Tutorato 8 Novembre

1 Esempio di induzione

Esercizio Data la seguente equazione di ricorrenza, trovare un limite superiore alla complessità di $T(n)$.

$$T(n) = \begin{cases} 4T\left(\frac{n}{2}\right) + \Theta(n) & n > 1 \\ 1 & n \leq 1 \end{cases}$$

Soluzione Dimostriamo per induzione $T(n) = O(n^2)$. Vogliamo mostrare $\exists c > 0$ tale che $T(n) \leq cn^2 - dn$ per ogni $n \geq 1$ e per qualche $d \in \mathbb{R}$.

CASO BASE. Per $n = 1$, $T(n) = 1 \leq c - d$. Posso scegliere $c > d$ e l'ipotesi è verificata per $n = 1$.

PASSO INDUTTIVO. Adesso supponiamo che $\exists c > 0$ tale che $\forall\, m < n$ $T(m) \leq cm^2 - dm$ per qualche $d < c$. Dobbiamo mostrare che questo implica $T(n) \leq cn^2 - dn$.

$$\begin{aligned} T(n) &= 4T\left(\frac{n}{2}\right) + \Theta(n) && \text{(I.I. su } \tfrac{n}{2}\text{)} \\ &\leq 4c\frac{n^2}{4} - 4d\frac{n}{2} + \Theta(n) \\ &\leq cn^2 - 2dn + \Theta(n) \\ &\leq cn^2 - dn - (dn - \Theta(n)) \\ &\leq cn^2 - dn \end{aligned}$$

Perchè l'ultima disuguaglianza sia vera devo scegliere d sufficiente grande da compensare la costante nascosta in $\Theta(n)$, ma questo posso sempre farlo (a patto di scegliere anche c sufficientemente largo).

Soluzione sbagliata Se provo a dimostrare che $\exists c > 0$ tale che $T(n) \leq cn^2$ per ogni $n \geq 1$, la dimostrazione non funziona.

CASO BASE. Per $n = 1$, $T(n) = 1 \leq c$. Posso scegliere $c > 1$ e l'ipotesi è verificata per $n = 1$.

PASSO INDUTTIVO. Supponiamo che $\exists c > 1$ tale che $\forall\, m < n$ $T(m) \leq$

cm^2. Dobbiamo mostrare che questo implica $T(n) \leq cn^2$ per ogni $n > 1$.

$$\begin{aligned} T(n) &= 4T\left(\frac{n}{2}\right) + \Theta(n) && \text{(I.I. su } \tfrac{n}{2}\text{)} \\ &\leq 4c\frac{n^2}{4} + \Theta(n) \\ &\leq cn^2 + \Theta(n). \end{aligned}$$

Per concludere dovrei far vedere $cn^2 + \Theta(n) \leq cn^2$ per qualche $c > 1$, ma questo non è mai vero. Inoltre non basta dire che $cn^2 + n$ sta in $\Theta(n^2)$, perchè affinchè l'induzione sia corretta la tesi da provare è $T(n) \leq cn^2$ (o, in altre parole, è come se stessi provando $T(n) \leq c'n^2$ cambiando costante rispetto all'ipotesi induttiva).

Se avete altri dubbi sull'induzione guardate sul Cormen pp.90-94.

2 Esercizi

Esercizio 1 - Heap d-arie A lezione avete visto come implementare Heapsort utilizzando una heap binaria, cioè un albero binario semicompleto. In generale, si possono considerare *heap d-arie*, cioè alberi in cui ogni nodo non-foglia (eccetto al più uno) ha d figli.

a) Come si può rappresentare un heap d-aria sotto forma di array?

b) Qual è l'altezza di un heap d-aria di n elementi?

c) Scrivete un algoritmo efficiente che estrae il massimo da un heap d-aria e stimatene la complessità.

Esercizio 2 - Stooge-Sort
Dato il seguente algoritmo di ordinamento:

Algorithm 1 STOOGE-SORT(A, p, r)

```
if A[p] > A[r] then
  exchange A[p] and A[r]
end if
if p + 1 < r then
  k = ⌊(r − p + 1)/3⌋
  STOOGE-SORT(A, p, r − k)
  STOOGE-SORT(A, p + k, r)
  STOOGE-SORT(A, p, r − k)
end if
```

a) Dimostrarne la correttezza.

b) Scrivere un'equazione di ricorrenza per la complessità e risolverla con un metodo a scelta tra quelli visti a lezione.

c) Confrontare il risultato con le complessità degli altri algoritmi visti a lezione. Si tratta di un algoritmo efficiente?

Esercizio 3 - Quicksort meets Hoare Originariamente Quicksort utilizzava il seguente algoritmo di partizione, introdotto da C.A.R. Hoare.

