
            else
               cout << V[i];
            break;
         default:
            cout << V[i];
      }
   }

   for (int x=0; x<N; x++)
      if (M[x][i] != V[x]) //CASO MIGLIORE O(1), CASO PEGGIORE O(m)
         b=true;
      i++;
}

```

17/10/2022

## una programmazione orientata agli oggetti

### LE CLASSI

- 1) Abstrazione;
- 2) information hiding;
- 3) encapsulation;
- 4) gerarchie.

### COME DEFINIRE UNA CLASSE

```
class nome_classe
{
    speciatore1: -----> Private, protected, public
        membri -----> dati + funzioni (o metodi)

    speciatore2:
        membri
}
```

**PRIVATE:** membri accessibili solo all'interno della classe  
speciatore di default

**PROTECTED:** sono accessibili solo degli oggetti della stessa classe o de gerarchie

**PUBLIC:** chiunque puo' accedervi'

le struct e le classi in C++ sono uguali, ma per default le struct hanno come identificatore public.

### ESEMPIO

```
class Rectangle
{
    int width, height; // non metto nullo, sono privati se le metto pubbliche non ho controllo e non sono protetto

    public:
        ① COSTRUTTORE (2 par.) ② COSTRUTTORE DI DEFAULT
        void set_height(int); // FIRMA FUNZIONE } SETTERS
        void set_width(int);
        int get_height(); } GETTERS
        int get_width();
        int area();
        return width * height; } // COSÌ SPORCO LA DEFINIZIONE DELLA CLASSE MEGLIO DEFINIRE L'IMPLEMENTAZIONE FUORI
}
```

INTERFACCIA DELLA CLASSE (progettazione della classe)

### USO DELLA CLASSE

```
int main()
{
    Rectangle rect; // variabile di tipo rettangolo
    rect.set_width(numero_base); // se gli passo 3, width di rect diventa 3
    rect.set_height(4);
    cout << rect.area();
    rect.get_height(); // restituisce l'altezza int di height
}
```

## IMPLEMENTAZIONE DI RECTANGLE (FUORI DAL FILE CLASS O Dopo DEFINIZIONE FIRME)

```
void Rectangle::setHeight(int y)
{
    height = y;
}
```

```
int Rectangle::getHeight() const
{
    return height;
}
```

② COMPORTAMENTO COSTRUTTORE (2 PER)    ④ COSTRUTTORE DI DEFAULT

## USO DI RECTANGLE

```
int main()
{
    Rectangle rect, rectb;
    rect.setHeight(4);
    rect.setWidth(3);
    rectb.setHeight(6);
    rectb.setWidth(5);

    cout << "AREA PRIMO: " << rect.area() << endl; //12
    cout << "AREA SECONDO: " << rectb.area() << endl; //30
}
```

## INIZIALIZZAZIONE DELL'OGGETTO CON COSTRUTTORE

```
① Rectangle(4,5);
② Rectangle::Rectangle(int x, int y)
{
    width = x;
    height = y;
}
```

NON FUNZIONE  
RISOLUTORE SCOPE

## USO DEL COSTRUTTORE

```
int main()
{
    Rectangle rect(4,5); //creo un oggetto rect di base 4 e altezza 5
    rect.setHeight(22); //sovriscrivo l'altezza
}
```

## COSTRUTTORE DI DEFAULT

Rectangle rect(4,5), rectb; //rectb fallisce, il costruttore e' uno ed e' definito con 2 parametri; l'unico modo e' definire un costruttore di default.

③ Rectangle();

④ Rectangle::Rectangle()
{
 width = 0;
 height = 0;
}

PRE-INIZIALIZZAZIONE (DEFINISCO LE VARIABILI QUANDO CREO LO SPAZIO IN MEMORIA PER L'OGGETTO)

Rectangle:: Rectangle (int x, int y) : width(x), height(y)  
{} //OBBLIGATORIE

Il segno quando costruisco e non dopo  
↳ utile se uso oggetti di altre classi nelle mie nuove classi

### USO DI CLASSI IN ALTRE CLASSI:

```
class Circle
{
    double radius;
public:
    Circle(double r): radius(r) {} // PRE-INIZIALIZZAZIONE
    double area()
    {   return radius * radius * 3.14; }
```

```
class Cylinder
{
    Circle base;
    double height;
public:
    Cylinder(double r, double h): base(r), height(h) {}
    {   base.Circle(r); NON POSSO USARE LA PRE-INIZIALIZZAZIONE
        double volume()
        {   return base.area() * height; }
    }
```

LE CLASSI

```
void f(Rectangle x) //Funzione che riceve un oggetto rettangolo
{ ... }
```

ESEMPIO

```
int main()
{
    Rectangle obj(3,4);
    Rectangle * foo, * bar, * baz; //puntatore a rettangolo
    foo = &obj; //mi serve per recuperare l'indirizzo in memoria di obj
    bar = new Rectangle(5,6);
    L> CREA UN NUOVO SPAZIO DI MEMORIA CHE PUO' CONTENERE UN RETTANGOLO
```

baz = new Rectangle[2]; //array di Rectangle di dimensione 2 con costruttore di default

INIZIALIZZARE IN UN ARRAY

```
baz = new Rectangle[2]{ {2,5}, {3,6} };
cout << obj.area();
cout << foo->area(); //area dell'oggetto puntato da foo
L> OPPURE cout << (*foo).area(); } DEREFERENZIO IL PUNTATORE
cout << bar->area(); //OPPURE (*bar).area();
cout << baz[0].area() << " " << baz[1].area();
L> E' UN OGGETTO PERCHE' HO GENERATO UN ARRAY DI RETTANGOLI E NON DI PUNTATORI A RETTANGOLI
delete bar; //se oggetto singolo
delete [] baz; //se vettore
delete obj; NO! obj e' una variabile, obj non punta a un area di memoria
```

OVERLOADING DEGLI OPERATORI

```
class myClass {
    int main
    {
        myClass a,b,c;
        C e+b; //se non sapevamo cosa e' myClass non sapevamo come si comporta +
        -> dobbiamo definire il comportamento degli operatori nella
        classe, in questo caso * = e +
        Con l'eseguimento (=>) posso non definire un modo mio per fare la
        copia, verrà fatta in automatico una COPIA SUPERFICIALE
        Se nella classe usiamo dei puntatori (memoria dinamica) le copie
        superficiale copia solo l'indirizzo.
```

## COME DEFINIRE GLI OPERATORI IN UNA CLASSE

tipo operatore simbolo operatore (parametri)

{ corpo }

### ESEMPIO

classe Vector

{ public:

int x,y;

(Vector()) { }

(Vector(int a, int b): x(a), y(b) { }

### DEFINIZIONE "+"

(Vector operator + (const (Vector &parametro)

friend (Vector operator + (const (Vector &, const (Vector &); //così x, y possono essere privati, "+" puo' accedervi comunque  
friend ostream operator << (ostream &, const (Vector &);

## IMPLEMENTARE UN OPERATORE FUORI DALLA CLASSE

Vector (Vector::operator + (const (Vector &parametro) **//PRIMO METODO OPPURE CAMBIO IL COMPORTAMENTO**  
{ **>TIPO DI RITORNO** **>VISIBILITA'** **<TIPO PARMETRO**

Vector Temp;  
Temp.x=(<sup>x1</sup>x+<sup>x2</sup>parametro.x);  
Temp.y=(y+parametro.y);  
return Temp;

int main()

{ (Vector foo(3,1), bar(1,2), result;  
result=foo + bar; //result.x=4, result.y=3

Possiamo definire gli operatori come voglio (es. il "+" puo' fare "-")

### \* SECONDO METODO

(Vector operator + (const (Vector &sinistro, const (Vector &destra) **//devo passare entrambi i membri su cui voglio fare "+"**  
{ (Vector Temp;

Temp.x=sinistro.x+destra.x;

**CRITICITA'** Se x e y fossero stati membri privati non avrei potuto scrivere

Temp.y=sinistro.y+destra.y;

la funzione per "+" fuori dalla classe.

return Temp;

}

## DEFINIAMO L'OPERATORE "<<" PER LA STAMPA

ostream& operator << (ostream &os, const (Vector &v)  
{ os<< "[" << v.x << ", " << v.y << "]";  
return os;

int main  
{ cout << v << endl; //mi serve le friend

## DEFINIAMO L'OPERATORE ">>" PER L'INPUT

```
istream &operator >>(istream &is, CVector &v)
{
    is >> v.x >> v.y;
    return is;
}
```

**NOTA** Anche qui ci serve la friend nella classe per accedere a x e y

## IL THIS

Individua il puntatore all'oggetto su cui stiamo lavorando all'interno della classe stessa.

```
class Dummy
{
public:
    bool isitme(Dummy &parametro)
    {
        return this == &parametro; // & mi serve perche' this e' un indirizzo
    }

int main()
{
    Dummy a;
    Dummy *b = &a;
    if (b->isitme(a)) // b e' un puntatore
        cout << "Si ";
    else
        cout << "NO ";
}
```

## THIS E CVECTOR OPERATORE "="

CVector CVector::operator=(const CVector &parametro) // this deve diventare uguale a parametro

```
{
    x = parametro.x;
    y = parametro.y;
    return *this;
}
```

**I MEMBRI STATICI** e' un tipo speciale associato all'intera classe in modo estetico e non e singoli oggetti (N.B. esistono anche le funzioni static).

Contiamo i tipi Dummy istanziati in memoria

```
class Dummy           // n va generato prima dell'esecuzione del programma int Dummy::n=0; (VAR.GLOBALE o LOAD)
{
public:
    static int n;
    Dummy() { n++; }
}
int main()
{
    Dummy a; // n=1
    cout << a.n; // OPPURE cout << Dummy::n; // tramite ogni oggetto Dummy posso accedere a n
    Dummy b[5]; // n=6
}
```

FUNZIONI MEMBRO COSTANTI

```

class myClass
{
public:
    int xe;
    myClass(int val): xe(val) {}
    int get() {return xe;}
    * int get() const {return xe;} // non modifica lo stato dell'oggetto e' la funzione ad essere costante
    const int &get() {return xe;} // cosi' restituisco un riferimento costante e' il tipo di ritorno che e' costante, ha senso solo con &
    const int &get() const {return xe;} // non modifica lo stato e restituisco un riferimento costante
}

int main()
{
    const myClass foo(10); // se e' const posso inizializzarlo solo durante la costruzione
    foo.xe=20; // NO!
    cout << foo.xe << endl; // stampa 10
    cout << foo.get() << endl; // NO! le variabili costanti possono usare solo funzioni costanti*
}

```

↳ Per questo motivo bisogna prevedere delle funzioni const per tutto (per la get e' meglio tenere solo la versione const).

E' POSSIBILE PREVEDERE DELLE VERSIONI DI FUNZIONI PER CONST E NORMALILE FUNZIONI NELLA CLASSE

```

const int &get() const {return xe;}
* int &get() {return xe;}

```

IL MAIN

```

int main()
{
    myClass foo(10);
    const myClass bar(20);
    cout << foo.get() << endl; // richiama la funzione non const
    cout << bar.get() << endl; // richiama la funzione const
    * foo.get() = 25; // visto che la funzione non const lavora su un riferimento posso modificare il riferimento allo variabile xe
    ↳ E' UGUALE A foo.xe=25; ↳ LEFT END OPERATOR
    bar.get() = 25; // NO! e' una costante
}

```

E' meglio definire una funzione di cui non dobbiamo modificare il contenuto sicuremente come const (tanto funzione sia per i tipi normali che per quelli const).

## METODI DELLE CLASSI

- costruttore di default;
- distruttore;
- costruttore di copie;
- operatore di assegnamento (COPY ASSIGNMENT)

## IL DISTRUTTORE

Utile per le classi che usano la memoria dinamica.

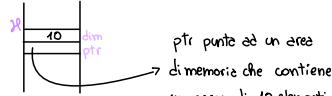
```
class Exemple
{
    int *ptr; int dim;

public:
    Exemple (int n) { dim=n; ptr = new int[dim]; }

} *
```

int main()

```
{ Exemple x(10);
}
```



## IL CODICE DEL DISTRUTTORE (~ => alt -126)

```
* ~Exemple() { delete []ptr; }
```

// in questo modo libero anche lo spazio di memoria a cui punta ptr.

E' importante cancellare l'area di memoria a cui punta un puntatore prima di cancellare lo stesso

## IL COSTRUTTORE DI COPIA

Consiste nel definire come si deve comportare il costruttore quando vogliamo definire un oggetto come copia di un altro

int main()

```
{ Exemple foo(15), x(20);
```

Exemple bar(foo); // usa il costruttore di copia

Exemple baz=foo; // anche in questo caso richiamo il

costruttore di copia. '=' e' interpretato come inizializzatore  
perche' lo faccio durante la dichiarazione di baz.

