L07. CENNI SULLA COMPLESSITÀ COMPUTAZIONALE

L'analisi della complessità computazionale è la stima del costo degli algoritmi in termini di risorse di calcolo, quando parliamo di risorse di calcolo ci riferiamo al tempo di esecuzione, avvolte ci riferiremo anche allo spazio di memoria.

Esempio:

Dato un vettore di n interi ordinati in maniera non decrescente, verificare se l'intero K è presente o meno nel vettore. Per risolvere il problema utilizziamo questo l'algoritmo:

```
int ricerca(int v[], int size, int k){
   int i;
   for (i=0; i<size; i++)
      if(v[i] == k) return i;
   return -1;
}
```

Algoritmo banale che effettua *un'intera visita* del vettore e quando trova l'elemento cercato restituisce la posizione nel vettore altrimenti restituisce -1. Questo algoritmo non è efficiente, dovuta al fatto che l'algoritmo non ha nessun vantaggio dal fatto che il vettore è ordinato, ma sappiamo invece che esiste un algoritmo migliore che è quello dello della ricerca binaria su un Array ordinato.

In particolare, vogliamo essere in grado di analizzare e valutare il tempo di esecuzione degli algoritmi, esistono diverse variabili che influenzano il tempo di esecuzione:

- **Macchina usata**, se seguiamo il nostro algoritmo su un supercalcolatore è differente che se lo eseguiamo su un dispositivo mobile;
- **Dimensione dei dati**, ad esempio per il problema dell'ordinamento, ordinare pochi numeri interi è diverso da ordinare i record degli studenti;
- **Configurazione dei dati**, nel problema precedente sapevamo che i dati erano ordinati e quindi potevamo regolarci di conseguenza.

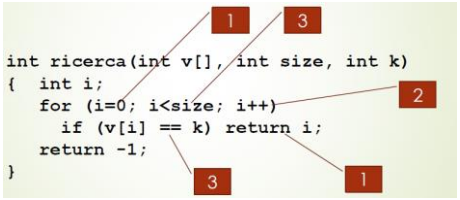
Quello che vogliamo ottenere è un modello astratto per la valutazione del tempo di esecuzione che:

- Sia indipendente dalla macchina usata, così che possa essere applicato in tante situazioni;
- Stimi il tempo in funzione della dimensione dell'input;
- Abbia un **comportamento asintotico**, quindi che prende in considerazione il tempo di esecuzione per taglie dell'input molto grandi perché è proprio per taglie di input grandi che si vede se un algoritmo efficiente oppure no;
- Ci fornisca una stima nel **caso peggiore** della configurazione dei dati, perché anche nel caso peggiore l'algoritmo deve dimostrare la sua efficienza.

Vediamo un esempio di modello di una macchina astratta che conta le istruzioni e le condizioni atomiche che sono presenti all'interno dell'algoritmo:

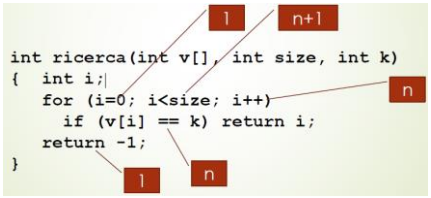
- Istruzioni e condizioni atomiche hanno costo unitario;
- Le strutture di controllo hanno un costo pari alla somma dei costi dell'esecuzione delle istruzioni interne, più la somma dei costi delle condizioni;
- Le chiamate a funzione hanno un costo pari al costo di tutte le istruzioni e condizione in esse contenute, non applichiamo alcun costo per il passaggio dei parametri;
- Istruzioni e condizioni con chiamate a funzioni hanno costo pari alla somma del costo delle funzioni invocate più uno.

Calcolare il costo per l'esempio precedente nel caso v[n] = {1, 3, 9, 17, 34, 95, 96, 101} e k=9.



Contiamo quante istruzioni o condizioni.
La prima inizializzazione della variabile verrà fatta una volta sola, il controllo della condizione verrà fatto tre volte, l'incremento verrà fatto due volte, il controllo della condizione del if verrà fatto tre volte, una volta verrà fatto il Return, in totale abbiamo 10 tra istruzioni e condizioni.

