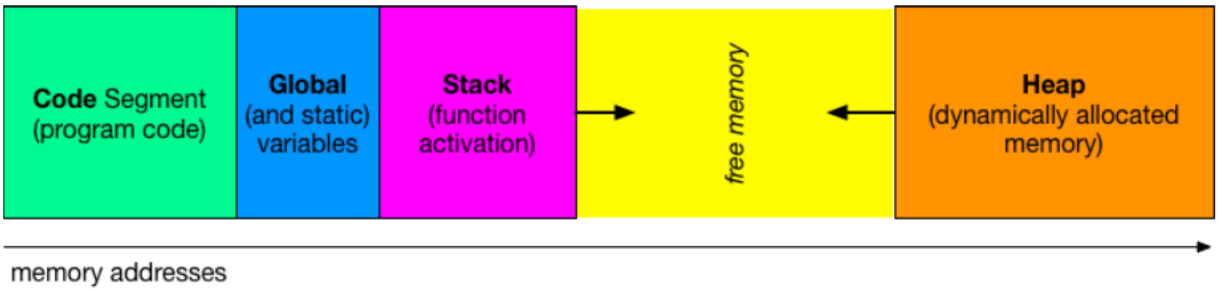
Riassunto di SDA

indice

- Riassunto di SDA
  - indice
  - 01 - Organizzazione della memoria, chiamate di funzioni, ricorsione
    - L'organizzazione della memoria
  - -se lo stack e l'heap s'incontrano si verifica un' errore di memoria(stack/heap overflow)
    - Record di attivazione
  - E' come una scatola chiusa, la funzione può fare tutto chiò che vuole senza interferire con l'ambiente di altre funzioni.
    - La ricorsione
      - Esempio di codice dell' algoritmo del pracheggio
      - Altri esempi di funzioni ricorsive
      - ricorsione: meccanismo computazionale
      - ricorsione ed induzione
  - 02 - Trattabilità e complessità computazionale
    - Problemi decidibili e indecidibili
      - Problemi decidibili
      - Problemi indecidibili
    - Complessità
      - Torri di Hanoi
      - Problema della ricerca della coppia di putni più vicini
      - Algoritmi esponenziali ed algoritmi polinomiali
    - Classi di complessità dei problemi
      - Classi di complessità
      - Problemi in NP
      - Analisi di complessità
      - Notazione asintotica e ordini di complessità
      - Ordini di Complessità
      - Analisi strutturale
        - Guida per il calcolo del costo al caso pessimo

01 - Organizzazione della memoria, chiamate di funzioni, ricorsione

L'organizzazione della memoria



1. **Code segment**
   - Contiene il codice eseguibile del programma, ossia le istruzioni della CPU
   - è un area di sola lettura, per evitare che il codice venga modificato durante l'esecuzione
   - include funzioni, metodi e istruzioni scritte nel programma
   - *esempio* in C contiene il codice compilato dal programma*
2. **Global segment**
   - Memorizza tutte le variablili globali e statiche del programma
   - le variabili sono allocate una volta sola e persistono per tutta la durata del programma
   - contiene dati sia inizializzati che non
   - *esempio* la variabile int `x = 5`; verrà memorizzata qui
3. **Stack**
   - è l'area di memoria utilizzata per la genstione delle chiamate di funzioni e di variabili locali. -Cresce 'verso il basso'(verso indirizzi di memoria più bassi)
   - Quando viene chiamata una funzione il suo contesto(variabili Locali, indirizzi di ritorno, ecc.) vengono allocati nello stack -*esempio* una variabile locale `int y = 10` sarà nello stack.
4. **heap**
   - è utilizzata per allocare dinamicamente la memoria.
   - è crescente verso l'alto(indirizzi di memoria più alti)
   - l'allocazione e deallocazione è gestita esplicitamente dal programmatore(es. `malloc/free` in C)
   - è più flessibile rispetto allo stack, è più lenta ed è più probabile che sia soggetta ad errori come memory leaks
5. **Free memory**
   - è la memoria libera tra lo stack e l'heap -se lo stack e l'heap s'incontrano si verifica un' errore di memoria(stack/heap overflow)

Record di attivazione

Ogni funzione crea un suo spazio di memorizzazione specifico nello stack chiamato **record di attivazione**. Esso include tutte le variabili locali, i parametri ed i valori di ritorno della funzione.