```
} bar=x; // normalmente fa la copia superficiale
```

Se ridefiniamo '=' per la classe funziona.

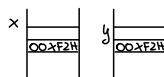
Exemple & operator = (const Exemple & x)

```
{ delete []ptr; // devo prima cancellare il contenuto precedente (foo)
```

(copy del codice)

return \*this;

### COPIA SUPERFICIALE



Puntano alle stesse aree (memoria condivisa)

Puoi dare molti problemi

meglio usare un costruttore di copia \*

### NELLA CLASSE (COPIA PROFONDA)\*

Exemple(const Exemple & x)

```
{ dim=x.dim;
    ptr = new int[dim];
    for (int i=0; i<=dim-1; i++)
        ptr[i] = x.ptr[i];
}
```

## COME SCRIVERE UNA CLASSE CHE FUNZIONA SU PIÙ TIPI DI DATI (TEMPLATE)

Usiamo TEMPLATE.

Scriviamo una classe coppia che puo' contenere un qualsiasi tipo di dato.

Template <class T> //T tipo generico di dato

```
class mypair  
{ T a, b;  
public:  
    mypair(T first, T second);  
}
```

DEFINISCO IL COSTRUTTORE FUORI

Template <class T>

```
    mypair<T>::mypair (T first, T second)
```

```
    { a=first;  
      b=second;  
    }
```

```
int main()  
{ my pair<int> a(10,20); //Ho istanziato su int  
    my pair<char> b('x','y'); //Ho istanziato su char  
}
```

Ho molto piu' controllo dell'auto perche' sono io a dichiarare il tipo.

**CRITICITA'** Dobbiamo assicurerci che il tipo T supporti qualsiasi tipo di funzione usera', e' necessario gestire piu' casi.

→ my pair<Exemple> c{foo, bar}; //devo essere sicuro di poter gestire anche une coppie di Exemple

21/10/2022

### PROBLEMI DEGLI ARRAY STANDARD.

- Dimensione fissa;
- indici fissi;
- dimensione "reale";
- inserimento in ultima posizione;
- per conoscere la dimensione e la capacità ho bisogno di una variabile.

### COSE UTILI DEGLI ARRAY STANDARD.

- Accesso diretto (basato sugli indici).

### CLASSE VECTOR

```
//file Vector.h  
#include <assert.h>  
Template <class R>  
  
class Vector  
{  
    R *Vec;  
    int sz; l'indice  
    int cap; la capacità  
  
public:  
    Vector(); costruttore di default  
    Vector(const Vector<R> &); costruttore di copia  
    ~Vector(); destruttore  
    int const size();  
    int const capacity();  
    R operator [] const (int); per indicizzare  
    R &operator [] (int); per indicizzare  
    Vector &operator = (const Vector<R> &); per assegnare  
    void push_back (const R &); inserisci in prima posizione libera (size)  
    const R &back const (); estrai l'ultimo elemento (solo visualizzazione)  
    void pop_back(); elimina in ultima posizione  
    void reserve (int size); riserva un certo spazio size nella memoria senza cambiare il numero di elementi significativi  
    void resize (int size); ridimensiona l'array in base alle size che gli passo come parametro, cambio il numero di elementi significativi  
}
```

Meglio inserire tutti i file class in una cartella per avere una nostra libreria.

**INFORMATION** Nasconde all'utente la **HIDING** parte implementativa della classe.

#IFNDEF VECTOR\_H  
#DEFINE VECTOR\_H *(alla fine #ENDIF)*  
#> DIRETTIVA  
#IFNDEF ... ENDIF evita di includere il file class più volte.

## L'IMPLEMENTAZIONE DEI METODI

Template <class R>

```
Vector<R>::Vector() // siamo fuori dal file class , ci serve l'identificatore
{
    sz=0; cap=1; // poi lo ridimensionerò
    Vec = new R[cap];
}
```

Template <class R>

```
Vector<R>:: ~Vector()
{
    delete [] vec;
}
```

Template <class R>

```
R &Vector<R>:: operator [] (int i) // la versione const e' uguale ma senza &(ref.) , restituisco una copia
{
    assert(i>=0 && i<=sz-1); // se la condizione non e' soddisfatta l'esecuzione del programma viene bloccata
    return vec[i]; // restituisco l'elemento in posizione i
}
```

Template <class R>

```
void Vector<R>::push_back(const R &e) // aggiungo l'elemento e in posizione size
{
    if(sz==cap)
        reserve(sz*2); // se sz uguale a prima e cap raddoppia
    vec[sz]=e; // uso reserve perché devo inserire a size, se uso reserve aggiungerei a cap (lascerei spazio vuoto)
    sz++;
}
```

Template <class R>

```
void Vector<R>::reserve(int size)
{
    if(size>cap)
    {
        cap=size;
        R *Tmp = new R[cap]; // uso Tmp come variabile temporanea , copio V e lo elimino ,assegno a V Tmp
        for (int i=0; i<=sz-1; i++)
            Tmp[i]=vec[i];
        delete [] vec;
        vec=Tmp;
    }
}
```

Template <class R>

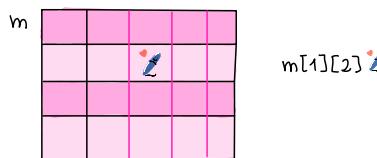
```
void Vector<R>::resize (int size)
{
    reserve(size);
    size=size;
}
```

### USO DELLA CLASSE VECTOR

```
#include <iostream>
using namespace std;
#include "Vector.h"; // Vector.h deve essere nella stessa cartella del main
int main()
{   Vector<int> mioVet; // creo un vettore con sz=0 e cap=1
    for (int i=0; i<10; i++)
        mioVet.push_back(i); // la dimensione viene controllata da push-back
    for (int i=0; i<=mioVet.size()-1; i++)
        cout<< mioVet[i]<< endl;
```

Svuota l'array da fine a inizio per casa

```
Vector<char> vc;
Vector<Vector<int>> m; // matrice di int (vettore multidimensionale)
```



}

### FUNZIONI INTERESSANTI DELLA LIBRERIA VECTOR

- max\_size
- empty
- shrink\_to\_fit // diminuisce cap e portale a size
- at // uguale a []
- front // visualizza il primo elemento
- insert // inserisce in posizione specifica (inserisci anche insiemi di elementi)
- begin - end (ITERATORI) // permettono di iterare sugli elementi for (Vector<int>::iterator i=v.begin(); i!=v.end(); ++i) scosse v  
↳ SIMILE AD UN PUNTATORE  
cout<< \*i << endl; e lo stampa

Vogliamo definire una classe che puo' gestire gli insiemi.

Le funzioni da gestire: ADD, REMOVE, ISIN etc.

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

#include <algorithm> // contiene degli algoritmi gi' pronti
#include <iiterator> // mi permettono di muovermi in une strutture dati

#include <vector>

Template <class T>

class insieme
{
    vector<T> s; // PARTE DATI

public:
    insieme(); // COSTRUTTORE DI DEFAULT
    insieme(const insieme& set): s(set.s) {} // usiamo il costruttore di vector grazie alle pre-inizializzazioni
    // IL COSTRUTTORE NON SERVE PERCHÉ DELLA DISTRIBUZIONE DELLA PARTE DINAMICA SI OCCUPA VECTOR

    bool empty() const
    { return s.size == 0; } // se la size di s è 0 l'insieme è vuoto

    bool add(const T& e); // RIFERIMENTO COSTANTE (uguale a passaggio per valore ma risparmio tempo)
    bool remove(const T& e);
    bool isin(const T& e);

    insieme<T>& unionTo(const insieme& set);
    insieme<T>& intersectTo(const insieme& set);
    insieme<T>& differenceWith(const insieme& set);

    Template <class U>
    friend ostream& operator << (ostream& o, const insieme<U>& set) // uso << perché siamo fuori dalla classe
}; // U = insieme di qualunque tipo

}
```

### DEFINIZIONE METODI

Template <class T>

```
bool insieme<T>::add(const T& e)
{
    if (find(s.begin(), s.end(), e) != s.end())
        // CONTENUTO IN ALGORITHM // restituisce un iteratore che punta all'elemento (se non c'è punta ad end). vuole 2 iteratori e l'elemento da trovare
        return false; // l'elemento e c'è già
    s.push_back(e);
    return true;
}
```

**SVANTAGGIO FIND:** Non so che algoritmo di ricerca usa;  
non ho pieno controllo delle complessità.

→ CONTENUTO IN ALGORITHM // restituisce un iteratore che punta all'elemento (se non c'è punta ad end). vuole 2 iteratori e l'elemento da trovare

return false; // l'elemento e c'è già

s.push\_back(e);

return true;

```

Templete <class T>

bool insieme::remove(const T& e)
{
    auto it = find(s.begin(), s.end(), e); //in it c'e' un iteratore che punta all'elemento e in s (se c'e')
    if(it!=s.end) // Se e' uguale a end non l'ho trovato
    {
        s.erase(it); //ese vuole un iteratore come parametro; si occupa di eliminare l'elemento puntato e di compattare
        return true;
    }
    return false; //l'elemento e' non e' presente
}

```

```

Templete <class T>

insieme<T>& insieme<T>::unionTo(const insieme<T> & set)
{
    insieme<T> *Temp = new insieme<T>(set); //inizializzo a set (costruttore di copia)
    for (int i=0; i<=s.size()-1; i++)
        temp->add(s[i]);
    return *Temp; //DREFERENZIO
}
FARE INTERSECT E DIFFERENCE E <

```

### USO DELLA CLASSE

```

int main()
{
    insieme<int> set, set2;
    set.add(1); set.add(2); set.add(3); etc...
    cout<< set;
    if (set.remove(1))
        cout<< "Rimosso 1\n";
    if (!set.remove(1))
        cout<< "1 non c'e' piu'\n";
    if (set.isIn(2))
        cout<< "2 c'e'\n";
}

```

## LE STRINGHE

```
#include <string>
```

Le stringhe sono viste come vettori di char.

La lunghezza di una stringa è restituita dal metodo `.length()`, con `[]` accediamo agli elementi, `+=` concatena (come `append()`), `find` permette di cercare una sottostringa e restituisce un iteratore che punta alle prime celle, `find-first-of` restituisce la posizione della prima occorrenza del carattere passato, `getline` permette di leggere tutto ciò che c'è sullo stream fino al ritorno a capo (terminatore),  
`>>` legge una serie di caratteri fino ad un terminatore

## ESEMPIO

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
#include <iomanip>

using namespace std;
int main()
{
    vector<string> vs;
    string s;
    while (getline(cin, s) && s != "end") // getline vuole lo stream da cui leggere e la stringa su cui scrivere
        vs.push_back(s); // LEGGO
    sort(vs.begin(), vs.end()); // ordina il vettore (non sappiamo che algoritmo viene usato) // ORDINO
    auto pos = unique(vs.begin(), vs.end()); // unique genera una porzione di vs che contiene solo valori unici, i duplicati si trovano in fondo
                                                // fino a pos tra elementi unici, dopo pos i duplicati restituisce un iteratore
    vs.erase(pos, vs.end()); // cancella i duplicati // ELIMINO i Duplicati
}
```

CONTAINERS

I più importanti contenitori di dati:

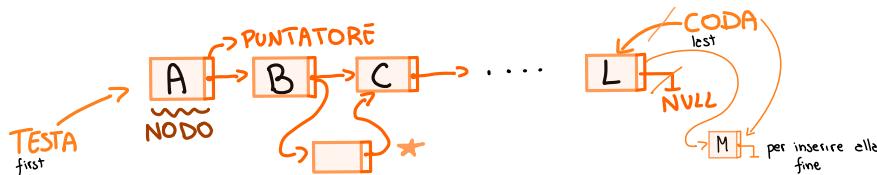
- ① Vector;
- ② List (Forward list);
- ③ Stack / Queue / Priority Queue  
ADAPTORS (Le code sono usate per gestire i BUFFER) (Lo STACK gestisce le chiamate ricorsive)
- ④ Unordered map / map  
ASSOCIATIVE CONTAINERS

LE LISTE DI DATI

Vogliono collezionare un certo numero di elementi sfruttando al massimo la memoria dinamica. Non permettono l'accesso diretto agli elementi.

Ciascun elemento conosce solo il suo successore. → PROBLEMA

Per inserire tre elementi cambio i puntatori interessati \*



Le liste e' strutturate come un'insieme di elementi (o nodi) dove ognuno di questi contiene il valore da contenere e un puntatore al successivo.

L'ultimo elemento punta a NULL.

TESTA e' un puntatore speciale che individua il punto d'accesso al primo elemento della lista, puo' essere utile averne uno anche alla fine (CODA)

Se non ho riferimento al primo elemento (TESTA) perdo le liste, non posso piu' accedervi.

SE LA LISTA E' VUOTA i puntatori testa e coda puntano a nullo, sono inizializzati a NULL. Se le teste non e' NULL la lista non e' vuota.

SPRECO → Spezio per il puntatore in ogni nodo COMPENSATO dalle dinamicita' con cui posso gestire il numero di nodi.

Tutti i nodi devono essere dello stesso tipo perch'e' il tipo a cui puntano i puntatori deve essere omogeneo.

LA CLASSE LISTIMPLEMENTAZIONE DELLA CLASSE NODE

```
Template <class T>

class Node
{
    T value; // valore nodo
    Node<T>* nextNode; // puntatore al next
    friend class List<T>; // per far accedere List a tutto
public:
    Node(const T& v): nextNode(0), value(v) {};
    T getValue() const
    {
        return value;
    }
}
```

```

Node<T> *getNextNode() const // vogliamo proteggere la classe list
{
    return nextNode;           // l'utente non deve poter cambiare l'indirizzo di un puntatore al next
}

```

### IMPLEMENTAZIONE DELLA CLASSE LIST

```

template <class T>
class List
{
protected: // visibile della classe e da classi derivate
    Node<T> *first;
    Node<T> *last;
    Node<T> *newNode(const T&); // funzione protetta

public:
    List(): first(0), last(0) {}; // COSTRUTTORE DI DEFAULT
    ~List(); // DISTRUTTORE Definire il distruttore del nodo che cancella se stesso e il successivo e' sbagliato; non posso cancellare 1, solo 2
        // VA BENE SE UN NODO DISTRUGGE SOLO SE STESSO

    void pushFront(const T&);
    void pushBack(const T&);
    bool popFront(T& v);
    bool popBack(T& v);
    bool empty const
    {
        return first==0;
    }

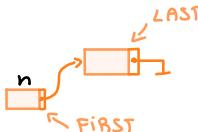
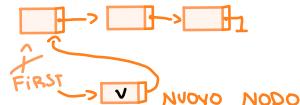
    Node<T> *front(); // per muoversi nella lista, puntatore al primo nodo
    void remove(const T& v); // cancella le occorrenze di v
    void deleteNode(Node<T> *&); // rimuovi nodo di cui ho il puntatore
}

```

```

template <class T>
void List<T>::pushFront(const T& v)
{
    Node<T> *n = newNode(v); // costruisco un nodo che contiene v e mi restituisce il puntatore
    if (empty())
        first=last=n;
    else
    {
        n->nextNode = first;
        first = n;
    }
}

```



```

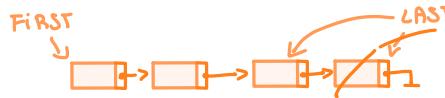
template <class T>
List<T>::~List()
{
    if (first == 0) // LISTA VUOTA
        return;
    Node<T> * curr = first;
    Node<T> * temp;
    while (curr != 0) // intendo che curr punta a qualche nodo
    {
        temp = curr;
        curr = curr -> nextNode; // nodo successivo
        delete Temp;
    } FIRST e LAST si cancellano automaticamente
}

```

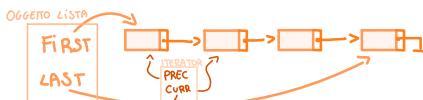
```

template <class T>
void List<T>::pop_back(T &v)
{
    if (empty())
        return false;
    Node<T> *Tmp = last;
    if (first == last) // c'e' un solo elemento
        first = last = 0;
    else
    {
        Node<T> *curr = first;
        while (curr->nextNode != last) // devo trovare il nodo che punta a LAST
            curr = curr->nextNode; // alla fine curr e' il penultimo elemento
        last = curr;
        last->nextNode = 0;
    }
    v = Tmp->value;
    delete Tmp;
    return true;
}

```



## IL FUNZIONAMENTO DEGLI ITERATORI



e' utile avere un oggetto iteratore curr, per spostarmi nella lista e un precedente  
L e' la lista a cui appartengono gli iteratori

GRAZIE AGGI ITERATORI POSSO IMMAGINARE QUESTI METODI:

moveAtFirst // inizio

moveAtLast // fine

operator \* // restituisce il valore delle celle a cui punta l'iteratore curr

operator ++ // avanzza di un nodo

operator -- // indietreggià di un nodo

insert(T) // inserisci un nuovo nodo tra prec e curr

l.begin(); // primo elemento etc..

### COME DEFINIRE UN ITERATORE PER LE LISTE

La classe iterator deve essere friend di Node e List!

