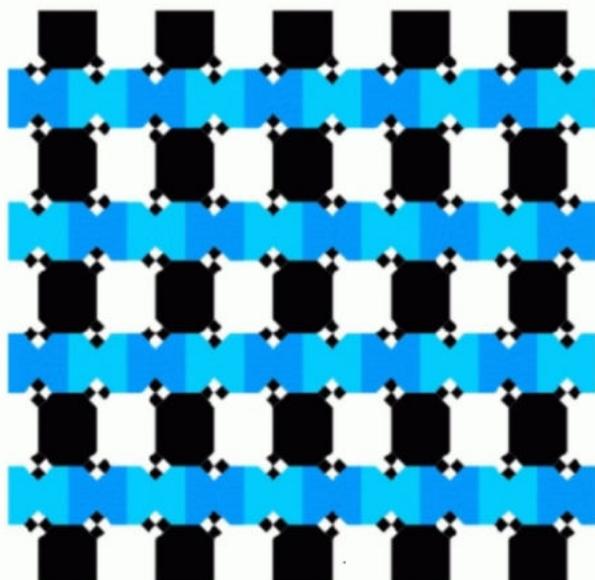




Scala

Introduzione alla Programmazione Parallelia

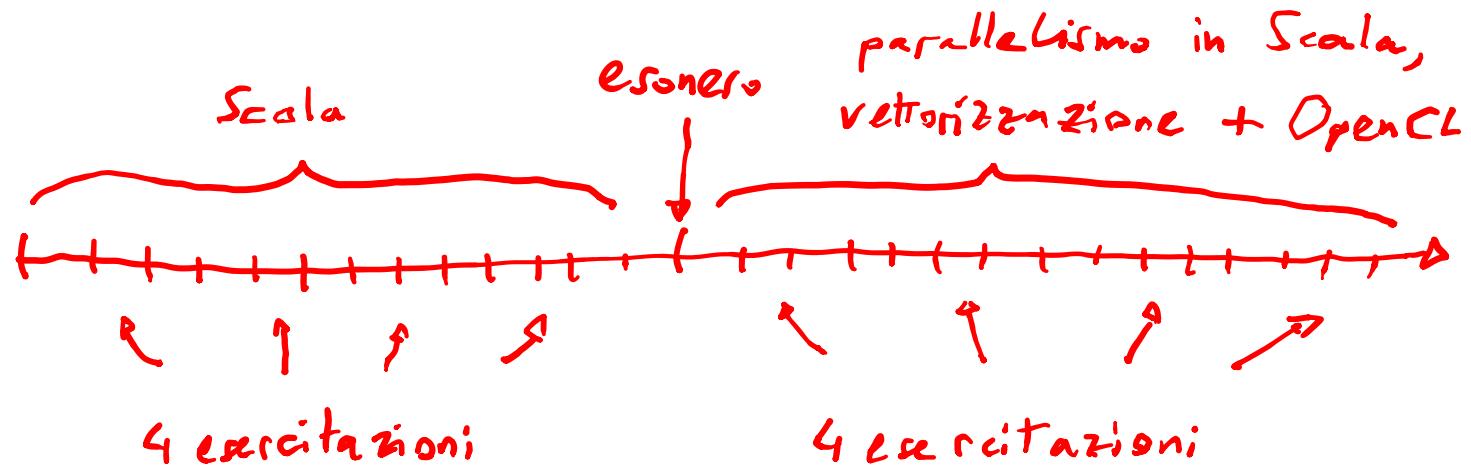
Ultimo aggiornamento: 6 gennaio 2016





Modalità esame

Necessario registrarsi su form online per seguire le esercitazioni (sezione esami sito corso)



Prrova di programmazione in Scala, C, OpenCL



Bonus esercitazioni

Esercitazioni	Voto	Esame													
		18	19	20	21	22	23	24	25	26	27	28	29	30	31
	18	2,50	2,24	1,63	1,01	0,57	0,36	0,28	0,26	0,25	0,25	0,25	0,25	0,25	0,25
	19	2,24	2,50	2,24	1,63	1,01	0,57	0,36	0,28	0,26	0,25	0,25	0,25	0,25	0,25
	20	1,63	2,24	2,50	2,24	1,63	1,01	0,57	0,36	0,28	0,26	0,25	0,25	0,25	0,25
	21	1,01	1,63	2,24	2,50	2,24	1,63	1,01	0,57	0,36	0,28	0,26	0,25	0,25	0,25
	22	0,57	1,01	1,63	2,24	2,50	2,24	1,63	1,01	0,57	0,36	0,28	0,26	0,25	0,25
	23	0,36	0,57	1,01	1,63	2,24	2,50	2,24	1,63	1,01	0,57	0,36	0,28	0,26	0,25
	24	0,28	0,36	0,57	1,01	1,63	2,24	2,50	2,24	1,63	1,01	0,57	0,36	0,28	0,26
	25	0,26	0,28	0,36	0,57	1,01	1,63	2,24	2,50	2,24	1,63	1,01	0,57	0,36	0,28
	26	0,25	0,26	0,28	0,36	0,57	1,01	1,63	2,24	2,50	2,24	1,63	1,01	0,57	0,36
	27	0,25	0,25	0,26	0,28	0,36	0,57	1,01	1,63	2,24	2,50	2,24	1,63	1,01	0,57
	28	0,25	0,25	0,25	0,26	0,28	0,36	0,57	1,01	1,63	2,24	2,50	2,24	1,63	1,01
	29	0,25	0,25	0,25	0,25	0,28	0,36	0,57	1,01	1,63	2,24	2,50	2,24	1,63	1,01
	30	0,25	0,25	0,25	0,25	0,28	0,36	0,57	1,01	1,63	2,24	2,50	2,24	1,63	1,01
	31	0,25	0,25	0,25	0,25	0,28	0,36	0,57	1,01	1,63	2,24	2,50	2,24	1,63	1,01
	32	0,25	0,25	0,25	0,25	0,28	0,36	0,57	1,01	1,63	2,24	2,50	2,24	1,63	1,01



Cos'è la programmazione funzionale PF?

Paradigma di programmazione incentrato
sulle funzioni senza side-effect composte
tra loro per ottenere un certo risultato.

Nel seguito vedremo le caratteristiche della
programmazione funzionale. Introdurremo il linguaggio
Scala che è multi-paradigma e consente
oggetti, classi e side-effect oltre ad essere
un linguaggio funzionale.

PROGRAMMAZIONE FUNZIONALE

è un paradigma di programmazione incentrato sulle funzioni senza side-effect composte tra loro. Vedremo SCALA che è multi-paradigma come si vedrà.

λ perche' associato a pfp?

λ calcolo è sistema formale introdotto negli anni 30 equivalente a macchine di Turing (introdotto da ALONZO CHURCH)

origini delle pfp:

Joh McCarthy ha inventato linguaggio LISP (comm
50) incorporando idee del λ-calcolo in un
linguaggio di programmazione.

FUNZIONI PURE — se risultato dipende solo
dall'input

// doll' input

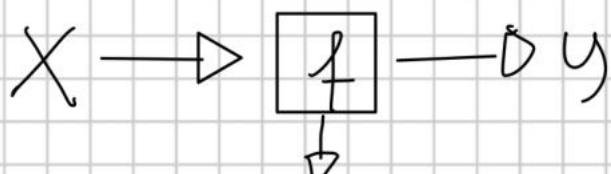
De mon fré side-effect

se non modifica l' input



funtione puro

```
int f(int x) { return x * x; }
```



side-effect

funtione non puro 2.

```
int y;
int g(int x) { return x + (y++); }
```

TRASPARENZA REFERENZIALE

Una funzione gode di trasparenza referenziale se riportando un'invocazione della funzione con il risultato della chiamata, il risultato complessivo non cambia.

$$\begin{array}{l} f(2) + 1 \rightarrow 4 + 1 \\ \parallel \\ 4 \end{array}$$

$$\begin{array}{l} g(2) + g(2) \rightarrow 2 \cancel{*} 2 \\ \parallel \\ 2 \quad 3 \end{array}$$

non ci è trasparente
referenziale!

MEMOIZZAZIONE

Poiché una funzione pura restituisce lo stesso valore a parità di valore d'input, il risultato può essere "memorizzato" cioè tenuto in un'opporta cache per evitare di ricalcolarlo.

FUNZIONI DI PRIMA CLASSE

Se una funzione può essere trattata come fosse un valore (passata a una funzione, restituita, tenuta in una variabile) allora viene chiamata funzione di prima classe.

VANTAGGI PROGRAMMAZIONE FUNZIONALE

1. MODULARITÀ: programma formato da funzioni pure che fungono da moduli.
2. COMPOSIZIONALITÀ: composizione per formare comportamenti più complessi.
3. PARALLELIZZABILITÀ: l'assenza di side-effect evita race condition (due core differenti incidono sulle stesse celle di memoria).
4. TESTABILITÀ: assente di side-effect rende più semplice testare funzioni (es. return "test" invece printf("test"))

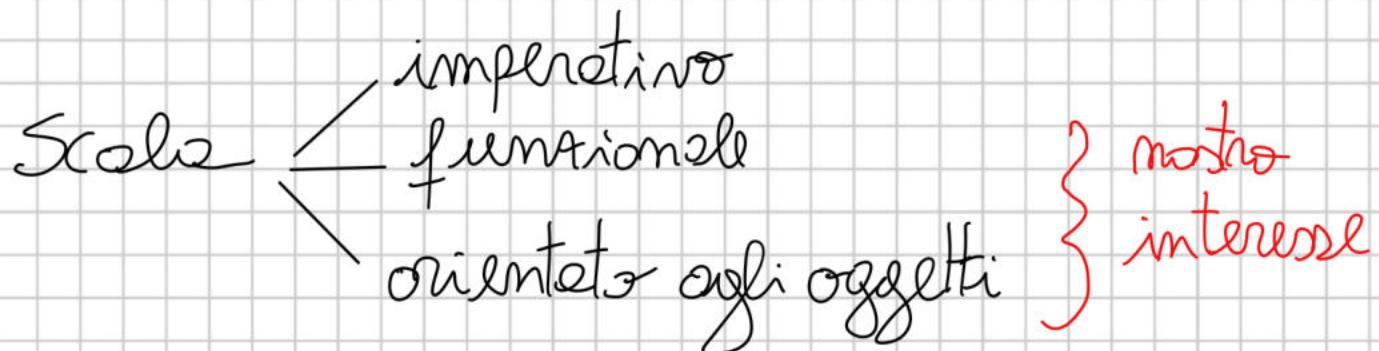
SVANTAGGI PROGRAMMAZIONE FUNZIONALE

1. I/O: I/O è interamente basato sulle mutazioni di stato, è difficile da gestire in modo funzionale.

APPROCCIO: Si isolano le parti di programma che fanno I/O e si scrive il resto in stile funzionale.

2. ASTRAZIONE MOLTO DIVERSA DALL'hardware
penalità prestazionali in alcuni casi

SCALA COME LINGUAGGIO IBRIDO MULTI-PARADIGMA





Svantaggi della programmazione funzionale

- I/O: incrementemente basato su notione di stato, è complesso da gestire in modo funzionale. Appraccio: si isolano le parti del programma che usano l'I/O e si scrive il resto in stile funzionale
- Terminologia: funzioni pure, trasparenza referenziale, ecc.
- Astrazione molto diversa dall'hardware porta penalità prestazionali.



Vantaggi: compattezza, costrutti espressivi

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

Java

```
object Hello extends App {  
    println("Hello")  
}
```

Scala

In Scala è possibile invocare qualsiasi metodo Java.

Scala gira sulla Java VM! (è compilato come Java .class)



Svantaggi DI SCALA

Sintassi spesso ostica
riduce leggibilità e
aumenta curva di apprendimento
curva

SCALA ha un interprete interattivo, riconosce i tipi delle variabili (sempre minuscole), non serve indentazione.

10 Int , 3.14 Double , 3.14f Float
'a' Char , true Boolean , "hello" String

Questi tipi primitivi possono effettuare metodi:

10.toInt , 10.toByte

Val x = 10 è immutabile (Val y: Int = 10)

Var x = 5 è mutabile

// /* ... */ sono i commenti

Val ++ = 50 nome anche simboli

10.+ (10) metodi anche simboli

"hello".+ (" world") = "hello world"

METODI

ARGOMENTI

↑

Tipo di ritorno

↑

RETURN

↑

def somma (x:Int, y:Int):Int = x+y

Per eseguire programma: ^{cognosco} **Mate** nome.sc
// ; sono optionali, è case sensitive

Per eseguire **scala** nome.sc **1.**

def prodotto (x:Int, y:Int): Int = x * y

Print (prodotto(10,20)) => 200

mate provaMain.scala **CONTIENE IL MAIN** **2.**

object ProvaMain extends App {
 def somma (x: Int, y: Int): Int = x + y
 Print(m(somma (x+y)))
}

scalac ProvaMain.scala

scala ProvaMain => 30

ricorsione.sc

ESEMPIO DI RICORSIONE

def fattoriale (n: Int): Int =
 if (n <= 1) 1
 else n * fattoriale (n - 1) **(non c'è return)**

Print(m(fattoriale(5)) => 120

* Con ricorsione è obbligatorio tipo di ritorno.

trattamento 2. sc

OPTIONALE
R

```
def sommaPrimiNNumeri(m: Int) : Int = {  
    if (m < 0) 0  
    else m + sommaPrimiNNumeri(m - 1)  
}
```

Print(m (sommaPrimiNNumeri(3))) 3+2+1 = 6

Unit è il nostro null

Altre notazioni:

if (expr) e1 else e2 SCALA



ESPRESSIONE

CONDIZIONALE

expr ? e1 : e2 C

TUPLA

(2,5) → (Int, Int) = (2,5)

Val t = (2,5)

t._1 = 2, t._2 = 5

val (a,b) = (2,5) ↗ a : Int = 2
↗ b : Int = 5

("hello", 2, 3.14) Le tuple possono avere valori diversi tra loro

COPPIA

$$2 \rightarrow 5 \Leftrightarrow (\text{Int}, \text{Int}) = (2, 5)$$

Fun1.sc

def f(x: Int, y: Int) = {

x * y

x - y

x + y → restituisce l'ultima espressione
che c'è in f

}

:load Fun1.sc

def stampa(s: String) = {

} Print(m(s)) → restituisce un tipo Unit, se
solamente side-effect

def stampa(i: Int) = {

} Print(m(i))

verifica se c'è ricorsione
di codice.

SCALA permette
overloading, stesso
nome, ma parametri
diversi.

@scala.annotation.tailrec

def factIter(f: Int, n: Int): Int =

if (n < 2) f else factIter(m * f, m - 1)

def fact(n: Int): Int = factIter(1, n)

METODI DI ORDINE SUPERIORE: Sono metodi che prendono come argomento funzioni o resti
tutte funzioni.

Esempio:

def stampa (f:Int => Int, x:Int) =
 phint(m (f(x)))

def doppio (x:Int) = 2*x

(Se ho più parametri $f:(\text{Int}, \text{Int}) \Rightarrow \text{Int}$)

N.B.: C'è una differenza tra metodi che viene definito con def) e funzione. Se si posse un metodo come argomento (di un altro metodo) di tipo funzione, il metodo viene convertito automaticamente a funzione. Lo stesso vale per l'assegnamento

def ifESE (b:Boolean, f1:Int => Int, f2:Int => Int) =
 if (b) f1 else f2

↳ tipo di ritorno una funzione

doppio - lo converte in funzione

È possibile scrivere esplicitamente un metodo
in una funzione scrivendo M —

$\text{if } \text{Else}(\text{true}, \text{doppio}, \text{triplo})(10) \rightarrow \text{Int} = 20$

FUNZIONI ANONIME

Ese.: $\text{def doppio}(x: \text{Int}) = 2 * x \leftarrow \text{metodo}$

$x \Rightarrow 2 * x \leftarrow \text{funzione anonima}$

oppure, esprimendo il tipo di x esplicitamente:

$(x: \text{Int}) \Rightarrow 2 * x$

In notazione lambda colto sarebbe
 $\lambda x. 2 * x$ (ma questo non è SCALA)

Esempi:

Vsl $f: \text{Int} \Rightarrow \text{Int} = x \Rightarrow 2 * x$

$f(10) \rightarrow 20$

$\text{if } \text{Else}(\text{true}, x \Rightarrow 2 * x, x \Rightarrow 3 * x)(10) \rightarrow 20$

FORMA ABBREVIATA per le funzioni anonime

Ese. $\boxed{- * 2} \xleftarrow{\text{equividente}} \boxed{x \Rightarrow x * 2}$

$$\text{val } g : (\text{Int}, \text{Int}) \Rightarrow \text{Int} = \underline{\quad} + \underline{\quad}$$

\uparrow
 $(x, y) \Rightarrow x + y$

Note: posso abbreviare

$$x \Rightarrow x * x ?$$

$$\underline{\quad} * \underline{\quad} ? \Rightarrow \text{no!}$$

$$\hookrightarrow (x, y) \Rightarrow x * y$$

Quindi $x \Rightarrow x * x$ non lo posso abbreviare direttamente

Nota: def myFun ($f : \text{Int} \Rightarrow \text{Int}$) = ...

$$1) \text{myFun}(2 * \underline{\quad}) \leftrightarrow \text{myFun}(x \Rightarrow 2 * x)$$

$$2) \text{myFun}(\underline{\quad}) \leftrightarrow \text{myFun}(x)$$

$$3) \text{myFun}(\text{identity}) \leftrightarrow \text{myFun}(x \Rightarrow x)$$

FUNZIONI ANNIDATE: è possibile dichiarare un metodo all'interno del corpo di un altro metodo

def fact (n: Int) = {

@scala.annotation.tailrec

def factIter (f: Int, m: Int) = ...

factIter(1, n)

Esercizio: Scrivere un metodo Componi che prende due funzioni f e g da Int a Int e calcola la composita che dato x restituisce $f(g(x))$

def

Componi ($f: \text{Int} \Rightarrow \text{Int}$, $g: \text{Int} \Rightarrow \text{Int}$): $\text{Int} \Rightarrow \text{Int} =$
 $X \Rightarrow f(g(X)) \rightarrow$ è una funzione

Componi ($2 * -$, $3 * -$) (10) $\rightarrow 60$

def Componi2 ($f: \text{Int} \Rightarrow \text{Int}$, $g: \text{Int} \Rightarrow \text{Int}$) = {

def aux ($X: \text{Int}$) = $f(g(X))$

aux -

}

CLOSURA: è una funzione che eccede le variabili dichiarate esternamente alla funzione stessa. Il valore di queste variabili al momento della creazione dell'oggetto funzione viene memorizzato nell'oggetto stesso

Vai $z = 10$

Vai $f: \text{Int} \Rightarrow \text{Int} = x \Rightarrow x + z$ // z vale 10 qui..

Println ($f(20)$) $\rightarrow 30$

$z = 100$

Println ($f(20)$) $\rightarrow 120$

Esempio CHIUSURA 2

```
def somma2 (a:Int) : Int => Int = {  
    def sommaA(b:Int) = a + b  
    sommaA _  
}
```

Val sommaThe = somma2(3)

Println(sommaThe(10)) → 13
 ↳ a + b

def somma2 (a:Int)(b:Int) = a + b

Val sommaDieci = somma2(10) — (converts in
Println(sommaDieci(10)) → 20
funzione)

CURRIFICAZIONE: normalmente una funzione richiede che tutti i parametri siano specificati. La currificazione è una tecnica in cui parametri possono essere passati uno alla volta con applicazioni successive.

Val f = (x:Int, y:Int) => x + y

↓ **CURRIFICAZIONE**

Val c = (x:Int) => (y:Int) => x + y

f(c 10, 20) → calcola 30

c((10)c20) → calcola 30

Il vantaggio della currying è che consente di applicare solo parzialmente gli argomenti:

Ese.: val somma3 = CC3

Println(somma3(10)) → 13

Val somma3 = CC3

ESERCIZIO

E1.scala

object E1 {

def applicaDueVOLTE(f: Int ⇒ Int, x: Int) = {
 f(f(x))

}}

E1Main.scala

object E1Main extends App {

val x = E1.aplicadueVolte(x ⇒ x + 1, 10)
println(x)

}

scalac E1Main.scala E1.scala

scala E1Main → 12

Esercizio

E1.scala

Versione
CORRIFICATA

object E1 {

def applicaDueVolte(f: Int => Int)(x: Int) = {
 f(f(x))

}

E1Main.scala

- + 1

object E1Main extends App {

val v = E1.appliDueVolte(x => x + 1) -
println(v(10))

}

scalac E1Main.scala E1.scala

scala E1Main -> 12

anche:

def applicaDueVolte(f: Int => Int) = {

def aux(x: Int) = f(f(x))

aux -

{

↳ lavoro per - o ✓ perché aux è una funzione

def applicaDueVolte(f: Int => Int) =

(x: Int) => f(f(x))

2> versione con
funzione omomima

LISTE (sono liste collegate) **IMMUTABILI**

`List(2, 3, 5, 7, 8) → List[Int] = List(2, 3, 5, 7, 8)`

Val `l = List(2, 3, 5, 7, 8)`

- `l.head` → `Int = 2` PRIMO ELEMENTO
- `l.tail` → `List[Int] = List(3, 5, 7, 8)` RESTO LISTA

Esempio: `def sum(l: List[Int]): Int =`

OPPURE

`if (l == List()) 0`
`else l.head + sum(l.tail)`

`if (l.length == 0)`

`if (l == Nil)`

N.B.: `==` equivale a `equals()` di Java,
è l'uguaglianza profonda.

`l == List(2, 3, 5, 7, 8) → true`

`l eq List(2, 3, 5, 7, 8) → false`, uguaglianza
superficiale, puntatori.

