Ripetizioni informatica

Mattia Marini

Agosto 2023/2024

Indice

Esercizi

Esercizi introduttivi

Esercizi su classi

Algoritmi

Problemi noti dp

1 Da c++ a java

Java è molto simile dal punto di vista della sintassi al c++. Non sarà molto complicato il passaggio

1.1 Aspetti simili

La sinttassi di Java è molto simile a quella di c++, ecco gli aspetti che rimangono invariati o quasi:

1.1.1 Dichiarazione delle variabili:

Sinstassi	Differenze rispetto al `c++`
`int x = 15`	invariato
`long x = 15`	invariato
`float x = 15.0f`	nota il `15.0f`, dove `f` sta per float
`double x = 15.0`	float ma con precision maggiore, 64 bit
`boolean x = true`	`boolean` anzichè `bool`
`String s = "stringa"`	dato `String` sarebbe una classe ma è trattato come tipo primitivo, dato che è usato molto frequentemente
`String v[] = new String[15]`	vettore di stringhe di dimensione 15
`int v[] = new int[15]`	vettore di interi di dimensione 15

1.1.2 Commenti:

- Commento riga singola: `// commento`
- Commento righe multiple: `/* commento */`

1.1.3 Cicli:

- Ciclo for:

```
for(int i = 0; i<15; i++){
  // qualcosa
}
```

- Ciclo while:

```
while(i < 15){
  // qualcosa
}
```

1.1.4 Operatori

Operatore	Descrizione
+ - * /	operatori matematici
&&	and logico
||	or logico
!	not logico

1.1.5 If e switch

```
if(a<b) {
  //codice
}
else(if a>b) {
  //codice
}
else {
  //codice
}
```

```
switch(espressione) {
  case x:
    // codice
    break;
  case y:
    // codice
    break;
  default:
    // codice
}
```

1.1.6 Funzioni

```
void nome_funzione1 (int arg_1, String arg_2){
  //corpo funzione
}

int nome_funzione2 (int arg_1, String arg_2){
  //corpo funzione
  return 5;
}
```

Una funzione è quindi definita indicando nel seguente ordine, esattamente come in c++:

1. Tipo di ritorno (o `void` se non ritorna nulla)
2. Nome della funzione
3. Parametri, racchiusi fra parentesi tonde e separati da virgole
4. Corpo della funzione fra graffe

1.2 Differenze

1.2.1 Stampa su terminale

Una delle feature usata moltissimo, ma completamente diversa dal c++ è la stampa su terminale:

```
System.out.println("Stampa questa cosa");
//stampa andando a capo prima di stampare

System.out.print("Stampa questa cosa");
//stampa SENZA andando a capo
```

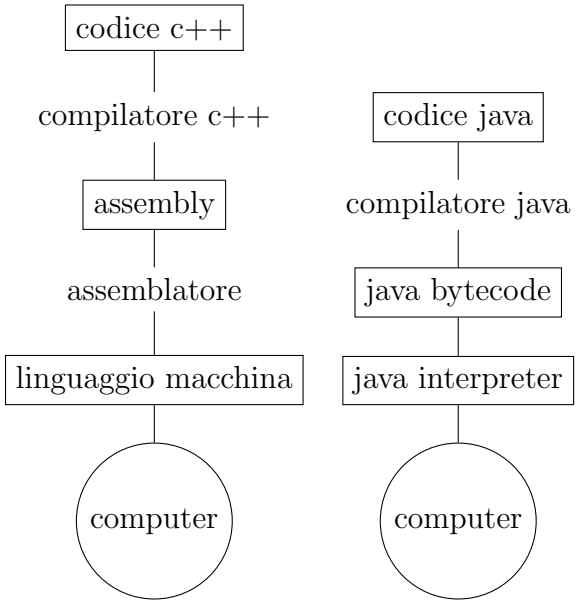
nota che le stringhe si possono concatenare con l'operatore +:

```
String a = "Hello";
String b = "world";

System.out.println(a + b);
//stampa "Hello world"

String c = a + b;
//Inizializza c a "Hello world"
```

1.2.2 Linguaggio interpretato vs compilato

A differenza di c++, java è un linguaggia interpretato

- *Linguaggio compilato*: il codice è "dato in pasto" a un compilatore, il quale lo converte in linguaggio macchina (di fatto in una sequenza di, 0 ed 1)
- *Linguaggio interpretato*: il codice è "dato in pasto" ad un compilatore, il quale lo converte però in bytecode, ossia un linguaggio di basso livello (molto difficile da leggere e scrivere), il quale è in grado di essere letto da un *interprete*



1.2.3 Memoria e puntatori

Java è un linguaggio ad alto livello che gestisce la memoria in maniera diversa rispetto al c++:

- c++: il compito di allocare e deallocare la memoria non più utilizzata è del programmatore
- java: la memoria viene deallocata in maniera automatica tramite un meccanismo chiamato garbage collection

Visto che in java la memoria è gestita in in maniera automatica, il programmatore non ne ha accesso diretto tramite puntatori: al contratio, i puntatori non esistono

1.2.4 Object orientation

Sebbene c++ sia un linguagio che permette di utilizzare classi ed oggetti, in java l'object orientation è forzata: ogni parte del programma deve essere contenuta all'interno di una classe

1.3 Esempio programma

```
class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

Cose da notare:

- La funzione main è contenuta all'interno di una classe "HelloWorld", il cui nome è arbitrario
- La funzione main è marcata come `static`, ciò vuol dire che la funzione esiste anche se non esiste un oggetto di tipo "HelloWorld", affroteremo meglio il modificatore `static` più avanti (sez. 2.4), per ora possiamo ignorarlo
- La funzione main è marcata come `public`, ciò vuol dire che la funzione è accessibile ovunque. Afffronteremo meglio questo modificatore più avanti, per ora possiamo ignorarlo
- La funzione main prende in ingresso un vettore di stringhe. Nel caso si avviasse l'applicazione da terminale e possibile passare al main dei parametri nel seguente modo:

  ```
  cd cartella_applicazione
  ./nome_applicazione parametro_1 parametro_2 ...
  ```

  in questo caso il vettore di stringhe `args` conterrà `parametro_1` e `parametro_2`. Penso non lo userete mai ma è buono saperlo

1.4 Funzioni utili

In java sono definite alcune funzioni utilissime. Qui una lista (non esaustiva) delle più comuni:

1.4.1 Stringhe

Supponiamo di avere `String s = "stringa stringa";`

Funzione	Descrizione
`s.lenght()`	Ritorna il numero di caratteri conenuti nella stringa (7 nel caso d'esempio)
`s.charAt(int index)`	Ritorna il carattere in posizione `index`
`s.indexOf(char carattere)`	Ritorna l'indice della prima occorrenza di `carattere` in `s`
`s.indexOf(String stringa)`	Ritorna l'indice della prima occorrenza della sotto-stringa `stringa` in `s`

```
String s = "stringa stringa";
s.lenght(); // 15
s.charAt(2); // 'r'
s.indexOf('r'); // 2
s.indexOf("ga"); // 5
```

1.4.2 Vettori

Supponiamo di avere `int v[] = new int[15];`

Funzione	Descrizione
`v.lenght`	ritorna il numero di elemnti contenuti nel vettore

1.4.3 Altre funzioni utili

Funzione	Descrizione
`Math.exp(float n)`	Ritorna e^n
`Math.log(float n)`	Ritorna $ln\,(n)$
`Math.abs(float x)`	Ritorna $\|x\|$ (valore assoluto di x)
`Math.sin(float x)`	Ritorna $\sin(x)$
`Math.cos(float x)`	Ritorna $\cos(x)$
`Math.tan(float x)`	Ritorna $\tan(x)$
`Math.asin(float x)`	Ritorna $\arcsin(x)$
`Math.acos(float x)`	Ritorna $\arccos(x)$
`Math.atan(float x)`	Ritorna $\arctan(x)$
`Math.max(float a, float b)`	Ritorna l'elemento maggiore fra a e b
`Math.min(float a, float b)`	Ritorna l'elemento maggiore fra a e b
`Math.floor(float x)`	Arrotonda per difetto x
`Math.ceil(float x)`	Arrotonda per eccesso x
`Math.round(float x)`	Arrotonda x

1.5 Esercizi

Esercizio 1: *Somma float*

Scrivi un programma che dichiari 3 float con valore 15, 12.5 e -12 e ne stampi la somma

Esercizio 2: *Concat stringhe*

Scrivi un programma che date due stringhe contenenti nome e cognome, stampi l'uno concatenao all'altro, separati da uno spazio

Esercizio 3: *Incrementa vettore*

Scrivi un programma che dato un vettore di interi, incrementi di 1 ogni suo elemento

Esercizio 4: *Funzione 1*

Ripetere l'esercizio precedente, spostando il codice all'interno di una funzione apposita

Esercizio 5: *Funzione stampa pari*

Creare una funzione che prenda in input un vettore di interi e ne stampi solo gli elementi pari

Esercizio 6: *Smista stringhe*

Creare una funzione che prenda in input un vettore di stringhe e un intero n. Ritornare un vettore contenente solo le stringhe con lunghezza minore o uguale n

2 Classi e oggetti

Immaginiamo di voler rappresentare un dato "custom", che non è presente di default in java. Supponiamo di dover rappresentare un punto su un piano cartesiano. Questo dato deve:

- Contenere informazioni riguardo la posizione (x e y)
- Permetterci di stamparlo a terminale
- Permetterci di calcolarne la distanza dall'origine e da un altro punto

In questo caso le classi permettono di fare esattamente ciò che vogliamo: *creare un nuovo tipo di dato che possa contenere dei valori e supportare determinati tipi di funzioni*

Una classe non è altro che un insieme di variabili, dette attributi, e di funzioni che possono agire su di esse, dette metodi)

La classe va a definire "lo scheletro" del dato che vogliamo creare. In poche parole andiamo a elencare quali dati contiene e quali funzioni supporta. Ad esempio, per definire una classe con le funzionalità elencate pocanzi potremmo scrivere il seguente codice:

```
class Coordinata {

  double x = 10.0; //attributo
  double y = 12.5; //attributo

  // metodo
  public void stampa() {
    System.out.println("(" + x + ", " + y + ")");
  }

  // metodo
  public double distanzaOrigine() {
```

```
    return Math.sqrt(x * x + y * y);
  }

  // metodo
  public double distanzaCoordinata(coordinata c) {
    double deltaX = c.x - x;
    double deltaY = c.y - y;
    return Math.sqrt(deltaX * deltaX + deltaY * deltaY);
  }

}
```

Per utilizzare questo dato, possiamo scrivere:

```
Coorinata nomeCoordinata = new Coordinata();
```

La variabile "nomeCoordinata" è detta oggetto o istanza di tipo `Coordinata`. Posso ora riferirmi agli attributi e metodi che ho definito prima nel seguente modo:

