Esercizi Ordinamenti

Problema 2.1 [CLRS10]

Insertion sort nel merge sort su array piccoli.
Benché merge sort venga eseguito nel caso peggiore in tempo $\Theta(n\lg n)$ ed insertion sort impieghi un tempo $\Theta(n^2)$, sempre considerando il caso peggiore, i fattori costanti dell'insertion sort lo rendono più veloce per valori piccoli di n.
Quindi ha senso usare insertion sort dentro il merge sort quando i sottoproblemi diventano sufficientemente piccoli. Si consideri una modifica del merge sort in cui n/k sottoliste di lunghezza k sono ordinate usando l'insertion sort e sono poi combinate usando il meccanismo standard di fusione, con k che deve essere determinato.

a. Mostrare che le n/k sottoliste, ciascuna di lunghezza k nel caso peggiore possono essere ordinate dall' insertion sort con un tempo $\Theta(nk)$.

Poiché l'insertion sort ha un tempo di esecuzione nel caso peggiore in $\Theta(k^2)$ per ordinare una lista di k elementi, e visto che abbiamo n/k liste da ordinare in totale avremo $n/k * \Theta(k^2)$, quindi $\Theta(nk)$ nel caso peggiore.

Problema 2.1 [CLRS10]

Insertion sort nel merge sort su array piccoli
Benché merge sort venga eseguito nel caso peggiore in tempo Θ(nlgn) ed insertion sort impieghi un tempo $\Theta(n^2)$, sempre considerando il caso peggiore, i fattori costanti dell'insertion sort lo rendono più veloce per valori piccoli di n.
Quindi ha senso usare insertion sort dentro il merge sort quando i sottoproblemi diventano sufficientemente piccoli. Si consideri una modifica del merge sort in cui n/k sottoliste di lunghezza k sono ordinate usando l'insertion sort e sono poi combinate usando il meccanismo standard di fusione, con k che deve essere determinato.

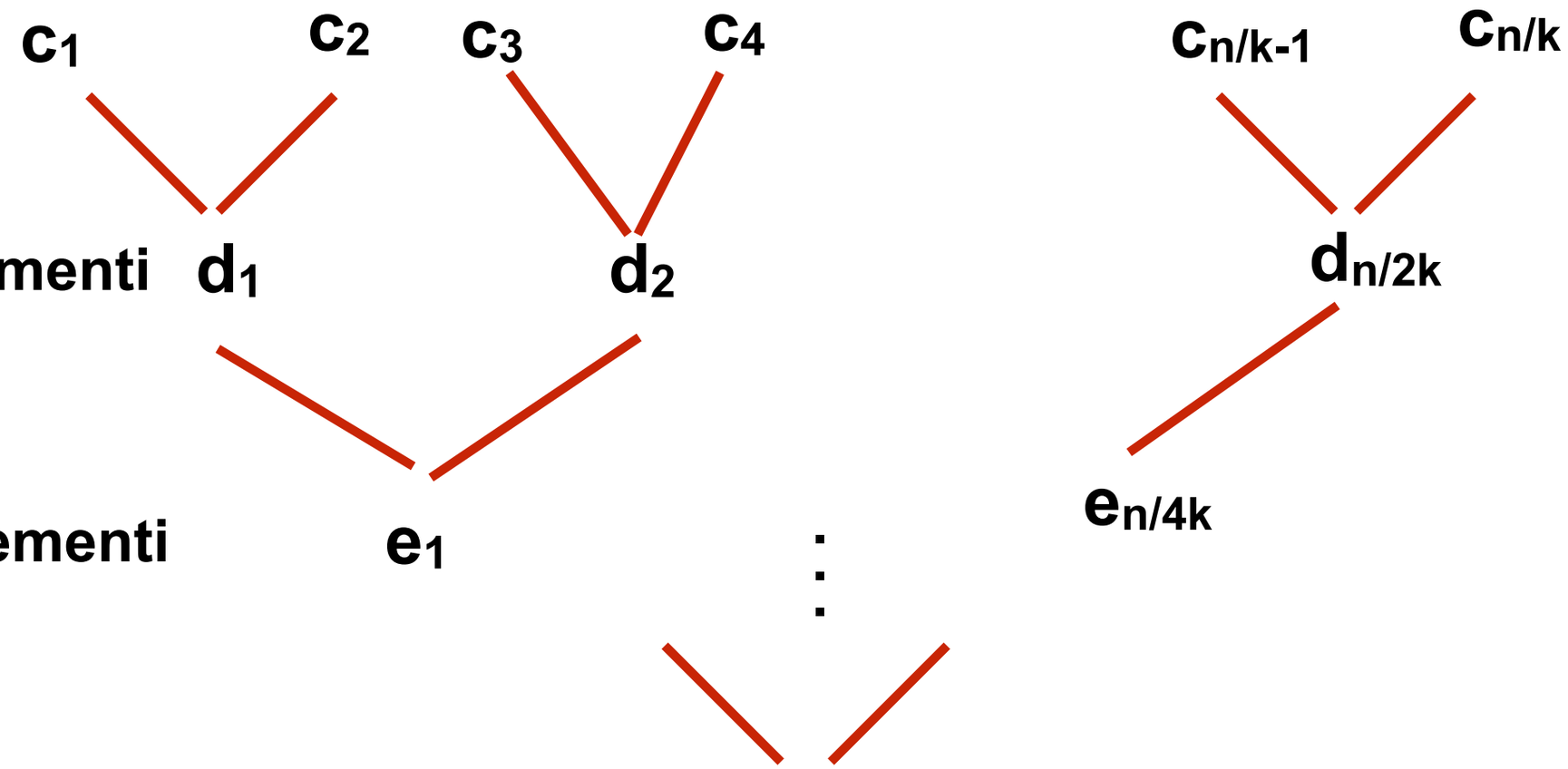
b. Mostrare che le sottoliste possono essere fuse in tempo Θ(nlg(n/k)), sempre considerando il caso peggiore.

