Algoritmi di Fibonacci

Luca Tagliavini

February 2021

Contents

1 Definizone di fibonacci

$$F_1 = 1$$
$$F_2 = 1$$
$$F_n = F_{n-1} + F_{n-2} \quad \forall n > 2$$

Esisterebbe una formula matematica che sfrutta *la sezione aurea* per calcolare l'n-esimo numero di fibonacci tramite semplici moltiplicaizoni ed elevamenti a potenza.
Tuttavia, il valore della sezione aurea e' un numero irrazionale e come tale non puo' essere rappresentato con precisione da un calcolatore (tramite una frazione di interi). Percio' su particolari valori, il risultato ottenuto non sarebbe quello corretto.

2 Implementazioni

2.1 Fibonacci nel modo matematico

```
double fib(int n) {
  return 1.0/Math.sqrt(5.0)*
  (Math.pow((1.0+Math.sqrt(5.0))/2.0, n) -
    Math.pow((1.0-Math.sqrt(5.0))/2.0, n));
}
```

E' la soluzione piu' veloce a livello di performance in quanto usando solo funzioni elementari come moltiplicazioni ed elevamento a potenza abbiamo un tempo di esecuzione *lineare*. Tuttavia e' anche un metodo che arrivando a valori attendo al 70-72 si iniziano ad ottenere errori apprezzabili.

> Questo approccio sara' anche piu' veloce dell'algoritmo che alla fine riterremo migliore.

2.2 Fibonacci seguendo la definizone ricorsiva

```
algoritmo fib(int n) -> int
  if n == 1 or n == 2 then
    return 1
  else
    return fib(n-1) + fib(n-2)
end
```

Tempo esponenziale, lavoro sprecato calcolando numerosissime volte $fib(1)$ o $fib(2)$.

2.3 Fibonacci iterativo

```
fib( int n ) {
  BigInteger prev = new BigInteger("1"), last=new BigInteger("1"), help;
  for (int i=3; i<=n; i++) {
    help = last;
    last = prev.add(last);
    prev = help;
  }
  return last;
}
```

Questa e' la migliore versione iterativa utilizzando solo tre variabili. L'algoritmo e' nettamente piu' veloce di quello ricorsivo in quanto rimuove le computazioni inutili di valori gia' calcolati, tenendo traccia dei valori precedenti in due variabili ausiliarie.

2.4 Fibonacci con le matrici

$A = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$
Per ogni $n \geq 2$ si ha:
$A^{n-1} = \begin{pmatrix} F_n & F_{n-1} \\ F_{n-1} & F_{n-2} \end{pmatrix}$
dove F_n e' il valore che volevamo ottenere e quindi useremo come risultato nell'algoritmo:

```
algoritmo fib(int n) -> int
```

$A = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$

```
  for i := 2 to n-1 do
```

$A = A \cdot \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$

```
  end
  return A[1][1]
end
```

Tuttavia questo algoritmo e' veloce tanto quanto (peggio in realta' poiche' esegue moltiplicazioni tra matrici) la versione iterativa.
Fortunatamente c'e' una tecnica furba per svolgere l'elevazione a potenza che ci e' necessaria grazie alla quale si riesce a ottenere un $O(\log n)$.

3 Stima del tempo d'esecuzione

Il calcolo del tempo d'esecuzione si svolge tramite la "conta" del numero di operazioni elementari che i nostri algoritmi svolgono. Le stime non hanno lo scopo

di essere precise nell'unita', ma vogliono arrotondare all'ordine di grandezza del tempo richiesto.

Ad esempio, se avessimo 5 operazioni di base (qualche if, qualche somma, qualche compare) arrotondiamo il numero di operazioni svolte a $O(1)$. Il numero di operazioni base non ci interessa particolarmente, quanto piuttosto come le operazioni e sottochiamate aumentano in base all'input. Nel nostro caso d'esempio di fibonacci ricorsivo i check non variano in base all'input, quindi possiamo approssimare con $O(n)$, poiche' la maggior parte del tempo viene spesa a svolgere chiamate ricorsive, non operazion di base.