```
template <class T>
class iterator
{
    Node<T> * prec;
    Node<T> * curr;
    List<T> & l; // come se la lista fosse in questa classe (in l)

public:
    iterator(List<T> & lista): l(lista), curr(lista.first), prec(0) {}

    T operator *()
    {
        return curr->value;
    }

    void operator ++
    {
        prec = curr;
        curr = curr->nextNode;
    }

    void insert(const T& val)
    {
        if (curr == first)
        {
            l.push_front(val); // o cose punte curr dopo b insert dobbiamo definirlo noi
            curr = l.first; // in questo caso punta al nuovo elemento
            prec = 0;
        }
        else if (prec == l.last)
        {
            l.push_back(val);
            curr = l.last;
        }
    }
}
```

```
else
{
    Node<T> *N = new Node(val);
    N->nextNode = curr;
    prec->nextNode = N;
    curr = N;
}
}
```

### LISTA DOPPIAMENTE LINKATA

Potrei pensare di aggiungere al nodo anche un puntatore al precedente ; più complesso ma più efficiente nell'accesso ai dati.

ADAPTORS:

<b>PILE</b> (STACK)	<b>CODA</b> (QUEUE)
LIFO	FIFO

LIFO = Last in first out

FIFO = First in first out

LE PILE vengono usate nelle chiamate ricorsive.

OPERAZIONI COMUNIPILE

- Push
- Pop
- Empty
- Top
- Back

CODA

- Push
- Pop
- Empty
- Front

IMPLEMENTAZIONESOLUZIONE 1

vector

PILE (indifferente)

PUSH -&gt; push\_back

POP -&gt; back + pop\_back

CODA (meglio usare le liste)

PUSH -&gt; push\_back

POP -&gt; pop\_front

ESEMPIO

Controllare se una data espressione è ben parentesizzata e date le posizioni di una parentesi aperta definire la posizione della parentesi che la chiude.

```
#include <iostream>
#include <cstring>
#include <stack>

using namespace std;
int main()
{
    string expr;
    int len, pos;
    bool esci=false;
    stack<int> s;
    getline(cin, expr); // legge fino al capo de cin e lo mette in expr
    len = expr.size(); // lunghezza di expr
    for(int i=0; i<len && !esci; i++)
    {
        if(expr[i]== '(')
            s.push(i);
        else if(expr[i]== ')') // CASO PEGGIORE: Trovo una parentesi chiusa ma lo stack è vuoto
        {
            if(s.empty())
                esci=true;
            else
            {
                pos=s.top();
                s.pop();
                cout<< "La parentesi aperta in "<< pos << " si chiude in "<< i << endl;
            }
        }
    }
}
```

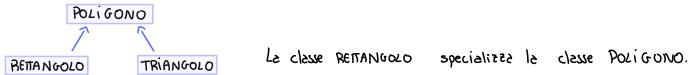
```

if(!s.empty())|| esci=true)
    cout<< "Non ben parentesiizzato!"<< endl;
else
    cout<< "Ben parentesiizzato!"<< endl;

```

## EREDITARIETÀ

Supponiamo di avere il concetto di POLIGONO (rappresentato per semplicità da base e altezza).



## LA SINTASSI

```

class nomeclassederivate: specificatore
{ ... }

```

+ PUBLIC  
+ PRIVATE  
+ PROTECTED

Definisce il massimo livello di visibilità che voglio consentire agli elementi di classi derivate.

con PROTECTED quello che ho pubblico diventa protetto, quello che è privato rimane tale

con Public non cambia niente; Private non ha senso, non posso accedere a nulla.

## ESEMPIO SU POLIGONO

```

class Poligono
{
protected:
    int base,altezza;
public:
    void setValues(int a, int b)
    {
        base=a;
        altezza=b;
    }
}

```

```

class Rettangolo: public Poligono // RETTANGOLO estende in modo pubblico Poligono
{
public:
    int area()
    {
        return base * altezza;
    }
}

```

class Triangolo: public Poligono

```

{
public:
    int area()
    {
        return base * altezza / 2;
    }
}

```

```

int main()
{
    Rettangolo r;
    Triangolo t;
    r.setValues(4, 5); // posso uscire perché RETTANGOLO l'ha ereditata da Poligono
    r.base // NO! base è un membro protetto
}

```

```
t.set_values(4,5);  
  
cout<<r.area()<<endl; //richiama de RETTANGOLO (20)  
  
cout<<t.area()<<endl; //richiama de TRIANGOLO (10)  
}
```

I membri pubblici sono accessibili a tutti; quelli protetti da elementi delle classi e delle derivate; privati solo da elementi della classe.

03/11/2022

## EREDITARIETÀ E POLIMORFISMO

```
class Rettangolo : public Poligono  
{ ... }
```

Con l'ereditarietà, usando `public`, eredito tutto tranne **costruttore e distruttore**. Se non lo specifico il costruttore di default viene chiamato automatico quando costruisco un oggetto di classe figlio; **membri dell'operatore "="**, bisogna ridefinirlo in classe figlio; **le friend** non vengono estese alle derivate; i membri privati.

**Posso sempre fare l'overloading per ridefinire i metodi ereditati.**

### COSA SUCCIDE CON LE CHIAMATE AI COSTRUTTORI NELL'EREDITARIETÀ

```
class Mother  
{  
    public:  
        Mother() { cout << "Mother: nessun parametro\n"; }  
        (2) Mother(int a) { cout << "Mother: un parametro\n"; }  
}  
  
class Daughter : public Mother  
{  
    public:  
        Daughter(int a) { cout << "Daughter: un perimetro\n"; }  
}  
  
class Son : public Mother  
{  
    public:  
        Son(int a) : Mother(a) { //inizializzazione; richiamo il costruttore che riceve un parametro della classe Mother  
            { cout << "Son: un parametro\n"; }  
}  
}
```

```
int main()  
{  
    Daughter Kelly(0); //STAMPA: Mother: nessun perimetro viene prima costruita la classe base (con default perché non specifico)  
    cout << Kelly; //STAMPA: Daughter: un perimetro  
  
    Son bud(0); //STAMPA: Mother: un perimetro viene costruita la classe base, grazie alla PRE-INIZIALIZZAZIONE il costruttore (2)  
    cout << bud; //STAMPA: Son: un perimetro  
}
```

## EREDITARIETÀ MULTIPLO

```
class Rettangolo : public Poligono, public output //Rettangolo eredita tutto sia da Poligono che da output  
{ ... }
```

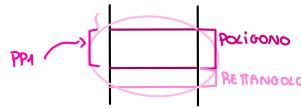
## IL POLIMORFISMO NELLO SPECIFICO

```

class Poligono
{
protected:
    int base, altezza;
public:
    void setValues(int a, int b)
    {
        base=a; altezza=b;
    }
}

int main()
{
    Rettangolo r;
    Triangolo t;
    Poligono * pp1 = &r;
    Poligono * pp2 = &t;
    pp1 -> setValues(4, 5);
    pp2 -> setValues(4, 5);
    cout<< pp1 -> area(); // NO! pp1 punta a un poligono che non ha un metodo area
    cout<< r.area(); OK!
}

```



PP1 punta alla sola parte Poligono di r  
(> e' come se se spesse quel e' la parte poligono in r)

Supponiamo di aggiungere un metodo area in poligono

\* virtual int area() {return 0;} \* // fare return 0 e' concettualmente sbagliato



ADESSO...

cout<< pp1 -> area(); // pp1 e' un poligono; richiamo area della classe poligono \* (STAMPA 0)

## VIRTUAL

Aggiungendo virtual diciamo che se l'oggetto e' un'estensione della classe padre, se nella classe figlio c'e' un overload della funzione virtual usa le funzioni overload. Funziona solo con i puntatori.

con VIRTUAL cout<< pp1 -> area(); // 20, richiamo area di rettangolo



virtual int area()=0; // meglio definire il metodo come VIRTUAL PURO e definire il metodo nelle classi derivate

Ogni classe che ha almeno un metodo virtuale puro e' una classe estratta - NON POSSO DEFINIRE OGGETTI, SOLO PUNTATORI

Grazie al Polimorfismo posso definire collection che contengono dati non omogenei.

vector<Poligono\*> Vp;

Vp puo' contenere oggetti di tutti i tipi "figli" di poligono

Vp[0]= new Rettangolo; // posso anche stampare nuovi oggetti di classi derivate di Poligono

Vp[0]->set\_values.. // accedervi etc.

### IL THIS NELLA CLASSE ASTRATTA (POLIGONO)

```
void printArea()
{
    cout << this->area(); } // richiama il metodo area della classe dell'oggetto this (che sia rettangolo o triangolo)

N.B. Area() in poligono e' un metodo virtuale puro
```

### STACK CON LISTE ED EREDITARIETÀ

```
template <class T>

class Stack : protected List<T>
{
public:
    Stack();
    ~Stack();
    void push(const T& v)
    {
        List<T>::push_front(v); } // posso fare lo stesso con top, pop etc.

    using List<T>::empty; // rende nuovamente visibile il metodo empty di List<T>, e quindi utilizzabile anche per oggetti di tipo Stack.
}
```

## 1) STAMPA DI UNO STACK

## SOLUZIONE ITERATIVA

```
void Stampa()
{
    if(pile.empty())
        cout << "Stack vuoto!" << endl;
    else
    {
        vector<int> mioV;
        while(!pile.empty())
        {
            mioV.push_back(pile.top());
            pile.pop();
        }
        for(int k=0; k<=mioV.size()-i; k++)
            cout << mioV[k] << endl;
        for(int k=mioV.size()-i; k>=0; k--)
            pile.push(mioV[k]);
    }
}
```

## SOLUZIONE RICORSIVA

```
void printStack(stack<int> s)
{
    if(s.empty())
        return;
    int val=s.top();
    printStack(s);
    s.pop();
    cout << val << " ";
    s.push(val);
}
```

**LIMITAZIONE:** non posso stampare stack costanti.

## 2) TOWER OF HANOI - STACK

```
void TowerOfHanoi(int n, stack<int> from, stack<int> to, stack<int> aux, char f, char t, char a)
```

```
{
    if(n==1)
    {
        to.push(from.top());
        from.pop();
        printRodInOrder(from,to,aux,f,t,a);
        return;
    }
    else
    {
        TowerOfHanoi(n-1,from,aux,to,f,t,a);
        to.push(from.top());
        from.pop();
        printRodInOrder(from,to,aux,f,t,a);
        TowerOfHanoi(n-1,aux,to,from,a,t,f);
    }
}
```

```
void printRodInOrder(stack<int> from, stack<int> to, stack<int> aux, char f, char t, char a)
{
    if(f=='A') {cout << "A |"; printStack(from); cout << endl;}
    else if(f=='B') {cout << "B |"; printStack(to); cout << endl;}
    else if(f=='C') {cout << "C |"; printStack(aux); cout << endl;}
    if(t=='A') {cout << "A |"; printStack(to); cout << endl;}
    else if(t=='B') {cout << "B |"; printStack(from); cout << endl;}
    else if(t=='C') {cout << "C |"; printStack(aux); cout << endl;}
    else if(a=='C') {cout << "C |"; printStack(to); cout << endl;}
}
```

```
int main()
{
    stack<int> s,d,e;
    int n;
    cin >> n;
    for(int K=n; K>=1; K--)
        s.push(K); //cerco s con n dischi (dal più grande al più piccolo)
    cout << "Lo stato iniziale e' " << endl;
    cout << "A " << endl;
    printStack(s);
    cout << "\n ";
    cout << "B " << endl;
    printStack(e);
    cout << "\n C " << endl;
    printStack(d);
    cout << endl;
    TowerOfHanoi(n,s,d,e,'A','B','C');
}
```

CLASSE MATRICE

`vector<vector<int>> m;` // PROBLEMA: Ogni riga puo' avere dimensione variabile

Vogliamo definire una matrice che abbia un numero fisso di righe e colonne, e le possibilita' di gestire matrici simmetriche

template <class T>

class matrix

{ protected: //perche' vogliamo ereditare

int rows, cols;

vector<vector<T>> met;

public:

matrixx () { rows=0; cols=0; }

matrixx (int r, int c, const T& initial) //VALORE CON CUI INIZIALIZZARE / Quando lo ricambiemo met e già stata costruita

{

rows=r;

cols=c;

met.resize(r); //sono sul vector esterno

for (int K=0; K<=met.size()-r; K++)

met[K].resize(c, initial); //SFRUITATO IL COSTRUTTORE DI VECTOR

}

~matrixx (const matrixx<T>& m)

{

rows = m.rows;

cols = m.cols;

met = m.met; //SFRUTTIAMO L'OPERATORE DI ASSEGNAZIONE DELLA CLASSE VECTOR

}

int r.size() const {return rows;}

int c.size() const {return cols;}

virtual void resize(int r, int c)

{

rows=r; cols=c;

met.resize(r);

for (int K=0; K<=met.size()-r; K++)

met[K].resize(c);

}

virtual T& operator () (int row, int col) //posso modificare, mi restituisce un riferimento (sole letture -> const T&)

{ assert (row < rows & col < cols); //controlla se row e col sono validi

return met[row][col]; }

```

void print()
{
    for (int i=0; i<rows-1; i++)
    {
        for (int j=0; j<cols-1; j++)
            cout<<(*this)(i,j) << " ";
        cout<< endl;
    }
}

```

CLASSE MATRICE SIMMETRICA (vale sempre  $m(i,j) = m(j,i)$ )

```

template <class T>

class SymmetricMatrix: public matrix<T>
{
    // EREDITA E TEMPLATE

    using matrix<T>::rows; // con rows mi riferisco a rows di Matrix

    OPPURE matrix<T>::rows ogni volta che mi serve rows

    OPPURE this->rows ogni volta che mi serve rows

    using matrix<T>::cols;

    using matrix<T>::met;

    public:

    SymmetricMatrix(int _rows, const T& initial)
    {
        rows=_rows; cols=_rows;

        met.resize(_rows);

        for (int i=0; i<_rows-1; i++)
            met[i].resize(i+1, initial);
    }

    T& operator()(int row, int col)
    {
        assert(row<rows && cols<col);

        if (row<col) return met[col][row];

        else return met[row][col];
    }
}

```

USO NEL MAIN

```

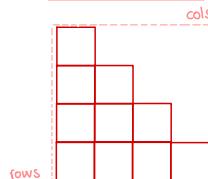
int main()
{
    SymmetricMatrix<int> m(4,0); int cont=0;

    for (int i=0; i<4; i++)
        for (int j=0; j<i; j++)
            m(i,j)= cont; cont++;

    for (int i=0; i<4; i++)
        for (int j=i; j<4; j++)
            cout<< m(i,j) << " ";
}

```

MATRICE SIMMETRICA



		cols
0	5 2 3	
5		
2		
3		

$m(i,j) = m(j,i)$  → quando scrivo in uno dei due  
cambio anche l'altro. → SOVRASCRIUTO

## LE FUNZIONI HASH

Permettono di associare ad una stringa alfabetica un numero (o chiave). Queste funzioni possono essere perfette o non perfette.

\* Funzioni hash perfette non ho problemi di sovrascrizione, ma ho bisogno di molto spazio;

\* Funzioni hash non perfette ho problemi di sovrascrizione => **Collisioni**: due stringhe restituiscono lo stesso numero

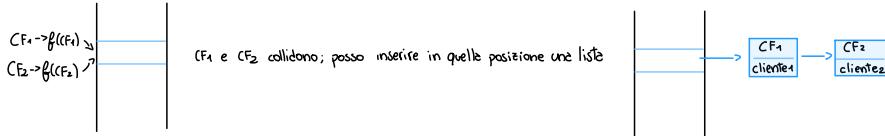
M R O R S S 8 E 2 D O 8 6 K  
 FF 82 99 82  
 / NOTAZIONE POSIZIONALE BASE 36 (26 LETTERE + 10 NUMERI)  
 ↴ CODICE ASCII  
 => codice · 36 posizione => ho un codice univoco, ma e' troppo grande

## HASH MAP

**unordered-map** Struttura dati che permette di mappare una coppia <chiave, valore>; l'ordine e' casuale.

**ordered-map** Uguale me le strutture dati che sta sotto viene mantenuta ordinata. Non ho sotto le hash, uso altre strutture come gli alberi che non fanno ricerche  $O(r)$ .

## RISOLUZIONE DELLE COLLISIONI



La complessità di ricerca adesso dipende dal numero di collisioni che dipendono delle qualità della funzione hash usata.

## TIPI DI FUNZIONI HASH

1) **Metodo del modulo** non e' una funzione hash generica. Viene applicata una qualsiasi funzione hash che restituisce un numero e a questo applico il modulo.

val hash % ncelle ho una garanzia sull'intervallo dei valori, ma mi aggiunge collisioni

2) **Metodo del quadrato centrale** funzione con la rappresentazione binaria

$$K = \text{Velocità binario chiave} \quad \bullet \text{calcoliamo } K^2$$

$$p = \text{numero di celle} \quad \bullet \text{estremo } m \text{ bit centrali de } K^2$$

$$m = \lfloor \log p \rfloor$$

3) **Metodo dell'aggiungimento**

$$m = \log p$$

Potrebbe essere utile ordinare il vettore delle collisioni in modo da poter usare le ricerche binarie.

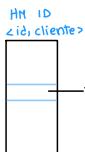
Un' proprietà di una tabella hash viene chiamata **RAPPORTO DI CARICO**  $f_c = m / m_{celle}$

se  $f_c$  e' alto le probabilità di collisione e' alta, ma sto usando poco spazio (ho più celle che spazio); al contrario ho poche collisioni.

$f_c$  0,2 0,4 1,5 → percentuale riempimento delle strutture dati.

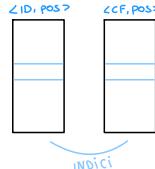
maccessi media 1,1 1,45 1,75 Possiamo dire che abbiamo accesso alle liste con  $O(1)$  anche con elementi non ordinati.

## BIBLIOTECA



Poco efficiente, ricopio 3 volte i clienti

## SOLUZIONE



Non uso il cliente, bensì la posizione per creare degli indici. LINEARE

Mi viene restituita la posizione all'interno del vettore `clienti` applicando l'hash map su ID o CF.

Accesso diretto => Accesso O(1)

Le posizioni e' fondamentale, non ho bisogno di ordinare mai una volta che inserisco un cliente la sua posizione non puo' cambiare.

## CODICE

```
#include <unordered_map>

using namespace std;

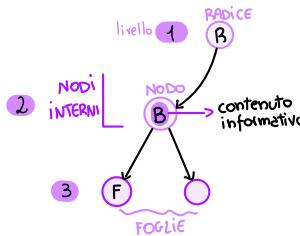
int main()
{
    unordered_map<string, string> mymap;
    mymap["Bakery"] = "Berbere"; //non uso gli indici ma le chiavi
    // nelle parentesi devi usare le chiavi

    string name = mymap["Bakery"]; //se esiste l'elemento con chiave Bakery, name diventa il valore, altrimenti NULL.

    string name2 = mymap["Terrecine"]; //non c'e', name2 = ""; crea nell'hash map una coppia <"Terrecine", "">
    if(name2 == "") //visto che ho creato uno spazio vuoto lo assegno è un valore
        mymap["Terrecine"] = "Giorgio";
}
```

GLI ALBERI

Gli alberi sono delle strutture dati gerarchiche (es. albero genealogico) basate sulla nozione di Nodi e Archi.

TIPI DI ALBERI

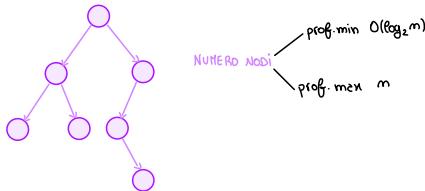
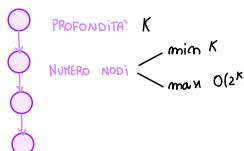
- Binari (per ogni nodo al massimo due figli);
- d-Ari (per ogni nodo massimo d figli);
- Generali (non c'e' un massimo).

Ciascun nodo, a parte le radice, ha esattamente un padre.

LIVELLO DEI NODI Il livello del nodo radice è 1, tutti gli altri hanno livello +1; numero di nodi da attraversare per arrivare alla radice

PROFOUNDITA' È il massimo fra tutti i livelli dei nodi (ultimo livello). Se l'albero è vuoto è uguale a zero.

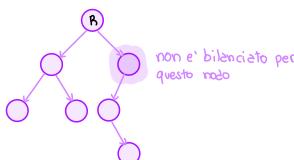
Gli alberi, al contrario delle altre strutture dati che sono lineari, non lo sono poiché ad ogni passo nell'albero il percorso può cambiare in base alle scelte che viene fatta.

GLI ALBERI BINARIALBERO DEGENEREALBERO PIENOTIPOLOGIE DI ALBERI

**Pieno:** Tutti i nodi hanno due figli, tranne l'ultimo livello.

**Completo:** Tutti i livelli tranne l'ultimo e il penultimo sono pieni; l'ultimo è riempito da sinistra a destra

**Bilanciato:** Se per ogni nodo  $m$ ,  $|\text{prof. sottoalbero sinistro di } m - \text{prof. sottoalbero destro di } m| \leq 1$  (dove volete per ogni nodo).

ESEMPIO

## RAPPRESENTAZIONE DEGLI ALBERI

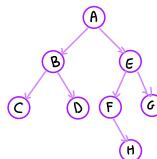
- Indicizzate; usano strutture statiche (es reg);
- Collegate; usano strutture dinamiche con puntatori etc.

## RAPPRESENTAZIONI INDICIZZATE

- VETTORE DEI PADRI

		PARENT	INFO			
0	1	2	3	4	5	

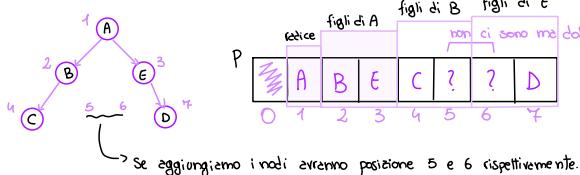
-1	0	1	1	0		
A	B	C	D	E		



**PRO** Rappresento un albero (anche generico, non solo binario) in una struttura semplice; se gestisco bene gli inserimenti non ho spreco di spazio.

**CONTRO** Scoprire chi sono i figli di un nodo ha un costo elevato; nel peggiore dei casi devo scorrere tutto l'array.

- VETTORE POSIZIONALE



Detto un nodo  $V$  su un albero d-Albero per accedere al figlio  $i$  di  $V$  posso usare  $P[d \cdot V + i]$

BIN  $d = 2$     Nodo  $V=1$  (A)     $P[2 \cdot 1 + 0]$ ;    Nodo  $V=3$      $P[2 \cdot 3 + 1]$ ;    PADRE  $[V/d]$

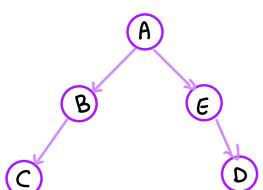
**PRO** Ho una regole precisa per trovare facilmente i figli di un nodo  $m$ .

**CONTRO** Non va bene per gli alberi generici; ho bisogno di uno spazio riservato grande quanto le profondità. Se i nodi sono vuoti devo comunque riservargli lo spazio  $\rightarrow$  SPERCO.

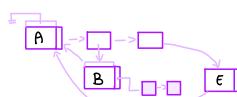
## RAPPRESENTAZIONI COLLEGATE

Non spreco spazio ma ho un over-head di puntatori.

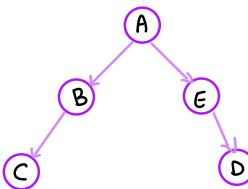
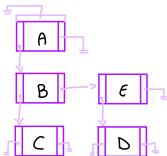
- RAPPRESENTAZIONE CON PUNTATORI AI FIGLI



- RAPPRESENTAZIONE CON LISTA DI FIGLI

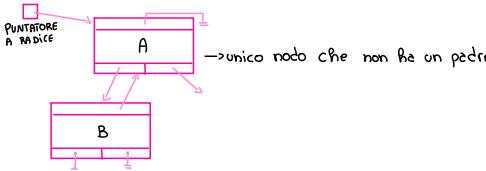


- RAPPRESENTAZIONE PRIMO FIGLIO / FRATELLO



Posso gestire alberi generali ma perdo l'eccesso diretto ai figli. Non spreco spazio, ci sono solo i nodi che esistono.

Il concetto di iteratore e' difficile da gestire; il BEGIN e' sicuramente la radice, ma non so cos'e' l'END. Per gestire gli alberi quindi utilizzeremo le NOTAZIONI FUNZIONALI definendo ogni nodo come radice di un sottoalbero.

NOTAZIONE CON PUNTATORI AI FIGLIIDEFINIZIONE ALBERI

```
enum Direzione {sin=0, des=1}; //enumetativi per identificare numero e direzione
template <class U>
struct SNodo
{
    U vinfo;
    SNodo* pPadre, pFiglio[2];
    SNodo(const U& inf) : vinfo(inf) // COSTRUTTORE DI DEFAULT
    {
        pPadre = pFiglio[0] = pFiglio[1] = 0;
    }
    ~SNodo() { delete pFiglio[sin]; delete pFiglio[des]; } //grazie a enum posso scrivere così
}
```

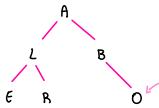
```
template <class T>
class AlberoB
{
    /* se SNodo ci serve solo in albero possiamo dichiararla qui come privata. */
protected:
    SNodo<T> * predice;
public:
    AlberoB() : predice(0) {};
    AlberoB(const T& a) //inizializza la radice ed è
    {
        predice = new SNodo<T>(a);
    }
    void insFiglio(Direzione d, AlberoB* AC) //per inserire un intero sottoalbero
    {
        assert(!nullo());
        assert(figlio(d) nullo());
        if(!AC.nullo())
        {
            predice -> pFiglio[d] = AC;
            AC.predice -> pPadre = predice;
        }
    }
}
```

```

bool nullo() const {return radice==0;} //controlla se la radice e' nulla
bool foglie() const //controlla se il nodo e' una foglia
{
    return !nullo() && figlio(sin).nullo() && figlio(des).nullo(); }
AlberoB<T> figlio(Direzione d) const //restituisco per copie senza problemi poche' copio solo la radice
{ assert(!nullo());
AlberoB<T> AC;
AC.radice = radice -> pfiglio[d]; =>
return AC;
}
} // chiedo il figlio sinistro di O che non esiste
        // mi restituisce un albero null

const T& radice() const; //contenuto informativo radice
AlberoB<T> padre() const; //padre del nodo
void modRadice(const T& e); //modifica contenuto nodo radice
AlberoB<T> estraiFiglio(Direzione d); //sgancia il nodo dell'albero e restituisce un nuovo albero con il nodo estratto come radice
AlberoB<T> copia() const; //copia di un albero
void svuota(); //svuota l'albero

```



// chiedo il figlio sinistro di O che non esiste

→ mi restituisce un albero null

### USO DELLA CLASSE ALBERO

```

int main()
{
    AlberoB<char> A1('A'); //albero di char con radice=A
    AlberoB<char> A2('B');
    A1.insFiglio(des, A2);
}

```

**COME CALCOLARE LA PROFONDITÀ DI UN ALBERO** - livello massimo dei nodi di un albero  $O(m)$

**DEFINIZIONE RICORSIVA** Massimo delle profondità dei miei figli +1. Vedo dal basso verso l'alto.

Proviamo su un albero di int.

```

int profondita'(AlberoB<int> A) //VALUTAZIONE PRE-ORDINE
{
    if (A.nullo())
        return 0;
    else
    {
        int p1 = profondita'(A.figlio(sin));
        int p2 = profondita'(A.figlio(des));
        if (p1 > p2) return p1+1; } OPPURE return (p1 > p2)? p1+1 : p2+1;
        else return p2+1;
}

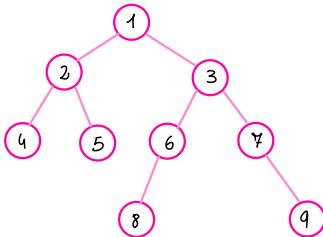
```

## COME VEDERE SE UN ALBERO E' BILANCIAUTO

```
bool bilanciato(AlberoB<int> A) // VALUTAZIONE POST-ORDINE
{
    if (A.nullo())
        return true;
    else
    {
        int p1 = profondite(A.figlio(SIN));
        int p2 = profondite(A.figlio(DES));
        return (abs(p1 - p2) <= 1 && bilanciato(A.figlio(SIN)) && bilanciato(A.figlio(DES)));
    }
}
```

PER CASA bool BilanciatoProfondo(AlberoB<int> A, int& p)

## VISITE SUGLI ALBERI



- PRE-ORDINE (anticipate) : 1 -> 2 -> 4 -> 5 -> 3 -> 6 -> 8 -> 7 -> 9
- POST-ORDINE (posticipate) : 4 -> 5 -> 2 -> 8 -> 6 -> 9 -> 7 -> 3 -> 1
- SIMMETRICA : 4 -> 2 -> 5 -> 1 -> 8 -> 6 -> 3 -> 7 -> 9
- PER LIVELLI : 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9

### 1) VISITA SIMMETRICA

```
void visitaSimmetrica(AlberoB<int> A)
{
    if (A.nullo())
        return;
    else
    {
        visitaSimmetrica(A.figlio(SIN)); // visita e sinistra
        cout << A.radice() << " "; // veluto
        visitaSimmetrica(A.figlio(DES));
    }
}
```

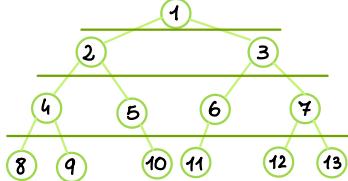
### 2) VISITA POST-ORDINE

```
void visitaPosticipate(AlberoB<int> A)
{
    if (A.nullo())
        return;
    else
    {
        visitaPosticipate(A.figlio(SIN)); // visita e sinistra
        visitaPosticipate(A.figlio(DES));
        cout << A.radice() << " "; // veluto
    }
}
```

### 3) VISITA PRE-ORDINE

