Algoritmi e Strutture di Dati

Complessità dei problemi

m.patrignani

Nota di copyright

Contenuto

- Definizioni
  - complessità $O(f(n))$, $\Omega(f(n))$ e $\Theta(f(n))$ di un problema
- Problemi e complessità
  - esempi di problemi di complessità ignota
  - lower bound per gli algoritmi di ricerca basati su confronti
  - lower bound per gli algoritmi di ordinamento per confronto

Problemi e complessità

- Sappiamo che
  - un algoritmo corretto per un problema computazionale è una "ricetta" per la sua soluzione
    - termina sempre
    - produce un output che, nella definizione del problema, corrisponde all'istanza in input
- Un problema ammette infiniti algoritmi corretti
  - di ogni algoritmo possiamo calcolare la complessità asintotica
- Alcuni problemi ammettono algoritmi più efficienti di altri problemi
  - i problemi hanno una complessità asintotica intrinseca?

Analisi della complessità dei problemi

- Obiettivo
  - classificare i problemi in base alla loro difficoltà di soluzione intrinseca
    - determinare la quantità di risorse che comunque è necessario spendere per risolverli
- Strumento
  - associare al problema la complessità dell'algoritmo più efficiente che lo risolve
- Inconveniente
  - dato un problema non è possibile considerare tutti gli infiniti algoritmi che lo risolvono
    - non possiamo determinare direttamente la complessità dell'algoritmo più efficiente

Complessità O($f(n)$) di un problema

- Un problema ha *complessità temporale O(f(n))* se **esiste** un algoritmo che lo risolve che ha complessità temporale O($f(n)$)
- In forma stenografica:

$$P \in O(f(n)) \Leftrightarrow \exists\ A \in O(f(n))$$

- O($f(n)$) sono le risorse *sufficienti* a risolvere il problema

Complessità O($f(n)$) di un problema

- Se un problema ha complessità temporale O($f(n)$)
  - è garantito che il problema possa essere risolto spendendo O($f(n)$) risorse
  - è possibile che il problema possa essere risolto spendendo meno di O($f(n)$) risorse
    - potrebbe esistere un algoritmo più efficiente che non conosciamo
  - $f(n)$ è un *limite superiore* (upper bound) alle risorse sufficienti a risolvere il problema
- Per dimostrare che un problema ha complessità O($f(n)$)
  - occorre produrre un algoritmo che lo risolva e che abbia complessità O($f(n)$)

Esempio: problema O($f(n)$)

```
SOMMA(A) ▷ restituisce la somma degli elementi dell'array A
1.  somma = A[0]
2.  for i = 1 to A.length-1
3.      somma = somma + A[i]
4.  return somma
```

- La complessità temporale dell'algoritmo `SOMMA` è O(n), dove n è il numero degli elementi dell'array A
- Il problema della somma di n interi
  - ha complessità temporale O(n)
  - è limitato superiormente da $f(n) = n$
  - "è O(n)"

Complessità $\Omega(f(n))$ di un problema

- Un problema ha *complessità temporale $\Omega(f(n))$* se **ogni** algoritmo che lo risolve ha complessità temporale $\Omega(f(n))$
- In forma stenografica:

$$P \in \Omega(f(n)) \Leftrightarrow \forall\ A \in \Omega(f(n))$$

- $\Omega(f(n))$ sono le risorse *necessarie* a risolvere il problema

Complessità $\Omega(f(n))$ di un problema

- Se un problema ha *complessità temporale $\Omega(f(n))$*
  - non è possibile che il problema possa essere risolto spendendo meno di $\Omega(f(n))$
  - non è detto che il problema sia risolvibile spendendo $O(f(n))$
  - $f(n)$ è un *limite inferiore* (lower bound) alle risorse necessarie per risolvere il problema
- Per dimostrare che un problema ha complessità $\Omega(f(n))$
  - non possiamo considerare tutti gli algoritmi che lo risolvono
  - non esiste un metodo preciso per determinare $\Omega(f(n))$
    - generalmente si ragiona sulla natura delle istanze e delle relative soluzioni

