Laboratorio di Algoritmi e Strutture Dati

Primo esercizio, terza parte: *QuickSort* con mediana di tre (punti: 3)

Efficienza di *QuickSort* randomizzato

A cosa serve, abbiamo detto, randomizzare *QuickSort*? L'idea è che vogliamo proteggerci da attacchi maligni che producono input volutamente difficili per l'esecuzione di *QuickSort*. Queste situazioni possono sembrare poco comuni, ma non è detto che lo siano: *QuickSort* è universalmente considerato l'algoritmo piú efficiente di ordinamento basato su confronti, ed è ragionevole assumere che sia usato spesso, per esempio, anche in servizi web che hanno bisogno di un passo di ordinamento; dunque una produzione maligna di input validi ma difficili potrebbe anche essere una parte di un attacco.

Efficienza di *QuickSort* randomizzato

In queste condizioni, peró dobbiamo immaginare che un ipotetico attaccante conosca molto bene anche le piú semplici implementazioni di generatori di numeri casuali, e sia in grado di prevederle.

Come abbiamo spiegato nella prima lezione di laboratorio, i numeri casuali sono in realtá pseudo casuali, e conoscendo la tecnica di generazione ed il seed è possibile ricostruirne la sequenza.

Una soluzione sarebbe costruire un generatore molto complesso e/o utilizzare seed realmente casuali per un'implementazione di *RandomizedQuickSort*.

Efficienza di *QuickSort* randomizzato

Purtroppo, la generazione complessa di numeri pseudo casuali incide sull'efficienza, peggiorando le prestazioni della versione randomizzata.

In questo esercizio di laboratorio cerchiamo un compromesso: un'implementazione deterministica che non sia cosí esposta a input maligni come quella originale.

Migliorare *QuickSort* non randomizzato

Sappiamo bene qual è il problema di sceglere un **pivot estremo**: si ottiene una partizione sbilanciata.

La scelta ideale sarebbe scegliere sempre il valore mediano dell'array da ordinare. Questa è una soluzione asintoticamente ottima, perchè grazie ad un algoritmo che risale al 1973, è possibile trovare la mediana di un array in tempo lineare.

Purtoppo questo è solo un risultato asintotico, perchè dal punto di vista pratico rende inefficiente tutto il processo. D'altra parte, possiamo trovare una semplice euristica: ci accontentiamo della mediana di un sotto gruppo di elementi dell'array preso a caso, ma di dimensione costante, ed il processo di trovare la mediana diventa di complessitá costante.

Sulla base della letteratura, troviamo che scegliere come mediana la mediana di **tre elementi** dell'array è sufficiente ad avere partizioni generalmente bilanciate.

Mediana di tre

Utilizziamo una tecnica **hardcoded** per trovare la mediana di tre elementi, indicizzati i, j, k, su un array A.

```
proc MedianOfThree (A, i, j, k)
 if (A[i] > A[j])
   then
  if (A[j] ≥ A[k])
     then return j
     else  if (A[i] < A[k])
     then return i
     else  return k
   else
  if (A[i] ≥ A[k])
     then return i
     else  if (A[j] < A[k])
     then return j
     else  return k
```

QuickSort con mediana di tre

Trovare la mediana di un numero costante di elementi in questo modo è poco elegante dal punto di vista del codice, ma estremamente efficiente.

Ad ogni chiamata ricorsiva di *QuickSort* adesso procediamo così: prendiamo il primo elemento, l'ultimo, e quello centrale, scegliamo il mediano tra questi, e lo utilizziamo come pivot, procedendo come nel caso randomizzato, cioè sostituendolo all'ultimo prima di chiamare la versione classica di *Partition*.

QuickSort con mediana di tre

```
proc MedianOfThreePartition (A, p, r)
{ s = MedianOfThree(A, p, r, (p+r)/2)
  SwapValue(A, s, 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 MedianOfThreeQuickSort (A, p, r)
{ if (p < r)
    then
  { q = MedianOfThreePartition(A, p, r)
    MedianOfThreeQuickSort(A, p, q − 1)
    MedianOfThreeQuickSort(A, q + 1, r) }
```

QuickSort con mediana di tre

Non è difficile vedere che è sempre possibile produrre un input maligno anche per questa versione di *QuickSort*: peró è molto piú complesso farlo, e questa versione è un ottimo compromesso tra tutti i problemi che le diverse versioni di *QuickSort* hanno.

Si possono studiare anche altri aspetti, come per esempio il numero ottimo di elementi di cui trovare la mediana (invece di tre) oltre il quale le prestazioni peggiorano nuovamente.

QuickSort con mediana di tre

Quindi vogliamo realizzare un esperimento dove alle implementazioni precedenti aggiungiamo *MedianOfThreeQuickSort*, 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

Come abbiamo visto la scelta ed il calcolo della mediana è di per sé un problema interessante, che si sposa bene con il nostro obbiettivo di arrivare a versioni efficienti, e realmente usate, di un algoritmo di ordinamento basato su confronti.