CASO PEGGIORE:



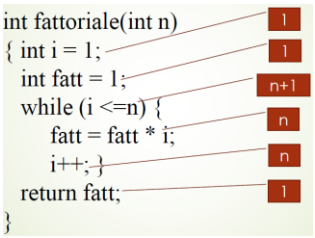
Il caso peggiori è quando l'elemento che cerchiamo non è presente. Quindi saremo costretti a scorrere l'intero vettore, facciamo di nuovo il conteggio dei passi necessari: l'inizializzazione viene fatta sempre una volta sola, il controllo verrà fatto n + 1 volte, incremento della variabile verrà fatto n volte, il controllo del if verrà fatto n volte, il return alla fine verrà fatto una volta sola, quindi in totale avremo **3n + 3 operazioni**.

CASO MEDIO:

Spesso siamo interessati anche a calcolare il caso medio. Tornando al nostro esempio di ricerca all'interno di un Array ordinato, il caso medio è quello in cui il numero cercato è presente e tutte le posizioni abbiano la stessa probabilità di contenere questo numero, quindi per N numeri la probabilità che il numero da cercare K sia in una di questa posizione, cioè nella posizione i è $\frac{1}{N}$.

DIMENSIONE DELL'INPUT:

Abbiamo detto che uno dei requisiti del modello è quello di calcolare il costo computazionale come funzione della dimensione dell'input. Bisogna intendersi su cosa sia la dimensione dell'input, in un vettore ci siamo riferiti alla dimensione dell'input come il numero di elementi contenuti nel vettore. Un ragionamento analogo si può fare per strutture sequenziali come le liste oppure come gli Stack e le code, nel caso di un albero potremmo andare a contare il numero di nodi, trovandoci invece di fronte ad un grafo la dimensione dell'input potrebbe essere data dal numero dei nodi più il numero degli archi. Nel caso invece del fattoriale abbiamo che l'input è un numero intero non limitato, chiaramente quanto più grande è questo numero tanti più passi dovremmo andare a fare, vediamo quanti passi effettuiamo per il calcolo del fattoriale:



Le prime due inizializzazioni vengono fatte una volta sola, il controllo del while viene fatto n + 1 volte, la moltiplicazione all'interno viene fatta n volte, l'incremento pure viene fatto n volte, l'operazione di ritorno viene fatto una volta sola, quindi abbiamo un totale di 3n + 4 operazioni.

Quindi considerando il parametro n intero avremo che la nostra funzione fattoriale richiede un numero di operazioni che è lineare rispetto ad n, infatti 3n + 4 è una funzione lineare. Se invece utilizzassimo un modo diverso per rappresentare n (l'input) potremmo considerarlo come il numero d di bit necessari per rappresentare il numero, in questo caso d sarebbe circa uguale al $\log_2 n$, quindi il costo diventerebbe $3x2^d+4$ che è una funzione esponenziale, quindi chiaramente il modo in cui scegliamo di rappresentare l'input ha un effetto anche sul costo dell'algoritmo.

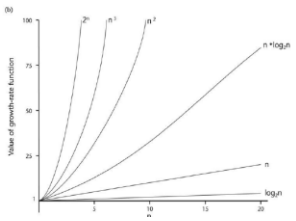
COMPORTAMENTO ASINTOTICO:

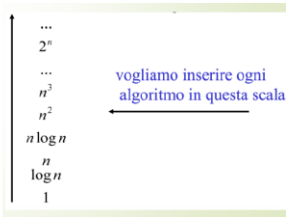
Un'altra caratteristica importante dei modelli e quello di dover avere un'analisi del comportamento asintotico, cioè il risultato che più ci interessa è quello che si ottiene per n (input) molto grandi, questo è dovuto al fatto che per valori piccoli di n il tempo richiesto e comunque basso, quindi qualunque algoritmo va bene, invece giudicare un algoritmo comporta dare un giudizio sull'efficienza dell'algoritmo per valori grandi di n, questo tipo di analisi viene detta **analisi del comportamento asintotico**.

