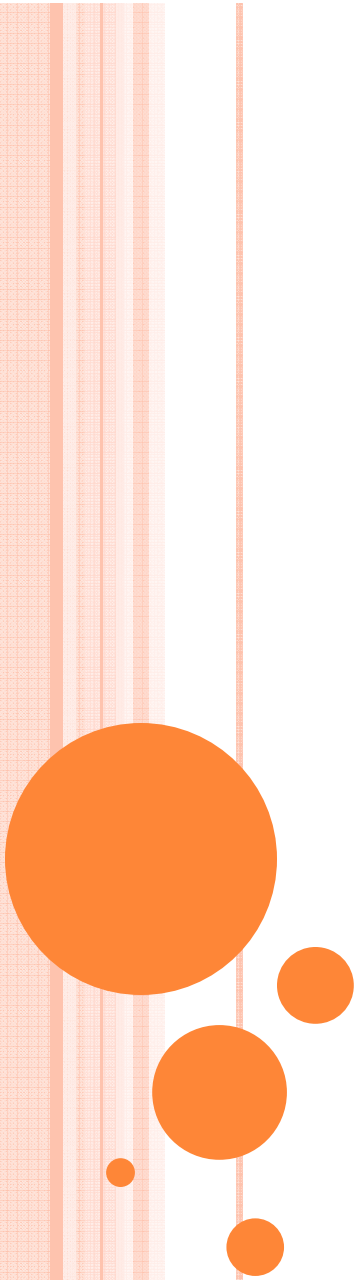

PROVETTE D'ESAME

Algoritmi e Strutture Dati

Esercizio 1

- Si ottengano limiti superiori e inferiori per la seguente ricorrenza

$$T(n) = \begin{cases} 4T(\sqrt{n}) + \log^2 n & n > 1 \\ 1 & n = 1 \end{cases}$$

Soluzione Esercizio 1

Poniamo $n = 2^k$. Sostituendo nella ricorrenza otteniamo:

$$\begin{aligned} T(2^k) &= 4T(\sqrt{2^k}) + \log^2 2^k \\ &= 4T(2^{k/2}) + k^2 \end{aligned}$$

Sostituiamo quindi la variable $T(2^k)$ con $S(k)$ e otteniamo (tramite Master Theorem)

$$S(k) = 4S(k/2) + k^2 = \Theta(k^2 \log k)$$

Ri-esprimendo la funzione nei termini di $T(n)$ e $k = \log n$, otteniamo

$$T(n) = \log^2 n \log\log n$$

ESERCIZIO 2

- Dato un albero binario T, il grado di sbilanciamento di un nodo *v* e' pari alla differenza, in valore assoluto, fra il numero di foglie presenti nel sottoalbero sinistro di *v* e il numero di foglie presenti nel sottoalbero destro di *v*.
- Il grado di sbilanciamento di un albero T è pari al massimo grado di sbilanciamento dei nodi di T.
- Scrivere un algoritmo che dato un albero T restituisce il grado di sbilanciamento dell'albero.
- Discuterne la correttezza e la complessità.

Soluzione Esercizio 2

- Tramite una post-visita, otteniamo per ogni nodo v il numero di foglie contenute nel sottoalbero radicato in v e il massimo grado di sbilanciamento dei nodi contenuti nel sottoalbero radicato in v.

```
RET unbalance(TREE T)
    if T = nil then
        return new RET(0, 0)
    if T.left = nil and T.right = nil then
        return new RET(1, 0)
    L ← unbalance(T.left)
    R ← unbalance(T.right)
    V ← new RET
    V.max ← max(L.max, R.max, |L.leafs − R.leafs|)
    V.leafs ← L.leafs + R.leafs
    return V
```

Soluzione Esercizio 2

- Per semplicità, assumiamo che esista una struttura dati RET con due campi:
  - leafs (numero di foglie)
  - max (massimo grado di sbilanciamento)
- La complessità è ovviamente O(log n).

```
RET unbalance(TREE T)
    if T = nil then
        return new RET(0, 0)
    if T.left = nil and T.right = nil then
        return new RET(1, 0)
    L ← unbalance(T.left)
    R ← unbalance(T.right)
    V ← new RET
    V.max ← max(L.max, R.max, |L.leafs − R.leafs|)
    V.leafs ← L.leafs + R.leafs
    return V
```

ESERCIZIO 3

- Scrivere un algoritmo che, dato un vettore A di n interi distinti (n pari), ritorna *true* se è possibile partizionare A in coppie di elementi che hanno tutte la stessa somma (intesa come la somma degli elementi della coppia), *false* altrimenti.
- Ad esempio: 7;4;5;2;3;6

  può essere partizionato in

  $7 + 2 = 4 + 5 = 3 + 6$
- Discutere la complessità e la correttezza.

