Problema 1 Analisi algoritmo

Si considerino i metodi Java di seguito illustrati.

```
// n >= k >= 0
static int bin(int n, int k) {
    if(k == 0 || k == n) return 1;
    if(k == 1 || k == n-1) return n;
    if(k > n-k) k = n-k;
    return prod(n, n-1, 2, k-1);
}
```

```
static int prod(int acc, int p, int q, int c) {
    int r = (acc*p)/q;
    if(c == 1) return r;
    return prod(r, p-1, q+1, c-1);
}
```

Sviluppare, *argomentando adeguatamente* (il 50% del punteggio dell'esercizio sarà sulle argomentazioni addotte), quanto segue:

(a) Determinare il costo temporale asintotico dell'algoritmo descritto da `bin(int,int)` in funzione della dimensione dell'input. [4/30]

(a)

L'algoritmo bin(int,int) è una funzione che presenta in ingresso due variabili intere "n" e "k" e

restituisce il valore del binomio (n k)^t (traposto).

la funzione effettua prima tre controlli sul parametro k,

di costo costante, se tali controlli vengono "superati" la funzione restituisce il valore di ritorno della funzione prod(int,int,int,int).

Tale funzione è utilizzata per il calcolo vero e proprio del binomio e viene eseguita k-2 volte in caso venisse richiamata

da bin(int,int). Dunque il costo asintotico del'algoritmo bin(int,int) dipende solo

ed esclusivamente dalla grandezza del parametro k.

Infatti se tale variabile soddisfa uno dei tre casi iniziali dell'algoritmo,

allora quest'ultimo viene eseguito in tempo costante ovvero pari a O(1), mentre

in caso contrario, bin(int,int) richiamerà k-2

volte la funzione prod(int,int,int,int) di costo costante pari a O(1). Dunque il costo nel caso peggiore è O(k).

Essendo k un numero, e non ad esempio la dimensione di un array, è possibile esprimere il valore k, come 2^z cioè pari al numeri di bit utilizzati per rappresentare tale valore k,

2^z = k ==> z = log k ==> il costo asintotico del algoritmo bin(int,int) è pari a O(2^(logk))

(b) Definire il significato di algoritmo *in-place* e, trascurando la memoria impiegata per gestire le ricorsioni runtime, spiegare se `bin(int,int)` è in-place oppure no. [2/30]

(b)

Un algoritmo in-place è un algoritmo che si effettua "sul posto", ad esempio insertion_sort è un algoritmo di ordinamento

in-place poiché ordina gli elementi

scambiandoli sullo stesso array in cui sono memorizzati e dunque non utilizza strutture dati

di appoggio, altri esempi sono heapify, selection-sort...

Un algoritmo potrebbe essere definito in-place se il costo dello spazio

di memoria utilizzato rimane costante, cioè pari a O(1).

Se trascuriamo la memoria impiegata per gestire le ricorsioni in runtime,

entrambi gli algoritmi presentano costo unitario O(1), e dunque si può affermare

che bin(int,int) è un algoritmo in-place.

N.B.: il costo dello spazio utilizzato da prod considerando la memoria impiegata per gestire le ricorsioni è pari a O(k)

Lineare rispetto all'input (esponenziale rispetto al numero di cifre)

Per ogni n, k interi,

se

) k = n

) k = n - 1

) k = 1

) k = 0

allora bin(int, int) e Theta(1)

altrimenti (caso peggiore), dato che prod(int, int)

chiama e ritorna prod(int, int, int, int), ha il suo stesso costo asintotico. Poichè il costo di prod(int, int, int, int)

dipende soltanto dall'ultimo (quarto) parametro, il costo è dato da (essendo q l'ultimo parametro)

{

T(q) = T(q - 1) + k

{ T(1) = k

T(q) = T(q - 1) + k = T(q - 2) + 2k = T(q - 3) + 3k = T(q - a) + 2a

scegliendo a = q - 1

T(q) = k + 2(q - 1), cioè Theta(q)

Inoltre, poichè se 2k > n allora q = n - k, se 2k > n

il costo totale dell'algoritmo sarà Theta(n-k), altrimenti sarà Theta(k)

cioè lineare (esponenziale rispetto al numero di cifre)

(a) Descrivere accuratamente (codice o pseudo-codice) l'algoritmo di Horner per determinare il valore di un polinomio in un punto [3/30] e dimostrarne il costo lineare [1/30].

a) Si ha che un polinomio dato da a_0x^0 + a_1x^1 + a_2x^2 + ... a_nx^n può essere riscritto

come a_0x^0 + x(a_1 + x(a_2 + x(....x(a_n-1 + a_nx)

perciò possiamo calcolare un polinomio come:

polinomio:

input: x, int[] coeff, int size , dove coeff è l'array di coefficienti e size è la sua dimensione

output:

int valore del polinomio

if(array = null) throw new IllegalArgumentException;

if(x = 0) return array[0]

if(x = 1) return sommatoria per i da 0 a size-1 di array[0]

else

return polinomio_aux(array[size -2] + array[size -1]*x, x, array, size -2);

polinomio:

input: v, x, int[] coeff, int size, dove v è il risultato del calcolo precedente

output: int valore del polinomio

if(size = 1)

return array[0] + v;

else

return polinomio(array[size - 1] + x * v, x, array, size - 1);

Questo algoritmo esegue, nel caso peggiore, size - 1 ricorsioni, dove size è il grado del polinomio,

e ad ogni ricorsione effettua operazioni a costo costante. Perciò ha costo

lineare Theta(size).

