Algoritmi e Strutture Dati

Complessità di un
algoritmo e di un problema

Soluzione domanda di approfondimento

- Qual è la complessità temporale degli algoritmi `Fibonacci6`, `Fibonacci4` e `Fibonacci2` in funzione della rappresentazione dell'input?
- Abbiamo detto che la complessità temporale viene misurata in funzione della dimensione dell'input; nel caso dei tre algoritmi in questione, l'input è un numero n, che può essere rappresentato usando $k=\log n$ bit. Quindi:
  - `Fibonacci6` costa $T(n)=\Theta(\log n)=\Theta(k)$, ed è quindi **polinomiale** (più precisamente, **lineare**) nella dimensione dell'input;
  - `Fibonacci4` costa $T(n)=\Theta(n)=\Theta(2^k)$, ed è quindi **esponenziale** nella dimensione dell'input;
  - `Fibonacci2` costa $T(n)=\Theta(\phi^n)=\Theta(\phi^{2^k})$, ed è quindi **superesponenziale** nella dimensione dell'input.

Caso peggiore, migliore e medio

- Come detto, misureremo le risorse di calcolo usate da un algoritmo in funzione della dimensione delle istanze
- Ma istanze diverse, a parità di dimensione, potrebbero richiedere risorse diverse! Ad esempio, se devo cercare un elemento x in un insieme di n elementi in input, il numero di confronti che farò dipenderà dalla posizione che x occupa nella sequenza.
- Distinguiamo quindi ulteriormente tra analisi nel caso peggiore, migliore e medio

Caso peggiore

- Sia tempo(I) il tempo di esecuzione di un algoritmo sull'istanza di input I

$$T_{worst}(n) = \max_{\text{istanze I di dimensione n}} \{\text{tempo}(I)\}$$

- Intuitivamente, $T_{worst}(n)$ è il tempo di esecuzione sulle istanze di ingresso che comportano più lavoro per l'algoritmo
- Definizione analoga può essere data per lo spazio

Caso migliore

- Sia tempo(I) il tempo di esecuzione di un algoritmo sull'istanza I

$$T_{best}(n) = \min_{\text{istanze I di dimensione n}} \{\text{tempo}(I)\}$$

- Intuitivamente, $T_{best}(n)$ è il tempo di esecuzione sulle istanze di ingresso che comportano meno lavoro per l'algoritmo
- Definizione analoga può essere data per lo spazio

Caso medio

- Sia $\mathcal{P}(I)$ la probabilità di occorrenza dell'istanza I

$$T_{avg}(n) = \sum_{\text{istanze I di dimensione n}} \{ \mathcal{P}(I) \text{ tempo}(I) \}$$

- Intuitivamente, $T_{avg}(n)$ è il tempo di esecuzione nel caso medio, ovvero il tempo di esecuzione atteso
- Può essere difficile da valutare: richiede di conoscere una distribuzione di probabilità sulle istanze, ed inoltre bisogna saper risolvere in forma chiusa una sommatoria
- Definizione analoga può essere data per lo spazio

Complessità temporale e spaziale di un algoritmo

- Denoteremo con T(n) il tempo di esecuzione dell'algoritmo su una generica istanza di ingresso di dimensione n. Varrà quindi:

$$T(n) \leq T_{worst}(n)$$

$$T(n) \geq T_{best}(n)$$

- Analogamente, per l'occupazione di memoria:

$$S(n) \leq S_{worst}(n)$$

$$S(n) \geq S_{best}(n)$$

Convenzioni

- Se scriverò che un algoritmo ha complessità $T(n) = O(g(n))$, intenderò che su ALCUNE istanze costerà $\Theta(g(n))$, ma sulle rimanenti costerà $o(g(n))$: questo accade quando il caso peggiore e il caso migliore hanno costi asintoticamente diversi
- Se scriverò che un algoritmo ha complessità $T(n)=\Theta(g(n))$, intenderò che su TUTTE le istanze costerà $\Theta(g(n))$: questo quindi accade quando il caso peggiore e il caso migliore hanno costi asintoticamente identici
- Infine, come detto precedentemente, da ora in avanti in $T(n)$ conteggerò esclusivamente le operazioni dominanti dell'algoritmo

Un caso di studio: il problema della ricerca

Sia dato un mazzo di n carte scelte in un universo U di $2n$ carte distinte, e si supponga di dover verificare se una certa carta $x \in U$ appartenga o meno al mazzo. Progettare un algoritmo per risolvere tale problema, e analizzarne il costo (in termine di numero di confronti) nel caso migliore, peggiore e medio.

