Corso di Laurea in Informatica e Tecnologie per la Produzione del Software *(Track B)* - A.A. 2018/2019

Laboratorio di Informatica

Algoritmi Fondamentali

(Parte 1)

docente: Cataldo Musto

cataldo.musto@uniba.it

Slides ispirate ai contenuti proposti
dal dott.Pasquale Lops. Grazie.

Algoritmi Fondamentali

- Cioè?

Algoritmi Fondamentali

- Cioè?
- Algoritmi che risolvono **problemi «comuni»**
  - Si tratta di soluzioni «standard», riconosciute come **«corrette»**
  - Algoritmi di fondamentale importanza nelle attività di programmazione
  - Descrivono **le soluzioni ottimali** per risolvere un determinato problema

Algoritmi Fondamentali

- Cioè?
- Algoritmi che risolvono **problemi «comuni»**
  - Si tratta di soluzioni «standard», riconosciute come **«corrette»**
  - Algoritmi di fondamentale importanza nelle attività di programmazione
  - Descrivono le **soluzioni ottimali** per risolvere un determinato problema

- I più diffusi risolvono task di **ordinamento** e **ricerca** dei dati
  - Non sono gli unici algoritmi fondamentali
  - Sono gli unici che studieremo nel corso ☺

Algoritmi Fondamentali

- **Non esiste un'unica soluzione** che risolve un determinato problema
- Come facciamo a dire **quale tra due soluzioni è quella ottimale**?

Algoritmi Fondamentali

- **Non esiste un'unica soluzione** che risolve un determinato problema
- Come facciamo a dire **quale tra due soluzioni è quella ottimale**?
  - **Complessità computazionale** degli algoritmi
  - Ciascun algoritmo ha la propria **complessità computazionale**
  - Tipicamente, gli algoritmi più efficienti (complessità computazionale più bassa) hanno una più alta complessità implementativa.
    - **Gli algoritmi più semplici da implementare sono i meno efficienti**

Algoritmi Fondamentali

- **Non esiste un'unica soluzione** che risolve un determinato problema
- Come facciamo a dire **quale tra due soluzioni è quella ottimale**?
  - **Complessità computazionale** degli algoritmi
  - Ciascun algoritmo ha la propria **complessità computazionale**
  - Tipicamente, gli algoritmi più efficienti (complessità computazionale più bassa) hanno una più alta complessità implementativa.
    - **Gli algoritmi più semplici da implementare sono i meno efficienti**

- La differenza di complessità si percepisce soprattutto in casi reali (es. la ricerca su Google). **Per piccole quantità di dati, le differenze sono impercettibili**

Complessità di un Algoritmo

- **Misura di quanto è «complesso» per un elaboratore eseguire quell'algoritmo**
  - **Quantità di risorse usate** dall'algoritmo

- **Di che risorse** parliamo?

Complessità di un Algoritmo

- **Misura di quanto è «complesso» per un elaboratore eseguire quell'algoritmo**
  - **Quantità di risorse usate** dall'algoritmo

- **Di che risorse** parliamo?
  - **Spazio**
    - Quantità di memoria occupata durante l'esecuzione
  - **Tempo**
    - Quantità di tempo impiegata per ottenere la soluzione

Complessità di un Algoritmo

- **Misura di quanto è «complesso» per un elaboratore eseguire quell'algoritmo**
  - **Quantità di risorse usate** dall'algoritmo
- **Di che risorse** parliamo?
  - **Spazio**
    - Quantità di memoria occupata durante l'esecuzione
  - **Tempo**
    - Quantità di tempo impiegata per ottenere la soluzione
      - Calcolabile in base al numero di volte in cui viene ripetuta l'operazione principale
        - Esempio: Confronti, Scambi, Addizioni, …
- **Minori** sono le risorse usate da un algoritmo, **minore sarà la sua complessità computazionale.**

Complessità di un Algoritmo

- **Come misuriamo il tempo necessario ad eseguire un algoritmo?**

Complessità di un Algoritmo

- **Come misuriamo il tempo necessario ad eseguire un algoritmo?**
  - Per convenzione si calcola numero di volte in cui viene ripetuta l'operazione principale
  - **Esempio:** Confronti, Scambi, Addizioni, …

```
for(int i=0; i<n; i++) {
        // Qual è la complessità di questo codice?
}
```