Esempio: problema $\Omega(f(n))$

- Consideriamo il problema del calcolo della somma di n interi
- Tutti gli algoritmi che risolvono il problema devono necessariamente prendere in considerazione gli n interi in input
  - altrimenti cambiando un valore di input l'algoritmo darebbe lo stesso output, e questo è assurdo
- Il problema della somma di n interi
  - ha complessità temporale $\Omega(n)$
  - è limitato inferiormente da $f(n) = n$
  - "è $\Omega(n)$"

Complessità $\Theta(f(n))$ di un problema

- Un problema ha *complessità temporale $\Theta(f(n))$* se se ha contemporaneamente complessità temporale $O(f(n))$ e $\Omega(f(n))$
  - non è possibile che il problema possa essere risolto spendendo meno di $O(f(n))$
  - esiste almeno un algoritmo che risolve il problema in $\Theta(f(n))$
- Limite inferiore e limite superiore coincidono
  - $f(n)$ è la complessità intrinseca del problema
- Non sempre è possibile determinare $\Theta(f(n))$
  - di molti problemi la complessità intrinseca è ignota

Esempio: problema $\Theta(f(n))$

- Per quanto detto sopra il problema della somma di n interi ha complessità $\Theta(n)$
  - l'algoritmo proposto per dimostrare che il problema è $O(n)$ è un algoritmo asintoticamente ottimo
    - possiamo desistere dalla ricerca di algoritmi più efficienti
    - è anche vero che questo algoritmo ha complessità temporale $\Theta(n)$

Problemi dalla complessità ignota

- Problema del commesso viaggiatore
  - trovare il circuito più breve che tocca n città
- Upper-bound
  - esiste un algoritmo che ha complessità $O(n^2 2^n)$
- Lower-bound
  - siccome occorre leggere l'input, il problema è $\Omega(n)$
  - non è mai stato dimostrato che il problema non possa essere risolto in tempo polinomiale
    - in realtà non è mai stato dimostrato che il problema non possa essere risolto in tempo lineare!

Problemi NP-completi

- Il problema del commesso viaggiatore appartiene ad una classe di problemi noti come problemi NP-completi
- I problemi NP-completi sono tutti equivalenti
  - se si trovasse un algoritmo polinomiale in grado di risolvere un qualunque problema NP-completo si potrebbero risolvere in tempo polinomiale tutti i problemi NP-completi
- Si ritiene (ma non è stato mai dimostrato) che un algoritmo polinomiale per un problema NP-completo non possa esistere

Lower bound di problemi comuni

- E' molto difficile dimostrare un lower bound per un problema
- Nel seguito dimostreremo dei lower bound limitati al caso in cui gli algoritmi utilizzati siano basati su confronti
- In particolare dimostreremo
  - lower bound $\Omega(\log n)$ per algoritmi di ricerca basati su confronti
  - lower bound $\Omega(n \log n)$ per algoritmi di ordinamento basati su confronti

Il problema della ricerca

- Il problema della ricerca può essere descritto come segue
  - è nota una collezione di coppie <chiave,valore>
  - un'istanza del problema è il valore di una chiave
  - la soluzione del problema è il relativo valore
    - oppure l'informazione che una coppia con tale chiave è assente nella collezione

Cosa sappiamo del problema della ricerca

- Dipendentemente dal tipo di struttura dati che adottiamo per la collezione di coppie <chiave,valore> il problema della ricerca ha diverse complessità nel caso peggiore

Liste	O(n)
Array non ordinati	O(n)
Array ordinati	O(log n)
Alberi rosso-neri	O(log n)

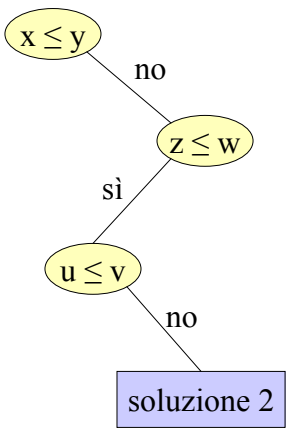
- Esiste un algoritmo più veloce di O(log n)?