Se ci concentriamo sul comportamento di un algoritmo al crescere della dimensione n dei dati all'infinito, allora quello a cui arriveremo è una situazione in cui classificheremo i nostri algoritmi in classi di complessità, quindi arriveremo a trascurare tutte le costanti moltiplicative ed additive, e invece faremo riferimento soltanto alla funzione matematica che mette in relazione il tempo di esecuzione con la taglia n dell'input.

Le classi di complessità più frequenti sono:

a	costante
a n+b	lineare
$a\,n^2 + bn + c$	quadratica
$a \log_g n + h$	logaritmica
a^n	esponenziale
n^n	esponenziale





Facciamo delle considerazioni, una prima cosa è che per piccole dimensioni dell'input tutti gli algoritmi hanno tempi di risposta non significativamente differenti, questo è il motivo per il quale le analisi si fanno sempre per n molto grandi quindi in tal modo possiamo analizziamo il comportamento asintotico.

NOTAZIONE O E Ω:

Utilizziamo un modello che è molto famoso nell'analisi degli algoritmi che si chiama **notazione asintotica** chee utilizza delle funzioni **Omicron** e **Omega**.

Consideriamo due funzioni f e g definite sui numeri naturali e con valori nel campo reale diremo che:

- f(n) è **O** di g(n), oppure f(n)∈**O**(g(n)), se esistono due costanti positive c ed n_0 tali che se n>= n_0, f(n) <= c g(n).
  Applicata alla funzione di complessità f(n), la notazione O ne limita superiormente la crescita e fornisce quindi una indicazione della bontà dell'algoritmo.
- f(n) è **Omega** di g(n), f(n)∈**Ω**(g(n)), se esistono due costanti positive c ed n_0 tali che se n>= n_0, c g(n) <= f(n).
  La notazione **Ω** limita inferiormente la complessità, indicando così che il comportamento dell'algoritmo non è migliore di un comportamento assegnato.

REGOLE PER LA VALUTAZIONE DELLA COMPLESSITÀ:

Una ***prima regola*** è la **scomposizione**, supponiamo che:

- alg è la sequenza di alg1 ed alg2;
- alg1 è O(g1(n));
- alg2 è O(g2(n));
- alg è O(max(g1(n),g2(n))).

A prevalere sarà quello che ha la complessità più grande.

```
i=0;
while(i<n) {                    g1(n)
  Stampastelle(i);
  i=i+1;
}
for(i=0; i<2*n; i++)            g2(n)
  scanf("%d", &numero);
```

Una ***seconda regola*** è quella dei **blocchi annidati**, supponiamo che:

- alg è composto da due blocchi annidati;
- blocco esterno è O(g1(n));
- blocco interno è O(g2(n));
- alg è O(g1(n)*g2(n)).

A prevalere sarà la moltiplicazione delle due complessità.

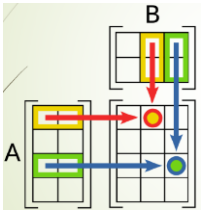
```
for(i=0; i<n; i++) {            g1(n)
  scanf("%d", &j);
  printf("%d", j*j);
  do {                          g2(n)
    scanf("%d", &numero);
    j=j+1;
  } while (j<=n);
```

Esempio calcolo complessità:

Consideriamo l'algoritmo per il prodotto di due matrici, ricordiamo che per poter applicare il prodotto di due matrici è necessario che il numero di colonne della prima matrice sia pari al numero di righe della seconda, in questo esempio abbiamo quindi una matrice A con N righe ed M colonne di cui vogliamo fare il prodotto con una matrice B con M righe e P colonne, il risultato sarà una matrice C con N righe e P colonne. Per poter effettuare il prodotto dobbiamo calcolare il risultato che metteremo in ciascuna cella della matrice, in particolare ciascuna cella conterrà una somma di prodotti.

Abbiamo bisogno di tre cicli iterativi un ciclo più esterno che scorre le righe della prima matrice, un ciclo intermedio che scorre le colonne della seconda matrice e un terzo ciclo più interno che invece calcolerà la somma di prodotti.

Prima della terza iterazione sarà necessario azzerare il contenuto della cella e quindi poi all'interno della terza iterazione andremo a sommare di volta in volta il prodotto ottenuto in questa iterata.