Complessità di un Algoritmo

- **Come misuriamo il tempo necessario ad eseguire un algoritmo?**
  - Per convenzione si calcola numero di volte in cui viene ripetuta l’operazione principale
  - **Esempio:** Confronti, Scambi, Addizioni, …

```
for(int i=0; i<n; i++) {
        // si dice che questo codice ha complessità «n»
        // perché il frammento viene eseguito n volte
}
```

Complessità di un Algoritmo - Esempio

```
for(int i=0; i<n; i++) {
// si dice che questo codice ha complessità «n»
// perché il frammento viene eseguito n volte
}
```

```
int i=0;
for(int i=0; i<n; i++) {
}
```

Che complessità ha questo codice?

Complessità di un Algoritmo - Esempio

```
for(int i=0; i<n; i++) {
// si dice che questo codice ha complessità «n»
// perché il frammento viene eseguito n volte
}
```

```
int i=0;
for(int i=0; i<n; i++) {
}
```

Sarebbe n+1.
Ma in realtà per convenzione si dice che la complessità è sempre **«n»**

Complessità di un Algoritmo - Esempio

```
for(int i=0; i<n; i++) {
// si dice che questo codice ha complessità «n»
// perché il frammento viene eseguito n volte
}
```

```
int i=0;
for(int i=0; i<n; i++) {
}
```

Quando effettuiamo calcoli di complessità in realtà calcoliamo delle **approssimazioni.**

Supponendo che il valore di n sia «grande», la differenza tra n ed n+1 è insignificante quindi l'istruzione che ha complessità «1» si può ignorare.

Complessità di un Algoritmo - Esempio

```
for(int i=0; i<n; i++) {
// si dice che questo codice ha complessità «n»
// perché il frammento viene eseguito n volte
}
```

```
int i=0;
for(int i=0; i<n; i++) {
}
```

Tipicamente quando si calcola la complessità si **cercano le istruzioni «dominanti»,** cioè quelle che vengono eseguite più volte.

Spesso (non sempre!) la complessità degli algoritmi è **data dal numero di operazioni che vengono effettuate nei cicli.**

Complessità di un Algoritmo

- **Complessità crescente:**
  - Costante O(*1*)
  - Logaritmica O(log n)
  - Lineare O(n)
  - nlog O(n log n)
  - Polinomiale O(n^m)
    - Quadratica O(n^2)
    - Cubica O(n^3)
    - …
  - Esponenziale O(k^n) $k > 1$

Complessità di un Algoritmo

- **Complessità crescente:**
  - Costante $O(1)$ ← Complessità **ottimale**
  - Logaritmica $O(\log n)$
  - Lineare $O(n)$
  - nlog $O(n \log n)$
  - Polinomiale $O(n^m)$
    - Quadratica $O(n^2)$
    - Cubica $O(n^3)$
    - …
  - Esponenziale $O(k^n)$ $k > 1$

Complessità di un Algoritmo
Livelli

- **Complessità crescente:**
  - Costante $O(1)$ ← Complessità **ottimale**
  - Logaritmica $O(\log n)$
  - Lineare $O(n)$
  - nlog $O(n \log n)$
  - Polinomiale $O(n^m)$
    - Quadratica $O(n^2)$
    - Cubica $O(n^3)$
    - …
  - Esponenziale $O(k^n)$ $k > 1$

Le operazioni collegate agli operatori presenti nel linguaggio (es. assegnazione, confronto, etc.) **hanno tutti complessità costante**, perché basta 1 operazione (1 istruzione) per risolvere il problema.