E' come una scatola chiusa, la funzione può fare tutto chiò che vuole senza interferire con l'ambiente di altre funzioni.

La ricorsione

La ricorsione è una tecnica di programmazione, ma anche uno strumento per risolvere problemi. Una funzione è ricorsiva quando al suo interno è presente una chiamata a se stessa.

- La funzione ricorsiva può risolvere direttamente dei casi del problema, detti *casi base*, in questo caso restituisce subito un risultato
- Se vengono passati dei dati che non costituiscono uno dei casi base chiama se stessa passando dei dati *ridotti* o *semplificati*
- ad ogni chiamata i dati si riducono fino ad arrivare ad uno dei **casi base**

Esempio di codice dell' algoritmo del pracheggio

```
#include <stdio.h>
#include <stdlib.h> //per random

double drand (double low, double high);

void park (double a, double b)
{
   double p;
   p = drand(a,b);
   printf("Auto parcheggiata nella posizione %f\n", p);
   if(p - a >= 1.0) // c'è spazio sufficente a sinistra per almeno un' Auto
   {
      park(a, p - 1.5);
   }
   if(b-p >= 1.0) // c'è spazio a destra per almeno un'auto
   {
      park(p + 1.5, b);
   }
}
int main(){
   double l = 10.0; // abbiamo una strada lunga 10
   park(1,l-1);
   return EXIT_SUCCESS;
}
```

Altri esempi di funzioni ricorsive

- **esempio 1**: il Fattoriale $f(n) = n!$ è definita ricorsivamente così:

$$f(n) = \begin{cases} 1 & se \quad n = 0 \\ n \cdot f(n-1) & se \quad n > 0 \end{cases}$$

```
long fattoriale(long n){
   if n == 0
   {
      return 1;
```

```
   }
   else
   {
      return n * fattoriale(n-1);
   }
}
```

Il calcolo viene ricondotto al calcolo del fattoriale di un numero più piccolo fino a raggiungere il caso base noto.

ricorsione: meccanismo computazionale

quando la funzione chiama se stessa essa si sospende per eseguire la chiamata ricorsiva. L'esecuzione riprende quando la chiamata interna termina la sequenza di chiamate ricorsive termina quando quella più annidata incontra uno dei casi si base Ogni chiamata alloca sullo *stack* nuove istanze dei parametri e delle variabili locali all' interno dei **record di attivazione**

ricorsione ed induzione

La ricorsione è basata sul principio di *induzione matematica*:

- se una proprietà P vale per $n = n_0$ (**Caso base**)
- e se posso provare che, *assumendola valida per* n, vale ancheper $n + 1$ (**passo induttivo**)
- allora P vale per ogni $n \geq n_0$

Risolvere il problema con un approccio ricorsivo comporta:

1. l'identificazione del **caso di base** in cui la soluzione è nota
2. la capacità di **esprimere il caso generico** in termini dello stesso problema ma in uno o più casi semplificati

- **Esempio 2**: Numeri di Fibonacci

```
inf f(int a, int b){
   assert(b>= 0);//assumiamo b >= 0
   if (b == 0)
      return a;
   else
      return f(a, b - 1) + 1;

}
```

La successione è definita come

n	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
F_n	1	1	2	3	5	8	13	21	34	55	89	144

- $F_0 = 1$
- $F_1 = 1$
- $F_n F_{n-1} + F_{n-2}$ per $n \geq 2$

Numeri di Fibonacci

```
int fibonacci(int n) {
if (n == 0 || n == 1)
return 1;
else
```

```
return fibonacci(n - 1) + fibonacci(n - 2;
}
```

02 - Trattabilità e complessità computazionale

Problemi decidibili e indecidibili

Non tutti i problemi computazionali possono essere risolti algoritmicamente, per esempio il problema della fermata(halting problem) (Turing, 1937) Non c'è una soluzione nel caso generale(cioè una dimostrazione che consenta di rispondere per un *qualunque algoritmo*) In alcuni casi specifici tuttavia è possibile dare delle risposte

Problemi decidibili

- un problema è **decidibile** (o **calcolabile**) se esiste un algoritmo che per ogni istanza è in grado di **terminare** la sua esecuzione giungendo ad una soluzione
- **Esempio**: *determinare se un numero è primo*

```
#include <stdbool.h>

bool primo (int n)
{
   int fattore;
   if (n== 1)
      return false;
   if (n == 2)
      return true;
   for (fattore = 2; fattore < n / 2; fattore++)
   if (n % fattore == 0)
      return false;
   return true;
}
```

