Laboratorio di Algoritmi e Strutture Dati

Primo esercizio, quarta parte: tail recursive *QuickSort* (punti: 4)

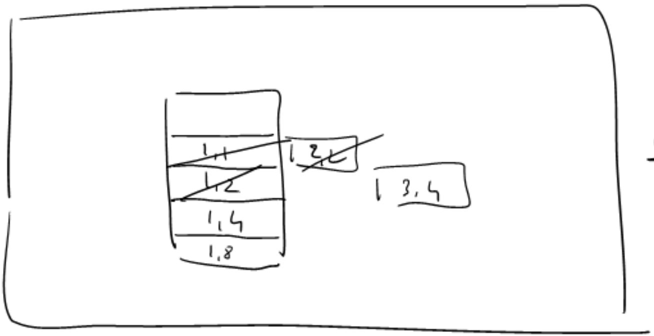
Il problema dello spazio

Abbiamo detto che un algoritmo di ordinamento si dice **in place** quando ogni esecuzione occupa lo stesso spazio in memoria (a parte quello per l'input) indipendentemente dalla lunghezza dell'input stesso. Chiamare *QuickSort* (e altri) in place è impreciso: le chiamate ricorsive occupano spazio in memoria, e il loro numero dipende dalla lunghezza dell'input. Questo è vero anche nella nostra implementazione con *Partition* fatto in place.

Il problema dello spazio

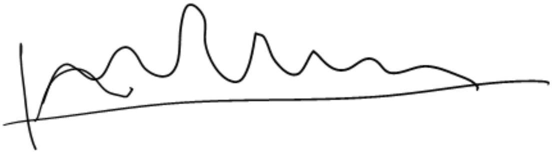
Il problema di occupare spazio per le chiamate ricorsive non è un problema qualsiasi di memoria. Infatti, input troppo lunghi possono dare luogo a **stack overflow**, che dipendono da quanta memoria è stata allocata per lo stack. Questo accade con le implementazioni di algoritmi ricorsivi come *QuickSort*, ed è di questo problema che ci occupiamo adesso. Introduciamo il concetto di **ricorsione tail**.

Quicksort (1,8)



Se sono fortunati, il pivot è sempre bilanciato

⇒ $O(\lg(n))$ è lo spazio occupato sullo stack



Altrimenti può arrivare a $O(n)$

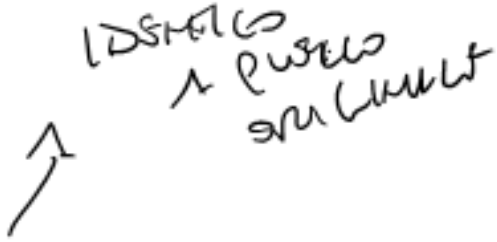
Il problema dello spazio

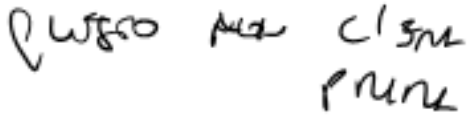
La doppia chiamata ricorsiva di *QuickSort* genera un'occupazione dello stack che dipende dalla bontà della scelta del pivot. Se il pivot non è una buona scelta, allora ci sarà un ramo ricorsivo che fa crescere lo stack in funzione della lunghezza dell'array da ordinare. Assumendo, com'è lecito, dei passaggi per indirizzo, il singolo **frame** occupa spazio costante, pertanto, nel caso peggiore, lo stack si occupa per $O(n)$ spazio. Questo puó essere troppo quando l'array da ordinare è molto grande.

La tail ricorsione

Quando un algoritmo effettua una chiamata ricorsiva come ultimo passaggio, si dice che quella chiamata è **tail ricorsiva**. Un algoritmo è **tail ricorsivo** se tutte le sue chiamate ricorsive lo sono. É *QuickSort* tail ricorsivo? Si. Pertanto, un compilatore ottimizzato (cioè che include la **tail recursion optimization, TRO**, si accorgerebbe di questa situazione, eliminando l'ultima chiamata ricorsiva. Possiamo visualizzare il risultato usando il nostro pseudo-codice, e scrivendo, in partenza, una versione di *QuickSort* comparabile con quella che si otterebbe ottimizzando la ricorsione in coda.

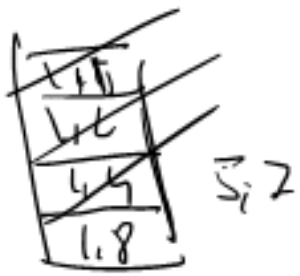
QuickSort tail ricorsivo

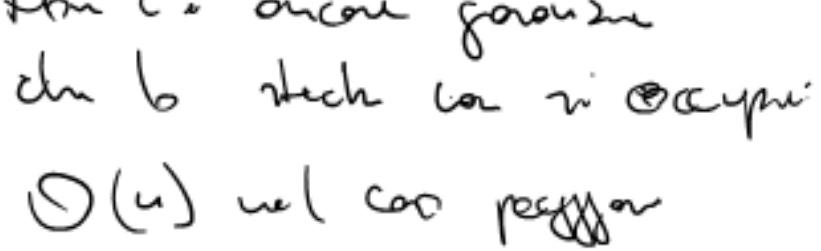




```
proc Partition (A, p, r)
  x = A[r]
  i = p − 1
  for (j = p to r − 1)
    if (A[j] ≤ x)
      then
        i = i + 1
        SwapValue(A, i, j)
  SwapValue(A, i + 1, r)
  return i + 1
```

```
proc TailQuickSort (A, p, r)
  while (p < r)
    do
      q = Partition(A, p, r)
      TailQuickSort(A, p, q − 1)
      p = q + 1
```





