# Laboratorio di algoritmi e strutture dati

Docente: Violetta Lonati

Prova di laboratorio svolta - Appello del 16 giugno 2021 (in presenza)

## 1 Più piccolo - Svolgimento

La proprietà è vera quando il massimo elemento di A è strettamente più piccolo del minimo elemento di B. Determinare il massimo di A richiede costo $O(n)$, determinare il minimo di B richiede costo $O(m)$, quindi si ottiene un algoritmo di costo complessivo $O(n+m)$:

```
int h( int A[], int B[], int n, int m) {
        // calcolo il massimo di A
        int maxA = A[0];
        for ( int i = 1; i < n; i++ )
                if ( A[i] > maxA )
                        maxA = A[i];

        // calcolo il minimo di B
        int minB = B[0];
        for ( int j = 1; j < m; j++ )
                if ( B[j] < minB )
                        minB = B[j];

        return maxA < minB;
}
```

Una soluzione alternativa, con lo stesso costo asintotico, si ottiene determinando il massimo di A e confrontandolo poi con tutti gli elementi di B. In questo caso si può interrompere il ciclo di confronto non appena si trova un elemento di B che non è più grande di `maxA`. Nel caso peggiore il costo è sempre $O(n+m)$, ma nei casi in cui $m$ è molto più grande di $n$ e il valore minimo di B si trova nelle prime posizioni di B, il costo si riduce a $O(n)$.

```
int h( int A[], int B[], int n, int m) {
        // calcolo il massimo di A
        int maxA = A[0];
        for ( int i = 1; i < n; i++ )
                if ( A[i] > maxA )
                        maxA = A[i];

        // confronto maxA con ogni elemento di B
        // (devono essere tutti strettamente maggiori)
        for ( int j = 0; j < m; j++ )
                if ( B[j] <= maxA )
                        return 0;
```

```
        return 1;
}
```

## Funzioni misteriose - Svolgimento

1. **Domanda:** Cosa stampa la funzione `f` se viene invocata sulla radice dell'albero di sinistra?
   **Risposta:**

   ```
   1 39
   1 15 79
   ```

2. **Domanda:** Cosa stampa la funzione `f` se viene invocata sulla radice dell'albero di destra?
   **Risposta:**

   ```
   1 39
   1 15 7 9
   1 15 7 7 13
   1 15 79
   ```

3. **Domanda:** Che altezza raggiunge lo stack se la funzione `f` viene invocata sulla radice dell'albero di destra? (si considerino solo le chiamate di funzione effettuate durante l'esecuzione di `f`).
   **Risposta:** Lo stack raggiunge la sua altezza massima quando viene visitata la foglia 13. Questo avviene alla quinta chiamata ricorsiva di `f_r`. Durante questa chiamata viene inoltre invocata la funzione `printArray`, quindi lo stack (considerando le chiamate fatte da `f`) raggiunge altezza 6.
4. **Domanda:** In generale, che altezza raggiunge lo stack se la funzione viene invocata sulla radice di un albero binario qualunque?
   **Risposta:** In generale, l'altezza massima raggiunta dallo stack è pari all'altezza dell'albero +2. Nota: l'altezza dell'albero è definita come la lunghezza del cammino più lungo dalla radice ad una foglia. La lunghezza di un cammino è pari al numero di archi che compongono il cammino (quindi pari al numero di nodi toccati 11). Quindi ad esempio l'albero di destra ha altezza 4.
5. **Domanda:** In generale, quante righe stampa la funzione se invocata sulla radice di un albero binario qualunque?
   **Risposta:** La funzione stampa tante righe quante sono le foglie dell'albero.
6. **Domanda:** Cosa stampa la funzione se invocata sulla radice di un qualunque albero binario che contiene solo numeri pari?
   **Risposta:** La funzione stampa n righe vuote, dove n è il numero delle foglie dell'albero.
7. **Domanda:** Completate la frase seguente:

   > Se `root` è il puntatore alla radice di un albero binario, allora l'invocazione della funzione `f(root)` produce in output ...