Problemi indecidibili

- un problema è **indecidibile** (o **non calcolabile**) se nessun algoritmo sia in grado di terminare la sua esecizone giungendo ad una soluzioni ad **ogni istanza**
  dato un programma `A`, non esiste un algoritmo per stabilire se esso terminerà o meno la sua esecuzione su un qualunque input **Problema della fermata(Turing, '37)**

Complessità

alcuni problemi, nonostante decidibili, possono richiedere un tempo di risoluzione notevole, per esempio il problema delle

Torri di Hanoi

- **Obiettivo**: spostare tutti i dischi dal piolo `a` al piolo `c`
- Ogni mossa sposta un disco in cima a un piolo con il vincolo che un disco non può poggiare su uno più piccolo
- **Descrizione delle mosse**: Supponiamo che i pioli siano etichettati con `A`, `B`, `C`, e i dischi numerati da 1 (il più piccolo) a n (il più grande).
- **algoritmo ricorsivo**
  i. sposta i primi $n-1$ dischi da `A` a `B`
  ii. sposta il disco n da `A` a `C`
  iii. sposta i primi $n-1$ dischi da `B` a `C`

per spostare gli n dischi effettuiamo un operazione elementare(la 2, spostamento di un disco) e delle operazioni complesse(la 1 e la 3) che, però, hanno la stessa struttura del problema originale, ossia lo spostamento di $n-1$ dischi, solo su un numero di dischi ridotto di un' unità.

- **stampa delle mosse**

```
void torri_hanoi(int n, char partenza, char intermedio, char destinazione) {
if (n > 1) {
// Sposto tutti gli n - 1 dischi dal piolo partenza
// al piolo intermedio usando il piolo destinazione come appoggio
torri_hanoi(n - 1, partenza, destinazione, intermedio);
/* Sposto il disco rimanente (più largo di tutti) sul piolo destinazione */
printf("disco %d: %c -> %c\n", n, partenza, destinazione);
// Sposto tutti gli n - 1 dischi dal piolo intermedio
// al piolo destinazione usando il piolo partenza come appoggio
torri_hanoi(n - 1, intermedio, partenza, destinazione);
} else {
// È rimasto un solo piolo, lo sposto dal piolo partenza a quello destinazione
printf("disco %d: %c -> %c\n", n, partenza, destinazione);
}
}
/* esempio di chiamata iniziale */
torri_hanoi(5, 'A', 'B', 'C');
```

-**numero di mosse**

> Per spostare gli n dischi servono $2^n - 1$ mosse.

Dimostrabile per induzione:

- **Caso base**, n = 1: è necessaria una sola mossa, e coerentemente $1 = 2^1 - 1$ù
- **Passo induttivo**, $n > 1$: per ipotesi induttiva lo spostamento di $n-1$ dischi richiede un numero di passi pari a $2^{n-1} - 1$

L'algoritmo sposta tutti gli $n-1$ dischi dal piolo di partenza al piolo intermedio, poi sposta il disco rimasto sul piolo di destinazione e infine risposta tutti gli $n-1$ dischi dal piolo intermedio a quello di destinazione:

$$\overbrace{(2n-1-1)}^{\text{n-1 dischi dal piolo A al B}} + \overbrace{1}^{disconaC} + \underbrace{(2^{n-1}-1)}_{\text{n-1 dischi dal piolo B al C}} \qquad (2^{n-1}-1)+1 = 2^n - 1$$

-**tempo di calcolo**: Supponiamo di fare una mossa al secondo, i tempi di calcolo seguono la seguente legge

n	**5**	**10**	**15**	**20**	**25**	**30**	**35**	**40**	**64**
tempo	$31s$	$17m$	$9h$	$12g$	$1a$	$34a$	$1089a$	$34865a$	$585 \cdot 10^9 a$