Complessità di un Algoritmo
Livelli

- **Complessità crescente:**
  - Costante $O(1)$ ← Complessità **ottimale**
  - Logaritmica $O(\log n)$
  - Lineare $O(n)$
  - nlog $O(n \log n)$
  - Polinomiale $O(n^m)$
    - Quadratica $O(n^2)$
    - Cubica $O(n^3)$
    - …
  - Esponenziale $O(k^n)$ $k > 1$ ← **Non trattabili** in modo efficace con un elaboratore

Semplificando, La notazione O(n) serve appunto a dire che la complessità è «approssimata» ad n

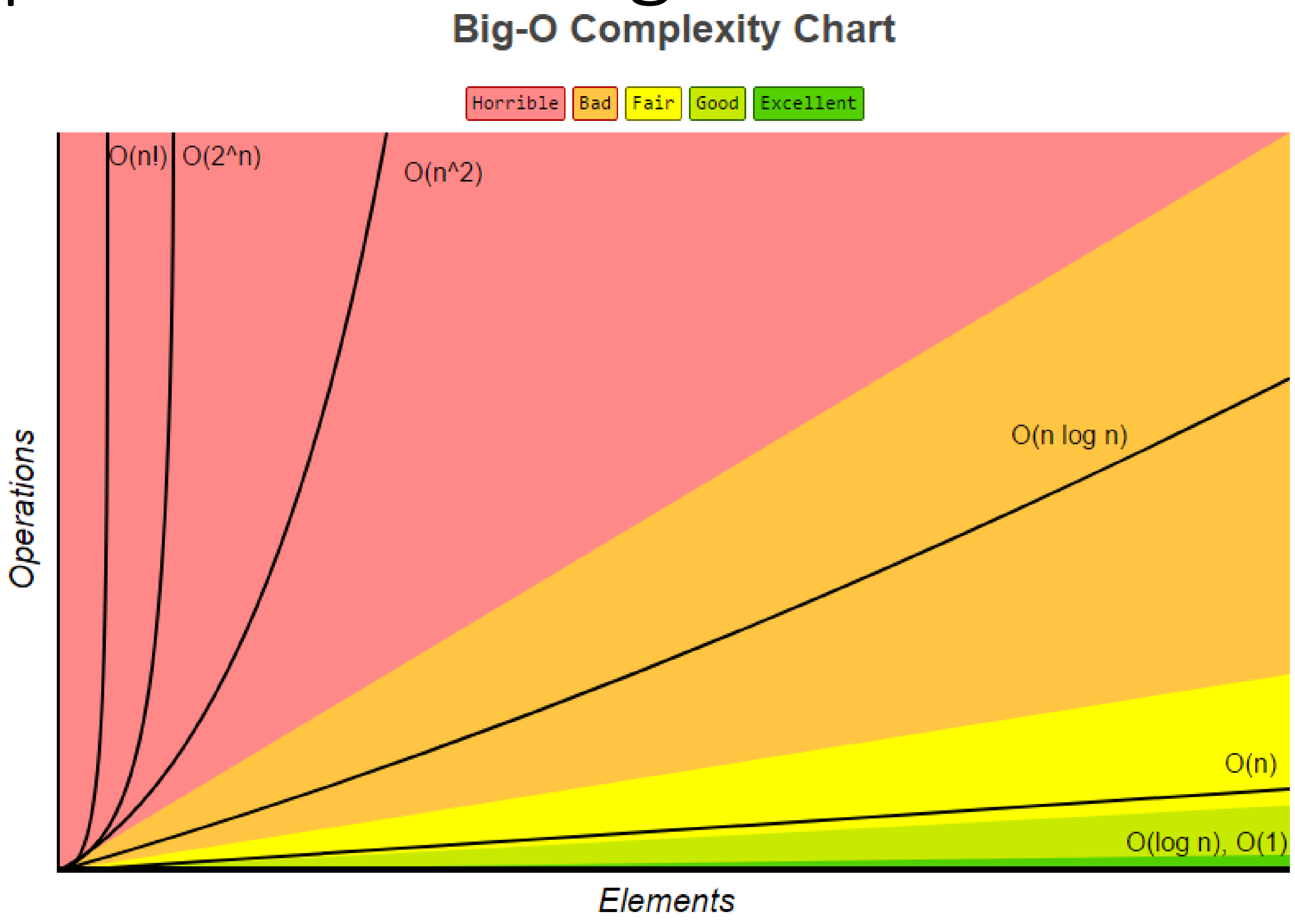
I concetti saranno spiegati in modo rigoroso nei corsi di Informatica Teorica ☺

