Corso di laurea in Informatica

Progettazione d'algoritmi

Introduzione

Angelo Monti

SAPIENZA
UNIVERSITÀ DI ROMA

Di cosa tratta questo corso

- **Algoritmi**

descriveremo alcuni algoritmi "classici" per risolvere problemi di base

- **Strutture dati**

Esploreremo le strutture dati più adatte da usare

- **Efficienza**

Valuteremo l'efficienza, calcolandone il costo computazionale.

- **Problem solving**

P.S. gli algoritmi verranno descritti tramite pseudocodice, per poterne dare una versione compatta senza perdersi in dettagli implementativi...

Vedremo particolari tecniche che si sono dimostrate utili per la soluzione di una vasta gamma di problemi:

- tecnica greedy
- divide et impera
- programmazione dinamica
- backtraking

Pagina web del corso

Alla pagina:

https://twiki.di.uniroma1.it/twiki/view/Algoritmi2/WebHome

troverete:

- Il programma del corso
- Un elenco di libri di testo utili
- Il diario delle lezioni
- Le modalità d'esame
- Delle dispense
- Un forum

Cominciamo……

Algoritmi efficienti, non efficienti e ottimi, ordine polinominale e superpolinominale.

SAPIENZA
UNIVERSITÀ DI ROMA

Algoritmi

- Un algoritmo si dice ***efficiente*** se la sua complessità è di ordine polinomiale nella dimensione n dell'input.

- Ovvero di complessità $O(n^c)$ per una qualche costante c.

- Di conseguenza un algoritmo è inefficiente se la sua complessità è di ordine superpolinominale:

- Un algoritmo è inefficiente se la sua complessità è di ordine superpolinominale:
  - *Esponenziale* è una funzione di ordine $\Theta(c^n) = 2^{\Theta(n)}$.
  - *Super-esponenziale* è una funzione che cresce più velocemente di un esponenziale.
    - Ad esempio $2^{\Theta(n^2)}$ ma anche $2^{\Theta(n \log n)}$.
  - *Sub-esponenziale* è una funzione che cresce più lentamente di un esponenziale vale a dire: $2^{o(n)}$.
    - Ad esempio $n^{\Theta(\log n)} = 2^{\Theta(\log^2 n)}$ oppure $2^{\Theta(n^c)}$ dove c è una costante inferiore ad 1.

- I problemi di cui si conoscono algoritmi subexponenziali e non polinomiali sono pochi e proprio per questo molto studiati.

- Ad esempio per i problemi della fattorizzazione e dell'isomorfismo tra grafi sono noti da tempo algoritmi superpolinomiali di complessità $2^{O\left(n^{1/3}\right)}$.

- Come caso di studio basti pensare al **test di primalità**.

- L'algoritmo banale (dato il numero n, cerca un eventuale divisore tra tutti i numeri tra 2 e $n-1$) è esponenziale perché ha complessità $O(n)$ (nota che la dimensione dell'input è $\log n$).

- Un algoritmo efficiente per questo problema dovrebbe avere complessità $O(\log^c n)$ per una qualche costante c mentre questo algoritmo ha complessità $O(2^{\log n})$.

- Nella ricerca dell'eventuale divisore fermarsi alla radice velocizza l'algoritmo ma non ne cambia la complessità asintotica che resta esponenziale:

- $$O\left(\sqrt{n}\right) = O\left(2^{\log\sqrt{n}}\right) = O\left(2^{\frac{1}{2}\log n}\right) = 2^{\Theta(\log n)}.$$

- Come caso di studio basti pensare al **test di primalità**.

- Fin dagli anni 70 erano noti per questo problema algoritmi sub-exponenziali ad esempio di complessità $(\log n)^{\Theta(\log\log n)}$.
- Erano noti anche algoritmi probabilistici (algoritmi che possono sbagliare con una probabilità che si può rendere piccola a piacere) di complessità polinomiale $O\left(\log^3 n\right)$.
- solo nel 2004 si è trovato un algoritmo polinomiale deterministico di tempo $O(\log^{12} n)$ che è stato poi velocizzato a $O(\log^3 n)$ anche se con costanti moltiplicative molto alte che non lo rendono competitivo con i ben noti algoritmi probabilistici.

Per uno stesso problema possono esistere svariati algoritmi, si consideri ad esempio il problema di ordinare una lista di n interi.

Un banale algoritmo superpolinominale è il seguente:

- genera tutte le possibili permutazioni dei valori da ordinare (complessità $O(n!)$) e verifica se la permutazione soddisfa l'ordinamento (complessità $O(n)$).

- La complessità complessiva è quindi $O(n \cdot n!)$.

- La funzione fattoriale è super-esponenziale ($n! = 2^{\Theta(n \log n)}$) e di conseguenza l'algoritmo è intrattabile.

