ESERCITAZIONI DI *INTRODUZIONE AGLI ALGORITMI* (A.A. 09/10, CANALE E-O)

APPUNTI ESERCITAZIONE N. 5

1. Quicksort

Introduciamo un altro algoritmo di ordinamento per confronti: il QuickSort. L'algoritmo è ricorsivo e basato come il MergeSort sul paradigma del *Divide et Impera* (dividere un problema in due sottoproblemi più semplici, risolvere i sottoproblemi e comporre le soluzioni).

L'idea è quella di rendere banale la fase di fusione, al costo di rendere non banale la fase di selezione dei sottoproblemi. L'osservazione fondamentale è che l'operazione di fusione di due vettori A_1, A_2 in un vettore ordinato A è banale se i due vettori sono ordinati e se ogni elemento di A_1 è minore uguale di ogni elemento di A_2.

Si procede come segue. Si sceglie un elemento x in A. Si partizionano i restanti elementi di A intorno ad x, ottenendo due sottovettori A_1, A_2 che partizionano i restanti elementi di A, tali che A_1 contiene tutti i restanti elementi $\leq x$, e A_2 contiene tutti i restanti elementi $> x$. Si richiama su A_1 e A_2 la stessa procedura, ricorsivamente (se contengono più di un elemento). Quando le due chiamate ricorsive terminano, si ottiene il risultato desiderando concatendando A_1, x, A_2.

Per semplificare l'analisi, assumiamo che A *non contenga elementi ripetuti*. Possiamo scrivere lo schema dell'algoritmo come segue.

algoritmo *QuickSort*(A)
Scegli un elemento x in A
Partiziona A intorno a x, calcolando
- $A_1 = \{a \in A \ : \ a < x\}$
- $A_2 = \{a \in A \ : \ a > x\}$

if $|A_1| > 1$ **then**
    QuickSort(A_1)
if $|A_2| > 1$ **then**
    QuickSort(A_2)
Concatena A_1, x, A_2 in A.

Fissando una procedura deterministica per scegliere x e una procedura per partizionare A rispetto ad x, lo schema precedente dà una specifica completa dell'algoritmo.

Possiamo analizzare la complessità dell'algoritmo prima di dare i dettagli delle suddette procedure. Ci basta osservare che è possibile partizionare un array A di n

---

Note preparate da Lorenzo Carlucci, carlucci@di.uniroma1.it. La discussione della complessità del QuickSort è essenzialmente tratta da *Introduction to Algorithms*, Cormen, Leiserson, Rivest, Stein.

elementi rispetto a un suo elemento x scelto come perno facendo $n-1$, ossia in tempo lineare in n. Per esempio, si può scorrere A da sinistra a destra confrontando $n-1$ elementi con x e mettendoli negli array A_1 o A_2 a seconda dell'esito del confronto.

Osserviamo che in questo modo l'algoritmo richiede una occupazione di memoria aggiuntiva proporzionale ad n (i due array A_1, A_2). Vedremo più tardi che è possibile partizionare in loco.

Assumendo un tempo costante per il caso base (array di dimensione ≤ 1), possiamo scrivere un'equazione di ricorrenza per il QuickSort come segue.

$$T(n) = \begin{cases} \Theta(1) \text{ se } n \leq 1 \\ n - 1 + T(a) + T(b) \text{ se } n > 1 \end{cases}$$

dove $a = |A_1|$, $b = |A_2|$. In generale vale che $a + b = n - 1$, dato che A_1 e A_2 partizionano gli elementi A meno il perno.

Lo studio dell'equazione di ricorrenza di sopra presenta una difficoltà che non abbiamo incontrato prima. Non è possibile prevedere la dimensione di a e di b ai diversi livelli della ricorsione, per un vettore generico in input. Il QuickSort divide il problema in due sottoproblemi, ma le dimensioni relative di questi sottoproblemi non sono uniformi durante la ricorsione, ma possono cambiare ad ogni livello di ricorsione, in funzione dell'input.

Dunque, in generale, l'albero della ricorsione del QuickSort sarà non soltanto *sbilanciato*, ma anche lo sbilanciamente sarà *non uniforme*. Per questo motivo l'analisi più accurata della complessità del QuickSort usa argomenti probabilistici, e valuta il costo atteso dell'algoritmo quando la scelta del perno viene effettuata in modo *random*.

2. Analisi del QuickSort

Cerchiamo qui di farci un'idea della complessità del QuickSort studiandone la complessità in diversi casi particolari.

In tutti i casi seguenti facciamo un'ipotesi molto forte, che chiamiamo **Ipotesi di Uniformità**. Assumiamo cioè che ad ogni livello della ricorsione le proporzioni dei sottoproblemi A_1, A_2 siano sempre le stesse: per es., assumiamo che QuickSort divida sempre il problema in due problemi di dimensione $1/2$ del problema originario, o in uno di dimensione $1/4$ e in un altro di dimensione $3/4$, etc.

(Caso 1: Input Sbilanciato) L'albero di ricorsione del QuickSort è particolarmente sbilanciato quando il perno x è il minimo o il massimo dell'array A.

- Se $x = \max(A)$ allora $|A_1| = 0$ e $|A_2| = n - 1$.
- Se $x = \min(A)$ allora $|A_2| = n - 1$ e $|A_1| = 0$.

In entrambi questi casi l'equazione di ricorrenza scritta sopra si semplifica notevolmente, e otteniamo la forma seguente.

$$T(n) = \begin{cases} \Theta(1) \text{ se } n \leq 1 \\ n - 1 + T(n-1) \text{ se } n > 1 \end{cases}$$

Sviluppando la ricorrenza otteniamo la progressione aritmetica e dunque una complessità $\Theta(n^2)$.