E se invece di fare una mossa al secondo potessi fare b mosse al secondo per quanti dischi m aggiuntivi rispetto a m riesco a risolvere il gioco in un dato tempo t?
Devoi risolvere l'equazione:

$$(2^{n+m} - 1)/b = t$$

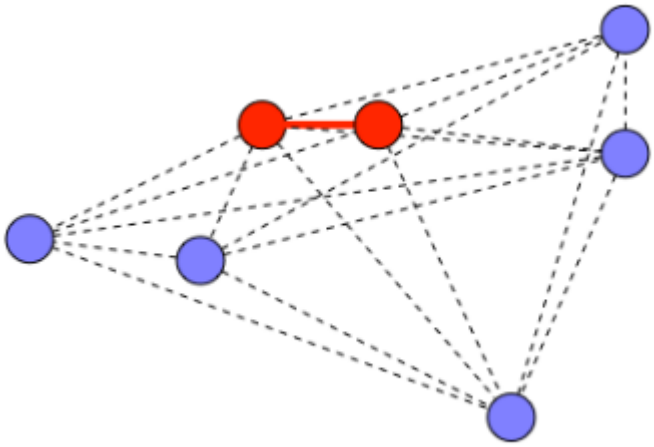
$$2^{n+m} = tb + 1$$
$$n + m = log_2(tb + 1) \approx log_2\, t + log_2\, b$$

poichè facendo una mossa al secondo nel tempo t riesco a risolvere n dischi:$n \approx log_2\, t$ e dunque $n + m \approx n + log_2\, b \implies m \approx log_2\, b$
Quindi, dato che $m \approx log_2\, b$, se **raddoppio l amia velocità ($b = 2$), riesco a risolvere 1 solo disco in più.
Ad esempio, per il numero di dischi *"trattabili"* in un giorno (<24h)

operazioni/sec	**1**	**10**	**100**	10^3	10^4	10^5	10^6	10^9
$n + m$	17	20	23	26	29	32	35	45

> Il problema delle torri di hanoi è **intrattabile**: è dimostrabile che non è possibile usare meno di $2^n - 1$ mosse. (non può esistere un algoritmo che usi un numero inferiore di mosse)

Problema della ricerca della coppia di putni più vicini
Dato un insieme di punti P disposti su di un piano, trovare il valore della distanza della coppia di punti che si trova a distanza minima



- **Problema della ricerca della coppia di punti più vicini

```
#include <math.h>
double puntiPiuVicini(/* insieme di punti */P, int n)
{
   double min = +INFINITY, d;
   for (i=0; i<n; i++)
   {
      for (j=0; j<n; j++)
      {
         d = distanza(P,i,j);// distanza fra i punti i e j
         /*confronto*/
         if(i != j && d <min)
         {
            min = d;
         }
      }
   }
   return min;
```

```
}
```

- **Tempo di calcolo**

Quante operazioni di calcolo della distanza fra punti verranno effettuate?

$$\sum_{i=0}^{n-1}\sum_{j=0}^{n-1} 1 = \sum_{i=0}^{n-1} n = n \cdot n = n^2$$

Supponendo di effettuare un controllo al secondo, i tempid i calcolo crescono secondo la seguente legge:

n	**5**	**10**	**15**	**20**	**25**	**30**	**35**	**40**
tempo	$25s$	$100s$	$4m$	$7m$	$10m$	$15m$	$20m$	$26m$

Se invece di fare un confronto al secondo, si fanno b confronti al secondo del tempo t riesco a risolvere il problema per uin vettore di dimensione $n + m$, dove $t = (n + m)^2/b$

$$n + m = (tb)^{1/2} = \sqrt{t}\sqrt{b} = n\sqrt{b}$$

Aumentare di un fattore moltiplicativo b (ossia b operazioni/sec) migliora di un fattore moltiplicativo circa pari a $\sqrt{b}$

operazioni/sec	**1**	**10**	**100**	10^3	10^4	10^5	10^6
$n + m$	64	202	640	2023	6400	20238	64000

Algoritmi esponenziali ed algoritmi polinomiali

L'algoritmo delle torri di Hanoi è un algoritmo **esponenziale**: richiede un tempo proporzionale ad una funzione esponenziale della dimensione dell' input.