Algorithm 2 HOARE-PARTITION(A, p, r)

```
1:  x ← A[p]
2:  i ← p − 1
3:  j ← r + 1
4:  while true do
5:    repeat
6:      j ← j − 1
7:    until A[j] ≤ x
8:    repeat
9:      i ← i + 1
10:   until A[i] ≥ x
11:   if i < j then
12:     exchange A[i] with A[j]
13:   else
14:     return j
15:   end if
16: end while
```

a) Spiegate come agisce HOARE-PARTITION su

$$A = (13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21).$$

b) Dimostrate la correttezza di HOARE-PARTITION. Suggerimento: argomentare il perchè le seguenti tre affermazioni sono vere:

- Gli indici i e j non accedono mai a valori al di fuori di $A[p : r]$.
- Quando HOARE-PARTITION termina, il valore j in output è $p \leq j < r$.
- Quando HOARE-PARTITION termina ogni elemento di $A[p : j]$ è inferiore o uguale a ogni elemento di $A[j + 1 : r]$.

c) Riscrivete QUICKSORT utilizzando HOARE-PARTITION.

3 Soluzioni

Es. 1.

a) Ricordiamo che per una heap binaria indicizzata a partire da 0 si aveva:

$$\textit{parent(i)} = \lfloor \frac{i-1}{2} \rfloor \qquad \textit{left\_child(i)} = 2i+1 \qquad \textit{right\_child(i)} = 2i+2.$$

Nel caso di una heap d-aria ogni nodo ha d figli dunque:

$$\textit{parent(i)} = \lfloor \frac{i-1}{d} \rfloor \qquad \textit{child(i,j)} = d \cdot i + j$$

dove *child(i,j)* rappresenta il figlio j-simo del nodo i.

b) Denotando con h l'altezza, per una heap binaria si aveva $2^h \leq n \leq 2^{h+1}$ dunque $h = \Theta(\log_2(n))$. Nel caso di una heap d-aria

$$\sum_{i=0}^{h-1} d^i + 1 \leq n \leq \sum_{i=0}^{h} d^i$$

(per vederlo, considerate il fatto che $\sum_{i=0}^{h} d^i$ è il numero di nodi di un albero d-ario completo). Sfruttando il fatto che per $d \geq 2$ si ha $d^h \leq \sum_{i=0}^{h} d^i \leq d^{h-1}$, dalla disuguaglianza di prima segue

$$d^{h-1} \leq n \leq d^{h+1}$$

e dunque $h = \Theta(\log_d(n))$

c) Per il caso binario avevamo:

Algorithm 3 MAX-HEAP-EXTRACT-MAX(A)

```
1: if A.heap-size < 1 then
2:    error "heap underflow"
3: end if
4: max ← A[1]
5: A[1] ← A[A.heap-size]
6: A.heap-size ← A.heap-size − 1
7: call MAX-HEAPIFY(A, 1)
8: return max
```

dove la function MAX-HEAPIFY$(A, 1)$ ristabilisce la proprietà di max-heap, violata per il nodo 1, scambiando ricorsivamente il nodo che viola la proprietà con il maggiore tra i suoi nodi figli. Nel caso d-ario MAX-HEAP-EXTRACT-MAX rimane uguale, ma la funzione MAX-HEAPIFY adesso deve trovare il massimo tra d figli, invece che tra due.

Poichè la funzione MAX-HEAPIFY viene chiamata al più $h = \Theta(\log_d(n))$ volte e ogni volta ha un costo $O(d)$ il costo totale di MAX-HEAP-EXTRACT-MAX è $O(d \log_d(n))$

Es. 2

Algorithm 4 MAX-HEAPIFY(A, i)

```
1: largest ← i
2: for k = 1 to d do
3:    if d-ARY-CHILD(i, k) ≤ A.heap-size and A[d-ARY-CHILD(i, k)] > A[largest] then
4:       largest ← d-ARY-CHILD(i, k)
5:    end if
6: end for
7: if largest ≠ i then
8:    exchange A[i] with A[largest]
9:    call d-ARY-MAX-HEAPIFY(A, largest)
10: end if
```

a) Per induzione sulla dimensione n di A.

Se $n = 2$ il primo if controlla se A è già ordinato ed eventualmente scambia gli elementi. Il secondo if non viene eseguito, e l'array è correttamente ordinato.

Supponiamo STOOGE-SORT sia corretto per $k < n$. Se A ha dimensione $n > 2$, il primo if eventualmente scambia il primo e l'ultimo elemento dell'array, Poi viene eseguito il secondo if su array di dimensione $< n$. Senza perdita di generalità supponiamo $n = 3m$. Per ipotesi induttiva, la prima chiamata ordina correttamente $A[0 : 2m]$, dunque in particolare $A[0 : m - 1] \leq A[m : 2m]$ La seconda chiamata ordina correttamente gli elementi di $A[m, 3m]$, dunque $A[m : 2m - 1] \leq A[2m - 1 : 2m]$. A questo punto gli elementi in $A[2m - 1 : 2m]$ sono correttamente ordinati e sicuramente gli elementi maggiori del vettore. Dunque l'ultima chiamata, che riordina $A[0 : 2m]$ finisce di ordinare correttamente A.

b)

$$T(n) = 3T\left(\frac{2}{3}n\right) + \Theta(1)$$

Con il metodo dell'esperto otteniamo $a = 3$, $b = \frac{3}{3}$ e $\log_b(a) = \log_{\frac{3}{2}}(3) \approx 2.7$. Dunque ci troviamo nel primo caso: $T(n) = \Theta\left(n^{\log_{\frac{3}{2}} 3}\right)$.

c) Poichè questo costo è maggiore di $\Theta(n^2)$, STOOGE-SORT performa peggio di tutti gli algoritmi visti a lezione!