Complessità di un Algoritmo
Livelli

- **Complessità crescente:**
  - Costante $O(1)$ ← Complessità **ottimale**
  - Logaritmica $O(\log n)$
  - Lineare $O(n)$
  - nlog $O(n \log n)$
  - Polinomiale $O(n^m)$
    - Quadratica $O(n^2)$
    - Cubica $O(n^3)$
    - …
  - Esponenziale $O(k^n)$ $k > 1$ ← **Non trattabili** in modo efficace con un elaboratore

Gli algoritmi più comuni hanno una complessità **polinomiale o lineare**. Gli algoritmi più efficienti hanno una **complessità logaritmica o vicina a quella lineare O(n log n)**

Complessità di un Algoritmo



Complessità di un Algoritmo

- Quando parliamo di complessità computazionale dobbiamo **distinguere diversi casi**
  - **Migliore**
    - Corrispondente alla configurazione iniziale che comporta il **minimo** numero di esecuzioni dell'operazione principale
  - **Peggiore**
    - Corrispondente alla configurazione iniziale che comporta il **massimo** numero di esecuzioni dell'operazione principale
  - **Medio**

Complessità di un Algoritmo

- Quando parliamo di complessità computazionale dobbiamo **distinguere diversi casi**
  - **Migliore**
    - Corrispondente alla configurazione iniziale che comporta il minimo numero di esecuzioni dell'operazione principale
  - **Peggiore**
    - Corrispondente alla configurazione iniziale che comporta il massimo numero di esecuzioni dell'operazione principale
  - **Medio**

- Ad esempio, in un algoritmo di ricerca la complessità computazionale è diversa se l'elemento da trovare è il **primo del vettore** **(caso migliore)**, **l'ultimo del vettore** **(caso peggiore)** o è al **centro del vettore** **(caso medio)**

Complessità di un Algoritmo

- Quando parliamo di complessità computazionale dobbiamo **distinguere diversi casi**
  - **Migliore**
    - Corrispondente alla configurazione iniziale che comporta il minimo numero di esecuzioni dell'operazione principale
  - **Peggiore**
    - Corrispondente alla configurazione iniziale che comporta il massimo numero di esecuzioni dell'operazione principale
  - **Medio**

- Allo stesso modo, in un algoritmo di ordinamento la complessità computazionale è diversa se il vettore è già ordinato **(caso migliore)**, **oppure ordinato in modo opposto** **(caso peggiore)** rispetto a quello che vogliamo.

Algoritmi di Ricerca

Ricerca

- **Problema:** Determinare se (e dove) un certo elemento *x* compare in un certo insieme di *n* dati (ad esempio un array)
  - Supponiamo di avere a disposizione n elementi 1 … *n*

Ricerca

- **Problema:** Determinare se (e dove) un certo elemento *x* compare in un certo insieme di *n* dati (ad esempio un array)
  - Supponiamo di avere a disposizione n elementi 1 … *n*
  - **Possibili esiti:**
    - **Elemento trovato** nell'insieme
      - Restituirne la posizione
      - Il fatto che l'elemento non sia stato trovato è rappresentabile tramite il valore di posizione 0
    - **Elemento non presente** nell'insieme

- **Input:** insieme di dati
- **Output:** posizione

Ricerca

- **Problema:** Determinare se (e dove) un certo elemento *x* compare in un certo insieme di *n* dati (ad esempio un array)
  - Supponiamo di avere a disposizione n elementi 1 … *n*
  - **Possibili esiti:**
    - **Elemento trovato** nell'insieme
      - Restituirne la posizione
      - Il fatto che l'elemento non sia stato trovato è rappresentabile tramite il valore di posizione 0
    - **Elemento non presente** nell'insieme

- **Input:** insieme di dati
- **Output:** posizione

Normalmente una funzione di ricerca
dovrebbe essere basata su questi parametri!
Es.) `int ricerca(int valore, int vettore[ ], int n)`

Ricerca Lineare Esaustiva

- **Scorrimento di tutti gli elementi dell'insieme**, memorizzando eventualmente la posizione in cui l'elemento è stato trovato
  - **Nessuna ipotesi di ordinamento**
    - L'algoritmo è applicabile anche per insiemi non ordinati
  - Utilizzabile quando si può accedere in sequenza agli elementi della lista

Ricerca Lineare Esaustiva - Algoritmo

- **Scandisce tutti gli elementi della lista**
  - Restituisce l'ultima (posizione di) occorrenza
  - Utile quando si vogliono ritrovare tutte le occorrenze del valore

```
j ← 0
posizione ← 0
finche j < n
      se array[j] = x allora posizione ← j
      altrimenti j ← j + 1
```

Note
Cosa succede se un elemento è presente più volte?