   **Risposta:** ...tante righe quante sono le foglie dell'albero: per ogni foglia è stampata una riga contenente la sequenza dei nodi con chiave dispari che si incontrano sul cammino dalla radice a quella foglia. Le foglie sono considerate da sinistra verso destra.
   NB: se su un cammino dalla radice ad una foglia ci sono solo nodi di chiave pari, verrà stampata una riga vuota.

## 2 Dipendenti - Svolgimento

**Modellazione e progettazione** Assumendo che gli $N$ dipendenti di Algoré siano indicati con un numero progressivo da 0 a $N-1$, svolgete i seguenti punti.

1. **Domanda:** Modellate la situazione con una struttura dati opportuna:

   **Risposta:** La situazione può essere modellata da una foresta di alberi.

   - **Domanda:** descrivete come si possono rappresentare i dipendenti e le loro relazioni con la struttura dati scelta;

     **Risposta:** Ciascun nodo rappresenta un dipendente e la sua chiave è il numero progressivo che identifica il dipendente; un nodo A è padre di un nodo B se il dipendente rappresentato da A è supervisore del dipendente rappresentato da B.

     Sappiamo che si tratta di alberi poiché ogni dipendente ha al massimo un supervisore. Il numero dei dipendenti subordinati non è limitato, quindi ogni nodo può avere un numero arbitrario di figli.

     Le radici degli alberi nella foresta rappresentano dipendenti di massimo livello; le foglie rappresentano dipendenti che non hanno alcun subordinato.

   - **Domanda:** riformulate, nei termini della struttura dati scelta, ciascuno dei compiti enunciati sopra.

     (a) Dato un certo dipendente, stampare l'elenco dei suoi subordinati.

     **Risposta:** Dato un numero $n$, stampare i figli del nodo con chiave $n$.

     (b) Contare quanti sono i dipendenti che non hanno alcun subordinato.

     **Risposta:** Contare le foglie.

     (c) Dato un certo dipendente, individuare chi è il suo supervisore.

     **Risposta:** Dato un numero $n$, individuare il padre del nodo con chiave $n$.

     (d) Dato un certo dipendente, stampare la lista dei dipendenti che si trovano sopra di lui gerarchicamente, partendo dal suo supervisore e risalendo la gerarchia fino a un dipendente di massimo livello.

     **Risposta:** Dato un numero $n$, stampare la lista dei nodi che si trovano nel cammino dal nodo con chiave $n$ (escluso) alla radice dell'albero di cui tale nodo fa parte.

     (e) Stampare l'elenco di tutti i dipendenti –non importa l'ordine–, indicando per ciascuno chi è il suo supervisore (tranne che nel caso di dipendenti di massimo livello).

     **Risposta:** Stampare tutte le chiavi dei nodi della foresta, indicando per ciascuno la chiave del nodo padre (tranne che per le radici).

     (f) Stampare l'elenco di tutti i dipendenti, in ordine di livello (prima tutti quelli di livello massimo, poi tutti quelli subordinati a quelli di livello massimo, ecc); non importa l'ordine tra i dipendenti di pari livello.

     **Risposta:** Stampare le chiavi dei nodi della foresta procedendo livello per livello: prima le radici, poi i nodi di profondità 1, poi quelli di profondità 2, e così via.

2. **Domanda:** Descrivete come è opportuno implementare la struttura dati scelta.

   **Risposta:** Una foresta di alberi può essere pensata come un tipo particolare di grafo orientato (gli archi si intendono orientati da padre a figlio), in cui ogni nodo ha al più un arco entrante (la sorgente di tale arco è il padre del nodo stesso, quando non si tratta di radice).

   Si può quindi scegliere di partire da una implementazione tipica dei grafi. Poiché gli alberi non sono densi (il numero di archi è pari al numero di nodi meno uno), l'implementazione con matrici di adiacenza non è indicata perché consuma troppo spazio inutilmente. E' invece preferibile un'implementazione con liste liste di adiacenza.

