Il Problema della Ricerca

Algoritmi 1 – Lezione 6

A. Monti
Sapienza Università di Roma

Corso: Algoritmi 1

Indice

Il problema della ricerca

- La **ricerca** è un problema fondamentale in informatica: consiste nel trovare un elemento specifico all'interno di un insieme di dati.

- Gli algoritmi di ricerca sono essenziali per molte applicazioni, come la gestione di database, l'elaborazione di grandi volumi di dati e l'ottimizzazione delle risorse.

Ricerca Lineare - Descrizione e Codice

L'algoritmo scorre sequenzialmente la lista degli elementi fino a trovare quello desiderato o fino a determinare che non è presente.

```
def ricerca(lista, x):
    """ Esegue una ricerca lineare per trovare
    l'indice dell'elemento x nella  lista.
    Parametri:   La lista e l'elemento da ricercare.
    Ritorna:     L'indice dell'elemento se trovato,
       altrimenti None.
    """
    for i in range(len(lista)):
        if lista[i] == x:
            return i
    return None
```

Ricerca Lineare: Complessità nei casi estremi

Consideriamo i due scenari classici: caso ottimo, caso pessimo.

- **caso ottimo.** Si verifica quando l'elemento da cercare è il primo della lista. Solo il primo confronto è necessario per trovare l'elemento e la complessità è $O(1)$.
- **caso pessimo.** Si verifica quando l'elemento da cercare non è presente nella lista. In questo caso si devono confrontare tutti gli elementi della lista. La complessità è $O(n)$ dove n è il numero totale di elementi della lista.

Nei casi, in cui non sia possibile determinare un valore stretto per il costo computazionale, ed in cui il caso migliore e quello peggiore si discostano, è naturale domandarsi quale sia il costo computazionale nel **caso medio** che rappresenta la situazione in cui l'elemento cercato si trova in una posizione casuale nella lista.

Ricerca Lineare: Complessità nel caso medio

- **Ipotesi:** Supponiamo che l'elemento cercato abbia probabilità uniforme di trovarsi in qualsiasi posizione della lista.
- Questo significa che la probabilità che x si trovi in ciascuna delle n posizioni è $\frac{1}{n}$.
- In questo caso, il numero medio di confronti necessari per trovare x è dato dalla media delle posizioni $1, 2, \ldots, n$:

  Numero medio di confronti:

$$\sum_{i=1}^{n} i \cdot \frac{1}{n} = \frac{1}{n} \sum_{i=1}^{n} i = \frac{1}{n} \frac{n(n+1)}{2} = \frac{n+1}{2} = \Theta(n)$$

  La complessità al caso medio rimane dunque di ordine lineare.
- **Nota importante:** In assenza di ulteriori informazioni sui dati, **ogni elemento deve essere considerato almeno una volta** nel caso generale.
  Questo implica un lower bound $\Omega(n)$
  Non possiamo fare asintoticamente meglio nel caso generale.

Ricerca Binaria - Idea generale

Condizione: l'algoritmo si applica a insiemi ordinati.

Idea di base: divide l'intervallo di ricerca a metà ad ogni passo.

Algoritmo:

1. Iniziamo con l'intervallo $[0, n-1]$.
2. Calcoliamo l'indice centrale: $\mathit{medio} = \frac{\mathit{inizio}+\mathit{fine}}{2}$
3. Confrontiamo $\mathit{lista}[\mathit{medio}]$ con l'elemento cercato x:
   - Se $\mathit{lista}[\mathit{medio}] = x$, l'elemento è trovato.
   - Se $\mathit{lista}[\mathit{medio}] < x$, cerchiamo nella metà destra: $\mathit{inizio} = \mathit{medio} + 1$.
   - Se $\mathit{lista}[\mathit{medio}] > x$, cerchiamo nella metà sinistra: $\mathit{fine} = \mathit{medio} - 1$.
4. Ripetiamo finché $\mathit{inizio} > \mathit{fine}$ oppure troviamo x.