Soluzione Esercizio 3

- Si ordini il vettore e si considerino le somme degli elementi i e $n-i+1$, con $1 \leq i \leq n/2$.
- Se sono tutti uguali, si ritorna *true* altrimenti si ritorna *false*.
- L'algoritmo consiste quindi nel scegliere il minore e il maggiore, e confrontarli con i secondi minori e maggiori,

  e così via..

```
checkPairs(integer[ ] A, integer n)
    sort(A, n)
    integer s ← A[1] + A[n]
    for i ← 2 to n/2 − 1 do
        if A[i] + A[n − i + 1] ≠ s then
            return false
    return true
```

Soluzione Esercizio 3

- **Complessità**: il costo dell'algoritmo è dominato dall'ordinamento, ed è quindi O(*nlogn*).
- Dimostrazione **Correttezza**: supponiamo per assurdo che esista un insieme di coppie che rispetti le condizioni per restituire *true*, in cui l'elemento maggiore M sia associato ad un elemento M' diverso dal minore m (m < M').
- Quindi il minore m è associato ad un elemento m' diverso dal massimo M (m'< M).
- Allora m+m'< M+M', il che contraddice l'ipotesi che tale insieme di coppie rispetti le condizioni per restituire *true*.

ESERCIZIO 4: INVERSIONI

- Progettare un algoritmo che dato un vettore V di *n* interi calcola il numero di inversioni.
- Un'inversione è una coppia di indici (i, j) tali che $i < j$ e $V[i] > V[j]$.
- Se ne discuta correttezza e complessità.

- Qualunque soluzione è ammessa, si noti tuttavia che è possibile ottenere una soluzione di costo $O(n \log n)$

Soluzione Esercizio 4

- E' possibile ottenere una soluzione modificando opportunamente l'algoritmo di Merge Sort.
- Durante l'operazione di *merge*, quando si seleziona il valore che si trova nella metà di destra, questo è invertito rispetto a tutti i valori che si trovano nella metà di sinistra e che non sono ancora stati inseriti nel vettore di appoggio.
- E' quindi sufficiente mantenere un contatore a cui verrà sommata la dimensione del vettore mancante. Questo valore, ritornato dall'operazione di merge, verrà progressivamente sommato dalla procedura MergeSort()

Soluzione Esercizio 4

```
Merge(integer A[ ], integer primo, integer ultimo, integer mezzo)
integer i, j, k, h
i ← primo; j ← mezzo + 1; k ← primo
counter ← 0
while i ≤ mezzo and j ≤ ultimo do
    if A[i] ≤ A[j] then
        B[k] ← A[i]
        i ← i + 1
    else
        counter ← counter + (mezzo − i + 1)
        B[k] ← A[j]
        j ← j + 1
    k ← k + 1
j ← ultimo
for h ← mezzo downto i do
    A[j] ← A[h]
    j ← j − 1
for j ← primo to k − 1 do  A[j] ← B[j]
return counter
```

Soluzione Esercizio 4

```
CountInversion(integer A[ ], integer primo, integer ultimo)
if primo < ultimo then
    integer mezzo ← ⌊(primo + ultimo)/2⌋
    return CountInversion(A, primo, mezzo) + CountInversion(A, mezzo + 1, ultimo)
                                           + Merge(A, primo, ultimo, mezzo)
```

ESERCIZIO 5: ANAGRAMMI

- Si supponga di avere in input un vettore di n stringhe di lunghezza massima k;
- Si scriva un algoritmo che stampa in output tutti i gruppi di anagrammi contenuti in queste n stringhe.
- Se ne discuta correttezza e complessità.
- Esempio di input:
  - rosa
  - pippo
  - poppi
  - raso
  - orsa
  - giappone
- Esempio di output:
  - rosa, raso, orsa
  - pippo, poppi

Soluzione Esercizio 5

- Ordiniamo tutte le stringhe internamente, ovvero una per una. Ad esempio, *rosa* diventa *aors*. (Tutte le stringhe che sono una anagramma dell'altra, una volta ordinate, coincidono).
- Si potrebbero ordinare le stringhe, ma la complessità sarebbe non ottimale.
- Invece se utilizziamo poi una tabella hash per identificare le stringhe che coincidono, il costo di questo algoritmo è $O(nk \log k + n)$.

Soluzione Esercizio 5

```
anagrams(Item [ ][ ] S, integer n)
    Hash H ← Hash()
    for i ← 1 to n do
        sorted ← sort(S[i])
        Set S ← H.lookup(sorted)
        if S = nil then
            S ← Set()
        S.insert(S[i])
        H.insert(sorted, S)
    foreach x : H.lookup(x) ≠ nil do
        Set S ← H.lookup(x)
        if S.dim() > 1 then
            print S
```