Algoritmo di ricerca sequenziale

Un primo algoritmo è quello di ricerca sequenziale (o esaustiva), che gestisce il mazzo di carte come una lista L non ordinata

```
    algoritmo ricercaSequenziale(lista L, elem x) → booleano
1.      for each ( y ∈ L ) do
2.          if ( y = x ) then return trovato
3.      return non trovato
```

Contiamo il numero di confronti (istruzione 2, **operazione dominante**):

$T_{best}(n) = 1$ x è in prima posizione

$T_{worst}(n) = n$ $x \notin L$ oppure è in ultima posizione

$T_{avg}(n) = P[x \notin L] \cdot n + P[x \in L$ e sia in prima posizione$] \cdot 1 + P[x \in L$ e sia in seconda posizione$] \cdot 2 + \ldots + P[x \in L$ e sia in n-esima posizione$] \cdot n$

Nel caso del mazzo di carte…

- Assumendo che le istanze siano equidistribuite, la probabilità che una carta appartenga (o non appartenga) al mazzo è ½, e la probabilità che l'elemento appartenga al mazzo e sia in posizione i-esima è ½·1/n

$$\Rightarrow T_{avg}(n) = ½\cdot n + ½\cdot 1/n\cdot 1 + ½\cdot 1/n\cdot 2 + \ldots + ½\cdot 1/n\cdot n =$$

$$= ½\cdot n + ½\cdot 1/n\cdot [1+2+\ldots+n] = ½\cdot n + ½\cdot\ 1/n\ \cdot [n\cdot(n+1)/2] =$$

$$= ½\cdot n + (n+1)/4 = (3n+1)/4$$

$$\Rightarrow T_{avg}(n) = T_{worst}(n) = \Theta(n)$$

- Quindi, secondo la nostra convenzione, il costo dell'algoritmo di ricerca sequenziale è $T(n)=O(n)$ (infatti, $T_{best}(n) = 1$)
- Come vedremo con il prossimo algoritmo, l'analisi del caso medio può rivelarsi molto complicata…

Algoritmo di ricerca binaria

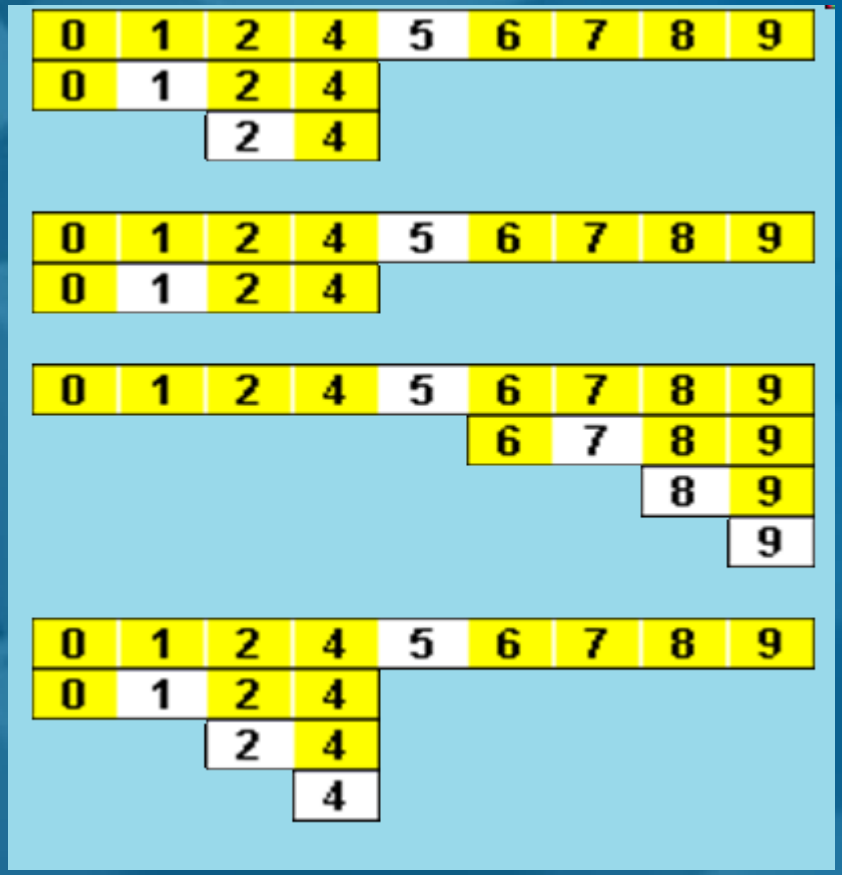
Se ipotizzassimo che il mazzo di carte fosse un array L ordinato, potremmo progettare un algoritmo più efficiente:

```
    algoritmo ricercaBinariaIter(array L, elem x) → booleano
1.      a ← 1
2.      b ← lunghezza di L
3.      while ( L[⌊(a + b)/2⌋] ≠ x ) do
4.          m ← ⌊(a + b)/2⌋
5.          if ( L[m] > x ) then b ← m − 1
6.          else a ← m + 1
7.          if ( a > b ) then return non trovato
8.      return trovato
```

