Laboratorio di algoritmi e strutture dati

Docente: Violetta Lonati

Prova di laboratorio svolta - Appello del 20 febbraio 2020 - Versione B

1 Musica - svolgimento e commenti

Si può scrivere un algoritmo lineare nella lunghezza del vettore. Questa è un possibile svolgimento:

```
int maxLungNonCrescente(int *arr, int n){
        int max = 1;
        int t = 1;

        for (int i = 1; i < n; i++) {
                if (arr[i] > arr[i - 1]) {
                        if(t > max)
                                max = t;
                        t = 1;
                }
                else {
                        t++;
                }
        }

        if (t > max)
                return t;
        else
                return max;
}
```

Si usa una variabile `max` per memorizzare la lunghezza massima tra quelle dei sottovettori che soddisfano la condizione richiesta. Si usa un ciclo per scorrere il vettore. Per tenere traccia della lunghezza della sottosequenza corrente si usa un contatore `t`, dichiarato e inizializzato a 1 fuori dal ciclo. Ad ogni iterata si confronta il valore corrente (indice `i`) con il precedente (indice `i-1`) e si verifica se il valore corrente è superiore (strettamente) al precedente, per individuare quando finisce una sottosequenza non crescente. Al termine di una sottosequenza, se `t` risulta maggiore di `max` allora si assegna a `max` il valore di `t`, poi si ri-inizializza `t` a 1. Se invece la sottosequenza non è terminata, si incrementa il contatore `t`. Al termine del ciclo, per contemplare il caso in cui la sottosequenza più lunga si trovi alla fine del vettore, si controlla un'ultima volta se `t` (cioé la lunghezza dell'ultima sottosequenza) è superiore a `max`.

Errori tipici Molti svolgimenti erano impostati più o meno come sopra, ma presentavano degli errori. Tra quelli più ricorrente segnalo i seguenti:

- non considerare il caso di valori consecutivi uguali (quindi comunque non crescenti), usando l'operatore di confronto >= invece di >;
- non contemplare il caso in cui la sequenza più lunga si trova alla fine del vettore;

- inizializzare il contatore `max` a 0 invece che 1;
- dimenticare di re-inizializzare il contatore `t` alla fine di ogni sottosequenza.

In altri casi sono stati implementati algoritmi che salvano dati intermedi; ad esempio memorizzando nella prima scansione le lunghezze delle sottosequenze non decrescenti, e calcolando solo alla fine il massimo riesaminando tutte le lunghezze precedentemente memorizzate. Pur mantenendo una complessità lineare, lo spazio occupato è inutilmente maggiore.

2 Lista misteriosa - svolgimento e commenti

Comprensione del codice.

1. La funzione `crea` costruisce una lista i cui elementi sono i suffissi della stringa `w` ricevuta come argomento: il primo elemento della lista è il suffisso di lunghezza 1 (formato cioè solo dall'ultima lettera di `w`), il secondo ha lunghezza 2 e cosi via fino all'ultimo che coincide con la parola intera.

   La costruzione della lista avviene in questo modo: prima viene creata una lista con un solo elemento, e cioè l'intera parola (ovvero quello che alla fine risulterà essere l'ultimo della lista), quindi il suo indirizzo viene passato alla funzione ricorsiva `rfun`, che aggiunge ricorsivamente gli altri elementi in testa alla lista. Il caso base è quello in cui l'elemento in testa alla lista è formato da una sola lettera; in questo caso l'indirizzo della testa viene restituito alla funzione `crea`, che a sua volta lo restituisce alla sua chiamante. Nel caso ricorsivo, invece, viene inserito in testa alla lista un nuovo elemento, dato dal suffisso che si ottiene da quello attualmente in testa togliendogli la prima lettera, e si invoca `rfun` sulla nuova testa.

   In particolare

   a) La funzione `crea` restituisce l'indirizzo del primo elemento della lista dei suffissi.
   b) La funzione `rfun` viene invocata un numero di volte pari alla lunghezza della stringa `w`.
   c) Nell'`if` di riga 2 si verifica se la stringa `l -> w` è formata da un solo carattere.
   d) L'assegnamento nella riga 5 serve a memorizzare, nel membro `w` del nodo `n`, l'indirizzo del secondo carattere della stringa contenuta nel nodo `l`.

2. La funzione `cfun` concatena a rovescio la stringa `w` con tutte le parole contenuti nella lista che inizia col nodo `l`. Ad esempio, se `w` è la stringa `zero` e la lista `l` contiene nell'ordine le stringhe `uno` e `due`, allora il risultato di `cfun` sarà la stringa `dueunozero`. Il primo argomento di `cfun` deve essere un puntatore che punta a una porzione di memoria opportunamente allocata per contenere la stringa risultante.

   Invocata passando come argomenti la parola vuota e la lista costruita con `build`, la funzione `cfun` costruisce la parola ottenuta concatenando i suffissi di `w`, a partire dal quello più lungo. In questo caso quindi, detta n la lunghezza di `w`, il puntatore deve puntare ad uno spazio di memoria precedentemente allocato di $n(n+1)/2+1$ caratteri (la somma di tutte le lunghezze dei suffissi di `w`, ovvero la somma di tutti gli interi da 1 a n, più il carattere di fine-stringa).