   Dal momento che i nodi sono identificati da chiavi intere da 0 a N-1, e che non ci sono compiti che prevedono l'aggiunta o la rimozione di nodi, ha senso implementare il grafo (la foresta di alberi) come

un array di liste di adiacenza; non ci sarà così bisogno di definire strutture ulteriori per i nodi degli alberi. La lista di adiacenza di un nodo di fatto contiene i figli di quel nodo, per questo motivo possiamo chiamare l'array delle liste di adiacenza `listaFigli`. Dunque, l'elemento `listaFigli[i]` indica la testa della lista di adiacenza (lista dei figli) del nodo con chiave `i`.

Poiché nei compiti (c), (d) e (e) è importante *risalire* da un nodo al padre, è utile mantenere un collegamento anche ai padri; questo si può fare con un ulteriore vettore `padre` di interi, in cui l'elemento `padre[i]` indica l'indice del padre del nodo con chiave *i*. Per le radici, indichiamo come padre il valore -1.

E' infine utile avere l'elenco delle radici per effettuare visite di alberi, ad esempio per il compito (f). Questo può essere fatto tramite un vettore `radici` di interi.

3. **Domanda:** Per ciascun compito, progettate e descrivete un algoritmo che consente di svolgere il compito, sfruttando le scelte di progettazione e implementazione fatte precedentemente. Gli algoritmi possono essere descritti a parole o in pseudocodice; può essere opportuno fare riferimento ad algoritmi noti.

   **Risposta:**

   a) Si scorre la lista di adiacenza `listaFigli[n]` del nodo con chiave `n`, stampandone gli elementi.

   b) Si scorre il vettore `listaFigli` delle liste dei figli contando quelle vuote (puntatore `NULL`).

   c) `padre[i]` è la chiave del nodo padre di `i`.

   d) Si può risalire verso l'alto partendo dal nodo `padre[i]`; basta un ciclo che si ferma quando si raggiunge la radice:

```
stampaImpiegatiSopra ( nodo i ) {
        i = padre[i];
        while( i != -1 ) {
                print( i )
        }
}
```

   e) Dato che non ha importanza l'ordine, basta scorrere il vettore `padre` stampando per ogni chiave `i` il supervisore `padre[i]`, purché sia diverso da -1.

   f) Se ci fosse un solo albero sarebbe sufficiente fare una visita in ampiezza del grafo. Poiché in generale c'è più di un albero, è necessario considerarli tutti assieme.

      Si possono costruire delle liste di nodi, una lista per ciascun livello. Per riempire le liste, si devono visitare i vari alberi uno alla volta, con una visita in ampiezza. Per implementare la visita in ampiezza è necessario usare una coda.

#### Nota bene - altre implementazioni

La stessa struttura dati (foresta di alberi) potrebbe essere implementata anche in altri modi. Oltre alla matrice di adiacenza già citata, si potrebbe partire dall'idea dell'implementazione degli alberi binari tramite strutture con due puntatori ai figli sinistro e destro, aggiungendo un puntatore al padre. Questa andrebbe però adattata in quanto il numero dei figli, qui, può essere superiore a 2. Inoltre, non si ha un solo albero ma si deve gestire un insieme di alberi, quindi non basta una sola variabile `root` per indicare la radice dell'albero, ma serve avere una qualche struttura dati che raccolga l'insieme degli alberi. A questo scopo si può usare –come sopra– una lista o un vettore di radici. Un'altra possibilità è quella di consderare un nodo *fittizio* aggiuntivo, da usare come radice e a cui collegare come figli le radici dei vari alberi (tali radici *vere* rappresentano i dipendenti di massimo livello).

Il difetto principale di questa soluzione è che non supporta la ricerca dei nodi a partire dal numero identificativo: per cercare il nodo che rappresenta il dipendente `n`, è necessario visitare tutta la foresta con costo, nel caso peggiore, lineare nel numero di nodi nella foresta. Nell'implementazione con gli array `padre` e `listaFigli`, invece, usando il numero identificativo come indice si aveva l'accesso diretto in tempo costante al nodo corrispondente.

Inoltre, scegliendo questa implementazione alternativa, alcuni compiti potrebbero rivelarsi più scomodi da implementare o più onerosi. Questo si deve in particolare al fatto che questa implementazione costringe a visitare l'albero in base alla sua struttura, cioè muovendosi solo di padre in figlio (o di figlio in padre). Nella implementazione descritta prima, invece, *scorrendo* i vettori si possono esaminare i nodi in base al loro numero e non in base alla loro posizione nell'albero.

