Esercizi – Strutture Dati 3

Anno Accademico 2022/2023

Dott. Staccone Simone

TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA

Esercizio 2 - 26/01/2022

Supponiamo di avere numeri compresi tra 1 e 100 in un albero di ricerca binario e di voler cercare il numero 45. Quale delle seguenti sequenze (anche più di una) potrebbe essere la sequenza di nodi esaminata?

- 5, 2, 1, 10, 39, 34, 77, 63.
- 1, 2, 3, 4, 5, 6, 7, 8.
- 50, 25, 26, 27, 40, 44.
- 50, 25, 26, 27, 40, 44, 42.

Esercizio 2 - 26/01/2022

- 5, 2, 1, 10, 39, 34, 77, 63 non è una possibile sequenza da controllare durante una ricerca di 45 in un albero di ricerca binario. Il primo elemento della sequenza deve essere l'elemento radice. Quando confrontiamo la chiave radice 5 con la chiave ricercata 45, vediamo che 5 < 45, e quindi 45, se presente nell'albero, deve essere nell'albero figlio destro del nodo radice. Il secondo elemento della sequenza (2) dovrebbe quindi essere la radice del sotto albero destro del nodo radice. Ma tutte le chiavi nel sotto albero giusto sono rigorosamente maggiore di 5, quindi 2 non può essere la radice chiave del sotto albero destro.

- 1, 2, 3, 4, 5, 6, 7, 8 è una possibile sequenza. Essa è la sequenza che viene controllata nell'albero degenerato costituito dagli elementi 1; 2; 3; 4; 5; 6; 7; 8, dove nessun nodo ha un figlio sinistro.

- 50, 25, 26, 27, 40, 44 è una possibile sequenza.

- 50, 25, 26, 27, 40, 44, 42 non è possibile. Tutti i passaggi sono nella giusta direzione tranne l'ultimo. Da 44 dovremmo scendere nel figlio giusto, che ha una chiave maggiore di 44. Andiamo invece a 42.

Esercizio 3 - 15/06/2022

Dato un array ordinato A di n interi, alcuni dei quali negativi, fornire una codifica nel linguaggio C di un algoritmo che trovi un indice i tale che: $0 \leq i \leq n - 1$ e $A[i] = i$ se tale indice esiste, -1 altrimenti. In caso di esistenza di più di un indice che soddisfi la condizione, l'algoritmo ne può tornare uno qualsiasi.

Quale è il valore minimo possibile per la complessità dell'algoritmo proposto?

*Si omette che nel vettore A non ci siano ripetizioni

Esercizio 3 - 15/06/2022

Due possibili soluzioni:

- Se il vettore non è ordinato → scansione lineare O(n)
- Se il vettore è ordinato → ricerca binaria su albero O(log(n))

```c
#include "utils.h"

int sol_iter(int n, int *arr){
        int i;
        for(i = 0;i<n;i++){
                if(arr[i] == i)
                        return i;
        }

        return -1;
}

int sol_rec(int n,int *arr){
        //TODO

        return 0;
}
```

Esercizio 3 - 15/06/2022

Sketch soluzione iterativa in python:

```
int binary_solution(int A[], int b, int c) {
    int m;
    if (b > c) { return -1; }

    m = ((c + b)/2);
    if (A[m] = m) { return m; }
    if (A[m] > m) {
        return(binary_solution(A, b, m-1));
    } else {
        return(binary_solution(A, m+1, c));
    }
}
```

Valid Parenteses

Data una stringa s contenente solo i caratteri '(', ')', '{', '}', '[‘ e ']’, determinare se la stringa di input contiene un incapsulamento di parentesi valido.

Un input è valido se:

1. Le parentesi aperte devono essere chiuse dallo stesso tipo di parentesi.
2. Le parentesi aperte devono essere chiuse nell’ordine corretto.
3. Tutte le parentesi chiuse hanno una parentesi aperta dello stesso tipo.

