Algoritmi per Fibonacci e conigli

Leonardo da Pisa (Fibonacci) si interessò di molte cose, tra cui il seguente problema di dinamica delle popolazioni:

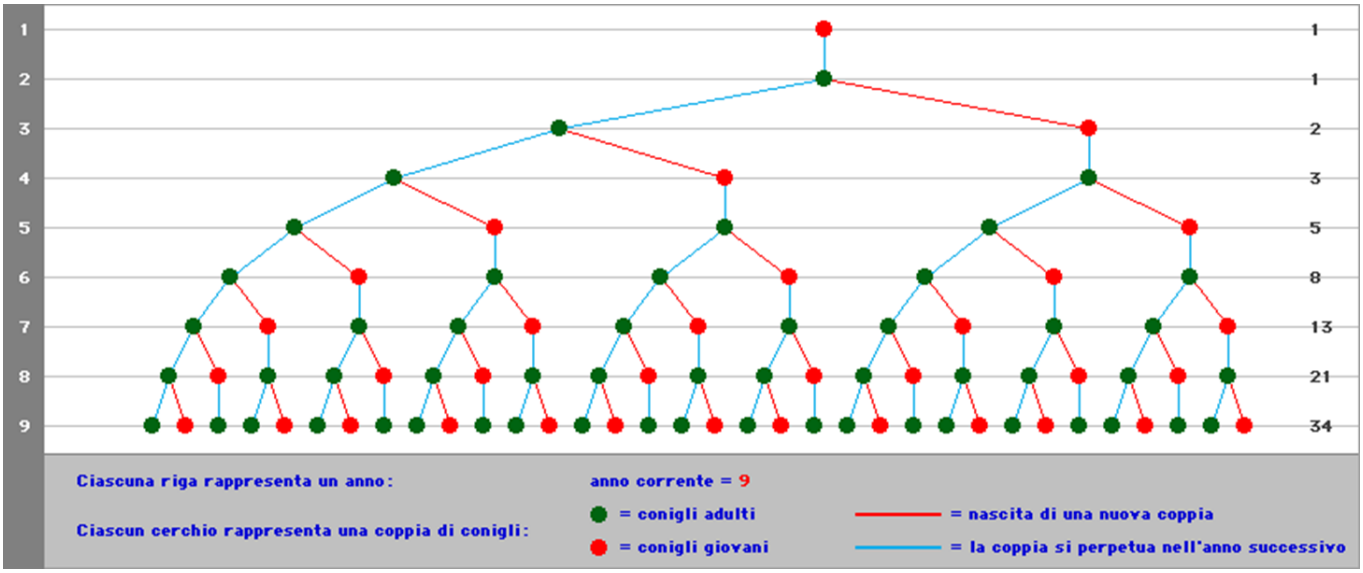
> **Quanto velocemente si espanderebbe una popolazione di conigli sotto appropriate condizioni?**

In particolare partiamo da una coppia di conigli in un'isola deserta, quante coppie si avrebbero nell'anno n?

Imponiamo delle regole di riproduzione:

> **Regole per la riproduzione:**
>
> Una coppia di conigli concepisce due coniglietti di sesso diverso ogni anno, i quali forniranno una nuova coppia.
>
> La gestazione dura un anno.
>
> I conigli cominciano a riprodursi soltanto al secondo anno dopo la loro nascita.
>
> I conigli sono immortali.

La riproduzione dei coniglio può essere descritta in un albero come segue:



Nell’anno n, ci sono tutte le coppie dell’anno precedente, e una nuova coppia di conigli per ogni coppia presente due anni prima.

Possiamo indicare con F_n il numero di coppie dell'anno n, quindi abbiamo la seguente **relazione di ricorrenza**:

$$F_n = \begin{cases} F_{n-1} + F_{n-2} \quad se \;\; n \geq 3 \\ 1 \quad se \;\; n = 1, 2 \end{cases}$$

Praticamente tramite il problema dei conigli siamo riusciti a ricondurci alla Sequenza di Fibonacci.

