Note

1. La leggibilità è un prerequisito: parti difficili da leggere potranno essere ignorate.
2. Quando si presenta un algoritmo è fondamentale spiegare l'idea soggiacente e motivarne la correttezza.
3. L'efficienza è un criterio di valutazione delle soluzioni proposte.

Consegna (1 punto) Consegnare tutti i fogli, con nome, cognome, matricola e l'indicazione esplicita *bella copia* o *brutta copia*.

Domande

Domanda A (5 punti) Dare la definizione di $O(f(n))$. Dimostrare che la ricorrenza che segue ha soluzione $T(n) = O(n)$

$$T(n) = \frac{2}{3}T(n-1) + 2n$$

Domanda B (4 punti) Indicare il codice prefisso ottenuto utilizzando l'algoritmo di Huffmann per l'alfabeto $\{a, b, c, d, e, f, g\}$, supponendo che ogni simbolo appaia con le seguenti frequenze.

a	b	c	d	e	f	g
3	8	7	12	6	23	21

Spiegare il processo di costruzione del codice.

Domanda C (5 punti) Realizzare una funzione `RevCountingSort(A,B,n,k)` che, dato un array `A[1..n]` contenente interi nell'intervallo `[0..k]`, restituisce in `B[1..n]` una sua permutazione ordinata in modo decrescente utilizzando una variante del counting sort. Valutarne la complessità.

Esercizi

Esercizio 1 (7 punti) Scrivere una funzione `RBTree(T)` che dato in input un albero binario di ricerca T, i cui nodi x, oltre ai campi $x.key$, $x.left$ e $x.right$, hanno un campo $x.col$ che può essere B (per "black") oppure R (per "red"), verifica se questo è un Red-Black tree. In caso negativo, restituisce -1, altrimenti restituisce l'altezza nera della radice. Valutarne la complessità.

Esercizio 2 (9 punti) Progettare una struttura dati per la gestione di un insieme dinamico di interi, con operazioni

- $New(S)$ crea un insieme vuoto;
- $Ins(S, x)$ inserisce l'elemento x nell'insieme S;
- $Half(S)$ cancella da S i $\lceil |S|/2 \rceil$ elementi più grandi.

Si richiede che una qualsiasi sequenza di n operazioni venga eseguita in tempo $O(n \log n)$.

i. Specificare le strutture dati di supporto utilite e lo pseudo-codice delle operazioni suddette (questo può ridursi ad una chiamata di un'operazione della struttura scelta).

ii. Dimostrare, mediante un'analisi ammortizzata della complessità, che una sequenza di n operazioni costa $O(n \log n)$.

Nota: Correzione, risultati e visione dei compiti: *Mercoledì 13 Settembre, ore 9:30 (da confermare)*

def: $O(f(n))$

$T(n) = \frac{2}{3}T(n-1) + 2n = O(n)$ dim $\exists c > 0 \; \exists n_0$ trovare condizioni su c ragionevoli

$T(n) \leq cn \quad \forall n \geq n_0$

$T(n) = \underbrace{\frac{2}{3}T(n-1)}_{\leq cn} + 2n \longrightarrow \leq \frac{2}{3}c(n-1) + 2n \overset{?}{\leq} cn$

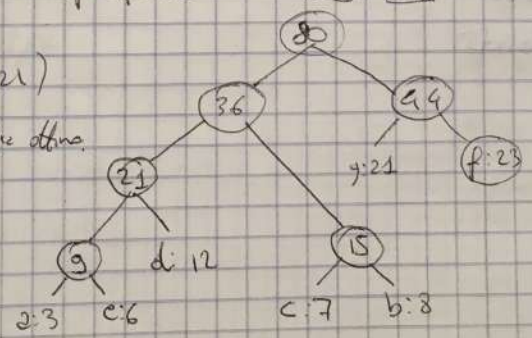
$\ldots \quad c\left(\frac{1}{3}n + \frac{2}{3}\right) \geq \frac{c}{3}n \geq 2n \qquad \forall n \quad c \geq 6$

ci interessa che sia vero $\Rightarrow$ vediamo i casi più semplici possibili

(B) Huffman

a	b	c	d	e	f	g
3	8	7	12	6	23	21

creo albero dove foglie sono numeri con le loro frequenze es [d:12] [e:6]



n.b. nel caso dei 2 (21 g:21) scelgo quello che mi pare $\Rightarrow$ ottengo un codifica ottima.