```
nomeCoordinata.x;
nomeCoordinata.y;

nomeCoordinata.stampa();
nomeCoordinata.distanaOrigine();
nomeCoordinata.distanzaCoordinata(new Coordinata())
```

Nota che:

- Il nome della classe "Coordinata" è scritto con la lettera maiuscola. Questa è una *convenzione* importante: il programma funzionerebbe anche se non mettessimo la maiuscola, però per questioni di leggibilità del codice si è deciso che il nome delle classi va sempre con la maiuscola
- La classe va dichiarata all'esterno della classe main. Se usate Eclipse o Intellij ogni classe è bene che venga dichiarata come `public class NomeClasse{}` in un file separato, con nome "NomeClasse.java"
- Le funzioni sono marcate come `public`. Questo significa che possono essere chiamate in qualsiasi punto del programma. Vedi sezione 2.2

2.1 Costruttore

Quando creiamo un oggetto è utile avere un modo per inizializzare i suoi attributi. Il costruttore di una classe è una funzione specializzata, definita all'interno della classe che fa proprio questo.

- Il costruttore è una funzione `public` che ha lo stesso nome della classe
- Il costruttore viene invocato implicitamente quando creiamo un oggetto
- Il costruttore non ha un valore di ritorno

Riprendendo l'esempio di prima, possiamo implementare un costruttore come segue:

```
class Coordinata {

  double x; // attributo
  double y; // attributo

  //costruttore
  public Coordinata(double newX, double newY) {
    x = newX;
    y = newY;
  }

  // metodo
  public void stampa() {
    System.out.println("(" + x + ", " + y + ")");
  }

  // metodo
  public double distanzaOrigine() {
    return Math.sqrt(x * x + y * y);
  }

  // metodo
  public double distanzaCoordinata(coordinata c) {
    double deltaX = c.x - x;
    double deltaY = c.y - y;
    return Math.sqrt(deltaX * deltaX + deltaY * deltaY);
  }

}
```

Ora possiamo (e dobbiamo) creare oggetti scrivendo

```
Coorinata nomeCoordinata = new Coordinata(15.0, 12.5);
```

dunque con la keyword `new` viene invocato implicitamente il costruttore della classe, che ne inizializza gli attributi x e y

2.1.1 La keyword "this"

Quando definiamo un costruttore può essere intuitivo dare ai parametri della funzione lo stesso nome dei corrispettivi attributi, tuttavia così facendo si creerebbe anbiguità:

```
class Coordinata {

  double x; // attributo
  double y; // attributo

  //costruttore
  public Coordinata(double x, double y) {
    x = x; //a quale x mi sto riferendo?
    y = y; //a quale y mi sto riferendo?
```

```
  }
  //...
}
```

in questo caso torna utile una keyword specifica: `this`. `this` all'interno di un classe non è altro che un riferimento alla istanza della classe stessa. Posso quindi riscrivere il costruttore così:

```
class Coordinata {

  double x; // attributo
  double y; // attributo

  //costruttore
  public Coordinata(double x, double y) {
    this.x = x; //this.x si riferisce all'attributo x, mentre x al
    parametro del costruttore
    this.y = y; //this.y si riferisce all'attributo y, mentre x al
    parametro del costruttore
  }
  //...
}
```

2.2 Modificatori visibilità

Abbiamo visto che prima del metodi si può usare la parola `public`. Questo è un modificatore di visibilità e serve a specificare da quale parte del codice è possibile invocare il metodo. I modificatori possono essere applicati anche agli attributi. I modificatori sono i seguenti:

Modificatore	Descrizione
`private`	visibile solo all'interno della classe stessa
`public`	visibile ovunque
`protected`	visibile solo all'interno della classe stessa e di quelle che la estendono
`"package"`	visibile all'interno di ogni file contenuto all'interno dello stesso package (cartella)

Nota:

- Capiremo meglio cosa voglia dire `protected` nella sezione **??**
- Di default, metodi e variabili hanno visibilità `"package"`, tuttavia non esiste una keyword per indicare questo tipo di visibilità. E' presente di default se non si indica nulla

2.3 Getters setters e encapsulation

Uno dei concetti chiavi della programmazione ad oggetti è *l'encapsulation*. L'encapsulation può essere visto come uno "stile" di programmazione secondo il quale i dati contenuti all'interno delle classi possono essere modificati e acceduti solo da metodi della classe stessa. Questo permette di garantirne l'uso corretto e nascondere ciò che non serve (*information hiding*).

Nel pratico, per ottenere l'encapsulation bisogna dichiarare gli attributi della classe come `private`, in modo che l'utente non possa modificarli tramite istanza.

Se è necessario accedere / modificare l'attributo privato è necessario farlo tramite metodi della classe stessa, chiamati rispettivamente *getters* e *setters*

```
class Coordinata {

  private double x; // attributo PRIVATO
  private double y; // attributo PRIVATO


  // getter
  double getX (){
    return x;
  }

  // getter
  double getY (){
    return y;
  }

  // setter
  void setX (double x){
    this.x = x;
  }

  // setter
  void setY (double y){
    this.y = y;
  }

  //...

}
```

- Il nome dei metodi getters e setters è per convenzione formato da `get/set` + `nome_variabile`
- Nota l'uso del `this` per evitare ambiguità nei setters

2.4 Modificatore static

Distringuiamo l'uso del modificatore static per le variabili e per i metodi

2.4.1 Static per metodi

Il modificatore static all'interno di una classe permette di *invocare il metodo staticamente, ossia senza bisogno di invocarlo tramite un'istanza della classe.* I metodi statici sono spesso utili per creare classi che non sono altro che gruppi di funzioni di utilità, ad esempio supponiamo voler creare una classe che contenga funzioni utili per effettuare calcoli geometrici:

```
class Geometria {

  public static double areaParallelogramma(double base, double altezza) {
    return base * altezza;
  }

  public static double areaCerchio(double r) {
    return 3.14 * r * r;
  }

}
```

Così facendo potrò invocare i metodi `areaParallelogramma` e `areaCerchio` scriverndo `Geometria.areaParallelogramma(10.0,12.5)` e `Geometria.areaCerchio(5.0)`

NB: è possibile invocare un metodo statico anche attraverso un'istanza della classe, tuttavia è sconsigliato e non ha molto senso

2.4.2 Static per gli attributi

Il modificatore static messo davanti alla dichiarazione di una variabile indica che la suddetta variabile è salvata in una zona di memoria speciale, lo `static segment`, ossia una porzione di memoria che viene allocata a compile time. Ciò significa che la variabile `static` sarà allocata prima dell'esecuzione del programma una volta sola. Di conseguenza ogni istanza della classe condividerà questa variabile. Vediamo un esempio:

```
class A{
  int nonStaticInt = 0;
  static int staticInt = 0;
  public A(){
    nonStaticInt++;
    staticInt++;
  }

}
```

se, ad esempio, all'interno del main scrivessi:

```
//nel main
A a1 = new A();
A a2 = new A();

System.out.println(a1.nonStaticInt);// 1
```

```
System.out.println(a2.nonStaticInt);// 1

System.out.println(a2.staticInt); //2
System.out.println(a1.sttaticInt); //2

System.out.println(A.staticInt); //2
```

Di fatto `a1.staticInt`, `a2.staticInt` e `A.staticInt` sono tre modi diversi per riferirsi alla medesima locazione di memoria, all'interno dello *static segment*

2.5 Parti utili della libreria standard

La libreria standard di java offre moltissime classi utili. Vediamo qui le più comuni

2.5.1 ArrayList

Utile per avere un vettore di lunghezza variabile. Si inizializza nel seguente modo:

```
ArrayList< tipo > nomeArray = new ArrayList< tipo >(dimensione)
```

Nota che:

- `dimensione` si può omettere, ottenendo un vettore con dimensiane nulla
- `tipo` deve essere una classe. Se voglio un arrayList di tipi primitivi devo utilizzare le classi wrapper (`Integer, Boolean, Double` ...)

Metodo	Descrizione
`v.get( index )`	ritnorna l'elemento all'indice `index`
`v.set( index , element)`	setta l'elemento a indice `index`
`v.add( element )`	inserisce `element` in fondo
`v.remove ( index )`	rimuove l'elemento a indice `index`
`v.size()`	ritorna il numero di elementi conetnuti
`v.clear()`	rimuove tutti gli elementi

2.5.2 Scanner

La classe scanner ci permette di leggere input utente da terminale (e anche da file, ma non ci servirà). Uno scanner si inizializza così:

```
Scanner nomeScanner = new Scanner(System.in);
```

Per leggere l'input da terminale ci sono i seguenti comandi:

Metodo	Descrizione
nextBoolean( )	Legge un boolean da terminale
nextByte()	Legge un byte da terminale
nextDouble()	Legge un double da terminale
nextFloat ()	Legge un float da terminale
nextInt()	Legge un int da terminale
nextLine()	Legge una String da terminale
nextLong( )	Legge un long da terminale
nextShort()	Legge uno short da terminale

Il metodo più comune è `nextLine()`, dato che ci restituisce l'intera riga come stringa, anche se contiene numeri

2.5.3 Random

Random fornisce un modo comodo per generare numeri casuali. Inizializza con

```
Random nome = new Random()
```

Metodo	Descrizione
nextInt(range)	genera un numero casuale nel range $[0, \text{range})$
nextFloat()	genera un float in range $[0.0, 1.0]$
nextDouble()	genera un double in range $[0.0, 1.0]$

2.5.4 LinkedList

Lista linkata. Utilizzabile in modo molto simile al `ArrayList`. L'accesso agli elementi è molto più lento, rimozioni inserimenti sono molto più veloci

2.5.5 Altre strutture dati utili

- `HashSet`: insieme matematico, possibile vedere se un elemento è contenuto in esso in maniera efficiente
- `Map`: utile poter collegare ad ogni valore un altro valore detto chaive. Si può risalire al valore tramite la chiave in maniera efficiente
- `Stack`
- `Queue`
- `PriorityQueue`: struttura nella quale è possibile accedere all'elemento maggiore in maniera efficiente

- `SortedSet`: struttra nella quale i dati mantengono sempre un ordinameto crescente. Non ammette duplicati

2.6 Esercizi

Esercizio 7: *Esercizio completo classi*

Creare due classi: una rappresenterà un oggetti di tipo auto, e l'altra un oggetto di tipo concessionario

- La `class Auto` deve contenere:
  - Attributi che rappresentino modello, casa produttrice, cilindrata, costo, peso e targa
  - Costruttore che prenda in input tutti gli attributi meno che la targa. Questa va generata secondo il seguente formato:

    `ll - ccc - ll`

    dove `l` è una lettera maiuscola e `c` è una cifra da 1 a 9 utilizzare un'apposita classe che rappresenti una targa, che renda impossibile ottenere targhe con formati sbagliati
  - Metodo `stampa` che stampi a terminale in modo riassuntivo tutti gli attributi
  - Metodo `perNeopatentato` che ritorni vero se e solo se la cilindrata è inferiore a 95 e il rapporto fra cilindrata e peso(in kili) è inferiore a 0.055
  - Variabile contenente il numero di auto create
- La `class Concessionario` deve contenere:
  - Un vettore che abbia dimensione massima impostata nel costruttore e che contenga oggetti di tipo Auto
  - Un attributo nome, proprietario
  - Un metodo `stampa` che stampi tutte le auto presenti nel concessionario con una breve descrizione
  - Un metodo `cerca` che prenda in input un modello di auto e restituisca il numero di modelli presenti nell'autoconcessionario, oppure -1 se non presente
  - Una funzione `autoNeopatentati` che stampi a video tutte le auto per neopatentati, oppure "non ho trovato nulla" nel caso non ce ne siano
  - Una funzione `filtra` che stampi a video tutte le auto con costo minore di quanto indicato come parametro, oppure "non ho trovato nulla" nel caso non ce ne siano

Soluzione overengeenered qui

Esercizio 8: *Frammenti di codice*

Per ognuno di questi frammenti di codice, indica l'output

1.