> EPr inciso é la stessa situazione del riempimento delle Terapie intensive in caso di malattie contagiose in assenza di controlle: possiamo raddoppiarne i posti ($b = 2$), ma il numedo di giorni che servirà per riempirle interamente si sposta solamente si qualche unità ma no raddoppia. L'algoritmo della ricerca della coppia di putni vicini è un' algoritmo **polinomiale** proporzionalmente a un polinomio della dimensione dell'input (solitamente di grado basso)

- **Dimensione dei dati per un problema generico**

Esistono diverse caratterizzazioni dimensionali per i dati di un problema: per la rappresentazone dei dati: k bit possono rappresentare interi in

$$\{0, 1, \dots, 2^k - 1\}S$$

Esempio, per $k = 3$

000	**001**	**010**	**011**	**100**	**101**	**110**	**111**
0	1	2	3	4	5	6	7

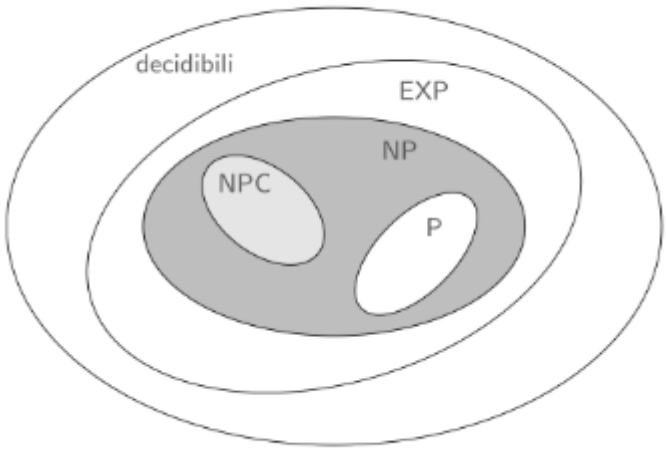
numero di elementi di una struttura dati: array, stringhe, liste, insiemi, alberi
Numero di celle di memoria: occupate dai dati (ognuna contiene $\approx log n$ bit)

-**Definizione**: Un algoritmo è detto ***polinomiale***, nella dimensione di in put n, se esistono due costanti $c, n_0 > 0$ tali che il numero di passi elementari da esso eseguiti è al più n^c, per ogni input di dimensione n e per ogni $n > n_0$

-**Definizione**: Un algoritmo è detto ***polinomiale***, nella dimensione di in put n, se esistono due costanti $c, n_0 > 0$ tali che il numero di passi elementari da esso eseguiti è al più c^n, per ogni input di dimensione n e per ogni $n > n_0$

- **Problema trattabile**: esiste un algoritmo polinomiale che lo risolve.
- **Problema intrattabile**: non esiste un algoritmo polinomiale che lo risolve.

Classi di complessità dei problemi



Classi di complessità

- P classe dei problemi risolvibili(*deterministicamente*) in tempo polinomiale
- EXP classe dei problemi risolvibili(*deterministicamente*) in tempo esponenziale
- NP classe dei probleni per i quali verificare una soluzione richiede tempo polinomiale(risolvibili(*non-deterministicamente*)in tempo polinomiale)
- NPC classe dei problemi *completi* per $NP£, detti NP -Completi

Problemi in NP

Basato sul concetto di *"certificato polinomiale"* per un problema computazionale Π

- chi ha la soluzione per in istanza di Π può convincere che la soluzione è giusta in tempo polinomiale
- chi non ha tale soluzione deve procedere per tentativi richiedendo tempo esponenziale

Esempio: date le tre variabili proposizionali $p, q, r \in \{T, F\}$ e la formula logica $\Phi \equiv (p \vee \bar{q} \vee r) \wedge (p \vee r)$, verificare che $p := T$, $q := F$ $r := F$ soddisfa la formula Φ è facile, basta sostituire i valori di verità e applicare le regole di valutazione:

$$(T \vee \bar{F} \vee F) \wedge (T \vee F) \equiv T \wedge T \equiv T$$

Per trovare la soluzione è necessario provare le "2^n combinazioni $(p, q, r) := (T, T, T), (T, T, F), (T, F, T), \ldots$ di valori di verità per p, q, r

NP è la classe di problemi che ammettono un certificato polinomiale

Ovviamente, $P \subseteq NP$

- infatti, quando il problema Π sia polinomiale (ossia $\Pi \in P$), il certificato coincide con la soluzione del problema

Riguardo la relazione inversa, non sappiamo se $NP \overset{?}{\subseteq}$ oppure $NP \overset{?}{\nsubseteq} P$

Analisi di complessità

Negli esempi precedenti abbiano fatto una stima grossolana del numero di operazioni necessarie per risolvere un problema di input n

Per avere delle stime più accurate è necessario introdurre un modello di calcolo generico(astratto) e una misura delle operazioni su tale modello di calcolo

