Rudimenti di Complessità

Corso di Programmazione II

Prof. Dario Catalano

Analisi di Complessità

- **Costo di un algoritmo** è in funzione di n (dimensione dei dati in input):
  - **tempo** = numero di **operazioni RAM** eseguite
  - **spazio** = numero di **celle di memoria** occupate (escluse quelle per contenere l'input)

Caso pessimo e caso medio

Complessità o costo computazionale f(n) in tempo e in spazio di un problema Π:

- **caso pessimo o peggiore** = costo **max** tra **tutte** le istanze di Π aventi dimensioni dei dati pari a n
- **caso medio** = costo **mediato** tra tutte le **istanze** di Π aventi dimensioni pari a n

IF (guardia) {blocco 1} else {blocco 2}

costo(guardia)+max{costo(blocco 1),costo(blocco 2)}

For (i=0; i<m; i++) {corpo}

$$\sum_{i=0}^{m-1} t_i$$

t_i = costo di **corpo** all'iterazione i

---

while (guardia) {corpo}

do {corpo} while (guardia)

$$\sum_{i=0}^{m} (t_i' + t_i)$$

t_i' = costo di **guardia** all'iterazione i

t_i = costo di **corpo** all'iterazione i

- Il costo di una funzione è dato dal costo del suo corpo (più il passaggio dei parametri)
  - Per le funzioni ricorsive le cose sono più complicate
- Il costo di una sequenza di istruzioni è la somma dei costi delle istruzioni nella sequenza

Funzione esponenziale

$f(n)=b^n$

Regole

- $(b^a)^c=b^{ac}$
- $b^a\ b^c=b^{a+c}$
- $b^a/b^c=b^{a-c}$

Somma Geometrica

$$\sum_{i=0}^{n} a^i = 1 + a + a^2 + \cdots + a^n = \frac{a^{n+1} - 1}{a - 1} \quad (a > 1)$$

Funzione costante

$f(n)=c$ c costante

Funzione logaritmica

$f(n)=\log_b n$ $(b>1)$

1. $x=\log_b (n) \Leftrightarrow b^x=n$
2. $\log_b(1)=0$

Funzione lineare

$f(n)=cn$ c costante (c non nulla)

Funzione nlog n

$f(n)=n \log n$

Funzione quadratica

$f(n)=c\ n^2$

```
for (i=0;i<n;i++)
  for(j=0;j<n;j++)
    do something
```

```
for (i=1;i<=n;i++)
  for(j=1;j<=i;j++)
    do something
```

Funzione cubica

$f(n)=c\ n^3$

```
for (i=0;i<n;i++)
  for(j=0;j<n;j++)
    for(k=0;k<n;k++)
        do something
```

Sommatorie

$$\sum_{i=a}^{b} f(i) = f(a) + f(a+1) + \cdots + f(b)$$

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Sommatorie – II

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

$$\sum_{i=1}^{n} i^3 = \frac{n^2(n+1)^2}{4}$$

Esempio

```
for(i=1;i<=n;i++)
   for(j=1;j<=i;j++)
      for(k=1;k<=j;k++)
          do something
```

Funzione polinomiale

$f(n)=a_0 + a_1 n + a_2 n^2 + \ldots + a_d n^d$

il grado è il valore della potenza più grande con a_d diverso da 0

Notazione Asintotica

Notazione asintotica al crescere di n:

g(n) = O(f(n)) sse esistono $c, n_0 > 0$:

$g(n) \leq c\, f(n)$ per ogni $n > n_0$

g(n) = Ω(f(n)) sse esistono $c, n_0 > 0$:

$g(n) \geq c\, f(n)$ per infiniti valori di $n > n_0$

g(n) = Θ(f(n)) sse g(n) = O(f(n)) e f(n) = O(g(n))

Limiti Superiori e Inferiori

- Per un dato problema Π consideriamo un algoritmo A che lo risolve
- Se A prende tempo t(n) diremo che O(t(n)) è un **limite superiore.**
- Se riusciamo a provare che nessun algoritmo può far meglio di t(n) diremo che Ω(t(n)) è un **limite inferiore**.
- A è ottimo se i due limiti coincidono
  - In tal caso la complessità computazionale *del problema* è Θ(t(n)).

