La Tecnica del Divide et Impera – Parte I

Algoritmi 2 – Lezione 13

A. Monti
Sapienza Università di Roma

Corso: Algoritmi 2

Indice

Introduzione: La Tecnica del Divide et Impera

- Il "Divide et Impera" è una strategia fondamentale per la progettazione di algoritmi, ispirata all'antica strategia politica e militare romana.
- L'idea in informatica è quella di risolvere un problema complesso suddividendolo in sotto-problemi più semplici.
- Il processo si articola in tre fasi:
  1. **Dividi (Divide)**: Scomporre il problema principale in sotto-problemi più piccoli.
  2. **Impera (Conquer)**: Risolvere i sotto-problemi, o direttamente se sono piccoli (caso base) o ricorsivamente.
  3. **Combina (Combine)**: Fondere le soluzioni dei sotto-problemi per ottenere la soluzione finale.

Esempi Classici di Divide et Impera

Questa tecnica è particolarmente efficace per problemi che hanno due proprietà:

- **Scomponibilità**: Il problema può essere suddiviso in modo efficiente.
- **Indipendenza dei Sotto-problemi**: I sotto-problemi possono essere risolti indipendentemente l'uno dall'altro, senza sovrapposizioni.

Molti degli algoritmi più famosi si basano su questo paradigma:

- **Merge Sort**: Dividere l'array, ordinare ricorsivamente le due metà e combinarle unendole in tempo lineare.
- **Binary Search**: Dividere l'array, confrontare l'elemento con il centro per scartare una metà. La fase di combinazione è banale.
- **Quicksort**: Partizionare l'array in base a un pivot e ordinare ricorsivamente le due partizioni. In questo caso, la fase di combinazione è trascurabile

Il Problema della Selezione

- Dato un insieme di **n** interi distinti e un intero **k**, $0 \leq k \leq n-1$, vogliamo determinare quale elemento si troverebbe nella **k-esima posizione** se l'insieme venisse ordinato in modo crescente.

- **Soluzione banale**: ordinare l'insieme in $O(n \log n)$ e restituire l'elemento in posizione k.

- **Casi particolari**:
  - $k = 0$: minimo $O(n)$
  - $k = n - 1$: massimo $O(n)$

- Ci si chiede: si può fare di meglio?

Quickselect: Selezione in Tempo Lineare (in Media)

L'idea alla base è simile a quella del Quicksort:

- **Fasi Fondamentali**:
  1. **Dividi (Scegli Pivot)**: Selezione di un pivot.
  2. **Impera (Partiziona)**: L'array viene diviso in due sotto-array: "minori" e "maggiori" rispetto al pivot.
  3. **Combina (Decidi e Ricorri)**: In base alla posizione del pivot, si continua la ricerca ricorsivamente **solo** sul sotto-array corretto, scartando l'altro.

Analisi di Quickselect

- **Caso Peggiore**: La complessità degenera a $O(n^2)$ se il pivot scelto è sempre l'elemento più piccolo del sotto-array.
- **Soluzione Pratica**: La scelta casuale del pivot renderà il caso peggiore estremamente improbabile.
- **Complessità Media**: La randomizzazione assicura una complessità attesa lineare, $O(n)$, rendendo Quickselect l'algoritmo più usato nella pratica.

Caso Peggiore di Quickselect: Esempio Estremo

Consideriamo un array già ordinato in ordine crescente, e supponiamo di voler trovare l'elemento più grande usando Quickselect. Se l'algoritmo sceglie sempre il primo elemento come pivot (il minimo), si verifica il seguente comportamento degenerato:

- **Primo passo (n elementi)**: Il pivot è il minimo. La partizione richiede circa n confronti e genera un sotto-array vuoto e uno con $n-1$ elementi. La ricorsione procede su quest'ultimo.
- **Secondo passo ($n-1$ elementi)**: Di nuovo, il pivot è il minimo, con altri $n-1$ confronti. Si continua sul sotto-array di $n-2$ elementi.
- **E così via**, fino a ridurre il problema a una dimensione unitaria.