```
void visitaAnticipata (AlberoB<int> A)
{
    if (A.nullo())
        return;
    else
    {
        cout << A.radice() << " "; //veluto
        visitaSimmetrica (A.figlio(SIN)); //visita a sinistra
        visitaSimmetrica (A.figlio(DES));
    }
}
```

PER CASA Calcola somme di tutti i nodi di un albero

VISITA PER LIVELLI - VISITA IN AMPIEZZA

La struttura dati per visitare un albero secondo la visita per livelli è LA CODE (costruiremo una code di alberi).

- inserisco 1; 1
- lo estrarro e inserisco i figli; 1 | 2 | 3
- veluto 2 , lo estrarro e inserisco i suoi figli; 1 | 2 | 3 | 4 | 5
- veluto 3, lo estrarro e inserisco i figli etc... 1 | 2 | 3 | 4 | 5 | 6 | 7

Usando un'altra code o una code di pair posso salvarmi anche il livello a cui si trova il nodo.

code dei livelli di ogni nodo

1	2	2	3	3	3	3
---	---	---	---	---	---	---

CODICE PER STAMPA

```

#include <queue>

typedef AlberoBint > AlberoBint; // modo per dare un nome diverso a un certo tipo: AlberoBint > viene identificato da AlberoBint (risp alias)

void visitaPerLivelli(AlberoBint > a)
{
    if(a.nullo()) return;
    std::queue < AlberoBint > q;
    q.push(a);
    while (!q.empty())
    {
        AlberoBint Temp = q.front();
        q.pop();
        cout<<Temp.radice()<< " "; // visualizzazione radice corrente estratta
        if(!Temp.figlio(SIN).nullo()) // se i figli del nodo corrente non sono nulli li aggiungo alla code
            q.push(Temp.figlio(SIN));
        if(!Temp.figlio(DES).nullo())
            q.push(Temp.figlio(DES));
    }
}
  
```

## FUNZIONE CERCA - $O(m)$ caso peggiore, $O(1)$ caso migliore

Scrivere una funzione `CERCA` che dato un albero binario `int` e un valore `int`, mi restituisce l'albero che ha come radice il valore se presente, oppure un albero `NULL`.

```

AlberoB<int> cerca (AlberoB<int> a, int v)
{
    if ((a.radice() == v) || (a.nullo())) return a;
    AlberoB<int> Tmp = cerca (a.figlio(SIN), v);
    if (!Tmp.nullo()) return Tmp;
    else return cerca (a.figlio(DES), v);
}

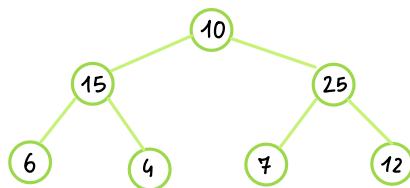
```

## INPUT DA FILE TESTO

WIN Type `input.txt` | `programma.exe` per windows

LINUX cat `input.txt` | `./programma`

## INPUT DI UN ALBERO



\* 10 // limitazione -> i valori devono essere univoci perché identifico  
 15: 10 s il nodo con il valore  
 25: 10 d  
 6: 15 s // quando inserisco devo cercare il nodo padre  
 4: 15 d => mi serve la funzione `cerca`  
 7: 25 s  
 12: 25 d  
 -1 // TAPPO

## CODICE

```

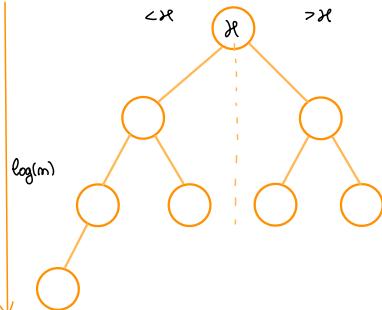
int main()
{
    string Temp;
    bool firstline=true;
    AlberoB<int> A(42); // costruisco A con nodo-radice 42 (velose è caso, serve solo a creare il nodo)
    while (getline(cin, Temp)) // ve' eventi finche leggo input
    {
        if (Temp == "-1")
            break; // esco dal while, ho letto il resto tutto
        if (firstline)
        {
            int v = stoi(Temp); // stoi converte da string a int
            A.modRadice(v);
            firstline=false;
            continue; // salta tutto il resto e ritorna all'inizio del while
        }
    }
}

```

```

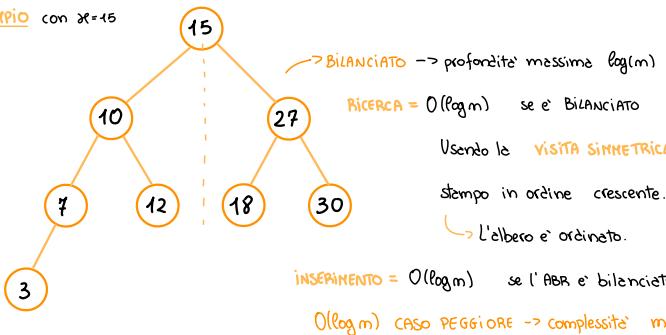
// l'input deve essere ben formato *
int indiceDuePunti = Temp.find(':''); // restituisce le posizioni di ':' nelle stringa
int indiceSpazio = Temp.find(' ');
string parteSinistra = Temp.substr(0, indiceDuePunti);
string parteDestra = Temp.substr(indiceDuePunti+1, indiceSpazio- indiceDuePunti);
Direzione dir = SIN; //enum
if(Temp[indiceSpazio+1] == 'd')
    dir = DES;
int valoreNodo = stoi(parteSinistra);
int valorePadre = stoi(parteDestra);
AlberoB<int> padreInCuiInserire = cerca(A, valorePadre);
if(!padreInCuiInserire.nullo())
{
    AlberoB<int> delInserire(valoreNodo);
    padreInCuiInserire.insFiglio(dir, delInserire);
}
} // chiude il while
}

```

ABR-ALBERI BINARI DI RICERCA

(mep-&gt; basate su alberi binari di ricerca)

Ordiniamo l'albero in base al nodo radice; e desira i valori maggiori di  $x$ , e sinistre quelli minori. In questo modo posso avere una ricerca più efficiente, scendendo completamente, già del primo confronto, metà albero. Con ogni nodo scendo di un livello.

ESEMPIO con  $x=15$ 

- $m \rightarrow \log(m)$  (PIENO, COMPLETO, (BILANCIATO))
  - $m \rightarrow m$  (DEGENERE)
- Le proprietà più importanti sono il BILANCIAMENTO.

VANTAGGI

- Rispetto alle tabelle hash gli ABR sono ordinati;
- Rispetto ai vettori, l'albero è più efficiente per l'inserimento e la cancellazione, e' una struttura molto dinamica; anche nell'ordinamento convengono gli ABR (che sono già ordinati).

La classe AlberoBinarioRicerca eredita dalla classe AlberoB; aggiungiamo semplicemente un metodo di inserimento che si occupa di rispettare l'ordine, non permette all'utente di aggiungere liberamente (come in insFiglio).

```
class ABR: protected AlberoB // decido dopo cosa mostrare all'utente (non devono essere visibili i metodi di modifica di AlberoB)
    OK per i metodi const che non modificano l'albero.
```

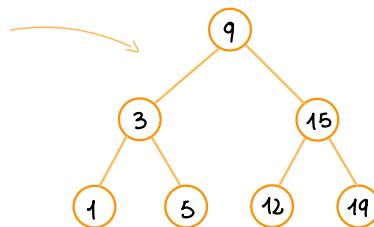
## DA VETTORE AD ABB



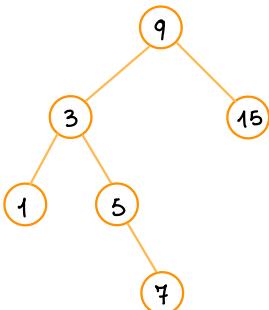
→ ARRAY ORDINATO

9 sarebbe il mio  $\pi$  che divide l'albero a "metà".

USANDO LA DIVIDE ET IMPERA



## GLI ALBERI BILANCIATI E SBILANCIATI



Potrebbe verificarsi una situazione simile → l'albero diventa **SBILANCIATO**.

In questo caso è necessario ribilanciarsi → manutenzione della classe **albero** perdo la ricerca a costo  $O(\log n)$ .

Esistono diversi modi per ribilanciare un albero, passo, per esempio, con delle regole girare i nodi.

→ Ogni **TOT** di inserimenti controllo se l'albero è bilanciato.

Non posso farlo ad ogni inserimento perché ha un costo elevato.

## I GRAFI

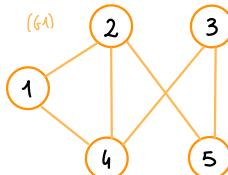
Sono strutture definite su nodi e archi.  $\Rightarrow G = \langle N, A \rangle$

Non c'è un ordinamento e non esiste gerarchia padre-figlio. Un arco può essere descritto dai nodi che collega, se non c'è ordine il grafo non è orientato.

I grafici si distinguono in:

- **NON ORIENTATI**
- **ORIENTATI**

### • **NON ORIENTATO**



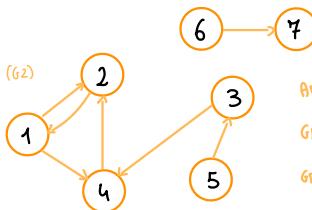
**ADJACENTI** → nodi raggiungibili

**GRADO** → numero di archi che incidono sul nodo (o di adiacenti)

num. minimo archi =  $\emptyset$

num. massimo archi =  $O(n^2)$

## • ORIENTATO



**ADJACENTI** → nodi raggiungibili

**GRADO DI ENTRATA** → archi entranti

**GRADO DI USCITA** → archi uscenti

num. minimo archi = 0

num. massimo archi =  $O(m^2)$ , tutti i nodi sono collegati tra di loro

Tutti i nodi sono importanti, non esiste "radice" - nodo eleggibile. => posso identificare un nodo con un numero

L'ordine di grandezza degli archi è quadratico rispetto ai nodi, e prescindere.

**GRAFO SPARSO**  $O(m)$  Quando ogni nodo mediamente è collegato a un solo nodo.

**GRAFO DENSO**  $O(m^2)$

**CAMMINO IN UN GRAFO** è una sequenza di archi consistenti che collegano due nodi.

•  $(G_1) = (1,2)(2,5)(5,3)$

•  $(G_2) = (1,2)(2,1)$  → **ciclo**, è un cammino che mi permette di tornare al nodo di partenza.

$(1,4)(4,2)(2,1)$  **CICLO**

$(5,3)(3,4)$

La lunghezza di un cammino è determinata dal numero di archi attraversati.

## PROPRIETÀ

①

**GRAFO CONNESSO** Un grafo non orientato si dice connesso se da un nodo posso raggiungere tutti gli altri, per i grafici non orientati.

**GRAFO FORTEMENTE CONNESSO** Se per il grafo orientato vale le proprietà ①

**GRAFO DEBOLMENTE CONNESSO** Se la versione non orientata del grafo orientato è connesso.

**COMPONENTI CONNESSE** Sono sottosinsiemi connessi in un grafo.

**CHIUSURA TRANSITIVA** Dato un grafo  $G = \langle N, A \rangle \rightarrow G^+ = \langle N, A^+ \rangle$ ; aggiungo al grafo un arco diretto da un nodo che era prima raggiungibile con un cammino  $> 1$ .

Gli alberi sono un caso particolare di grafo orientato connesso, senza cicli in cui esiste un unico cammino elementare per ogni coppia di nodi.

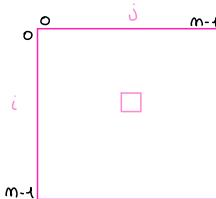
Nei grafici il contenuto informativo può stare sia sui nodi che sui archi. Se associo un'informazione anche agli archi ho un **GRAFO PESATO**.

## STRUTTURA DATI DI RAPPRESENTAZIONE

I nodi possono essere rappresentati da una struttura statica → il numero di nodi m è dato e non cambia.

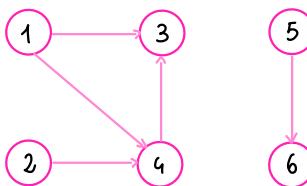
RAPPRESENTAZIONE DEI GRAFI

E' possibile rappresentare i grafici con una matrice di booleane.



$$G(i,j) = \begin{cases} \text{false} \\ \text{true} \end{cases}$$

e true se esiste l'arco tra i e j, false altrimenti

ESEMPIO

SPAZIO  $O(m^2)$  m nodi

$G(i,j) \rightarrow O(1)$  esiste l'arco  $(i,j)$

PROSSIMO ADIACENTE  $O(m)$

nodo di arrivo						
0	1	2	3	4	5	6
0						
1					T	T
2					T	
3						T
4						
5						T
6						

T=true

gli altri sono F

Per i nodi non c'e' bisogno di alcune strutture di rappresentazione se non ho contenuto informativo. Se ho contenuto informativo devo prevedere un vettore che lo contenga.

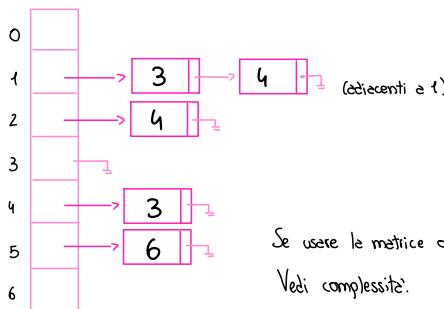
Se gli archi sono pesati posso usare una matrice di peir, una nuova matrice o il solo valore del peso.

**PRO** Ho accesso diretto alla matrice (posso vedere se  $(i,j)$  sono collegati facilmente).

**CONTRO\*** Se il grafo e' SPARSO, molti valori sono False, non hanno senso e sprecano spazio.

Scoprire se due nodi sono adiacenti costa  $O(m)$ , devo scorrere tutte le righe.

Per non sprecare spazio \* posso usare le LISTE DI ADIACENZA.



SPAZIO  $O(m) \rightarrow m:m$  archi

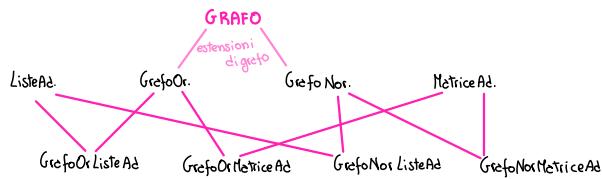
$G(i,j) \rightarrow O(m)$  esiste l'arco  $(i,j)$

PROSSIMO ADIACENTE  $\rightarrow O(1)$

Se uso le matrici o le liste dipende dalle azioni che devo fare.

Vedi complessità.

## DIVERSE RAPPRESENTAZIONI



## CLASSE GRAFO

```
class Grafo
{
protected:
    int vn, vm;
    Matrice<bool> archi;

public:
    Grafo (int n);
    bool operator () (int i,int j) const; // per usere G(i,j)
    void operator () (int i,int j, bool b); // per aggiungere o eliminare archi con G(i,j,True)
    int n(); //num. nodi (get vn)
    int m(); //num. archi (get vm)
}
```

## ESEMPIO

Dato un grafo non orientato calcoli il grado di ogni nodo (il numero di nodi adiacenti).

non uso `list<int>` perché altrimenti potrei perdere il riferimento diretto ai nodi esaminati

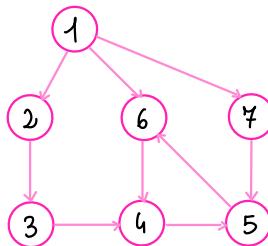
```
vector<int> numAdiacenti (const Grafo& G) // se il grafo e' rappresentato da una matrice O(m2), con liste O(m3)
{
    vector<int> grado(G.n()); //n = numero di nodi in G
    for (int i=0; i<=G.n()-1; i++) //scorro i nodi di G
    {
        grado[i]=0;
        for (int j=0; j<=G.n()-1; j++) //scorro i potenziali adiacenti al nodo i
            if (i!=j && G(i,j)) //escludo gli autoanelli
                grado[i]++;
    }
    return grado;
}
```

## ESEMPPIO

Disorientare un grafo.