• `l.last` → `Int = 8` ULTIMO ELEMENTO

(Ora complessità O(n))

• $\begin{matrix} 1 & \dots & l \\ \downarrow & & \uparrow \\ \text{SCALARE} & \text{LISTA} \end{matrix} \Rightarrow List(1, 2, 3, 5, 7, 8)$

Es.: `def` `taddoppiā (l: List[Int]): List[Int] = {
 if (l == Nil) Nil
 } else (l.head * 2) :: taddoppiā (l.tail)`

`Val l = List(1, 2, 3, 4, 5)`

`taddoppiā (l) → List(2, 4, 6, 8, 10)`

Es.: `def` `mappā (f: Int => Int, l: List[Int]): List[Int] = {
 if (l == Nil) Nil
 } else f(l.head) :: mappā (f, l.tail)`

`Val l = List(2, 4, 6, 3, 8)` divisione intera

`mappā (x => x / 2, l) → List(1, 2, 3, 1, 4)`

• `Val t = ("uno", "due", "tre") // TUPLA`

`t._1 → "uno"`

`t._2 → "Due"`

`t._3 → "tre"`

↗ COPPIA DI LISTE → fa `SPLIT` all'indice 3

• `l. splitAt (3) → (List(2, 4, 6), List(3, 8))`

• `l. take (3) → List(2, 4, 6) // Primi 3 elementi`

• `l. drop (3) → List(3, 8) // toglie i primi 3 elementi`

• `l. size o l.length → Int = 5 // dimensione`

• List. range(1, 10) → List(1, 2, 3, 4, 5, 6, 7, 8, 9)

↳ costruisce una lista che va da start a end-1

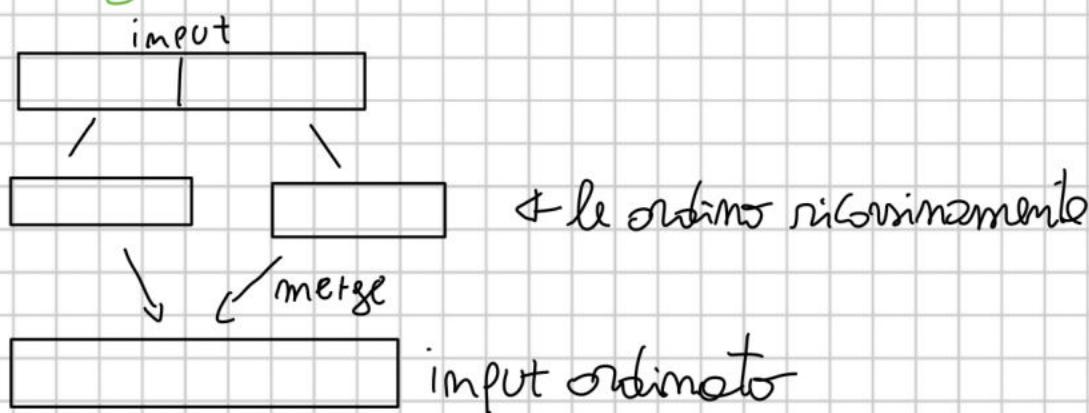
• l. sum → Int = 23 // fa la somma

• l. zipWithIndex

↳ List [(Int, Int)] = List((2, 0), (4, 1), (6, 2), (3, 3))

// crea una nuova lista di coppie con i valori precedenti di l e il loro indice.

mergeSort.sc



```
def merge (a: List[Int], b: List[Int]): List[Int] = {
    if (a == Nil) b
    else if (b == Nil) a
    else if (a.head < b.head) a.head :: merge(a.tail, b)
    else b.head :: merge(a, b.tail)}
```

```

def mergeSort (l: list[Int]): List[Int] = {
  if (l == Nil) Nil
  else {
    val (a, b) = l.split(l.length / 2)
    val sx = mergeSort(a)
    val dx = mergeSort(b)
    merge(sx, dx)
  }
}

```

Altri metodi:

- $l.isEmpty$ // mi dice se è vuota
- $l.init$ // mi ride la lista tranne l'ultimo elemento $(1, 2, 3) \rightarrow (1, 2)$
- $List(5, 1) \dots List(2, 5, 4) \rightarrow List(5, 1, 2, 5, 4)$
// concatenare le liste

val $v = 0 :: \overbrace{List(1, 2, 3)}^{\text{ha costo } O(1)}$

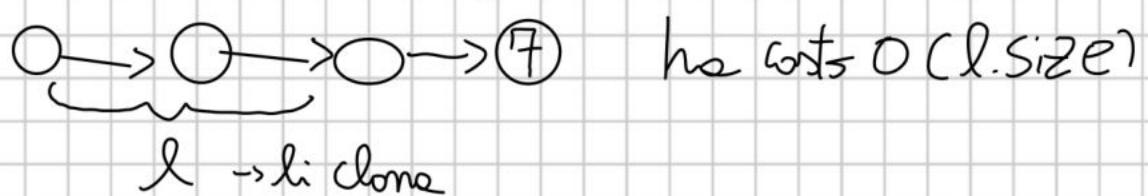
```

graph LR
    0((0)) --> 1((1))
    1 --> 2((2))
    2 --> 3((3))
    3 --> 4((4))

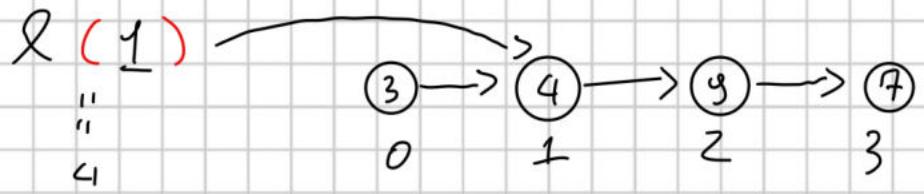
```

val $t = v :: l$ ci mette $O(v.size)$

• val $t = l :+ 7$ // aggiunge in coda



- Val $\ell = \text{List}(3, 4, 9, 7)$



Il `l(i)` mi dà il valore della posizione *i*-esima

- . l reverse → List (9, 7, 4, 3)

↳ reversib. la liste, la const. O(n)

- l. max → g // massimo

- l. Min \rightarrow 3 // Minimo

- `l. slice(a, b)` → mi dà i valori tra i due indici della lista (a incluso b escluso)

↳ l. slice(1, 3) → List(4, 9)

ha costo $O(n)$ Perche' deve conoscere tutti gli elementi

- Val t = List(2, 3, 5) → coppie

- l. $\text{Zip}(t) \rightarrow \text{List}[(\text{Int}, \text{Int})] = \text{List}((3, 2), (4, 3), (9, 5))$

// dà una lista di coppie con i valori negli stessi indici
di let (la lunghezza è il min tra let)

l. `sliding(2,2).toList`

$l = List(1, 3, 4, 9, 7)$

$\hookrightarrow List[List[Int]] = List(List(1, 3), List(4, 9), List(7))$

// dà una lista di liste in cui il primo parametro
mi dice la lunghezza delle sottoliste, e il secondo
mi dice di quanto avanzare tra una lista e l'altra

`l.sliding(5,1) -> List(List(1,3,4,9,7))`

`l.sliding(4,1) -> List(List(1,3,4,9), List(3,4,9,7))`

`l.sliding(4,2) -> List(List(1,3,4,9), List(4,9,7))`

Esercizi

```
def mySize(l: List[Int]): Int = {  
    if (l.isEmpty) 0 // oppure l == Nil / l == List()  
    else 1 + mySize(l.tail) ↳ O(m)  
}
```

$\hookrightarrow l.\text{init} \rightarrow O(m^2)$

// versione generica

```
def mySize[T](l: List[T]): Int = {  
    if (l.isEmpty) 0  
    else 1 + mySize(l.tail)}
```

```
def myRev[T](l: List[T]): List[T] = {  
    if (l.isEmpty) Nil  
    else myRev(l.tail) :+ l.head}
```

$\hookrightarrow O(m^2)$

$\hookrightarrow O(n)$ per n volte
ricorsive

// Versione di myRev() più efficiente

```
def myRev2 [T](l: list[T]) = {
    def aux [T](t: list[T], t: List[T]): List[T] = {
        if (t.isEmpty) t
        else aux(t.tail, t.head :: t)
    }
    aux(l, Nil)
}
```

↳ RICORSIONE
DI CODA
@scala.annotation.tailrec

↳ ha costo $O(n)$ perché $t.head$ ha costo $O(1)$
per n chiamate ricorsive, abbiamo sfruttato la
ricorsione di coda.

```
def inters [T] (a: List[T], b: List[T]): List[T] = {
    if (a == Nil || b == Nil) Nil
    if else (b.contains(a.head)) a.head :: inters(a.tail, b)
    else inters(a.tail, b)
}
```

↳ costo $O(b.size^2)$ perché viene fatta b volte
la ricorsione e ci mette $O(b)$ la contains nel
caso peggiore

· $l.slice(a, b)$ mi dà gli elementi a compres
l b esclusi

$List(1, 2, 3, 4).slice(1, 3) \rightarrow List(2, 3)$

METODI ORDINE SUPERIORE SU LISTE

l. count ($X \Rightarrow X > 4$) // $\rightarrow 4$

↳ mi dice quanti elementi soddisfano quella condizione (Int)

l. exists ($_ < 4$)

↳ mi dice se esiste (Boolean) un valore che soddisfa quella condizione

l. filter ($_ == 4$)

↳ mi dà una lista degli elementi che soddisfano quella condizione [List[T]]

l. forall ($_ < 4$)

↳ verifica se tutti quanti gli elementi verificano il predicato, exists solamente uno (Boolean)

l. foreach ($X \Rightarrow \text{println}(X)$) // NON SERVE

↳ applica la funzione ad ogni elemento

$l = \text{List}(1, 3, 4, 9, 7) \rightarrow 1\ 3\ 4\ 9\ 7$

// anche l foreach $X \Rightarrow \text{println}(X)$

• l. map (- * 2)

↳ dà una nuova lista dove ha applicato la funzione ad ogni elemento

$$\text{List}(1, 3, 9, 3, 7) \rightarrow \text{List}(2, 6, 18, 18, 14)$$

Esercizi: // lista con posizione elementi pari

```
def pospari [T] (l: List[T]): List[T] = {  
    l.zipWithIndex.filter(x => x._2 % 2 == 0)  
    .map(_._1)  
}
```

Usa zipWithIndex per ottenere lista di coppie indice, valore e cui applico filter che mi dà le coppie con il secondo elemento pari, mentre il map mi dà una lista solo con i primi elementi delle coppie pari, gli indici,

• Val l = List(1, 1, 2, 3, 2, 5)

l. distinct → List(1, 2, 3, 5) // l.toSet.toList

(→ rimuove duplicati, ha costo O(n))

l. sorted

↳ nuova lista ordinata

```
def palindrome[T](l: List[T]): Boolean = {  
    l.reverse == l  
}
```

$l.\text{flatten}$ contengono tutti gli elementi delle liste di liste
 $\Rightarrow \text{List}(\text{List}(1), \text{List}(2), \text{List}(3)) \rightarrow \text{List}(1, 2, 3)$

COSTI

$\text{splitAt}(K) \mid O(K)$

$\text{take}(K) \mid O(K)$

$\text{List.range}(a, b) \mid O(b - a)$

$\text{reverse} \mid O(n)$

$\text{sum}, \text{min}, \text{max} \mid O(n)$

$\text{zip}(a, b) \mid O(\min(a, b))$

$\text{sliding} \mid O(n)$

$\text{slice}(a, b) \mid O(b)$

$\text{contains} \mid O(n)$

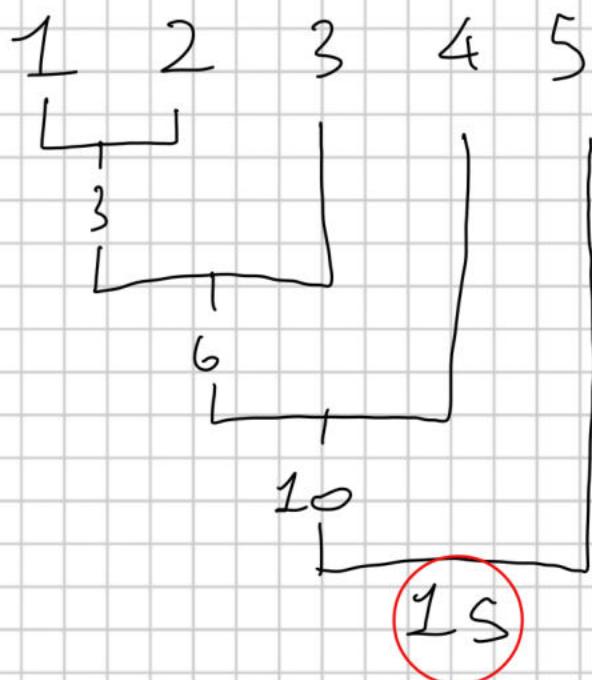
- $l.\text{partition} (_\% 2 == 0)$

crea due liste, quelle con i valori che soddisfano la condizione e quelli no

METODO REDUCE

Ved | $l = \text{List}(1, 2, 3, 4, 5)$

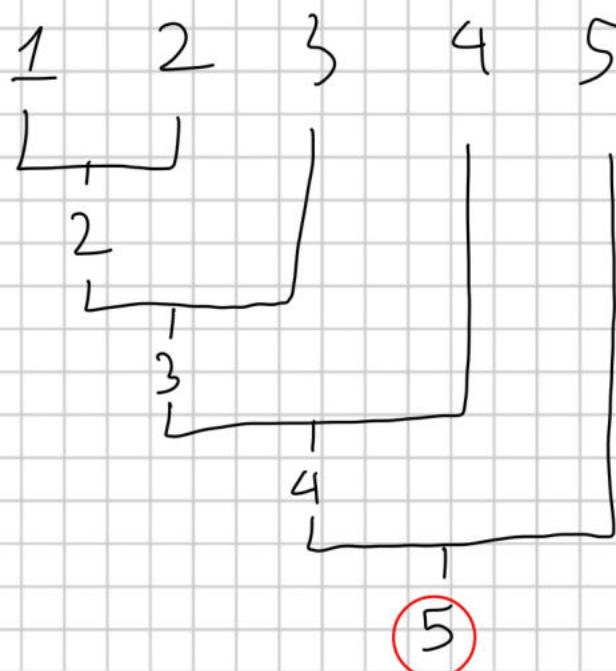
$l.\text{reduce}(-+ -)$



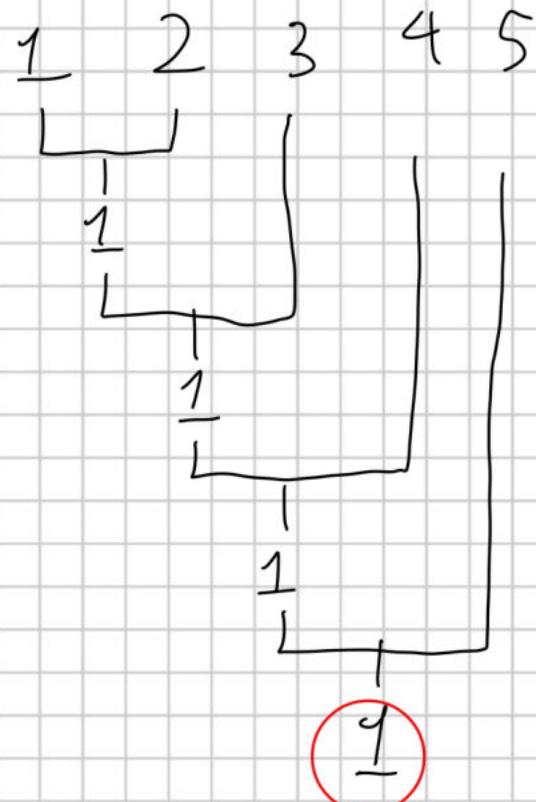
○ RISULTATO

REDUCE

$l.\text{reduce}(-\max -)$



$l.\text{reduce}(-\min -)$



Alternativa: $\text{min} \equiv \text{reduce}((x,y) \Rightarrow x \min y)$

l. $\text{reduce}((x,y) \Rightarrow x \min y)$

VARIANTE: $\text{foldLeft}(z)(f)$

$\begin{matrix} \uparrow & \uparrow \\ \text{valore} & \text{funzione} \\ \text{iniziale} & \text{a due argomenti} \\ & (\text{come reduce}) \end{matrix}$

E.s.:

l. $\text{foldLeft}(0)(_ + _)$

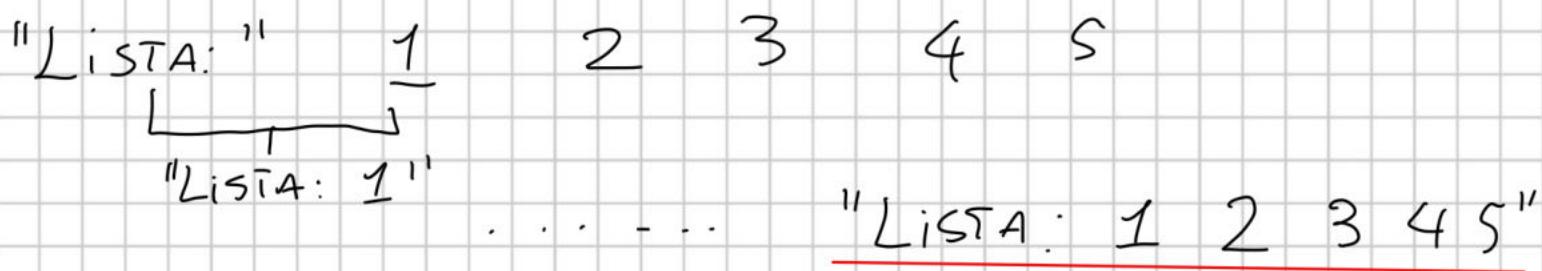


Usa valore iniziale
e come partenza

N.B.: z può avere un tipo diverso da quelli degli argomenti di f

E.s.:

l. $\text{foldLeft}("Lista: ")((s,x) \Rightarrow s + " " + x)$



`takeWhile` estende il più lungo prefisso che soddisfa una certa proprietà

Ese:

val p = (2, 4, 0, 1, 2, 3)

p. `takeWhile(_ % 2 == 0)` $\rightarrow (2, 4, 0)$

`dropWhile` come `takeWhile` le elimina invece di prenderle

p. `dropWhile(_ % 2 == 0)` $\rightarrow (1, 2, 3)$

`groupBy` partitiona gli elementi come specificato da una data funzione

Ese:

val l = List(1, 2, 3, 4, 5)

l. `groupBy(_ % 2 == 0).toList`

\hookrightarrow List((false, List(1, 3, 5)), (true, List(2, 4)))

val q = ("uno", "quattro", "tre")

q. `groupBy(_.length).toList` \rightarrow PARTIZIONA
PER VALORI DIVERSI

List((7, List(quattro)), (3, List(uno, tre)))

• $l.\text{groupBy}(\text{identity}).\text{toList}$

↳ $\text{List}((5,\text{List}(5)),(1,\text{List}(1)),(2,\text{List}(2)),(3,\text{List}(3)),(4,\text{List}(4)))$

• $\text{List}(2,2,2).\text{groupBy}(\text{identity}).\text{toList}$

↳ $\text{List}((2,\text{List}(2,2,2)))$

• $\text{List}("proua","pame","acqua","ammo")$

↳ $_.\text{groupBy}(_.\text{charAt}(0)).\text{toList}$

↳ $\text{List}('a',\text{List}("acqua","ammo")),('p',\text{List}("pame","proua"))$

Exerciciò:

// Selecció elements distints. Usar `groupBy`

```
def myDistinct(l: List[Int]) = {  
    val p = l.groupBy(identity).toList  
    p.map(coppia => coppia._1  
    )
```

val t = List(1, 2, 2, 1, 3, 1)

myDistinct(t)

↳ $\text{List}(1,2,3)$

// variante

```
def myDistinct (l : List[Int]) =  
  l.groupBy (identity).toList.map(_._1)
```

• maxBy / minBy calcolano il

massimo / minimo di una lista rispettivamente a una certa funzione degli elementi.

```
val q = List ("uno", "quattro")
```

• maxBy(_.length)

→ restituisce "quattro"

• minBy (_ . length)

→ restituisce "uno"

Metodi su tipi generici

```
def mySize(l: List[Int]): Int =  
  if (l.isEmpty) 0 else 1 + mySize(l.tail)
```

↳ questa solo su liste d'interi.

```
def mySize[T](l: List[T]): Int =  
  if (l.isEmpty) 0 else 1 + mySize(l.tail)
```

↳ eccette valori generici T

Esempi:

// verifica se una lista d'interi è ordinata

@scala.annotation.tailrec

```
def isSorted(l: List[Int]): Boolean = {  
  if (l.size < 2) true  
  else if (l.head > l.tail.head) false  
  else isSorted(l.tail)  
}
```

↳ RICORSIONE DI CODA

// mostra revisione ricorsiva del metodo map

```
def myMap[A, B](l: List[A], f: A => B): List[B] = {  
  if (l.isEmpty) Nil  
  else f(l.head) :: myMap(l.tail, f)  
}
```

myMap (l, $\underline{x: \text{Int}} \Rightarrow 2 * x$)

↳ va specificato

// Verifica se una lista a è una sottolista di un'altra lista b

// Ese.: a = List(2,3) e b = List(1,2,3,4)

```
def sottolista[T](a:List[T], b:List[T]): Boolean = {
    if (a.size > b.size) false
    else b.slice(0, a.size) == a || sottolista(a, b.tail)
}
```

↳ RICORSIONE

a = "burg", b = "ham burger"

ham|burger
burg

|amb|urger
burg

|m|burger
burg

|burg|er
burg

|burg|e|
burg

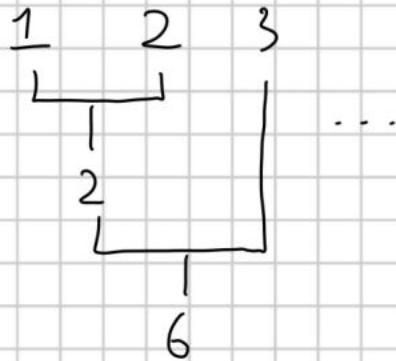
TRUE

Usiamo metodo slice

// Calcolo fattoriale usando i metodi delle liste

```
def myFact (n: Int) : Int = {  
    List.range(1, n+1). reduce(_*_)}
```

myFact(3)



N.B.: (1.to(3)).toList

↳ Come range => List(1, 2, 3) (ma 3 compreso)

(1.until(3)).toList

↳ List(1, 2) (esclude ultimo elemento)

// Somma primi n numeri

```
def somma(n: Int): Int =  
(1.to(n)).reduce(_+_)
```

Esercizio

// Date liste elementi, trova gli indici di uno specifico elemento
// Es.: List(1, 2, 1, 1, 3) e X = 1 => List(0, 2, 3)

```
def trovaIndici[T](l: List[T], x: T): List[Int] = {  
    l.zipWithIndex.filter(_._1 == x).map(coppia => coppia._2)  
}  
} ↓  
list di coppie  
valori / indice  
↓  
coppia => coppia._1 == x  
→ filtro le coppie  
che contengono x  
↑  
coppia => coppia._2  
→ prende  
gli  
indici
```

Usa solo metodi delle liste, senza ricorsione.

