algoritmi e strutture di dati

complessità dei problemi

m.patrignani

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 ordinamento per confronto
- bucket sort
  - ordinamento in tempo lineare

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 necessarie per risolvere il problema
- per dimostrare che un problema ha complessità O(f(n))
  - è sufficiente 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
somma = A[0]
for i = 1 to A.length-1
    somma = somma + A[i]
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 $\Theta(n)$
      - attenzione: non è vero il viceversa: se un algoritmo è $\Theta(n)$ non vuol dire che il problema è $\Theta(n)$

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!
- nota
  - si sa che se esistesse un algoritmo che risolve in tempo polinomiale questo problema esisterebbero algoritmi per risolvere in tempo polinomiale tutti i problemi appartenenti ad una classe di problemi detta NP

algoritmi di ordinamento

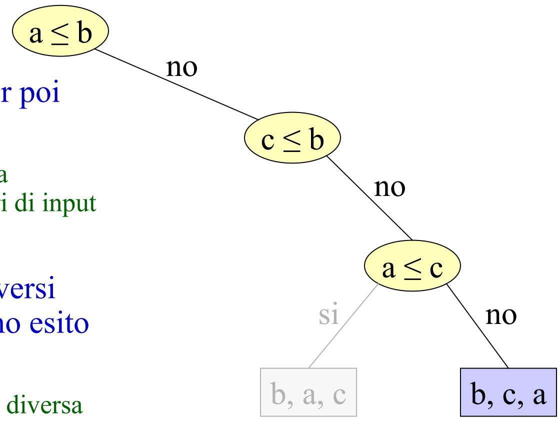
- `MERGE-SORT` ha una complessità temporale $\Theta(n \log n)$
  - è questa la complessità intrinseca del problema dell'ordinamento di una sequenza di interi?
- non riusciremo a dimostrare che il problema dell'ordinamento è $\Theta(n \log n)$
  - dovremmo dimostrare che ogni possibile algoritmo di ordinamento ha complessità $\Omega(n \log n)$
  - lo dimostreremo solamente per una vasta famiglia di algoritmi di ordinamento, chiamati algoritmi di ordinamento "per confronto"

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

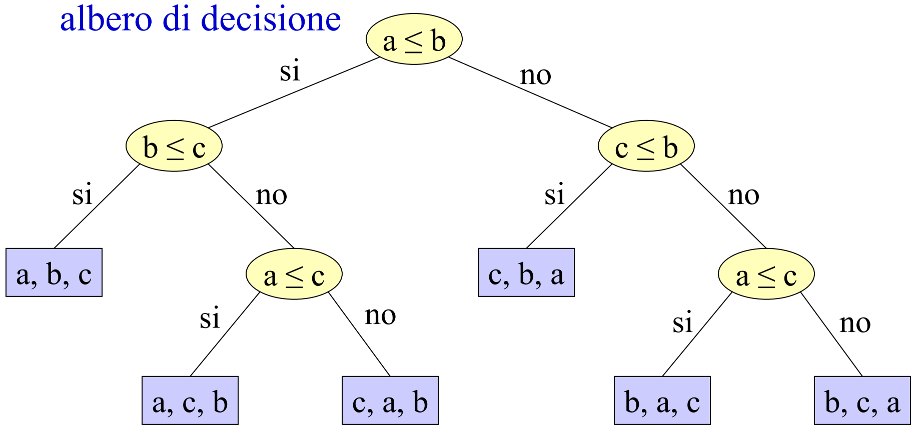
esecuzione di un algoritmo per confronto

- 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



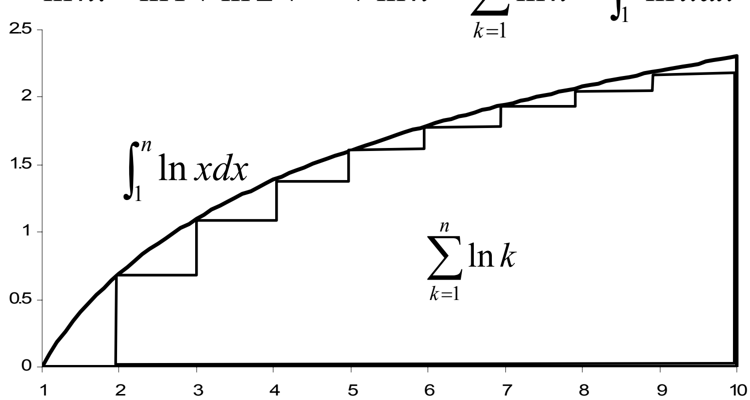
numero di confronti necessari

- 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: $\boxed{\int u \frac{dv}{dx} dx = uv - \int v \frac{du}{dx} dx}$
- nel nostro caso

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

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

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

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

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

ordinamento in tempo lineare

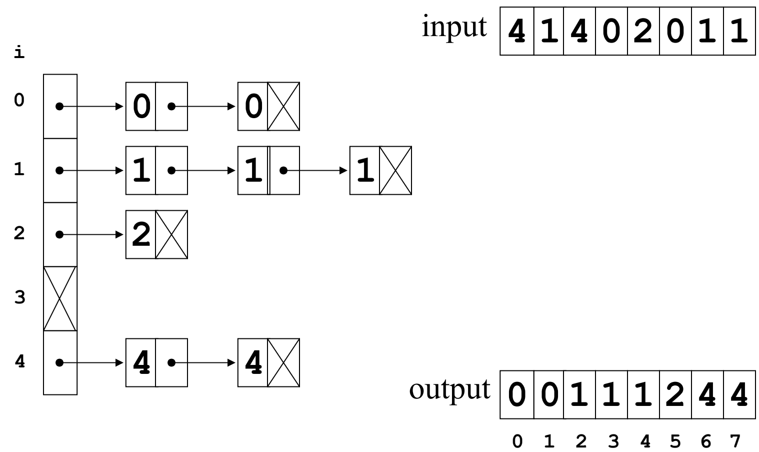
- in alcuni casi è possibile ordinare degli elementi senza eseguire confronti tra i loro valori
- in questi casi si possono concepire algoritmi con complessità Θ(n) nel caso peggiore
- generalmente occorre fare assunzioni sui valori degli n elementi

bucket-sort

- assume che gli n elementi in input siano interi minori o uguali a k, con k noto e $k \in \Theta(n)$
  - k può essere maggiore, minore o uguale ad n
  - l'input può contenere ripetizioni
  - alcuni interi in $[0\ldots k]$ potrebbero non essere presenti
- la strategia è la seguente
  - creo una lista (bucket) per ogni i minore o uguale a k
    - tempo $\Theta(k) = \Theta(n)$
  - scorro l'array di input e metto l'elemento corrente nel bucket che corrisoponde al suo valore
    - tempo $\Theta(n)$
  - svuoto i bucket ponendo i loro elementi nell'array di output
    - tempo $\Theta(n)$

esempio di bucket-sort

- suppongo n=8 e k=4



BUCKET-SORT

- algoritmo per ordinare *n* interi minori o uguali a *k*, con $k \in \Theta(n)$

```
BUCKET-SORT(A,k)          ▷ A=input, k=valore massimo
▷ creo un array B di k bucket (liste)
for i = 0 to B.length-1
    B[i] = EMPTY-LIST()        ▷ inizializzo i bucket vuoti
for i = 0 to A.length-1
    INSERT(B[A[i]],A[i])
i = 0                     ▷ mi posiziono su A[0]
for j = 0 to B.length-1
    while not IS-EMPTY(B[j])
        A[i] = EXTRACT-FIRST(B[j])
        i = i + 1
```

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 $\in \Omega(n)$.