```
void disOrienta(const Grefo& G, Grefo& G1) // G è orientato e vogliamo che in G1 ci siano il corrispondente non orientato
{
    assert(G.n() == G1.n());
    (G1.svuota());
    for(int i=0; i <= G.n()-1; i++)
        for(int j=0; j <= G.n()-1; j++)
            if(i!=j && G(i,j)) // ogni volta che in G esiste l'arco (i,j)
                {
                    G1(i,j,true); // aggiungo in G1 l'arco (i,j)
                    G1(j,i,true); // e l'arco (j,i) (entrambi i versi)
                }
}
```

## LE VISITE SUI GRAFI



### ORDINE DI VISITA

• DFS = 1 2 3 4 5 6 7

mi devo ricordare quelli nodi che ho visitato o potrei ritrovarmi in un loop se considero solo gli adiacenti: 1 2 3 4 5 6 4 6 4 5 etc..

A seconda del nodo di partenza ho un numero di nodi visitati diverso

• BFS = 1 2 6 4 3 4 5

cerca tra i nodi più vicini

Possibili visite:

• DFS (profondità)

• BFS (ampiezza)

### CODICE DELLA VISITA DFS

$O(n^2)$  perché i visitati li ricontrollo comunque

```
void dfs(int s, const Grefo& G, vector<bool>& visitati) // s=nodo di partenza , visitati = nodi visitati
{
    visitati[s] = true;
    for(int j=0; j <= G.n()-1; j++) // cerco gli adiacenti a s
        if(s!=j && G(s,j) && !visitati[j]) // adiacente diverso da s , che esiste e che non lo già visitato
            dfs(j, G, visitati); // visitati si aggiorna ricorsivamente ogni volta gli passo l'adiacente al nodo di partenza precedente
}
```

## ESEMPIO

Chiusure transitiva di un grafo G. Restituisce le chiusure transitive di G.

Grafo chiusureTransitive (const Grafo & G)

```
{   Grafo   Gc (G.n()); //Gc ha n nodi (come G)
    vector<bool> visitati (G.n());
    for (int i=0; i<=G.n()-1; i++)
    {
        visitati. assign (visitati.size(), false); // tutti i valori di assegnati vengono assegnati a false visitati=false ogni volta che chiamo dfs
        dfs(i, G, visitati);
        for (int j=0; j<=G.n()-1; j++)
            if (i!=j && visitati[j]) //se raggiungo j lo aggiungo a Gc (aggiungo l'arco (i,j))
                Gc(i,j,True);
    }
    return Gc;
}
```

X CASA Scrivi la funzione bfs che restituisce il nodo da cui siamo arrivati (es. visito 2 partendo da 1, restituisco 1, altrimenti restituisco -1).

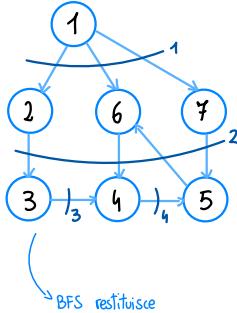
VISITA BFS

```

vector<int> bfs (const Graph & G , int s)
{
    vector<bool> visitati (G.n(), false); //inizializzo a falso
    vector<int> P (G.n(), -1); //vector di predecessori (-1 se non c'è predecessore)

    queue<int> q; //prossimi adiacenti da valutare
    q.push(s);
    visitati[s] = true;
    while (!q.empty()) //finché ho nodi da valutare vado avanti
    {
        int u = q.front();
        q.pop();
        //Valuta u
        for (int v=0; v<=G.n()-1; v++)
            if (u!=v && G(u,v) && !visitati[v]) //per ogni adiacente
            {
                q.push(v);
                visitati[v] = true;
                p[v] = u; //il predecessore del nodo v è u
            }
    }
    return p;
}

```



BFS restituisce

-1	-1	1	2	6	7	1	1
0	1	2	3	4	5	6	7

COSTO BFS  $O(m^2)$ CAMMINI MINIMI CON BFS

Ricostruisce il cammino minimo da  $s$  a  $t$  dato il vettore di predecessori  $p$  restituito da `bfs`, in modo ricorsivo.

```
void ricostruisciCammino(int s, int t, const vector<int> & p, queue<int> & cammino)
```

```

{ if (s==t)
    cammino.push(s);
    else if (t== -1)
        return;
    else
        { ricostruisciCammino(s, p[t], p, cammino)
        cammino.push(t);
    }
}
```

Il cammino minimo viene restituito  
delle queue `cammino` (infatti viene  
passato per riferimento).

## CAMMINO DA S A T RICORSIVO CON DFS

Restituisci un cammino da s e t usando le visite in profondità. Funzione ricorsiva.

```
bool TrovaCammino (cont Grefo& G, int s,int t, vector<bool>& visitati, list<int>& cammino)
```

```
{ if(s==t) //sono già su t
```

```
{ visitati[t]=True;
```

```
return True;
```

```
}
```

COSTO O(m<sup>2</sup>) nel caso peggiore

```
else
```

```
{ visitati[s]=true;
```

```
bool Trovato=false;
```

```
for(int j=0, j<=G.n()-1 && !Trovato, j++) //visita in profondità; entra nel for se ancora non ha trovato un cammino
```

```
if(s!=j && G(s,j) && !visitati[j]) //se j ha adiacenti
```

```
{ cammino.push_back(j);
```

```
Trovato = TrovaCammino (G, j, t, visitati, cammino);
```

```
if(!Trovato) //se non ha trovato un cammino devo togliere j dal cammino perché non mi permette di arrivare a t
```

```
cammino.pop_back();
```

```
}
```

```
return Trovato;
```

```
}
```

```
}
```

## ESERCIZIO SIMIL-ESAME

Scrivere un funzione che prende in input un grafo orientato e pesato G, dove ogni nodo in G ha associato un valore intero positivo chiamato deposito. Deve restituire "Yes" se per ogni nodo v del grafo vengono le seguenti condizioni:

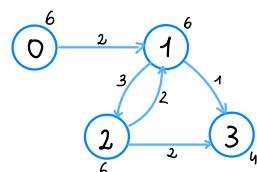
- la somma dei pesi degli archi entranti in v è minore del suo deposito;
- il deposito di ogni nodo è minore o uguale del deposito di v.

Se un nodo non ha archi entranti le condizioni si assumono verificate.

### FUNZIONI INTEGRATE

$G(i,j)$  restituisce il peso dell'arco, Ø se l'arco non esiste.

$gdep(i)$  restituisce il deposito associato al nodo i.



```
for(int i=0 ; i<=G.n()-1; i++)
```

```
for(int j=0; j<=G.n()-1; j++)
```

$G(j,i)$  archi j entranti in i (i rappresenta il nodo v delle tracce)

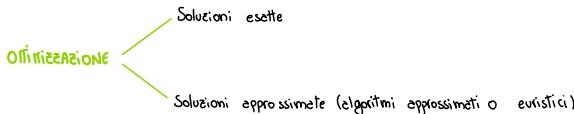
② for (int i=0; i<=G.n()-1; i++)  
for (int j=0; j<=G.n()-i; j++)  
if (i!=j && G(j,i)!=0)  
if (g.dep(j) < g.dep(i))  
ESCI

① for (int i=0; i<=G.n()-1; i++)  
{ somme=0;  
for (int j=0; j<=G.n()-i; j++)  
if (i!=j && G(j,i))  
somme+=g(j,i);  
if (somme >= g.dep(i)) ESCI

①+2 for (int i=0; i<=G.n()-1; i++)  
{ somme=0;  
for (int j=0; j<=G.n()-i; j++)  
if (i!=j && G(j,i))  
if (g.dep(j) < g.dep(i)) ESCI  
somme+=g(j,i);  
if (somme >= g.dep(i)) ESCI  
}

TIPI DI PROBLEMI

- Decisionali (del tipo Si-No);
- Ricerche di una soluzione (se esiste), anche l'ordinamento rientra in questa tipologia;
- Ottimizzazione . si vuole cercare la migliore soluzione esistente;

TECNICHE DI RISOLUZIONE

- Divide et impere (cond necessaria -> il problema deve poter essere diviso in più sottoproblemi), è applicabile sicuremente su problemi decisionali e di ricerca;
- Tecnica golosa (greedy) adatta a problemi di ottimizzazione, spesso è usata per cercare soluzioni euristiche -> TECNICA EURISTICA;
- Programmazione dinamica;
- Backtracking adatta a problemi di ricerca;
- Brute force (quando non ho altre tecniche da applicare).

TECNICA GOLOSA  $O(m^2)$  per valori e costi dinamici;  $O(m)$  per valori e costi statici con dati ordinati (valori specifici).

Adatta ad insiemi di elementi che hanno un valore (statico o dinamico) e un costo (statico o dinamico).

$$\left\{ \begin{array}{l} \max F: \sum_i x_i \cdot v(x_1 \dots x_m) \\ C(x_1 \dots x_m) \leq C_{\max} \rightarrow \text{Vincoli su costi} \\ 0 \leq x_i \leq q_i \quad i=1 \dots m \end{array} \right.$$

$$\left\{ \begin{array}{l} \min F: \sum_i x_i \cdot c(x_1 \dots x_m) \\ V(x_1 \dots x_m) = V_{\min} \\ 0 \leq x_i \leq q_i \quad i=1 \dots m \end{array} \right.$$

BRUTE FORCE genera tutte le combinazioni possibili delle variabili, controlla quelli soddisfano i vincoli e si conserva la combinazione migliore. Ha complessità esponenziale (combinare  $m$  elementi tra di loro).

Le tecniche golose risolvono il problema per passi -> cerca di attribuire un valore alle variabili; una volta che viene attribuito un valore, questo non cambia, neanche se mi accorgo che c'è una soluzione migliore. È quasi impossibile che venga trovata la soluzione ottima.

Queste tecniche si basano sul rapporto valore/costo per ottenere il valore specifico (indice di convenienza) per trovare una soluzione.

Per ogni stadio delle scelte golose:

scelte golose -> soluzione migliore per il "sottoproblema" che ho; aggiungi scelta e confrontala con le precedenti.

Divide il problema ogni volta in 2 sottoproblemi o stadi, uno grande  $i$ , l'altro  $m-i$ . A ogni passo (stadio) cerca la soluzione migliore).

## IL CAMBIO DI UN ASSEGNO - TECNICA GOLOSA

Abbiamo un certo valore di assegno. La banca vuole dare meno monete possibili. Si vuole ottimizzare le scelte, sapendo anche i tagli delle monete.

VALORE ASSEGNO 73,55

TAGLI DELLE MONETE 200, 100, 50, 20, 10, 5, 2, 1, 0.5, 0.2, 0.1

Vogliamo massimizzare i tagli, il valore è il valore di ogni taglio. I costi sono il numero di tagli (ogni taglio ha costo 1).

### APPLICAZIONE

Cambiato 0 per 200 e 100, arrivò a 50 e fecce la divisione intera tra 73,55 e 50  $\rightarrow 1$

Cambiato 70 (50+20)

DIVISIONE INTERA TRA RESIDUO E TAGLIO

Cambiato 72 (50+20+2)

200 0 RESIDUO 73,55

Cambiato 73 (50+20+2+1)

100 0 RESIDUO 73,55

Cambiato 73,50 (50+20+2+1+0,5)

50 1 RESIDUO 73,55

Cambiato 73,60 (50+20+2+1+0,5+0,1)

etc.

### CODICE

```
struct Tcambio
{
    vector<int> m_Monetie; // quante banconote abbiamo di quelle monete
    float velore;
    Tcambio(int m) : m_Monetie(m,0), velore(0) {}
}
```

### X CASA

Tcambio cambio(const vector<float>& tagli, float VeloreAssegno)

## KNAPSACK 0-1

$$\left\{ \begin{array}{l} \text{max } \sum_i x_i \cdot v_i \\ \sum_i c_i \cdot x_i \leq C \\ x_i \in \{0, 1\} \end{array} \right.$$

Ho  $m$  variabili da sistemare  $\rightarrow$  BRUTE FORCE provo le combinazioni binari di tutte  $O(2^m)$

**SCelta Golosa** Se c'è spazio prendo l'oggetto, altrimenti no.

- Ordiniamo per valore decrescente il valore specifico;
- Scelta golosa  $\rightarrow$  c'entra? lo prendo, altrimenti no.

Alla fine probabilmente lo zaino non è completamente pieno

Non abbiamo le garanzie di avere la soluzione ottima alla fine dell'algoritmo, infatti:

### CONTROESEMPIO

Cmax	50	Vspec	
V <sub>1</sub>	60 Golosa	c <sub>1</sub> 10 6 ✓	Velore complessivo: 160 non e' ottima
V <sub>2</sub>	100	c <sub>2</sub> 20 5 ✓	
V <sub>3</sub>	120 ottima	c <sub>3</sub> 30 4 X	

→ OTTIMA → riempie anche le bisecce

Per avvicinarmi di più alla soluzione ottima devo chiedermi se esiste un oggetto che riempie le bisecce e ha un valore superiore a tutti gli altri.

Con la tecnica golosa e questo accorgimento ho le garanzie di avere al 50% la soluzione ottima.

### CODICE

