ASD Homework 2

Francesco Iannaccone e Matteo Conti

October 2021

1 Homework Esercitativi

1.1 Ricorrenze

- $T(n) = T(\sqrt{n}) + \Theta(lg(lg(n)))$

  Procedo inizialmente ad una sostituzione di variabili: $n = 2^m$, la ricorrenza diventa quindi: $T(2^m) = T(2^{m/2}) + \Theta(\lg(m))$, a questo punto, considerando $T(2^m) = S(m)$ si ottiene: $S(m) = S(\frac{m}{2}) + \Theta(\lg(m))$.

  Per risolvere questa ricorrenza non è possibile utilizzare il terzo caso del teorema dell'esperto, infatti $\lg(n)$ non è polinomialmente più grande di n^0.

  Utilizzando il metodo dell'albero di ricorrenza si ottiene la seguente espressione: $S(m) = \sum_{i=0}^{\lg(m)-1}(\lg(\frac{m}{2^i})) + \lg(\frac{m}{2^(\lg(m))})T(1) = \Theta(\lg^2(m))$, procedendo ad invertire la trasformazione utilizzata: $m = \lg(n) \Rightarrow T(n) = \Theta(\lg^2(\lg(n)))$.

- $T(n) = 10T(\frac{n}{3}) + 17n^{1.2}$

  $17n^{1.2-\epsilon} = O(n^{\log_3(10)-\epsilon}) \approx O(n^{2.0959-\epsilon})$. Per il primo caso del teorema dell'esperto possiamo affermare che $T(n) = \Theta(n^{\log_3(10)})$.

1.2 Ordinamento

Per ordinare in tempo lineare n numeri compresi tra 0 e n^3-1 possiamo utilizzare il radix sort procedendo ad ordinare ogni gruppo di cifre binarie con il counting sort.

In particolar modo, raggruppando le cifre in gruppi di dimensione $r = lg(n)$ e poiché la dimensione minima in bit su cui possono essere rappresentati i numeri da 0 a $n^3 - 1$ è proprio pari a $lg(n^3 - 1 - 0 + 1) = 3lg(n)$, otteniamo che in queste condizioni il radix sort esegue come:

$$T(n) = \Theta\left(\frac{b}{r}(n + 2^r)\right) = \Theta\left(\frac{3lg(n)}{lg(n)}(n + 2^{lg(n)})\right) = \Theta(6n) = \Theta(n)$$

.

2 Homework di verifica

2.1 Costruire un heap mediante inserimento

- Nonostante entrambe le procedure organzzino l'heap producendo un max-heap, non è detto che, per il modo in cui sono realizzate, producano sempre lo stesso max-heap a partire da uno stesso array di input. Dimostriamolo con un controesempio, considerando il vettore $[1, 2, 3]$: BUILD-MAX-HEAP scambia l'1 con il 3 producendo il max-heap $[3, 1, 2]$. A partire dallo stesso vettore invece BUILD-MAX-HEAP_v2 inserisce prima il 2, producendo $[2, 1, 3]$ e poi inserisce il 3 ottenendo in definitiva il max-heap $[3, 2, 1]$.

- Nel caso peggiore (ovvero quando si inserisce un valore nell'heap più grande della radice v[0]), MAX-HEAP-INSERT avrà complessità $\Theta(\lg(n))$ poiché bisognerà effettuare tanti swap quanti sono i livelli sopra l'elemento considerato, ovvero $h = \lg(n)$.

  BUILD-MAX-HEAP_v2 richiede che la procedura MAX-HEAP-INSERT sia invocata per $n - 1$ volte su un heap la cui dimensione aumenta sempre di 1. Si avrà quindi:

  $$T(n) = \sum_{i=2}^{n} \Theta(\lg(i)) = \Theta(\lg(n)+\lg(n-1)+\ldots+\lg(2)) = \Theta(\lg(n!)) = \Theta(n\lg(n))$$

  dove, per l'ultimo passaggio, è stata utilizzata la nota approssimazione di *Stirling*.