Il modo più naturale di estendere la fusione di più array ordinati in un unico array ordinato sarebbe quello di calcolare il minimo tra tutti i primi elementi degli array ordinati e copiarlo in un nuovo array, ripetendo poi questo passo tra i primi elementi in ogni array da fondere e non ancora copiati nel nuovo array. Però calcolare il minimo tra n/k elementi ha un tempo di esecuzione Θ(n/k) e poiché andrebbe ripetuto n volte, si ottiene un tempo $\Theta(n^2/k)$.

La merge su n/k arrays

Sia $n/k = 2^j$

c_i è un array di k elementi



Si fondono gli array a due a due, poi i risultanti ancora a due a due, fino ad arrivare ad ottenere gli n elementi.
Su ogni livello complessivamente si hanno n elementi coinvolti nella fusione, e l'altezza dell'albero è pari all'esponente h di 2 tale che $2^h k = n$, da cui $2^h = n/k$ e quindi $h = j = \lg n/k$.
In conclusione per completare la fusione si impiega un tempo $\Theta(n \lg(n/k))$

Problema 2.1 [CLRS10]

Insertion sort nel merge sort su array piccoli
Benché merge sort venga eseguito nel caso peggiore in tempo Θ(nlgn) ed insertion sort impieghi un tempo $\Theta(n^2)$, sempre considerando il caso peggiore, i fattori costanti dell'insertion sort lo rendono più veloce per valori piccoli di n. Quindi ha senso usare insertion sort dentro il merge sort quando i sottoproblemi diventano sufficientemente piccoli. Si consideri una modifica del merge sort in cui n/k sottoliste di lunghezza k sono ordinate usando l'insertion sort e sono poi combinate usando il meccanismo standard di fusione, con k che deve essere determinato.

c. Dato che l'algoritmo modificato viene eseguito, nel caso peggiore, in un tempo Θ(nk + nlg(n/k)), qual è il più grande valore asintotico (notazione Θ) di k come funzione di n, per cui l'algoritmo modificato ha lo stesso tempo di esecuzione asintotico del merge sort standard?

d. Come, nella pratica, dovrebbe essere scelto il valore di k?

Deve essere k = Θ(lg n), infatti in tal caso Θ(nk + nlg(n/k)) = Θ(nlg n)

In pratica si dovrebbe scegliere k sperimentalmente trovando il valore di n per cui l'insertion sort è più veloce del mergesort.
Questo dipende dalle costanti coinvolte, per esempio se $n \leq 58$ allora risulta $n^2 \leq 10\ n \lg n$.