ALGORITMI ORDINAMENTO IN SCALA

MERGE SORT

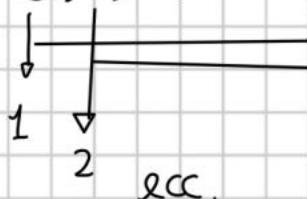
Se lista vuota o l'ore Nil

altrimenti:

- ordina ricorsivamente prima metà della lista
- ordina ricorsivamente seconda metà della lista
- fonde le due metà ordinate

ALGORITMO DI FUSIONE merge

Es. (1, 3, 5, 6) | (2, 4)



```

def merge(a: List[Int], b: List[Int]): List[Int] = {
    if (a.isEmpty) b
    else if (b.isEmpty) a
    else if (a.head < b.head) a.head :: merge(a.tail, b)
    else b.head :: merge(a, b.tail)
}

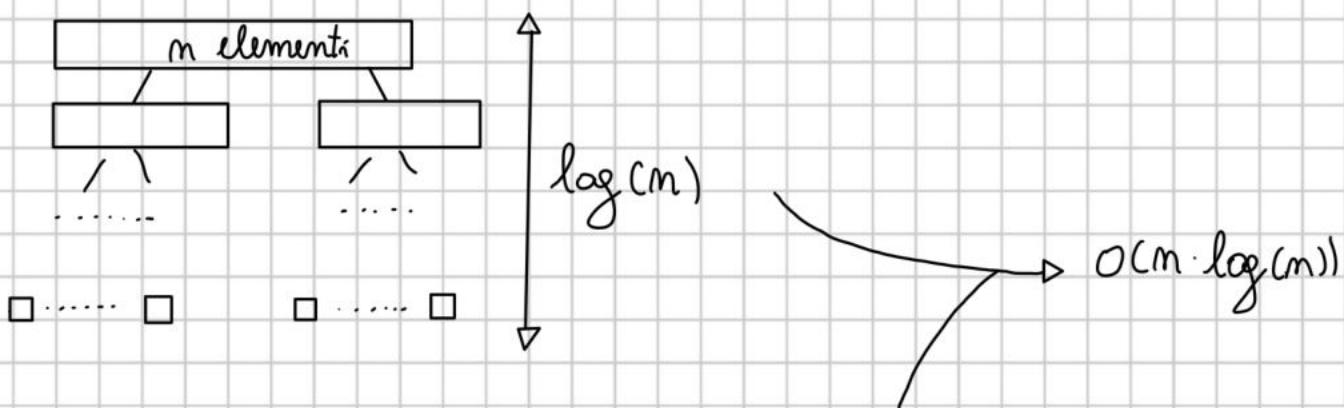
```

Hip: Le due sottoliste a e b sono ordinate

```

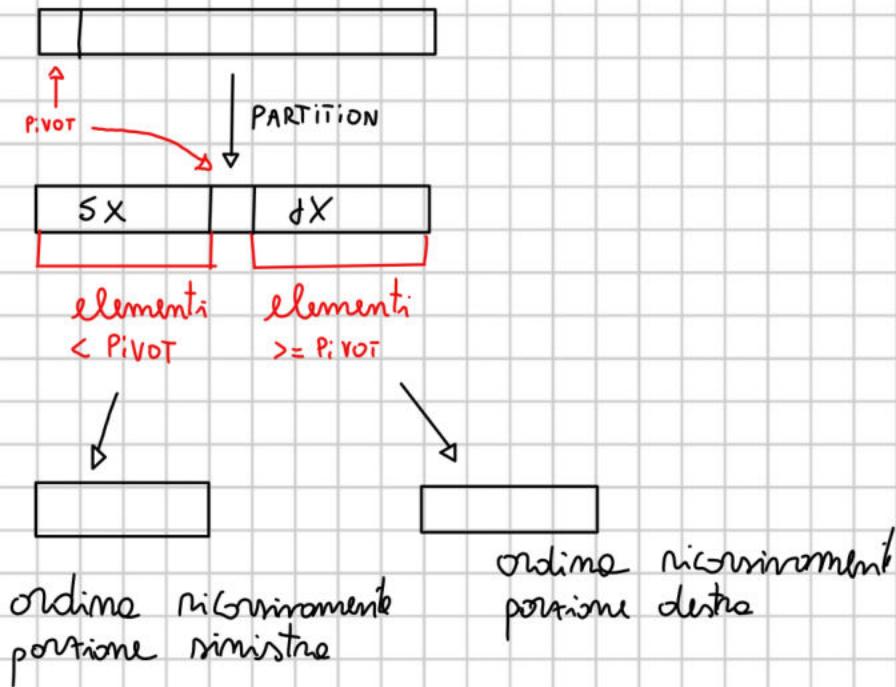
def mergeSort(l: List[Int]): List[Int] = {
    if (l.size < 2) l
    else {
        val (l1, l2) = l.splitAt(l.size / 2)
        val a = mergeSort(l1)
        val b = mergeSort(l2)
        merge(a, b)
    }
}

```



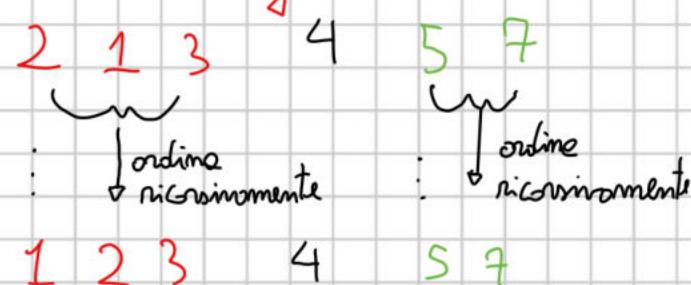
Su ciascun livello la merge impiega $O(m)$

QUICKSORT



```
def quickSort (l : List[Int]) : List[Int] = {
    if (l.isEmpty) Nil
    else {
        val sx = l.filter (x => x < l.head)
        val dx = l.filter (x => x >= l.head).tail
        quickSort(sx) :: (l.head :: quickSort(dx))
    }
}
```

ES.: 4 2 1 5 3 7



$O(n^2)$ nel caso peggiore, come per esempio una lista di elementi tutti uguali (QUICKSORT)

// QUICKSORT CON COMPARATORE

```
def quickSort[T](cmp: (T, T) => Int)(l: List[T]): List[T] =  
{  
    if (l.isEmpty) l  
    else {  
        val sx = l.filter(x => cmp(x, l.head) < 0)  
        val dx = l.filter(x => cmp(x, l.head) >= 0).tail  
        pivot  
        } }  
        quickSort(cmp)(sx) :::(l.head :: quickSort(cmp)(dx))
```

val f = quickSort((x: Int, y: Int) => x - y) —
f(List(4, 2, 5, 1))
↳ 1, 2, 4, 5
COMPARATORE

COMPARATORE: cmp : (T, T) => Int

cmp(a, b)

< 0	Δe	$a < b$
$= 0$	Δe	$a == b$
> 0	Δe	$a > b$

y.length

Es.: (x: String, y: String) => if (x.length < y.length) -1
else if (x.length == y.length) 0 else 1

// variazione della funzione flatten definita usando i metodi sulle liste

```
def myFlatten [T](l: List[List[T]]): List[T] = {  
    l.reduce(_ :: _)  
}
```

INSERTIONSORT

// Algoritmo di ordinamento per inserimento, mi serve un metodo che dato un elemento x e una lista l ordinate, inserisce x in l correttamente.

```
def insert(x: Int, l: List[Int]): List[Int] = {  
    if (l.isEmpty) List(x)  
    else if (x < l.head) x :: l  
    else l.head :: insert(x, l.tail)  
}
```

```
def insertionSort(l: List[Int]): List[Int] = {  
    if (l.size < 2) l  
    else insert(l.head, insertionSort(l.tail))  
}
```

↓

ha costo $O(n)$, farà $l.tail$
dove ricorrere tutto
nel caso peggiore

=> costo $O(n^2)$

// calcolo del prodotto simbolico $\{1, n\} \times \{1, m\}$

// $\text{prod}(n) = \text{List}((1, 1), \dots, (1, m), (2, 1), \dots, (2, m), \dots, (n, m))$

def prod(m: Int) : List[(Int, Int)] = {
 (1 to m).map(x => (1 to m).map(y => (x, y)))
 .toList, flatten
}

// usando for comprehension (genero collezioni)

def prod2(m: Int) = {
 val l = for {
 x <- 1 to m
 y <- 1 to m
 } yield (x, y) yield tira fuori l'elemento
 l.toList
}

// variante che genera triple

def prod3(m: Int) = {

val l = for {

x <- 1 to m

y <- 1 to m

z <- 1 to m

} yield (x, y, z)

l.toList
}

// variante prod2 con passo 2 sugli indici

def prod2_passo2 (n:Int) = {

val l = for {

x < 1 to n

y < 1 to n

} yield (2*x, 2*y)

} l.toList

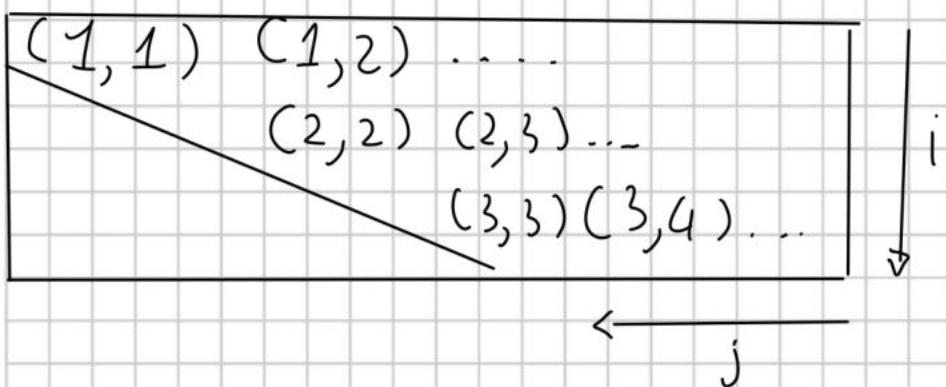
// prod3 usando map e flatten

def prod3_variante(n:Int) = {

(1 to n).map(x => (1 to n).map(y => (1 to n).map(z => (x, y, z)))) .flatten .flatten .toList

}

// indice coppie matrice triangolare



proprietà è che $j \geq i$

```
def matriceTriangolare(m: Int) = {
    val l = for {
        i ← 1 to m
        j ← 1 to m
    } yield (i, j)
    l.toList
}
```

// variante

```
def matriceTriangolare(m: Int) = {
    val l = for {
        i ← 1 to m
        j ← 1 to m
        if (j >= i) FILTRO
    } yield (i, j)
    l.toList
}
```

// scrivere dato una lista l di elementi T e un predicato che dato un elemento di tipo T restituisce V/F, filtre tutti gli elementi che lo soddisfano

```
def myFilter(l: List[T], p: T => Boolean): List[T] = {
    if (l.isEmpty) l
    else if (p(l.head)) l.head :: myFilter(l.tail, p)
    else myFilter(l.tail, p)
}
```

// versione ricorsiva del metodo forall

```
def myForall [T] (l: List[T], p: T => Boolean): Boolean = {  
    if (l.isEmpty) true  
    else p(l.head) && myForall (l.tail, p)  
}
```

// versione ricorsiva di exists

```
def myExists [T] (l: List[T], p: T => Boolean): Boolean = {  
    if (l.isEmpty) false  
    else p(l.head) || myForall (l.tail, p)  
}
```

PATTERN MATCHING SU COSTANTI

```
def f(m: Int) = {  
    m match { // come SWITCH  
        case 0 => "Zero"  
        case 1 => "Uno"  
        case 2 => "Due"  
        case _ => "Non so"  
    }  
}
```

f(0) → "Zero" ...

Esempio:

```
def f(s: String) = {
  s.toLowerCase() match {
    case "zero" => 0
    case "uno" => 1
    case "due" => 2
    case _ => ...
  }
}
```

→ in Java e C
non posso fare
switch su
stringhe

MATCH MULTIPLO SU COSTANTI

```
def verificaPariDispari(m: Int) = {
  m % 10 match {
    case 0 | 2 | 4 | 6 | 8 => "" + m + " è pari"
    case _ => "" + m + " è dispari"
  }
}
```

MATCHING SU VARIABILI

```
def f(x: Any) = {
  x match {
    case i: Int => "" + i + " è intero"
    case f: Float => "" + f + " è float"
    case d: Double => "" + d + " è double"
    case s: String => ...
    case _ => "non so"
  }
}
```

Any è il tipo generico dei tipi, è
una superclasse, qualsiasi tipo.

```

def f(x: Any) = x match {
    case _ : Int => "Intero"
    case _ : Float => ...
    case _ : String => ... // Se non ci interessa il valore
    case _ => "ignoto"
}

```

MATCHING CON CONDIZIONI

```

def f(m: Int) = {
    m match { → FILTRO
        case x if (x < 0) => "negativo"
        case _ => "non negativo"
    }
}

```

// oppure

```

def f(m: Int) = m match {
    case _ if (m < 0) => "negativo"
    case _ => "non negativo"
}

```

MATCHING SU LISTE

```

def mySize[T](l: List[T]): Int =
    l match {
        case Nil => 0 // h vale l.head e t vale l.tail
        case h :: t => 1 + mySize(t) // ricorso
    }

```

MATCHING SU COPPIE

```
def merge2 (l1: List[Int], l2: List[Int]): List[Int] =  
  (l1, l2) match {  
    case (Nil, Nil) => Nil  
    case (l, Nil) => l  
    case (Nil, l) => l  
    case (h1 :: t1, h2 :: t2) if (h1 < h2) => h1 :: merge(t1, l2)  
    case (h1 :: t1, h2 :: t2) => h2 :: merge2 (l1, t2)  
  }
```

// altro esempio, è ordinato?

```
def ordinata (l: List[Int]): Boolean = l match {  
  case Nil => true  
  case h :: Nil => true  
  case h :: t if (h <= t.head) => ordinata (t)  
  case _ => false  
}
```

// myMap con Matching

```
def myMap [A, B] (l: List[A], f: A => B): List[B] = l match {  
  case Nil => Nil  
  case h :: t => f(h) :: myMap (t, f)  
}
```

CLASSI

JAVA

```
class Punto {  
    private double x, y;  
    public double dist;  
    public Punto(double x, double y) {  
        this.x = x;  
        this.y = y;  
        this.dist = Math.sqrt(x*x + y*y);  
    }  
    public double getX() { return x; }  
    public double getY() { return y; }  
}
```

In SCALA:

```
class Punto(x: Double, y: Double) {  
    def getX = x  
    def getY = y  
    val dist = Math.sqrt(x*x + y*y)  
}
```

→ PRIVATE

→ è già il costruttore
primario.

val p = new Punto(10, 20) // crea una classe

p.getX => 10.0

p.getY => 20.0

p.dist => 22.36 // è un val

val q = new Punto(10, 20)

p == q => false Non abbiamo definito equals!

Ereditarietà:

- Nelle classi Scala **this** denota come in Java un riferimento all'oggetto su cui si sta lavorando (costruttore, metodi)
- Tutti i metodi nel costruttore primario direttamente metodi della classe
- Il riferimento **null** denota mancante di oggetto come Java.

```
class Punto3D (x: Double, y: Double, z: Double) extends Punto(x,y){  
    def getZ = z  
    override def toString = "(" + x + ", " + y + ", " + z + ")"  
}
```

// ereditarietà come in JAVA.

```
val p = new Punto3D(10, 20, 30)  
p.getX = 10.0 // metodo superclasse  
p.getY = 20.0  
p.getZ = 30.0
```

val q: Punto = p, q punta allo stesso indirizzo
q.getZ => ERRORE

```
class Punto(x: Double, y: Double) {  
    def getX = x  
    def getY = y  
    val dist = Math.sqrt(x*x+y*y)  
    override def toString = "(" + x + ", " + y + ")"  
}
```

↳ ridefinito il toString

Se non lo definisco toString in Punto3D
ma quello della classe extends, lo eredita,
va sovrascritto.

override val dist = Math.sqrt (x*x + y*y + z*z)

↳ override di una variabile o metodo.

val p: Punto = q (Punto 3D)

println(p) => (10, 20, 30)

↳ ma **late binding** → vince il tipo effettivo
della classe d'appartenenza

Esempio:

```
class A(x: Int) {  
    override def toString = "X=" + x  
}
```

```
class B(x: Int, y: Int) extends A(x) {  
    override def toString = super.toString + ", " + "y=" + y  
}
```

richiama quello di A

val a = new A(10)

↳ x = 10

val b = new B(10, 20) => x = 10, y = 20

`val c: A = b` \Rightarrow $x=10, y=20$

late binding

`val d: B = e` \Rightarrow Type Mismatch, non posso
fare un oggetto di una super
classe o uno di una sotto
classe.

Se metto class A(val x: Int) {

}

`2. x = 10 !`

\hookrightarrow crea un getter
automatico,

\nearrow } def getX = x

class B (x: Int, val y: Int) ...

`def print(x: A) = {`

`x match {`

`case e: A => printUnit // ritorna Unit`

`case b: B => printUnit(b)`

{ }

\hookrightarrow fa un CAST!

`print(b) => X=10, y=20 : A`

`print(a) => X=10 : A`

matcha sempre il primo ! (late binding)

```
def print(x:A) = {  
    x match {  
        case b:B => println("B:" + b)  
        case a:A => println("A:" + a)  
    }  
}  
↳ fa un CAST!
```

matcha correttamente ora, vorrà messi i casi più specifici delle sottoclassi per prima!

Dentro CASE - da un'eccezione , sempre meglio metterlo.

case class P(x: Int, y: Int)

val p = P(10, 20) // possa ommettere new

val q = P(10, 20)

p == q => true // mi fa trascurare profonde,
mi redifinisce equals.

p.x => 10

p.hashCode => ... // già deto

p.toString() (10, 20)

class Q(x: Int, y: Int)

val q = Q(10, 20)

q.x errore! (non ha getters)

case class mi fornisce getters automatici,
equals, hashCode, toString.

```
def test(x: P) = p match {
    } case P(x, y) => println ("x = " + x + ", y = " + y)
```

Val p = P(10, 20)

test(p) => x=10, y=20

Vengono recuperate automaticamente x e y gli attributi della classe

case class Studente (nome: String, eta: Int)

// usiamo case su Studente;

// - getters per nome, eta

// - metodo toString

// - metodo hashCode

// - metodo equals (==, !=)

// - non serve mettere

(private nome → lo metto private)

Val l: List[Studente] = List(Studente(...), ...)

l.filter(_.eta <= 20) => ... O(n)

// Poco codice rispetto a Java

l.groupBy(_.eta).toList)

List((eta, List(Studente(...))), ...))

OVERLOADING

→ COSTRUTTORE PRIMARIO

```
class Intero(val x: Int) {  
    def this(s: String) =  
        this(Integer.parseInt(s))  
}
```

Ho due costruttori uno che prende Int e
uno string (OVERLOADING)

Se avessi usato CASE CLASS

Bisogna chiamare "Object" in SCALA: permette di definire uno spazio
di nomi associato tipicamente a una classe che consente di definire
quegli che in Java sarebbero identificati con "static"

```
class Punto(val x: Double, val y: Double)
```

// Companion object della classe Punto: ha lo stesso nome
delle classe punto

```
object Punto {
```

```
    val PI = 3.14 // come se fosse static  
}
```

```
object Main {  
    def main(s: Array[String]) = {  
        println("Hello World")  
    }  
}
```

Puoi interagire solo un object, mentre le classi hanno più istanze.

METODO apply: Serve per definire metodi che possono essere invocati direttamente su un'istanza di oggetto o su un nome di object.

Esempio: List(1, 2)

metodo apply
applicato al nome
di un object

```
class Prova(l: List[Int]) {  
    def apply(i: Int) = l(i)  
}
```

// il metodo apply definito consente di applicare gli argomenti passati direttamente sul riferimento a un oggetto Prova

List(1, 2) → equivalente a List.apply(1, 2)

val p = new Prova(List(1, 2))

p(0) = 1

p(1) = 2

p(2) → ERRORE

```
case class Intero(i: Int) {  
    def this(s: String) = this(Integer.parseInt(s))  
}
```

```
object Intero {  
    def apply(s: String) = new Intero(s)  
}
```

// Companion object

Intero("10") → 10

```
case class Intero2(i: Int)
```

```
object Intero2 {  
    def apply(s: String) = new Intero2(Integer.parseInt(s))  
    def apply(i: Int) = new Intero2(i)  
}
```

sealed abstract class Tree

case class E() extends Tree

case class T(l: Tree, e: Int, r: Tree) extends Tree

// abstract non ho instance solo sottoclassi

// sealed che li sottoclassi solo in questo modulo

val e = E()

val t = T(E(), 10, E())

sealed abstract class Tree

// metodi che conto il numero di nodi di un albero

def size(): Int = this match {

case E() => 0

case T(l, e, r) => 1 + l.size + r.size

}

case class E() extends Tree

case class T(l: Tree, e: Int, r: Tree) extends Tree

def size(t: Tree): Int = t match {

case E() => 0

case T(l: Tree, e: Int, r: Tree) => 1 + size(l) + size(r)

}

val e = E() size(e) => 0

val t = T(T(E(), 1, E()), 2, E()) size(t) => 2

val t = T(E(), 10, E()) size(t) => 1

```

def equal(t1: Tree, t2: Tree): Boolean = {
  (t1, t2) match {
    case (EC(), EC()) => true
    case (_, EC()) => false
    case (EC(), _) => false
    case (T(l1, e1, r1), T(l2, e2, r2)) => {
      e1 == e2 && equal(l1, l2) && equal(r1, r2)
    }
  }
}

```

$\text{equal}(EC(), EC()) \Rightarrow \text{true}$

$\text{equal}(E, E) \Rightarrow \text{true}$

Memoization
to choose

sealed abstract class Tree

```

def size: Int = this match {
  case E() => 0
  case T(l: Tree, e: Int, r: Tree) =>
    1 + l.size + r.size
}

```

case class E() extends Tree

case class T(l: Tree, e: Int, r: Tree) extends Tree

object Tree extends App {

val e = E()

val t = T(EC(), 10, EC())

print(t.size) // prints 1

}

Sealed abstract class Tree

def size: Int = this match {

case E () => 0

case T (l: Tree, e: Int, r: Tree) =>

} 1 + l.size + r.size

def depth: Int = this match {

case E () => 0

case T (l: Tree, e: Int, r: Tree) =>

} 1 + (l.depth \max r.depth)

} \hookrightarrow calcola la profondità

sealed abstract class Exp

case class Const(i: Int) extends Exp

case class Add(e1: Exp, e2: Exp) extends Exp

case class Mul(e1: Exp, e2: Exp) extends Exp

object Exp extends App {

val c = Const(10)

val a = Add(Const(10), Const(20))

val m = Mul(Const(10), Const(20))

print(c)

print(a)

} print(m)

\Rightarrow Stampa il toString

// Continues EXP

Sealed abstract class Exp {

```
def eval: Int = this match {
    case Const(i) => i
    case Add(e1: Exp, e2: Exp) => e1.eval + e2.eval
    case Mul(e1: Exp, e2: Exp) => e1.eval * e2.eval
}
```

case class X() extends Exp

Sealed abstract class Exp {

```
def eval(x: Int): Int = this match {
    case X() => x
    case Const(i) => i
    case Add(e1: Exp, e2: Exp) => e1.eval(x) + e2.eval(x)
    case Mul(e1: Exp, e2: Exp) => e1.eval(x) * e2.eval(x)
}
```

val e = Add(X(), Const(1))

print(e.eval(10)) => 11

// Derivative functions

sealed abstract class Exp

```
def deriv: Exp = this match {
    case X() => Const(1)
    case Const(i) => Const(0)
    case Add(e1, e2) => Add(e1.deriv, e2.deriv)
    case Mul(e1, e2) =>
        Add(Mul(e1, e2.deriv), Mul(e1.deriv, e2))
}
```

def + (e: Exp) = Add(this, e)

def * (e: Exp) = Mul(this, e)

case class X() extends Exp

case class Const(i: Int) extends Exp

case class Add(e1: Exp, e2: Exp) extends Exp

case class Mul(e1: Exp, e2: Exp) extends Exp

object Deriv extends App {

val e1 = X() * X() [= X().*(X())]

PrintExp(e1) → Mul(X(), X())

val e2 = (X() + Const(1)) * (Const(1) + X())

}

Vi sono due tipi di passaggio di parametri in Scala:

- 1) per valore, come visto finora (come C e JAVA)
- 2) per nome, come vediamo ora.

