Lezione_04_DeA

2.4.6. Esercizi

2.4.6.1. Analisi complessità in tempo

Sia A un algoritmo che ricevuto in input un array X di $n \geq 1$ interi esegue

- $c_1 n$ operazioni per ogni intero pari in X
- $c_2 \lceil \log_2 n \rceil$ operazioni per ogni intero dispari in X
  dove c_1 e c_2 sono costanti positive.
  Analizzare la complessità in tempo dell'algoritmo esprimendola con $\Theta(\cdot)$.

Per un'istanza particolare (es: array di tutti interi pari) ho $\Theta(n^2)$, ho quindi ricavato un lower bound.

- Taglia dell'istanza: n
- Complessità: $t_A(n)$
- Upper bound: $\forall$ istanza di taglia n, A esegue
  $\leq n\max\{c_1 m, c_2\lceil \log_2 n \rceil\} \leq n\max\{c_1, c_2\}\max\{n, \log_2 n\} = n\max\{c_1, c_2\}n$ operazioni
  $\implies t_A(n) \in O(n^2)$ **(1)**
- Lower bound basato su una particolare istanza:
  Sia X un array di interi pari $\implies$ per tale X, A esegue
  $n \cdot c_1 \cdot n = c_1 n^2$ operazioni $\implies t_A(n) \in \Omega(n^2)$ **(2)**
- Lower bound basato su tutte le istanze:
  $\forall$ istanza di taglia n, A esegue
  $\geq n\min\{c_1 n, c_2\lceil \log_2 n \rceil\} \geq n\min\{c_1, c_2\}\min\{n, \log_2 n\} = n\min\{c_1, c_2\}\lceil \log_2 n \rceil$
  operazioni $\implies t_A(n) \in \Omega(n \log n)$
  Questo lower bound è peggiore rispetto a quello trovato prima.

2.4.6.1. Descrizione insertion sort

Il seguente pseudocodice descrive l'algoritmo di ordinamento chiamato `InsertionSort` (Si assume che la sequenza S da ordinare sia una variabile globale che dopo la fine dell'algoritmo è accessibile e ordinata).

Algoritmo `InsertionSort(S)`
Input: Sequenza $S[0 \ldots n-1]$ di n chiavi.
Output: Sequenza S ordinata in senso crescente.

```
for i<-1 to n-1 do{
     curr<-S[i];
     j<-i-1;
     while((j>=0) AND (S[j] > curr)) do{
          S[j+1]<-S[j];
          j<-j-1;
     }
     S[j+1]<-curr;
}
```

Notazione

Nel caso degli algoritmi di ordinamento, in analogia al libro di testo, usiamo il termine *sequenza* per denotare la collezione di elementi da ordinare che assumiamo rappresentata come array.

1. Trovare opportune funzioni $f_1(n)$, $f_2(n)$, $f_3(n)$ tali che le seguenti affermazioni siano vere, per una qualche costante $c > 0$ e per n abbastanza grande.
   - Per ciascuna istanza di taglia n l'algoritmo esegue $\leq cf_1(n)$ operazioni.
   - Per ciascuna istanza di taglia n l'algoritmo esegue $cf_2(n)$ operazioni.
   - Esiste un'istanza di tagli a n per la quale l'algoritmo esegue $\geq cf_3(n)$ operazioni

     La funzione $f_1(n)$ deve essere la più piccola possibile, mentre le funzioni $f_2(n)$ e $f_3(n)$ devono essere le più grandi possibili.
2. Sia $t_{IS}(n)$ la complessità al caso pessimo dell'algoritmo. Sfruttando le affermazioni del punto precedente trovare un upper bound $O(\cdot)$ e un lower bound $\Omega(\cdot)$ per $t_{IS}(n)$.

2.4.7. Regola di buon senso

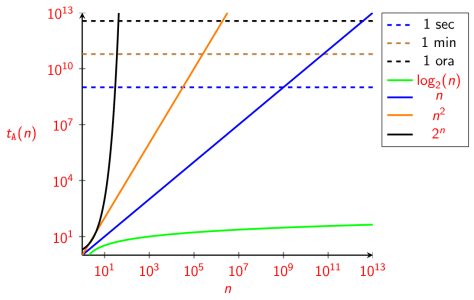
Una complessità polinomiale (o migliore) implica un algoritmo efficiente, mentre una complessità esponenziale implica un algoritmo inefficiente.
Supponiamo che la complessità $t_A(n)$ sia espressa in nanosecondi e definiamo

$$n_\tau \equiv \text{max taglia di un'istanza risolvibile in tempo } \tau$$

Per ottenere n_τ, risolviamo $t_A(n_\tau) = \tau$ rispetto a n_τ.

2.4.7.1. Esempi

$t_A(n)$	n_τ **per** $\tau = 10^9$ **(1 sec)**	n_τ **per** $\tau = 60 * 10^9$ **(1 min)**	n_τ **per** $\tau = 3600 * 10^9$ **(1 ora)**
$\log_2 n$	$2^{10^9} = \infty$	∞	∞
n	10^9	$6 * 10^{10}$	$3.6 * 10^{12}$
n^2	$10^{4.5}$	$\approx 8 * 10^{4.5}$	$\approx 1.8 * 10^6$
2^n	≈ 30	≈ 36	≈ 42



2.4.7.2. Esempio

La crittografia a chiave pubblica è molto usata dai protocolli di sicurezza.
L'invio di un messaggio cifrato da Alice a Bob funziona (in maniera approssimata) in questo modo:

- Bob possiede una chiave privata k_1 e una chiave pubblica k_2;
- Alice invia a Bob un messaggio m cifrato con k_2;
- Bob decifra il messaggio ricevuto da Alice con k_1;

L'*Algoritmo RSA* sviluppato nel 1977 da Rivest, Shamir, Adleman (*Turing Award 2002*) è il più famoso algoritmo di crittografia a chiave pubblica. La sua "sicurezza" si basa sulla difficoltà di risolvere il seguente problema.