In totale, ad ogni passo si esegue una partizione su un array di dimensione decrescente, con un numero di confronti che segue la ricorrenza:

$$T(n) = T(n-1) + O(n)$$

La cui soluzione è $T(n) = O(n^2)$, mostrando come una scelta pessima del pivot possa compromettere gravemente le prestazioni dell'algoritmo.

Implementazione di Quickselect

```
import random

def Quickselect(A, k):
'''
Trova l'elemento che si troverebbe all'indice k
se l'array A fosse ordinato.
k deve essere compreso tra 0 e len(A)-1.
'''
# Scelta casuale del pivot
perno = random.choice(A)
# Partizionamento in due gruppi:
minori, maggiori = [ ], [ ]
for x in A:
    if x < perno:
        minori.append(x)
    elif x > perno:
        maggiori.append(x)
len_minori = len(minori)
if k < len_minori:
    # L'elemento cercato è  nel gruppo dei minori.
    return Quickselect(minori, k)
elif k == len_minori:
    # L'elemento cercato è il pivot,
    return perno
else:
    # L'elemento cercato è nel gruppo dei maggiori.
    return Quickselect(maggiori, k - len_minori - 1)
```

Quickselect: Caso Medio

- **Teorema** Quickselect ha tempo medio $O(n)$.

Dimostrazione. Calcoliamo il numero medio di confronti tra elementi della lista perché, in questo tipo di algoritmi, i confronti sono l'**operazione dominante** che ne determina la complessità. Indichiamo questa quantità con $E[T(n)]$.

- Se il pivot è l'i-esimo elemento più piccolo i due sotto-array avranno dimensioni $i-1$ e $n-i$
- Poiché a noi basta un limite superiore al valore di $E[T(n)]$ assumiamo pessimisticamente che la ricorsione avvenga sempre sul sotto-array più grande, di dimensione $\max(i-1, n-i)$
- Poiché il pivot è scelto in modo casuale, ogni possibile scelta ha una probabilità di $\frac{1}{n}$, ne segue che il costo atteso su tutte le possibili scelte del pivot è

$$\sum_{i=1}^{n} \frac{1}{n} \cdot E\left[T\left(\max(i-1, n-i)\right)\right].$$

- Sfruttando la simmetria del sotto-problema più grande possiamo semplificare la sommatoria:

$$\frac{2}{n} \sum_{i=\lfloor n/2 \rfloor}^{n-1} E[T(i)]$$

Possiamo concludere quindi che

$$E[T(n)] \leq n + \frac{2}{n} \sum_{i=\lfloor n/2 \rfloor}^{n-1} E[T(i)]$$

dove n è il numero di confronti richiesti per partizionare l'array.

Questa ricorrenza risolta (ad esempio col metodo di sostituzione) dà $O(n)$.

Risoluzione della Ricorrenza: Caso Medio

$$E[T(n)] \leq n + \frac{2}{n} \sum_{i=\lfloor n/2 \rfloor}^{n-1} E[T(i)]$$

Assumiamo per induzione che $E[T(i)] \leq ci$ per ogni $i < n$. Sostituendo questa ipotesi nella ricorrenza otteniamo:

$$cn \leq n + \frac{2}{n} \sum_{i=\lfloor n/2 \rfloor}^{n-1} ci$$

Per valori di n sufficientemente grandi, la sommatoria può essere approssimata dall'integrale della funzione $f(x) = cx$:

$$\sum_{i=\lfloor n/2 \rfloor}^{n-1} ci \approx \int_{n/2}^{n} cx \, dx$$

Risolviamo l'integrale:

$$\int_{n/2}^{n} cx \, dx = c \left[\frac{x^2}{2} \right]_{n/2}^{n} = c \left(\frac{n^2}{2} - \frac{(n/2)^2}{2} \right) = c \left(\frac{n^2}{2} - \frac{n^2}{8} \right) = c \frac{3n^2}{8}$$

Ora sostituiamo il risultato dell'integrale nella nostra disuguaglianza:

$$cn \leq n + \frac{2}{n} \left(c \frac{3n^2}{8} \right)$$

e, con semplici passaggi algebrici otteniamo $c \leq 4$.
La disuguaglianza è dunque soddisfatta per qualsiasi costante $c \leq 4$. Abbiamo quindi dimostrato che $E[T(n)] \leq 4n = O(n)$.