2.2 Sorting almost sorted list

- L'array di n elementi col maggior numero di inversioni è quello ordinato in ordine decrescente: $(n, n-1, \ldots, 1)$, che presenta $\sum_{i=0}^{n-1} n - i - 1 = \frac{n(n-1)}{2}$ inversioni.

- Per ogni elemento i del vettore V l'insertion sort effettua tanti swap quanti sono gli elementi j precedenti a i per i quali risulta che $V[j] > V[i]$ e $i > j$. Il numero totale degli scambi è dunque pari ad L, ovvero il numero totale di inversioni dell'array. La complessità dell'insertion sort cresce con lo stesso andamento del numero di inversioni L. Nel caso dell'array quasi ordinato, se ogni elemento è al più a k slot dalla sua posizione corretta esisteranno al più k elementi con i quali risulta invertito, di conseguenza avremo al più k inversioni per ognuno degli n elementi. Il numero totale di inversioni L $= \Theta(nk)$ e lo è, per quanto detto, anche la complessità dell'insertion sort.

`INSERTION-SORT(A){`	
`    for j=2 to n`	$\sum_{i=1}^{n} c_1$
`        key <- A[j]`	$\sum_{i=1}^{n-1} c_2$
`        i <- j-1`	$\sum_{i=1}^{n-1} c_3$
`        while i>0 and A[i]>key`	$\sum_{i=1}^{n-1} \sum_{j=1}^{k+1} c_4$
`            A[i+1] <- A[i]`	$\sum_{i=1}^{n-1} \sum_{j=1}^{k} c_5$
`            i <- i-1`	$\sum_{i=1}^{n-1} \sum_{j=1}^{k} c_6$
`        A[i+1] <- key`	$\sum_{i=1}^{n-1} c_7$
`}`	

  Considerando l' ipotesi di quasi ordinamento di A, per quanto descritto sopra, il ciclo while all'interno del for della procedura INSERTION-SORT viene eseguito nel peggior caso possibile $\sum_{i=1}^{n-1} \sum_{j=1}^{k+1} c_4 = \sum_{i=1}^{n-1} c_4(k+1) = c_4(n-1)(k+1) = \Theta(nk)$.

-
```
SORT_ALMOST_SORTED(v, k){
    n = len(v)

    v[1,...,k]
    for i <- 1 to k+1
        a[i] <- v[i]
```

```
    buid_min_heap(a, k+1)

    for i <- 1 to n - (k+1)
        v[i] <- a[1]
        a[1] <- v[i+(k+1)]
        min_heapify(a,(k+1), 1)

    for i <- 1 to k+1
        v[i+k] <- a[1]
        a[1] <- a[k+2-i]
        min_heapify(a, (k+1) - i, 1)
}
```

L'idea su cui si basa l'algoritmo è quella di costruire un min-heap di k+1 elementi, estrarre il minimo tra questi (che corrisponde alla radice del min-heap) e procedere aggiungendo l'elemento successivo dell'array v come radice operando, successivamente, Min-Heapify sul nuovo elemento e ricostruendo il min-heap. L'algoritmo può essere applicato poiché siamo certi, per le proprietà di quasi ordinamento del vettore, che nei primi k+1 elementi rimasti troveremo il minimo del vettore. Si procede in questo modo all'ordinamento dei primi n - (k+1) elementi. Per ordinare i successivi k+1 elementi si può procedere con un approccio analogo, utilizzando però un min-heap di dimensione ridotta ad ogni iterazione.

La complessità computazionale può essere stimata come segue: il metodo di copia ha complessità k+1, successivamente viene eseguito per n-(k+1) volte il metodo Min-Heapify, di complessità $\lg(k+1)$, sul sotto-array di k+1 elementi, ed infine abbiamo un algoritmo di ordinamento dei rimanenti k+1 elementi con complessità $(k+1)\lg(k+1)$. Risulta dunque:

$$T(n) = \Theta(k+1)+(n-(k+1))\Theta(\lg(k+1))+\Theta((k+1)\lg(k+1)) = \Theta(n\lg(k))$$