`def p(x:Int) = { println(x); x }`

`def f(a:Int, c:Boolean) = if(c) a else b`

`f(p(10), p(20), true) → 10 20 e ride 10`

`f(p(10), p(20), false) → 10 20 e ride 20`

// Come faccio a stampare solo il valore presso dall' if... else?

(con 2)

> PASSAGGIO PER NOME

`def f2(a: => Int, b: => Int, c: Boolean) = if(c) a else b`

`f2(p(10), p(20), true) // solo 10`

`f2(p(10), p(20), false) // solo 20`

// la valutazione di un'espressione passata per nome viene differente al momento in cui l'espressione viene valutata (nell'esempio tramite if... else)

`def MyWhile(cond: => Boolean)(body: => Unit): Unit = {`

`if (!cond) ()`

`else {`

`body`

`} MyWhile(cond)(body)`

`}`

→ In modo ricorsivo si realizza un While

Var i=0

```
MyWhile(i < 10) { // body del ciclo
    Print(m(i))
    i = i + 1
}
```

Esercizio 1.:

```
1  sealed abstract class Tree {
2      def treeTest:Boolean = this match {
3          case E() => true
4          case I(l,e,r) => (l,r) match {
5              case (E(),E()) => true
6              case (I(_,el,_),E()) => el < e && l.treeTest
7              case (E(),I(_,er,_)) => e < er && r.treeTest
8              case (I(_,el,_),I(_,er,_)) => {
9                  el < e && e < er && l.treeTest && r.treeTest
10             }
11         }
12     }
13 }
14
15 // albero non vuoto
16 case class I(l:Tree, e:Int, r:Tree) extends Tree
17
18 // albero vuoto
19 case class E() extends Tree
20
```

Esercizio 2.:

```
1  object E2 {
2      def getModel(n:Int):List[Shape] = {
3          (1 to n).toList.map(i=>Circle(0.5*i/n, 0.5*i/n, 0.5*i/n))
4      }
5 }
```

Esercizio 2: For

Frame2D.scala

E2.scala

E2Main.scala

+

```
1 object E2 {  
2     def getModel(n:Int):List[Shape] = {  
3         // (1 to  
4         // n).toList.map(i=>Circle(0.5*i/n,0.5*i/n,0.5*i/n))  
5         { for {  
6             i <- 1 to n  
7         } yield Circle(0.5*i/n,0.5*i/n,0.5*i/n)  
8     } } .toList  
9 }  
10 }
```

Esercizio 4

```
1 object E4 {  
2     def isAnagramOf(a:String,b:String) =  
3         a.sorted == b.sorted  
4 }
```

Esercizio 3

```
E3.scala  
1 case class Film(id:Int, titolo:String, anno:Int)  
2 case class Regista(id:Int, nome:String)  
3 case class DirettoDa(idFilm:Int, idRegista:Int)  
4  
5 case class DB(film:List[Film], registi:List[Regista], regie:List[Di  
6     I  
7     private val filmByIDFilm = film.groupBy(_.id)  
8     private val regieByIDRegista = regie.groupBy(_.idRegista)  
9     def registiConFilm(p:Film=>Boolean):List[Regista] = {  
0     registi.filter(  
1         regista => (regieByIDRegista contains regista.id) &&  
2             val elencoRegie = regieByIDRegista(regista.id)  
3             val elencoFilm = elencoRegie.map(  
4                 regia=>filmByIDFilm(regia.idFilm)).flatten  
5                 elencoFilm.exists(p)  
6             }  
7         }  
8     }  
9 }
```

METODI IMPLICIT

È possibile dichiarare un metodo "implicit"

Caso 1: $\text{X} \circlearrowleft^{\text{CLASSE A}} m(\dots)$

dove m è un metodo della classe A dell'oggetto X.

Se m non appartiene ad A:

Allora viene cercato un altro metodo dichiarato come implicit nello scope corrente che realizza una conversione da A \Rightarrow B, dove B è una classe che contiene il metodo m. A questo punto viene invocato il metodo m della classe B ottenuto dopo la conversione da A a B

object Prova extends App {

 val a = 10 somma 20

}

$\hookrightarrow 10 \in \text{Int}$ allora cerca metodo implicit da convertire a MioIntero.

object MioIntero {

 implicit def int2MioIntero(i: Int) = new MioIntero(i)

}

case class MioIntero(i: Int) {

 def somma(j: Int) = i + j

}

```
import MiaLista._
```

```
object Prova extends App {
```

```
  val l = List(1, 0, -1, 5).MaggioriZero()
```

```
  print(l)
```

FILE: MiaLista.scala

```
class MiaLista(l: List[Int]) = {
```

```
  def maggioriZero: List[Int] = l.filter(_ > 0)
```

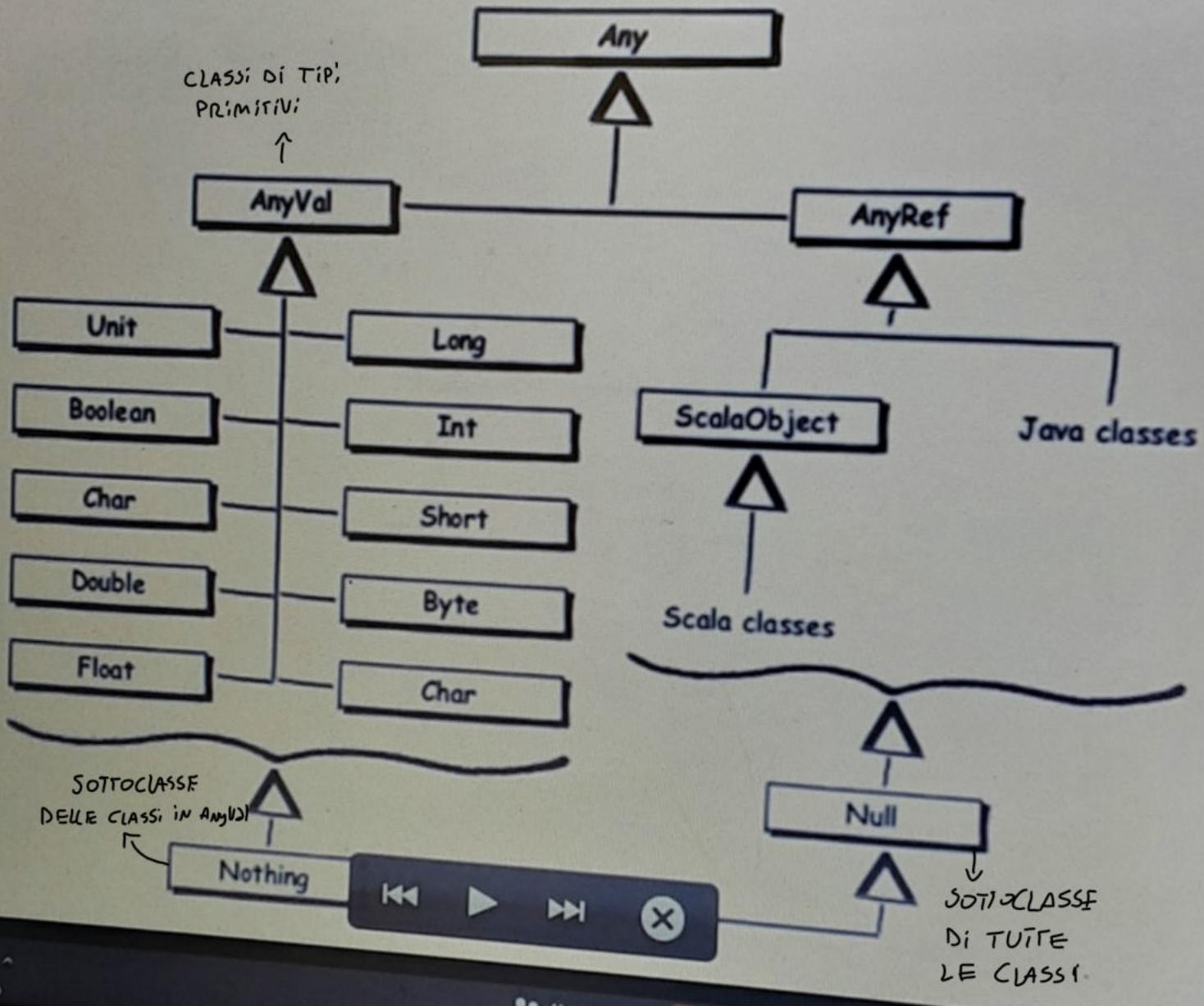
```
}
```

```
object MiaLista {
```

```
  implicit def lista2MiaLista(l: List[Int]) = new MiaLista(l)
```

~ import scala.language.implicitConversion

GERARCHIA CLASSI

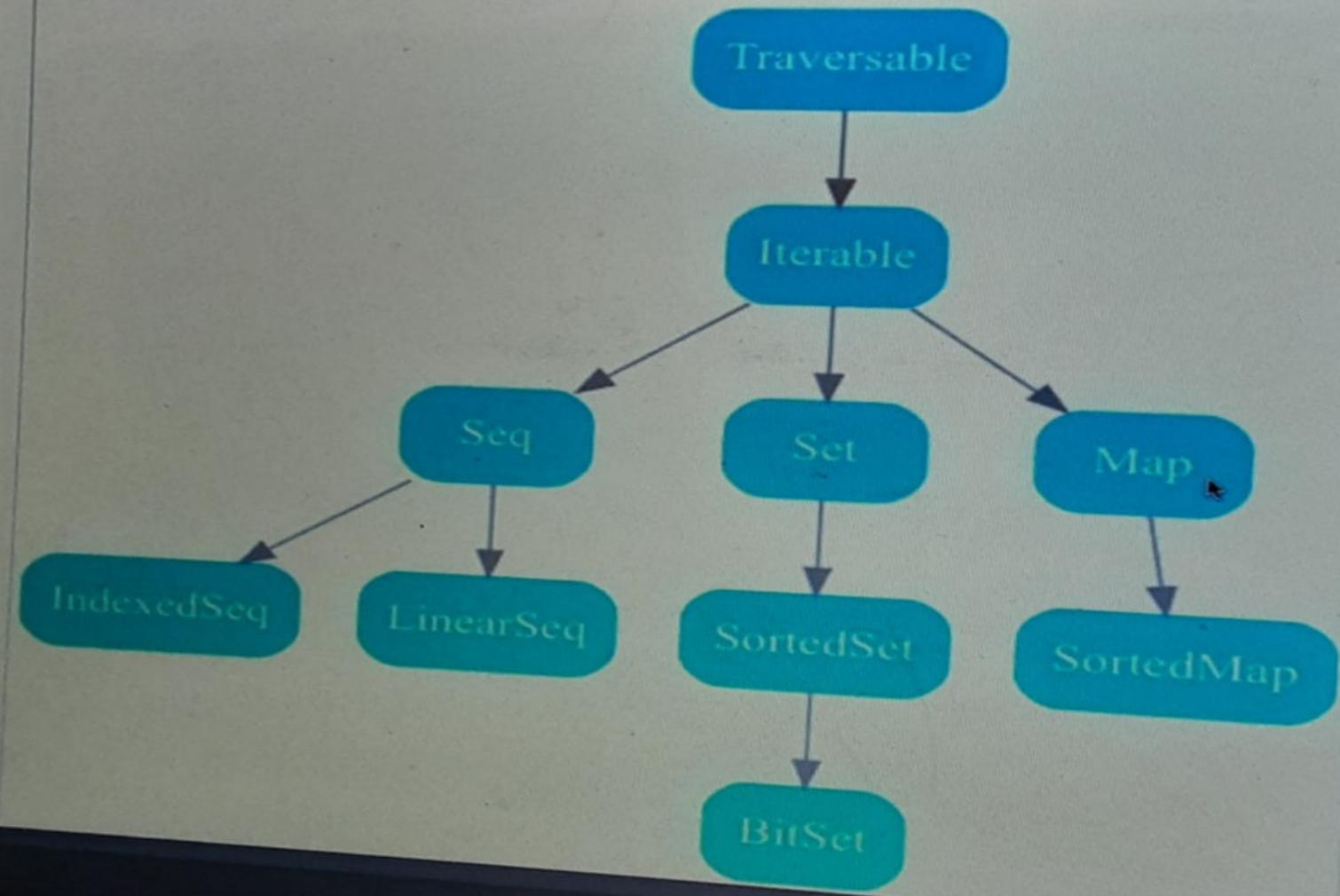


```

object SubClass extends App {
  def stampa(x: AnyVal) = x match {
    case i: Int => print("Int:" + i)
    case d: Double => print("Double:" + d)
    case _ => print("Non so...")
  }
}
  
```

Se metto case s: String => ... No INCOMPATIBILE con
ANYVAL => USO ANY

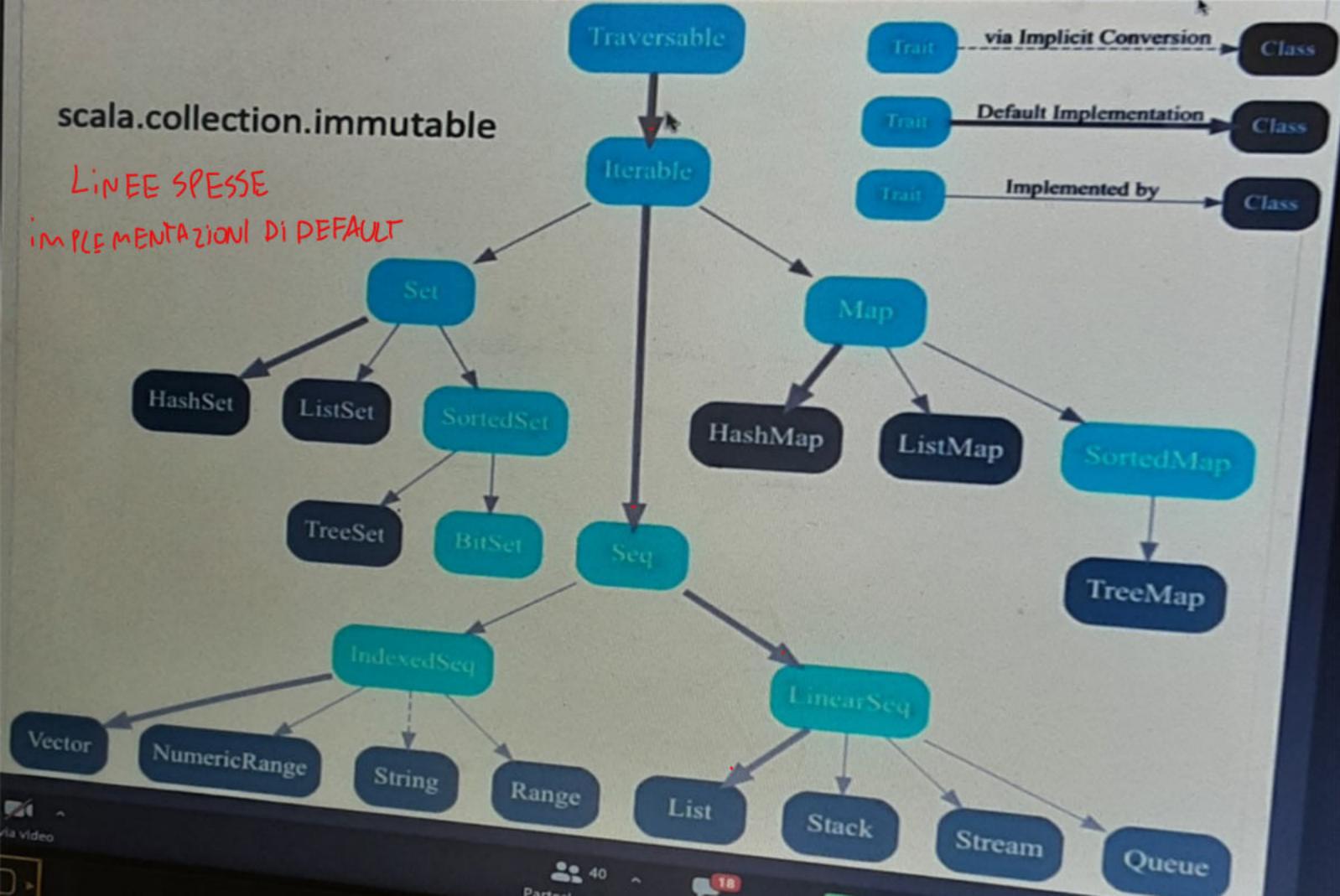
INTERFACCE



scala.collection.immutable

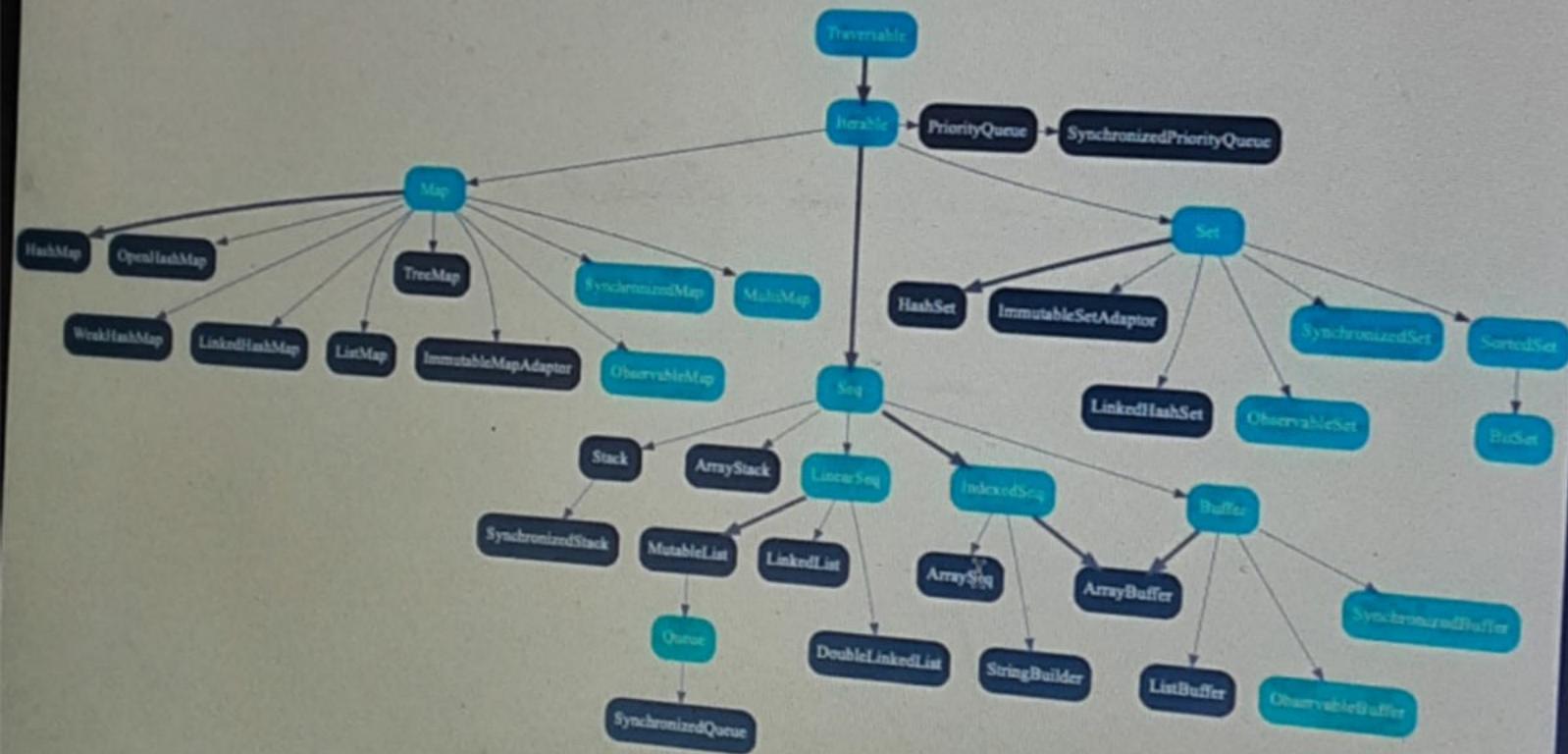
LINEE SPESSE

IMPLEMENTAZIONI DI DEFAULT



MUTABILI

scala.collection.mutable



implicit

Scenario 2: Come lo SCENARIO 1, ma si applica
al passaggio di parametri (rispetto all'assegnamento
di dello SCENARIO 1)

Esempio:

def m(x: A) ...

val o = m(x: B(...))

m(o)

(→ Se B non è una sottoclasse di A si verifica se esiste
un metodo di conversione implicit da B → A)

Esercitazione:

②

```
import scala.language.implicitConversions

class MyVect[T](v: Vector[T]) {
    def isMappedFrom[S](m: Vector[S], f: T => S) = {
        if (v.size != m.size) false
        else (0 until v.size).forall(i => m(i) == f(v(i)))
    }
}
```

Object E2 {

```
    implicit def vect2MyVect[T](v: Vector[T]) = new MyVect(v)
```

①

Object E1 {

```
    def scalarProd(a: Seq[Double], b: Seq[Double]): Double = {
        (a zip b).foldLeft(0.0)((acc, pair) => acc + pair._1 * pair._2)
    }
}
```

// ricorsivo

Object E1 {

```
    def scalarProd2(a: Seq[Double], b: Seq[Double]): Double = {
        def qux(l: Seq[(Double, Double)]): Double = {
            if (l.isEmpty) 0.0
            else l.head._1 * l.head._2 + qux(l.tail)
        }
        qux(a zip b)
    }
}
```

4.