Ricerca Lineare Esaustiva - Algoritmo

- **Scandisce tutti gli elementi della lista**
  - Restituisce l’ultima (posizione di) occorrenza
  - Utile quando si vogliono ritrovare tutte le occorrenze del valore

```
j ← 0
posizione ← 0
finche j < n
      se array[j] = x allora posizione ← j
      altrimenti j ← j + 1
```

Note
Se un elemento è presente più volte, restituisce solo l’ultima **posizione**

Ricerca Lineare Esaustiva– Programma C

```
int ricerca(int a[ ], int n, int j) {
  j = 0;
  posizione = 0;
  while (j < n) {
        if ( a[j] == x )
             posizione = j;
       j = j + 1;

  }
}
```

Codice non completo ☺

Ricerca Lineare Esaustiva - Complessità

- **Complessità**
  - Basata sul numero di confronti (cioè sul numero di **cicli effettuati**)
    - Caso migliore:
    - Caso peggiore:
    - Caso medio:

Ricerca Lineare Esaustiva - Complessità

- **Complessità**
  - Basata sul numero di confronti (cioè sul numero di **cicli effettuati**)
    - Caso migliore: **O**(**n)**
      - **Perché effettua comunque tutti i cicli**
    - Caso peggiore: **O**(**n)**
      - Si devono controllare comunque tutti gli elementi fino all’ultimo
    - Caso medio: **(*n* + 1) / 2 → O(n)**
      - Supponendo una distribuzione casuale dei valori

Ricerca Lineare Esaustiva - Considerazioni

- **Complessità**
  - Basata sul numero di confronti (cioè sul numero di **cicli effettuati**)
    - Caso migliore: **O(n)**
      - **Perché effettua comunque tutti i cicli**
    - Caso peggiore: **O(n)**
      - Si devono controllare comunque tutti gli elementi fino all'ultimo
    - Caso medio: **(*n* + 1) / 2 → O(n)**
      - Supponendo una distribuzione casuale dei valori

- **Possiamo migliorare l'algoritmo?**
  - A volte non interessa scandire tutta la lista
    - Ci si può fermare appena l'elemento viene trovato

Ricerca Lineare con Sentinella - Algoritmo

- **Si ferma alla prima occorrenza del valore**
  - Restituisce **la prima (posizione di) occorrenza**
  - Utile quando
    - **Si è interessati solo all'esistenza**, oppure
    - Il valore, **se esiste, è unico**

```
j ← 0
posizione ← -1
mentre (j < n) e (posizione < 0)
  se lista(j) = x allora posizione ← j
  j ← j + 1
```

Ricerca Lineare con Sentinella - Programma C

```
int ricerca(int a[ ], int n, int j) {
  j = 0;
  posizione = -1;
  while ((j < n) && (posizione < 0)) {
        if ( a[j] == x )
             posizione = j;
       j = j + 1;
  }
  return posizione;
}
```

Codice non completo ☺

Ricerca Lineare con Sentinella - Programma C

```
int ricerca(int a[ ], int n, int j) {
  j = 0;
  posizione = -1;
  while ((j < n) && (!trovato)) {
        if ( a[j] == x ) {
              posizione = j;
              trovato = true;
        }
        j = j + 1;
  }
  return posizione;
}
```

Implementazione alternativa con sentinella

Ricerca Lineare con Sentinella
Considerazioni

- Complessità
  - Basata sul numero di confronti
    - **Caso migliore:**
    - **Caso peggiore:**
    - **Caso medio:**
- Ottimizzato per i casi in cui è applicabile

Ricerca Lineare con Sentinella
Considerazioni

- Complessità
  - Basata sul numero di confronti
    - **Caso migliore: O(1)**
      - Elemento trovato in prima posizione
    - **Caso peggiore: O(*n*)**
      - Elemento in ultima posizione o assente
    - **Caso medio: (*n* + 1) / 2 → O(n)**
      - Supponendo una distribuzione casuale dei valori
- Ottimizzato per i casi in cui è applicabile

