QuickSort


3 1 4 5 9 2 8 6
1 2 3 4 5 9 8 6
1 2 3 4 5 9 8 6
1 2 3 4 5 9 8 6
1 2 3 4 5 8 6 9
1 2 3 4 5 6 8 9
1 2 3 4 5 6 8 9

QuickSort

- Una “greatest hit” dell'informatica
- Molto usato in pratica
- Interessante analisi di complessità
- Θ(n log n) nel caso medio
- Algoritmo “in place” (richiede poco spazio extra)

Nota: la descrizione dell'algoritmo quickSort fornita in queste slide è precisa e più che sufficiente alla sua corretta implementazione, ma trascura volutamente alcuni dettagli su come strutturare il codice e quali sono i parametri ed i valori di ritorno delle funzioni: i problemi lasciati aperti vanno analizzati, progettati ed implementati come parte del Laboratorio #2

Idea base di quickSort: partizionare rispetto a un perno (“pivot”)

- Prendi un elemento dell'array (il “pivot”)
- Partiziona gli elementi dell'array in modo tale che
  - quelli a sinistra del pivot siano minori del pivot
  - quelli a destra siano maggiori (o uguali) del pivot
- Dopo queste operazioni, il pivot è nella sua posizione finale
- Richiama quickSort sulle due parti dell'array ottenute

Due fatti importanti della partizione

- Richiede tempo lineare nella dimensione dell'array, $\Theta(n)$.
- Non richiede spazio extra
- Riduce la dimensione del problema

Quicksort, descrizione ad alto livello

[C.A.R. Hoare, 1961]

QuickSort(array A)
{
if lunghezza(A)=1 return;
p=scegliPivot(A);
Partiziona A rispetto al pivot p (metti a sinistra di p gli elementi < e a destra gli elementi >);
Riordina ricorsivamente la prima parte di A (gli elementi minori di p);
Riordina ricorsivamente la seconda parte di A (gli elementi maggiori di p);
}

Questa è una descrizione ad alto livello e trascura alcuni dettagli implementativi la cui comprensione e valutazione è necessaria per realizzare l'algoritmo: è necessario progettare accuratamente come passare da questo livello descrittivo all'implementazione. In particolare bisogna ragionare su come implementare la scelta del pivot, il partizionamento attorno ad esso, e il modo per effetuare le due chiamate ricorsive sulle due parti di A che contengono gli elementi < e > di p

Come partizionare (modo “facile”)

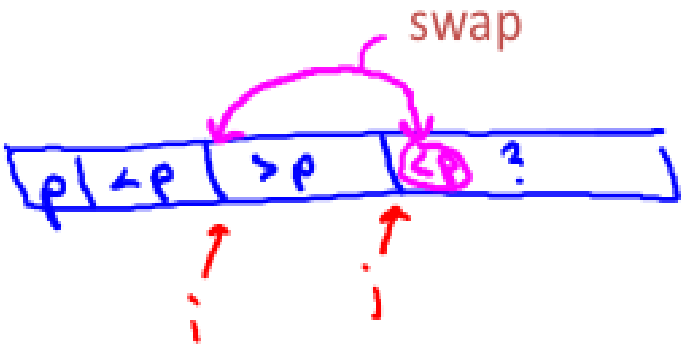
- Se uso un array di appoggio di dimensione n, è facile!
- La complessità temporale è Θ(n) ma spreco spazio per l'array di appoggio

Come partizionare (modo “difficile”, senza array di appoggio)

- Vorremmo riuscire a partizionare “in place” (cioé, senza usare un array di appoggio di dimensione n), mantenendo la complessità temporale Θ(n).
- Assunzione: il primo elemento dell'array è il pivot (se non lo è, faccio uno swap prima della procedura di partizionamento)
- Idea: faccio una sola scansione dell'array; invariante: tutto ciò che ho scandito fino ad ora, è partizionato rispetto al pivot

Come partizionare (modo “difficile”, senza array di appoggio)

```
Partition (A,l,r)                    [ input corresponds to A[l...r]]
        - p:= A[l]
        - i:= l+1
        - for j=l+1 to r
                - if A[j] < p                [if A[j] > p, do nothing ]
                    -swap A[j] and A[i]
                    - i:= i+1
        - swap A[l] and A[i-1]
```



*Questa è una descrizione ad alto livello e trascura alcuni dettagli implementativi che nell'implementazione bisogna assolutamente considerare: in particolare, partition deve restituire l'**indice del pivot p** per poter effettuare le due chiamate ricorsive sugli elementi minori di p e quelli maggiori o uguali*

Come partizionare (modo “difficile”, senza array di appoggio)

- Complessità temporale: Θ(n) perché faccio un numero di operazioni costante per ogni elemento dell'array, e scandisco l'array una sola volta
- Lavora “in place” mediante scambi ripetuti

The importance of being a good Pivot...