Adesso che sappiamo come funziona la sequenza di Fibonacci possiamo calcolare F_n:

Algoritmo Fibonacci 1

Possiamo usare una funzione matematica per calcolare direttamente i numeri di Fibonacci, infatti abbiamo che:

$$F_n = \frac{1}{\sqrt{5}}(\phi^n - \neg\phi^n)$$

Dove:

$$\phi = \frac{1+\sqrt{5}}{2} \approx +1,618 \qquad \neg\phi = \frac{1-\sqrt{5}}{2} \approx -0,618$$

Adesso l'algoritmo scritto in pseudocodice è il seguente:

```
Algoritmo fibonacci1(intero n) ---> intero:   //Funzione da intero a intero
        return 1/sqrt(5)(\phi^n-\neg\phi^n)       //Corpo della funzione
```

Il problema principale con questo pseudocodice è che si basa molto sull'accuratezza di ϕ e $\neg\phi$ e man mano che i numeri diventano sempre più grandi o aumentiamo l'accuratezza delle cifre decimali oppure l'algoritmo è sbagliato.

Ad esempio con 3 cifre decimali di accuratezza (Come abbiamo fatto sopra) abbiamo:

n	fibonacci1(n)	Arrotondamento	F_n
3	1.99992	2	2
16	986.698	987	987
18	2583.1	2583	2584

Come fa notare la tabella già al diciottesimo numero abbiamo un errore e quindi non è un algoritmo corretto.

Algoritmo Fibonacci 2

Possiamo utilizzare un approccio tramite una definizione ricorsiva e tramite la Tecnica del divide et impera

```
Algoritmo fibonacci2(intero n) ---> intero:
        if(n<=2) then return 1
        else return fibonacci2(n-1) + fibonacci2(n-2)   //Ricorsione
```

Partiamo dicendo che questo algoritmo è corretto.

Cerchiamo di analizzarne il tempo di esecuzione, siccome sono le prime lezioni, usiamo un modello di calcolo abbastanza semplice da calcolare anche se non estremamente preciso.

Facciamo che ogni linea di codice costa un'unità di tempo. E' abbastanza comodo come modello perché usa una misura indipendente dalla piattaforma utilizzata.

Quindi quante linee di codice eseguiamo:

- Se $n \leq 2$: 1 linea di codice.
- Se $n = 3$: Quattro linee di codice, perché teniamo conto della ricorsione.

Adesso la nostra equazione di ricorrenza è:

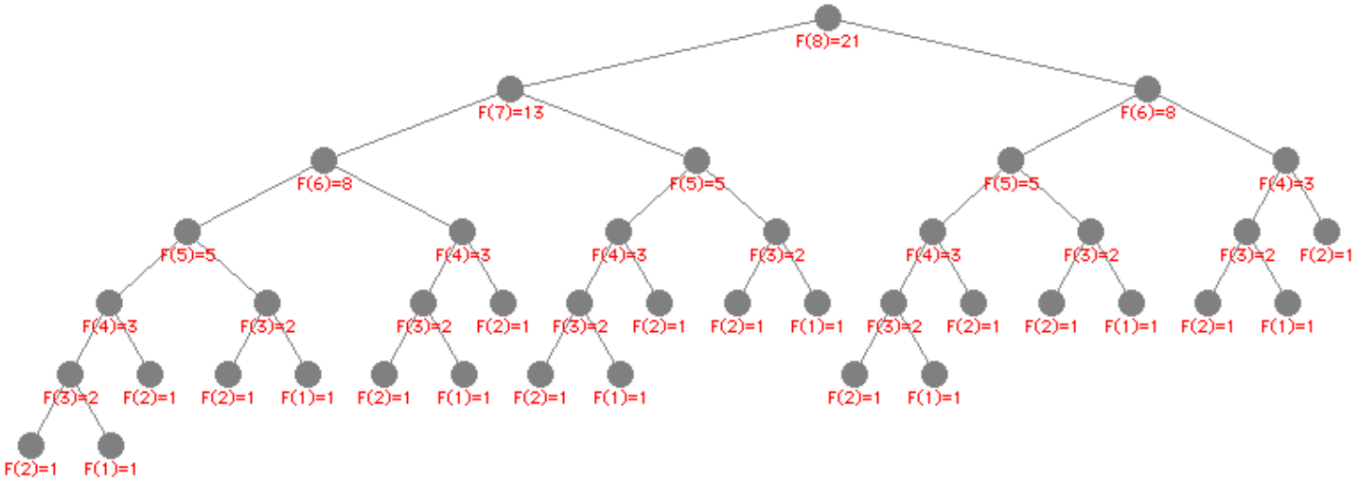
$T(n)$ = numero di linee di codice eseguite, nel caso peggiore, dall'algoritmo su un input n.