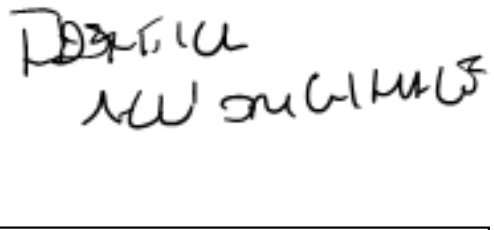
QuickSort tail ricorsivo

Osserviamo che questa versione non è piú tail-ricorsiva: la chiamata ricorsiva rimasta non è piú l'ultima operazione della funzione. Di fatto, l'ottimizzazione automatica delle procedure tail-ricorsive (osserviamo che, in sostanza, questa ottimizzazione ha senso anche per le procedure che non sono ricorsive, ma terminano con una chiamata a funzione), ha senso quando queste sono fondamentalmente semplici: per fissare le idee, si pensi al caso della funzione fattoriale, comunemente usata come esempio prima di ricorsione e poi di eliminazione della stessa ai fini dell'efficienza. *TailQuickSort*, cosí come prodotto da noi, va nella direzione di uso vantaggioso dello stack, ma non ha la garanzia di evitare che questo venga occupato fino a $O(n)$ come nel caso originale.

QuickSort tail ricorsivo ottimizzato

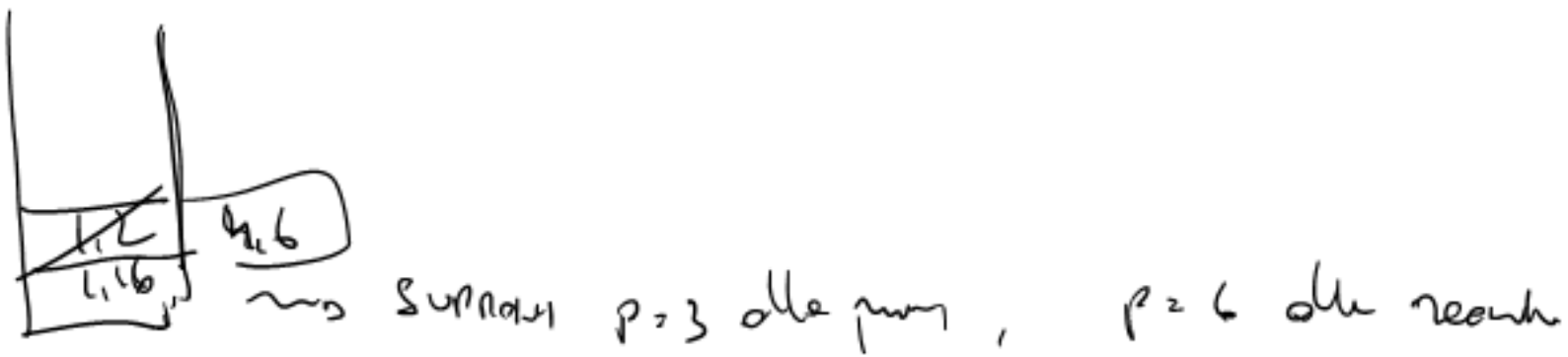
Nel contesto quindi di un'ottimizzazione che non solo è temporale ma anche spaziale di *QuickSort*, proponiamo una versione ulteriormente ottimizzata per minimizzare la quantitá di memoria usata dallo stack. Questa versione è ancora ricorsiva, non tail-ricorsiva, effettua ad ogni chiamata una sola ricorsione, ma si assicura di farlo sul lato piú conveniente dell'array.

QuickSort tail ricorsivo ottimizzato



```
proc Partition (A, p, r)
  x = A[r]
  i = p − 1
  for (j = p to r − 1)
    if (A[j] ≤ x)
      then
        i = i + 1
        SwapValue(A, i, j)
  SwapValue(A, i + 1, r)
  return i + 1
```

```
proc OptimalTailQuickSort (A, p, r)
  while (p < r)
    do
      q = Partition(A, p, r)
      if (q < (p + r)/2)
        then
          OptimalTailQuickSort(A, p, q − 1)
          p = q + 1
        else
          OptimalTailQuickSort(A, q + 1, r)
          r = q − 1
```



QuickSort ottimo

Nelle versioni realmente implementate in librerie comuni di *QuickSort*, ne esiste almeno una che contiene la combinazione di tutte le euristiche viste: eliminazione della tail ricorsione, ottimizzazione della chiamata ricorsiva rimanente, scelta pseudo-ottimale del pivot (per esempio con la mediana di tre), e caso base (per array più piccoli di un certo k) fatto con un algoritmo iterativo, tipicamente *HeapSort*, ma puó anche essere *InsertionSort*. Chiamiamo questa versione **ottima**.

QuickSort ottimo

Vogliamo quindi realizzare un esperimento dove alle implementazioni precedenti aggiungiamo questa versione ottima con tutte e le euristiche menzionate, sia in tempo che in spazio, misurando il tempo di esecuzione sperimentale per input di lunghezza crescente, con multiple ripetizioni per la stessa lunghezza, su tutti gli algoritmi. Il risultato richesto prevede, come sempre, una rappresentazione grafica delle curve di tempo e una dimostrazione sperimentale di correttezza attraverso test randomizzati e funzioni antagoniste.

Conclusione

Scopo di questo primo esercizio di laboratorio era dunque quello di arrivare ad una versione **real life** di un algoritmo noto come è *QuickSort*, per evidenziare la differenza tra teoria e pratica.