Esempi:

Input: ‘( )’ Output: True

Input: ‘( ) [ ] { }’ Output: True

Input: ‘( ]’ Output: False

Input: ‘([( ) [ ]])’ Output: True

Input: ‘([( ) {] })’ Output: False

Valid Parenteses

- L’idea è quella di utilizzare uno stack per controllare il matching delle varie parentesi
- Ogni qual volta trovo una parentesi aperta faccio il push sullo stack.
- Ogni qual volta trovo una parentesi chiusa faccio il pop sullo stack e verifico che la parentesi di cui ho fatto il pop è dello stesso tipo della parentesi chiusa considerata.
- Inoltre controllo la lunghezza della stringa (Se è di lunghezza dispari sono sicuro che l’output dovrà essere false)

Valid Parentheses

```
bool isValid(char* s) {
        int len = strlen(s);
        if (len % 2 != 0)
                return false;

        int n = len / 2;
        char *stack = malloc(n);
        int idx = 0;
        char c, top;

        for (int i = 0; i < len; i++) {
                c = s[i];
                if (c == '(' || c == '[' || c == '{') {
                        if (idx == n)
                                return false;
                        else
                                stack[idx++] = c;
                } else {
                        if (idx == 0)
                                return false;
                        top = stack[idx - 1];
                        if (top == '(' && c == ')' || top == '[' && c == ']' || top == '{' && c == '}') {
                                idx--;
                        }else
                                return false;
                }
        }
        return idx == 0;
}
```

K-Shift

Sia A un vettore ordinato di dimensione contenente n interi positivi distinti, che è stato traslato ("shiftato") di k < n posizioni a destra.

Ad esempio, il vettore [35, 42, 1, 7, 15, 18, 28, 30] è un vettore ordinato che è stato shiftato k = 2 posizioni a destra, mentre il vettore [15, 18, 28, 30, 35, 42, 1, 7] è un vettore ordinato che è stato shiftato k = 6 posizioni a destra.

Scrivere un algoritmo che prende un vettore ordinato traslato A come input e restituisce il valore di k corrispondente.

L'algoritmo deve operare in tempo O(log n). Discutere informalmente la correttezza della soluzione proposta e calcolare la complessità computazionale.

K-Shift

Il problema può essere risolto modificando opportunamente la ricerca dicotomica, in cui cerchiamo di individuare la posizione k del minimo assumendo che il vettore sia stato traslato di un fattore k.

L'idea è la seguente: si considera un sottovettore compreso fra gli indici i e j, e si considera l'elemento centrale m = b(i + j)/2 (prendendo la parte intera inferiore).

• Se A[i] > A[m], allora il minimo si troverà fra i ed m, in quanto il sottovettore di sinistra non è ordinato e quindi il minimo si troverà in quel sottovettore; si noti che necessariamente si ha che A[m] < A[j], in quanto se fosse A[m] > A[j] il vettore sarebbe ordinato in senso opposto.

• Se A[m] > A[j], allora il minimo si troverà fra m e j, in quanto il sottovettore di destra non è ordinato e quindi il minimo si troverà in quel sottovettore; si noti che necessariamente si ha che A[i] < A[m], in quanto se fosse A[i] > A[m] il vettore sarebbe ordinato in senso opposto.

• Se entrambe le condizioni non sono vere, allora A[i] < A[m] < A[j], e il vettore è ordinato. In questo caso bisogna restituire 0.

K-Shift

Come caso base, quando si è rimasti con due elementi, quindi j = i + 1, allora il minimo si trova in j (se l'array fosse ordinato, avremmo già restituito zero) e il numero di shift è pari a j − 1 (in vettori che iniziano da 1). Una dimostrazione più precisa dovrebbe essere basata su induzione sulla dimensione del sottovettore, con caso base 2

```
int shift(int[] A, int i, int j)
    if j == i + 1 then
        return j − 1
    else
        int m = ⌊(i + j)/2⌋
        if A[i] > A[m] then
            return shift(A, i, m)
        else if A[m] > A[j] then
            return shift(A, m, j)
        else
            return 0
```


La complessità è O(log(n)) dato che deriva dalla ricerca dicotomica.

Possible Sums

Scrivere un algoritmo che, dati in input un vettore X contenente n interi distinti e un intero w, stampi tutti i modi in cui è possibile ottenere esattamente il valore w sommando un sottoinsieme dei valori contenuti in X.

