Informatica per Statistica
Riassunto della lezione del 31/10/2012

Igor Melatti

Alcuni algoritmi ricorsivi

- Problema della ricerca

  input: sequenza di n numeri $\langle a_1, \ldots, a_n \rangle$ e un numero k

  output: intero $i \in [0, n]$ t.c., detto $K = \{j \in [1, n] \mid a_j = k\}$ l'insieme degli indici della sequenza di input che identificano numeri uguali a k, allora $i = \min K$ se $K \neq \emptyset$, e $i = 0$ altrimenti

  - quindi si vuole cercare k dentro la sequenza, se lo si trova allora si deve dare in output il più piccolo indice di un elemento uguale a k (potrebbero essercene più d'uno), o 0 se k non è affatto presente

- La dimensione dell'input è la lunghezza della sequenza, ovvero n
- L'algoritmo *Search* di Figura 1 risolve questo problema

```
1 Search(A, k)
2   for i ← 1 to length(A) do
3     if (A[i] = k) then
4       return i
5   return 0
```

Figure 1: Ricerca di un valore

- Complessità: $\Theta(n)$ nel caso peggiore
  - nel caso migliore sarebbe $O(1)$: è il caso in cui k è subito presente nel primo elemento dell'array
  - il caso peggiore, invece, è quello in cui k non è presente, e *Search* scorre tutti gli n elementi della sequenza (ogni volta eseguendo un calcolo richiedente tempo costante)
  - da qui il $\Theta(n)$ nel caso peggiore

- **esercizio**: riformulare il problema e l'algoritmo *Search* per far sì che venga tornato non il primo, ma l'ultimo indice di un elemento uguale a k (e sempre 0 se l'elemento non è trovato)
- **esercizio**: implementare l'algoritmo *Search* in C

- Problema della ricerca su sequenze ordinate

  input: sequenza di n numeri $\langle a_1, \ldots, a_n \rangle$ t.c. $\forall i \in [1, n-1]$ si ha che $a_{i+1} \geq a_i$, e un numero k

  - l'input è come sopra, ma in più si sa che la sequenza è già ordinata (crescentemente)

  output: intero $i \in [0, n]$ t.c., detto $K = \{j \in [1, n] \mid a_j = k\}$ l'insieme degli indici della sequenza di input che identificano numeri uguali a k, allora $i = \min K$ se $K \neq \emptyset$, e $i = 0$ altrimenti

  - l'output è come sopra

- La dimensione dell'input è sempre la lunghezza della sequenza, ovvero n
- L'algoritmo *Ordered-Search* di Figura 1 risolve questo problema

```
1 Ordered-Search(A, k)
2   for i ← 1 to length(A) do
3     if (A[i] = k) then
4       return i
5     if (A[i] > k) then
6       return 0
7   return 0
```

Figure 2: Ricerca di un valore su un array ordinato

- **Esercizio**: implementare l'algoritmo *Ordered-Search* in C
- Complessità: $\Theta(n)$ nel caso peggiore
  - nel caso migliore sarebbe $O(1)$: è il caso in cui k è subito presente nel primo elemento dell'array
  - il caso peggiore, invece, è quello in cui k non è presente ed è maggiore di tutti gli elementi della sequenza, e *Search* scorre tutti gli n elementi della sequenza (ogni volta eseguendo un calcolo richiedente tempo costante)
  - da qui il $\Theta(n)$ nel caso peggiore
  - tuttavia, se k non è presente e non è più grande di tutti gli elementi della sequenza, allora *Ordered-Search* impiega meno passi di *Search*

- infatti, mentre *Search* scorre comunque tutto l'array, *Ordered-Search* si ferma e torna 0 non appena k diventa più piccolo dell'elemento $A[i]$ attuale
- in questo caso, è infatti inutile andare avanti: se k è già strettamente minore di $A[i]$, allora sarà strettamente minore (e quindi diverso) anche degli elementi successivi ad $A[i]$, che sono più grandi di $A[i]$ stesso
- in formule, dato che $\forall j > i$ si ha che $A[i] \leq A[j]$, allora $k < A[i]$ implica $k < A[j]$ $(\forall j > i)$
- in particolare, questo implica che c'è un altro caso in cui *Ordered-Search* fa un solo passo, ovvero quando k non è presente ed è minore di tutti gli elementi dell'array
- **esercizio**: riformulare problema ed algoritmo (ed analisi) nel caso in cui l'ordinamento sia quello inverso