Mediana delle Mediane: Strategia e Intuizione

- L'algoritmo garantisce una complessità di $O(n)$ nel caso peggiore, a differenza di Quickselect che ha $O(n^2)$.
- **Idea Fondamentale**: Scegliere un pivot che sia garantito non essere troppo vicino agli estremi, scartando una frazione costante di elementi ad ogni passo.
- **Descrizione dell'Algoritmo**:
  1. Dividi l'array in gruppi di 5 elementi.
  2. Calcola la mediana di ogni gruppo (costo costante).
  3. Richiama ricorsivamente l'algoritmo sulla lista delle mediane per trovare la "mediana delle mediane" che sarà il pivot.
  4. Partiziona l'array originale usando questo pivot.
  5. Ricorri sul sotto-array corretto.

Codice: Mediana delle Mediane

```
def selezione(A, k):
    """
    Trova l'elemento che si troverebbe all'indice k
    se l'array A di elementi distinti fosse ordinato.
    Usa l'algoritmo Mediana delle Mediane.
    Garantisce una complessità di O(n) nel caso peggiore.
    """
    n = len(A)
    # per array piccoli.
    if n < 10:
        A.sort()
        return A[k]
    # Divisione in gruppi di 5 e calcolo delle mediane
    mediane = []
    for i in range(0, n, 5):
        gruppo = sorted(A[i : i+5])
        mediana_gruppo = gruppo[len(gruppo) // 2]
        mediane.append(mediana_gruppo)
    # Ricorsione per trovare la "mediana delle mediane"
    pivot = selezione(mediane, len(mediane) // 2)
    # Partizione dell'array originale A attorno al pivot
    minori, maggiori = [], []
    for x in A:
        if x < pivot:
            minori.append(x)
        elif x> pivot:
            maggiori.append(x)
    len_minori = len(minori)
    # Ricorsione sul sotto-array corretto
    if k < len_minori:
        # L'elemento è nel gruppo dei minori
        return selezione(minori, k)
    elif k == len_minori:
        # L'elemento è il pivot stesso
        return pivot
    else:
        # L'elemento è nel gruppo dei maggiori
        # Scaliamo k sottraendo gli elementi minori e il pivot
        k_nuovo = k - len_minori - 1
        return selezione(maggiori, k_nuovo)
```

Analisi del Pivot nella Mediana delle Mediane

L'efficacia dell'algoritmo dipende dalla "qualità" del pivot p (la mediana delle mediane). Dobbiamo trovare il numero minimo di elementi che il pivot è garantito a scartare.

- Almeno la metà delle $\lceil n/5 \rceil$ mediane è minore a p. Questo corrisponde a circa $n/10$ gruppi.
- Per ognuno di questi $n/10$ gruppi, la loro mediana è inferiore a p, e almeno altri 2 elementi nel gruppo sono anch'essi inferiori a p.
- Il numero di elementi garantiti essere inferiori a p è almeno $3 \cdot (n/10) = 3n/10$.
- Simmetricamente, almeno $3n/10$ elementi sono garantiti essere superiori a p.

Di conseguenza, la chiamata ricorsiva finale opererà su un sotto-array di dimensione al più $n - 3n/10 = \mathbf{7n/10}$.

Ricorrenza della Mediana delle Mediane

- Il costo totale $T(n)$ è la somma dei costi lineari e delle due chiamate ricorsive: una sulla lista delle mediane (di dimensione $\lceil n/5 \rceil$) e una sul sotto-array partizionato (di dimensione massima $7n/10$).

$$T(n) \leq O(n) + T(\lceil n/5 \rceil) + T(7n/10)$$

- La ricorrenza può essere facilmente risolta col metodo di sostituzione e dà $O(n)$.