Ad esempio, se X = [−1, 3, 6, 4, 5, 7, 8] e w = 13, l'algoritmo deve stampare, non necessariamente in questo ordine:

5 + 8
6 + 7
3 + 6 + 4
-1 + 6 + 8
-1 + 3 + 4 + 7
-1 + 3 + 6 + 5

Discutere informalmente la correttezza della soluzione proposta e calcolare la complessità computazionale. Non perdete tempo a realizzare funzioni di stampa "sofisticate", è sufficiente stampare – per ogni somma - l'elenco dei numeri che la compongono.

Possible Sums

```
printSums(int[] X, int n, int v, int i, int[] S)
if i == n + 1 then
    if v == 0 then
        printVector(X, S, n)
else
    foreach c ∈ {0, 1} do
        S[i] = c
        printSums(X, n, v − c · X[i], S, i + 1)
```

```
printVector(int[] X, int[] S, int n)
int i = 1
boolean first = false
while i < n do
    if S[i] == 1 then
        print X[i]
        if not first then
            print " + "
            first = true
    println
```

Possible Sums

- Utilizziamo la tecnica di backtrack, enumerando tutti i possibili sottoinsiemi.

- La funzione ricorsiva printSums() prende in input il vettore X e la sua dimensione; prende inoltre in input il valore target v, che viene decrementato tutte le volte che si sceglie un valore. Per realizzare il backtrack, si passa anche una maschera di booleani (valori 0/1) che rappresenta l'insieme di valori scelti. Il parametro i invece rappresenta l'attuale elemento del vettore i che viene considerato, analizzato dall'ultimo al primo. La chiamata iniziale è printSums(X, n, v, 0, S).

- Per ogni elemento X[i], ci sono due possibilità: lo utilizziamo oppure no in una possibile somma. Questo è rappresentato dalla linea foreach $c \in \{0, 1\}$.

- Memorizziamo la scelta in S[i] e chiamiamo printSums() ricorsivamente, modificando v in maniera opportuna. Questo ci porta ad esplorare un albero di decisione di dimensione 2^n .

- In generale, nel caso pessimo ogni volta che si realizza la somma vengono stampati fino ad n valori, quindi la complessità è $O(n2^n)$

Find Single

Sia A un vettore contenente n ≥ 1 interi, non necessariamente ordinato, con n dispari. Nel vettore sono memorizzati un certo numero di interi distinti; a parte un valore che compare una volta sola, tutti gli altri interi compaiono due volte in posizioni consecutive.

Scrivere un algoritmo int findSingle(int[ ] A, int n) che restituisca l'intero che compare una volta sola. Spiegare il funzionamento e discutere la complessità computazionale dell'algoritmo proposto. Soluzioni in tempo lineare o superiore non verranno considerate.

Ad esempio, con l'input A = [1, 1, 4, 4, 2, 2, 0, 1000, 1000, −2, −2] l'algoritmo deve restituire 0 in quanto l'unico numero che compare una volta sola

Find Single

- Il problema è risolvibile tramite un algoritmo divide-et-impera che considera il sottovettore A[i . . . j],
- inizialmente A[1 . . . n]. Il caso base occorre quando i = j, ovvero c'è un solo elemento, che è quello da restituire. Nel caso generale, si considera l'elemento centrale m del sottovettore considerato se dispari, o l'elemento precedente se pari.
- Se l'elemento m ed m + 1 sono uguali, significa che prima di m si trovano tutte coppie e quindi il valore cercato si trova a destra, dalla posizione m + 2 in poi. Se sono diversi, si trova a sinistra, entro la posizione m.
- E' possibile verificare che questo approccio funziona correttamente anche con tre elementi.
- Volendo fare una dimostrazione di correttezza più approfondita, l'invariante è che il sottovettore che viene passato in input all'algoritmo inizia in una posizione dispari, ha dimensione dispari, e contiene l'elemento singolo. All'inizio, questo è vero perché i = 1, n dispari, e per assunzione contiene l'elemento singolo. Ad ogni passo, le tre proprietà vengono rispettate; al termine, essendoci un solo elemento, questo è l'elemento singolo.