In ogni chiamata si eseguono due linee di codice oltre a quelle eseguite nelle chiamate ricorsive.

$$T(n) = 2 + T(n-1) + T(n-2) \qquad T(1) = T(2) = 1$$

In generale, il tempo richiesto da un algoritmo ricorsivo è pari al tempo speso all'interno della chiamata più il tempo speso nelle chiamate ricorsive.
Possiamo rappresentare questo algoritmo ricorsivo tramite il **Metodo dell'albero della ricorsione**

I nodi rappresentano alle chiamate ricorsive, i figli dei noti sono le sottochiamate.



Per calcolare $T(n)$ possiamo assegnare ad ogni nodo un valore corrispondente al numero di linee di codice che vengono eseguite.

- I nodi interni hanno valore 2
- Le foglie hanno valore 1

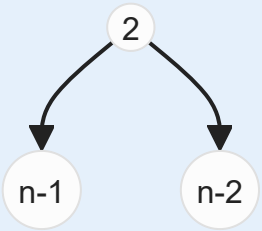
Contiamo quindi il valore totale, ma per fare questo calcolo dobbiamo introdurre dei lemmi:

- **Lemma 1:** Il numero di foglie dell'albero della ricorsione di fibonacci2(n) è pari a F_n

Dimostrazione per induzione su n

Passo Base: n=1,2

In questo caso il numero di foglie sarà uguale ad #foglie $= 1 =$F_1$ = F_2
Passo Induttivo: n>2



Possiamo considerare grazie all'ipotesi induttiva che il numero delle foglie sui due rami, chiamiamoli T_1 e T_2, siano uguali a #foglie $T = F_{n-1} + F_{n-2} = F_n$

- **Lemma 2:** Il numero di nodi interni di un albero in cui ogni nodo interno ha due figli è pari al numero di foglie -1

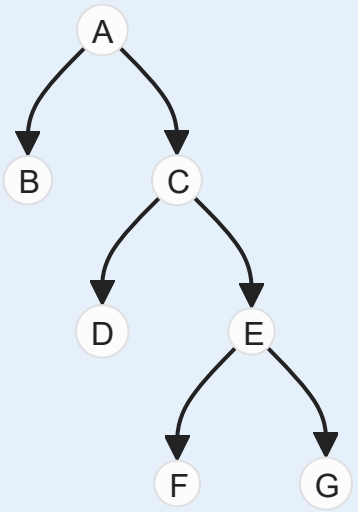
Dimostrazione per induzione su n, considerando f = numero di foglie, i = Nodi interni.

Passo Base: n <= 2

1

In questo caso: $i = 0$ e $f = 1$, il che è verificato.

Passo Induttivo: n>2



Chiamiamo questo albero T e tagliamo la coppia di foglie più "profonde", creando così un albero chiamato T' con f' foglie e i' nodi interni.

Per costruzione: $i' = i - 1$ e $f' = f - 1$
Per ipotesi induttiva: $i' = f' - 1$

Quindi: $i - 1 = f - 1 - 1$, cioè $i = f - 1$

Quindi con Fibonacci2 eseguiamo:

$$F_n + 2(F_n - 1) = 3F_n - 2$$

Cioè la prima linea di codice addizionata alle due linee di codice della sottochiamata.

Adesso Fibonacci2 è lento infatti:

$$T(n) \approx F_n \approx \phi^n$$

E' molto lento come algoritmo perché il numero di linee di codice eseguite cresce in maniera esponenziale, basti pensare che per computare $n = 100$ ci mettiamo 8000 anni.

Algoritmo Fibonacci 3

Il motivo per cui Fibonacci2 è lento è perché continua a ricalcolare sempre la soluzione dello stesso sottoproblema.
Nell'esempio di sopra Fibonacci(4) viene calcolato sei volte.