- Miusra di **performance** di un *algoritmo* espresse in funzione della dimensione dei dati in input n
  - **tempo** numero di operazioni RAM(elementari) eseguite
  - **spazio** numero di celle di memoria occupate(in aggiunta a quelle per l'input)
- **Complessità** o **Costo computazionale** $f(n)$ in tempo e *spazio* di un problema Π:
  - caso *pessimo* o *peggiore*: costo massimo fra tutte le istanzs di Π aventi dimensioni dei dati pari a n
  - caso *medio*: costo mediato tra tutte le istanza di Π aventi dimensioni pari a n
  - ~~caso *ottimo*: costo minimo fra tutte le istanze di Π aventi dimensione dei dati pari a n~~

Scopi:

- Stimare il tempo impiegato per elaborare un dato input
- Stimare il più grande input gestibile in tempi "ragionevoli"
- **Confrontare l'efficenza di algoritmi diversi**
- ottimizzare le parti più importanti

Dimensione dell' input

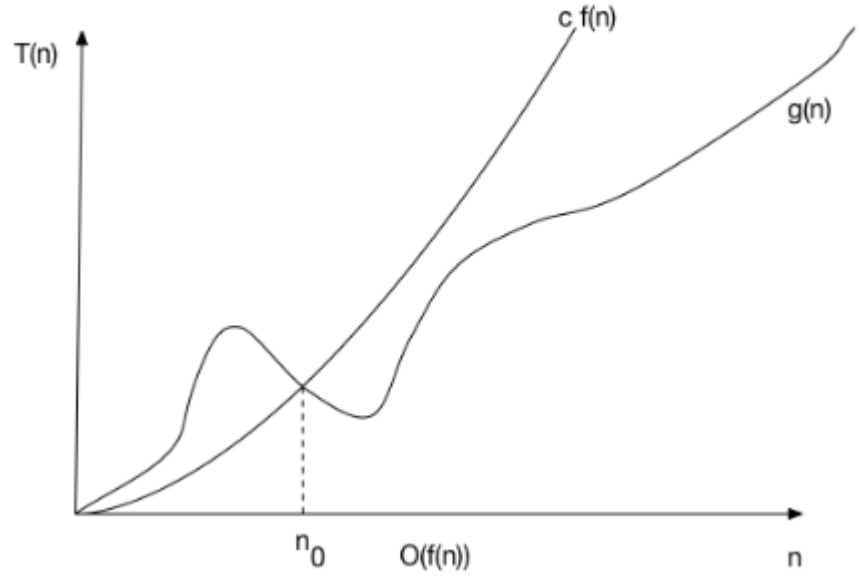
- Costo *logaritmico*: la taglia dell' input è il numero di bit necessari per rappresentarlo
  - Esempio: moltiplicazione di numeri binari lunghi n bit
- Costo *uniforme*: la taglia dell' input è il numero di elementi di cui è costituito
  - Esempio: ricerca minimo in un vettore di n elementi

> Possiamo assumere che gli "elementi" siamo rappresentati da un numero costante di bit e pertanto le due misure coincidono a meno di una costante moltiplicativa

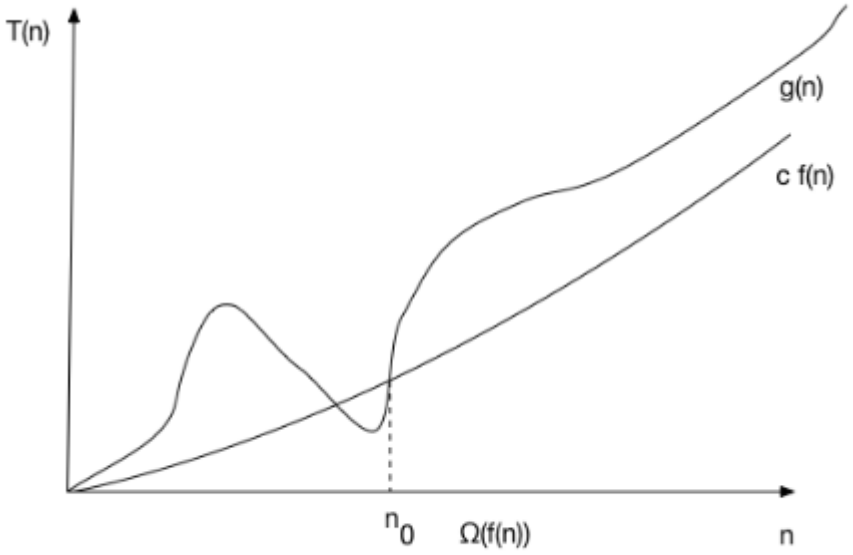
Notazione asintotica e ordini di complessità