Confronta x con l'elemento centrale di L e prosegue nella metà sinistra o destra in base all'esito del confronto

Approfondimento: dimostrare formalmente la correttezza dell'algoritmo.

Esempi su un array di 9 elementi



Cerca 2

Cerca 1

Cerca 9

Cerca 3

3<4 quindi a e b si invertono

Analisi dell'algoritmo di ricerca binaria

Contiamo i confronti eseguiti nell'istruzione 3 **(operazione dominante):**

$T_{best}(n) = 1$ l'elemento centrale è uguale a x

$T_{worst}(n) = \lfloor \log n \rfloor + 1 = \Theta(\log n)$ $x \notin L$

Infatti, poiché la dimensione del sotto-array su cui si procede si dimezza dopo ogni confronto, dopo l'i-esimo confronto il sottoarray di interesse ha dimensione $n/2^i$. Quindi, nel caso peggiore, dopo $i = \lfloor \log n \rfloor + 1$ confronti, si arriva ad avere $a > b$.

$T_{avg}(n) = P[x \notin L] \cdot (\lfloor \log n \rfloor + 1) + P[x \in L \text{ e sia in posizione centrale}] \cdot 1$
$+P[x \in L \text{ e sia in posizione centrale nelle 2 sottometà}] \cdot 2 +$
$+P[x \in L \text{ e sia in posizione centrale nelle 4 sotto-sottometà}] \cdot 3 + \ldots +$
$+P[x \in L \text{ e sia in una delle } 2^{\lfloor \log n \rfloor} \approx n/2 \text{ posizioni raggiungibili con a=b}] \cdot (\lfloor \log n \rfloor + 1)$

4	3	4	2	4	3	4	1	4	3	4	2	4	3	4

Nel caso del mazzo di carte…

Se il mazzo di carte ci venisse dato ordinato, applicando la ricerca binaria avremmo:

$$\Rightarrow \quad T_{avg}(n) = \tfrac{1}{2}\cdot(\lfloor \log n \rfloor + 1) + \tfrac{1}{2}\cdot 1/n \cdot 1 + \tfrac{1}{2}\cdot 2/n \cdot 2 + \tfrac{1}{2}\cdot 4/n \cdot 3 + \ldots$$
$$+ \tfrac{1}{2}\cdot 2^{\lfloor \log n \rfloor}/n \cdot (\lfloor \log n \rfloor + 1)$$

Questa sommatoria è di difficile risoluzione, e mi affido quindi all'analisi asintotica. Osservo che se $x \notin L$, ossia nella metà dei casi, pago $(\lfloor \log n \rfloor + 1)$, mentre nell'altra metà dei casi, ossia quando $x \in L$, pago un valore minore di $(\lfloor \log n \rfloor + 1)$

$$\Rightarrow T_{avg}(n) < \tfrac{1}{2}\cdot(\lfloor \log n \rfloor + 1) + \tfrac{1}{2}\cdot(\lfloor \log n \rfloor + 1) = \lfloor \log n \rfloor + 1$$

e poiché $T_{avg}(n) > 1/2 \cdot \lfloor \log n \rfloor$, ne consegue che $T_{avg}(n) = \Theta(\log n)$

$$\Rightarrow T_{avg}(n) = T_{worst}(n) = \Theta(\log n)$$

$\Rightarrow$ Anche in questo caso, scriveremo $T(n)=O(\log n)$ (infatti, $T_{best}(n)=1$)

Upper e lower bound di un problema

Delimitazione superiore e inferiore alla complessità di un problema

Definizione (*upper bound* di un problema): Un problema P ha una delimitazione superiore alla complessità $O(g(n))$ rispetto ad una certa risorsa di calcolo (spazio o tempo) se esiste un algoritmo (che quindi abbiamo già progettato) che risolve P e il cui costo di esecuzione (nel caso peggiore) rispetto a quella risorsa è $O(g(n))$ (ad esempio, nel caso della risorsa tempo, deve essere $T(n)=O(g(n))$).