```
public class Esercizio1 {
    public static void main(String[] args) {
        A a = new A();
        a.stampaX(5);
    }
}
class A{
    int x = 0;
    void stampaX(int x){
        System.out.println(x);
    }
}
```

2.

```
public class Esercizio2 {
    public static void main(String[] args) {
        A a1 = new A(5);
        A a2 = new A(5);
        System.out.println(a1.x + a2.x);
    }
}
class A{
    static int x = 0;
    public A(int x){
        x += x;
    }
}
```

3.

```
public class Esercizio3 {
    public static void main(String[] args) {
        System.out.println(15 +15.0 + "stringa");
    }
}
```

4.

```
public class Esercizio4 {
    static void incrementa(int x){
        x++;
    }
    public static void main(String[] args) {
        int x = 0;
```

```
            incrementa(x);
            System.out.println("x");
        }
    }
```

Esercizio 9: *Trova l'errore, se presente*

Individua, se presente, l'errore nei seguenti spezzoni di codice e spiega la causa

1.

```
class Esercizio5{
    int x = 15;

    static void stampaX(){
        System.out.println(x);
    }
}
```

2.

```
public class Esercizio6 {

  public static void main(String[] args) {
    A a = new A();
    a.modificaX();
  }

}

class A {


  void modificaX(){
    x = 0;
  }

}
```

3.

```
public class Esercizio7 {

  public static void main(String[] args) {
    A a = new A();
    System.out.println(a.x);
  }

}
```

```
class A {
  private int x = 15;
}
```

2.6.1 Esercizi completi

Esercizio 10: *Classe persona*

Scrivi un programma creando una classe `Persona`, con attributi `nome` e `età`. Crea due istanze, imposta gli attributi usando il costruttore e stampa il nome e l'età.

Esercizio 11: *Classe cane*

Scrivi un programma creando una classe `Cane` con attributi `nome` e `razza`. Crea due istanze, imposta gli attributi usando il costruttore e modifica gli attributi usando getters e setters e stampali. Che visibilità dai agli attributi?

Esercizio 12: *Classe libreria*

Scrivi un programma creando una classe `LibreriaMusicale`, con una collezione di oggetti `Canzone` e metodi per aggiungere, rimuovere e far partire canzoni casuali. La classe `Canzone` deve contenere titolo, artista e durata. La classe `LibreriaMusicale` deve permettere di riordinare le canzoni in base a Titolo, Artista e Durata. Usare una classe ausiliaria per contenere i metodi di riordinamento. Aggiungere anche un metodo per calcolare la durata media di tutti i brani

3 Ereditarietà e polimorfismo

L'ereditarietà è un concetto chiave della programmazione ad oggetti. Di base, il meccanismo di ereditarietà ci permette di *estendere* una classe riutilizzando il codice della classe estesa.

Dal punto di vista logico, questo concetto torna utile nel momento in cui dobbiamo modellare delle classi che sono *sotto categorie logiche* l'una dell'altra

3.0.1 Esempio veicolo

Immaginiamo di voler modellare delle classi per rappresentare dei veicoli: vogliamo implementare le seguenti classi:

- `class Veicolo`: veicolo generico
- `class VeicoloAgricolo`

- `class Moto`

In questo caso notiamo come `VeicoloAgricolo` e `Moto` costituiscano delle *sotto categorie logiche* di `Veicolo`: veicoli agricoli e veicoli da strada sono veicoli. Per questa ragione probabilmente dovremmo riutilizzare del codice della classe `Veicolo` all' interno delle classi `Moto` e `VeicoloAgricolo`. Qui ci viene in contro il meccanismo di ereditarietà

La classe veicolo conterrà dati e funzionalità generiche:

```
class Veicolo {

  String modello = "Undefined";
  int costo;

  public Veicolo(String modello, int costo) {
    this.modello = modello;
    this.costo = costo;
  }

  public void stampa(){
    System.out.println("Veicolo, modello " + modello + ", costo " +
    costo);
  }

}
```

possiamo ora creare le classi `VeicoloAgricolo` e `VeicoloStrada`, andando ad aggiungere dettagli alla classe `Veicolo`, senza però avere duplicazione di codice:

```
class VeicoloAgricolo extends Veicolo {

  String impiegoAgricolo;

  public VeicoloAgricolo(String modello, int costo, String
    impiegoAgricolo) {
    super(modello, costo);
    this.impiegoAgricolo = impiegoAgricolo;
  }

  public void stampaImpiego(){
    System.out.println(impiegoAgricolo);
  }

}

class Moto extends Veicolo {
  boolean isCross;

  public Moto(String modello, int costo, boolean isCross) {
    super(modello, costo);
    this.isCross = isCross;
```

```
  }

}
```

3.1 Esempio forme

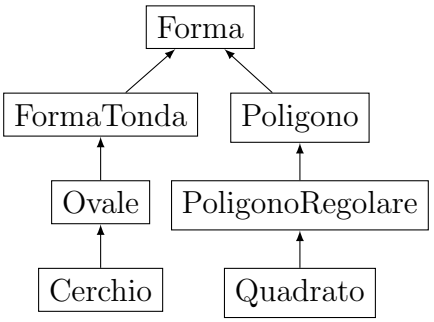
Vediamo un altro esempio in cui l'ereditarietà ci permette di modellare in maniera intuitiva una situazione reale:

```
public class EsempioForma {

}

class Forma {

  int borderR = 0, borderG = 0, borderB = 0;
  int fillR = 255, fillG = 255, fillB = 255;
  float perimetro;

}

class FormaTonda extends Forma {

}

class Ovale extends FormaTonda {
  float raggioX;
  float raggioY;

  public Ovale(float raggioX, float raggioY) {
    this.raggioX = raggioX;
    this.raggioY = raggioY;
  }

}

class Cerchio extends Ovale {
  float raggio;

  public Cerchio(float raggio) {
    super(raggio, raggio);
    this.raggio = raggio;
  }

}

class Poligono extends Forma {
  int numeroLati;

  public Poligono(int numeroLati) {
```

```
    this.numeroLati = numeroLati;
  }

}

class PoligonoRegolare extends Poligono {
  float lunghezzaLati;

  public PoligonoRegolare(float lunghezzaLati, int numeroLati) {
    super(numeroLati);
    this.lunghezzaLati = lunghezzaLati;
  }

}

class Quadrato extends PoligonoRegolare {
  float lato;

  public Quadrato(float lato) {
    super(lato, 4);
    this.lato = lato;
  }

}
```

3.2 Class diagram

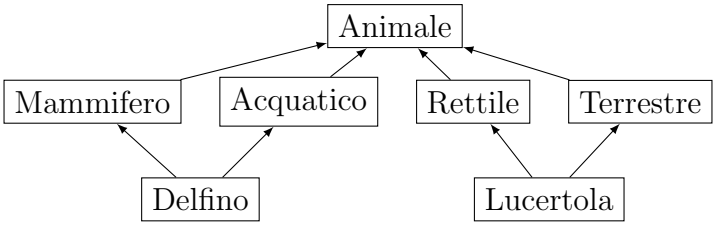
Tramite ereditarietà abbiamo visto che è possibile modellare una relazione di tipo "is a", ossia *quando un oggetto B estende un oggetto A, allora B "is a" A*. Dal punto di vista logico, abbiamo visto che, ad esempio, un quadrato è un poligono regolare. Questo tipo di relazione è spesso rappresentato graficamente tramite il diagramma delle class. Una freccia $\longrightarrow$ rappresenta una relazione di tipo "is a". Il diagramma delle classi dell'esercizio 3.1 sarebbe il seguente:



Nota come la struttura creatasi è un albero. Questa è detta gerarchia delle classi.

3.2.1 Eseditarietà singola vs multipla

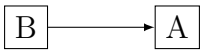
Nota anche come una qualsiasi classe può estendere solo una classe. Quando è presente questo vincolo si parla di ereditarietà singola. Java ha ereditarietà singola, mentre in c++ è supportata l'ereditarietà multipla, più flessibile ma presenta più complicazioni



Un'ereditarietà di questo tipo non sarebbe possibile in java, tuttavia lo è in linguaggi quali c++ o Python

3.3 Overriding

Come abbiamo visto fin'ora, dire che una classe B estende una classe A, significa che B è una versione più dettagliata di A, una sua sottocategoria logica. A tutti gli effetti, B *is a* A.



Si legge *"B is a A"*

Per questa ragione in codice, supponendo di avere due classi:

```
class A{
}

class B extends A{
}
```

possiamo scrivere delle cose di questo tipo

```
A a1 = new B();
A a2 = new A();

B b1 = new B();
```

occhio che la scrittura seguente è sbagliata (compiler error)

```
B b1 = new A(); //errore
```

3.4 Overriding, overloading e polimorfismo

Fino ad ora abbiamo visto come estendere classi e il significato dal punto di vista logico di questa operazione. Tuttavia l'ereditarietà ci permette di ricreare comportamenti molto più complessi tramite i meccanismi di overriding e overloading

3.4.1 Overriding

Nel momento in cui una classe B estende una classe A, è possibile "sovrascrivere" un metodo di A in B, in modo tale da modificare il comportamento di quest'ultimo a seconda che venga utilizzato da un'istanza di tipo A o B. Per utilizzare l'overriding devo ridefinire

un metodo nella subclass, assicurandomi che questo abbia la [1]stessa signature e stesso valore di ritorno. Supponiamo di avere:

```
class A{
  void stampa(){
    System.out.println("Hello from A");
  }
}

class B extends A{
  void stampa(){
    System.out.println("Hello from B");
  }
```

Allora, all'interno del main posso fare qualcosa di questo tipo:

```
A a = new A();
a.stampa();
B b = new B();
b.stampa();

/*
Hello from A
Hello from B
*/
```

3.4.2 Overloading

L'overloading è un altro modo che abbiamo di far assumere comportamenti diversi al programma in base al contesto. Per utilizzare l'overloading devo creare due funzioni con lo stesso nome, ma parametri diversi. Anche il valore di ritorno può differire. Vediamo un esempio:

```
public class Overloading {
  public static void main(String[] args) {
    stampa();
    stampa("Input");
    stampa(5);
    int v[] = {1,2,3,4,3,2,1};
    System.out.println(stampa(v));

  }

  public static void stampa() {
    System.out.println("0 argomenti");
  }

  public static void stampa(String text) {
```

[1]Con signature o firma si intende il nome della funzione e i parametri presi in input

```
    System.out.println("Stampo ciò che mi è stato dato in input:\n" +
    text + "\n");
  }

  public static void stampa(int x) {
    for (int i = 0; i < x; i++) {
      System.out.println(i + 1);
    }
  }

  public static int stampa(int v[]){
    System.out.println("");
    for(int x : v)
      System.out.print(x + " ");
    return v.length;
  }
}
```

Nota come

- Le funzioni qui sono all'interno della stessa classe. Ciò che determina quale versione viene chiamata è il numero ed il tipo degli argomenti
- Il valore di ritorno può cambiare, tuttavia non può essere l'unica cosa diversa fra una funzione e l'altra (quale verisone chiamo?)
- Le funzioni non devono necessariamente essere nella stessa classe

3.4.3 Polimorfismo

Overriding e overloading sono due metodi che abbiamo per applicare il concetto di polimorfismo.

> In informatica, il termine polimorfismo ("avere molte forme") viene usato in senso generico per riferirsi a espressioni che possono rappresentare valori di diversi tipi (dette espressioni polimorfiche).

In particolare si può distinguere fra:

- Polimorfismo sui dati: informalmente, ciò che ci permette di fare `A a = new B();` se B estende A
- Polimorfismo sui metodi: ciò che possiamo ottenere tramite overloading e overriding.

3.5 Esercizi

Esercizio 13: *Gerarchia di animali*

Scrivi del codice per rappresentare una gerarchia di animali. Le classi da rappresentare sono:

- `Animale`. Ha un `verso`, `peso`, `dimensione` e `sesso`. La dimensione è un enum che può assumere tre valori
- `Mammifero`
- `Rettile`
- `Delfino`
- `Lucertola`

Scrivere poi una funzione `toString`, che ritorni una stringa contenente le informazioni riguardo l'animale.

Implementare poi due classi: `cavallo` e `asino`. Creare una funzione `accoppia` che stampi a video il corretto incrocio:

	cavallo	cavalla
asino	/	mulo
asina	bardotto	/

3.6 Tipo statico, dinamico e dynamic binding

Fino ad ora, quando abbiamo parlato di "tipo" di una variabile ci siamo riferiti al tipo statico di questa. Tuttavia, supponendo che `B extends A`, e dichiarando:

```
A x = new B();
```

qual'è il tipo di `x`?

Nel caso precedente, il tipo statico di `x` è `A`, mentre quello dinamico è `B`. Potremmo quindi affermare che:

- Il tipo statico, o semplicemente "tipo", è il tipo indicato nella parte sinistra della dichiarazione della variabile
- Il tipo dinamico è il tipo effettivo del valore contenuto nella variabile. Il tipo dinamico può cambiare infatti a *runtime*

La possibilità di avere "un tipo dinamico", ci permette di creare comportamenti interessanti e talvolta difficili da capire, soprattutto tramite overriding e overloading

3.6.1 Dynamic bining

Partiamo vedendo i seguenti esempi di codice:

```
public class es1 {

   public static void main(String[] args){
    A a = new B();
    a.stampa();
   }

}

class A {
  public void stampa(){
    System.out.println("A");
  }
}

class B extends A{
  public void stampa(){
    System.out.println("B");
  }
}
```

```
  public class es2 {

  public static void main(String[] args) {
    A x = new B();
    stampa(x);
  }

  public static void stampa(A a) {
    System.out.println("A");
  }

  public static void stampa(B b) {
    System.out.println("A");
  }

}

class A {
}

class B extends A {
}
```

Quale sarà l'output in questi casi?

```
public class es3 {

  public static void main(String[] args) {
    A x = new B();
```

```
    x.funcB();
  }

}

class A {
  public void funcA() {
    System.out.println("funcA");
  }
}

class B extends A {
  public void funcB() {
    System.out.println("funcB");
  }

}
```

In questo caso, invece, come mai ci viene dato un errore? Se invece apportassi una piccola modifica alla linea 5?

```
public class es3 {

  public static void main(String[] args) {
    A x = new B();
    ((B)x).funcB();
  }

}

class A {
  public void funcA() {
    System.out.println("funcA");
  }
}

class B extends A {
  public void funcB() {
    System.out.println("funcB");
  }

}
```

Riassumendo i comportamenti mostrati in questi pezzi di codice possiamo dire che:

- Il tipo statico determina:
  - Le funzioni che posso chiamare tramite un'istanza
  - L'overload che viene selezionato nel momento in cui una funzione prende in input una istanza
- Il tipo dinamico determina:

– Quale override della funzione viene selezionato (fra tutti, viene chiamata l'override "più recente")

3.6.2 Casting

Nota che tramite casting io posso "forzare" il tipo statico di una variabile. Se è una variabile di tipo `A` ha un tipo dinamico `B`, allora facendo il casting nel seguente modo posso accedere ai metodi e agli attributi dichiarati in `B`. Posso ad esempio fare:

```
public class es3 {

  public static void main(String[] args) {
    A x = new B();
    ((B)x).funcB();
  }

}

class A {
  public void funcA() {
    System.out.println("funcA");
  }
}

class B extends A {
  public void funcB() {
    System.out.println("funcB");
  }

}
```

Attenzione: se si prova a castare una variabile ad un tipo dinamico che non coincide con quello corrente si ottiene un errore a runtime!

3.7 Esercizi

1.
```
public class Test {
    public static void main(String[] args) {
        A obj = new B();
        obj.m(new D());
    }
}
class A {
    void m(C c) { System.out.println("g"); }
}
class B extends A {
    void m(C c) { System.out.println("h"); }
    void m(D c) { System.out.println("i"); }
}
class C { }
```

```
class D extends C { }


  public class Test {
    public static void main(String[] args) {
        A obj = new B();
        ((B)obj).m(new D());
    }
}
class A {
    void m(C c) { System.out.println("g"); }
}
class B extends A {
    void m(C c) { System.out.println("h"); }
    void m(D c) { System.out.println("i"); }
}
class C { }
class D extends C { }
```

2. 3

3.7.1 Link a esercizi carini

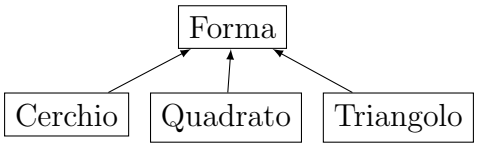
https://pg2.giuliopime.dev/home

- Username: abbiamovinto
- Password: defacto

https://training.olinfo.it

3.8 Classi astratte

Spesso, abbiamo bisogno di una classe che fornisca una "base", che vada poi espansa tramite altre classi che la estendono. Consideriamo ad esempio il class diagram seguente:

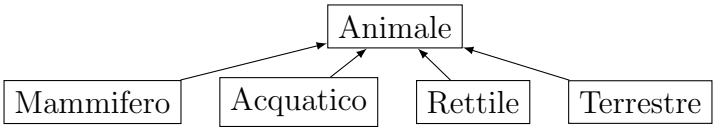


In questo caso non ha senso avere un'istanza di tipo `Forma`. Una forma, all'interno della nostra applicazione sarà sicuramente un `Cerchio`, `Quadrato` o `Triangolo`. Per questo instanzieremo oggetti di questo tipo, non di tipo `Forma`. In questa situazione ha senso che la classe `Forma` sia astratta

La keyword `abstract` può essere usata in due contesti:

- Davanti alla definizione di una classe, per indicare che questa non può essere istanziata
- Davanti ad un metodo di una classe astratta, per indicare che questo non ha implementazione, ma verra overridato nelle classi che estenderanno la classe astratta

Un altro esempio di classe astratta può essere la classe `Animale`, nel seguente contesto:



3.8.1 Interfacce

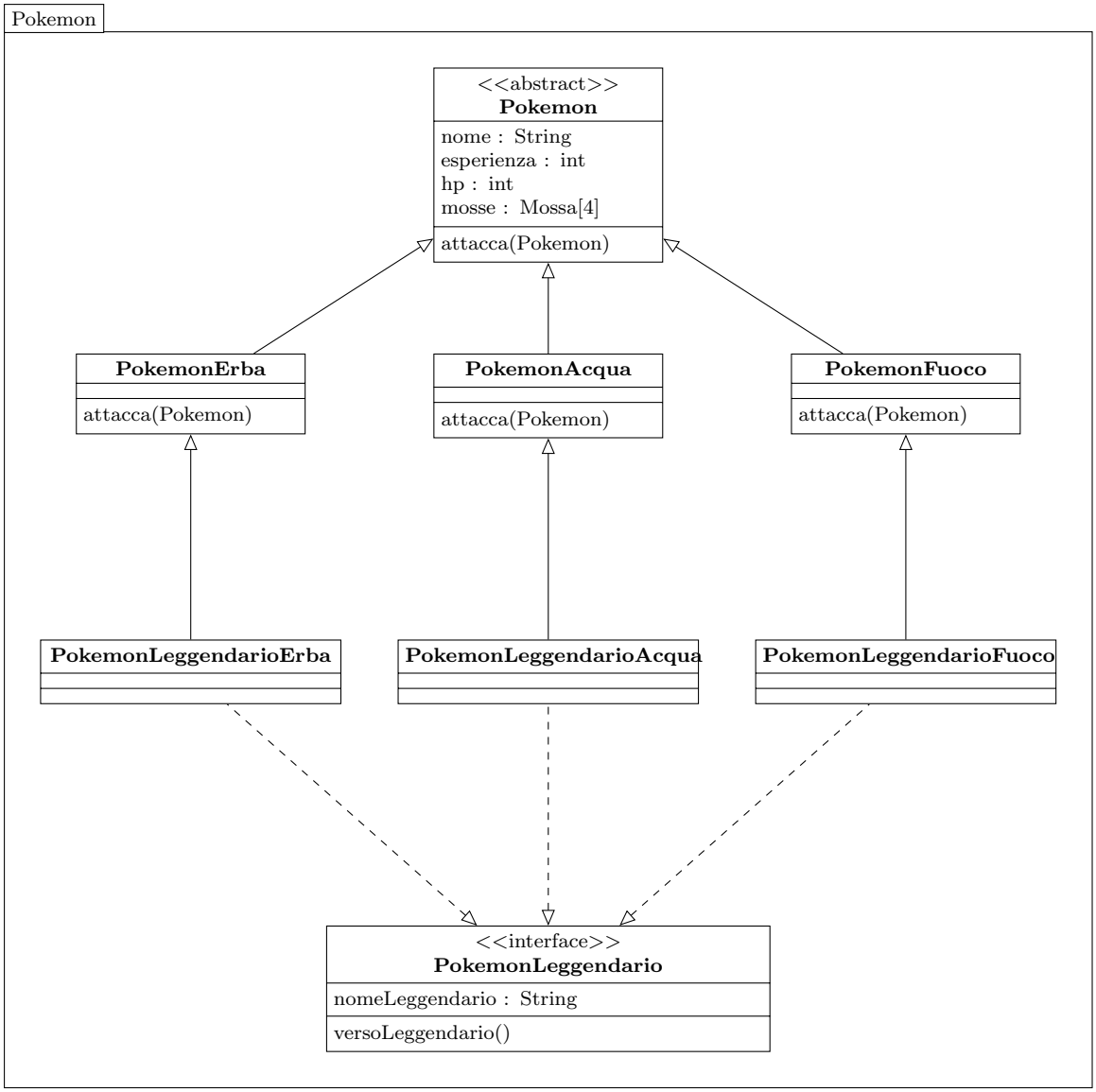
L'interfaccia è una sorta di *"classe completamente astratta"*, ossia una classe i cui metodi non hanno implementazione. In un certo senso, un'interfaccia ci permette di definire appunto dei metodi attraverso i quali possiamo *interfacciarci* alla classe che la implementa, la quale dovrà necessariamente fornire implementazione

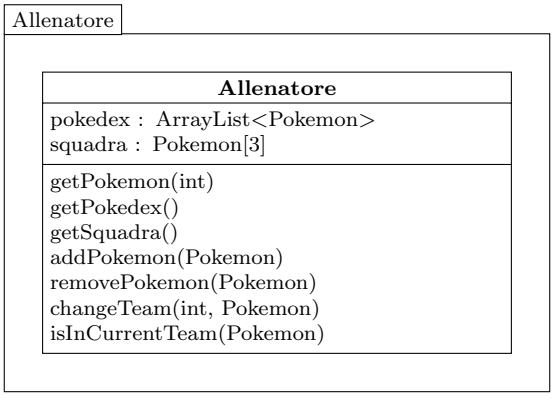
3.9 Esercizi

Esercizio 14: *Esercizio pokemon*

Scrivere un programma che simuli il videogioco di pokemon

Indizio: seguire il seguente *class diagram*


Pokemon
<<abstract>>
Pokemon
nome : String
esperienza : int
hp : int
mosse : Mossa[4]
attacca(Pokemon)
PokemonErba
attacca(Pokemon)
PokemonAcqua
attacca(Pokemon)
PokemonFuoco
attacca(Pokemon)
PokemonLeggendarioErba
PokemonLeggendarioAcqua
PokemonLeggendarioFuoco
<<interface>>
PokemonLeggendario
nomeLeggendario : String
versoLeggendario()


Allenatore
Allenatore
pokedex : ArrayList<Pokemon>
squadra : Pokemon[3]
getPokemon(int)
getPokedex()
getSquadra()
addPokemon(Pokemon)
removePokemon(Pokemon)
changeTeam(int, Pokemon)
isInCurrentTeam(Pokemon)

4 Introduzione algoritmi

Nella risoluzione di problemi di programmazione, spesso vengono utilizzate delle idee di base simili. Possiamo dire che gli algoritmi possono essere classificati in base al tipo di tecnica che utilizziamo per risolvere il problema.

Queste tecniche sono spesso indispensabili per questioni di efficienza (un algoritmo inefficiente potrebbe impiegare troppo tempo per essere eseguito, rendendolo inutilizzabile). Noi vedremo alcune delle più importanti

4.1 Misurare l'efficienza

I problemi di programmazione, tendenzialmente, non consistono in altro che implementare una funzione che, presi in input determinati dati, ne restituisce altri in output. Un fattore da importantissimo, da tenere in considerazione sempre, tuttavia, è la quantità di cicli di calcolo che il programma deve fare, in base all'input che ci viene dato. Vediamo un esempio:

```
class esempio1 {
public static void main(String[] args) {
  int v[] = { 1, 2, 3, 5, 2, 9, 11, -1 };
  System.out.println(massimo(v));
}

public static int massimo(int v[]) {
  int max = v[0];
  for (int i = 1; i < v.length; i++) {
    if (max < v[i])
      max = v[i];
  }

  return max;
}
}
```

In questo caso, l'algoritmo(contenuto nella funzione `massimo`) cerca il valore massimo nel vettore. Domande

- Qual'è la dimensione dell'input?
- Se l'input è di dimensione n, quanti cicli deve fare la funzione `massimo`?

In questo caso, la dimensione dell'input è costituita dalla dimensione del vettore `v`, ossia 8.

Per come abbiamo scritto l'algoritmo, dato in input un vettore di dimensione 8, il ciclo verrà ripetuto 8 volte. Più in generale, dato un input un vettore di dimensione n, l'algoritmo eseguirà n cicli per restituire un risultato.

Formalmente, di dice che l'algoritmo che abbiamo scritto ha complessità $O(n)$

4.2 Notazione O

Per indicare la complessità di un algoritmo, è molto usata la notazione O (*big* O).

Definizione 1: *Notazione big O*

La notazione *big* O serve per indicare la complessita di un algoritmo in funzione della dimensione del input che viene fornito al programma. In particolare, va indicata una funzione fra parentesi, che descrive il numero di cicli che il programma deve eseguire nel peggiore dei casi, ad esempio:

$$O(n) \quad O\left(n^2\right) \quad O(n \log n) \quad O\left(\sqrt{n}\right)$$

Ad esempio, dire che un algoritmo ha complessità $O\left(n^2\right)$ significa che dato in input un dato di dimensione n, impiegherà nel peggiore dei casi n^2 cicli per portare a termine l'esecuzione

Nella definizione data sopra, ci potrebbero essere 2 cose che creano dei dubbi:

- Cosa significa che la notazione esprime il numero di cicli eseguiti *"nel peggiore dei casi"*?
- Come va identificata la dimensione dell'input?

4.2.1 Chiarificazione - "nel peggiore dei casi"

Per rispondere alla prima domanda vediamo un frammento di codice:

```
class esempio2 {
  public static void main(String[] args) {

    int v[] = { 1, 2, 3, 4, 5 };
    System.out.println(cointains(5, v));
    System.out.println(cointains(1, v));
    System.out.println(cointains(-2, v));
  }

  public static boolean cointains(int element, int v[]) {
    for (int i = 0; i < v.length; i++) {
      if (v[i] == element)
        return true;
    }
    return false;
  }
}
```

La funzione `contains` ritorna true se `element` è contenuto nell'array, altrimenti `false`

- Nella prima chiamata la funzione esegue 5 cicli
- Nella seconda chiamata la funzione esegue 1 solo ciclo
- Nell'ultima chiamata la funzion e esegue 5 cicli

Qual è la complessità dell'algoritmo?

Nonostante il numero di cicli cambi in base a come è formato l'input, l'algoritmo ha lo stesso complessità $O(n)$, perché nel peggior caso, dato un input di dimensione n, esegue n cicli

4.2.2 Dimensione dell'input

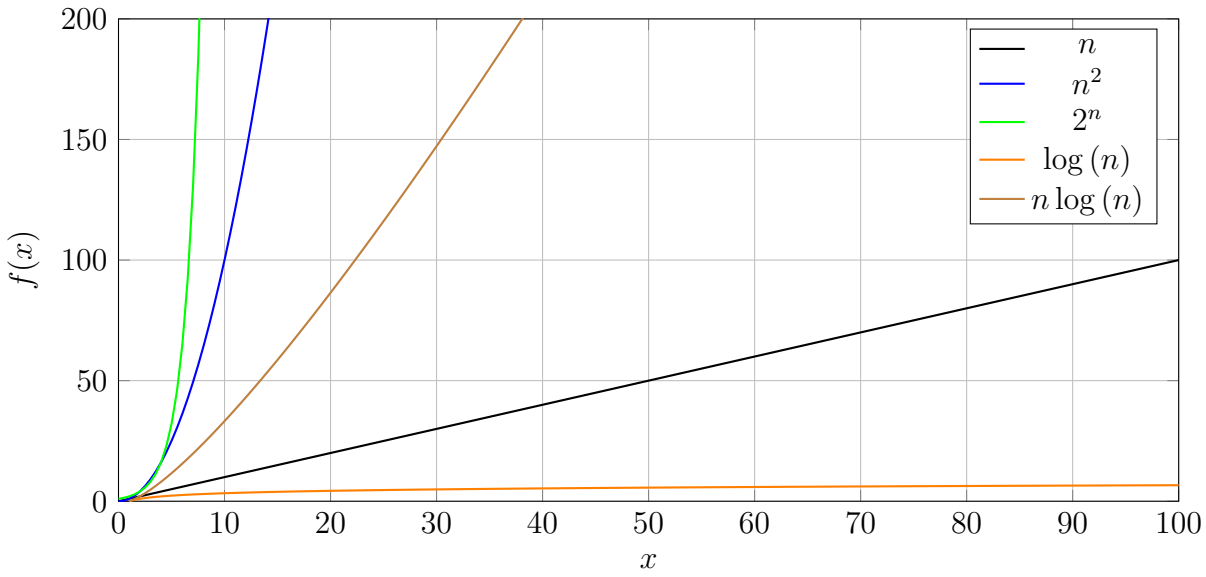
La dimensione dell'input, non è necessariamente la lunghezza di un array, possiamo prendere come esempio il seguente algoritmo che somma tutti i numeri interi da 0 a n:

```
class esempio3 {
  public static void main(String[] args) {
    System.out.println(sumUpTo(10));
  }

  public static int sumUpTo(int n) {
    int sum = 0;

    for (int i = 1; i <= n; i++)
      sum += i;

    return sum;
  }
}
```

In questo caso, l'algoritmo ha complessità $O(n)$, dove n però, non è la lunghezza di un vettore, ma il valore di n, dato che è quello che influenza il numero di cicli eseguiti. Solitamente è evidente quale dimensione determina il numero di cicli eseguiti

4.3 Complessità comuni

La maggior parte di algoritmi ha una complessità fra le seguenti:



4.4 Esercizi di base

Esercizio 15: *Massimo vettore*

Dato un vettore `v`, trovare l'elemento di valore massimo

Esercizio 16: *Compara stringhe 1*

Date due array di caratteri `s1` e `s2`, ritornare `true` se sono uguali, `false` altrimenti

Esercizio 17: *Compara stringhe 2*

Date due stringhe `s1` e `s2`, ritornare i seguenti valori:

- -1 se `s1` è alfabeticamente minore di `s2`
- 1 se `s1` è alfabeticamente maggiore di `s2`
- 0 se le stringhe sono uguali

Esercizio 18: *Fattoriale*

Dato in input un intero positivo `n`, se ne calcoli il fattoriale

Esercizio 19: *Massimo ricorsivo*

Creare una funzione ricorsiva che trovi l'elemento massimo di un vettore

Esercizio 20: *Controllo contenuto vettore*

Dati due vetttori `v1` e `v2` di dimensione uguale, ritornare `true` se contengono gli stessi elementi, `false` altrimenti (chiaramente, l'ordine degli elementi contenuti può cambiare)

Nota

- I vettori devono contenere gli stessi elementi nella stessa quantità
- Si assuma che i valori contenuti nel vettore siano interi nel range $[0, 100)$

Input	Output	Discussione
v1: 1 1 1 v2: 1 1	false	Stessi elementi ma in quantità diversa
v1: 1 1 2 1 -5 v2: -5 1 2 1 1	true	Stessi elementi nella steessa quantità

Esercizio 21: *Stampa matrice*

Creare una funzione che presa in input una matrice di interi `m`, la stampi

Esercizio 22: *Inverti matrice*

Creare una funzione che presa in input una matrice di interi `m`, ritorni un'altra matrice con righe e colonne invertite:

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 & 15 \\ 16 & 17 & 18 & 19 & 20 \\ 21 & 22 & 23 & 24 & 25 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 6 & 11 & 16 & 21 \\ 2 & 7 & 12 & 17 & 22 \\ 3 & 8 & 13 & 18 & 23 \\ 4 & 9 & 13 & 19 & 23 \\ 5 & 10 & 15 & 20 & 25 \end{bmatrix}$$

5 Programmazione dinamica

Una delle tecniche più importanti, per risolvere problemi di programmazione, è detta programmazione dinamica.

La tecnica risolutiva della programmazione dinamica consiste nella scomposizione di un problema in sottoproblemi più semplici i cui risultati vengono memorizzati e combinati per ottenere la soluzione del problem originario

Dalla definizione questa tecnica sembra estremamente generica e vaga, vediamo degli esempi per rendere tutto più chiaro.

5.0.1 Esempio programmazione dinamica 1

Un esempio di problema risolvibile tramite programmazione dinamica è il calcolo del di un numero di fibonacci. La sequenza di fibonacci è così definita:

$$\text{Fib}(n) = \begin{cases} \text{Fib}(n-1) + \text{Fib}(n-2) & x > 2 \\ 1 & x = 2 \\ 0 & x = 1 \end{cases}$$

```
class esempio1 {
  public static void main(String[] args) {
    System.out.println(fibonacci(13));
  }

  public static int fibonacci(int n) {
    int fib[] = new int[n];
    fib[0] = 0;
```

```
    fib[1] = 1;
    for (int i = 2; i < n; i++) {
      fib[i] = fib[i - 1] + fib[i - 2];
    }

    return fib[n - 1];
  }
}
```

In questo è chiaro chiaro come possiamo salvare i risultati intermedi per poi riutilizzarrli: per calcolare $\text{Fib}(n)$ utilizziamo $\text{Fib}(n-1)$ e $\text{Fib}(n-2)$ per questo salviamo tutti i risultati nel vettore `fib`

Possiamo ora affrontare più formalmente i requisiti necessari per poter affrontare un problema tramite la tecnica della programmazione dinamica

5.1 Quando affrontare un problema tramite programmazione dinamica

A grandi linee, abbiamo bisogno che:

- Il problema sia scomponibile in sottoproblemi più semplici da risolvere
- I sottoproblemi siano "sovrapposti" (overlapping subproblems), ossia che la soluzione del sottoproblema $n-1$ serva per risolvere il sottoproblema n
- Vi siano dei sottoproblemi che siano *"casi base"*, la cui soluzione sia immediata. A partire da questi si possono costruire tutte le altre soluzioni

Collegando questi requisiti al prolema di fibonacci visto in precedenza, avremo che:

- Sottoproblemi: calcolo $\text{Fib}(n-1), \text{Fib}(n-2), \ldots, \text{Fib}(2), \text{Fib}(1)$
- Overlapping subproblems: i problemi sono "overlapping" in quanto per calcolare $\text{Fib}(n)$ ho bisogno di $\text{Fib}(n-1)$ e $\text{Fib}(n-2)$. Per risolvere un sottoproblema ho bisogno delle soluzioni di altri sottoproblemi
- Esistenza casi base: nel caso in cui debba calcolare $\text{Fib}(1)$ e $\text{Fib}(2)$ la soluzione è immediata

5.2 Problemi classici di programmazione dinamica

Esercizio 23: *Sottosequenza massima (Kadane's problem) (link)*

Dato in input un vettore `v`, contenente interi (anche negativi), si trovi la [a]sottosequenza che abbia somma degli elementi massima. Si ritorni quest'ultima

Input

La dimensione n del vettore e sulla nuova riga gli elementi del vettore separati da uno spazio

Output

La somma degli elementi della sottosequenza con somma massima

Input	Output	Discussione
5 1 2 3 4 5	15	La sottosequenza è data dall'intero vettore
8 -2 -3 4 -1 -2 1 5 -3	7	La sottosequenza è data dall'intervallo $[2, 6]$
4 -2 -3 -1 -11	0	Si assuma che la sottosequenza nulla abbia somma 0

Complessità ottimale: $O(v.size)$

[a]Una sottosequenza è un insieme di elementi adiacenti all'interno del vettore

Un approccio naif sarebbe quello di generare tutte le sottosequenze possibili e confrontarne la somma, stampando quella massima. Questo approccio tuttavia sarebbe davvero inefficiente, tuttavia è molto semplice da implementare:

```
public static int subsequenceIneff(int v[]) {

int max = 0;

for (int i = 0; i < v.length; i++) {

  int currSum = 0;
  for (int j = i; j >= 0; j--) {
    currSum += v[j];
    if (currSum > max)
      max = currSum;
  }

}

return max;