> Dato che l'accuratezza dei risultati non è il nostro interesse principale utilizzeremo una notazione asintotica, che si avvicina a come si comporta l'algoritmo al crescere delle dimensioni del suo input

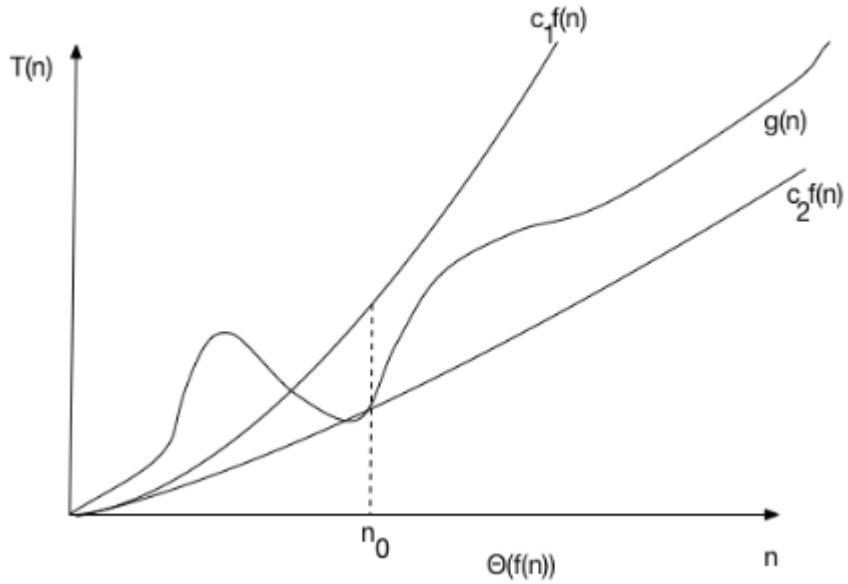
- **Limitazione superiore** $O(f(n))$
  Definiamo $g(n) = O(f(n))$ se e solo se $\exists\ c, n_0 : g(n) \leq cf(n) \forall n > n_0$



- **Limitazione inferiore** $\Omega(f(n))$
  Definiamo $g(n) = \Omega(f(n))$ se e solo se $\exists\ c, n_0 : g(n) \geq cf(n) \forall n > n_0$



- **Limitazione** $\Theta(f(n))$
  Definiamo $g(n) = \Theta(f(n))$ se e solo se $\exists c_1, c_2, n_0 : c_1 f(n) \geq g(n) \geq c_2 f(n) \forall n > n_0$



Ordini di Complessità

$f(n)$	10^1	10^2	10^3	10^4	**Tipo**
$log\ n$	3	6	9	13	Logaritmico
$\sqrt{n}$	3	10	31	100	sublineare
n	10	100	1000	10000	lineare
$n\ log\ n$	30	664	9965	132877	loglineare
n^2	10^2	10^4	10^6	10^8	quadratico
n^3	10^3	10^6	10^9	10^12	cubico
2^n	1024	10^30	10^300	10^3000	esponenziale

$$O(1) \subset O(log\ n) \subset O(n^h) \subset (n^h log\ n) \subset O(a^n) \subset O(n^n)$$

Proprietà della notazione asintotica

Per stimare gli ordini di grandezza non è necessario applicare le definizioni ma si possono usare le seguenti regole per semplificare i calcoli

- **Riflessività**: per ogni costante $c, c \cdot f(n)$ é $O(f(n))$(lo stesso per Ω e Θ)
- **Transitività**: se $g(n) = O(f(n))$e $f(n) = O(h(n))$allora$g(n) = O(h(n))$(lo stesso per Ωe Θ)
- **Simmetria**: $g(n) = \Theta(f(n))$ se e solo se $f(n) = \Theta(g(n))$
- **Simmetria trasposta**: $g(n) = \Theta(f(n))$ se e solo se $f(n) = \Omega(g(n))$
- **Somma**: $f(n) + g(n) = O(max\{f(n), g(n)\})$(lo stesso per Ω e Θ)
- **Prodotto**: $g(n) = O(f(n)), h(n) = O(q(n))$allora $g(n) \cdot h(n) = O(f(n) \cdot q(n))$(lo stesso per Ω e Θ)

Analisi strutturale

Per gli algoritmi descritti in linguaggi imperativi come il C é possibile definire delle regole che consentono di stimare la complessità computazionale direttamente dalla struttura del programma.