Definizione (*lower bound* o *complessità intrinseca* di un problema): Un problema P ha una delimitazione inferiore alla complessità $\Omega(g(n))$ rispetto ad una certa risorsa di calcolo (spazio o tempo) se ogni algoritmo (anche quelli non ancora progettati!!!) che risolva P ha costo di esecuzione (nel caso peggiore) $\Omega(g(n))$ rispetto a quella risorsa (ad esempio, nel caso della risorsa spazio, deve essere $S(n)= \Omega(g(n))$).

Ottimalità di un algoritmo

Definizione: Dato un problema P con complessità intrinseca $\Omega(g(n))$ rispetto ad una certa risorsa di calcolo (spazio o tempo), un algoritmo che risolve P è ottimo (in termini di complessità asintotica, ovvero a meno di costanti moltiplicative e di termini additivi/sottrattivi di "magnitudine" inferiore) se ha costo di esecuzione $O(g(n))$ rispetto a quella risorsa, e quindi la sua complessità asintotica risulta la migliore possibile.

Obiettivo principale di un algoritmista: Dato un problema P, trovare un algoritmo ottimo (in genere rispetto alla risorsa **tempo**) che risolva P. Ciò può essere ottenuto dimostrando da un lato che il problema è intrinsecamente difficile (alzando il suo lower bound), e dall'altro progettando algoritmi sempre più efficienti (abbassando quindi il suo upper bound).

Convenzioni

- Da ora in poi, quando parlerò di upper bound di un problema, mi riferirò alla complessità temporale del MIGLIORE ALGORITMO che sono stato in grado di progettare sino a quel momento (ovvero, quello con minore complessità temporale nel caso peggiore).

- Da ora in poi, quando parlerò di lower bound di un problema, mi riferirò alla PIÙ GRANDE delimitazione inferiore alla complessità temporale del problema che sono stato in grado di dimostrare sino a quel momento.

Alcuni esempi

- L'algoritmo di ricerca sequenziale di un elemento in un insieme **non ordinato** di n elementi costa $T(n) = O(n)$, in quanto su alcune istanze costa $\Theta(n)$, mentre su altre costa $o(n)$
- Ovviamente, per quanto sopra, l'upper bound del problema della ricerca di un elemento in un insieme non ordinato di n elementi è pari a $O(n)$
- Si osservi ora che il lower bound del problema della ricerca di un elemento in un insieme non ordinato di n elementi è pari a $\Omega(n)$: infatti, ogni algoritmo di risoluzione deve per forza di cose guardare tutti gli elementi dell'insieme per decidere se l'elemento cercato appartiene o meno ad esso!

  ⇒ l'algoritmo di ricerca sequenziale è ottimo!
- Invece, l'algoritmo di ricerca binaria di un elemento in un insieme **ordinato** di n elementi ha complessità temporale $T(n) = O(\log n)$. Questo non è in contraddizione con il lower bound che ho appena dato, perché stiamo parlando di due problemi diversi: ricerca in un insieme ordinato oppure non ordinato. Come vedremo più avanti, anche l'algoritmo di ricerca binaria è ottimo per il problema della ricerca di un elemento in un insieme ordinato di n elementi, perché dimostreremo (con una tecnica non banale) che il lower bound di tale problema è $\Omega(\log n)$.

Alcuni esempi (2)

- L'algoritmo `Fibonacci4` per il calcolo dell'n-esimo numero della sequenza di Fibonacci costa $T(k=\log n) = \Theta(2^k=n)$, in quanto su tutte le istanze costa sempre $\Theta(2^k=n)$
- L'algoritmo `Fibonacci6` per il calcolo dell'n-esimo numero della sequenza di Fibonacci costa $T(k=\log n) = \Theta(k=\log n)$, in quanto su tutte le istanze costa sempre $\Theta(k=\log n)$
- Ovviamente, per quanto sopra, l'upper bound del problema del calcolo dell'n-esimo numero della sequenza di Fibonacci è pari a $O(\log n)$
- Si osservi ora che il lower bound del problema del calcolo dell'n-esimo numero della sequenza di Fibonacci è pari a $\Omega(1)$: infatti, sicuramente ogni algoritmo di risoluzione deve per forza di cose leggere l'input, al costo di 1 operazione (questo è il cosiddetto lower bound banale di ogni problema), ma non sono in grado di definire altre operazioni necessarie a tutti gli algoritmi!

  ➡ l'algoritmo `Fibonacci6` non è ottimo!