Laboratorio di algoritmi e strutture dati

Algoritmi di ricerca e di ordinamento*

Docente: Violetta Lonati

1 Confronto di algoritmi

Si considerino i due algoritmi `AlgoX` and `AlgoY` riportati sotto e si svolgano i seguenti punti.

1. Si tracci algoX (cioè: si simuli l'esecuzione tenendo traccia dello stato del programma) per `table` indicata qui sotto e `x` pari a 14. Si elenchino in particolare tutti i valori della variabile `i` durante l'esecuzione.

2. Si tracci AlgoY (cioè: si simuli l'esecuzione tenendo traccia dello stato del programma) per `table` indicata qui sotto e `x` pari a 14. Si scrivano in particolare i valori della variabili `low`, `mid` e `high` subito dopo l'esecuzione della riga 6 (ogni volta che viene eseguita).

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
table	-9	-1	0	13	14	14	12	29	31	24	36	36	44	44	8

3. Si indichi quali delle seguenti affermazioni sono vere per AlgoX e/o per AlgoY. Per ciascuna affermazione si usi una delle cinque opzioni seguenti: **X** (= l'affermazione è corretta solo per AlgoX), **Y** (= l'affermazione è corretta solo per AlgoY), **X & Y** (= l'affermazione è corretta sia per per AlgoX che per AlgoY), **no** (= l'affermazione non è corretta né per AlgoX né per AlgoY), **non so** (= non so la risposta).

   a) L'algoritmo esamina gli elementi partendo dall'indice minimo al massimo.
   b) L'algoritmo cerca l'elemento massimo di `table`.
   c) Come secondo valore, l'algoritmo restituisce sempre l'indice minore per l'elemento `x`, se questo è contenuto in `table`.
   d) L'algoritmo ordina gli elementi in `table`.
   e) L'algoritmo restituisce tutti gli indici in cui si trova l'elemento `x`.
   f) L'algoritmo esamina tutti gli elementi di `table`.
   g) L'algoritmo è corretto solo se `table` è ordinato.
   h) L'algoritmo alla fine restituisce sempre `false, -1`.

4. Si argomenti se la seguente affermazione è vera o falsa: "AlgoX è più efficiente di AlgoY per cercare un singolo elemento in un vettore".

*Ultimo aggiornamento: 19 ottobre 2022 - 12:31:57

5. Quali nomi più esplicativi si potrebbero dare agli algoritmi AlgoX e AlgoY?

6. È facile modificare AlgoX in modo che restituisca tutti gli indici in cui si trova `x`? Cambia la complessità dell'algoritmo?

7. È facile modificare AlgoY in modo che restituisca tutti gli indici in cui si trova `x`? Cambia la complessità dell'algoritmo?

Listing 1: AlgoX

```
1  func algoX(table []int, x int) (bool, int) {
2    for i, el := range table {
3      if el == x {
4        return true, i
5      }
6    }
7    return false, -1
8  }
```

Listing 2: AlgoY

```
1  func algoY(table []int, x int) (bool, int) {
2    low, high := 0, len(table)-1
3
4    for low <= high {
5      mid := (low + high) / 2
6      if table[mid] < x {
7        low = mid + 1
8      } else if table[mid] > x {
9        high = mid - 1
10     } else {
11       return true, mid
12     }
13   }
14   return false, -1
15 }
```

2 Algoritmi di ordinamento

Gli esercizi di questa parte di scheda si riferiscono a tre algoritmi di ordinamento elementari: l'ordinamento per inserimento (*insertion sort*), l'ordinamento per selezione (*selection sort*) e l'ordinamento per fusione (*mergesort*). Tutti e tre sono basati sui confronti tra elementi. Per semplicità i vettori da ordinare che considereremo conterranno sempre numeri interi.

Potete implementare ogni algoritmo con una funzione e scrivere un main per testarle. Può essere utile scrivere delle funzioni ausiliarie, ad esempio: una funzione che legga da standard input una serie di numeri interi, li memorizzi in un vettore e restituisca restituito; una funzione che crei e restituisca un vettore popolato con numeri interi pseudocasuali.

2.1 Ordinamento per inserimento

Scrivete una funzione che legga da standard input una sequenza di interi distinti terminati da 0, memorizzandoli in un vettore ordinato (valutate se è più opportuno usare un *array* o una *slice*): ogni volta che viene letto un nuovo intero, il vettore viene scorso fino a trovare l'esatta collocazione del numero, quindi si crea lo spazio per il nuovo numero spostando in avanti i numeri successivi già memorizzati.