Dimostrazione: Tempo Lineare nel Caso Peggiore

- Per dimostrare che $T(n) = O(n)$, dobbiamo mostrare che per la ricorrenza

$$T(n) \leq a \cdot n + T(\lceil n/5 \rceil) + T(7n/10)$$

dove a è una fissata costante, esiste una costante c sufficientemente grande tale che $T(n) \leq c \cdot n$.

- Assumiamo per induzione che $T(k) \leq c \cdot k$ per tutti i valori $k < n$. Sostituendo l'ipotesi nella ricorrenza (e ignorando per semplicità i 'ceil'):

$$T(n) \leq a \cdot n + c\frac{n}{5} + c\frac{7n}{10}$$

Procediamo con semplici passaggi algebrici :

$$T(n) \leq a \cdot n + c\left(\frac{9n}{10}\right)$$

Vogliamo che questa espressione sia minore o uguale a $c \cdot n$:

$$a \cdot n + c\frac{9n}{10} \leq c \cdot n$$

ovvero $\boldsymbol{10 \cdot a \leq c}$.
Poiché a è una costante fissa, possiamo sempre scegliere una costante c che soddisfi questa condizione (es. $c = 10 \cdot a$). La nostra ipotesi induttiva è quindi verificata. Questo dimostra che $T(n) \leq c \cdot n$, e pertanto $\boldsymbol{T(n) = O(n)}$. □

Quickselect vs. Mediana delle Mediane: Trade-off Pratici

- **Quickselect (con pivot casuale)** è solitamente preferito nella pratica nonostante il suo caso peggiore teorico di $O(n^2)$.
- È più semplice da implementare e ha un fattore costante inferiore, il che lo rende più veloce nella realtà.
- Il caso peggiore è estremamente raro in una buona implementazione randomizzata.
- **Mediana delle Mediane** è usata in contesti critici (es. sistemi in tempo reale) dove è necessaria una garanzia deterministica di tempo lineare, poiché il suo overhead computazionale è significativo.

La scelta tra le due strategie dipende dunque da una valutazione tra semplicità, efficienza media e garanzie nel caso peggiore.

ESERCIZI

Suggerimento: Prima di cercare le soluzioni in rete, prova a ragionarci da solo: gli esercizi che seguono sono pensati per aiutarti a mettere alla prova la tua comprensione, la tua capacità di sviluppare una prova di correttezza e costruire un algoritmo corretto.
Solo dopo averci riflettuto, confronta la tua soluzione con altre possibili versioni.

Esercizi

1. Si considerino i tre algoritmi seguenti, progettati per risolvere il medesimo problema. Si chiede di determinare la complessità asintotica di ciascun algoritmo e di specificare quale dei tre sia il più efficiente. Si assuma che per tuttti e tre gli algoritmi il costo della suddivisione in sottoproblemi è costante.
   1. **Algoritmo A:** Risolve un problema di dimensione n dividendolo in 5 sottoproblemi, ciascuno di dimensione $n/2$. La ricombinazione delle soluzioni richiede tempo lineare.
   2. **Algoritmo B:** Risolve un problema di dimensione n dividendolo in 2 sottoproblemi, ciascuno di dimensione $n-1$. La ricombinazione delle soluzioni richiede tempo costante.
   3. **Algoritmo C:** Risolve un problema di dimensione n dividendolo in 9 sottoproblemi, ciascuno di dimensione $n/3$. La ricombinazione delle soluzioni richiede tempo $O(n^3)$.
2. Si valutino i tre algoritmi seguenti, che risolvono il medesimo problema, al fine di determinare il più efficiente. Si assuma che per tuttti e tre gli algoritmi il costo della suddivisione in sottoproblemi è lineare in n.
   1. **Algoritmo A:** Risolve un problema di dimensione n dividendolo in 4 sottoproblemi, ciascuno di dimensione $n/16$. La fase di ricombinazione delle soluzioni ha un costo quadratico.
   2. **Algoritmo B:** Risolve un problema di dimensione n dividendolo in 3 sottoproblemi, ciascuno di dimensione $n-4$. La fase di ricombinazione ha un costo costante.
   3. **Algoritmo C:** Risolve un problema di dimensione n dividendolo in 3 sottoproblemi, ciascuno di dimensione $n/5$. La fase di ricombinazione ha un costo pari a $O(n^2 \log n)$.