Algoritmi basati su confronti

- Un algoritmo di ricerca è detto "*algoritmo di ricerca basato su confronti*" se il flusso delle operazioni dipende esclusivamente dal confronto tra la chiave cercata ed una chiave della collezione
- Esempio
  - nella ricerca binaria si accede all'elemento intermedio dell'intervallo di ricerca e si ricorre su uno dei due sottointervalli generati in base al confronto della chiave cercata con il valore della chiave dell'elemento intermedio

Esecuzione di una ricerca per confronto

- Immaginiamo di lanciare un algoritmo di ricerca basato su confronti
- L'algoritmo eseguirà un certo numero di confronti per poi produrre un output
  - l'output è un'opportuna cella di memoria
- Uno qualsiasi dei valori della collezione potrebbe essere l'output giusto



Albero di decisione

- Possiamo definire un albero i cui nodi interni sono i vari confronti eseguiti dall'algoritmo e le cui foglie sono le possibili risposte
- Questo albero è un albero binario con n foglie
  - l'altezza dell'albero è $\Omega(\log n)$
  - il numero dei confronti necessari per individuare una foglia è $\Omega(\log n)$
- Ne consegue che nel caso peggiore una ricerca implica $\Omega(\log n)$ confronti

Algoritmi di ordinamento

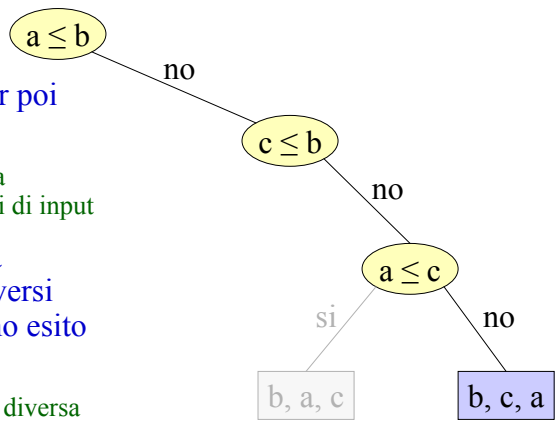
- Con considerazioni analoghe dimostreremo un lower bound sugli algoritmi di ordinamento per confronto
- Gli algoritmi più veloci che conosciamo, come il `MERGE_SORT`, hanno una complessità temporale $\Theta(n \log n)$
- Dimostreremo che tutti gli algoritmi di ordinamento per confronto hanno una complessità nel caso peggiore $\Omega(n \log n)$

Algoritmi di ordinamento per confronto

- Un algoritmo di ordinamento è detto "*algoritmo di ordinamento per confronto*" se il flusso delle operazioni dipende dal confronto tra due elementi della sequenza
- Esempio
  - nel `MERGE_SORT` l'operazione `MERGE` confronta i valori delle due sotto-sequenze ordinate per ottenere un'unica sequenza ordinata

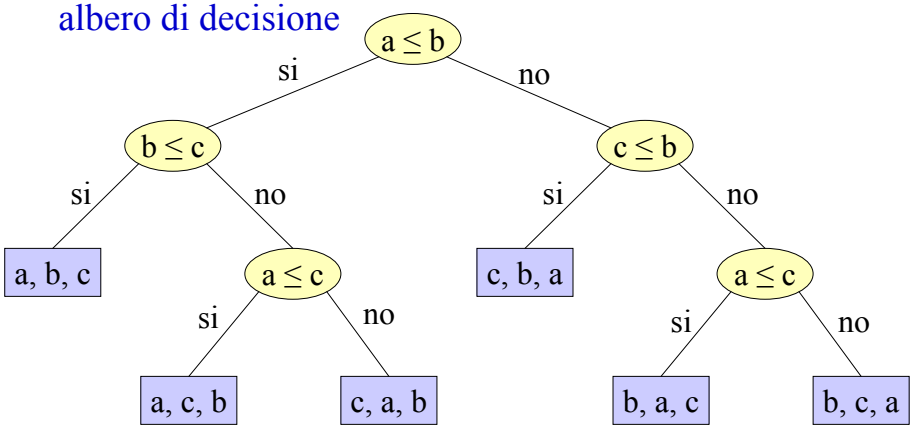
Esecuzione di un algoritmo per confronto

- Immaginiamo di lanciare un algoritmo di ordinamento per confronto con una generica sequenza di input (a,b,c)
- L'algoritmo eseguirà un certo numero di confronti per poi produrre un output
  - l'output è un'opportuna permutazione dei valori di input
- Se lo lanciamo con una sequenza con valori diversi alcuni confronti avranno esito diverso
  - l'output prodotto è una diversa permutazione dei valori di input