Object E4 {

def repeat(m: Int)(body: => Unit): Unit = {

if (m < 1) ()

else {

body

repeat(m - 1)(body)

}

③

Per far sì che T sia comparabile

object E3 {

def moobSort[T](v: Vector[T])(implicit cmp: T => Ordered[T]) = {

v.permutations.find(x => x == x.sorted).get

}

Option[T]

- **None**: denota assenza di un valore
- **Some[T]**: presente di un valore

List(3, 1, 4).find(_ < 2).get
↓
Some(1)

Come si estrae un valore da un Option? Usando il metodo **get** sull'Option che vale Some(...)

Per discriminare se ho un Some o un None posso usare PATTERN MATCHING:

val o = List(3, 1, 5).find(_ < 2)

o match {
 case None => println("non trovato")
 case Some(x) => println("trovato" + x)}

// variante con getOrElse

println(o.getOrElse("non trovato"))
(o viene preso solo se o è un None)

```
E1.scala — E1 (git: master)  
object E1 {  
  def annataPiuVecchia(produttori: List[Produttore],  
                        vitigni: List[Vitigno],  
                        vini: List[Vino],  
                        produttore: String): Option[Int] = {  
    try {  
      val indiceProd = produttori.find(p => p.nome == produttore).get.idProd  
      val viniProd = vini.filter(v => v.idProd == indiceProd)  
      Some(viniProd.minBy(v => v.annata).annata)  
    }  
    catch {  
      case e: Exception => None  
    }  
  }  
}
```

ECCEZIONI

import java.io._

```
object Eccezioni extends App {
    val f = try {
        new FileReader("Eccezioni.scala")
    } catch {
        case e: FileNotFoundException => println("file not present")
        case e: IOException => println("IO exception")
    }
    println(f)
}
```

try {...} catch {...} è un' espressione in Scala
che restituisce il risultato del try o del catch

lazy val v = 10

→ v: Int = <lazy>

Il valore di v non viene calcolato

2 + 2 / 2 => 3

println(v) => 10 // qui viene assegnato

RICERCA BINARIA

```
object BimSearch {  
    def search[T](v: Vector[T], x: T)(implicit cmp: T => Int) : Boolean = {  
        def aux(a: Int, b: Int): Boolean = {  
            if (a >= b) false  
            else {  
                val mid = (a+b)/2  
                if (v(mid) == x) true  
                else if (x < v(mid)) aux(a, mid)  
                else aux(mid+1, b)  
            }  
        }  
        aux(0, v.length)  
    }  
}
```

REDUCE (VARIANTE)

```
object MyRed {  
    implicit def list2MyList[T](l: List[T]): MyRed[T] = {  
        myRed(l)  
    }  
    case class MyRed[T](l: List[T]) {  
        def myReduce(f: (T, T) => T): Option[T] = {  
            def aux(m: List[T]): T = { // m non vuoto  
                require(!m.isEmpty)  
                if (m.tail.isEmpty) m.head  
                else f(m.head, aux(m.tail))  
            }  
            if (l.isEmpty) None else Some(aux(l))  
        }  
    }  
}
```

FUNZIONE di PROFILAZIONE (misura il tempo)

```
object Prof {  
    def profile[T](body: => T): (T, Long) = {  
        val start = System.currentTimeMillis  
        val v = body  
        val t = System.currentTimeMillis - start  
        (v, t)  
    }  
}
```

STREAM o LAZY LIST

concatena elementi con lista Lazy

```
def myStream: LazyList[Int] = 1 #:: 2 #:: 3 #:: LazyList.empty
```

Lazy perché i valori non vengono calcolati subito.

```
def pari(m: Int): LazyList[Int] = m #:: pari(m+2)
```

pari(0) not computed

```
pari(0).take(5).toList => List(0, 2, 4, 6, 8)
```

↪ Abbiamo un STREAM infinito

```
def fib(a: Int, b: Int): LazyList[Int] = a #:: fib(b, a+b)
```

```
fib(0, 1).take(5).toList => List(0, 1, 1, 2, 3)
```

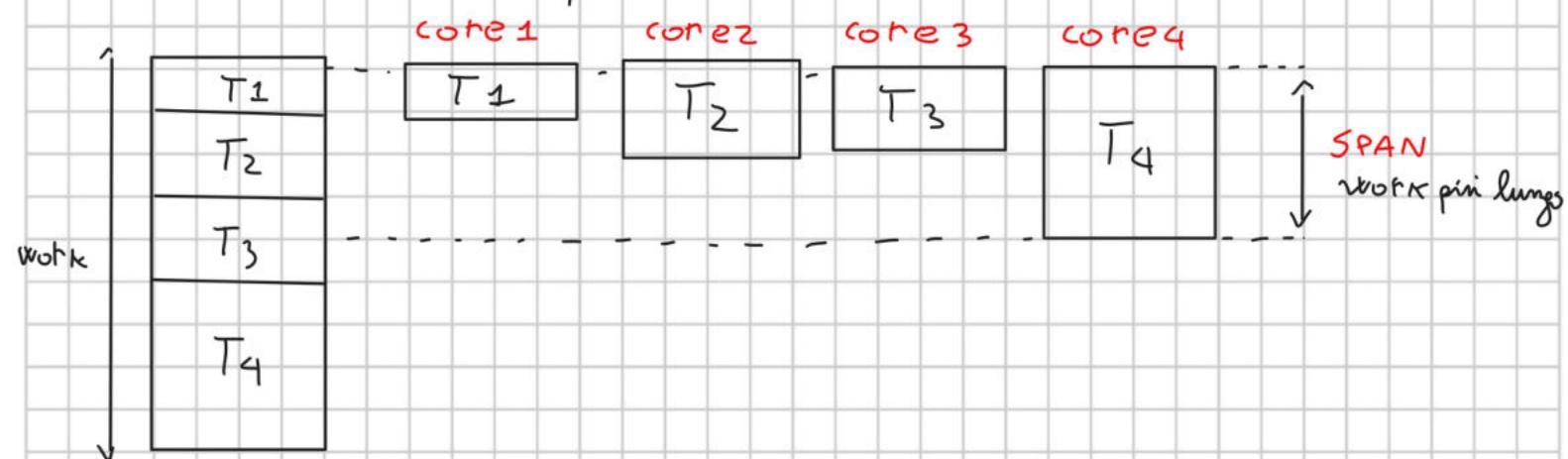
```
fib(1, 1).take(5).toList => List(1, 1, 2, 3, 5)
```

Elemento più frequente in una sequenza

```
object PiuFreq {
    def piuFrequente[T](l: Seq[T]): Option[(T, Int)] = {
        if (l.isEmpty) None
        else {
            val m = l.groupBy(identity).reduce((x, y) => if (x._2.size >
                Some(m._1, m._2.size)) y._2.size) X else y)
        }
    }
}
```

PARALLELISMO

Affiorano dei TASK (porzioni di codice)



$$\text{Speedup} = \frac{T_{\text{SEQUENZIALE}}}{T_{\text{PARALLELO}}}$$

Distingueremo parti parallelistabili da un certo codice da parti intrinsecamente sequenziali

LEGE DI Amdahl

Supponiamo di dividere l'esecuzione di un programma in due parti:

A = parte sequenziale

B = parte parallelizzabile

con $k = m^o$ di core disponibili

Allora

$$T_A = \text{tempo esecuzione A}$$

$$T_B = \text{tempo esecuzione B}$$

$$\begin{aligned} T_A &= (1-\alpha)T \\ T_B &= \alpha T \end{aligned} \quad \left. \begin{array}{l} \\ \end{array} \right\} T = T_A + T_B$$

Per un certo α

Facendo girare B in parallelo otteniamo uno speedup pari a:

$$S = \frac{T}{T'_B + T_A} \quad \text{dove } T'_B = \frac{T_B}{K}$$

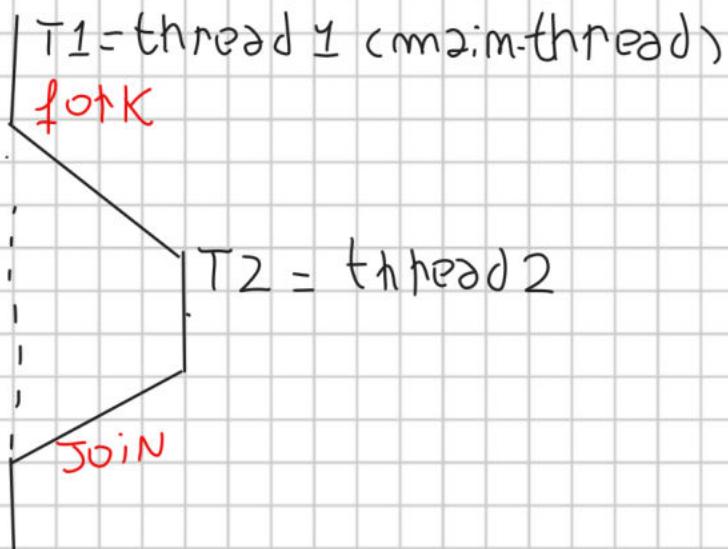
$$S = \frac{\cancel{T}}{\frac{\alpha \cancel{T}}{K} + (1-\alpha) \cancel{T}} = \frac{1}{\frac{\alpha}{K} + 1 - \alpha}$$

Esempio:

$\alpha = 0.5$ (codice parallelizzabile), $K = 2$ core

$$S = \frac{1}{\frac{0.5}{2} + 1 - 0.5} = \frac{1}{0.25 + 0.5} = \frac{1}{0.75} = \boxed{1,33}$$

FORK-JOIN model:



htop: temps exécution
sur chaque core

THREAD IN SCALA

object Task {

def doTask(m: Long) : Unit = {

var i = 0

while (i < m) {

i += 1

}

}

object ForkJoinMain extends App {

val m = 220000000L

val r = new Runnable {

def run() = { Task.doTask(m) }

val t = new Thread(r)

t.start()

Task.doTask(m)

t.join()

}

// implementiamo il funzionamento di thread in funzioni Unit

object ParUnit {

def par (a: => Unit) (b: => Unit) = {

val t = new Runnable {

} def run = {

val t = new Thread(t)

t.start()

b

t.join()

}

Se vogliamo che a e b restituiscano qualcosa e non Unit

object ParUnit {

def par [A, B] (a: => A) (b: => B) = {

val resA: Option[A] = None

val t = new Runnable {

def run() = {

} resA = Some(a)

}

val t = new Thread(t)

t.start()

val resB = b

t.join()

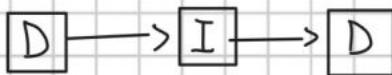
(resA.get, resB)

}

TASSONOMIA DI FLYN :

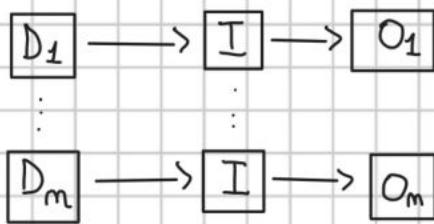
Architettura si divide in 4 parti.

SISD = single instruction - single data

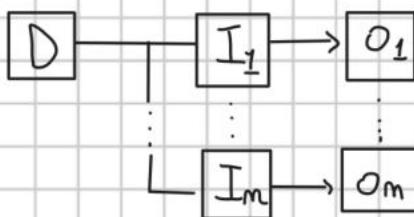


è il modello sequenziale standard

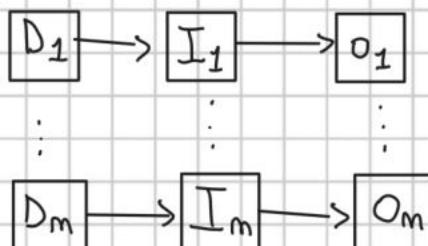
SIMD = single instruction multiple data



MISD = multiple instruction - single data

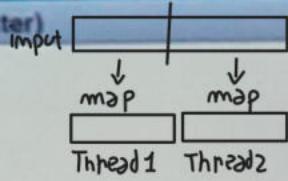


MIMD = multiple instruction multiple data



MIMD si riferisce al modello multi-core

TextMate File Edit View Navigate Text File Browser Bundles Window

ParMap.scala — ParMap (git: master) Input 

```

1 import Par._
2
3 object ParMap {
4     def map[T,S](l: List[T], f: T => S) = {
5         val (a,b) = l.splitAt(l.size/2)
6         val (ma,mb) = par {
7             a.map(f)
8         }
9         {
10             b.map(f)
11         }
12         ma :::: mb
13     }
14 }

```

```

object Prof {
    def profile[T](body: => T): (T, Double) = {
        val start = System.nanoTime
        val v = body
        val t = System.nanoTime - start
        (v, t * 1E-9)
    }
}

```

Se abbiamo poco WORK, la versione sequenziale potrebbe risultare migliore di queste parallele.

RaceConditionMain.scala

```

import Par._

object RaceConditionMain extends App {
    var sum = 0
    var i = 0
    while (i < 1000000000) {
        i += 1
        sum += 1
    }
    i = 0
    while (i < 1000000000) {
        i += 1
        sum -= 1
    }
    println("Sequential: " + sum)

    sum = 0
    par {
        var i = 0
        while (i < 1000000000) {
            i += 1
            sum += 1
        }
    }
    var i = 0
    while (i < 1000000000) {
        i += 1
        sum -= 1
    }
    println("Parallel: " + sum)
}

```

Per accedere a dati condivisi ci sono bisogno
di sincronizzazione: semafori, ...

→ incrementismo e decrementismo

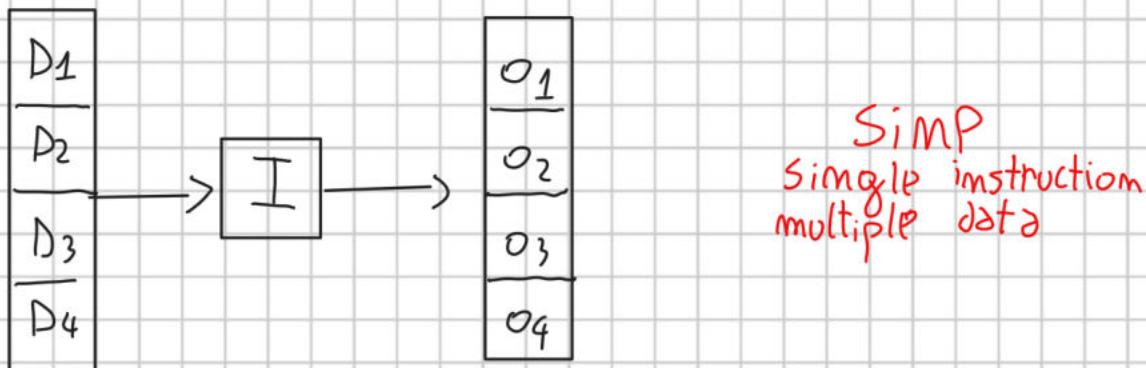
Sum di 1 in parallelo, viene

O come quello sequenziale? NO

Race condition

Seq: 0 parallel: 1789...

VETORIZZAZIONE: classica intuizione del modello SIMD, Esempio:



Più concretamente si possono ad esempio sommare dati multipli con una stessa istruzione

$$\begin{array}{c} A_1 \\ \hline A_2 \\ \hline A_3 \\ \hline A_4 \end{array} + \begin{array}{c} B_1 \\ \hline B_2 \\ \hline B_3 \\ \hline B_4 \end{array} = \begin{array}{c} C_1 \\ \hline C_2 \\ \hline C_3 \\ \hline C_4 \end{array}$$

\overrightarrow{A} \overrightarrow{B} \overrightarrow{C}

Somme parallele realizzate da una sola istruzione macchina

Questo tipo di parallelismo si chiama **instruction-level parallelism**.

Le CPU moderne offrono una batteria di istruzioni instruction-level che coprono diverse operazioni possibili.

Lunga storia: i produttori di CPU hanno incorporato nelle

CPU numerose istruzioni SIMD principalmente per scopi ludici di grafica computerizzata per videogiochi.

La storia parte dalle estensioni mmx (1996), poi:

3DNow! - 1998

SSE - 1999

SSE2 - 2001

SSE3 - 2004

SSE4 - 2006

AVX - 2008

AVX2 - 2012

AVX512 - 2015

I compilatori forniscono degli **intrinsics**, cioè chiamate che assomigliano a chiamate a funzione, ma in realtà si mappano su istruzioni macchina.

Questo ci permette di non scrivere in ASSEMBLY ma in C/C++, molto comodo per il programmatore.

Esempio: siamo due variabili a e b del tipo **_m128i**

Queste variabili possono essere interpretate come:

1) 4 interi di 32 bit

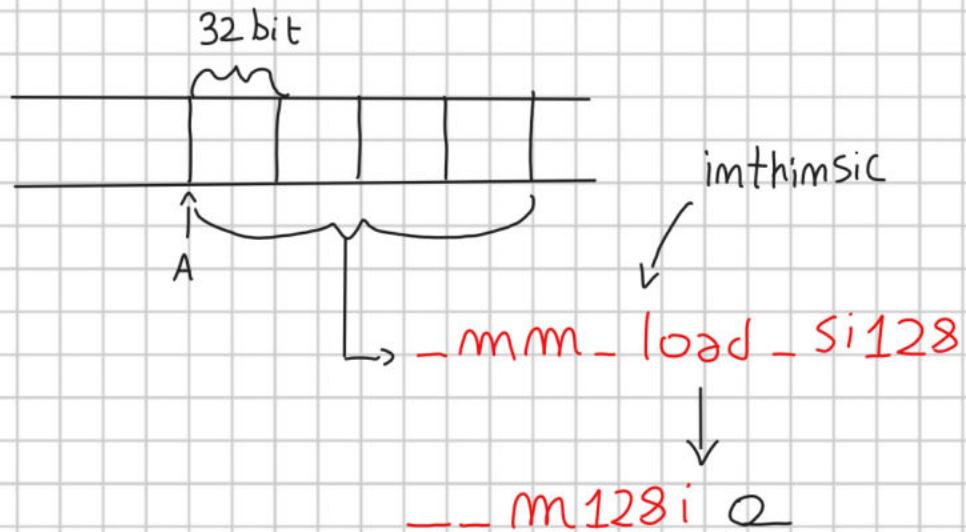
2) 8 interi di 16 bit

3) 16 interi di 8 bit

L'istruzione $c = \underline{\text{mm_add_epi32}}(a, b)$

fa l'assunzione 1) e calcola 4 somme in parallelo

Per inizializzare delle variabili vettoriali c'è un'operazione di copia dei dati da memoria:



In pratica:

__m128i o;

o = _mm_load_si128 ((const __m128i *) A)

Esempio somma di 4 int alla volta in parallelo:

Void somma (int A[4], int B[4], int C[4]) {

__m128i o, b, c;

o = _mm_load_si128 ((const __m128i *) A);

b = _mm_load_si128 ((const __m128i *) B);

c = _mm_add_epi32 (o, b);

_mm_store_si128 ((__m128i *) C, c);

↳ salvo in memoria il risultato.

```

#include <stdio.h>
#include <immintrin.h>

void somma(int A[4], int B[4], int C[4]) {
    C[0] = A[0] + B[0];
    C[1] = A[1] + B[1];
    C[2] = A[2] + B[2];
    C[3] = A[3] + B[3];
}

void somma_sse(int A[4], int B[4], int C[4]) {
    __m128i a, b, c;
    a = _mm_load_si128((const __m128i*)A);
    b = _mm_load_si128((const __m128i*)B);
    c = _mm_add_epi32(a,b);
    _mm_store_si128((__m128i*)C, c);
}

int main() {
    int A[4] = { 1, 2, 3, 4 };
    int B[4] = { 4, 3, 2, 1 };
    int C[4];
    somma_sse(A,B,C);
    printf("%d %d %d %d\n", C[0], C[1], C[2], C[3]); // stampa 5 5 5 5
    return 0;
}

```

EQUIVALENTI

vecsum_sse.c SOMMA CON M

```

1 #include <immintrin.h>
2 #include "vecsum.h"

3     SUM
4 void somma_sse(int A[4], int B[4], int C[4]) {
5     __m128i a, b, c;
6     a = _mm_load_si128((const __m128i*)A);
7     b = _mm_load_si128((const __m128i*)B);
8     c = _mm_add_epi32(a,b);
9     _mm_store_si128((__m128i*)C, c);
10 }

11
12 void vecsum(int* A, int* B, int* C, int n) {
13     int i;
14     for (i=0; i+3<n; i+=4) sum_sse(A+i, B+i, C+i);
15     for (; i<n; i++) C[i] = A[i] + B[i];
16 }

17     ↳ CASO SE M NON MULTIPLO DI 4

```

Tipi vettoriali: oggetti vettoriali che possono essere tenuti in particolari registri della CPU. Vengono chiamati **packed data type**

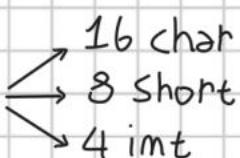
Tipi	Dimensioni	Estensioni	Registri Usati
<code>_m64</code>	8 byte	MMX	8 (MM0-MM7)
<code>_m128i</code>	16 byte	SSE	16 (XM0-XM15)
<code>_m256i</code>	32 byte	AVX	16 (YM0-YM15)

Operazioni più comuni:

`_m128i _mm_load_si128 (_m128i const * mem_addr)`

Restituisce oggetto `_m128i` (packed data) con i 16 byte di dati interni da memoria.

