▼ **2.0 - Introduzione agli algoritmi**

Un **algoritmo** rappresenta una procedura per risolvere un problema in un numero finito di passi.

▼ **2.1 - Ingredienti di un algoritmo**

Differenza tra algoritmo e programma

Un **algoritmo** è una descrizione fatta in un alto livello di una procedura. Gli algoritmi non possono essere eseguiti in memoria e possono utilizzare una quantità illimitata di memoria.

Un **programma** invece è l'implementazione di un algoritmo. Deve essere scritto in un qualche linguaggio di programmazione, può essere eseguito su un computer e deve tenere conto dei limiti di memoria di quest'ultimo.

Ingredienti di un algoritmo

Un algoritmo prende in input alcuni valori, esegue una sequenza finita di operazioni e produce un output.

Esistono infiniti set di istruzioni che risolvono lo stesso problema, ovvero forniscono lo stesso output per lo stesso input.

Cosa ci serve per poter sviluppare algoritmi?

Capire il problema che vogliamo risolvere, chiedendoci anche se esistono proprietà matematiche legate ad esso.

Apprendere in che modo stimare l'efficienza di un algoritmo, in termini di tempo e memoria.

Studiare tecniche algoritmiche e strutture dati note, in quanto problemi differenti spesso condividono la stessa struttura di base.

▼ **2.2 - Algoritmi per il calcolo dei numeri di Fibonacci**

Algoritmo 1: formula chiusa

Esiste una **formula chiusa** per calcolare il valore di F_n, ovvero $\frac{1}{\sqrt{5}}\left(ø^n - \widehat{ø}^n\right)$, dove ø e $\widehat{ø}$ sono due costanti irrazionali.

L'algoritmo dunque può semplicemente restituire il valore restituito da tale formula, il problema sta però nell'implementazione informatica di tale algoritmo, in quanto un computer non può memorizzare delle costanti irrazionali, e utilizzando dei valori approssimati risulterà nell'approssimazione del risultato, cosa non accettabile.

Algoritmo 2: soluzione ricorsiva 1

Il secondo algoritmo prevede la conversione diretta della seguente **formula ricorsiva**

$$F_n = \begin{cases} 1 & \mathrm{n} \leq 2 \\ F_{n-1} + F_{n-2} & \mathrm{n} > 2 \end{cases}$$

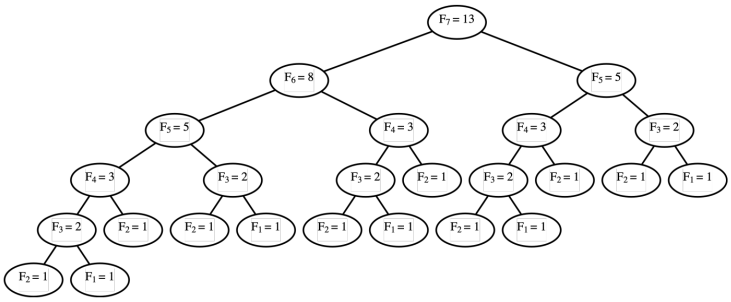
in un algoritmo ricorsivo.

Facciamo una stima della memoria e del tempo necessario al calcolo.

Stima della memoria

Ogni chiamata ricorsiva causa l'allocazione di un record di attivazione sullo stack, dunque la quantità di memoria utilizzata da tale algoritmo è causata unicamente dalle **chiamate ricorsive**.

Bisogna fare però attenzione alla quantità di chiamate ricorsive che il programma effettua in una singola esecuzione in quanto, come possiamo notare dal seguente **albero binario**, questo algoritmo non richiede solo la memoria per calcolare il numero n dato in input, ma anche per tutti i numeri inferiori ad esso.



Albero di ricorsione per n = 13.

Stima del tempo

Per stimare il tempo necessario all'esecuzione dell'algoritmo stimiamo innanzitutto il numero il numero $T(n)$ di nodi nell'albero di ricorsione, sapendo che l'albero $T(n)$ contiene 1 nodo + i nodi dei sottoalberi $T(n-1)$ e $T(n-2)$:

$$T(n) = \begin{cases} 1 & \text{n} \leq 2 \\ T_{n-1} + T_{n-2} + 1 & \text{n} > 2 \end{cases}$$

Da questa relazione possiamo inoltre stimare un limite inferiore per $T(n)$ sapendo che $T(n-1) \geq T(n-2)$:

$T(n) = T(n-1) + T(n-2) + 1$
$\geq 2T(n-2) + 1$
$\geq 4T(n-4) + 2 + 1$
$\geq 8T(n-6) + 4 + 2 + 1$
$\geq ...$
$\geq 2^{\frac{n}{2}} + \frac{2^{\frac{n}{2}} - 1}{2-1}$
$\geq 2^{\frac{n}{2}}$

Da ciò abbiamo ottenuto che la funzione presenta un **numero esponenziale di nodi** maggiore di $2^{\frac{n}{2}}$.