(Caso 2: Input Perfettamente Bilanciato) L'albero della ricorsione del QuickSort è ottimamente bilanciato quando il perno x è il mediano dell'array A. In questo caso il problema è ridotto a due sottoproblemi di dimensione inferiore a $n/2$. Dato che il tempo di esecuzione di QuickSort è monotono (andrebbe dimostrato!), possiamo porre in questo caso

$$T(n) \leq n - 1 + 2T(n/2).$$

Risolvendo la ricorrenza otteniamo una limitazione superiore $O(n \log(n))$.

(Caso 3: Input Sbilanciato ma Uniforme) Supponiamo che la scelta del perno sia tale che ad ogni passo della ricorsione il problema venga ridotto in un sottoproblema di dimensione $n \cdot r$ e in uno di dimensione $n(1-r)$, dove r è un razionale positivo minore di 1. Scriviamo $r = p/q$, con $p < q$. L'equazione di ricorrenza diventa allora

$$T(n) = \begin{cases} \Theta(1) \text{ se } n \leq 1 \\ n - 1 + T(\frac{p \cdot n}{q}) + T\left(\frac{(q-p) \cdot n}{q}\right) \text{ se } n > 1 \end{cases}$$

L'argomento è analogo a quanto visto nel caso del MergeSort sbilanciato. Fissiamo, per semplicità, $r = 2/9$, $1 - r = 7/9$. Disegnando l'albero della ricorsione osserviamo quanto segue.

- Il contributo complessivo di un livello completo è uguale a $c \cdot n$
- Lungo i rami più corti la ricorrenza termina dopo $\log_{9/2}(n)$ passi.
- Lungo i rami più lunghi la ricorrenza termina dopo $\log_{9/7}(n)$ passi.

Dal primo punto segue anche che il contributo complessivo di un livello incompleto è comunque $\leq c \cdot n$. Dato che l'altezza massima di un ramo è $\log_{9/7}(n)$, possiamo tranquillamente concludere la seguente limitazione superiore.

$$T(n) = O(n \cdot \log_{9/7} n).$$

(Caso 4: Caso Misto) Per farci un'idea del caso generico, in cui partizioni di diverse proporzioni si distribuiscono imprevedibilmente sui nodi dell'albero della ricorrenza, consideriamo la seguente approssimazione. Supponiamo che la suddivisione in sottoproblemi si alterni tra il caso più sbilanciato $(0,\ n-1)$ e il caso più bilanciato $(n/2,\ n/2)$. Cosa possiamo dire in questo caso della complessità di QuickSort?

Il primo livello dell'albero avrà un nodo di costo $\Theta(n)$ (la partizione del vettore iniziale), un figlio di costo 0 (il vettore vuoto), e un figlio con chiamata ricorsiva su un sottovettore di dimensione $n-1$. Questa chiamata contribuirà un costo di $\Theta(n-1)$ (costo della partizione sul vettore di dimensione $n-1$) e due chiamate ricorsive, entrambe su vettori di dimensione circa $(n-1)/2$. Il costo combinato dei due livelli è dunque $\Theta(n) + \Theta(n-1)$, i.e., $\Theta(n)$, lo stesso costo di un livello di ricorsione nel caso perfettamente bilanciato. A questo punto restano da risolvere due sottoproblemi di dimensione $(n-1)/2$. La stessa situazione si verifica al primo livello dell'albero di ricorsione nel caso perfettamente bilanciato: un costo $\Theta(n)$ per la partizione iniziale, e due sottoproblemi di dimensione $(n-1)/2$ da risolvere. Possiamo così concludere che il costo del QuickSort in questo caso sarà ancora dell'ordine di $O(n \log(n))$ (da notare che l'altezza dell'albero di ricorsione nel caso alternante differisce dall'altezza dell'albero di ricorsione nel caso bilanciato per una costante).

In conclusione, qui sopra abbiamo analizzato il costo del QuickSort in tre casi sotto ipotesi di uniformità (sbilanciato $\Theta(n^2)$, perfettamente bilanciato $O(n\log(n))$ e sbilanciato $O(n\log(n))$), e in un caso misto non uniforme ($O(n\log(n))$). *Non* abbiamo però *dimostrato rigorosamente* che $\Theta(n^2)$ è la complessità nel caso peggiore, né che $O(n\log(n))$ è la complessità nel caso medio. Abbiamo soltanto analizzato la complessità dell'algoritmo su alcune istanze che sono intuitivamente significative e che intuitivamente approssimano il caso migliore, peggiore e medio.

3. Procedure di Partizione in loco

Torniamo ora all'algoritmo di partizione e mostriamo come si possa fare *in loco*. Diamo un esempio diverso da quello del libro di testo. L'algoritmo partiziona la regione dell'array A contenuta tra l'indice p e l'indice r attorno al perno $A[r]$. L'algoritmo ritorna anche l'indice della posizione del perno alla fine della partizione. Questo valore viene usato per determinare su quali sottovettori effettuare le chiamate ricorsive.

```
Partiziona(A, p, r)
x ← A[r]
i ← p − 1
for j = p to r − 1 do
    if A[j] ≤ x then
        i ← i + 1
        scambia A[i], A[j]
scambia A[i + 1], A[r]
return i + 1
```

Esercizio. Tracciare le operazioni dell'algoritmo sull'array $A = 3, 2, 8, 7, 1, 6, 4$, $p = 1$, $r = 7$

Esercizio. Dimostrare che il ciclo For nell'algoritmo Partiziona soddisfa il seguente invariante e concludere che l'algoritmo è corretto.

(1) Se k è tra p e i allora $A[k] \leq x$.
(2) Se k è tra $i + 1$ e $j - 1$, allora $A[k] > x$.
(3) Se $k = r$ allora $A[k] = x$.