> La stima strutturale al caso pessimo risulta essere particolarmente agevole grazie all' assorbimento degli ordini di crescita inferiori nella notazione asintotica

Guida per il calcolo del costo al caso pessimo

- La complessità di una sequenzqa di istruzioni è data dalla somma delle complessità delle singole istruzioni della sequenza
- Il costo di una chiamata a funzione è il costo del suo corpo più il passaggio dei parametri(qualora siano a loro volta delle operazioni non elementari)
  - Le funzioni ricorsive saranno trattate a parte
- la complessita di una condizione `if(guardia)blocco1 else blocco2` è data da:

$$costo(guardia) + max\{costo(blocco1), costo(blocco2)\}$$

- la complessità di un ciclo con un numero determinato di iterazione `for i = 0; i<m; i++` è data da:

$$\sum_{i=0}^{m-1} costo(corpo)_i + 2costante$$

costo del corpo all' iterazione i, più due volte un costo costante dovuto alla valutazione di inizializzazione / incremento e condizione

- la complessità di un ciclo con un numero indeterminato di iterazioni `while(guardia)corpo` è data da:

$$\sum_{i=0}^{m-1} costo(guardia)_i + costo(corpo)_i + costo(guardia)_m$$

dove m é il massimo numero di volte in cui la guardia risulta soddisfatta. **Esempio**
Costo computazionale nel calcolo del minimo di un vettore (versione naive)

```
int minNaive(int a[], int n)
{
   int i,j;
   bool is_min;
   for (i = 0;i < n; i++)
   {
      is_min = true;
      for(j = 0, j< n; j++)
      {
         if(a[i]> a[j])
         {
            is_min = false;
         }
      }
   if(is_min)
   {
```

```
      return a[i];
    }
    }
}
```

$$T(n) = 2c_1 \cdot n + c_2 \cdot n + 2c_3 \cdot n \cdot n + c_4 \cdot n \cdot n + c_5 \cdot n \cdot n + c_6 \cdot n + c_7 =$$
$$n^2(2c_3 + c_4 + c_5) + n(2c_1 + c_2 + c_6) + 1(c_7)$$
$$T(n) = n^2\underbrace{(2c_3 + c4 + c_5)}_{c'} + n\underbrace{(2c_1 + c_2 + c_6)}_{c''} + 1\underbrace{(c_7)}_{c'''}$$
$$O(n^2) + O(n) + O(1) = O(n^2)$$

*(per le regole di **riflessività** e **somma** della notazione asintotica)*

Caso medio Supponendo che i valori nell' array siano distribuiti uniformemente, il minimo ha la probabilità $\frac{1}{n}$ di trovarsi in un punto j del vettore e il csoto dell'algoritmo, per la posizione j è dato da:

$$jn(1 - \frac{1}{n})^{j-1}$$

- $j \cdot n$ è il numero di elementi verificati(il numero di iterazioni del ciclo esterno j moltiplicato per quelle del ciclo interno n)
- $jn(1 - \frac{1}{n})^{j-1}$è la probabilità che il minimo non fosse nessun elemento in posizione minore di j(la probabilità che non sia il primo è $(1 - \frac{1}{n})$, che non sia il secondo $(1 - \frac{1}{n})^2$, che non sia il terzo $(1 - \frac{1}{n})^3$, e così via)

Il** valore atteso** (la media) del numero di valori verificati è dato dalla somma di tali valori per tutte le posizioni j, diviso per il numero n di valori

$$\mathrm{E}[T(n)] = \sum_{j=1}^{n} \frac{jn(1 - \frac{1}{n})^{j-1}}{n} = (1 - 2(1 - \frac{1}{n})^n)n^2$$

si ha

$$\lim_{n \to \infty}(1 - \frac{1}{n})^n = \frac{1}{e}$$

dunque

$$\mathrm{E}[T(n)] \approx (1 - \frac{2}{e})n^2 = O(n^2)$$