- $n! = 1 \cdot 2 \cdot 3 \cdots \cdot n > \frac{n}{2} \cdot \left(\frac{n}{2} + 1\right) \cdots \cdot n > \left(\frac{n}{2}\right)^{\frac{n}{2}} = 2^{\frac{n}{2} \log \frac{n}{2}} = 2^{\Omega(n \log n)}$

- $n! = 1 \cdot 2 \cdot 3 \cdots \cdot n < n \cdot n \cdots \cdot n = n^n = 2^{n \log n} = 2^{O(n \log n)}$

Possiamo anche considerare quest'altro algoritmo: cerca il minimo tra gli n valori e mettilo in prima posizione; cerca il minimo tra gli $n-1$ valori restanti e mettilo in seconda posizione; e cosi via.

```
def SelectionSort(lista):
    n=len(A)
    for i in range(n-1):
        minimo=i
        for j in range (i+1,n):
            if lista[j] < lista[minimo]:
                minimo=j
        lista[i], lista[minimo] = lista[minimo], lista[i]
```

La complessità è $\Theta(n^2)$, si tratta quindi di un algoritmo polinomiale.

Se uso una struttura dati opportuna, l'heap, la cui costruzione per n elementi costa $\Theta(n)$ e dove l'estrazione del minimo costa $O(\log n)$ ottengo un algoritmo (noto come ***heapsort***) che ordina gli n elementi in tempo $O(n \log n)$.

```
def ordina(lista):
    heap=costruisci_heap(lista)
    lista1=[ ]
    for _ in range(len(lista)):
        a=estrai_minimo(heap)
        lista1.append(a)
    return lista1
```

Non bisogna confondere la complessità dell'algoritmo con la complessità del problema.

- Un algoritmo di complessità $O(g(n))$ per un problema produce una limitazione superiore alla complessità del problema.

- Se si dimostra che ***qualunque algoritmo*** per quel problema ha complessità $\Omega(f(n))$, si è stabilita una limitazione inferiore alla complessità del problema.

- Se $f(n) = g(n)$ allora l'algoritmo è detto ottimo, perché la sua complessità in ordine di grandezza risulta la migliore possibile.

Calcolare limitazioni inferiori significative ai problemi in genere non è un compito semplice.

- Sono noti pochi modi generali di dimostrazione per limitazioni inferiori. Vediamone due molto semplici:

- **Dimensione dei dati:**

se un problema ha in ingresso n dati e richiede di esaminarli tutti, allora una limitazione inferiore della complessità è $\Omega(n)$.

Ad esempio per il problema della ricerca del massimo in una lista, l'algoritmo banale che scorre la lista è ottimo.

Per l'ordinamento non è sufficiente questo ragionamento.

- **Eventi contabili:**

se un problema richiede che un certo evento sia ripetuto almeno m volte, allora una limitazione inferiore della complessità è $\Omega(m)$.

Ad esempio per gli algoritmi di ordinamento basati sul confronto si può dimostrare che se n sono gli interi da ordinare allora bisogna effettuare almeno $n \log n$ confronti e quindi $\Omega(n \log n)$ è il limite inferiore alla complessità e di conseguenza l'algoritmo heapsort è ottimo.

Nota che se aggiungo vincoli al problema dell'ordinamento il lower bound $O(n \log n)$ non vale più perché potrei utilizzare algoritmi che non si basano sul confronto.

- Ad esempio se gli elementi della lista da ordinare son di soli due tipi ho un vincolo sull'intervallo dei numeri che debbo ordinare (in questo caso 2) allora uso il *countingSort* e ho la risposta in $O(n)$.

- Più in generale l'algoritmo si applica quando i valori da ordinare sono interi positivi.

- La complessità dell'algoritmo è $O(k+n)$ dove k è il massimo tra gli elementi da ordinare.

- Nota che in generale l'algoritmo non è polinomiale.

E' raro sapere che un algoritmo è ottimo per un certo problema.
Ad esempio per il prodotto di due matrici quadrate $n \times n$ l'algoritmo insegnato a scuola (algoritmo di Gauss) richiede $O(n^3)$ tempo:

- infatti per ciascuna delle n^2 posizioni della matrice risultante debbo calcolare un prodotto scalare tra due vettori lunghi n.

Per lungo tempo si è pensato che quello fosse anche il tempo ottimo. Strassen nel 1969 ha trovato un algoritmo di complessità $O(n^{log_2 7}) = O(n^{2.8})$ basato sulla tecnica del divide-et-impera.

Da allora ci sono stati vari miglioramenti, attualmente *mi pare* che siamo a $O(n^{2.37})$ Coppersmith-Winograd

Fino ad oggi il lower bound più forte è $\Omega(n^2)$ (basato sul fatto che gli n^2 elementi della matrice vanno tutti calcolati).

Molti congetturano l'esistenza di un algoritmo a complessità $\Theta(n^2)$.