(b) Progettare un algoritmo (codice o pseudo-codice) che dati n array di interi già ordinati in senso non decrescente, costruisca e restituisca un nuovo array ordinato in senso non decrescente contenente tutti e soli i valori presenti negli array in input. Non sono ammessi side-effects. Assumere che l'input sia fornito attraverso un array di n riferimenti agli array ordinati; l'output sarà fornito attraverso un riferimento al nuovo array. Analizzare il costo dell'algoritmo.1 [5/30]

E' possibile inserire i valori degli array in un min_heap (o max_heap) e poi estrarli ad uno ad uno,

ponendoli in un array di dimensione sufficiente. Tale algoritmo, simile ad

un heapsort,

avrebbe perciò costo O(P * log(P)) dove P è la somma delle dimensioni di tutti gli array

merge:

input: int[n][] arrays, array di n array

output: int[size]

int size <- 0;

MinHeap h <- new MinHeap()

for(Array a : arrays)

for(int n: a)

{

h.insert(n);

size++;

}

int result[] = new int[size];

for(int i <- 0; i < size; i++)

result[0] <- h.removeMin();

return result;

(c) Per descrivere la struttura statica di un programma P viene costruito un grafo diretto i cui nodi sono associati alle funzioni (o metodi) di P e che contiene un arco diretto (u, v) se la funzione u contiene almeno una chiamata alla funzione v. Il main viene associato a un nodo denominato s. Il grafo ammette cappi, ovvero archi del tipo (w, w), nel caso la funzione w sia ricorsiva

Ciò premesso, progettare un algoritmo (codice o pseudo-codice) che, dato un grafo diretto che descrive un programma P, decide se P è privo di ricorsioni di qualunque tipo: sia quelle dirette (le classiche) che le indirette, vale a dire schemi ricorsivi in cui una funzione t_1 può chiamare una funzione t_2, che a sua volta può chiamare t_3, ecc., fino a chiamare $t_h = t_1$, per $h > 1$. [4/30]

c) E' sufficiente una visita DFS, di costo O(|V| + |E|) essendo |V| il numero di vertici ed |E| il numero di archi.

Se viene trovato un arco back,

allora esiste un ciclo di chiamate e perciò una ricorsione

Per semplicità,

assumiamo che il grafo sia rappresentato tramite liste di adiacenza e di avere definite

```
        Enum Status <- {UNEXPLORED, EXPLORING, EXPLORED};

        class Info {

Status status;

Iterator neighborsIterator;
        }

trova_ricorsioni:

input: Graph g

output: uno stack che ha in cima un ciclo, se ne esistono;
null altrimenti

HashMap<GraphNode, Info> map = new HashMap();

        for(GraphNode nd : g.getNodes())

map.put(nd, new Info(UNEXPLORED, 0, nd.getNeighbors().getNeighborsIterator());

Stack<GraphNode> stack <- new Stack();  //pila per mantenere il record di attivazione, così da evitare ricorsioni

while( status[i] per i qualsiasi = UNEXPLORED)
        {
            for(GraphNode nd : nd.getNodes())
```

```
if(map.get(nd).getStatus().equals(UNEXPLORED))

{

  stack.push(nd);

        break;

                }

            if(stack.isEmpty()) return null;

            while(!stack.isEmpty())

{

if(!map.get(stack.peek()).getNeighborsIterator().hasNext())

{

                        map.get(stack.peek()).setStatus(EXPLORED);
                        stack.pop();

}
GraphNode nextNeighbor = map.get(stack.peek()).getNeighborsIterator().next();

GraphNode nextNeighborInfo = map.get(nextNeighbor);

if(nextNeighborInfo.getStatus().equals(UNEXPLORED)) //arco tree
```

```
{
                        nextNeighborInfo.setStatus(EXPLORING);

stack.push(nextNeighbor);
                        continue;
                }

else if(nextNeighborInfo.getStatus().equals(EXPLORING)  //arco back (può essere anche un cappio), ciclo!
                {
                        stack.push(nextNeighbor);

return stack;

}                                       //non ci interessano gli archi cross e forward

}
        }
```

L'utilizzo di una HashMap per accedere a dei dati e di una pila

invece che della ricorsione non inficiano sul costo totale, poichè la hashmap ha costo di accesso costante,

perchè il numero dei nodi non cambia perciò il fattore di carico resta buono; mentre gestire la pila ha lo

stesso costo asintotico di un meccanismo ricorsivo, cioè O(1), e dovrebbe

essere anche più veloce in pratica

Problema 4 Calcolo e stampa di expression trees

Un expression tree è un albero binario che rappresenta una espressione aritmetica su un insieme di valori. Nel caso in esame prendiamo in considerazione expression trees in cui ciascun nodo interno è associato a un'operazione binaria in $\{+, -, \times, /\}$ su operandi memorizzati nelle foglie come costanti in virgola mobile. Nell'esempio in figura l'expression tree rappresenta l'espressione $(3 + (5 - 1))/(2 \times (7/4))$.

algoritmo : print_expression

print_expression:

input: bst, che rappresenta un'espressione

output: stampa sullo schermo l'espressione rappresentata dal bst

if(bst = null) return;

else call print_expression_aux(bst.getRoot());

print_expression_aux:

input: nd, radice del bst

output: stampa sullo schermo l'espressione rappresentata dal bst

```
if((nd.getLeftChild() = null)&&(nd.getRightChild() != null))

print("(" + nd.getRightChild().getValue() + ")");

else if((nd.getLeftChild() != null)&&(nd.getRightChild() = null))

print("(" + nd.getLeftChild().getValue() + ")");

else if((nd.getLeftChild() = null)&&(nd.getRightChild() = null))
```

```
print("(" + nd.getValue() + ")");


else
        {
print("(");


print_expression_aux(nd.getLeftChild);


print(nd.getValue());


print_expression_aux(nd.getRightChild);


print(")");     }
```