Ad esempio, il conteggio del numero di foglie con l'implementazione "ad albero" richiede necessariamente di visitare ciascun albero partendo dalla radice con un algoritmo di questo tipo: se l'albero è fatto da un solo nodo, si restituisce 1, altrimenti si restituisce la somma del numero di foglie dei sottoalberi della radice.

```
int contaFoglieInAlbero ( nodo p ) {
        if p == NULL
                return 0;
        if p -> listaFigli == NULL // p è una foglia
                return 1

        int sum = 0
                foreach q in p -> listaFigli
                        sum += contaFoglieInAlbero(q)
        return sum
}
```

Se si usa una radice fittizia e la si passa alla funzione, si ottiene il numero di foglie in tutta la foresta.

4. **Domanda:** Spiegate come modifichereste le risposte ai punti procedenti se i dipendenti fossero identificati da un nome (dunque da una stringa) invece che un numero progressivo.

   **Risposta:** Le stringhe non possono essere usate come indici per i vettori `listaFigli` e `padre`. Per ogni nodo ci sarà dunque bisogno di una struttura con il nome, un puntatore alla lista di adiacenza e un puntatore al padre; le liste di adiacenza non saranno semplici liste di interi ma liste di puntatori a nodi; in fase di creazione dei nodi bisogna tenere traccia dell'ultimo intero usato. Servirà una lista dei nodi radice (o una radice fittizia). Con questa implementazione si ha il problema della ricerca dei nodi (non si può più accedere direttamente alla posizione utile dei vettori usando l'indice). Inoltre i nodi possono essere visitati solo seguendo i collegamenti padre-figlio.

   Un'alternativa a questo approccio è quella di associare comunque ad ogni dipendente un numero progressivo, e usare questi numeri per indicizzare il vettore dei padri e quello delle liste di adiacenza come nella implementazione scelta all'inizio. Bisognerà definire un metodo per assegnare i numeri progressivamente e tenere traccia della corrispondenza nodi-numeri (in entrambe le direzioni). Ad esempio si può usare aggiungere alla struttura dei nodi un membro che indica il suo numero progressivo, e un vettore che consenta di individuare, dato un intero `i`, il nodo corrispondente `nodo[i]`.

**Implementazione** Assumendo ancora che gli $N$ dipendenti di Algoré siano indicati con un numero progressivo da 0 a $N-1$. definite, in linguaggio C, uno o più tipi di dati utili a rappresentare i dipendenti e le loro relazioni, in base alle scelte fatte ai punti precedenti. Scrivete quindi una funzione C per ciascuno dei compiti enunciati sopra.

```
struct f{
```

```
   int num_trees; // numero di alberi
   int num; // numero di nodi
   struct listnode **listaFigli; // array (allocato dinamicamente) di liste di figli
   int *padre; // array di interi
   int *radici; // array di interi
};

typedef struct f *foresta;

void stampaSubordinati( foresta f, int i ) {
   struct listnode *p;
   for ( p = f -> listaFigli[i]; p != NULL; p = p -> next)
      printf( "%d ", p -> key);
}

// i nodi sono visitati in ordine di chiave e non con una visita dell'albero
int quantiSenzaSubordinati( foresta f ) {
   int count = 0;
   for (int i = 0; i < f->num; i++) {
      if ( f -> listaFigli[i] == NULL )
         count++;
   }
   return count;
}

int supervisore( foresta f, int i ){
   return f -> padre[i];
}

void stampaImpiegatiSopra( foresta f, int i) {
   i = f -> padre[i];
   while( i != -1 ) {
      printf( "%d ", i );
      i = f -> padre[i];
   }
   printf( "\n");
}


void stampaImpiegatiConSupervisore ( foresta f ) {
   for ( int i = 0; i < f->num; i++ ) {
      printf( "%d", i );
      if ( f -> padre[i] != -1 )
         printf( ":%d", f -> padre[i] );
      printf( " " );
   }
   printf( "\n" );
}
```