Ricerca Binaria - Codice Iterativo

Implementazione iterativa:

```
def Ricerca_Binaria(lista, x):
    """ Esegue la ricerca binaria in modo iterativo.
    Parametri: lista ordinata e l'elemento x da cercare.
    Ritorna: l'indice dell'elemento se trovato, altrimenti None.
    """
    inizio = 0
    fine = len(lista) - 1
    while inizio <= fine:
        # calcola la posizione centrale  dell'intervallo
        medio = (inizio + fine) // 2
        if lista[medio] == x:
            # Elemento trovato
            return medio
        elif lista[medio] < x:
            # Cerca nella metà destra
            inizio = medio + 1
        else:
            # Cerca nella metà sinistra
            fine = medio - 1
  # Elemento non trovato
    return None
```

Ricerca Binaria - Codice ricorsivo

Implementazione ricorsiva: invece di un ciclo, la funzione si richiama ricorsivamente su sottointervalli più piccoli.

Chiamata iniziale: *Ricerca_Binaria_Ric*(*lista*, x, 0, *len*(*lista*) $-$ 1)

```
def Ricerca_Binaria_Ric(lista, x, inizio, fine):
    """ Ricerca binaria ricorsiva.
    Parametri: lista ordinata, elemento x da cercare,
               e intervallo [inizio, fine].
    Ritorna: indice dell'elemento se trovato, altrimenti None.
    """
    if inizio > fine:
        # intervallo vuoto, elemento non trovato
        return None
    # posizionati nel mezzo dell'intervallo
    medio = (inizio + fine) // 2
    if lista[medio] == x:
        # elemento trovato
        return medio
    elif lista[medio] < x:
        # Cerca nella metà destra
        return Ricerca_Binaria_Ric(lista, x, medio + 1, fine)
    else:
        # Cerca nella metà sinistra
        return Ricerca_Binaria_Ric(lista, x, inizio, medio - 1)
```

Ricerca Binaria - Complessità nei casi estremi

Consideriamo i due scenari classici: caso ottimo e caso pessimo.

- **Caso ottimo.** Si verifica quando l'elemento da cercare si trova esattamente al centro della lista ordinata. In questo caso, l'algoritmo lo individua al primo confronto e la complessità è $O(1)$.

- **Caso pessimo.** Si verifica quando l'elemento non è presente nella lista. In questo caso, l'intervallo di ricerca viene dimezzato ad ogni iterazione fino a ridursi a un solo elemento.
  Dopo k iterazioni, la dimensione dell'intervallo è $\frac{n}{2^k}$.
  Il ciclo termina quando $\frac{n}{2^k} = 1$, da cui:

$$n = 2^k \quad \Rightarrow \quad k = \log_2 n$$

  Pertanto, la complessità nel caso pessimo è $O(\log n)$

Nei casi in cui il caso migliore e quello peggiore differiscono significativamente, è naturale chiedersi quale sia la complessità nel **caso medio**.
La analizzeremo nella prossima slide.

Ricerca Binaria - Complessità nel caso medio (1/2)

- **Motivazione:** Poiché caso migliore e caso peggiore non hanno lo stesso costo, è naturale analizzare il **caso medio**.
- **Assunzioni:**
  - Il numero di elementi è una potenza di 2 (per semplicità; non cambia il risultato asintotico).
  - L'elemento cercato ha probabilità uniforme di trovarsi in qualsiasi posizione della lista.
- **Osservazione:** quante sono le posizioni $n(i)$ raggiungibili alla i-esima iterazione:
  - Iterazione 1: si raggiunge solo la posizione $n/2$, quindi $n(1) = 2^0 = 1$.
  - Iterazione 2: si raggiungono due posizioni $(n/4, 3n/4)$, quindi $n(2) = 2^1 = 2$.
  - Iterazione 3: si raggiungono quattro posizioni $(n/8, 3n/8, 5n/8, 7n/8)$, quindi $n(3) = 2^2 = 4$.
  - ...
- In generale, si eseguono i confronti se x si trova in una delle $n(i) = 2^{i-1}$ posizioni raggiungibili in i iterazioni.