Es. 3

a) Indicizzando A a partire da 0, prima di entrare nel while loop si ha $x = 13$, $i = -1$, $j = 12$. Nella prima iterazione, i due loop interni si fermano a $j = 10$ e $i = 0$ per cui dopo lo scambio avremo $A = (6, 19, 9, 5, 12, 8, 7, 4, 11, 2, 13, 21)$. Nella seconda iterazione, i loop interni si fermano a $j = 9$ e $i = 1$, per cui dopo lo scambio avremo $A =$

$(6, 2, 9, 5, 12, 8, 7, 4, 11, 19, 13, 21)$. Nella terza iterazione i loop interni si fermano a $j = 8$ e $i = 9$ dunque non avviene nessuno scambio e viene restituito in output $j = 8$.

b)
- Iniziamo osservando che all'inizio di ogni iterazione (subito dopo **while** true **do**), si ha $i < j$. Questo è vero la prima volta che si entra nel while esterno, ma è vero anche dopo, altrimenti saremmo usciti dal loop nell'iterazione prima. Inoltre i è inizializzato a p e nel programma viene solo incrementato, dunque $p \leq i$, mentre j viene inizializzato a r e viene solo decrementato dunque $j \leq r$. Dunque all'inizio di ogni iterazione si ha $p \leq i < j \leq r$. Ora dobbiamo mostrare che i loop interni, non possono mai portare a $j < p$ o $i > r$. Per farlo, mostriamo che esistono k e k' tali che:
  * $i < k \leq j$ e $A[k] \geq x$ (cioè il loop che incrementa i si ferma al più quando raggiunge il valore che j aveva all'inizio del while)
  * $i \leq k' < j$ e $A[k'] \leq x$ (cioè il loop che decrementa j si ferma al più quando raggiunge il valore che i aveva all'inizio del while)

  Notiamo che all'inizio della prima iterazione questo è vero, perchè $i = p - 1$ e $j = r + 1$ e $h = p$ dunque posso prendere $k = k' = h$. Dalla seconda iterazione in poi, sappiamo che all'inizio $i < j$ dunque posso prendere $k = j$ e $k' = i$. Poichè nell'iterazione precedente avevamo scambiato $A[i] \geq x$ con $A[j] \leq x$, adesso ho $A[k] = A[j] \geq x$ (dopo lo scambio) e $A[k'] = A[i] \leq x$ (dopo lo scambio), che è quello che volevamo. Dunque $p \leq i, j \leq r$ durante tutta l'esecuzione.
- Dall'argomento precedente sappiamo $p \leq j \leq r$, ma poichè j viene decrementato almeno una volta a ogni iterazione e il while viene eseguito almeno una volta $j < r$.
- Mostriamo per induzione sul numero di iterazioni del while, che alla fine di ogni iterazione tutti gli elementi di $A[p : i]$ sono minori o uguali a x e tutti gli elementi di $A[j : r]$ sono maggiori o uguali a x. Questo è vero all'inizio (0 iterazioni) perchè i due array sono vuoti. Ora supponiamo che alla fine di un'iterazione si abbia $A[p : i] \leq x$ e $A[j : r] \geq x$. Il loop che decrementa j prosegue finchè non arrivo a j' tale che $A[j'] \leq x$. Dunque $A[j' + 1 : r] \geq x$ (altrimenti il loop si sarebbe fermato prima). Allo stesso modo, il loop che incrementa i prosegue finchè i' non è tale che $A[i'] \geq x$, dunque $A[p : i' - 1] \leq x$ (altrimenti il loop si sarebbe fermato prima). Se ora $i' < j'$ scambio gli elementi. Dopo lo scambio $A[p : i'] \leq x$ e $A[j' : r] \geq x$ e abbiamo provato la tesi. Se invece $i' \geq j'$ l'algoritmo termina restituendo j' per cui si ha $A[p : j'] \leq x$. Infatti, sappiamo che $A[p : i' - 1] \leq x$. Inoltre $j' \leq i'$ implica che al massimo $A[p : j']$ ha un elemento in più di $A[p : i' - 1]$ che è $A[j'] \leq x$. Infine, avevamo già mostrato che $A[j' + 1 : r] \geq x$, dunque anche in caso di terminazione la tesi vale.

c) QUICKSORT funziona esattamente nello stesso modo, ma chiama HOARE-PARTITION come algoritmo di partizione

Algorithm 5 HOARE-QUICKSORT(A, p, r)

1: **if** $p < r$ **then**
2: $q \leftarrow$ HOARE-PARTITION(A, p, r)
3: **call** HOARE-QUICKSORT(A, p, q)
4: **call** HOARE-QUICKSORT$(A, q + 1, r)$
5: **end if**