Un problema si dice **intrattabile** se non può avere algoritmi efficienti.

Esempi ovvi di problemi provatamente intrattabili sono:

- Stampare tutte le stringhe binarie di lunghezza n.

  Il problema ha un limite inferiore $2^{\Omega(n)}$.

- stampare tutte le permutazioni di n elementi.

Il problema ha un limite inferiore $2^{\Omega(n \log n)}$

Esiste tutta una serie di importanti problemi decisionali (vale a dire con risposta vero o falso) di cui si sospetta l'intrattabilità questi problemi hanno la singolare proprietà che se per uno di questi si riuscisse a dimostrare l'intrattabilità come corollario si otterrebbe la prova di intrattabili per tutti gli altri. Questo sarà argomento del corso di automi e complessità

Perché per algoritmo efficiente si intende un algoritmo a complessità $O(n^c)$ per un generico intero c e non più semplicemente $O(n^2)$ o $O(n^3)$?

- **Tesi di Church-Turing:** I modelli di calcolo realistici sono equivalenti dal punto di vista computazionale. Se qualcosa è non calcolabile ad esempio sulla macchina di Turing lo resterà su qualunque macchina di calcolo automatico.

- **Tesi di Church-Turing estesa** I modelli di calcolo realistici sono tra loro polinomialmente correlati. **Il concetto di trattabilità è dunque indipendente dalla macchina.**

La tesi estesa è tutt'altro che accettata universalmente. Molti ad esempio ritengono che con l'avvento dei calcolatori quantistici molti problemi che attualmente risultano intrattabili diventeranno trattabili (tipo la fattorizzazione per cui gia esiste un algoritmo quantistico polinomiale).
Allo stato attuale per questi problemi non si può escludere l'esistenza di un algoritmo classico efficiente, quindi l'effettiva potenza computazionale delle macchine quantistiche è tutta da dimostrare.

Corso di laurea in Informatica
Introduzione agli Algoritmi

Esercizi per casa

SAPIENZA
UNIVERSITÀ DI ROMA

ESERCIZI

- Data una lista di n interi vogliamo determinare se la lista ha un elemento di maggioranza assoluta (vale a dire un elemento che compare nella lista almeno $\left\lfloor \frac{n}{2} \right\rfloor + 1$ volte).

Progettare un algoritmo efficiente (possibilmente ottimo) per questo problema.

- Dati $2^{n-1} + 1$ distinti sottoinsiemi ottenuti a partire da un insieme S di n elementi. Progettare un algoritmo ottimo che testi se tra i sottoinsiemi ne esistano due, A e B, con $A \cup B = S$.

ESERCIZI

- Data una lista di n interi distinti ed un intero k vogliamo sapere se nella lista sono presenti due elementi la cui somma dia k e nel caso ci siano quali sono.

1) Progettare un algoritmo che risolve il problema in $O(n^2)$
2) Progettare un algoritmo che risolve il problema in tempo $o(n^2)$
3) Progettare un algoritmo che risolve il problema in tempo ammortizzato $O(n)$

- Data una lista di n interi distinti ed un intero k vogliamo contare le triple di elementi della lista che hanno come somma k.

1 Progettare un algoritmo che risolve il problema in tempo $O(n^3)$.
2) Progettare un algoritmo che risolve il problema in tempo $o(n^3)$.
3) Progettare un algoritmo che risolve i problema in tempo *ammortizzato* $O(n^2)$

Soluzione di complessità $O(n^2)$ per il primo esercizio:

```
def majority(A):
    n = len(A)
    for i in range(n):
        if A.count(A[i]) >= n//2 +1:
            return A[i]
    return None
```

Complessità computazionale:

- La funzione $A.count(A[i])$ conta le occorrenze di $A[i]$ ed ha costo $\Theta(n)$.
- Il ciclo for viene eseguito al massimo $O(n)$ volte
- quindi la complessità è $O(n^2)$

Soluzione di complessità $O(n \log n)$ per il primo esercizio:

```
def majority(A):
    n = len(A)
    A.sort()
    t = 1
    for i in range(1,n):
        if A[i] == A[i-1]:
            t += 1
            if t >= n//2 +1:
                return A[i]
        else:
            t = 1
    return None
```

Complessità computazionale:

- **Ordinamento:** $O(n \log n)$.
- **Scansione lineare:** $O(n)$.
- **Totale:** $O(n \log n)$

Soluzione di complessità $O(n)$ per il primo esercizio:

```
def esercizio1(A):
    B=[]
    for x in A:
        if B!=[] and B[-1]!=x:
            B.pop()
        else:
            B.append(x)
    if B and A.count(B[0])> len(A)//2:
        return B[0]
    return None
```

Complessità computazionale:

1. Se in A c'è l'elemento di maggioranza al termine deve essere in B.
2. Gli elementi presenti in B son tutti uguali.