Questi dati possono rappresentare



```

graph TD
    A[16 byte] --> B[8 short]
    A --> C[4 int]
  
```

Il tipo effettivo dipenderà dalle particolari operazioni che ci faremo sopra

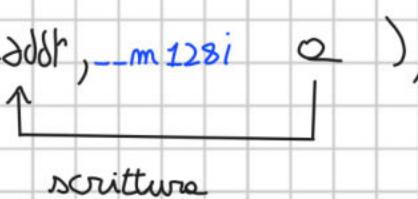
Più in particolare:

Intero	Capienza		
<code>_m64</code>	<code>char[8]</code>	<code>Short[4]</code>	<code>Int[2]</code>
<code>_m128i</code>	<code>char[16]</code>	<code>Short[8]</code>	<code>Int[4]</code>
<code>_m256i</code>	<code>char[32]</code>	<code>Short[16]</code>	<code>Int[8]</code>

Copie da rete o memoria:

`Void _mm_store_si128 (_m128i * mem_addr, _m128i o);`

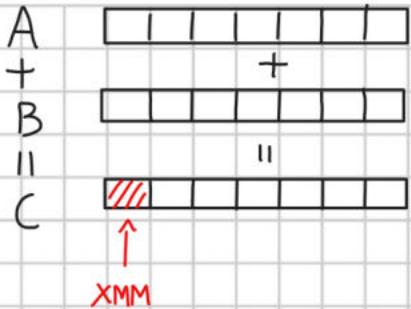
COPIA OGGETTO VETTORIALE `o`
ALL'INDIRIZZO DI MEMORIA `mem_addr`



```

graph LR
    A[mem_addr] --> B[o]
    B --> C[scrivere]
  
```

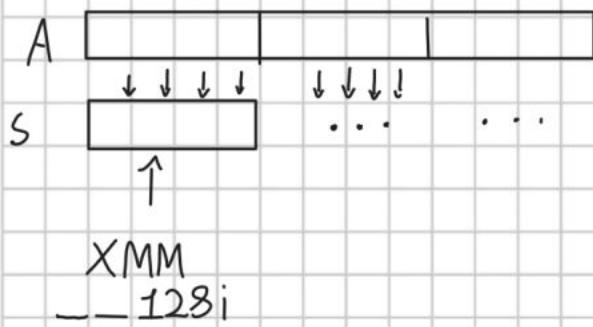
scrivere



registro vettoriale SSE di 16 byte

Nell'Avx ci sono registri vettoriali di dimensione 32 byte

Vediamo ora somma degli elementi di un vettore:



Alla fine sommiamo gli elementi del vettore S e otteniamo il risultato desiderato.

E

```

#include <stdio.h>

int arraysum(int *v, int n) {
    int i, sum = 0;
    for (i=0; i<n; ++i) sum += v[i];
    return sum;
}

int arraysum_unroll(int *v, int n) {
    int i, c[4] = {0}, sum = 0;
    for (i=0; i+3<n; i += 4) {
        c[0] += v[i+0]; // c[0] = *(v+i)
        c[1] += v[i+1];
        c[2] += v[i+2];
        c[3] += v[i+3];
    }
    for (; i<n; ++i) sum += v[i];
    return sum+c[0]+c[1]+c[2]+c[3];
}

int arraysum_sse(int *v, int n) {
    // scrivere la soluzione vettorizzata qui...
    return 0;
}

int main() {
    int v[] = { 1,1,1,1,1,1,1,1,1,1 };
    int x = arraysum_sse(v,10);
    printf("x=%d\n", x);
}

```

#include <iomanip.h>

```

int arraysum_sse (int* V, int m) {
    int i, res[4], sum = 0;
    __m128i s = _mm_set_epi32(0,0,0,0), inizializzo vettore m128i e  
sette i valori
    for(i=0, i+3 <m, i+=4) {
        __m128i vv = _mm_load_si128((const __m128i*)(V+i));
        s = _mm_add_epi32(s,vv);
    }
    _mm_store_si128((__m128i*)res,s),
    for(;i<m;i++) sum += V[i]
    return sum + res[0]+res[1]+res[2]+res[3];
}

```

```

int Cdiff (const char *X, const char *y, int m) {
    int i, j, cmt = 0;
    unsigned char mcmt[16];
    _m128i XV, yV, res, vcmt, vome;
    vome = _mm_set_epi8(1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1);
    vcmt = _mm_set_epi8(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0);
    for(i=0; i+15 < m; i+=16) {
        XV = _mm_load_si128((const _m128i*)(X+i));
        yV = _mm_load_si128((const _m128i*)(y+i));
        res = _mm_cmpeq_epi8(XV, yV);
        res = _mm_add_epi8(res, vome);
        vcmt = _mm_add_epi8(vcmt, res);
    }
    for(; i < m; i++)
        if(X[i] != y[i]) cmt++;
    _mm_store_si128(_m128i*)mcmt, vcmt);
    for(j=0; j<16; j++) cmt += mcmt[j];
}
return cmt;
}

```

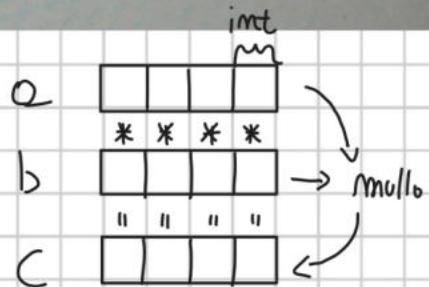
Qui potremmo ottenere OVERFLOW in vcmt se superiamo valori > 255 *

PRODOTTO SCALARE DUE VETTORI

```
#include "dotprod.h"
#include <immintrin.h>

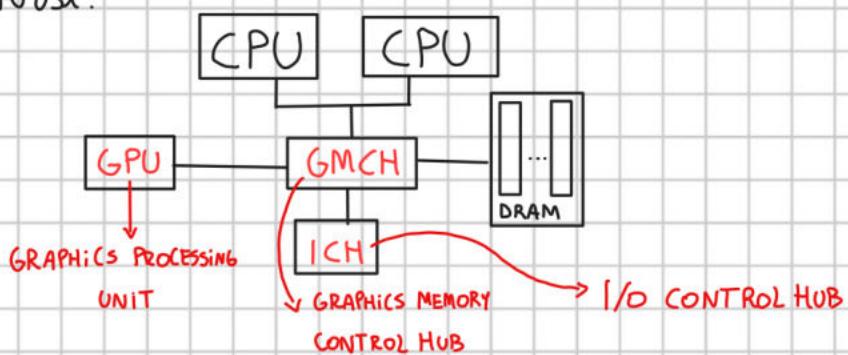
int dotprod_equiv(int* A, int* B, int n) {
    int i, s = 0;
    for (i=0; i<n; ++i) s += A[i] * B[i];
    return s;
}

int dotprod(int* A, int* B, int n) {
    int i, D[4];
    __m128i a, b, c, d;
    d=_mm_set_epi32(0,0,0,0);
    for(i=0; i+3 < n; i+=4) {
        a=_mm_load_si128((const __m128i*)(A+i));
        b=_mm_load_si128((const __m128i*)(B+i));
        c=_mm_mullo_epi32(a,b);
        d=_mm_add_epi32(d,c);
    }
    _mm_store_si128((__m128i*)D, d); // COPIA REGISTRO VETTORIALE A MEMORIA
    for(; i < n; i++) D[0] += A[i] * B[i]
    return D[0] + D[1] + D[2] + D[3]
}
```



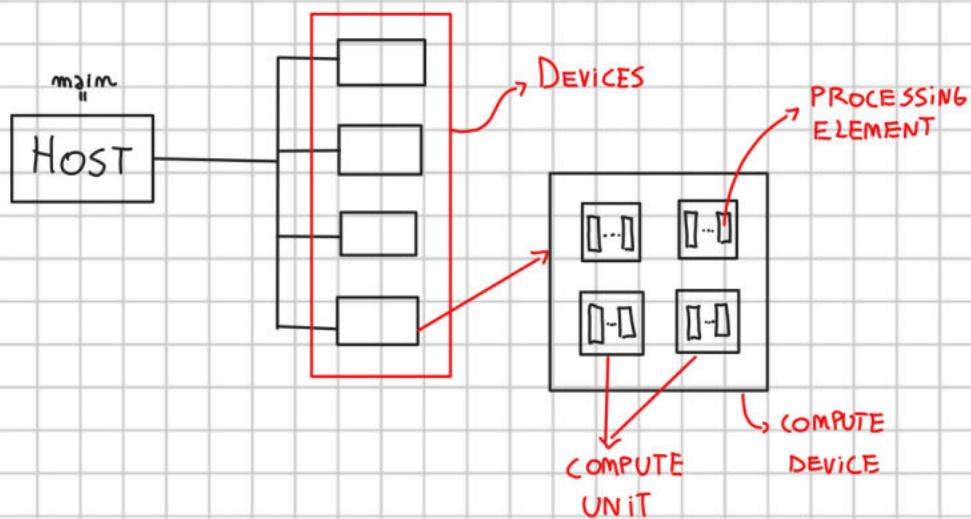
OPENCL

Le piattaforme moderne sono **eterogenee**, cioè contengono dispositivi di calcolo diversi.



OpenCL = modello di calcolo per le programmazione di sistemi eterogeni.

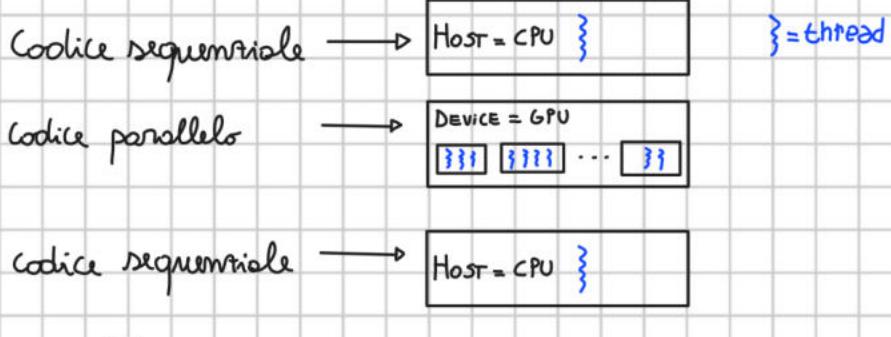
Modello di piattaforma:



Anatomie di un'applicazione OpenCL

- Codice sequenziale su un thread nell'Host (CPU)
- Codice parallelo gira su molti thread nei device, ciascuno in un processing element.

OpenCL Application



Modello di esecuzione:

L'applicazione OpenCL gira su un Host che invia lavoro da eseguire ai device

- **WORK-ITEM**: unità di lavoro base di un device (GPU) → è un Thread
- **KERNEL**: codice che descrive le azioni dei WORK-ITEM. → codice del Thread Scritte in un dialetto del C (OpenCL C)
- **PROGRAMMA**: collezione di Kernel e altre funzioni di appoggio.
- **CONTESTO**: l'ambiente in cui girano i WORK-ITEM sui device.
- **CODA DEI COMANDI**: coda usata dall'applicazione Host per inviare lavoro da svolgere su un device. (Es.: WORK-ITEM = istanze di esecuzione di un Kernel).

Corrispondenza tra modello Hardware e modello di esecuzione.

modello ESECUZIONE

modello HARDWARE

WORK ITEM
(THREAD)

← → PROCESSING ELEMENT (CORE DEL DEVICE)



WORK GROUP

(Blocco di THREAD)

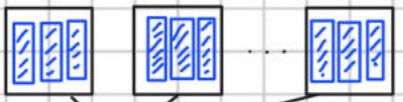
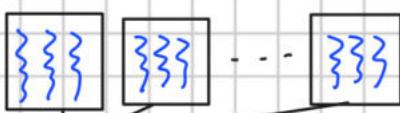


← → COMPUTE UNIT
(STREAMING MULTI-PROCESSOR)



Istanza di esecuzione
di un Kernel (GRIGLIA DI
ESECUZIONE)

← → COMPUTE DEVICE
(GPU ad esempio)



Work Group

STREAMING MULTI-PROCESSOR

Il calcolo su un device avviene definendo un dominio computazionale **N-dimensionale**, con valori tipici di $N=1, 2$

Ese.: Calcolo su array $N=1$, Calcolo su matrici $N=2$

VERSIONE SEQUENZIALE

```
Void SQUARE( INT* input,
              INT* output,
              INT m) {
    INT i;
    FOR (i=0; i<m; i++)
        OUTPUT[i] = input[i] * input[i];
}
```

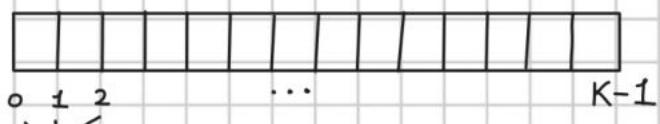
VERSIONE OPENCL (KERNEL)

```
-- KERNEL Void SQUARE( __ GLOBAL INT* input,
                        __ GLOBAL INT* output,
                        INT m) {
    INT id = get_global_id(0);
    IF (id >= m) return;
    output[id] = input[id] * input[id];
}
```

↳ qui nel kernel solo un'operazione aritmetica

Dominio computazionale ($N=1$)

(global dimension)



id del Work Item

N.B.: Ogni work item ha un suo id numerico che lo identifica univocamente. Nel codice assumiamo che gli id dei work item siano usati come indici degli array:

$$\text{OUTPUT}[ID] = \text{INPUT}[ID] * \text{INPUT}[ID]$$

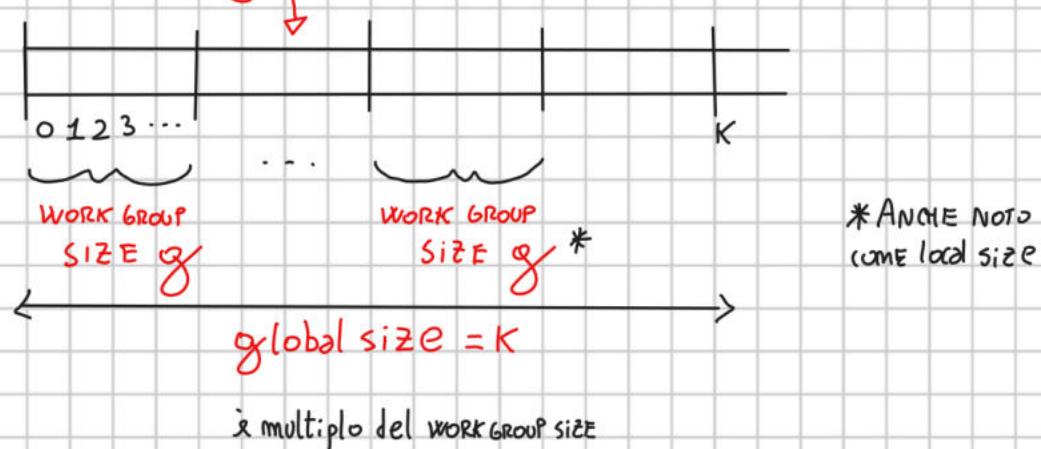
↑ ↑ ↑

indice di un WORK ITEM (THREAD)

Vediamo come istanziare il Kernel visto in modo da avere almeno N thread?

Perché non esattamente N thread?

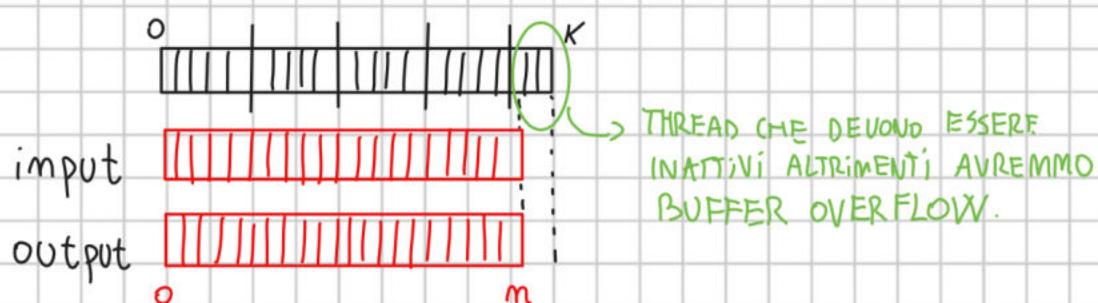
Perché i WORK ITEM si raggruppano in WORK GROUP e il numero totale di WORK ITEM è un multiplo delle dimensioni di un WORK GROUP: $NDRange$ è il dominio computazionale



Quindi avremo che K è il più piccolo multiplo delle WORK GROUP SIZE che è maggiore di N .

Si ha: $K = \left\lfloor \frac{N + g - 1}{g} \right\rfloor \cdot g$

Dobbiamo quindi istanziare nel indice HOST K WORK ITEM (dimensione globale) in modo da avere un thread per ogni prodotto da calcolo.



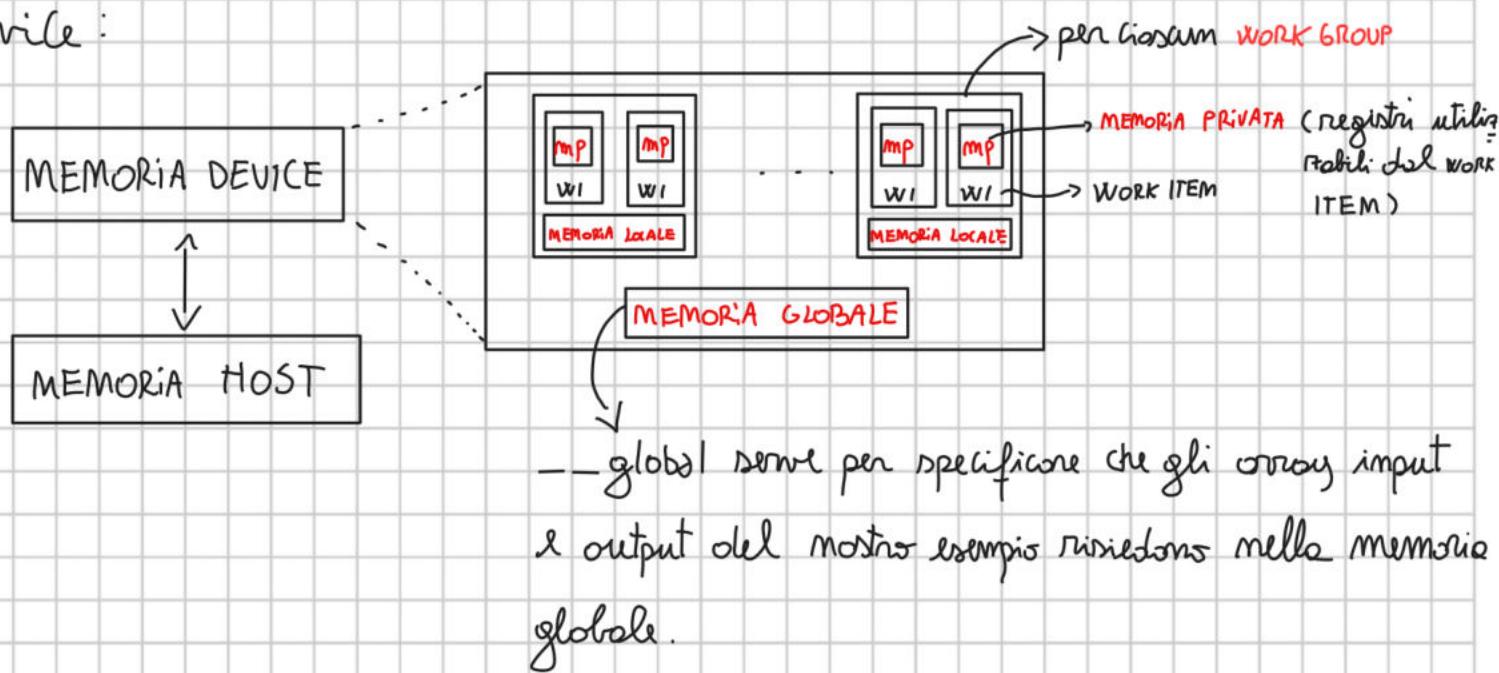
Quanto è il motivo per cui scriviamo:

`if(id >= N) return;`

proprio per evitare buffer overflow.

Abbiamo visto il codice DEVICE (Kernel). Elenchiamo ora i passi necessari per il codice host per istanziare una griglia monodimensionale e creare tutte le risorse necessarie all'esecuzione del programma openCL.

Prima di fare questo mostriamo il modello di memoria di un device:



N.B.: Il fatto che host e device abbiano memorie distinte implica che dovremo fare operazioni di copia:

1) Da Host → DEVICE (GPU)

per caricare l'input nella GPU

2) Da DEVICE → Host

per recuperare il risultato del calcolo

N.B.: Per gestire blocchi di memoria nel device ci sono un allocatore di memoria dedicato al device, invisibile dall' Host.