Questo algoritmo è utile per riempire un vettore mantenendolo ordinato ad ogni passo.

2.2 Ordinamento per selezione

Scrivete una funzione che riceva una slice di interi e la ordini usando l'algoritmo `SelectionSort`: alla fine dell'esecuzione, la slice originaria passata come argomento dovrà risultare ordinata. Può essere utile restituire la stessa slice (ordinata), ad esempio per poterla passare come argomento ad altre funzioni, come in `fmt.Println(selectionSort(v))`.

Versione iterativa La funzione `selectionSortIter( int a[])` ripete la seguente operazioni tante volte quanto è lunga la slice da ordinare: per ogni prefisso di lunghezza `n` (con `n` inizialmente pari alla lunghezza della slice) cerca nel prefisso l'elemento massimo e lo scambia con quello nell'ultima posizione del prefisso. Ad esempio, se il vettore da ordinare è `[15 96 44 22 54 28 83]`, allora:

- il primo scambio sarà tra 96 e 83:
  `[15 83 44 22 54 28 96]`
- il secondo scambio sarà tra 83 e 28:
  `[15 28 44 22 54 83 96]`
- il terzo scambio sarà tra 54 e 54 (nessun effetto):
  `[15 28 44 22 54 83 96]`
- il quarto scambio sarà tra 44 e 22:
  `[15 28 22 44 54 83 96]`
- il quinto scambio sarà tra 28 e 22:
  `[15 22 28 44 54 83 96]`
- il sesto scambio sarà tra 22 e 22 (nessun effetto):
  `[15 22 28 44 54 83 96]`
- il settimo scambio sarà tra 15 e 15 (nessun effetto):
  `[15 22 28 44 54 83 96]`

(nota: l'ultimo passaggio è sempre senza effetto).

Versione ricorsiva Scrivete una versione ricorsiva dello stesso algoritmo: la funzione `selectionSortRec` deve funzionare come segue:

- innanzitutto cerca nel vettore l'elemento massimo e lo sposta nell'ultima posizione;
- poi richiama se stessa ricorsivamente per ordinare i primi $n-1$ elementi del vettore.

La base della ricorsione è data dai vettori di lunghezza 0 o 1, che sono sempre ordinati.

2.3 Algoritmo per fusione

Scrivete una funzione che implementa l'algoritmo ricorsivo `mergeSort`. La funzione:

1. divide il vettore in due sotto-vettori di dimensione circa uguale;
2. ordina il sotto-vettore di sinistra richiamando se stessa;
3. ordina il sotto-vettore di destra richiamando se stessa;
4. integra (*merge*) i due vettori in un vettore ordinato.

La base della ricorsione è, anche qui, data dai vattori di lunghezza 0 o 1, che sono sempre ordinati.

La parte di integrazione (merge) di due vettore ordinati `a1` e `a2` funziona con un vettore di supporto: si scorrono entrambi i vettori da sinistra a destra usando due indicatori `i1` e `i2` rispettivamente; ad ogni passo si confronta `a1[i1]` con `a2[i2]` e si sceglie l'elemento più piccolo, lo si copia nel vettore di supporto (nella prima posizione libera) e si incrementa l'indicatore relativo ad esso. Quando `i1` esce da `a1` oppure `i2` esce da `a2`, la parte rimanente dell'altro vettore viene copiata nel vettore di supporto. Alla fine si copia il contenuto del vettore di supporto nel vettore originale.

Nota: stili di implementazione

Gli algoritmi richiamati sopra possono essere implementati con stili diversi.

- Si può scegliere un'implementazione più "letterale" in cui l'algoritmo in pseudocodice è tradotto letteralmente. Questo stile rende più evidenti le operazioni che si devono svolgere e può aiutare a comprendere e valutare la complessità degli algoritmi.
- Si può preferire un uso più idiomatico del linguaggio, sfruttando ad esempio il doppio assegamento, o notazioni sintattiche come `a[low:high]`. Questo stile richiede una maggiore conoscenza del linguaggio di programmazione usato e può rendere i programmi più sintetici o più leggibili (per chi conosce bene le specificità del linguaggio!); d'altra parte potrebbe portare a sottovalutare i costi di certe operazioni "nascoste".

Riguardate i programmi che avete scritto per valutare che stile/i avete usato e per pensare a possibili variazione nello stile di implementazione.