Esercizi

3 Progettare un algoritmo che, dati due interi, una base x e un esponente n, calcoli il valore x^n in tempo $O(\log n)$.

4 Si consideri un vettore `V`, ordinato e contenente n interi distinti.
Progettare un algoritmo che verifichi l'esistenza di un *punto fisso*, ovvero un indice i tale che $V[i] = i$. La complessità temporale richiesta per l'algoritmo è di $O(\log n)$.

5 Si definisce **gap** di un vettore `V` un indice i, tale che $V[i] < V[i+1]$.

Dato un vettore `V` di $n \geq 2$ interi dove il primo elemento è strettamente inferiore all'ultimo, si chiede di:

1. Dimostrare che `V` contiene almeno un gap.
2. Progettare un algoritmo che identifichi la posizione di un gap con una complessità temporale di $O(\log n)$.

6 Si definisce **doppio-gap** di un vettore `V` un indice i, per cui vale $V[i+1] - V[i] \geq 2$.

Dato un vettore `V` di $n \geq 2$ interi che soddisfa la disuguaglianza dove l'ultimo ed il primo elemento del vettore differiscono di almeno n, si chiede di:

1. Dimostrare che `V` contiene necessariamente almeno un doppio-gap.
2. Progettare un algoritmo efficiente che individui la posizione di un doppio-gap con una complessità temporale di $O(\log n)$.

Esercizi

7 Sviluppare un algoritmo basato sulla tecnica del divide et impera che, data una stringa binaria di lunghezza n, in tempo $O(n)$, restituisca il numero di sottostringhe che cominciano con 0 e terminano con 1.

8 Un vettore di n interi V è *continuo* se $|V[i+1] - V[i]| \leq 1$ per ogni $0 \leq i < n$. Si dice *zero* del vettore un indice k tale che $V[k] = 0$.

1. Dato un vettore V continuo di $n \geq 2$ interi tale che il primo elemento è negativo e l'ultimo positivo, provare che V ha almeno uno zero.
2. Progettare un algoritmo che, dato un vettore V di $n \geq 2$ interi continuo e tale che $V[1] < 0$ e $V[n-1] > 0$, trovi uno zero in $O(\log n)$ tempo.

9 Progettare un algoritmo che, dato un intero n, calcoli il valore $\lfloor \sqrt{n} \rfloor$ in $O(\log n)$ tempo, usando solo operazioni aritmetiche.

10 Si consideri un vettore di n interi, ottenuto dalla rotazione di un vettore originariamente ordinato in modo crescente. Si chiede di progettare un algoritmo che identifichi l'elemento massimo presente nel vettore con una complessità temporale di $O(\log n)$.

Esercizi

11 Si consideri il problema della ricerca del *sottovettore di valore massimo*. *Dato un vettore V di n interi, si definisce* ***sottovettore*** *una qualsiasi sequenza non vuota di elementi consecutivi e* ***valore*** *di un sottovettore la somma dei suoi elementi. Il problema consiste nel trovare il valore massimo tra tutti i possibili sottovettori.*

Si chiede di progettare due algoritmi per risolvere questo problema, entrambi basati sulla tecnica del **Divide et Impera**, ma con diverse complessità:

1. Un primo algoritmo con una complessità temporale di $O(n \log n)$.
2. Un secondo algoritmo, più efficiente, con una complessità temporale lineare, ovvero $O(n)$.

12 Si definisce **spessore** di un vettore di interi `V` la differenza tra il suo valore massimo e il suo valore minimo.

Dati un vettore `V` di n interi e un valore soglia C, si chiede di progettare un algoritmo che identifichi il sottovettore di lunghezza massima il cui spessore non superi C.

La complessità richiesta per l'algoritmo è di $O(n \log n)$.