Ricerca Binaria - Complessità nel caso medio (2/2)

- **Probabilità:** che x si trovi in una di queste posizioni è:

$$\frac{n(i)}{n} = \frac{2^{i-1}}{n}$$

- **Numero medio di confronti:**

$$E = \sum_{i=1}^{\log n} i \cdot \frac{2^{i-1}}{n} = \frac{1}{n} \sum_{i=1}^{\log n} i \cdot 2^{i-1}$$

- Poiché:

$$\sum_{i=1}^{k} i \cdot 2^{i-1} = (k-1) \cdot 2^k + 1$$

- Otteniamo:

$$E = \frac{1}{n} \left((\log n - 1) \cdot 2^{\log n} + 1 \right) = \log n - 1 + \frac{1}{n}$$

- **Conclusione:** il numero medio di confronti si discosta di meno di uno dal numero massimo, quindi la complessità nel caso medio è $O(\log n)$.

Confronto tra Ricerca Lineare e Binaria

- La ricerca lineare è più semplice e non richiede un insieme ordinato, ma può risultare inefficiente per insiemi di grandi dimensioni.

- La ricerca binaria è molto più efficiente, ma richiede un pre-ordinamento dell'insieme.

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. Sia dato un vettore A di interi ed un intero x. Il problema consiste nel determinare il numero di volte che x compare in A.
   - Progettare un algoritmo che risolva il problema per qualsiasi vettore A.
   - Progettare un algoritmo più efficiente del precedente, assumendo che A sia già ordinato.

   Per ciascun algoritmo, descrivere a parole l'idea alla base della soluzione, scrivere lo pseudocodice e analizzare il tempo di esecuzione asintotica.

2. Sia dato un vettore A di interi maggiori o uguali a 0. Il problema consiste nel determinare se in A sono presenti elementi duplicati.
   - Progettare un algoritmo che risolva il problema per qualsiasi vettore A.
   - Progettare un algoritmo più efficiente del precedente, assumendo che A sia già ordinato.

   Per ciascun algoritmo, descrivere a parole l'idea alla base della soluzione, scrivere lo pseudocodice e analizzare il tempo di esecuzione asintotica.

3. Sia dato un vettore A di numeri interi e due valori a e b con $a \leq b$. Il problema consiste nel determinare quanti elementi di A sono compresi nell'intervallo chiuso $[a, b]$.
   - Progettare un algoritmo che risolva tale problema per qualsiasi vettore A.
   - Progettare un algoritmo più efficiente del precedente, assumendo che A sia già ordinato e contenga solo valori distinti.

   Per ciascun algoritmo, descrivere a parole l'idea alla base della soluzione, scrivere lo pseudocodice e analizzare il tempo di esecuzione asintotica.

Esercizi

1. Sia dato un vettore A di numeri interi e un valore x. Il problema consiste nel determinare se esistono due elementi distinti di A la cui somma sia esattamente x.
   - Progettare un algoritmo che risolva tale problema per qualsiasi vettore A.
   - Progettare un algoritmo più efficiente del precedente, assumendo che A sia già ordinato.

   Per ciascun algoritmo, descrivere a parole l'idea alla base della soluzione, scrivere lo pseudocodice e analizzare il tempo di esecuzione asintotica.
2. Sia dato un vettore A di numeri interi e due valori a e b con $a \leq b$. Il problema consiste nel determinare quanti elementi di A sono compresi nell'intervallo chiuso $[a, b]$.
   - Progettare un algoritmo che risolva tale problema per qualsiasi vettore A.
   - Progettare un algoritmo più efficiente del precedente, assumendo che A sia già ordinato e contenga solo valori distinti.

   Per ciascun algoritmo, descrivere a parole l'idea alla base della soluzione, scrivere lo pseudocodice e analizzare il tempo di esecuzione asintotica.
3. Sia dato un vettore A di numeri interi e un valore x. Il problema consiste nel determinare il minimo valore maggiore di x presente in A, o restituire *None* nel case in cui tale elemento non esista.
   - Progettare un algoritmo che risolva tale problema per qualsiasi vettore A.
   - Progettare un algoritmo più efficiente del precedente, assumendo che A sia già ordinato.

   Per ciascun algoritmo, descrivere a parole l'idea alla base della soluzione, scrivere lo pseudocodice e analizzare il tempo di esecuzione asintotica.