• E ora, una soluzione *ricorsiva*: *Binary-Search* (ricerca binaria) di Figura 3

```
1  Binary-Search(A, p, r, k)
2    if (p > r) then
3      return 0
4    q ← ⌊(p+r)/2⌋
5    if (A[q] = k) then
6      return q
7    if (A[q] > k) then
8      return Binary-Search(A, p, q − 1, k)
9    else
10     return Binary-Search(A, q + 1, r, k)
```

Figure 3: Ricerca binaria

- l'idea è la seguente: si confronta k con l'elemento mediano della sequenza, ovvero quello di indice q
- se sono uguali, il problema è risolto, q è l'indice cercato
- se non sono uguali, allora k sarà o strettamente maggiore o strettamente minore di $A[q]$
- se è strettamente minore, allora sfruttando il fatto che la sequenza è ordinata si può essere sicuri che k, se è presente, si troverà nella parte inferiore della sequenza (ovvero, quella con indici minori di q)
- quindi, si può procedere usando ricorsivamente la stessa tecnica alla sottosequenza con indici minori di q

- altrimenti, se è strettamente maggiore, allora sfruttando il fatto che la sequenza è ordinata si può essere sicuri che k, se è presente, si troverà nella parte superiore della sequenza (ovvero, quella con indici maggiori di q)
- quindi, si può procedere ricorsivamente usando la stessa tecnica alla sottosequenza con indici maggiori di q
- da qui l'algoritmo *Binary-Search*
- è chiaro che occorre che abbia come parametri di input, oltre che l'intera sequenza e k, anche gli indici di inizio (p) e fine (r) della sottosequenza da considerare
- da notare che ci sono 2 casi base, uno per quando la sottosequenza da considerare è vuota (righe 2–3) e uno quando l'elemento è stato trovato (righe 5–6)

- **Esercizio**: implementare l'algoritmo *Binary-Search* in C
- La complessità è $O(\log(n))$ nel caso peggiore
  - l'idea è la seguente: ogni volta che si fa una ricorsione (cioè una chiamata ricorsiva), si dimezza la lunghezza della sequenza da esaminare
  - quindi, con j chiamate ricorsive, la lunghezza della sequenza originale si è ridotta di un fattore 2^j
  - quindi, dato che quando si arriva alla sequenza vuota ci si ferma, al più si fanno un numero m di chiamate ricorsive t.c. $2^m = n$
    - a rigore, andrebbe bene solo se n è una potenza di 2
    - bisognerebbe invece dire che m è il più piccolo intero t.c. $2^m \geq n$
    - tuttavia, si può far vedere che questo implica un fattore di correzione al più costante, che viene quindi nascosto dalla notazione O
  - quindi, $m = \log(n)$
  - sempre $O(1)$ nel caso migliore (k viene subito trovato nel punto mediano)

- **Esercizio**: in realtà, *Binary-Search* non risolve esattamente il problema proposto, ma una sua variante; quale?
- Infine, una soluzione ricorsiva al problema dell'ordinamento: *Merge-Sort* (ordinamento per immersione) di Figura 4
  - l'idea è la seguente: si ordinano ricorsivamente la prima e la seconda metà della sequenza
  - dopodiché, si mischiano (procedura *Merge* a riga 8 di Figura 4) le due sottosequenze ordinate così ottenute, in modo da formare un'intera sequenza ordinata

```
1  Merge-Sort(A, p, r)
2    if (p < r) then
3      q ← ⌊(p+r)/2⌋
4      Merge-Sort(A, p, q)
5      Merge-Sort(A, q+1, r)
6      Merge(A, p, q, r)
7
8  Merge(A, p, q, r)
9    i ← 1
10   j ← p
11   m ← q+1
12   while j ≤ q and m ≤ r do
13     if (A[j] < A[m]) then
14       B[i] ← A[j]
15       j ← j+1
16       i ← i+1
17     else
18       B[i] ← A[m]
19       m ← m+1
20       i ← i+1
21   while j ≤ q do
22     B[i] ← A[j]
23     j ← j+1
24     i ← i+1
25   while m ≤ r do
26     B[i] ← A[m]
27     m ← m+1
28     i ← i+1
29   for i ← p to r do
30     A[i] ← B[i−p+1]
```