Problemi intrattabili

- Supponiamo di avere un algoritmo il cui tempo di calcolo sia $O(2^n)$ (es $2^n - 1$) (1 operaz/sec)

n	5	10	15	20	25	30	35	40
tempo	31 s	17 m	9 h	12 g	1 a	34 a	1089 a	34865 a

- Migliorare di un fattore moltiplicativo **N** (**N** operaz/sec) migliora le prestazioni solo di + **log N**
- Se con una macchina da **1 op/se**c riesco a processare input di dimensione **X** input in tempo **T,** con una da **1000 op/sec** posso arrivare a **X+10** in tempo **T**)

Problemi trattabili

- Supponiamo di avere un algoritmo il cui tempo di calcolo sia $O(n^2)$ (es n^2) (1 operaz/sec)

n	5	10	15	20	25	30	35	40
tempo	25 s	100 s	225 s	7 m	11 m	15 m	21 m	27 m

- Migliorare di un fattore moltiplicativo **N** (**N** operaz/sec) migliora le prestazioni di un fattore **moltiplicativo √ N**
- Se con una macchina da **1 op/se**c riesco a processare input di dimensione **X** input in tempo **T,** con una da **1000 op/sec** posso arrivare a **10X** in tempo **T**)

Esempio: calcolo di potenze

Versione ricorsiva

$$\text{power}(x,n) = \begin{cases} 1 & \text{se } n=0 \\ x*\text{power}(x,n-1) & \text{altrimenti} \end{cases}$$

```
power(x,n)
   if (n=0) return 1;
   else return x*power(x,n-1)
```

- n chiamate ricorsive
  - tempo e spazio O(n)

Esempio: calcolo di potenze

Versione ricorsiva

$$power(x,n) = \begin{cases} 1 & \text{se } n=0 \\ x*power(x,(n-1)/2)^2 & \text{se } n \text{ è dispari} \\ power(x,n/2)^2 & \text{se } n \text{ è pari} \end{cases}$$

- log n chiamate ricorsive
- tempo e spazio O(log n)

```
power(x,n)
   if (n dispari)
   { y=power(x,(n-1)/2);
     return x*y*y;
   }
   else {   y=power(x,n/2);
            return y*y; }
```

Esempio: Segmenti di Somma Massima

- Segmento: sequenza di elementi consecutivi in un array a
  - a array di n interi
  - a[i,j] segmento se $0 \leq i \leq j \leq n-1$
- Determinare il segmento di somma massima
  - Banale se gli elementi sono tutti positivi (o tutti negativi)
  - A parità di somma si predilige il segmento più corto

Prima Soluzione

```
SommaMassima1 ( a )  // a contiene n elementi, almeno 1 positivo
    max = 0;
    For (i = 0; i < n; i = i+1) {
        For (j = i; j < n; j = j+1) {
            somma = 0;
            For (k = i; k <= j; k = k+1)
                somma = somma + a[k];
            if (somma > max) max = somma;
        }
    }
    return max;
}
```

Seconda Soluzione: Idee

- Una volta calcolata somma(a[i,j-1]) evitiamo di ripartire da capo per somma(a[i,j])
- Utlizziamo il fatto che

  somma(a[i,j])=somma(a[i,j-1]) + a[j]

- Questo ci permette di risparmiare un ciclo for
  - Dunque otteniamo una soluzione quadratica

Seconda Soluzione

SommaMassima2 (a) // *a contiene n elementi, almeno 1 positivo*

```
max = 0;
For (i = 0; i < n; i = i+1) {
    somma = 0;
    For (j = i; j < n; j = j+1) {
        somma = somma + a[j];
        if (somma > max) max = somma;
    }
}
return max;
```

Terza Soluzione

- Sfruttiamo meglio la struttura combinatoria del problema
- Abbiamo $O(n^2)$ possibili segmenti
- Tra i segmenti di eguale lunghezza solo uno può avere somma massima
  - I potenziali candidati sono quindi O(n)
  - Inoltre tali candidati sono tutti disgiunti.

Terza Soluzione

- Un segmento di somma massima a[i,j] deve avere le seguenti caratteristiche

1. Ogni prefisso di a[i,j] ha somma positiva (per ogni $i \leq k < j$)
2. Il segmento a[i,j] non può essere esteso a sinistra
   - Somma(a[k,i-1]) ≤ 0 per ogni $0 \leq k \leq i-1$