(c) RevCountingSort(A, B, n, k) I: A[1..n] A[i] ∈ {0..k}
O: B

Θ(k) C[0..k] = 0 // C[i] = # elementi di A di valore i

Θ(n) for i = 1 to n do
  C[A[i]]++

$\forall j \in \{0,\dots,k\}$ voglio sapere quanti elementi in A sono $\leq j$

Θ(k) for j = k-1 to 0 do
  C[j] = C[j] + C[j+1]

C | 0 | 1 | .. | k |

C[j] = quanti valori in A $\leq j$

a ciascun elemento sommo il precedente

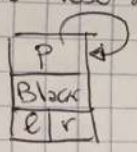
Θ(n) for i = n to 1
  B[C[A[i]]] = A[i]
  C[A[i]]--

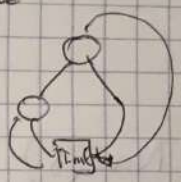
· Per mantenere la stabilità faccio for i: n elementi

Θ(k+n) la complessità

RBtree (T)

T.nil = nodo sentinella






```
if (T.root = T.nil)   albero vuoto e radice
    return 0;  <= altrimenti è T.nil
else if
    return RB_Rec (T.root)
```

↑ funzione ricorsiva

RB-Rec (x) <= return RB height se è RB-tree, -1 altrimenti

```
if (x == T.nil) return 0;
if (x.color == Red) and (x.l.color = R or x.r.color = R)
    return -1
else
    left = RB-Rec (x.left)
    right = RB-Rec (x.right)
    if (left = -1) or (right = -1)   // se uno dei due sottoalberi va soddisfa già
      or (left ≠ right)
        count = -1
    else
        if x.color = Black
            count = left + 1
        else
            count = left
return count;
```

CASO non si usa t.nil

```
if (x.l = nil
    if x.r ≠ nil
        return -1
    else if x.color = R
        return -1
    else
        CASINO
```

(2) New(S)

Ins(S, x)

Half(S)

Idea migliore mantenere un Max-heap ⇒ $A[i] \geq A[2i]\ A[2i+1]$

S = (A, max heap
S.heapsize)

New(S)
  alloco S
  S.heapsize = 0 $(O(1))$

Ins(S, x)
  Heap-Insert(S, x) $O(\log|S|)$

Half(S)
  for i = 1 to S.heapsize $|S| \cdot \log|S|$
    ExtractMax(S)

Heapify(S)

for i=1 to S.heapsize $|S| \cdot \log|S|$
  ExtractMax(S)

Idea: pagare in anticipo il costo di estrazione del costo $\log|S|$

così poi costerà 0 l'estrazione

funz potenziale
$\Phi(S) = \sum_{i=1}^{|S|} \log i = \log|S|!$ ← per tutti gli elementi che ho inserito ho già pagato l'estrazione (ciascuno)

OP	c	$\Delta(\Phi)$	$\hat{c}$
New	1	/	1
Ins	$\log|S|$	$\log|S|$	$2\log|S|$
ExtractMax	$1+\log|S|$	$-\log|S|$	1
Heapify	$1+\sum_{i=\frac{|S|}{2}}^{|S|}$	$-\sum_{i=\frac{|S|}{2}}^{|S|} \log i$	1

n operazioni costano

S_0 S_1 ... S_n struttura iniziale e struttura finale

$S_0 = \emptyset$

$\leq 2 \sum_{i=1}^{n} \log|S_i| \leq 2n \log n$ ($|S_i| \leq n$ sempre)