Analizziamo la complessità, abbiamo tre blocchi annidati quindi possiamo applicare la regola in cui per ottenere la complessità di un programma in cui ci sono dei blocchi annidati basta moltiplicare la complessità dei blocchi ed è quello che facciamo anche qui. Vediamo che il blocco più esterno viene fatto un numero di volte che è O(N), il secondo blocco viene fatto un numero di volte che è O(P), il terzo blocco viene eseguito un numero di volte pari a O(M). Quindi in totale dobbiamo moltiplicare questi valori e abbiamo che la complessità asintotica del programma è O(N*P*M).

```
float A[N][M], B[M][P], C[N][P];
int i, j, k;

for(i=0; i<N; i++)              O(N)
  for(j=0; j<P; j++) {          O(P)
    C[i][j]=0;
    for(k=0; k<M; k++)          O(M)
      C[i][j]+=A[i][k] * B[k][j];
  }
```

Una ***terza regola*** è quella di avere dei **sottoprogrammi ripetuti**:

- alg applica ripetutamente un certo insieme di istruzioni la cui complessità all'i-esima esecuzione vale $f_i(n)$;
- il numero di ripetizioni è g(n);
- alg è O($\sum_{i=1}^{g(n)} fi\,(n)$);
- per $f_i(n)$ tutte uguali … O(g(n) f(n)). *Esempio: Insertion Sort*.

Una ***quarta regola*** è l'**operazione dominante**:

- Sia f(n) il costo di esecuzione di un algoritmo alg;
- Un'istruzione i è dominante se viene eseguita g(n) volte, con f(n) <= a g(n);
- Se un algoritmo ha una operazione dominante allora è O(g(n)). *Esempio: ordinamento, il confronto tra due elementi dell'array è dominante.*

COMPLESSITÀ DEI PROBLEMI:

Abbiamo parlato finora di complessità degli algoritmi, un discorso invece un po' più complesso riguarda la complessità dei problemi.

Studiare la complessità di un problema è molto diverso da studiare la complessità di un algoritmo, la relazione tra problema e algoritmo è di uno a molti, cioè un problema può essere risolto da diversi algoritmi, ciascuno dei quali può avere una sua complessità.

Trovare un limite superiore alla complessità del problema è abbastanza semplice, per porre un limite superiore a tempo di esecuzione di un problema basta trovare un algoritmo che lo risolva con quella complessità, se troviamo ad esempio un algoritmo che ha complessità O(g(n)) e che risolvo il dato problema, allora possiamo dire che quel problema a complessità O(g(n)).

Porre invece un limite inferiore al tempo di esecuzione di un problema è molto più difficile, perché una volta affermato che quel problema ha un dato tempo di esecuzione, che può essere Ω(g(n)), vuol dire che non esiste un algoritmo che lo risolve meglio di quella complessità.

Questo potrebbe riuscire abbastanza difficile da dimostrare poiché quell'algoritmo potrebbe esistere ma non è stato ancora trovato, quindi per trovare un limite inferiore dobbiamo attuare delle strategie e fare delle dimostrazioni matematiche che ci consentono di stabilire che quello è il limite inferiore non si può fare di meglio.

Esistono delle **strategie** che ci consentono di individuare i limiti inferiori, una di queste e analizzare la **taglia dei dati**.

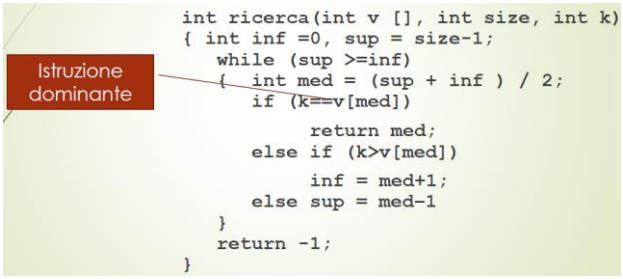
Supponiamo che la taglia sia di dimensione n, se sappiamo che l'algoritmo analizza tutto l'input allora chiaramente l'algoritmo sarà Ω(n), come esempio possiamo citare la ricerca di un elemento o del massimo all'interno di un Array, chiaramente questi algoritmi hanno complessità lineare poiché sono costretti ad analizzare l'intero input, è possibile comunque trovare dei limiti inferiori più stretti, quindi analizzando la taglia dell'input troviamo un limite inferiore ma non è detto che sia un limite inferiore stretto.