```
struct Oggetto {int costo, valore; }

bool greaterThan (Oggetto o1, Oggetto o2)
{
    return (o1.valore / o1.costo) > (o2.valore / o2.costo); } // se è true O1 è più grande di O2

vector<bool> knapsack_01 (vector<Oggetto>& oggetti, int Cmax) // Cmax = capacità max
{
    int n = oggetti.size();
    sort(oggetti.begin(), oggetti.end(), greaterThan); // ordino gli oggetti per valore specifico O(m log m)

    vector<bool> sol(n, false);
    int Cspeso = 0;

    for (int i=0; i < n && Cspeso < Cmax; i++)
    {
        if (Cspeso + oggetti[i].costo <= Cmax)
            { sol[i] = true;
              Cspeso += oggetti[i].costo;
            }
    }
    return sol;
}
```

O(m)

Molto probabilmente non ho però la soluzione ottima, però ho speso m log m (è differenza delle BRUTE FORCE).

PROGRAMMAZIONE DINAMICA - non e' euristica - BOTTOM UP

Cerca di pianificare le ricerche della soluzione finale, risolvendo dei sottoproblemi (come divide et impere, approccio Top Down) in modo Bottom up, parte dai sottoproblemi più semplici. Parte delle soluzioni semplici e le ricombina.

\* Questa tecnica deve necessariamente scorrere tutto lo spazio di ricerca perché parte dal basso. Conviene nei problemi in cui abbiamo bisogno di calcolare tutte le sotto soluzioni e negarci di ricalcolarle. **PRINCIPIO DI OTTIMALITA'**

Abbiamo già usato questo approccio nell'esempio di Fibonacci.

## Fibonacci

## Complessità polinomiale

## RICORSIVA

```
int fib(int n)
{
    if (n == 0)
        return 0;
    else if (n == 1)
        return 1;
    else
        return fib(n-1) + fib(n-2);
}
```

## ITERATIVA

```
int fib(int n)
{
    int f1, f2 = 0;
    for (int i = 0; i < n; i++) {
        f1 = f1 + f2;
        f2 = f1;
        f1 = f2;
    }
    return f1;
}
```

**ESEMPIO DI PROGRAMMAZIONE DINAMICA**  
periamo da fib(0) e fib(1) per risolvere fib(n)

Per applicare questa tecnica è necessario poter dividere il problema in più sottoproblemi.\*

Venne usata nei problemi di ottimizzazione prioritariamente e nei problemi in cui dobbiamo ricercare una soluzione, se esiste.

È una tecnica molto potente ma complicata.

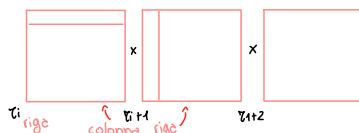
## ESEMPIO - Prodotto tre matrici

Prodotto tre matrici:  $M = M_1 \times M_2 \times \dots \times M_4 \times \dots \times M_{m-1} \times M_m$

Possiamo:  
1)  $M_1 \times (M_2 \times (\dots \times (M_{m-1} \times M_m)))$

Le parentesiizzazione ha complessità esponenziale ( $m^m$ )

2)  $((\dots \times M_{m-1}) \times M_m)$



$O(i_1 \times i_{1+1} \times i_{1+2})$  costo per moltiplicare 2 matrici

$$\begin{array}{ccccc} 1) & M_1 & \times & M_2 & \times M_3 & \times M_4 \\ & 10 & & 20 & & 50 & 1 & 100 \end{array}$$

$$\Rightarrow \begin{array}{ccccccc} M_1 & \times & (M_2 & \times & (M_3 & \times & M_4)) \\ 10 & & 20 & & 50 & & 100 \\ & & & & 5000 & & \\ & & & & 20 \cdot 50 \cdot 100 & & \\ & & & & 10 \cdot 20 \cdot 50 \cdot 100 & & \rightarrow \text{numero di operazioni} \end{array}$$

La parentesiizzazione migliore fa 2.200 passi, la peggiore ne fa 125.000 in questo caso.

**PARENTESIZZAZIONE MIGLIORE**  $(M_1 \times (M_2 \times M_3) \times M_4)$  oppure  $(M_1 \times M_2) \times (M_3 \times M_4)$

$$M = M_1 \times M_2 \times \dots \times M_4 \times \dots \times M_{m-1} \times M_m$$

Il punto più complesso è trovare le posizioni in cui "rompere" la moltiplicazione; chiameremo il punto K.

**CASO PIÙ SEMPLICE** ho solo due matrici, il costo è 1

$\ell=4$  il principio è lo stesso, ho già calcolato i costi per le triple

$$\ell=3 \quad \boxed{\phantom{0}} \boxed{\phantom{0}} \boxed{\phantom{0}}$$

avendo già calcolato le soluzioni e coppie è più semplice, ho già conservato il costo minimo

$$(M_1 \times M_2 \times M_3)$$

$$M_1 \times (M_2 \times M_3)$$

$$(M_1 \times M_2) \times M_3$$

↳ trovare K facile

$$\ell=2 \quad \boxed{\phantom{0}} \quad \boxed{\phantom{0}} \quad \boxed{\phantom{0}} \quad \boxed{\phantom{0}} \quad \text{analizzo il costo e coppie}$$

$$\ell=1 \quad \text{SOLUZIONE}$$

### FORMULA DI RICORRENZA

$m_{ij}$  costo minimo per calcolare il prodotto da  $m_{ii}$  ad  $m_{jj}$

sottoproblemi possibili

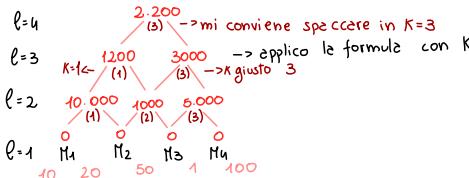
$\min$  problema originario

$$m_{ij} = \begin{cases} 0 & \text{non devo fare nulla (se } i=j\text{)} \\ \min_{1 \leq k \leq j} [m_{ik} + m_{k+1,j} + z_i z_{k+1} \cdot z_{j+1}] & \end{cases}$$

↳ O(m) lineare

ESEMPIO

$$(M_1 \times M_K) \times (M_K \times M_j)$$



0	10.000	1200	2200
	0	1000	3000
		0	5000
			0

↳ soluzione (0, m-1)

$$M_1 \times M_2 \times M_3 \times M_4$$

$$\Rightarrow (M_1 \times M_2 \times M_3) \times (M_4)$$

$$(M_1 \times (M_2 \times M_3)) \times (M_4) \quad \rightarrow \text{parentesiizzazione corretta}$$

### IL CODICE

```
int MinProdottiMatrici(vector<vector<int>> r, int n) // r=dimensioni matrici, n=numero di matrici
{
    vector<vector<int>> m(n, vector<int>(n)); // inizializziamo ogni elemento a n
    for (int v=1; v<=n-1; v++) // v= numero di "passi"
    {
        for (int i=0; i<=n-v-1; i++)
        {
            int j=i+v;
            m[i][j] = INT_MAX;
            for (int k=i; k<j; k++)
            {
                m[i][j] = min(m[i][j], m[i][k] + m[k+1][j]);
            }
        }
    }
    return m[0][n-1];
}
```

```

int K=i;
m[i][j] = m[i][K] + m[K+1][j] + v[i]*v[K+1]*v[j+1];
for(int K=i+1; K<j; K++) //cerco i K
{
    int mTemp = m[i][K] + m[K+1][j] + v[i]*v[K+1]*v[j+1];
    m[i][j] = min(mTemp, m[i][j]);
}
}

return m[0][n-1];
}

```

### SOLUZIONE DINAMICA - PROBLEMA KNAPSACK 0-1

Abbiamo **m** oggetti  $O_1 \dots O_m$ , di questi abbiamo:

**valori**  $v_1 \dots v_m$   
**costi**  $w_1 \dots w_m$

$$S(m, w)$$

**Costo massimo**  $W$ , costo massimo degli oggetti possibili da inserire (somme)

**CASO BANALE**  $m=0 \Rightarrow S(0, w)$  non ho scelte  $\rightarrow$  ho 0 oggetti, non aggiungo niente  $w=0 \leq w \leq W$   
 $W=0 \Rightarrow S(i, 0)$  ho i oggetti ma ce neanche 0, non aggiungo niente  $i=0 \leq i \leq m$

$S(i, w)$  vuol dire: piezza gli oggetti compresi tra  $O_i$  e  $O_i$   $\rightarrow$  cerca di piezzare i primi i oggetti

↳ RAPPRESENTAZIONE GENERALE DI OGNI SOTTOPROBLEMA INTERMEDI

Prendiamo un generico  $S(i, w)$  con  $(i, w > 0)$  **NON BANALE**

\* vedo a vedere se posso aggiungere i  $\rightarrow$  ello esino:

\* se  $w_i > w \Rightarrow O_i$  non puo' essere selezionato

→ risoluvi  $S(i-1, w)$

\* se  $w_i \leq w$

\* inserisci  $O_i$  e risoluvi  $S(i-1, w-w_i)$  scegli la soluzione

\* lascia  $O_i$  e risoluvi  $S(i-1, w)$  con valore massimo

#### CALCOLO DEI VALORI

$v(i, w)$  restituisce il valore di un sottoproblema

**CASI BANALI**  $v(0, w) : 0$

$v(i, 0) : 0$

$$v(i, w) = \begin{cases} v(i-1, w) & \text{se } w_i > w \\ \max \begin{cases} (v_i + v(i-1, w - w_i)) \text{ // inserisco } i \\ (v(i-1, w)) \text{ // lascio } i \text{ e prendo il valore max che lascia pi\`{e} capacit\`{e} fino a } i-1 \end{cases} \end{cases}$$

$\Rightarrow$  per calcolare  $v(i, w)$  ho bisogno delle soluzioni di  $v(i-1, w-w_i)$  e  $v(i-1, w)$

$$\Rightarrow \begin{array}{c} \overbrace{1 \quad 2 \quad 3 \quad 4 \quad \dots \quad w} \\ \left[ \begin{array}{cccc|c} 1 & 0 & 0 & 0 & \dots & 0 \\ 2 & 0 & & & & \\ 3 & 0 & & & & \\ \vdots & \vdots & & & & \\ m & 0 & & & & \end{array} \right] \end{array} \quad O(m \times w) \text{ costo costante}$$

0 > valore che voglio conoscere (valore massimo)

IL BACKTRACKING (usato in visita DFS)

Queste tecniche è adeguate a quei problemi per cui trovare una soluzione è complicato. Tipicamente questi algoritmi hanno complessità esponenziale. Cerca di costruire pezzo per pezzo la soluzione; appena si accorge che una soluzione non è ottenuta, non calcola le sue sottosoluzioni e cambia approccio risolutivo.

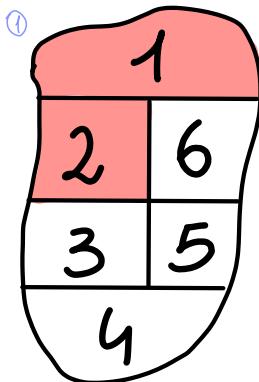
**PROBLEMA TIPICO** Coloreabilità di una mappa geografica. (Matrice di booleani o grafo)

**OBIETTIVO** Assegna un colore ad ogni nazione:

- nessuna nazione confinante ha lo stesso colore
- usando al più 6 colori (il problema è ancora semplice per l'esempio)
- usando al più un colore (semplice perché è impossibile)
- usando al più 3 colori (tipico)
- usando al più K colori (generale)

Assumiamo che i 3 colori siano rosso, verde e blu.

**RISOLUZIONE**



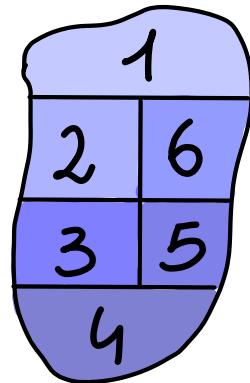
Provo ad assegnare rosso a 1 e 2

mi accorgo che non va bene;

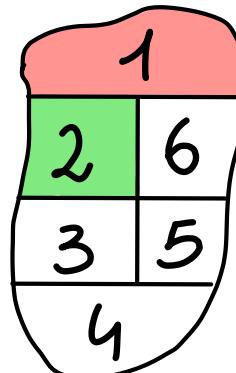
Torno indietro e riassegno a 2 :

rosso-rosso come combinazione

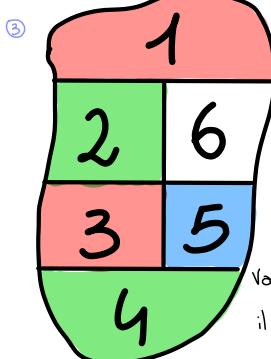
iniziale non va bene.



②

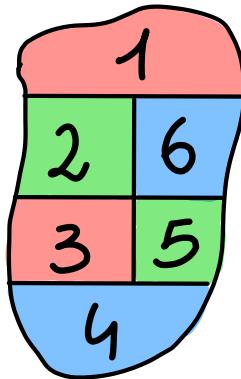


OK!

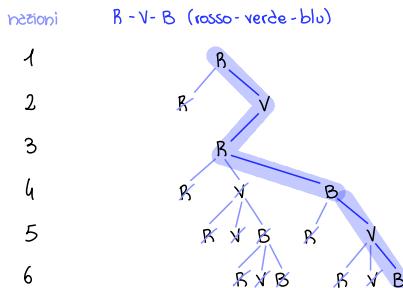


Vado avanti inserendo rosso-verde-blu in ordine; se il rosso non va bene provo il verde etc... facendo così arrivo al 6 e non ho colori disponibili.  
→ Tutte le soluzioni rosso-verde-rosso-verde non vanno bene

④



Faccio backtracking - torno indietro alle 3 nazioni e cambio i colori da lì, sapendo che le sole combinazione rosso-verde non funziona.



### PASSI DEL BACKTRACKING

- 1) Rappresentazione del problema (con che strutture posso rappresentare il mio problema?)
- 2) Definire il dominio di ciascuna sottosoluzione;
- 3) È necessario avere un dominio finito di valori (soluzioni);
- 4) Definire la rappresentazione delle soluzioni (per le cartine potrei usare un vector di colori, ho finito quando il vector ha 6 elementi).

### PSEUDO CODICE PER IMPLEMENTARE IL BACKTRACKING

```

List sol = {}

bool solve(sol)
{
    xe = min-val; // primo valore del dominio delle soluzioni

    while (xe <= max-val) // finché non finisce il dominio
    {
        if (!canAdd(xe, sol)) continue; // controllo se puoi aggiungere xe alla soluzione
        add(xe, sol);

        if (isComplete(sol)) //verifica se hai una soluzione completa
            return true;

        elseif (!solve(sol)) //chiamo ricorsivamente la solve e cerco di completare sol
            return true;

        remove(xe, sol); // xe non è valido, lo rimuovo da sol
        xe = next(xe); // pongo xe uguale al successivo
    }

    else
        xe = next(xe);
    }

    return false; //non ho trovato soluzioni
}

```

Questo algoritmo risolve ogni problema di backtracking; bisogna solo personalizzare le funzioni: `canAdd`, `add`, `isComplete`, `remove` e `next` in base al problema per cui dovranno utilizzarla.

#### CODICE - MAPPA GEOGRAFICA

`input`: Grafo mappe

`soluzione`: `vector<Colori>`

`dominio`: Rosso, Verde, Blu

↳ enum Colori {Rosso=0, Verde=1, Blue=2}; Il Rosso con 0, verde con 1 etc.

`min-vcl`: 0

`max-vcl`: 2

Mettemo in una struct ciò che ci serve per implementare il backtracking

struct BTProblem

| Grafo mappe;

vector<Colori> sol;

BTProblem(Grafo g): mappe(g) {} ; //prendo in input la mappa

}

Definiamo la funzione solve e i metodi usati in essa

void add(Colori x, BTProblem& bt)

{ bt.sol.push\_back(x); }

void remove(Colori x, BTProblem& bt)

{ bt.sol.pop\_back(); } //potrebbe essere utile farsi restituire l'ultima x aggiunta

Colori next(Colori x)

{ return ++x; } DA PROVARE! ; potrebbe dare problemi a causa dell'enum; meglio usare un vector di int

bool canAdd(Colori x, BTProblem bt) //se sono arrivato alla canAdd la soluzione SICURAMENTE non e' ancora completa

{ int size=bt.sol.size(); //numero di nazioni colorate fino ad ora

for(int i=0; i<size-1; i++)

if(bt.mappe(i, size) && bt.sol[i] != x)

//controllo se esiste l'arco tra la nazione i e la nazione size che devo colorare, se esiste sono addio centi; se sono  
adiacenti e il colore che voglio attribuire alla nazione size e' uguale a quello della nazione i ho fallito

return false;

return true; //l'altro non posso dare quel colore alla nazione size

}

