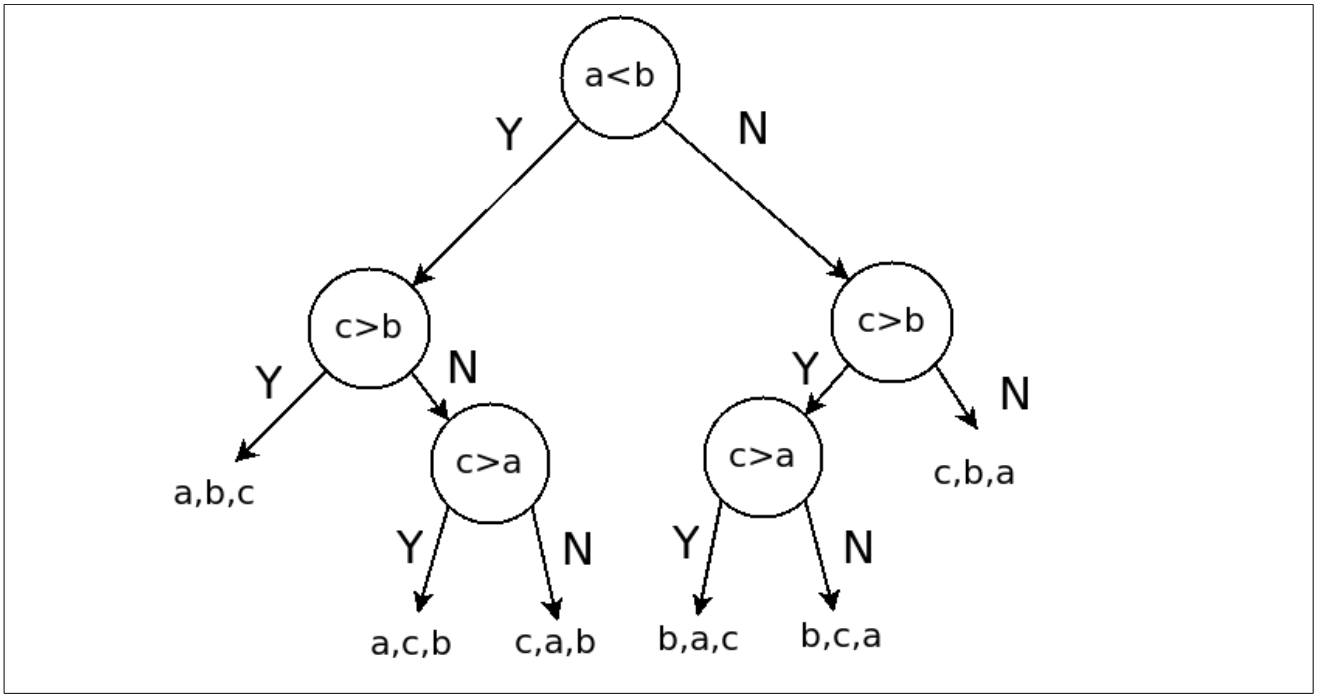
FONDAMENTI DI INFORMATICA II – modulo Algoritmi e Strutture dati

24 luglio 2009 - ANNO ACCADEMICO 2008/2009

NOME: **COGNOME:** **MATRICOLA:**
PROGETTO: **SI** **NO** **(barrare SI se si intende svolgere lo scritto ridotto)**

Esercizio 1
Mostrare l'albero di decisione dell'algoritmo di ordinamento quicksort con un array di 3 elementi a,b c.