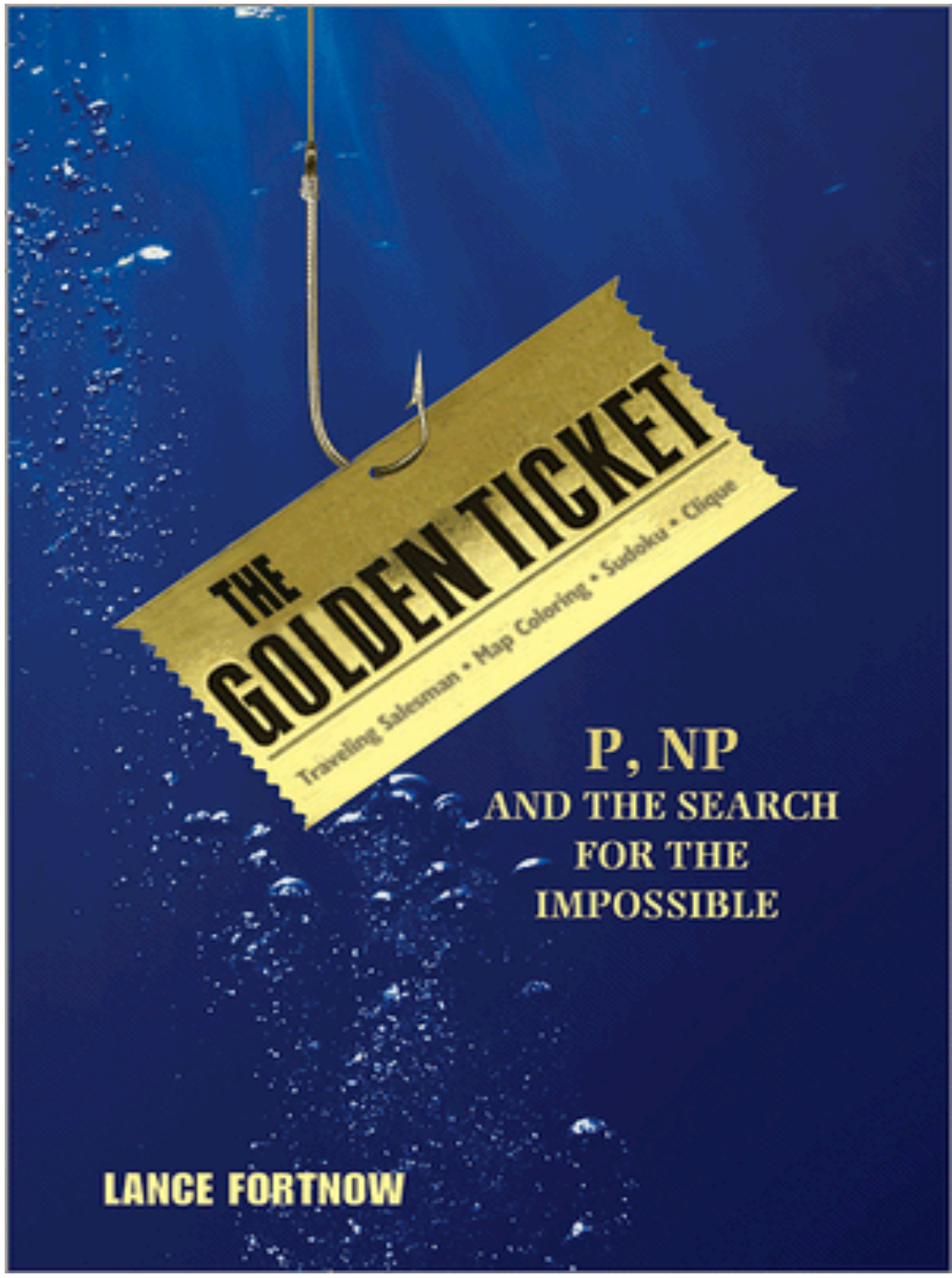
Terza Soluzione

```
SommaMassima3 ( a )  // a contiene n elementi, almeno 1 positivo
    max = 0;
    somma= max;
    For (j = 0; j < n; j = j+1) {
        if (somma > 0) {
              somma = somma + a[j];
        } else {
             somma = a[j];
        }
        if (somma > max) max = somma;
    }
    return max;
```

Commenti

- La soluzione proposta ha complessità O(n)
- Tale complessità è *asintoticamente ottima* (sia in termini di spazio che di tempo)
  - L'algoritmo deve poter leggere l'input
  - L'algoritmo utilizza solo un numero *costante* di locazioni per le variabili di appoggio.

Approfondimenti



Shtetl-Optimized

The Blog of Scott Aaronson

Quantum computers are not known to be able
to solve NP-complete problems in polynomial time,
and can be simulated classically with exponential slowdown.

« My daily dose of depression
More tender nuggets »

Logicians on safari

Sean Carroll, who many of you know from Cosmic Variance, asked the following question in response to my last entry:

> I'm happy to admit that I don't know anything about "one-way functions and interactive proofs." So, in what sense has theoretical computer science contributed more in the last 30 years to our basic understanding of the universe than particle physics or cosmology? (Despite the fact that I'm a cosmologist, I don't doubt your statement — I'd just like to be able to explain it in public.)