Prova_parziale 2021_01_18 soluzione

Sia *T* un albero binario i cui nodi *x* hanno i campi *left*, *right*, *key*. L'albero si dice **k-limitato**, per un certo valore *k*, se per ogni nodo *x* la somma delle chiavi lungo ciascun cammino da *x* ad una foglia è minore o uguale a *k*.

a. Scrivere una funzione **efficiente in C** *k-limitato(u,k)* che dato in input la radice *u* di un albero *T* e un valore *k* verifica se *T* è *k-limitato* e ritorna *1* se *T* è *k-limitato*, *0* altrimenti.
b. Valutare la complessità della funzione, **indicando eventuali relazioni di ricorrenza**.

```
#include <iostream>
using namespace std;

struct Node {
        int key;
        Node* left;
        Node* right;
        Node(int val): key(val), left(nullptr), right(nullptr) {}
};

int klimitatoAux(Node* u, int k,int tot){
        if(u == nullptr) return 1; //O(1)
        tot += u->key;
        int left = klimitatoAux(u->left, k, tot);
        int right = klimitatoAux(u->right, k, tot);
        if(tot <= k && left == 1 && right == 1){
                return 1;
        } else {
                return 0;
        }
}

int klimitato(Node* u, int k){
        if(u == nullptr) return 0;
        int count = 0; // O(1)
        return klimitatoAux(u,k,count);
}

int main(){
        Node* root = new Node(10);
        root->left = new Node(20);
        root->left->left = new Node(30);
        root->left->right = new node(40);
        root->right = new Node(40);
        root->right->right = new Node(30);
        root->right->left = new Node(10);
        cout << klimitato(root, 100) << endl;
        return 0;
}
```

Si deve organizzare una gara di programmazione. Ogni programmatore ha un punteggio che esprime la sua abilità (più alto è il punteggio migliore è il programmatore). Ogni programmatore è abbinato a un altro programmatore e la differenza fra i loro punteggi è detta "scarto".

a. Scrivere un algoritmo **efficiente** *int scarto(int n, int punteggi[])* che dati *n* programmatori con *n pari* e i loro *punteggi* restituisce il **minimo scarto totale** (somma degli scarti delle varie coppie) che si può ottenere pianificando in modo ottimale le coppie nella gara.
b. Calcolare e giustificare la complessità dell'algoritmo proposto.

Si devono scrivere le eventuali procedure/funzioni ausiliarie utilizzate.

```cpp
#include <iostream>
using namespace std;

void swap(int& a, int& b){
	int temp = a;
	a = b;
	b = temp;
}

int partition(int punteggi[], int low, int high){
	int i = low;
	int pivot = punteggi[high];

	for(int j = low; j < high; j++){
		if(punteggi[j] < pivot){
			swap(punteggi[j],punteggi[i]);
			i++;
		}
	}

	swap(punteggi[i],punteggi[high]);
	return i;
}

void quick(int punteggi[], int low, int high){
	if(low < high){
		int pivot_location = partition(punteggi, low, high);
		quick(punteggi, low, pivot_location-1);
		quick(punteggi, pivot_location+1, high);
	}
}

int scarto(int n, int punteggi[]){
	quick(punteggi,0,n-1);// ricordo che n è pari
	int scart = 0;
	for(int i = 0; i < n; i += 2){
		scart += punteggi[i] + punteggi[i+1];
	}
	return scart;
}

int main(){
	int arr[12] = {0,2,5,3,8,12,9,1,7,13,17,16};
	cout << scarto(12, arr) << endl;
	return 0;
}
```

- La complessità di questo algoritmo risulta essere:
  - Essendoci un $O(n^2)$ e $O(n)$ per via del quicksort e del for, prevale la complessità del quicksort, portando la funzione ad una complessità $\Theta(nlogn)$ con caso peggiore $O(n^2)$.

Si calcoli la complessità asintontica dei seguenti algoritmi (in funzione di n) e si stabilisca quale dei due è preferibile per n sufficiente grande:

```
MyAlgorithm1( int n )
int
  a, i, j

if ( n > 1 ) then
   a = 0
   for i = 1 to n
      for j = 1 to n
         a = a + (i+1)*(j+1)
      endfor
   endfor
   for i = 1 to 7
      a = a + MyAlgorithm1(n/3)
   endfor
   return a
else
   return n-1
endif
```

```
MyAlgorithm2( int n )
int
  a, i

if ( n > 1 ) then
   a = 0;
   for i = 1 to n
      a = a + (i+1)*(i+2)
   endfor
   for i = 1 to 16
      a = a + MyAlgorithm2(n/4)
   endfor
   return a
else
   return n-1
endif
```

Si forniscano giustificazioni *formali*. In caso contrario l'esercizio non verrà valutato pienamente, anche in presenza di risposte corrette.

- Algoritmo numero 1:
  - Al di fuori delle operazioni costanti come dichiarazioni e for ripetuti per un tempo costante, troviamo un doppio for annidato e una chiamata ricorsiva effettuata sette volte. Dovendo studiare quando $n \to \infty$ abbiamo che $T(n) = 7T(n/3) + \Theta(n^2)$. ora troviamo che $n^{log_3 7}$ fa $n^{1.33}$. Utilizzando il teorema Master ci troviamo nella terza casistica dove $f(n) = \Omega(n^{log_b a+\epsilon})$, dove in questo caso $\epsilon = 0.67$. Bisogna comunque controllare la *regularity condition* $af(n/b) \leq cf(n)$ per una costante $c < 1$, quindi abbiamo $7(\frac{n}{3})^2 \leq cn^2$ con c risultante $\frac{7}{9}$. Concludiamo dicendo che il tempo di esecuzione della funzione è $\Theta(n^2)$.
- Algoritmo numero 2:
  - Utilizzando lo stesso sistema usato nel precedente esercizio, notiamo un for singolo effettuato n volte e una chiamata ricorsiva eseguita 16 volte. Da qui arriviamo quindi ad avere $T(n) = 16T(n/4) + \Theta(n)$. in questo caso avremo un n^2 trovandoci così nel primo caso, dove $f(n) = O(n^{log_b a-\epsilon})$ con conseguente tempo di esecuzione della funzione $\Theta(n^2)$.