Un'altra tecnica per individuare un limite inferiore è quella degli **eventi contabili**.

Se ad esempio sappiamo che per risolvere un problema dobbiamo ripetere un evento un certo numero di volte allora sappiamo che il numero di volte che questo evento viene ripetuto sarà sicuramente un limite inferiore alla complessità del problema, come esempio consideriamo il problema di generare tutte le permutazioni di n oggetti, siamo costretti a fare l'operazione di generazione un numero di volte che è pari a tutte le permutazioni degli oggetti che sappiamo essere n!.

Tornando al problema dell'ordinamento sappiamo che l'evento che avviene più spesso è quello del confronto e si può dimostrare matematicamente che tutti gli algoritmi che operano per confronti devono effettuare questa operazione almeno n log n volte, quindi Ω(n log n), è un limite inferiore al problema dell'ordinamento.

Vediamo un esempio in cui applichiamo l'**istruzione dominante**, consideriamo il problema della ricerca binaria che abbiamo già risolto iterativamente:



Nella ricerca binaria abbiamo un'istruzione dominante che è quella della verifica se l'elemento che stiamo cercando è uguale all'elemento corrente, quindi l'operazione dominante sarà un confronto.

Dobbiamo capire quante volte viene eseguito questo confronto, osserviamo che la dimensione del problema si dimezza ad ogni ciclo, inizialmente è n poi abbiamo n mezzi, poi n quarti e così via, ci fermeremo quando la dimensione del problema diventa 1, a quel punto avremo fatto circa log n iterazioni. Avendo fatto quindi un numero costante di confronti per ogni iterazione, il numero massimo di confronti sarà O(log n) quindi il tempo di esecuzione della nostra funzione sarà O(log n).

VALUTAZIONE COMPLESSITÀ DELLE FUNZIONI RICORSIVE:

Negli algoritmi ricorsivi la soluzione di un problema si ottiene applicando lo stesso algoritmo ad uno o più sotto problemi che saranno di taglia inferiore a quella iniziale. Possiamo esprimere la complessità di questi algoritmi nella forma di una relazione di ricorrenza, abbiamo un albero decisionale in cui possiamo valutare alcune caratteristiche dell'algoritmo per decidere la sua complessità, la prima cosa che osserviamo è il lavoro di combinazione, cioè quel lavoro che fa l'algoritmo per preparare le chiamate ricorsive e poi il lavoro che farà dopo aver fatto le chiamate ricorsive per rielaborare i risultati ottenuti. Nell'albero decisionale che vedremo considereremo soltanto il caso in cui il lavoro di combinazione possa essere costante oppure lineare, quindi abbiamo due scelte, poi, avanzati di un livello, andremo a vedere la forma dell'equazione di ricorrenza che può essere con o senza partizione dei dati, ed infine l'ultima cosa che andremo a valutare sarà il numero di termini ricorsive, cioè il numero di chiamate ricorsive nella funzione. A questo punto aggiungeremo un valore che ci esprime la complessità del nostro algoritmo ricorsivo. L'albero decisionale sarà costruito in questo modo:

1. Lavoro di combinazione costante
   a) $T(n) = a_1 T(n-1) + a_2 T(n-2) + \dots a_h T(n-h) + b$ per $n > h$
      - Esponenziale con n: se sono presenti almeno 2 termini (l'algoritmo contiene almeno 2 chiamate ricorsive)
      - Lineare con n: se è presente un solo termine (singola chiamata ricorsiva)
   b) $T(n) = a\,T(n/p) + b$ per $n > 1$
      - log n se a = 1 (singola chiamata ricorsiva)
      - $n^{\log_p a}$ se a > 1 (più chiamate ricorsive)
2. Lavoro di combinazione lineare
   a) $T(n) = T(n-h) + b\,n + d$ per $n > h$
      Quadratico con n
   b) $T(n) = a\,T(n/p) + b\,n + d$
      - Lineare con n se $a < p$
      - n log n se $a = p$
      - $n^{\log_p a}$ se $a > p$