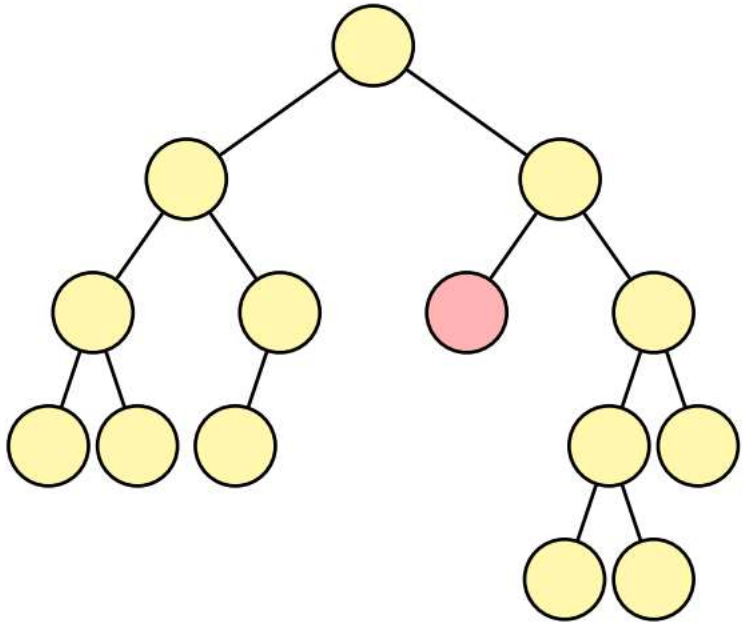
Find Single

```
findSingle(int[] A, int n)
    return fsRec(A, 1, n)
```

```
fsRec(int[] A, int n)
    if i == j then
        return A[i]
    else
        int m = ⌊(i + j)/2⌋
        if m mod 2 == 0 then
            m = m − 1
        if A[m] == A[m + 1] then
            return fsRec(A, m + 2, j)
        else
            return fsRec(A, i, m)
```

Profondità minima

Scrivere un algoritmo che prenda in input un albero binario T e restituisca la sua profondità minima, ovvero la minima profondità fra tutte le profondità delle foglie. Spiegare il funzionamento e discutere la complessità dell'algoritmo proposto. Ad esempio, nell'albero sottostante, la profondità minima è pari a 2, la profondità del nodo rosa



Profondità minima

- L’algoritmo proposto è una semplice postvisita. L’idea è di calcolare, per ogni nodo t, la distanza minima fra t e un nodo foglia nel sottoalbero radicato in t.
- Per questo motivo, se il nodo è una foglia, restituiamo 0.
- Altrimenti, restituiamo la distanza minima fra i figli sinistro/destro e le foglie radicate nei rispettivi sottoalberi, a cui aggiungiamo 1 per contare t.
- Per gestire la possibilità che un nodo abbia un solo figlio, utilizziamo il valore $+\infty$ per i nodi nil, che così non verrà mai selezionato da min().
- Il valore restituito nella radice è il valore che cerchiamo. La complessità dell’algoritmo è pari a quella di una visita di un albero, ovvero $\Theta(n)$.

Profondità minima

```
int minDepth(TREE T)
    if T == nil then
        return +∞
    else if T.left == T.right == nil then
        return 0
    else
        return 1 + min(minDepthRec(T.left), minDepthRec(T.right));
```

Questo algoritmo visita tutti i nodi dell'albero.

È possibile migliorare l'algoritmo scrivendo una visita in ampiezza che si ferma alla prima foglia. In alcuni casi, questo può comportare un notevole miglioramento, a scapito di un maggior uso della memoria. La complessità nel caso pessimo, tuttavia, non cambia: in 1 un albero perfetto, la profondità minima è pari all'altezza dell'albero, quindi bisogna comunque visitare metà dell'albero prima di fermarsi, con costo O(n).

Profondità minima

```
int minDepth(TREE t)
    int level = 0
    QUEUE Q = Queue()
    Q.enqueue(t)
    Q.enqueue(nil)
    while not Q.isEmpty() do
        TREE u = Q.dequeue()
        if u == nil then
            level = level + 1
            Q.enqueue(nil)
        else
            if u.left == u.right == nil then
                return level
            else
                if u.left ≠ nil then
                    Q.enqueue(u.left)
                if u.right ≠ nil then
                    Q.enqueue(u.right)
```