Esercizio 2
Sia dato il seguente heap:

[90 87 16 75 39 12 0]

mostrare lo stato dello stesso e le chiamate ad up e down:

A) dopo l'inserzione dell'intero 45
B) dopo l'estrazione di un elemento dallo heap ottenuto al passo A

[90 87 16 75 39 12 0 45]
up(7)

[87 75 16 45 39 12 0]
down(0), down(1), down(3)

Situazione	*Esercizi*	*Tempo*
Senza progetto	**1,2,3,4,5**	80 min
Con progetto	**1,2,3**	40 min

1	*2*	*3*	*4*	*5*

Esercizio 3
Calcolare la complessità del blocco (indicando le relazioni di ricorrenza di tempo e risultato per ogni funzione) in funzione di n:

```
{
  int a = 0;
  for (int i=0; i <= g(n) + f(n); i++)
     a += 1;
}
```

con le funzioni **f** e **g** definite come segue:

```
int f(int x) {
   if (x<=1)
     return 1;
return 2*f(x/2) + 2*f(x/2)+1;
}
```

```
int g(int x) {
  if x <= 1 return 1;
  int a=0;
  for (int i=0; i <= f(x)*f(x); i++)
     a++;
  return a + g(x-1);
}
```

Indicare per esteso le relazioni di ricorrenza e, per ogni comando ripetitivo, il numero di iterazioni e la complessità dell'iterazione singola.

$T_f(1)= k_1$
$T_f(n)= k_2 + 2T_f(n/2)$ $T_f(n)$ è $O(n)$
$R_f(1)= 1$
$R_f(n)= 1 + 4R_f(n/2)$ $R_f(n)$ è $O(n^2)$

Calcolo $T_g(n)$
for:
numero iterazioni: $O(n^2*n^2)=O(n^4)$ Complessità singola iterazione: $O(n)$
complessità del for: $O(n^5)$

$T_g(1)= k_1$
$T_g(n)= k_2\ n^5+ T_g(n-1)$ $T_f(n)$ è $O(n^6)$
$R_g(0,1)= 1$
$R_g(n)= n^4 + R_g(n-1)$ $R_f(n)$ è $O(n^5)$

Calcolo for del blocco:
numero iterazioni: $R_g(n) = O(n^2)+O(n^5) = O(n^5)$
Complessità singola iterazione: $T_f(n) + T_g(n) = O(n) + O(n^6) = O(n^6)$

Complessità totale blocco = $O(n^5)* O(n^6) = O(n^{11})$

Esercizio 4

Sia dato un albero binario di ricerca ad etichette di tipo intero. Si scriva una funzione

`Node* cercapadre(Node* t, int key)`

che restituisce il puntatore al nodo padre del nodo con etichetta **`key`**.

Supponendo che l'albero sia quasi bilanciato, si calcoli la complessità di **`cercapadre`** in funzione del numero di nodi n dell'albero,

```
Node* cercapadre(Node* t, int key) {
  if (!t || t->label == key)
    return NULL;
  if (t->label>key) {
    if (!t->left) return NULL;
    if (t->left->label == key)
      return t;
    return cercapadre(t->left,key);
  }
  // t->label < key
  if (!t->right)
    return NULL;
  if (t->right->label == key)
    return t;
  return cercapadre(t->right,key);
}
```

Complessità O(log n)

Esercizio 5

Scrivere una funzione **`bool maxheap(Node* t)`** che ritorna **`true`** se l'albero binario bilanciato di radice **`t`** è un max-heap. Si supponga che l'albero contenga solo etichette intere positive. Si calcoli la complessità della soluzione proposta in funzione del numero di nodi dell'albero.

```
bool maxheap (Node* t, int &x) {
if (!t) { x=-1; return true; }
int x_l, x_r;
bool left=maxheap(t->left, x_l);
bool right=maxheap(t->right, x_r);
x=max(t->label,max(x_l,x_r));
return left && right && (t->label >= x);
}
```

```
bool maxheap2 (Node* t) {
  if (!t || (!t->left))
    return true;
  return ((t->label >= t->left->label) &&
   (t->label >= t->right->label) &&
   maxheap2(t->left) &&
   maxheap2(t->right))
}
```

In entrambi i casi la complessità è O(n)