Ricerca Binaria (o Dicotomica)

- Algoritmo di ricerca **più efficiente**
  - **Complessità computazionale più bassa**
- **Vincolo:** applicabile a insiemi di **dati ordinati**
  - Guadagno in efficienza, ma richiede eventualmente anche l'applicazione di un algoritmo di ordinamento

Ricerca Binaria (o Dicotomica)

- Algoritmo di ricerca **più efficiente**
  - **Complessità computazionale più bassa**
- **Vincolo:** applicabile a insiemi di **dati ordinati**
  - Guadagno in efficienza, ma richiede eventualmente anche l'applicazione di un algoritmo di ordinamento

- **Idea:** confrontare il valore cercato con quello al centro della lista, e se non è quello cercato, basarsi sul confronto per escludere la parte superflua e concentrarsi sull'altra parte

Ricerca Binaria (o Dicotomica) - Esempio

***x* = 29**

Vettore iniziale | 2 | 4 | 7 | 11 | 24 | 25 | 29 | 32 | 38 | 44 | 53 | 61 |

Ricerca Binaria (o Dicotomica) - Esempio

***x* = 29**

Vettore iniziale | 2 | 4 | 7 | 11 | 24 | 25 | 29 | 32 | 38 | 44 | 53 | 61 |

I tentativo ~~| 2 | 4 | 7 | 11 | 24 | 25 |~~ 29 | 32 | 38 | 44 | 53 | 61 |

↑ Eliminati 6

Ricerca Binaria (o Dicotomica) - Esempio

***x* = 29**

Vettore iniziale | 2 | 4 | 7 | 11 | 24 | 25 | 29 | 32 | 38 | 44 | 53 | 61 |

I tentativo ~~| 2 | 4 | 7 | 11 | 24 | 25 |~~ 29 | 32 | 38 | 44 | 53 | 61 |

↑ Eliminati 6

II tentativo | 2 | 4 | 7 | 11 | 24 | 25 | 29 | 32 | ~~38 | 44 | 53 | 61 |~~

↑ Eliminati 4

III tentativo | 2 | 4 | 7 | 11 | 24 | 25 | 29 | 32 | 38 | 44 | 53 | 61 |

↑ Trovato!

- **X=31:** non trovato in 4 tentativi

Ricerca Binaria (o Dicotomica)

Pseudocodice

Se l'elemento centrale è quello cercato

allora è l'elemento cercato stato trovato in quella posizione

altrimenti

se è minore di quello cercato **allora**

analizzare la metà lista successiva

altrimenti

analizzare la metà lista precedente

Ricerca Binaria (o Dicotomica)

Pseudocodice

Se l'elemento centrale è quello cercato

allora è l'elemento cercato stato trovato in quella posizione

altrimenti

se è minore di quello cercato **allora**

analizzare la metà lista successiva

altrimenti

analizzare la metà lista precedente

L'idea intuitiva deve essere inserita in un ciclo.

Quando esce dal ciclo l'algoritmo?

Ricerca Binaria (o Dicotomica)

L’idea intuitiva deve essere inserita in un ciclo.

Quando esce dal ciclo l’algoritmo?

Se l’elemento viene trovato
Oppure
Se l’elemento non c’è

Pseudocodice

Se l’elemento centrale è quello cercato

allora è l’elemento cercato stato trovato in quella posizione

altrimenti

se è minore di quello cercato **allora**

analizzare la metà lista successiva

altrimenti

analizzare la metà lista precedente

Ricerca Binaria (o Dicotomica)

Pseudocodice

Se l'elemento centrale è quello cercato

allora è l'elemento cercato stato trovato in quella posizione

altrimenti

se è minore di quello cercato **allora**

analizzare la metà lista successiva

altrimenti

analizzare la metà lista precedente

L'idea intuitiva deve essere inserita in un ciclo.

Quando esce dal ciclo l'algoritmo?

Se l'elemento viene trovato
Oppure
Se l'elemento non c'è

Quando capiamo che un elemento non c'è? Se la lista che resta da analizzare è composta da un solo elemento!