- L'efficienza di quickSort dipende da come scegliamo il pivot!

QuickSort con pivot 1° elem e array ordinato

- Supponiamo di scegliere come pivot sempre il primo elemento dell'array. Qual è la complessità di quickSort su un array già ordinato?

QuickSort con pivot 1° elemento e array ordinato

$\Theta(n^2)$

QuickSort con pivot elemento mediano e array qualunque

- Supponiamo di scegliere “per magia” (cioé, a costo costante) come pivot sempre l'elemento mediano dell'array. Qual è la complessità di quickSort su un array qualunque, in questa condizione ottimale e “magica”?

QuickSort con pivot elemento mediano e array qualunque

$\Theta(n \log n)$

Le argomentazioni per questo risultato sono simili a quelle usate per mergesort

Come scegliere un buon pivot?

- IDEA: scelta casuale (random) del pivot

ovvero, ad ogni chiamata ricorsiva, scegliamo come pivot uno dei numeri nell'array, a caso (la probabilità di essere scelto è la stessa per ogni elemento).

Con questo approccio alla scelta del pivot, abbiamo una versione di quickSort “randomizzata”: è il primo esempio di algoritmo randomizzato, nel quale due esecuzioni diverse sullo stesso input possono svolgersi in modo diverso! Naturalmente, il risultato finale delle due esecuzioni deve essere lo stesso!

QuickSort randomizzato

- Scegliendo il pivot a caso tra gli elementi disponibili, si spera di sceglierlo “abbastanza buono” “abbastanza spesso”
- Abbastanza buono: un pivot che partizioni l'array in modo tale che 25% degli elementi sia < del pivot e 75% sia >=, è sufficiente per restare nella complessità $\Theta(n \log n)$ [non lo dimostriamo]
- Abbastanza spesso: metà degli elementi dà una partizione 25%-75% o anche migliore

QuickSort randomizzato

- Per ogni array di dimensione n, il tempo di esecuzione di quickSort randomizzato nel caso medio è Θ(n log n)

  [non lo dimostriamo]

Ottimizzazioni di quickSort

- Un miglioramento del quickSort consiste nel scegliere il pivot come il **valore mediano** di tre elementi scelti a caso.
- Si può ottimizzare ulteriormente chiamando insertionSort invece di quickSort quando gli array diventano "abbastanza piccoli", ma quanto piccoli dipende dall'architettura e dal compilatore usati.

Numeri casuali in C++

- La funzione rand() genera un numero compreso nell'intervallo [0, RAND_MAX], dove RAND_MAX è una costante che dipende dal compilatore usato.
- Il generatore di numeri pseudo casuali produce una sequenza di numeri apparentemente casuali; in realtà la sequenza ad un certo punto si ripete e risulta prevedibile
- Per ovviare a questo inconveniente è necessario far generare la sequenza partendo ad ogni esecuzione con un valore diverso (un seme diverso).
- La funzione srand inizializza il generatore di numeri con un valore che passiamo come argomento
- La funzione time(NULL) restituisce un valore intero ottenuto dal clock interno, pari al numero di secondi trascorsi dal 1/1/1970

Numeri casuali in C++

- In questo modo l’istruzione srand(time(NULL)) inizializza il generatore ad ogni esecuzione con un valore (seme) diverso e le successive istruzioni rand() produrranno valori differenti.
- In genere abbiamo bisogno di chiamare srand una sola volta prima della prima chiamata la funzione rand.
- Per generare valori casuali in un intervallo dato, si usa la funzione “modulo” (%).

QuickSort, verso il codice

```
void quickSort(array A) /* A è un array di interi */
{
qs(A, 0, size(A)-1);   /* chiamo qs, la funzione che definiremo ricorsivamente, su tutto l'array A, ovvero la parte di array identificata dagli estremi 0 e size(A)-1 */
}
```

QuickSort, verso il codice

```
void qs(array A, int inizio, int fine)
{
    if (inizio < fine)
     {
        int pivot_index = partizionaInPlace(A, inizio, fine); /* la funzione partizionaInPlace fa due cose: 1) modifica la parte di A compresa tra inizio e fine, indici inclusi, effettuando la partizione “in place”; per fare questo deve selezionare il pivot_index, mettere l'elemento in posizione pivot_index -- ovvero il pivot -- all'inizio della sottosequenza compresa tra inizio e fine, implementare il partizionamento in place, rimettere a posto il pivot; 2) restituisce l'indice del pivot (non il pivot ma il suo indice!!!); questo è necessario perché dobbiamo richiamare qs sulle due sottoparti di A comprese rispettivamente tra inizio e pivot_index -1, e tra pivot_index+1 e fine */
        qs(A, inizio, pivot_index-1);
        qs(A, pivot_index+1, fine);
     }
}
```