```
bool isComplete(BTProblem bt)
{ return (bt.sol.size() == bt.mappe.n()); } // se il numero di elementi in sol e il numero di nodi nel grafo sono uguali ho finito
```

#### NOTA

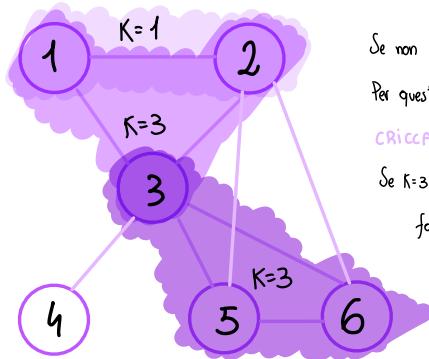
Conviene passare bt per riferimento costante, se non devo modificare nulla.

Mai implementare i controlli di confine in isComplete, oppure sfocio in un algoritmo brute force -> genero tutte le combinazioni possibili.

A volte potrebbe essere necessario verificare la validita' di alcuni vincoli alla fine.

## CRICCA

Ipotizzando di avere un grafo è un sottoinsieme di nodi del grafo, tale che ogni coppia di nodi è direttamente collegata tra loro con un arco.



Se non ho limiti sul sottoinsieme mi basta avere due nodi collegati tra loro.

Per questo solitamente si ricerca una cricca di cardinalità K

## CRICCA DI CARDINALITÀ K

Se K=3 per controllare se esiste la cricca:

```
for(i—  
    for(j—  
        for(k—
```

`if(G(i,j) && G(j,k) && G(i,k)) allora esiste`

se K è diverso da 3 devo  
combinare tutto il codice  
→ con K generico usiamo il  
backtracking

$G = (V, E)$  con m nodi ( $|V| = m$ ) e dato  $K$  ( $K \leq m$ ) cricca di cardinalità  $K$   $W \subseteq V$  ( $|W| = k$ ) tale che  $\forall (u, v) \in W \cup (v, u) \in E$ .

## SOLUZIONE - LISTA DI NODI

INPUT Grafo + K

SOLUZIONE

2
4
6

MIN-VAL 0  
MAX-VAL  $G.n() - 1$

## SOLUZIONE - VECTOR DI BOOL

INPUT Grafo + K

SOLUZIONE

T
F
T
T

MIN-VAL 0 (False)  
MAX-VAL 1 (True)

m=4  
vector<bool>  
nodo i ∈ W  
T vuol dire che ho  
messo il nodo nel  
sottoinsieme, F altrimenti  
ho una soluzione quando nel vector ho K True

Per inserire un nuovo  $\alpha$  devo controllare:

se tutti i nodi nel vector sono direttamente collegati è sì;

## CODICE - LISTA DI NODI

```
struct Tcricca
{
    Grafo g;
    int K;
} vector<int> sol;

bool solve(Tcricca& bt)
{
    if(bt.sol.size() == K)
        return true;
    for(int i=0; i<bt.sol.size(); i++)
        if(!bt.g(bt.sol[i], j))
            return false;
    for(int i=0; i<bt.sol.size(); i++)
        for(int j=i+1; j<bt.sol.size(); j++)
            if(bt.g(bt.sol[i], bt.sol[j]))
                if(bt.add(bt.sol[i], bt.sol[j]))
                    if(solve(bt))
                        return true;
    return false;
}
```

bool solve(Tcricca& bt)

...  
 $j=0$ ;

while ( $n < bt.g.n()$ )

```
void add(int &e, Tcricca &bt)
{
    bt.sol.push_back(e);
}

bool isComplete(Tcricca &bt)
{
    return (bt.sol.size() == bt.K);
}
```

```
* for(int i=0; i<bt.sol.size(); i++)
    if(!bt.g(bt.sol[i], j))
        return false;
```

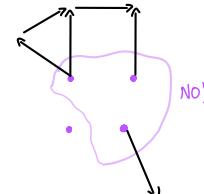
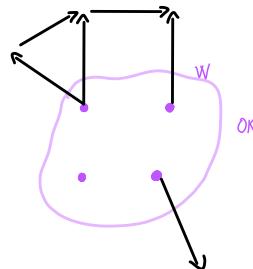
return true;

```
bool canAdd(int &e, Tcricca &bt)
{
    //devo controllare se ho già inserito n
    if(find(bt.sol, e) != bt.sol.end())
        *
```

## KERNEL

Dato  $G = (V, E)$  orientato determinare se  $G$  ha un Kernel  $W \subseteq V$  tale che:

- nessuna coppia di nodi in  $W$  è collegata;
  - $\forall v \in V - W \exists w \in W$  tale che  $(w, v) \in E$
- ↳ quando diventerà valida ho trovato il kernel



INPUT Grafo

SOLUZIONE Lista di nodi

MIN-VAL 0

MAX-VAL  $g, n()$

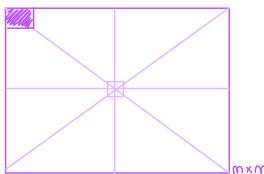
## CODICE

```
struct TKernel
{
    ——
}

bool canAdd(int xe, TKernel bt)
{
    if (find(bt.sol.begin(), bt.sol.end(), xe) != bt.sol.end())
        return false;
    for (int i=0; i<bt.sol.size()-1; i++)
        if (bt.g(bt.sol[i], xe) || bt.g(xe, bt.sol[i]))
            return false;
    return true;
}
```

```
bool isComplete(TKernel bt)
{
    for (int i=0; i<bt.g.n(); i++)
        if (find(bt.sol.begin(), bt.sol.end(), i) == bt.sol.end())
            if (!esisteArco(bt, i))
                return false;
    return true;
}
```

## N-QUEENS



Abbiamo  $m$  regine



→ righe delle scacchiere

soluzione  $\text{sol}[i]$  è le colonne in cui piazziamo le regine  
i sulle righe i

se metto la regina qui, non posso più metterne su quelle righe.

MIN-VAL (0,0) } per soluzione  
MAX-VAL ( $m-1, m-1$ ) } con matrice

MIN-VAL 0 } soluzione con  
MAX-VAL  $m-1$  } vettore

12/12/2022

### ESERCIZI RIEPILOGATIVI

#### 1) Dividing coins

Leggi in input una sequenza di numeri, terminata da -1, (sono monete), stampa le differenze minime delle divisione tra 2 persone.

INPUT 3 5 12 9 -1  
OUTPUT 1

INPUT 2 5 3 1 4 -1  
OUTPUT 0

#### SOLUZIONE GOLOSA

Ho in vet la sequenza di numeri

in somme ho le somme di tutti i numeri

max1 = Massimo(vet); vet.remove(max1); 12 7  
max2 = Massimo(vet); vet.remove(max2); 9 5  
int differenzaPerciale1 = somma - max1; 17 11  
int differenzaPerciale2 = somma - max2; 20 13  
while(vet.size() != 0)

{  
if(differenza2 > differenza1)  
{  
 max2 = Massimo(vet); vet.remove(max2); 5 3  
 differenza2 -= max2; 15 10  
 max1 = Massimo(vet); vet.remove(max1); 3 2  
 differenza1 -= max1; 14 9  
 else il contrario  
}  
}  
return differenza1 - differenza2; (in abs)

```
int Massimo(vector<int> vet )  
{ int max=0;  
for (int k=1; K <= vet.size()-1; k++)  
    if(max < vet[k])  
        max = vet[k]  
return max;
```

ESISTE UN CONTRO ESEMPIO  
PER CUI NON FUNZIONA

LA SOLUZIONE DINAMICA È QUELLA CORRETTA

etc somma = 29  

17	T
15	T
14	T
12	T
8	T
5	T
3	T
0	T

 DALL'ALTO VERSO IL BASSO  
vedo se è possibile la somma 14  
in questo caso è possibile -> le differenze minime sarebbero le differenze tra  $\frac{i^* + \text{somma}}{2}$  (arrotondato)

## IL CODICE

```
int main()
{
    vector<int> coins;
    int somme=0;
    int coin;
    cin >> coin;
    while(coin != -1)
    {
        coins.push_back(coin);
        somme += coin;
        cin >> coin;
    }
    bool possibiliSomme[somme + 1] = {false};
    possibiliSomme[0] = True;
    for(int coin: coins)
        for(int j = somme - coin ; j >= 0; j--)
            if(!possibiliSomme[j])
                possibiliSomme[j + coin] = True;
    bool trovato = false;
    for(int i = somme / 2; i >= 0 && !trovato; i--)
        if(!possibiliSomme[i])
        {
            Trovato = True;
            cout << "Difference minima = " << (somme - 1) - i;
        }
}
```

APPELLO DEL 22/02/2022

4 Scrivere una funzione che prenda in input un insieme di stringhe  $A$  (di cardinalità  $n$ ) e un insieme di triple ordinate  $C$  dove ogni tripla contiene elementi distinti di  $A$ . La funzione deve restituire YES se è possibile associare ad ogni elemento  $a \in A$  un numero da 1 ad  $n$ , di seguito indicato con  $f(a)$ , tale che le seguenti condizioni siano vere:

- non ci sono due o più elementi con lo stesso numero, quindi  $f(a) \neq f(b)$  per ogni  $a, b \in A$ , e
- per ogni tripla  $(x, y, z) \in C$ , è vero che  $f(x) < f(y) < f(z)$  oppure  $f(z) < f(y) < f(x)$ .

Se non è possibile creare una associazione che renda vere queste condizioni, la funzione deve restituire NO.

Si può assumere che:

- $A$  sia rappresentato come un `vector<string>`
- $C$  sia rappresentato come un `vector<Triple>` dove `Triple` è una classe già definita che ha tre campi pubblici  $x, y, z$  rappresentanti rispettivamente gli elementi (stringhe) della tripla,
- Non esistano due o più triple con gli elementi nello stesso ordine.

Esempio utilizzo `Triple`: sia  $t$  una istanza di `Triple` con gli elementi (abc, def, ghi), allora  $t.x == abc$ ,  $t.y == def$ , e  $t.z == ghi$ .

Esempio: in questo caso la funzione restituirà YES poiché è possibile associare ad ogni elemento di  $A$  un numero da 1 a 5 tale che le condizioni di cui sopra siano rispettate.

In particolare, supponiamo di aver associato i seguenti numeri:  $a \rightarrow 2, b \rightarrow 1, c \rightarrow 3, d \rightarrow 5, e \rightarrow 4$ . La prima condizione è soddisfatta (non ci sono due elementi con lo stesso numero); la seconda condizione è anch'essa soddisfatta, poiché:

- per la tupla  $(a, e, d)$  vale  $f(a) < f(e) < f(d)$ ,
- per la tupla  $(b, c, d)$  vale  $f(b) < f(c) < f(d)$ ,
- per la tupla  $(c, a, b)$  vale  $f(b) < f(a) < f(c)$ ,
- per la tupla  $(d, e, c)$  vale  $f(c) < f(e) < f(d)$ .

$$n = 5$$

$$A = \{a, b, c, d, e\}$$

$$C = \{(a, e, d), (b, c, d), (c, a, b), (d, e, c)\}$$

$$\min\_val = 1$$

$$\max\_val = m$$

Si applica il BACKTRACKING.

```
struct Tsoluzione
{
    vector<string> A;
    vector<Triple> C;
    int n; //> A.size()
    vector<int> sol; //> soluzione
}
```

ESERCIZIO SULLA PROGETTAZIONE DELLE CLASSI

```
int main()
{
    Prodotto* p1 = new Libro("id1", 10.0, "I promessi sposi", "Manzoni");
    Prodotto* p2 = new Computer("id3", 700.000, "hp ex8p3");
    p1->stampaPrezzoScontato(); // stampa effeso: 4.0 (10-30%)
    p2->stampaPrezzoScontato(); // stampa effeso 665.0 (700 - 51%)
```

```

cout << *p1 << endl; //stampa etessa: Libro: id1, 10.0, I promessi sposi, Manzoni
cout << *p2 << endl; //stampa etessa: Prodotto: id3, 700
delete p1;
delete p2;
return 0;
}

```

Scrivvi le classi Prodotto, Computer e Libro.

#### CLASSE PRODOTTO

```

class Prodotto
{
protected://oppure private e definisco get e set
    string id;
    double prezzo;
public:
    Prodotto(string i, double p): id(i), prezzo(p){}

    virtual void stampaPrezzoScontato() const
    {
        cout << prezzo << endl;
    }

    friend ostream& operator << (ostream& o, const Prodotto& p)
    {
        return p.stampa(o);
    } //richiamo cosi' la stampa giusta a seconda del tipo di prodotto che gli passo

    double getPrezzo() {return prezzo; }

    string getId() {return id; }

    virtual ~Prodotto() {} //il distruttore e' virtuale perche' dipende dal tipo di prodotto

protected:
    virtual ostream& stampa(ostream& out) const //stampa generica di un qualsiasi prodotto
    {
        out << "Prodotto: " << id << ", " << prezzo;
        return out;
    }
}

```

### CLASSE LIBRO

```
class Libro: public Prodotto
{
    string titolo;
    string autore;
public:
    Libro(string id, double p, string t, string a): Prodotto(id, p), titolo(t), autore(a) {}

    void stampaPrezzoScontato() const
    {
        cout << getPrecio() * 0,3 << endl;
    }

protected:
    ostream& stampa(ostream& out) const
    {
        out << "Libro: " << getId() << ", " << etc...
        return out;
    }
}
```

### CLASSE COMPUTER

```
class Computer: public Prodotto
{
    string modello;
private:
    Computer(string id, double p, string n): Prodotto(id, p), modello(n) {}

    void stampaPrezzoScontato() const
    {
        cout << getPrecio() - getPrecio() * 0,05 << endl;
    }
} //non scrivo le stampa, perche' il main detto voleva che le stampa per computer fosse quelle generiche di prodotto.
```