Ricerca Binaria (o Dicotomica)

Pseudocodice

Finchè la parte di lista da analizzare contiene più di un elemento
e quello cercato non è stato trovato
Se l'elemento centrale è quello cercato
allora è stato trovato in quella posizione
altrimenti se è minore di quello cercato
allora
analizzare la metà lista successiva
altrimenti
analizzare la metà lista precedente

Ricerca Binaria (o Dicotomica)

- **Problema:** Scelta della posizione da analizzare
  - Più vicina ad uno dei due estremi
    - Caso migliore: restringe più velocemente il campo
    - Caso peggiore: elimina sempre meno elementi
  - **Centrale**
    - Compromesso che bilancia al meglio i casi possibili

- Necessità **di ricordare la porzione valida**
  - Prima posizione
  - Ultima posizione
  - Al primo ciclo dell'algoritmo corrisponde agli estremi dell'intero vettore, **poi si restringe**

Ricerca Binaria (o Dicotomica)

```
int ricerca(int a[ ], int n, int j) {
  posizione = -1;
  first = 0; last = n-1;
  while ((first <= last) and (posizione = -1)) {
        j = (first + last) / 2;  // arrotondato per difetto
                if (lista[j] = x)
                        posizione = j;
                else
                        if( x>lista[j] ) // se il cercato è maggiore del mediano
                                first = j + 1;
                else last = j – 1;
        }
 return posizione;
}
```

Ricerca Binaria (o Dicotomica)

- **Complessità**
  - Numero minimo di accessi:
    - Valore trovato al centro della lista
  - Numero massimo di accessi:

Ricerca Binaria (o Dicotomica)

- **Complessità**
  - Numero minimo di accessi: **1**
    - Valore trovato al centro della lista
  - Numero massimo di accessi: $\mathbf{log_2}\ \boldsymbol{n}\ \mathbf{+ 1}$

Ricerca Binaria (o Dicotomica)

- **Complessità**
  - Numero minimo di cicli: **1**
    - Valore trovato al centro della lista
  - Numero massimo di cicli: $\mathbf{\log_2 \boldsymbol{n} + 1}$
    - Esempio: $n = 128$
    - Primo ciclo = 128 elementi, secondo ciclo = 64…. Settimo ciclo = 2, Ottavo ciclo = 1
    - Massimo 7 cicli → $7 = \log_2 n = 7$

Ricerca Binaria (o Dicotomica)

- **Complessità**
  - Numero minimo di cicli: **1**
    - Valore trovato al centro della lista
  - Numero massimo di cicli: $\mathbf{\log_2 \boldsymbol{n} + 1 = O(\log_2 \boldsymbol{n})}$
    - Esempio: $n = 128$
    - Primo ciclo = 128 elementi, secondo ciclo = 64…. Settimo ciclo = 2, Ottavo ciclo = 1
    - Massimo 7 cicli → $7 = \log_2 n = 7$

Ricerca Binaria (o Dicotomica)

- **Complessità**
  - Numero minimo di cicli: **1**
    - Valore trovato al centro della lista
  - Numero massimo di cicli: $\mathbf{\log_2 \mathit{n} + 1 = O(\log_2 \mathit{n})}$
    - Esempio: $n = 128$
    - Primo ciclo = 128 elementi, secondo ciclo = 64…. Settimo ciclo = 2, Ottavo ciclo = 1
    - Massimo 7 cicli → $7 = \log_2 n = 7$

- **Più efficiente della ricerca sequenziale! Ma richiede che i dati siano ordinati in qualche modo (dal più piccolo al più grande, o alfabeticamente)**
  - Usata per consultare dizionari, elenchi telefonici.. Etc …Non adatta a scenari in cui non è presente un «ordinamento» tra i dati! (es. un catalogo prodotti o un catalogo musicale)

Esercizio 13.1

- Realizzare una funzione C che implementi l'algoritmo di **ricerca sequenziale con sentinella oppure di ricerca binaria**
- Richiamare la funzione in un `main()` , contenente un vettore di valori e stampare in output il risultato della ricerca
- **(A casa)** Testare il programma con una suite di test `CUnit`

Domande?

©Bojack Horseman