}
```

Complessità: $O(n^2)$

Un approccio più efficiente può essere implementato tramite programmazione dinamica. L'idea è la seguente:

- Salvo la sottosequenza con somma maggiore che finisce in posizione i-esima
- Calcolo la sottosequenza con somma maggiore che finisce in pos $i+1$ utilizzando la sottosequenza con somma maggiore che finisce in pos i. Chiamiamo questo vettore `dp`

Immaginiamo di salvarci i risultati intermedi in un vettore: questo vettore avrà dimensione n e conterrà in posizione i il sottovettore di somma massima che finisce in posizione i. Notiamo innanzitutto che calcolare `dp[0]` è scontato:

- Se `v[0] > 0` allora il sottovettore è costituito da un singolo elemento, ovvero `v[0]`
- Se `v[0] <= 0` allora il sottovettore è il sottovettore nullo, il quale ha sempre somma 0

Per calcolare invece `dp[i]`, ragiono nel seguente modo:

- Se `dp[i-1] + v[i] > 0` allora `dp[i] = dp[i-1] + v[i]` (mi conviene prendere la miglior sottosequenza che termina nella posizione prima e sommarci `v[i]`, anche se questo è negativo)
- Se `dp[i-1] + v[i] <=0` allora `dp[i] = 0` (conviene "ripartire a formare il vettore", dato che concatenando la subsequence con somma maggiore che termina in $i-1$ aggiungerei solo una quantità negativa)

Esercizio 24: *Cuttinig rod*

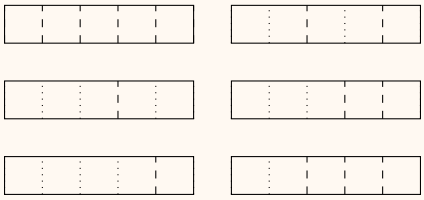
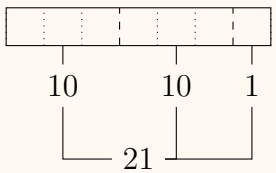
Dato un cilindro di lunghezza `n` e un vettore `v` di dimensione `n`, che in `v[i]` contenga il prezzo di un cilindro lungo `i+1`, si stampi il prezzo massimo che posso ottenere tagliando il cilindro in quante parti voglio

Input

La dimensione n(la lunghezza del cilindro) e sulla nuova riga gli n interi positivi che costituiscono gli elementi di v(ossia i prezzi di ogni taglio di cilindro)

Output

Il prezzo massimo che posso ottenere suddividendo il cilindro

Input	Output	Discussione
5 1 2 3 4 5	5	Posso tagliare il cilindro in molti modi per ottenere il valore 5: 
7 1 4 10 8 5 10 13	21	In questo caso ciò che conviene fare è spezzare il cilindro in 2 pezzi di lunghezza 3 e 1 di lunghezza 1: 

Complessità ottimale: $O\left(n \cdot v.size\left()\right)\right)$

L'idea di base per risolvere il problema è la seguente:

- Creo un vettore `dp` all'interno del quale salvo nella *i-esima* cella il valore massimo che posso ottenere suddividendo un cilindro di lunghezza $i+1$
- Costruisco il vettore `dp` partendo dal caso base: `dp[0] = prezzi [0]`, in quanto ho un solo modo di suddividere un cilindro lungo 1
- Contanto sul fatto che il vettore `dp` contenga il il prezzo maggiore che posso ottenere suddividendo il cilindro in un dato modo, calcolo `dp[i+1]` sfruttando i dati contenuti nelle celle precedenti del vettore `dp`.In particolare, supponendo di dover calcolare la posizione i del vettore `dp`, devo:

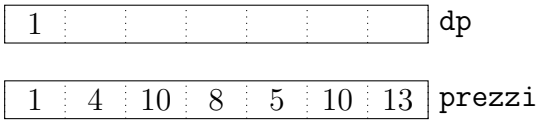
- Calcolare il prezzo che otterrei mettendo in posizione i un pezzo di ogni lunghezza, da 1 a i
- Confrontare i prezzi ottenuti
- `dp[i]` è il valore massimo fra questi prezzi

Vediamo un esempio grafico. Supponiamo di avere in input il vettore `prezzi`, con valori:

1 4 10 8 5 10 13

Ripercorriamo gli step appena descritti.

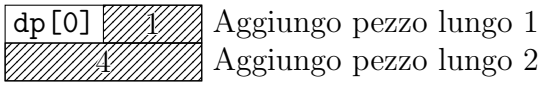
- Creo un vettore `dp` all'interno del quale salvo nella *i-esima* cella il valore massimo che posso ottenere suddividendo un cilindro di lunghezza $i + 1$
- Costruisco il vettore `dp` partendo dal caso base: `dp[0] = prezzi [0]`, in quanto ho un solo modo di suddividere un cilindro lungo 1. Quindi nel nostro caso:



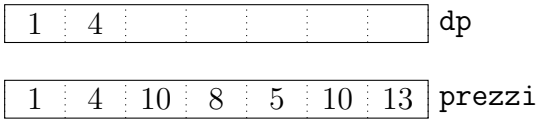
  Chiaramente, un cilindro id lunghezza 1 non può essere suddiviso, quindi l'unico prezzo possibile è il prezzo del cilindro lungo 1, ossia 1 nel nostro caso

- Contanto sul fatto che il vettore `dp` contenga il il prezzo maggiore che posso ottenere suddividendo il cilindro in un dato modo, calcolo `dp[i+1]` sfruttando i dati contenuti nelle celle precedenti del vettore `dp`.In particolare, supponendo di dover calcolare la posizione i del vettore `dp`, devo:
  - Calcolare il prezzo che otterrei mettendo in posizione i un pezzo di ogni lunghezza, da 1 a i
  - Confrontare i prezzi ottenuti
  - `dp[i]` è il valore massimo fra questi prezzi

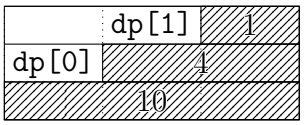
  Iniziamo quindi calcolando `v[1]`. So di avere già calcolato il prezzo migliore per suddividere un cilindro di lunghezza 1. Quindi per arrivare ad avere un cilindro di lunghezza 2 ho due alternative:

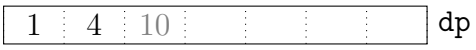


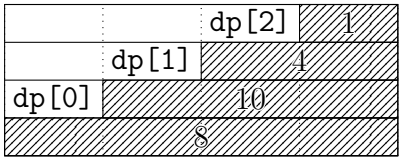
  Ora devo decidere se mi convenga prendere un solo cilindro da 2 oppure suddividerlo in due pezzi da 1. Mi basta però vedere quale delle opzioni ha prezzo maggiore. In questo caso conviene prendere un pezzo da 2 con costo 4. Dp diventa:

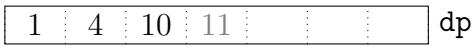


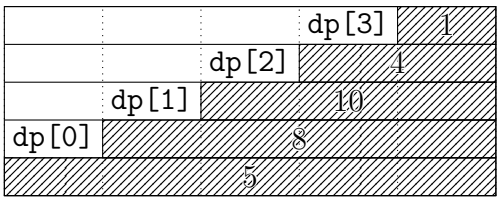
Ripetiamo il passaggio per $i = 3$

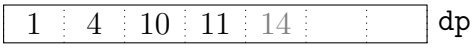


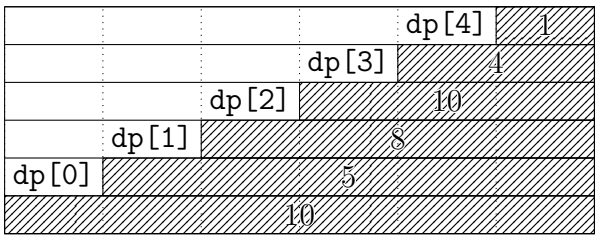


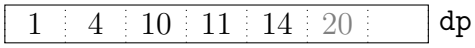


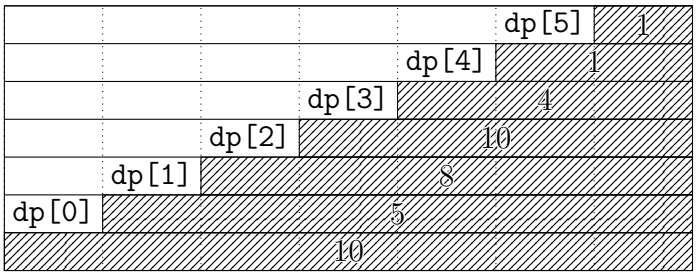


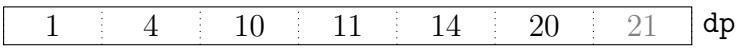












Esercizio 25: *Somma e media (link)*

Dati in input un numero N, e successivamente N interi, calcolare la media aritmetica e la somma di questi ultimi

Input:

Sulla prima riga l'intero N, sulla seconda riga N interi separati da uno spazio

Output:

Due interi: rispettivamente la somma degli N numeri e la loro media aritmetica

Input	Output	Discussione
1 12	5	Somma e media coincidono e hanno valore 12
7 1 2 34 -56 33 23 89	126 18	Somma e media coincidono e hanno valore 12

Complessità ottimale: $O(N)$

Esercizio 26: *Majority element (link)*

Dato un array *nums* di dimentione n, ritornare il *majority element*. Il *majority element* è l'elemento che appare di più di $\frac{n}{2}$ volte. Si può assumere che l'elemento esista sempre nell'array

Input:

Sulla prima riga l'intero n, sulla seconda riga n, ossia gli elementi di *nums*

Output:

Un intero, il majority element

Input	Output	Discussione
3 3 2 3	3	3 appare più di $3/2 = 1$ volta
7 2 2 1 1 1 2 2	2	2 appare più di $7/2 = 3$ volte

Complessità ottimale: $O(n)$

Esercizio 27: *Longest Common Subsequence (link)*

Date in input due stringhe *S1* ed *S2*, si ritorni la lunghezza della *longhest common subsequence*, ossia della [a]sottosequenza più lunga comunque ad esntrambe le stringhe.

Input:

Due stringhe, una per riga, composte da caratteri maiuscoli compresi fra A e Z

Output:

Un intero, la lunghezza della più lunga sottosequenza comune ad entrambe le stringhe

Input	Output	Discussione
AGGTAB GXTXAYB	4	La sottosequenza comune con lunghezza maggiore è *"GTAB"*, ed ha lunghezza pari a 4
AABBCCD AABBD	5	La sottosequenza comune con lunghezza maggiore è *"AABBD", ed ha lunghezza pari a 5*

Complessità ottimale: $O\left(s_1.lenght \cdot s_2.lenght\right)$

[a]Con sottoseuquenza si intende la una stringa che si può ottenere da un'altra eliminando determinati caratteri: *bedbreakfast* è una sottosequenza di *bedandbreakfast*. A differenza di ciò che accade in un sottovettore, i caratteri non devono essere necessariamente contigui: *bedbreakfast* è sottosequenza ma non sottovettore; *breakfast* è sottosequenza e sottovettore

Per risolvere questo problema dobbiamo utilizzare una matrice di supporto, che salverà i valori intermedi e ci permetterà di utilizzare la programmazione dinamica. Prendiamo come esempio il le stringhe *"AGGTAB"* e *"GXTXAYB"*. La matrice di supporto deve avere la seguente forma:

		G	X	T	X	A	Y	B	s1, j
	0	0	0	0	0	0	0	0	
A	0								
G	0								
G	0								
T	0								
A	0								
B	0								
s2, i									

Quindi la struttura della matrice è la seguente:

- Un orentamento rappresenta una stringa, l'altro l'altra (nota che le stringhe non sono salvate nella matrice, sono riportate in figura solo per rendere il procedimento più chiaro)
- La prima colonna e la prima riga sono riempite di zeri. Questo serve perché ci permette di evitare di incappare in indici negativi quando eseguiremo l'algoritmo
- Siano `s1` e `s2` le stringhe, nella cella di indice (i, j), salveremo la lunghezza della longest common subsequence per `s1.substring(0,i)` e `s2.substring(0,j)`

		G	X	T	X	A	Y	B	s1, j
	0	0	0	0	0	0	0	0	
A	0								
G	0						(1,6)		
G	0								
T	0								
A	0								
B	0								

s2, i

Ad esempio, nella cella evidenziata, con indice $(1, 6)$, va salvata la lunghezza della LCS delle stringhe "GXTXAY" e "AG"

Detto questo, possiamo riempire la tabella secondo i seguenti criteri:

- Se `s1[i-1] == s2[j-1]` ciò significa che la soluzione ottimale per quel sottoproblema è data dalla soluzione ottimale per il sottoproblema con le medesime stringhe senza però questultimo carattere. La soluzione di questo problema si trova nella cella $(i - 2, j - 2)$
- Se `s1[i-1] != s2[j-1]` allora non ho modo di migliorare la lunghezza della LCS aggiungendo un elemento alle sottostringhe dei problemi precedenti. La soluzione ottimale è quindi da calcolare confrontando le soluzioni ottimali precedenti, in particolare sia `dp` la matrice:

  ```
  dp[i][j] = Math.max(dp[i-1][j], dp[i][j-1])
  ```

  Volendo ad esempio calcolare il problema per le sottostringhe "GXT" e "AGGTA" posso partire dalle soluzioni dei sottoproblemi per le stringhe "GX", "AGGTA" e "GXT", "AGGT". Mi basta prendere la maggiore di queste due.

  Nota anche che non mi serve controllare la soluzione del sottoproblema "GX", "AGGT" in quanto colonne e righe sono tutte ordinate in maniera crescente, quindi è impossibile che la cella $(i - 1, j - 1)$ abbia un valore maggiore della cella $(i, j - 1)$ o $(i - 1, j)$

Esercizio 28: *Minimum coin (link)*

Vengono dati in input un array contenente un array `coins` (il quale rappresenta monete di diverso taglio) e un numero intero `ammount` (il quale rappresenta il quantitativo totale di monete). Calcolare il numero minimo di monete che si possono utilizzare per arrivare alla somma `ammount` . Si assuma di avere un numero infinito di monete per ogni taglio.

Input:

Due righe. Sulla prima gli interi `ammount` e `n` (la dimensione di `coins`), mentre sulla seconda gli elementi di `coins` separati da uno spazio. Nota che gli elementi di `coins` non vengono necessariamente dati in ordine crescente

Output:

Un intero, il numero minimo necessario di monete per arrivare ad `ammount`. Se la combinazione non fosse presente ritornare -1

Input	Output	Discussione
11 3 1 2 5	3	Per ottenere la somma 11 posso utilizzare 2 monete da 5 e 1 da 1
4 1 3	-1	Non è possibile ottenere una somma di 4 con sole monete da 3

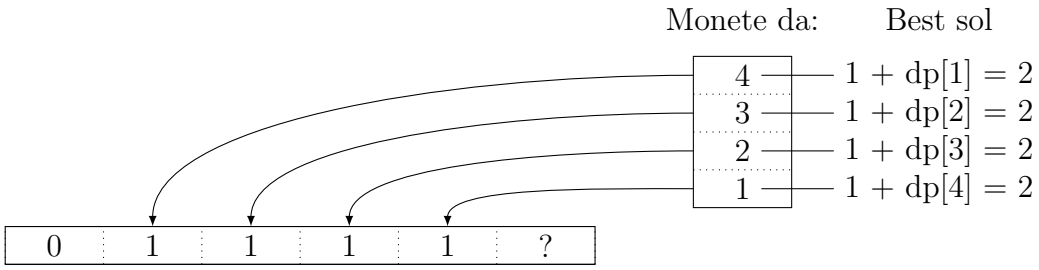
Complessità ottimale: $O\left(coins.size \cdot amount\right)$

Approccio intuitivo (sbagliato): cerco di riempire la somma con monete quanto più grandi possibile. Ad esempio con questo input, tuttavia, non funziona: somma: 20, monete: 15 13 7 1

L'approccio corretto è molto simile al problema *cutting rod*. Di fatto è come se dovessimo "riempire" un cilindro lungo `ammount` con pezzi di dimensioni contenute in `coins`.

Procediamo così:

- Creo vettore `dp` di dimensione `ammount`, all'interno del quale salvo nella cella i il numero minore di monete per creare una somma pari ad i
- `dp[0]=0`, ossia ho modo di creare una somma pari a zero con zero monete
- Per calcolare la cella i-esima del vettore `dp`, devo ragionare nel seuguente modo:
  - Per ogni moneta che abbbia valore inferiore a i, calcoliamo il minor numero di monete che possiamo usare utilizzando il vettore `dp`, in analogia con il problema *cutting rod*. Supponiamo di avere `amount=5, coins=[1, 2, 3, 4]`



La miglior souzione per una somma pari a 5 è quindi 2. Possiamo usare 2 monete in modi diversi ((3,2), (4,1)) per ottenere la somma 5

- Mettendo in ordine il concetto intuitivo dobbiamo creare `dp[i]` mettendo nel seguente modo:
  - Per ogni elemento di `coins` che sia minore di `i` calcolo il numero minimo di monete che è necessario per arrivare ad una somma di `i` utilizzando `dp`. Supponendo di dover includere una moneta di valore `value`, allora il minor numero di monete per arrivare a `i` è dato da

    ```
    dp[i-value] + 1
    ```

  - Il valore minimo di monete per arrivare alla somma `i` è il valore minimo fra tutti quelli calcolari al punto precedente
  - Occhio ai casi nei quali non è possibile ottenere una somma specifica tramite le monete a disposizione. In questi casi metteremo il valore -1 nel vettore `dp`

Esercizio 29: *Unique paths (link)*

Un robot si muove su di una griglia `n x m`. Il robot inizialmente è posizionato sulla cella [0] [0] e deve arrivare alla cella $[m-1]\,[n-1]$. Il robot può muoversi solamente verso il basso e verso destra. Dati due interi `n, m` che indicano la dimensione della griglia, calcolare il numero di percosi possibili

Input

Gli interi `m` e `n`

Output

Il numero di percorsi possibili

Input	Output	Discussione
2 2	2	I percorsi possibili sono $[(R \to D), (D \to R)]$
3 2	3	I percorsi possibili sono $(R \to D \to D)$, $(D \to D \to R)$, $(D \to R \to D)]$

Complessità ottimale: $O(n \cdot m)$

L'idea di base è la seguente:

- Creo matrice `dp` che salva nella generica cella $[i]\,[j]$ il numero di percorsi tramite i quali posso arrivare in $[i]\,[j]$
- Inizializzo la prima colonna e la prima riga della matrice a 1: per raggiungere le celle della prima riga e colonna ho un solo modo, ossia rispettivamente spostarmi a destra o spostarmi in basso (non posso tornare indietro)
- Per ogni cella che avanza calcolo il valore come la somma della cella a sinistra e della cella a sopra: questo perche per arrivare nella cella $[i]\,[j]$ posso passare per la cella $[i-1]\,[j]$ oppure per la cella $[i]\,[j-1]$. La somma dei modi che ho per arrivare nelle suddette celle è il numero di modi che ho per arrivare nella cella corrente
- Una volta generata l'intera tabella, nella ultima cella in basso a destra avro il risultato al problema

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & / & / & / & / & / \\ 1 & / & / & / & / & / & / \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & / & / & / & / \\ 1 & / & / & / & / & / & / \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 4 & / & / & / \\ 1 & / & / & / & / & / & / \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 4 & 5 & / & / \\ 1 & / & / & / & / & / & / \end{bmatrix}$$

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 4 & 5 & 6 & / \\ 1 & / & / & / & / & / & / \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 1 & / & / & / & / & / & / \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & / & / & / & / & / \\ 1 & 3 & / & / & / & / & / \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & / & / & / & / \\ 1 & 3 & 6 & / & / & / & / \end{bmatrix}$$

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 4 & / & / & / \\ 1 & 3 & 6 & 10 & / & / & / \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 4 & 5 & / & / \\ 1 & 3 & 6 & 10 & 15 & / & / \end{bmatrix}$$

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 4 & 5 & 6 & / \\ 1 & 3 & 6 & 10 & 15 & 21 & / \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 1 & 3 & 6 & 10 & 15 & 21 & 28 \end{bmatrix}$$

Esercizio 30: *Unique paths II (link)*

Un robot si muove su di una griglia `n x m`. Il robot inizialmente è posizionato sulla cella $[0]\,[0]$ e deve arrivare alla cella $[m-1]\,[n-1]$. Il robot può muoversi solamente verso il basso e verso destra. Sulla griglia possono essere presenti degli ostacoli attraverso i quali il robot non può passare. Data una matrice `obstacles`, nella quale le celle con valore 1 indicano le celle con ostacoli, trovare il numero di percorsi possibili per arrivare nell'ultima cella in fondo a destra

Input

Gli interi `m` e `n` e nelle `m` righe successiva gli elementi della matrice `obstacles`

Output

Il numero di percorsi possibili

Input	Output	Discussione
3 3 0 0 0 0 1 0 0 0 0	2	I percorsi possibili sono $[(R \to R \to D \to D), (D \to D \to R \to R)]$

Complessità ottimale: $O\,(n \cdot m)$

L'idea è molto simile al problema Unique paths, con l'unica differenza che dobbiamo tenere in considerazione i casi in cui una rotta è preclusa da un ostacolo. Inoltre, anzichè creare una nuova matrice `dp`, possiamo usare direttamente la matrice `obstacles` che ci viene data.

- Riempio la prima colonna e riga della matrice `obstacles` con valore 1 fino al primo ostacolo. Dal primo ostacolo in poi avrò solo celle irraggiungibili. Setto le celle irraggiungibili con valore -1, in quanto usando 1 rischierei di confondere le celle irraggiungibili con le celle raggiungibili da 1 solo cammino
- Calcolo le celle rimanenti con la stessa logica usata in Unique paths, tenendo conto però che:
  - Nel caso ci sia un ostacolo nella cella a sinistra o in alto a quella corrente non posso arrivare da quella direzione. Se non rimane nemmeno una rotta possibile posso impostare il valore della della a -1, in quanto non riesco a ragiungerla in nessun modo
  - Nel caso ci sia un ostacolo nella cella corrente (`obstacles[i][j] == 1`) cambio il valore e metto -1, onde evitare ambiguità come spiegato precedentemente
- Ancora una volta, nella cella $[m-1, n-1]$ ci sarà la soluzione del problema

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \xrightarrow{\text{riempio prima riga e colonna}} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \xrightarrow{\text{cambio ostacolo in -1}} \begin{bmatrix} 1 & 1 & 1 \\ 1 & -1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & -1 & 1 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 \\ 1 & -1 & 0 \\ 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 \\ 1 & -1 & 1 \\ 1 & 1 & 2 \end{bmatrix}$$

6 Ripassone

6.1 Classi

- Una classe è un insieme di metodi (funzioni) e attributi (variabili).
- La classe definisce il modello e la struttura del dato che vogliamo rappresentare. Il dato effettivo si chiama oggetto o istanza
- Quando creaiamo una classe viene chiamato il costruttore
  - Il costruttore ha forma : `public nomeClasse(...){...}`
  - Si può fare qualunque cosa nel costruttore ma di solito si inizializzano gli attributi della classe
  - Se non viene definito esplicitamente, allora le il compilatore ne inserisce uno che non fa nulla

6.2 Ereditarietà

L'ereditarietà è un meccanismo che permette di, per l'appunto, ereditare il codice di una classe all'interno di un'altra.

- Si effettua utilizzando la keyword `extend`
- Se `B extend A`, allora `B` possiede tutti gli attributi ed i metodi di `A`

6.2.1 Keyword super

La keyword `super` può essere usata in una classe che estende un'altra in due modi:

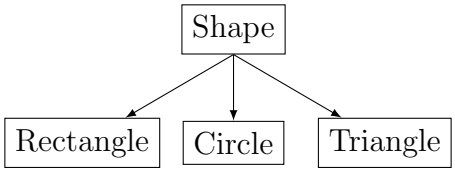
- `super(arg1, arg2...)` chiama il costruttore della super classe
- `super.method(arg1, arg2...)` il metodo indicato della super classe

6.2.2 Principio di sostituzione di Liskov

Se `B extend A`, *allora in un programma ben scritto posso sosituire ogni istanza di* `A` *con una di* `B`*, ottenendo lo stesso comportamento*

6.3 Classi e metodi astratti

Una classe astratta è una classe che non può essere istanziata. Questo serve quando dal punto di vista logico una classe deve per forza essere una delle sue sottoclassi. Ad esempio nella seguente gerarchia:



Ha senso istanziare un oggetto di tipo `Shape`?

Una classe astratta può contenere metodi astratti, ossia metodi che non abbiano un corpo. Questi servono per far si che la loro implementazione sia obbligatoria per ognuna della sottoclassi

6.4 Override e overloading

- **Override:** ridefinisco una funzione con stesso nome, argomenti e valore di ritorno in una sottoclasse
- **Overload:** definisco una funzione che abbia lo stesso nome ma *parametri e valore di ritorno diversi* nella stessa classe o in una classe che estende

Ad esempio, supponiamo di avere due classi `A` e `B` e `B extend A`

`A`	`B`
`f1()` `f1(String s)`	`f1(int a)` `f1(float a)`
`f2()` `f2(String s)`	`f2()` `f2(String s)`

Quando si fa overload, il valore di ritorno non può essere l'unica differenza nella [2]signature delle due funzioni.

6.5 Tipo statico e dinamico

Il tipo statico decreta:

- Quali funzioni posso chiamare tramite un'istanza
- Quale versione dell'override è chiamata

Il cast modifica il tipo statico. Il tipo dinamico decreta invece:

- Quale versione dell'override viene chiamata
- Risultato operatore instanceof

Quindi a tutti gli effetti, il tipo dinamico indica la "conformazione" della zona di memoria indicata da una variabile

[2]Con signature si intende l'insieme di nome di una funzione, tipo e numero dei suoi argomenti

6.5.1 instanceof

`instanceof` è un operatore che ritorna true se il tipo dinamico della prima variabile estende quello della seconda:

```
variabile instanceof Tipo
```

Ritorna `true` se `Type(variabile) extends Tipo`

6.6 Cast

Il cast è un meccanismo per cambiare il tipo statico di una istanza di una classe. Si effettua mettendo davanti ad una variabile il tipo a cui vogliamo castare fra parentesi tonde: `(Tipo)variabile`. Supponendo di avere `B extends A`

- `downcast` (`(B)A`) permette di "convertire" una classe in una sottoclasse. `A -> B`. Può fallire a *runtime* se il tipo dinamico di `A` è più in basso nella gerarchia delle classi rispetto a `B`
- `upcast` permette di "convertire" una classe nella sua superclasse. Non può fallire a patto che `A` appartenga alla stessa gerarchia

6.7 Late binding

Il late binding è il meccanismo per cui, nel momento in cui chiamo un metodo su di una istanza di tipo statico `A` e tipo dinamico `B`, l'override della funzione chiamata è quella definita in [3]`B`

[3] In realtà, viene chiamato l'override presente nella "versione più recente", ossia quella che si trova più il basso nella gerarchia delle classi