Esercizio 1

Modificare il MergeSort in modo da eseguire l'InsertionSort su array di dimensione al più k. Ovvero la fase di divisione in due parti della sequenza iniziale termina quando il numero di elementi è minore o uguale a k anziché 1.

L’algoritmo

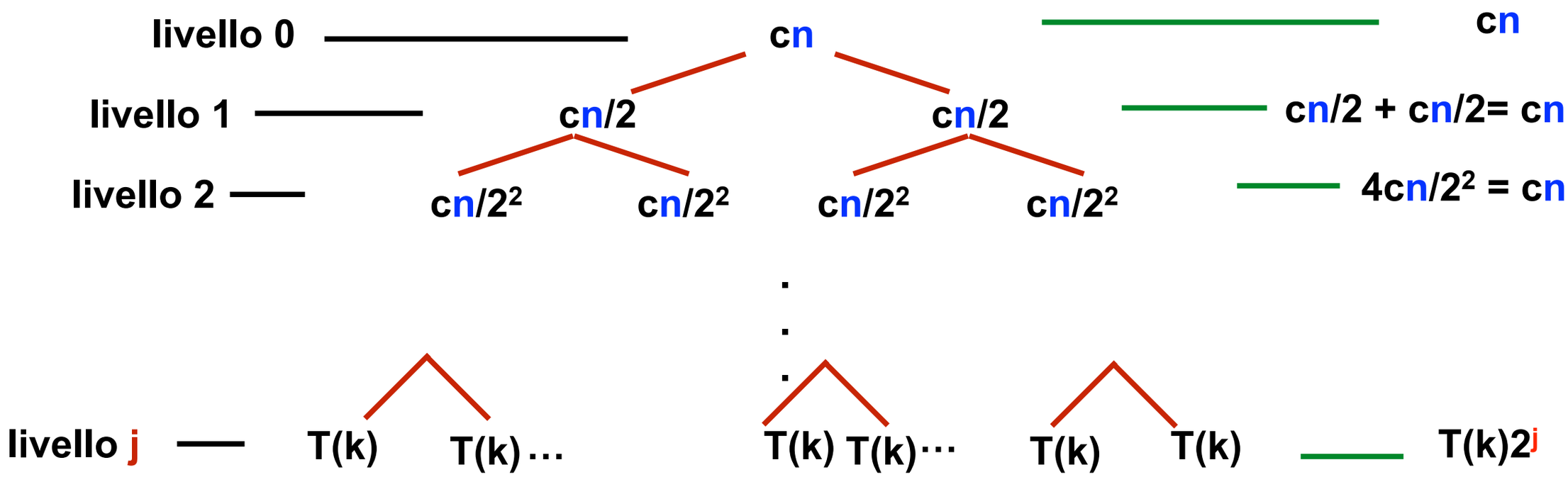
L’algoritmo da analizzare è quindi:

```
MergeInsSort(A,i,f,k)
input: A è un array, i,f,k sono interi positivi, i primi due determinano la
porzione di A su cui l’algoritmo è chiamato e k è il valore di soglia al di
sotto del quale l’array è ordinato con l’insertionSort.
output: A è ordinato crescente.
n =  f - i + 1
if n > k then
    m =  (i+f)/2
    MergeInsSort(A,i,m, k);
    MergeInsSort(A,m+1,f,k);
    merge(A,i,m,f);
else
    InsertionSort(A[i:f]);
```

La relazione di ricorrenza

$T(n) = 2T(n/2) + \theta(n)$ se $n>k$
$T(n) = \theta(n^2)$ altrimenti

$n = 2^h$



Quindi $T(n) = jcn + T(k)2^j$
Qual'è il valore di j?
Si deve dividere n per una potenza di 2 fino a che, per la prima volta, $n/2^j \leq k$, cioè fino a quando $n/k \leq 2^j$ cioè fino al $\lg n/k$. Dunque
$T(n) \leq jcn + T(k)2^j = \lg n/k\ cn + T(k)\ n/k = cn(\lg n/k) + dk^2n/k =$
$cn \lg n/k + dnk.$
In conclusione $T(n) = O(n\lg(n/k) + nk)$,
Si può dimostrare che $T(n) = \Theta(n \lg(n/k)+ nk)$

Esercizio 2

Dato un array A di n interi in cui solo lg n sono diversi tra loro mentre i rimanenti sono tutti uguali tra loro, descrivere un algoritmo che ordina A in tempo O(n).
Nell'esempio B ha 4 elementi di cui quindi 2 diversi e 2 uguali e A ha 8 elementi di cui 3 diversi e i rimanenti uguali tra loro.