Passi tipici di un programma Host:

- 1) Query della piattaforma (piattaforma include un insieme di device)
- 2) Query del device (o dei device) che si intende utilizzare.
- 3) Creazione CONTESTO per il DEVICE da utilizzare.
- 4) Gestione code dei comandi per il contesto.
- 5) Compilazione di un programma OpenCL da far girare.
- 6) Instantiazione di un Kernel preso dal programma.
- 7) Allocazione dei buffer di memoria nelle memorie globale del DEVICE.
- 8) Caricamento dei dati di input da Host o DEVICE.
- 9) Passaggio dei parametri al Kernel (Es. indirizzi dei buffer di input e di output).
- 10) Instantiazione di un NDRange (dominio computazionale dei WORK ITEM) di dimensioni locali e globali opportune.
- 11) Copie dei buffer di output dal DEVICE all' Host.
- 12) Rilascio delle risorse allocate (su Host e) DEVICE.

devinfo

```
gcc -O2 devinfo.c -o devinfo -lOpenCL
```

```
#include <stdio.h>
#include <stdlib.h>

#define CL_USE_DEPRECATED
#define CL_SILENCE_DEPRECATED

#ifndef __APPLE__
#include <OpenCL/opencl.h>
#else
#include <CL/cl.h>
#endif

// Use a static data size
#define DATA_SIZE (20*1024)
```

```
// Simple compute kernel that computes the square of an input array
const char *KernelSource = "\n" \
"__kernel void square(\n"
"    __global float* input,\n"
"    __global float* output,\n"
"    const unsigned int count)\n"
"{
"    int i = get_global_id(0);
"    if(i < count)
"        output[i] = input[i] * input[i];
"}\n";
```

Kernel come strings

↳ ve eseguito sul DEVICE

```
int main(int argc, char** argv)
{
    int err;                                // error code returned from api calls
    float* data;                            // original data set given to device
    float* results;                         // results returned from device
    unsigned int correct;                   // number of correct results returned
    size_t global;                          // global domain size for our calculation
    size_t local;                           // local domain size for our calculation
    cl_platform_id platform_id;            // compute platform id
    cl_device_id device_id;                // compute device id
    cl_context context;                   // compute context
    cl_command_queue commands;             // compute command queue
    cl_program program;                  // compute program
    cl_kernel kernel;                    // compute kernel
    cl_mem input;                        // device memory used for the input array
    cl_mem output;                       // device memory used for the output array

    // Allocate chunks on host
    data = malloc(DATA_SIZE*sizeof(float));
    results = malloc(DATA_SIZE*sizeof(float));

    if (data == NULL || results == NULL) {
        printf("Error: Failed to allocate input/output buffers on host!\n");
        return EXIT_FAILURE;
    }

    // Fill our data set with random float values
    //
    int i = 0;
    unsigned int count = DATA_SIZE;
    for(i = 0; i < count; i++)
        data[i] = rand() / (float)RAND_MAX;

    // Connect to a platform
    err = clGetPlatformIDs(1, &platform_id, NULL);
    if (err != CL_SUCCESS) {
        printf("Error: no OpenCL platform available!\n");
        return EXIT_FAILURE;
    }

    // Connect to a compute device
    // try to connect to a GPU
    err = clGetDeviceIDs(platform_id,
                         CL_DEVICE_TYPE_GPU, 1,
                         &device_id, NULL);

    if (err != CL_SUCCESS) {

        // if no GPU is available, use CPU
        err = clGetDeviceIDs(platform_id,
                             CL_DEVICE_TYPE_CPU, 1,
                             &device_id, NULL);
        if (err != CL_SUCCESS) {
            printf("Error: no device available!\n");
            return EXIT_FAILURE;
        }
    }
}
```

```
// Create a compute context
//
context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);
if (!context) {
    printf("Error: Failed to create a compute context!\n");
    return EXIT_FAILURE;
}

// Create a command commands
//
commands = clCreateCommandQueue(context, device_id, 0, &err);
if (!commands) {
    printf("Error: Failed to create a command commands!\n");
    return EXIT_FAILURE;
}

// Create the compute program from the source buffer
//
program = clCreateProgramWithSource(
context, 1, (const char **) &KernelSource, NULL, &err);
if (!program) {
    printf("Error: Failed to create compute program!\n");
    return EXIT_FAILURE;
}

// Build the program executable
//
err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
if (err != CL_SUCCESS) {
    size_t len;
    char buffer[2048];

    printf("Error: Failed to build program executable!\n");
    clGetProgramBuildInfo(program, device_id, CL_PROGRAM_BUILD_LOG, sizeof(buffer),
printf("%s\n", buffer);
    return EXIT_FAILURE;
}

// Create the compute kernel in the program we wish to run
//
kernel = clCreateKernel(program, "square", &err);
if (!kernel || err != CL_SUCCESS) {
    printf("Error: Failed to create compute kernel!\n");
    return EXIT_FAILURE;
}

// Create the input and output arrays in device memory for our calculation
//
input = clCreateBuffer(context, CL_MEM_READ_ONLY, sizeof(float) * count, NULL, NULL);
output = clCreateBuffer(context, CL_MEM_WRITE_ONLY, sizeof(float) * count, NULL, NULL);
if (!input || !output) {
    printf("Error: Failed to allocate device memory!\n");
    return EXIT_FAILURE;
}
```

```

// Write our data set into the input array in device memory
//
err = clEnqueueWriteBuffer(commands, input, CL_TRUE, 0, sizeof(float) * count, data,
if (err != CL_SUCCESS) {
    printf("Error: Failed to write to source array!\n");
    return EXIT_FAILURE;
}

// Set the arguments to our compute kernel
//
err = 0;
err = clSetKernelArg(kernel, 0, sizeof(cl_mem), &input);           ↑ N° ARGOMENTO
err |= clSetKernelArg(kernel, 1, sizeof(cl_mem), &output);
err |= clSetKernelArg(kernel, 2, sizeof(unsigned int), &count);
if (err != CL_SUCCESS) {
    printf("Error: Failed to set kernel arguments! %d\n", err);
    return EXIT_FAILURE;
}

// Get the maximum work group size for executing the kernel on the device
//
err = clGetKernelWorkGroupInfo(kernel, device_id, CL_KERNEL_WORK_GROUP_SIZE, sizeof(          ↓
if (err != CL_SUCCESS) {
    printf("Error: Failed to retrieve kernel work group info! %d\n", err);
    return EXIT_FAILURE;
}

printf("-- local size = %lu items\n", local);

// Execute the kernel over the entire range of our 1d input data set
// using the maximum number of work group items for this device
//
global = ((count+local-1)/local)*local;           ↗ MINIMO MULTIPLO DELLA LOCAL SIZE CHE È > DI N COUNT
err = clEnqueueNDRangeKernel(commands, kernel, 1, NULL, &global, &local, 0, NULL, NULL);
if (err != CL_SUCCESS) {
    printf("Error: Failed to execute kernel!\n");      ↗ MONODIMENSIONALE
    return EXIT_FAILURE;
}

// Wait for the command commands to get serviced before reading back results
//
// clFinish(commands); → SI PUÒ METTERE

// Read back the results from the device to verify the output
//
err = clEnqueueReadBuffer(commands, output, CL_TRUE, 0, sizeof(float) * count, result);
if (err != CL_SUCCESS) {
    printf("Error: Failed to read output array! %d\n", err);
    return EXIT_FAILURE;
}

```

byte da leggere
↓
risultato sul dispositivo

```
// Validate our results
//
correct = 0;
for (i = 0; i < count; i++)
    if(results[i] == data[i] * data[i])
        correct++;

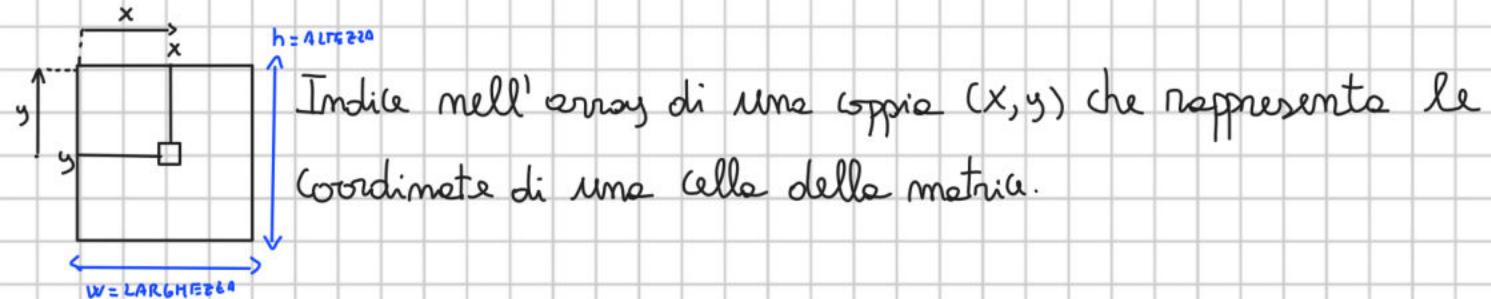
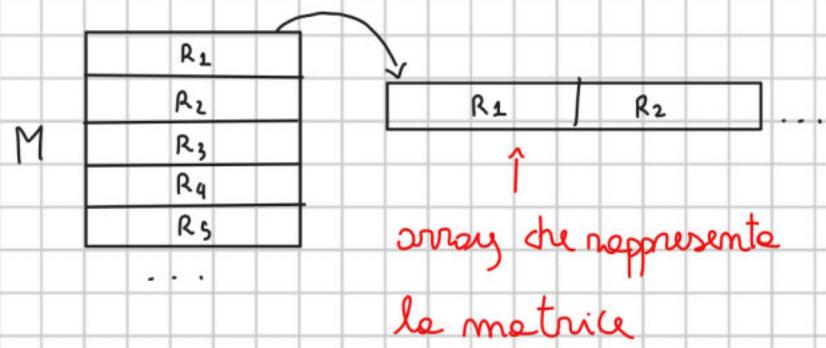
// Print a brief summary detailing the results
//
printf("Computed '%d/%d' correct values!\n", correct, count);

// Shutdown and cleanup
//
clReleaseMemObject(input);
clReleaseMemObject(output);
clReleaseProgram(program);
clReleaseKernel(kernel);
clReleaseCommandQueue(commands);
clReleaseContext(context);
free(data);
free(results);

// Sayonara...
return 0;
}
```

→ RILASCIO RISORSE

Rappresentazione row-major di matrici



$$\text{Index} = \text{IDX}(x, y) = y * w + x$$

Assumiamo che la matrice rappresenti un'immagine a toni di grigio. Ogni cella di tipo `unsigned char` codifica un tono di grigio, dove 0=NERO e 255=BIANCO. Valori intermedi sono toni di grigio.

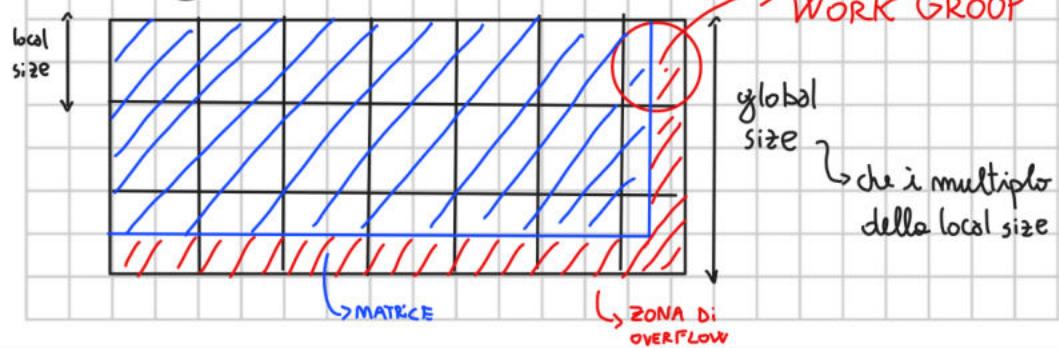
Esempio di applicazione OpenCL per scorrere un'immagine, dimensione $=$
do il valore dei toni di grigio.

$$\text{Output}[\text{IDX}(X, y, w)] = \text{Input}[\text{IDX}(X, y, w)] / 2$$

\uparrow \uparrow

Imagine output imagine input

ND Range bidimensionale



DARKEN

```
#define LOCAL_SIZE 8
#define KERNEL_NAME "darker"

// -----
// darken
// -----
// data-parallel GPU version

void darken(unsigned char* I, unsigned char* O, int h, int w,
            clut_device* dev, double* td) {

    int      err;          // error code
    cl_kernel kernel;      // execution kernel
    cl_mem    din;          // input matrix on device
    cl_mem    dout;          // output matrix on device
    cl_event  evt;          // performance measurement event

    // create the compute kernel
    kernel = clCreateKernel(dev->program, KERNEL_NAME, &err);
    clut_check_err(err, "failed to create kernel");

    // allocate input matrix on device as a copy of input matrix on host
    din = clCreateBuffer(dev->context,
                         CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                         h*w*sizeof(unsigned char), I, NULL);
    if (!din) clut_panic("failed to allocate input matrix on device memory");

    // allocate output matrix on device
    dout = clCreateBuffer(dev->context,
                          CL_MEM_WRITE_ONLY,
                          h*w*sizeof(unsigned char), NULL, NULL);
    if (!dout) clut_panic("failed to allocate output matrix on device memory");

    // set the arguments to our compute kernel
    err = clSetKernelArg(kernel, 0, sizeof(cl_mem), &din);
    err |= clSetKernelArg(kernel, 1, sizeof(cl_mem), &dout);
    err |= clSetKernelArg(kernel, 2, sizeof(int), &h);
    err |= clSetKernelArg(kernel, 3, sizeof(int), &w);
    clut_check_err(err, "failed to set kernel arguments");
```

```

// execute the kernel over the range of our 2D input data set
size_t local_dim[] = { LOCAL_SIZE, LOCAL_SIZE };
size_t global_dim[] = { w, h };
global_dim[0] = ((global_dim[0]+LOCAL_SIZE-1)/LOCAL_SIZE)*LOCAL_SIZE; // round up
global_dim[1] = ((global_dim[1]+LOCAL_SIZE-1)/LOCAL_SIZE)*LOCAL_SIZE; // round up

err = clEnqueueNDRangeKernel(dev->queue, kernel, 2,
                             NULL, global_dim, local_dim, 0, NULL, &evt);
clut_check_err(err, "failed to execute kernel");

// copy result from device to host
err = clEnqueueReadBuffer(dev->queue, dout, CL_TRUE, 0,
                           h*w*sizeof(unsigned char), 0, 0, NULL, NULL);
clut_check_err(err, "failed to read output result");

// return kernel execution time
*td = clut_get_duration(evt);

// cleanup
clReleaseMemObject(din);
clReleaseMemObject(dout);
clReleaseKernel(kernel);
}

```

Kernel di DARKEN

```

#define IDX(x,y,w) ((y)*(w)+(x))

__kernel void darken(__global unsigned char* I,
                     __global unsigned char* O,
                     int h, int w) {

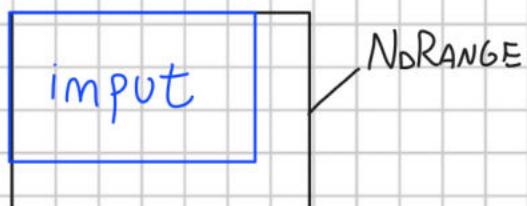
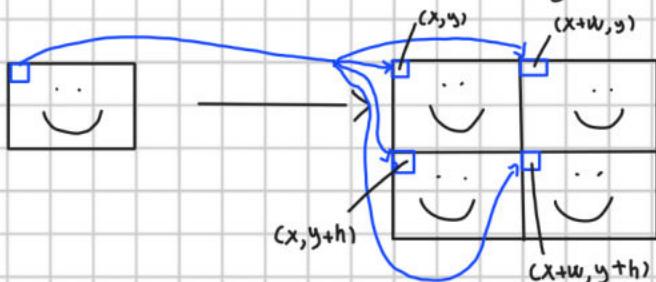
    int x = get_global_id(0);
    int y = get_global_id(1);

    if (x >= w || y >= h) return; // overflow nell'NDRANGE

    O[IDX(x,y,w)] = I[IDX(x,y,w)] / 2;
}

```

Esercizio: Date un'immagine di input di dimensioni $w \times h$, generare un "POSTER" di dimensioni $2w \times 2h$ ottenuto come segue.



Kernel

```
#define IDX(x,y,w) ((y)*(w)+(x))

__kernel void poster(__global unsigned char* I,
                     __global unsigned char* O,
                     int h, int w) {

    int x = get_global_id(0);
    int y = get_global_id(1);

    if (x >= w || y >= h) return;

    unsigned char pixel = I[IDX(x,y,w)];

    O[IDX(x,y,2*w)] = pixel;
    O[IDX(x+w,y,2*w)] = pixel;
    O[IDX(x,y+h,2*w)] = pixel;
    O[IDX(x+w,y+h,2*w)] = pixel;
}
```

POSTER.C

```
void poster(unsigned char* in, int w, int h,
           unsigned char** out, int* ow, int* oh,
           clut_device* dev, double* td) { // dimensioni
    matrice di output
    int err; // error code
    cl_kernel kernel; // execution kernel
    cl_mem din; // input matrix on device
    cl_mem dout; // output matrix on device
    cl_event evt; // performance measurement event
    *ow = 2*w;
    *oh = 2*h;
    *out = malloc(2*w*2*h*sizeof(unsigned char));
    if (*out == NULL) clut_panic("failed to allocate ou| matrix on HOST device memory");
    // create the compute kernel
    kernel = clCreateKernel(dev->program, KERNEL_NAME, &err);
    clut_check_err(err, "failed to create kernel");
    // allocate input matrix on device as a copy of input matrix on host
    din = clCreateBuffer(dev->context,
                         CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                         h*w*sizeof(unsigned char), im, NULL);
    if (!din) clut_panic("failed to allocate input matrix on device memory");
    // allocate output matrix on device
    dout = clCreateBuffer(dev->context,
                          CL_MEM_WRITE_ONLY,
                          4*h*w*sizeof(unsigned char), NULL, NULL);
    if (!dout) clut_panic("failed to allocate output matrix on device memory");
    // set the arguments to our compute kernel
    err = clSetKernelArg(kernel, 0, sizeof(cl_mem), &din);
    err |= clSetKernelArg(kernel, 1, sizeof(cl_mem), &dout);
    err |= clSetKernelArg(kernel, 2, sizeof(int), &h);
    err |= clSetKernelArg(kernel, 3, sizeof(int), &w);
    clut_check_err(err, "failed to set kernel arguments");
    // execute the kernel over the range of our 2D input data set
    size_t local_dim[] = { LOCAL_SIZE, LOCAL_SIZE };
    size_t global_dim[] = { w, h };
    global_dim[0] = ((global_dim[0]+LOCAL_SIZE-1)/LOCAL_SIZE)*LOCAL_SIZE; // round
    up
```

Esercizio: Scrivere codice C con vettorizzazione che verifica se tutti gli elementi di un array sono compresi tra un dato minimo e un dato massimo.

Esempio: min = 2, max = 5

V = {2, 5, 3, 4, 2} SI

V' = {0, 5, 3} NO

Utilizziamo un intrinsec `_mm_cmplt_epi16(a, b)`

```
#include <immintrin.h>
// -----
// inrange
// -----
// versione SSE

int inrange(const short* data, unsigned n, short min, short max) {
    __m128i minv, maxv, datav, res;
    int i;
    minv = _mm_set_epi16(min-1, min-1, min-1, min-1, min-1, min-1, min-1, min-1);
    maxv = _mm_set_epi16(max+1, max+1, max+1, max+1, max+1, max+1, max+1, max+1);

    for (i=0; i+7<n; i+=8) {
        datav = _mm_load_si128((__m128i const*)(data+i));
        res = _mm_cmplt_epi16(datav, minv);
        if (!_mm_test_all_ones(res)) return 0;
        res = _mm_cmplt_epi16(maxv, datav);
        if (!_mm_test_all_ones(res)) return 0;
    }

    for (; i<n; i++)
        if (data[i]<min || data[i]>max) return 0;

    return 1;
}
```

Esercizi

Implicit

A1.scala x A2.scala x

```
1 import scala.language.implicitConversions
2
3 object A2 {
4     implicit def toMyString(x:Int) = MyInt(x)
5 }
6
7 case class MyInt(x:Int) {
8     def minMax(y:Int) = if (x<=y) (x,y) else (y,x)
9 }
10
```

Liste

A1.scala x

```
1 object A1 {
2     def select(cars:List[Car], owners:List[String]) =
3         owners.map(o => cars.filter(_.owner == o).reduce((c1,c2) => if (c1.year > c2.year) c1 else c2)).map(_.model)
4
5 }
```

Scale2x.cl

```
#define IDX(x,y,w) ((y)*(w)+(x))

__kernel void scale2x(__global unsigned char* in,
                      __global unsigned char* out,
                      int w, int h) {

    int x = get_global_id(0);
    int y = get_global_id(1);

    if (x >= w || y >= h) return;

    out[IDX(x, y, w)] = (
        in[IDX(2*x, 2*y, 2*w)] +
        in[IDX(2*x+1, 2*y, 2*w)] +
        in[IDX(2*x, 2*y+1, 2*w)] +
        in[IDX(2*x+1, 2*y+1, 2*w)])
    ) >> 2;
}
```

Scale2x.cl

```
#define LOCAL_SIZE 32
#define KERNEL_NAME "scale2x"

void scale2x(unsigned char* in, int w, int h,
             unsigned char** out, int* ow, int* oh,
             clut_device* dev, double* td) {

    int err; // error code
    cl_kernel kernel; // execution kernel
    cl_mem din; // input matrix on device
    cl_mem dout; // output matrix on device
    cl_event evt; // performance measurement event

    // size of the output image
    *oh = h >> 1;
    *ow = w >> 1;

    // allocate output image on host
    *out = malloc((*ow)*(*oh)*sizeof(unsigned char));
    if (*out == NULL) clut_panic("failed to allocate output matrix on host");

    // create the compute kernel
    kernel = clCreateKernel(dev->program, KERNEL_NAME, &err);
    clut_check_err(err, "failed to create kernel");

    // allocate input matrix on device as a copy of input matrix on host
    din = clCreateBuffer(dev->context,
                         CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                         w*h*sizeof(unsigned char), in, NULL);
    if (!din) clut_panic("failed to allocate input matrix on device memory");

    // allocate output matrix on device
    dout = clCreateBuffer(dev->context,
                          CL_MEM_WRITE_ONLY,
                          (*ow)*( *oh)*sizeof(unsigned char), NULL, NULL);
    if (!dout) clut_panic("failed to allocate output matrix on device memory");

    // set the arguments to our compute kernel
    err = clSetKernelArg(kernel, 0, sizeof(cl_mem), &din);
    err |= clSetKernelArg(kernel, 1, sizeof(cl_mem), &dout);
    err |= clSetKernelArg(kernel, 2, sizeof(int), ow);
    err |= clSetKernelArg(kernel, 3, sizeof(int), oh);
    clut_check_err(err, "failed to set kernel arguments");

    // execute the kernel over the range of our 2D input data set
    size_t local_dim[] = { LOCAL_SIZE, LOCAL_SIZE };
    size_t global_dim[] = { *ow, *oh };
    global_dim[0] = ((global_dim[0]+LOCAL_SIZE-1)/LOCAL_SIZE)*LOCAL_SIZE; // round up
    global_dim[1] = ((global_dim[1]+LOCAL_SIZE-1)/LOCAL_SIZE)*LOCAL_SIZE; // round up

    err = clEnqueueNDRangeKernel(dev->queue, kernel, 2,
                                 NULL, global_dim, local_dim, 0, NULL, &evt);
    clut_check_err(err, "failed to execute kernel");

    // copy result from device to host
    err = clEnqueueReadBuffer(dev->queue, dout, CL_TRUE, 0,
                             (*ow)*( *oh)*sizeof(unsigned char), *out, 0, NULL, NULL);
    clut_check_err(err, "failed to read output result");

    // return kernel execution time
    *td = clut_get_duration(evt);

    // cleanup
    clReleaseMemObject(din);
    clReleaseMemObject(dout);
    clReleaseKernel(kernel);
}
```

```

1 //ELEMENTI DELL ALBERO MODIFICATI DA F
2 sealed abstract class Tree[T] {
3     def map[S](f:T=>S):Tree[S] = this match {
4         case E()          => E()
5         case L(e)         => L(f(e))
6         case N(l, e, r)  => N(l.map(f), f(e), r.map(f))
7     }
8 }
9 case class E[T]() extends Tree[T]
10 case class L[T](e:T) extends Tree[T]
11 case class N[T](l:Tree[T], e:T, r:Tree[T]) extends Tree[T]
12

