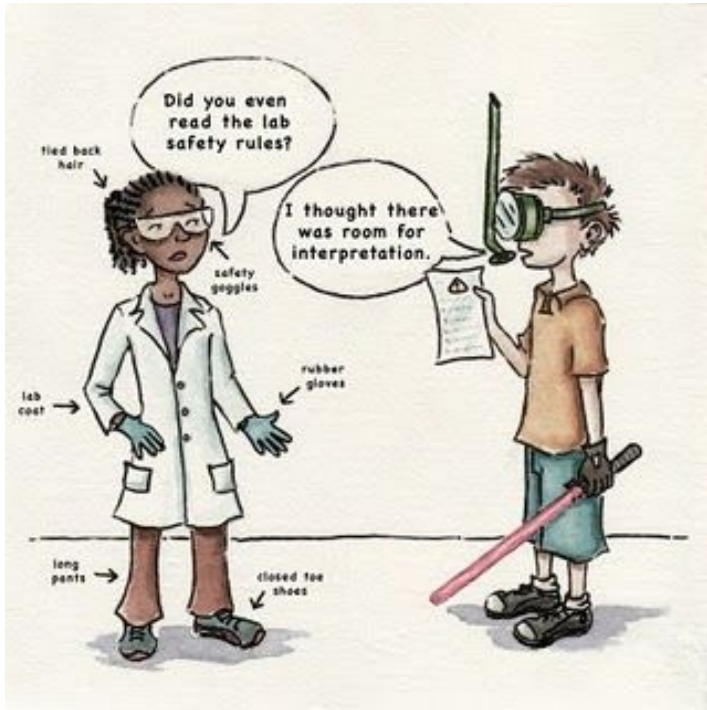
# Laboratorio di Algoritmi e Strutture Dati

## 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**.

# 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? Non come lo conosciamo noi: infatti ci sono due chiamate ricorsive come ultimi passaggi. Puó essere reso tale? Questo non è vero per tutti gli algoritmi ricorsivi non tail, ma lo è per *QuickSort*.

# *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

*QuickSort* tail ricorsivo in questa versione ha un vantaggio: i compilatori piú efficienti si accorgono che l'algoritmo è tail ricorsivo e automaticamente eliminano questa chiamata ricorsiva, producendo un codice oggetto che è essenzialmente **iterativo**. In questo modo, possiamo ignorare il fatto che non abbiamo limitato l'occupazione dello stack (infatti avremmo potuto migliorare la versione precedente con un piccolo controllo in piú). Nelle versioni realmente implementate in librerie comuni di *QuickSort*, ne esiste almeno una che contiene la combinazione di tutte le euristiche viste: tail ricorsione, mediana di tre, e caso base (per array più piccoli i un certo $k$) fatto con un algoritmo diverso, tipicamente *HeapSort*.

# *QuickSort* tail ricorsivo

Quindi vogliamo realizzare un esperimento dove alle implementazioni precedenti aggiungiamo *TailQuickSort* con tutte e tre le euristiche nominate, 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.