B=

2	5	2	7
0	1	2	3

A=

5	2	2	3	2	2	7	2
0	1	2	3	4	5	6	7

Suggerimento: Utilizzare l'algoritmo 3-PARTITION che divide gli elementi di una sequenza in tre parti: i più piccoli dell'elemento preso come pivot, gli uguali e a seguire i maggiori.

L’algoritmo

Bisogna individuare gli elementi uguali.
Osserviamo che se n ≥ 8 si ha che lg n = 3 < n/2 = 4 e quindi si hanno necessariamente due uguali consecutivi, altrimenti la ricerca deve essere una normale ricerca di duplicati su un array non ordinato.
Una volta trovato un elemento x che si ripete lo usiamo come perno nella tripartition. Poi ordiniamo le porzioni dei minori e dei maggiori di x con un mergesort.

```
function ORDINA(A)
input: A è un array di n elementi, con n ≥ 8 e in cui solo lg n sono diversi tra loro mentre i rimanenti sono tutti uguali tra loro.
output: A è ordinato crescente.
n = A.length
i  = 0;
while i < n-1 and A[i] ≠ A[i + 1] do i++;
trovo i primi due duplicati consecutivi
A[i] è il valore ripetuto n - log n volte
scambia A[i] con A[n-1]
(j,k) = TriPartizione(A,0,n)
MergeSort(A[1:j])
MergeSort(A[k+1:n])
```

Dove
TriPartizione(A,p,r)
input: un array A e due indici 0≤p≤r≤ A.length
output: due indici i e j, che individuano la prima e l’ultima copia di A[r], gli elementi a sinistra di i sono minori di A[r], quelli a destra di A[j] sono maggiori.

Tempo di esecuzione

function ORDINA(A)
input: A è un array di n elementi, con n ≥ 8 e in cui solo lg n sono diversi tra loro mentre i rimanenti sono tutti uguali tra loro.
output: A è ordinato crescente.
n = A.length
i = 0;
while i < n-1 and A[i] ≠ A[i + 1] do i++; **O(lg n)**
trovo i primi due duplicati consecutivi
A[i] è il valore ripetuto n - log n volte
scambia A[i] con A[n-1] **O(n)**
(j, k) = TriPartizione(A,0,n) **O(lg n *lg lg n)**
MergeSort(A,1,j) **O(lg n *lg lg n)**
MergeSort(A,k+1,n)

Ma lg n* lg lg n = O(n) perché esistono c ed n_0 tali che lg n lg lg n ≤ cn, infatti lg lg n ≤ lg n e lg n ≤ $n^{1/2}$ per n≥16, per cui lg n*lg lg n ≤ c $n^{1/2}$ $n^{1/2}$ = c n, per ogni n ≥ n_0 =16

Esercizio 3

Descrivere un algoritmo per ordinare una sequenza di interi di cui log n sono diversi (ovvero l'insieme determinato dalla sequenza contiene log n elementi) in tempo O(n log log n).
Si può usare O(lg n) memoria aggiuntiva.

Nell'esempio B ha 4 elementi e solo due diversi, mentre A ne ha 8, ma solo 3 diversi.

B=

2	5	2	5
0	1	2	3

A=

5	2	2	5	10	2	10	2
0	1	2	3	4	5	6	7

L’algoritmo

Per individuare, nel limite di tempo O(n log log n), gli elementi distinti di A usiamo due array ausiliari B e M lunghi log n, il primo, B, per i valori diversi di A e il secondo per le rispettive molteplicità.
Quindi M[i] deve essere la molteplicità di B[i] nell’array A.
Per rimanere nei limiti di tempo utilizziamo la ricerca binaria in B per capire se un elemento di A è già presente in B, se lo è si incrementa la sua molteplicità in M, altrimenti va inserito in B ma nella posizione indicata dalla ricerca binaria, così da mantenere B ordinato.
Infine scriviamo gli elementi di B in A tenendo conto delle molteplicità.

Si diano i dettagli dell’algoritmo e lo si analizzi.