Algoritmo 3: soluzione iterativa

Visto che la soluzione precedente ricalcolava gli stessi numeri di Fibonacci più volte è facilmente pensabile una soluzione iterativa che memorizza all'interno di un **array** i numeri di Fibonacci calcolati:

```
int fib3(int n) {
  let F[1 ... n] be an array of int
  F[1] = 1
  F[2] = 2

  for (int i = 3; i <= n; i++) {
```

```
    F[i] = F[i - 1] + F[i - 2]
  }

  return F[n]
}
```

Stima della memoria

Somma delle variabili utilizzate, ovvero di tutti i numeri memorizzati nell'array, **proporzionale al numero n dato in input**.

Stima del tempo

Calcoliamo il numero delle operazioni elementari del programma utilizzato: $4 + 3(n - 2)$, ovvero $3n - 2$, anch'esso **proporzionale al numero n dato in input**.

Algoritmo 4: soluzione efficiente in memoria

Visto che per calcolare un certo numero di Fibonacci occorre avere **in memoria solamente i due numeri precedenti**, non è necessario memorizzare tutti i numeri all'interno di un array, ottenendo quindi questo algoritmo:

```
int fib4(int n) {
  a = 1
  b = 1
  for (int i = 3; i <= n; i++) {
    c = a + b
    a = b
    b = c
  }
  return b
}
```

Stima della memoria

La memoria utilizzata è **costante** per qualunque numero n inserito come input, in quanto le variabili utilizzate sono sempre al massimo 5.

Stima del tempo

Calcoliamo tutte le operazioni elementari del programma, ottenendo $3 + 5(n - 2)$, ovvero $5n - 7$, **proporzionale al numero n dato in input**.

Algoritmo 5: potenza di matrici

Nonostante l'amplia ottimizzazione in tempo e memoria fatta finora possiamo sfruttare un ulteriore teorema matematico riguardante le matrici al fine di ottimizzare ulteriormente il nostro algoritmo.

Il teorema in questione è il seguente:

Consideriamo $A = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$, per ogni $n \geq 2$ abbiamo che

$$A^{n-1} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1} = \begin{pmatrix} F(n) & F(n-1) \\ F(n-1) & F(n-2) \end{pmatrix}$$

Sfruttando questo teorema è possibile costruire il seguente algoritmo:

```
int fib5(int n) {
  A = {{1, 1}, {1, 0}}
  for (int i = 2; i <= n; i++) {
    A = A × {{1, 1}, {1, 0}}
  }
  return A[1][1]
}
```

Utilizzando questo algoritmo in realtà non otteniamo **alcun miglioramento** in quanto le operazioni rimangono proporzionali ad n, mentre la memoria rimane costante, ma l'utilizzo di matrici ci consentirà di utilizzare una proprietà di queste in grado di velocizzare l'operazione di potenza.

Algoritmo 6: potenza di matrici velocizzata

La proprietà delle matrici che utilizziamo per velocizzare il tempo di esecuzione dal calcolo del numero di Fibonacci è la seguente:

$$A^n = \begin{cases} (A^{\frac{n}{2}})^2 & \text{n } pari \\ (A \times (A^{\frac{n-1}{2}})^2) & \text{n } dispari \end{cases}$$

L'algoritmo da utilizzare è dunque il seguente:

```
FibMat fibMatPow(int n) {
  A = {{1, 1}, {1, 0}}

  if (n > 1) {
    M = fibMatPow(n / 2)
    A = M × M
    if (n % 2 != 0)
      A = A × {{1, 1}, {1, 0}}
  }

  return A
}

int fib6(int n) {
  M = fibMatPow(n − 1)
  return M[1][1]
}
```

Stima del tempo

Il tempo di calcolo è dimostrabile essere **proporzionale a** $\log_2 n$.

Stima della memoria

La memoria è costante per ognuna delle $log_2 n$ chiamate di procedura, dunque anch'essa è **proporzionale a** $\log_2 n$.

Sommario

Possiamo fare un resoconto in notazione asintotica riguardante gli algoritmi di calcolo dei numeri di Fibonacci appena visti:

Algoritmo	Tempo	Memoria
Fib2	$\Omega(2^{n/2})$	$O(n)$
Fib3	$O(n)$	$O(n)$
Fib4	$O(n)$	$O(1)$
Fib5	$O(n)$	$O(1)$
Fib6	$O(\log n)$	$O(\log n)$

Resoconto algoritmi di Fibonacci in termini di valore dell'input.

Solitamente però l'efficienza di un certo algoritmo viene valutata in termini di dimensione dell'input, ovvero di numero di bit utilizzati per rappresentarlo, e non del valore effettivo che ha l'input. Operando in questa maniera otteniamo il seguente resoconto:

Algorithm	Time	Space
Fib2	$\Omega(2^{2^{\lvert n\rvert}})$	$O(2^{\lvert n\rvert})$
Fib3	$O(2^{\lvert n\rvert})$	$O(2^{\lvert n\rvert})$
Fib4	$O(2^{\lvert n\rvert})$	$O(1)$
Fib5	$O(2^{\lvert n\rvert})$	$O(1)$
Fib6	$O(\lvert n\rvert)$	$O(\lvert n\rvert)$

Resoconto algoritmi di fibonacci in termini di dimensione dell'input.