Figure 4: Merge Sort

    - nota: ovviamente, le due sottosequenze saranno ordinate al loro interno, ma l'intera sequenza non è, in generale, ordinata
    - ad esempio, se la sequenza fosse $\langle 1, 10, 7, 2, 20, 1\rangle$, allora ordinando separatamente le due metà si otterrebbe $\langle 1, 7, 10, 1, 2, 20\rangle$, che non è ordinata
  - come nel caso della ricerca binaria, occorre specificare gli estremi della sottosequenza che si sta ordinando
  - qui il caso base si ha per le sequenze lunghe 1, che per definizione sono già ordinate; in tale occasione, infatti, *Merge-Sort* non fa nulla (fallisce subito il primo `if`), e neanche *Merge* viene chiamata
  - per quanto riguarda la *Merge*, l'idea è la seguente:
    - si sa che le sottosequenze $A[p..q]$ e $A[q+1..r]$ sono già ordinate
    - allora per formare una sequenza ordinata $A[p..r]$ basta prendere alternativamente da una sottosequenza o dall'altra il numero più piccolo
    - a tal proposito, la *Merge* copia in una sequenza *ausiliaria* B (che serve cioè solo per fare il calcolo) elementi o da $A[p..q]$ (righe 14–16) o da $A[q+1..r]$ (righe 18–20), a seconda di quale sottosequenza ha attualmente il valore più piccolo (riga 13)
    - il ciclo `while` di righe 21–24 copia la parte rimanente di $A[p..q]$ in B, nel caso tutti gli elementi di $A[q+1..r]$ siano già stati copiati in B (quindi non c'è più bisogno di nessun confronto)
    - analogamente, il ciclo `while` di righe 25–28 copia la parte rimanente di $A[q+1..r]$ in B, nel caso tutti gli elementi di $A[p..q]$ siano già stati copiati in B (quindi non c'è più bisogno di nessun confronto)
    - infine, la sequenza ordinata B è copiata dentro A (righe 29–30)

- La complessità è $O(n\log(n))$ nel caso peggiore
  - abbozzo di idea: c'è un $O(\log(n))$ per un ragionamento simile al caso della ricerca binaria, mentre il termine $O(n)$ viene dal fatto che *Merge* nel caso peggiore richiede $\Theta(n)$
- Quindi il *Merge-Sort* è migliore dell'*Insertion Sort* (che ordinava in tempo $O(n^2)$ nel caso peggiore)
- Da notare tuttavia che il *Merge-Sort* non ordina *in loco* come l'*Insertion Sort*: oltre che di A, ha bisogno di un ulteriore array B
  - l'*Insertion Sort* richiede una memoria aggiuntiva costante (3 variabili)
  - il *Merge-Sort* (o meglio, la procedura *Merge*) richiede un vettore aggiuntivo B, la cui lunghezza nella prima chiamata sarà n, ovvero tanto quanto A

- **Esercizio**: migliorare lo pseudocodice per la procedura *Merge*
  - facile: rivedere l'incremento di i alle righe 16 e 20
  - meno facile: riorganizzare la procedura in modo che faccia un solo ciclo su B, seguito poi dal ciclo che copia B in A (righe 29–30)
  - spiegare come mai, in ogni caso, la complessità della procedura *Merge* è comunque $\Theta(n)$
- **Esercizio**: come occorre chiamare *Merge-Sort* per ordinare l'intera sequenza?
- **Esercizio**: realizzare un *wrapper* per *Merge-Sort*
  - un wrapper è una funzione scritta per nascondere i dettagli implementativi di un'altra
  - nel caso del *Merge-Sort*, il dover aggiungere i parametri p e r dovrebbe essere nascosto a chi vuole semplicemente usare la funzione (che va visto come un cliente), senza sapere come effettivamente lavora al suo interno
  - un cliente sa che il problema dell'ordinamento ha come input semplicemente la sequenza, e non vuole sapere altro
  - quindi non vuole sapere cosa passare a *Merge-Sort* come valori iniziali di p e r
  - chi lo sa è chi scrive la funzione stessa, che si deve anche preoccupare (se vuol fare le cose precise) di scrivere una funzione wrapper (che potrebbe essere chiamata *Merge-Sort-Ready*), che prenda come input la sola sequenza di numeri, e chiami opportunamente *Merge-Sort*
  - rovesciando la prospettiva, talvolta si scrive prima il wrapper e poi la funzione ricorsiva vera, che ha più parametri per effettuare la ricorsione
  - in questo caso, la funzione ricorsiva (*Merge-Sort*) viene chiamata *helper*
- **Esercizio**: scrivere un algoritmo per il problema della ricerca non ordinata, nel quale prima si ordinano gli elementi e poi si usa la ricerca binaria. Questo approccio conviene?