2.4.7.2.1. Integer factorization (per interi prodotti di 2 primi)

> Dato un intero N prodotto di due primi p, q, determinare p e q.

Che taglia dell'istanza scelgo?

Posso scrivere, ad esempio, questo algoritmo banale per risolvere il problema:

```
for p<-2 to floor{sqrt{N}} do{
    if(N mod p = 0) then return {p, N/p};
}
```

Esprimiamo la complessità in funzione del numero n bit che servono per rappresentare N. Se $p, q \in \Theta(\sqrt{N})$ la complessità è $\Theta(\sqrt{N}) = \Theta(2^{n/2})$

2.4.7.2.1.1. Esempio numerico

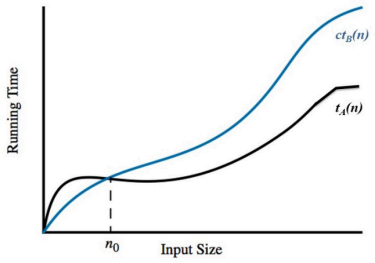
Prendendo $n = 1024$ ottengo $2^{n/2} = 2^{512} \geq 10^{154}$. Sul computer più potente del mondo ($< 10^{19}$ flop/sec) l'algoritmo banale richiederebbe circa $\frac{10^{154}}{10^{19}} = 10^{135}$ secondi. Considerando che un anno ha meno di 10^8 secondi, sarebbero più di 10^{127} anni di calcolo.

> **Nota bene**
>
> Esistono algoritmi più efficienti, ma sempre con complessità esponenziale.

> **Dalla motivazione del A.M. Turning Award 2002 assegnato a Rivest, Shamir e Adleman**
>
> RSA is used in almost all internet-based commercial transactions. Without it, commercial online activities would not be as widespread as they are today. It allows users to communicate sensitive information like credit card numbers over an unsecure internet without having to agree on a shared secret key ahead of time. Most people ordering items over the internet don't know that the system is in use unless they notice the small padlock symbol in the corner of the screen. RSA is a prime example of an abstract elegant theory that has had great practical application.

2.4.8. Efficienza asintotica degli algoritmi

Dati A, B due algoritmi che risolvono il problema computazionale Π, $t_A(n)$ e $t_B(n)$ sono le complessità rispettivamente di A e B al caso pessimo. Se $t_A(n) \in o(t_B(n))$, allora A è "*asintoticamente più efficiente*" di B.



Nota bene: n_0 potrebbe essere *molto* grande.

> **Caveat sull'analisi asintotica al caso pessimo**
>
> 1. *Le costanti trascurate potrebbero essere elevate* e, in pratica, potrebbero avere un impatto elevato sulle prestazioni.
> 2. *Il caso pessimo potrebbe essere costituito solo da* ***istanze patologiche*** mentre per tutte le istanze di interesse la complessità potrebbe essere asintoticamente migliore.
>   Cosa fare in questo caso?
>   - Restringere il dominio delle istanze, mantenendo quelle di interesse ed escludendo quelle patologiche.
>   - Fare un'analisi al caso medio.

2.5. Analisi di correttezza

2.5.1. Induzione

Per provare che una proprietà $Q(n)$ è vera $\forall n \geq n_0$ si procede così:

- Si sceglie un intero $k \geq 0$;
- **Base:** si dimostra $Q(n_0)$, $Q(n_0+1)$, ..., $Q(n_0+k)$;
- **Passo induttivo:** si fissa un valore $n \geq n_0+k$ *arbitrario* e si dimostra che $Q(m)$ vera $\forall m : n_0 \leq m \leq n \implies Q(n+1)$ vera.

> **Osservazioni**
>
> - "$Q(m)$ vera $\forall m : n_0 \leq m \leq n$" è chiamata *ipotezi induttiva*.
> - La dimostrazione deve valere *per ogni* $n \geq n_0+k$
> - Di solito, ma non sempre, $k=0$ (il libro di testo descrive l'induzione con $n_0=1$).

2.5.1.1. Esempio

$$Q(n) : \sum_{i=0}^{n} i = \frac{n(n+1)}{2} \quad \forall n \geq 0$$

Nota bene: Implica che $Q(n) \in \Theta(n^2)$.

- $n_0 = 0, \quad k = 0$
- Base: $Q(0) : \sum_{i=0}^{0} i = 0 = \frac{0 \cdot 1}{2}$ ✓
- Passo induttivo: Fisso $n \geq n_0 + n = 0$ arbitrario.
  Ipotesi induttiva: $\sum_{i=0}^{m} i = \frac{m(m+1)}{2} \quad \forall 0 \leq m \leq n$
  Dimostriamo che $Q(n+1)$ è vera:
  $\sum_{i=0}^{n+1} i = n + 1 + \sum_{i0}^{n} i = n + 1 \frac{n(n+1)}{2} =$ per ipotesi induttiva
  $= \frac{(n+1)(n+2)}{2} \implies Q(n+1)$ è vera

2.5.1.2. Esercizio

Dimostrare per induzione la seguente proprietà:

$$Q(n) : \sum_{i=0}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} \quad \forall \geq 0$$

Nota bene: Implica che $\sum_{i=0}^{n} i^2 \in \Theta(n^3)$

> **Osservazione**
>
> In generale $\sum_{i=0}^{n} i^k \in \Theta(n^{k+1})$, con k costante.