Possiamo introdurre un nuovo algoritmo tramite la tecnica della programmazione dinamicaL'idea è quella di creare un array dove memorizzare le soluzioni dei sottoproblemi nell'ennesima posizione troviamo il valore cercato.

```
Algoritmo fibonacci3(intero n) --> intero
        Sia Fib un array di n interi
        Fib[1] <-- 1; Fib[2] <-- 1
        for i=3 to n do
                Fib[i] <-- Fib[i-1]+Fib[i-2]
        return Fin[n]
```

- Linee 1,2,5 eseguite una sola volta.
- Linea 3 eseguita $\leq n$ volte.
- Linea 4 eseguita $\leq n$ volte.

$T(n) \leq n + n + 3 = 2n + 3$
$T(45) \leq 93$

Di conseguenza questo algoritmo è 38 milioni di volte più veloce dell'algoritmo fibonacci2.
Proprio perché fibonacci3 impiega tempo proporzionale a n invece di esponenziale in n come fibonacci2.

Fino ad adesso ci siamo interessati solo del tempo d'esecuzione, ma non è la sola risorsa di calcolo importante, infatti la **quantità di memoria** è altrettanto importante, perché se il nostro algoritmo occupa più memoria di quella disponibile allora non otterremo mai la soluzione.
Per questo motivo introduciamo Fibonacci4.

Algoritmo Fibonacci 4

L'algoritmo precedente usa un array di dimensione fissa, cioè n, ma pensandoci a noi non ci serve tutto l'array, ma solo gli ultimi due elementi.

Possiamo ridurre lo spazio a poche variabili.

```
Algoritmo fibonacci4(intero n) --> intero
        a <-- 1, b <--1
        for i = 3 to n do
                c <-- a+b
                a <-- b
                b <--c
        return c
```

Adesso $T(n) \leq 4n + 2$, quindi è più veloce o lento di fibonacci3

Introduciamo quindi la **notazione asintotica**, semplicemente noi vogliamo calcolare $T(n)$ in modo qualitativo, quindi va bene perdere un po' in precisione ma guadagnando in semplicità.

Quindi possiamo ignorare:

- Costanti moltiplicative.
- Termini di ordine inferiore.

Adesso ci poniamo la domanda:

> **è sensato misurare la complessità di un algoritmo contando il numero di linee di codice eseguite?**

Se andiamo a capo più spesso, aumenteranno le linee di codice sorgente, ma certo non il tempo richiesto dall'esecuzione del programma.
Per rispondere alla domanda dobbiamo prima svolgere altre lezioni, nel frattempo:

> **Diremo che $f(n) = O(g(n))$ se $f(n) \leq c * g(n)$ per qualche costante c ed n abbastanza grande.**

> **Possiamo calcolare F_n in tempo inferiore a $O(n)$?**

Si, con la particolarità che i prossimi algoritmo fanno uso di particolari proprietà matematiche.

Algoritmo Fibonacci 5

Prima di scrivere lo pseudocodice dobbiamo dimostrare questa proprietà delle matrici:

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n = \begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix}$$

Dimostrazione induzione su n

Fissiamo per convenzione $F_0 = 0$ (Da notare che $F_2 = F_1 + F_0$)

Passo base, $n = 1$:

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^1 = \begin{bmatrix} F_2 & F_1 \\ F_1 & F_0 \end{bmatrix}$$

Passo induttivo, $n > 1$:

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1} * \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

Adesso questo per ipotesi induttiva è uguale a:

$$\begin{bmatrix} F_n & F_{n-1} \\ F_{n-1} & F_{n-2} \end{bmatrix} * \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

Per il prodotto per matrici (Righe per colonne) abbiamo:

$$\begin{bmatrix} F_n + F_{n-1} & F_n \\ F_{n-1} + Fn - 2 & F_{n-1} \end{bmatrix} = \begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix}$$

```
Algoritmo fibonacci5(intero n) --> intero n
        M <-- (1 0)    //Non sapevo come rappresentare la matrice, quindi lo
        M <-- (0 1)    //Messa su due linee diverse, in realtà e solo una
        for i=1 to n-1 do
                M <-- M * (1 1)
                M <-- M * (1 0)
        return M[0][0]
```

Adesso questo algoritmo ha sempre complessità $O(n)$, difatti usato per far capire che anche se abbiamo le proprietà giuste le dobbiamo comunque usare in maniera corretta.