```

Sse

```

#include "minmax.h"
#include <immintrin.h>

void minmax_sse(const unsigned char* v, int n, unsigned char* pmin, unsigned char* pmax){

    int i;
    unsigned char min[16];
    unsigned char max[16];
    __m128i vmin = _mm_set1_epi8(0xFF);
    __m128i vmax = _mm_set1_epi8(0x00);
    *pmin = 0xFF;
    *pmax = 0x00;

    for (i=0; i + 15 < n; i += 16) {
        __m128i vv = _mm_loadu_si128((const __m128i*)(v+i));
        vmin = _mm_min_epu8(vmin, vv);
        vmax = _mm_max_epu8(vmax, vv);
    }

    for (; i<n; ++i) {
        if (v[i] < *pmin) *pmin = v[i];
        if (v[i] > *pmax) *pmax = v[i];
    }

    _mm_storeu_si128((__m128i*)min, vmin);
    _mm_storeu_si128((__m128i*)max, vmax);

    for (i=0; i<16; ++i) {
        if (min[i] < *pmin) *pmin = min[i];
        if (max[i] > *pmax) *pmax = max[i];
    }
}

```

```

1 // Scrivere un metodo currificato `def buildMatrix(rows:Int, cols:Int)(f:(Int,Int) => Double):Vector[Vector[Double]]` che,
2 // dato un numero di righe `rows` e un numero di colonne `cols` e una funzione `f` da coppie di indici `(i,j)`
3 // a `Double`, restituisce un `Vector[Vector[Double]]` `v` tale che per ogni `i` in `[0,rows-1]` e per ogni `j` in
4 // `[0,cols-1]` si ha `v(i)(j) == f(i,j)`.  

5  

6 // Scrivere la soluzione qui...
7  

8  

9 object E1 {
10     def buildMatrix(rows:Int, cols:Int)(f:(Int,Int) => Double):Vector[Vector[Double]] =
11         (for (i < 0 until rows)
12             yield (for (j < 0 until cols) yield f(i,j))).toVector
13     }.toVector
14 }
15  

16  

17 // Scrivere una versione parallela `fibPar` del metodo `fib` definito nel file `Fib.scala` usando
18 // fork-join in Scala mediante il costrutto `par`.
19  

20 /*programma di partenza
21 * object Fib {
22     def fib(a:Int, b:Int)(n:Int):Long =
23         if (n < 2) a
24         else if (n == 2) b
25         else fib(a,b)(n-1) + fib(a,b)(n-2)
26     }*/  

27  

28 object E2 {
29     lazy val NUM_CORES = Runtime.getRuntime().availableProcessors()
30     def fibPar(a:Int, b:Int)(n:Int, t:Int = NUM_CORES):Long =
31         if (n < 3 || t < 2) Fib.fib(a,b)(n)
32         else {
33             val (l,r) = Par.par(fibPar(a,b)(n-1, t/2))(fibPar(a,b)(n-2, t/2))
34             l + r
35         }
36     }  

37  

38 #include "e3.h"
39 #include <immintrin.h>
40  

41 // -----
42 // matmul: prodotto di matrici
43 // -----
44 // SSE version
45  

46 static void add_prod(const short* src, short* dst, short x, int n) {
47     int j;
48     m128i vx=_mm_set1_epi16(x); //alloco un vettore con tutti i byte impostati ad x
49     for (j=0; j+7<n; j+=8){ //avanzo di 8 perchè sono short (short=16bit -> 128/16=8)
50         m128i vsrc=_mm_loadu_si128((const __m128i*)(src+j)); //carico 128 bit a partire da src+j
51         m128i vdst=_mm_loadu_si128((const __m128i*)(dst+j)); //carico 128 bit a partire da dst+j
52         m128i vmul = _mm_mullo_epi16(vx, vsrc); //moltiplico i primi 128 bit di src per x (di 16 in 16)
53         vdst = _mm_add_epi16(vdst, vmul); //sommo i primi 128 bit di vmul ai primi 128 di vdst
54         _mm_storeu_si128((__m128i*)(dst+j), vdst); //copio in memoria a partire da dst+j i 128 bit appena calcolati
55     }
56  

57     for (; j<n; ++j) dst[j] += x * src[j]; //continuo bit a bit se la lunghezza del vettore non è multiplo di 8
58 }
59  

60 void matmul(const short** a, const short** b, short** c, int n) {
61     int i, j, k;
62     for (i=0; i<n; ++i)
63         for (j=0; j<n; ++j) c[i][j] = 0;
64     for (i=0; i<n; ++i)
65         for (k=0; k<n; ++k)
66             add_prod(b[k], c[i], a[i][k], n);
67 }
68  

69 // -----
70 // matmul_seq
71 // -----
72 // sequential version
73  

74 static void add_prod_seq(const short* src, short* dst, short x, int n) {
75     int j;
76     for (j=0; j<n; ++j) dst[j] += x * src[j];
77 }

```

```

1 #include "e4.h"
2 #include <immintrin.h>
3
4 // -----
5 // count_occ: contare occorrenze di x in un vettore
6 // -----
7 // SSE version
8
9 int count_occ(const char* v, int n, char x) {
10    int i,j,cnt = 0;
11    _m128i vx=_mm_set1_epi8(x); //alloco un vettore con tutti i byte impostati ad x
12    _m128i vcnt=_mm_setzero_si128(); //alloco un vettore con tutti i byte impostati a 0
13    char buf[16]; //alloco un vettore di 16 char (128 bit)
14    for (i=0; i+15<n; i+=16){
15        _m128i vv=_mm_loadu_si128((const _m128i*)(v+i)); //carico 128 bit a partire da v+i
16        _m128i vres=_mm_cmpeq_epi8(vv,vx); //confronto byte a byte vv e vx e metto nel byte corrispondente di vres
17                                         // -1 se i due byte sono uguali, 0 altrimenti
18        vcnt=_mm_add_epi8(vres,vcnt); //sommo i 128 bit appena calcolati al vettore vcnt
19        if(i%2048==0){ //per evitare di andare in overflow
20            _mm_storeu_si128(( _m128i*)buf, vcnt); //mi sposto il risultato corrente di vcnt in buf
21            for(j=0; j<16; j++) cnt-=buf[j]; //sommo byte a byte i valori in vcnt (sottraggo perchè in vcnt i valori
22                                         //saranno negativi e in questo modo li sommo)
23            vcnt=_mm_setzero_si128(); //azzero vcnt
24        }
25    }
26
27    for ( ; i<n; ++i) cnt += v[i] == x; //continuo bit a bit se la lunghezza del vettore non è multiplo di 8
28    _mm_storeu_si128(( _m128i*)buf, vcnt); //mi sposto il risultato corrente di vcnt in buf
29    for(j=0; j<16; j++) cnt-=buf[j]; //aggiorno il contatore sommando gli ultimi elementi
30    return cnt;
31}
32
33
34 // -----
35 // count_occ_seq
36 // -----
37 // sequential version
38
39 int count_occ_seq(const char* v, int n, char x) {
40    int i, cnt = 0;
41    for (i=0; i<n; ++i) cnt += v[i] == x;
42    return cnt;
43}

```

```

1 #include "e3.h"
2 #include <immintrin.h>
3
4 // -----
5 // check_rows: se somma elementi ogni riga=0, return 1, altrimenti 0
6 // -----
7 // SSE/AVX version
8
9 int check_rows(const char* m, int n) {
10    int i, j, x;
11    char buf[16];
12    int cnt=0;
13    for (i=0; i<n; i++) {
14        _m128i vsum=_mm_setzero_si128(); //inizializzo vettore somma a zero
15        for (j=0; j+15<n; j+=16){
16            _m128i vm=_mm_loadu_si128((const _m128i*)(m+i*n+j)); //carico i primi 128 bit a partire da m[i*n+j]
17            vsum=_mm_add_epi8(vm,vsum); //sommo al vettore vsum i 128 bit appena caricati
18        }
19        _mm_storeu_si128(( _m128i*)buf, vsum); //carico nel buffer il vettore somma
20        for(x=0; x<16; x++) cnt+=buf[x]; //calcolo la somma di tutti gli elementi di vsum
21        for ( ; j<n; ++j){ //se la lunghezza della riga non è multiplo di 128
22            cnt += m[i*n+j]; //sommo byte a byte gli elementi rimanenti della riga i-esima
23        }
24        if(cnt!=0) return 0; //se la somma della riga corrente è diversa da 0 -> return false
25    }
26    return 1;
27}
28
29
30 // -----
31 // check_rows_seq
32 // -----
33 // sequential version
34
35 int check_rows_seq(const char* m, int n) {
36    int i, j;
37    for (i=0; i<n; ++i) {
38        char sum = 0;
39        for (j=0; j<n; ++j) sum += m[i*n+j];
40        if (sum != 0) return 0;
41    }
42    return 1;
43}

```

```

1 // Scrivere un metodo `def subList(l>List[T]):Boolean` applicabile su un oggetto `List[T]` `s` che restituisce `true`
2 // se e solo se tutti gli elementi di `l` appaiono anche in `s` nello stesso ordine, anche non consecutivamente.
3
4 import scala.language.implicitConversions
5
6 object E1 {
7   implicit def seq2MySeq[T](s>List[T]):MyList[T] = new MyList(s)
8 }
9
10 import E1._
11
12 class MyList[T](s>List[T]) {
13   def subList(l>List[T]):Boolean = {
14     val intersezione=s.intersect(l)
15     if(intersezione==l) true
16     else false
17   }
18 }

```

```

1 // Il metodo fornito `def sum3(s:Set[Int]):Set[Set[Int]]` calcola i sottoinsiemi
2 // di cardinalità 3 dell'insieme in input `s` tali che la somma dei loro elementi
3 // sia pari a 0. La soluzione sequenziale proposta ha complessità quadratica
4 // (https://en.wikipedia.org/wiki/3SUM). Si richiede di scrivere una versione
5 // parallela `def sum3PAR(s:Set[Int]):Set[Set[Int]]` che usa fork-join in Scala
6 // mediante il costrutto `par`. La soluzione deve usare un numero di thread pari
7 // al numero di core logici disponibili.
8
9 object E2 {
10   def sum3(s:Set[Int]):Set[Set[Int]] = {
11     val v = s.toVector
12     val ris = for {
13       i <- 0 until v.size
14       j <- i+1 until v.size
15       target = -(v(i)+v(j))
16       if (target != v(i) && target != v(j))
17       if (s.contains(target))
18     } yield Set( v(i), v(j), target )
19     ris.toSet
20   }
21
22 lazy val NUM_CORES = Runtime.getRuntime().availableProcessors()
23
24 def sum3Par(s:Set[Int]):Set[Set[Int]] = {
25   val v = s.toVector
26   def sum3MinMax(min:Int,max:Int):Set[Set[Int]] = {
27     val ris = for {
28       i <- min until max
29       j <- i+1 until v.size
30       target = -(v(i)+v(j))
31       if (target != v(i) && target != v(j))
32       if (s.contains(target))
33     } yield Set( v(i), v(j), target )
34     ris.toSet
35   }
36   def aux(min:Int, max:Int, t:Int):Set[Set[Int]] = {
37     if (t < 2 || min >= max)
38       sum3MinMax(min, max)
39     else {
40       val mid = (min+max)/3
41       val (ris1, ris2) = Par.par { aux(min, mid, t/2) } { aux(mid, max, t/2) }
42       ris1.union(ris2)
43     }
44   }
45   aux(0, s.size, NUM_CORES)
46 }

```

```

1 // Definire un metodo `isPrime` applicabile su interi `Int` che restituisce `true` se il numero
2 // è primo, e `false` altrimenti.
3
4 // Scrivere la soluzione qui...
5 import scala.language.implicitConversions
6
7 object E1{
8     implicit def myPrime(i:Int)=myInt(i)
9 }
10
11 case class myInt(i:Int){
12     def isPrime={
13         if(i==0 || i==1 || i<0) false
14         else{
15             val l=List.range(2,i).map(x=>i%x==0).contains(true)
16             !l
17         }
18     }
19 }

```

MyFo4

```

1 // Definire un nuovo costrutto imperativo `myFor` che simula il `for` del C come
2 // nel seguente esempio:
3 // myFor(i=0)(i<10)(i+=1) { ... } equivalente a for(i=0; i<10; i++) { ... } in C.
4 // L'implementazione deve essere ricorsiva.
5
6 // Scrivere la soluzione qui...
7
8 object E2{
9     def myFor(i:Unit)(test: =>Boolean)(incr: =>Unit)(body: =>Unit):Unit={
10         if(!test) ()
11         else{
12             body
13             incr
14             myFor(i)(test)(incr)(body)
15         }
16     }
17 }

```

// Si vuole definire un nuovo costrutto mywhile della forma: mywhile(test, expr) { instr }

// che ripete le istruzioni instr fintantoché test è vero. All'inizio di ciascuna iterazione deve

// essere valutata l'espressione expr e stampato il risultato.

```

6 object A2 {
7     def mywhile(test: =>Boolean, s: =>Any)(body: =>Unit):Unit =
8         if (!test) ()
9         else {
10             println(s)
11             body
12             mywhile(test,s)(body)
13         }
14 }

```

```
1 #include <immintrin.h>
2 #include "inrange.h"
3
4 #define CMIN(a,b) ((a)<(b) ? (a):(b))
5
6 // -----
7 // inrange: dati tre vettori controlla che v1[i]<=v2[i]<=v3[i]
8 // -----
9 // SSE version
10
11 int inrange(const short* min, unsigned minn,
12             const short* v, unsigned n,
13             const short* max, unsigned maxn) {
14
15     int i;
16     __m128i minv, vv, maxv, res, res2;
17
18     n = CMIN(n, CMIN(minn, maxn));
19
20     for (i=0; i+7<n; i+=8) {
21         minv = _mm_loadu_si128((__m128i const*)(min+i));
22         vv   = _mm_loadu_si128((__m128i const*)(v+i));
23         maxv = _mm_loadu_si128((__m128i const*)(max+i));
24         res  = _mm_cmpgt_epi16(vv, minv);
25         res2 = _mm_cmpeq_epi16(vv, minv);
26         res  = _mm_add_epi16(res, res2);
27         if (!_mm_test_all_ones(res)) return 0;
28         res  = _mm_cmpgt_epi16(maxv, vv);
29         res2 = _mm_cmpeq_epi16(maxv, vv);
30         res  = _mm_add_epi16(res, res2);
31         if (!_mm_test_all_ones(res)) return 0;
32     }
33
34     for (; i<n; ++i)
35         if (v[i]<min[i] || v[i]>max[i]) return 0;
36
37     return 1;
38 }
```

```

1 #define IDX(x,y,w) ((y)*(w)+(x))
2
3 _kernel void myaddframe(_global unsigned char* I,
4                         _global unsigned char* O,
5                         int h, int w,
6                         unsigned char frame_grey, int frame_size) {
7
8     int x = get_global_id(0);
9     int y = get_global_id(1);
10
11    if(x>=w+2*frame_size || y>=h+2*frame_size) return;
12
13    if(x<frame_size || x>=w+frame_size || y<frame_size || y>=h+frame_size)
14        O[IDX(x,y,w+2*frame_size)] = frame_grey;
15
16    else{
17        unsigned pixel = I[IDX(x-frame_size,y-frame_size,w)];
18        O[IDX(x,y,w+2*frame_size)] = pixel;
19    }
20}

```

```

#define IDX(x,y,w) ((y)*(w)+(x))
_kernel void poster(_global unsigned char* I,
                    _global unsigned char* O,
                    int h, int w) {
    int x = get_global_id(0);
    int y = get_global_id(1);
    if (x >= w || y >= h) return;
    unsigned pixel = I[IDX(x,y,w)];
    O[IDX(x,y,2*w)] = pixel;
    O[IDX(x+w,y,2*w)] = pixel;
    O[IDX(x,y+h,2*w)] = pixel;
    O[IDX(x+w,y+h,2*w)] = pixel;
}

```

```

1 //Si vuole scrivere un metodo Scala select che, data una lista di automobili, trova per ogni
2 //anno di immatricolazione di una qualche auto della lista il nome del proprietario più giovane
3 //che ha un'auto immatricolata in quell'anno.
4
5 object A1{
6     def select(cars:List[Car])={
7         cars.groupBy(x=>x.year).toList.map(x=>(x._1, x._2.map(x=>x.owner)).reduce((a,b)=> if (a.age<b.age) a else b).name)
8     }
9 }
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42

```

```

1 // Il metodo `isMappedFrom`, applicabile a un vettore l, verifica se un altro vettore m è
2 // ottenibile da l applicando la funzione f a ciascun elemento di l.
3
4 // Scrivere una versione parallela `isMappedFromPar` del metodo `isMappedFrom` usando
5 // un approccio divide et impera basato sul costrutto `par` fornito. Usare un numero di
6 // thread pari al numero di core logici disponibili.
7
8 // Suggerimento: ispirarsi alla soluzione dell'esercizio 38 -
9 // PIsRot (https://season-lab.github.io/PFP/eserc/soluzScala)
10
11 import scala.language.implicitConversions
12
13 class MyVec[T](l:Vector[T]) {
14
15     private def isMappedFromInRange[S](m:Vector[S], min:Int, max:Int, f:T=>S):Boolean =
16         (min until max).forall(i => m(i) == f(l(i)))
17
18     // versione sequenziale
19     def isMappedFrom[S](m:Vector[S], f:T=>S) =
20         if (l.size != m.size) false
21         else isMappedFromInRange(m, 0, l.size, f)
22
23     // versione parallela
24     def isMappedFromPar[S](m:Vector[S], f:T=>S) = {
25         def aux(min:Int, max:Int, t:Int = E2.NUM_CORES):Boolean =
26             if (min >= max) true
27             else if (t < 2) isMappedFromInRange(m, min, max, f)
28             else {
29                 val mid = (min + max) / 2
30                 val (a,b) = Par.par { aux(min, mid, t/2) } { aux(mid, max, t/2) }
31                 a && b
32             }
33             if (l.size != m.size) false
34             else aux(0, l.size)
35     }
36
37
38 object E2 {
39     lazy val NUM_CORES = Runtime.getRuntime().availableProcessors()
40     implicit def vec2MyVec[T](l:Vector[T]) = new MyVec(l)
41 }
42

```

//Si vuole scrivere un metodo Scala minMax che calcola simultaneamente minimo e massimo di
//un albero binario di ricerca e il numero di chiamate ricorsive effettuate. Si ricordi che in un
//albero binario di ricerca la chiave di ogni nodo è maggiore o uguale alle chiavi nel suo
//sottoalbero sinistro e minore o uguale alle chiavi nel suo sottoalbero destro. La soluzione
//deve usare il minor numero possibile di chiamate ricorsive.

```
7 sealed abstract class Tree(){
8     def minMax={
9         val resMin=minimo(this,Integer.MAX_VALUE,0) //resMin=(min, chiamateMin)
10        val resMax=massimo(this,0,0) //resMax=(max, chiamateMax)
11
12        (resMin._1, resMax._1 , resMin._2+resMax._2)
13    }
14
15    //funzione che ritorna il minimo ed il numero di chiamate per trovarlo
16    def minimo(t:Tree, min:Int, i:Int):(Int,Int)=t match{
17        case E()=>(min, i)
18        case T(l,e,r) => minimo(l,e,i+1)
19    }
20
21    //funzione che ritorna il massimo ed il numero di chiamate per trovarlo
22    def massimo(t:Tree, max:Int, i:Int):(Int,Int)=t match{
23        case E()=>(max, i)
24        case T(l,e,r) => massimo(r,e,i+1)
25    }
26
27}
28 case class E() extends Tree()
29 case class T(l:Tree, x:Int, r:Tree) extends Tree()
```

// Un albero binario è di ricerca se l'elemento contenuto in ogni suo nodo v è maggiore
// o uguale all'elemento nella radice del sottoalbero sinistro di v (se non vuoto)
// e minore o uguale all'elemento nella radice del sottoalbero destro di v (se non vuoto).
// Scrivere un metodo `isSearchTree` che, dato un albero binario con elementi interi,
// restituisce true se l'albero è di ricerca, e false altrimenti.

```
7 sealed abstract class BinTree {
8     def isSearchTree:Boolean= this match{
9         case E() => true
10        case T(l,e,r) => (l,r) match{
11            case (E(),E()) => true
12            case (T(l1,e1,r1), E()) => (e1<=e) && l.isSearchTree
13            case (E(), T(l2,e2,r2)) => (e2>=e) && r.isSearchTree
14            case (T(l1,e1,r1), T(l2,e2,r2)) => (e1<=e) && (e2>=e) && l.isSearchTree && r.isSearchTree
15        }
16    }
17
18    // albero non vuoto
19    case class T(l:BinTree, e:Int, r:BinTree) extends BinTree
20
21    // albero vuoto
22    case class E() extends BinTree
```

```

import scala.language.implicitConversions

object E3 {
    implicit def seq2MySeq[T](s>List[T]):MyList[T] = new MyList(s)
}

class MyList[T](s>List[T]) {
    import E3._
    def subList(l>List[T]):Boolean = {
        if (l.isEmpty) true
        else if (s.isEmpty) false
        else if (s.head == l.head) s.tail.subList(l.tail)
        else s.tail.subList(l)
    }
}

```

```

object E4 {
    def piuGiovane(s:Vector[Studente], e:Vector[Eta]):Option[String] = {
        if (s.isEmpty) None
        else {
            val m:Map[Int,Vector[Eta]] = e.groupBy(_.id)
            Some(s.reduce((x,y) => if (m(x.id)(0).eta < m(y.id)(0).eta) x else y).nome)
        }
    }
}

```

```

1 #include <immintrin.h>
2 #include "e2.h"
3
4
5 // -----
6 // count_occ
7 // -----
8 // SSE version
9
10 int count_occ(const char* v, int n, char x) {
11     int i, cnt = 0;
12     unsigned char cntv[16];
13     __m128i xv, xv, res, vone, vcnt;
14     xv = _mm_set_epi8(x);
15     vone = _mm_set_epi8(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1);
16     vcnt = _mm_setzero_si128();
17     for (i=0; i + 15 < n; i+=16) {
18         if (i % 256 == 0) {
19             _mm_store_si128((__m128i*)cntv, vcnt);
20             for (int j=0; j<16; j++) cnt += cntv[j];
21             vcnt = _mm_setzero_si128();
22         }
23         xv = _mm_loadu_si128((const __m128i*)(v + i));
24         res = _mm_cmpeq_epi8(xv, vv);
25         res = _mm_add_epi8(res, vone);
26         vcnt = _mm_add_epi8(vcnt, res);
27     }
28     _mm_store_si128((__m128i*)cntv, vcnt);
29     for (int j=0; j<16; j++) cnt += cntv[j];
30     for (; i<n; ++i) cnt += v[i] == x;
31     return n - cnt;
32 }

33
34 // -----
35 // count_occ_seq
36 // -----
37 // sequential version
38
39 int count_occ_seq(const char* v, int n, char x) {
40     int i, cnt = 0;
41     for (i=0; i<n; ++i) cnt += v[i] == x;
42     return cnt;
43 }

```

Import scala.language.implicitConversion

ContainsSlice