Albero di decisione

- L'esecuzione di un algoritmo di ordinamento per confronto equivale alla discesa in un immaginario albero di decisione



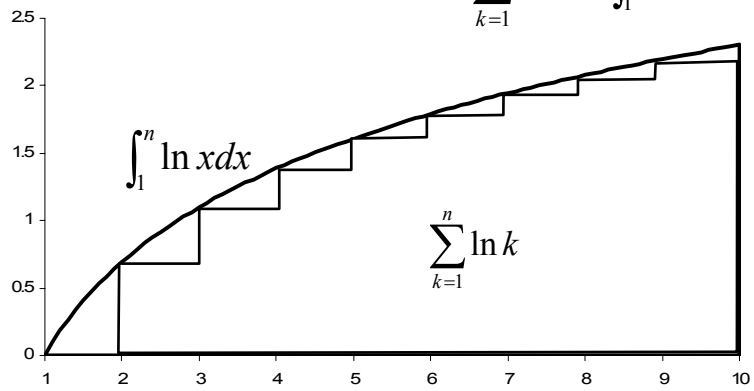
Numero di confronti necessari

- Tutte le permutazioni degli elementi da ordinare devono essere foglie dell'albero di decisione
  - ogni possibile permutazione dei valori di input deve essere raggiungibile
  - se n sono gli elementi da ordinare le possibili permutazioni sono $n!$
- Il numero di confronti eseguiti nel caso peggiore equivale al cammino più lungo tra la radice ed una foglia
  - l'altezza di un albero binario con $n!$ foglie è almeno $\log_2 n!$
  - il problema dell'ordinamento per confronto è $\Omega(\log_2 n!)$

Approssimazione di Stirling

- Consideriamo la funzione ln *n*!
  - nel calcolo asintotico la base del logaritmo è indifferente

$$\ln n! = \ln 1 + \ln 2 + \cdots + \ln n = \sum_{k=1}^{n} \ln k \approx \int_1^n \ln x dx$$



Calcolo di $\int_1^n \ln x dx$

- Integrazione per parti: $\int u \frac{dv}{dx} dx = uv - \int v \frac{du}{dx} dx$
- Nel nostro caso

  $u = \ln x$ $v = x$

  - $dv/dx = 1$; $du/dx = 1/x$

$$\int \ln x \cdot 1 dx = x \ln x - \int x \frac{1}{x} dx = x \ln x - x$$

$$\sum_{k=1}^{n} \ln k \approx \int_1^n \ln x dx = (x \ln x - x)\Big|_1^n = n \ln n - n + 1$$

Ordinamento per confronto: lower bound

- L'esecuzione di un algoritmo di ordinamento per confronto corrisponde alla discesa in un albero di decisione con $n!$ foglie
  - n è il numero di elementi da ordinare
- Nel caso peggiore il numero di confronti (nodi interni nel cammino radice-foglia) è $\Omega(n \ln n)$
  - `MERGE_SORT` è un algoritmo di ordinamento per confronto asintoticamente ottimo

Domande sulla complessità dei problemi

1. Supponiamo che il problema P abbia complessità O(n). E' possibile che esista un algoritmo A che risolve P che abbia una complessità $\Omega(n^2)$?
2. Supponiamo che un problema P abbia complessità $\Theta(n^2)$. Può esistere un algoritmo A che risolve P e ha compessità $\Omega(n)$?
3. Supponiamo che un problema P abbia complessità $\Theta(n)$. Può esistere un algoritmo A che risolve P e ha complessità $\Theta(n^2)$?

Soluzioni

1. $P \in O(n)$. Può esistere $A \in \Omega(n^2)$?
   - Sì, se $P \in O(n)$ vuol dire che esiste un (opportuno) algoritmo $A' \in O(n)$. Gli altri algoritmi, tra cui A, che risolvono P possono avere complessità arbitrariamente elevata
2. $P \in \Theta(n^2)$. Può esistere $A \in \Omega(n)$?
   - Sì. Non solo, tutti gli algoritmi che risolvono P hanno complessità $\Omega(n^2)$ e dunque anche $\Omega(n)$
3. $P \in \Theta(n)$. Può esistere $A \in \Theta(n^2)$?
   - Sì, ciò non contraddice $P \in O(n)$ né $P \in \Omega(n)$.