Algoritmo Fibonacci 6

Possiamo calcolare la n-esima potenza elevando al quadrato la ($\lfloor n/2 \rfloor$)-esima potenza e se n è dispari eseguiamo un ulteriore moltiplicazione.

> **Per esempio:**
>
> Invece di eseguire $3^8 = 3 * 3 * 3 * 3 * 3 * 3 * 3 * 3$, possiamo fare:
>
> $3^2 = 9 \quad 3^4 = (9)^2 = 81 \quad 3^8 = (81)^2 = 6561$
>
> Di conseguenza abbiamo eseguito solo tre prodotti invece di sette.

Di seguito il codice scritto in python:

```python
def ProdottoMatrici(A, B):
    # Assumiamo che A e B siano matrici quadrate di dimensione 2x2
    return [
        [
            A[0][0] * B[0][0] + A[0][1] * B[1][0],
            A[0][0] * B[0][1] + A[0][1] * B[1][1]
        ],
        [
            A[1][0] * B[0][0] + A[1][1] * B[1][0],
            A[1][0] * B[0][1] + A[1][1] * B[1][1]
        ]
    ]

def PotenzaMatrici(M, n):
    if n == 0:
        return [[1, 0], [0, 1]] #Ritorna la matrice identità
```

```
    else:
        M = PotenzaMatrici(M, n//2)
        M = ProdottoMatrici(M, M)
        if n % 2 == 1:
            M = ProdottoMatrici(M, [[1, 1], [1, 0]])
        return M


def fibonacci6(MatriceFibonacci, n):
    if n == 0:
        return [[1, 0], [0, 1]] #Ritorna la matrice identità
    else:
        M = PotenzaMatrici(MatriceFibonacci, n//2)
        M = ProdottoMatrici(M, M)
        if n % 2 == 1:
            M = ProdottoMatrici(M, [[1, 1], [1, 0]])
        return M

MatriceFibonacci = [[1, 1], [1, 0]]
n = int(input("Inserisci un numero: "))

M = fibonacci6(MatriceFibonacci, n-1)
print(M[0][0])
```

```
#Complessità temporale: O(log(n))
```

Tutto il tempo è speso nella procedura PotenzaMatrice, all'interno della quale si spende tempo costante e si esegue una chiamata ricorsiva con input $\lfloor n/2 \rfloor$.

Pertanto l'equazione di ricorrenza è: $T(n) \leq T(\lfloor n/2 \rfloor) + c$ che risulta per il **metodo dell'iterazione**:

$T(n) \leq c + T(\lfloor n/2 \rfloor)$
$T(n) \leq 2c + T(\lfloor n/4 \rfloor)$
$T(n) \leq 3c + T(\lfloor n/8 \rfloor)$
.
.
.
$T(n) \leq ic + T(\lfloor n/2^i \rfloor)$

Per $i = \lfloor log_2 n \rfloor$ si ottiene:

$$T(n) \leq c\lfloor log_2 n \rfloor + T(1) = O(log_2(n))$$

Quindi fibonacci6 è esponenzialmente più veloce di fibonacci3.

Due parole sulla complessità spaziale

Algoritmo non ricorsivo: dipende dalla memoria (ausiliaria) allocata; es. variabili, array, matrici, strutture dati, ecc.

Algoritmo ricorsivo: dipende dalla memoria (ausiliaria) allocata da ogni chiamata e dal numero di chiamate che sono contemporaneamente attive.

> 🖊 **Una chiamata usa sempre almeno memoria costante**

Analizzare l'albero della ricorsione aiuta a capire le chiamate attive allo stesso momento. Infatti le chiamate attive formano un **cammino** P che va dalla radice al nodo

Possiamo usare questa idea per calcolare lo spazio di fibonacci2 e fibonacci6.

Riepilogo

	Tempo di esecuzione	Occupazione di memoria
fibonacci2	$O(\phi^n)$	$O(n)$
fibonacci3	$O(n)$	$O(n)$
fibonacci4	$O(n)$	$O(1)$
fibonacci5	$O(n)$	$O(1)$
fibonacci6	$O(log_2(n))$	$O(log_2(n))$