Scrittura di codice.

1. Il tipo `Node` va definito come un puntatore a una struttura con un membro chiamato `w` (di tipo puntatore a carattere) e un membro chiamato `next` (di tipo puntatore al tipo strutturato in via di definizione). La definizione del tipo strutturato è ricorsiva, dunque è necessario usare un *tag* per la struttura, che va poi combinato con l'uso `typedef`. Questo si può fare in due modi:

```
typedef struct node {
        char *w;
        struct node *next;
} *Node;
```

oppure, equivalentemente:

```
struct node {
        char *w;
        struct node *next;
}

typedef struct node *Node;
```

2. Una possibile versione iterativa di `cfun` è la seguente. Il ciclo esterno serve per scorrere la lista e concatenare, ad ogni iterata, la parola nel nodo corrente. Poiché la concatenazione deve avvenire al rovescio (cioè, la prima parola della lista deve stare in fondo alla stringa costruita), ad ogni iterata devo spostare in avanti la stringa già memorizzata in `w` per liberare lo spazio necessario per la parola nel nodo corrente. Il primo ciclo for serve proprio a questo: `nw` è la lunghezza della stringa attualmente già memorizzata, mentre `ncurr` è la lunghezza della parola nel nodo corrente, che va concatenata in testa. Si noti che se `ncurr` è inferiore a `nw`, nel copiare in avanti la stringa c'è il rischio di una sovrapposizione, quindi è necessario partire dall'ultima posizione (terminatore di stringa incluso) tornando indietro fino alla posizione 0.

```
char *cfun_iter( char *w, Node l ){
        while ( l != NULL ) {
                int ncurr = strlen( l -> w );
                int nw = strlen( w );
                for ( int i = nw; i >= 0; i-- ) {
                        w[ ncurr + i ] = w[i];
                }
                for ( int i = 0; i < ncurr; i++ ) {
                        w[i] = l -> w[i];
                }
                l = l -> next;

        }
        return w;
}
```

Invece di spostare ad ogni iterata la parola memorizzata in `w`, si potrebbe calcolare preventivamente la somma delle lunghezze delle parole memorizzate nella lista e copiare le varie parole una alla volta, carattere per carattere, nella posizione giusta, a partire dal fondo.

Un altro modo di affrontare il problema è quello di costruire la lista inversa (scorrendo la lista e inserendo ogni nodo in testa alla nuova lista di via di costruzione) e poi scorrere la nuova lista per concatenare i nodi nell'ordine voluto. Importante ricordarsi di spostare in avanti la stringa `w` nel caso in cui questa non sia inizialmente vuota.

3. Non c'è bisogno di modificare `crea` ma soltanto `rfun`. Invece di costruire la lista dei suffissi si deve costruire la lista dei prefissi; quindi non basta memorizzare l'indirizzo del primo carattere da considerare ma è necessario copiare il prefisso fino alla posizione giusta, dopo aver allocato lo spazio necessario. Costruendo la lista ricorsivamente, si parte da quello che sarà l'ultimo elemento della lista, e cioè il prefisso più lungo. La ricorsione termina quando si arriva al prefisso di lunghezza 1.

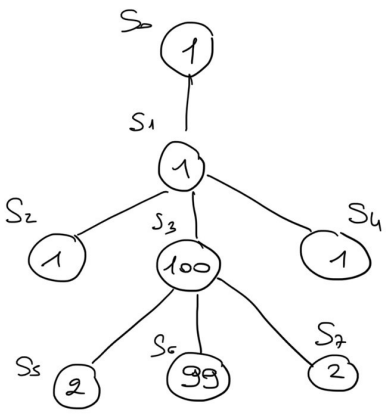
Ecco una possibile implementazione:

```
Node rfun2( Node l ) {
        if ( l -> w[0] != '\0' && l -> w[1] == '\0' )
```

```
            return l;
        Node n = malloc( sizeof(struct node) );
        int len =  strlen( l -> w );
        n -> w = malloc( len + 1);
        strcpy( n -> w, l -> w );
        n -> w[ strlen(l -> w) - 1 ] = '\0';
        n -> next = l;
        return rfun2( n );
}
```

3 Videosorveglianza - Svolgimento e commenti

1. In prima istanza si può pensare di modellare la situzione con un *grafo*, in cui i nodi rappresentano le stanze e gli archi i corridoi: tra due nodi c'è un arco se e solo se le due stanze sono attigue, cioé collegate da un corridoio. Il testo specifica però che "una qualunque stanza è collegata con la stanza di ingresso da un un unico percorso" e questo significa che il grafo che rappresenta un deposito è in realtà un *albero*, la cui radice rappresenta la stanza di ingresso. Il deposito dell'esempio è raffigurato più esplicitamente come albero nella figura seguente:



   Un impianto è dunque un sottoinsieme S di nodi dell'albero (i nodi in S rappresentano le stanze in cui vi è una videocamera): per essere definitio impianto, tale sottoinsieme S deve soddisfare due proprietà: i) per ogni nodo v che non sta in S, o il padre di v sta in S o ci sta almeno uno dei suoi figli (ogni stanza che non è sorvegliata direttamente deve essere sorvegliata indirettamente); ii) se due nodi sono uno figlio dell'altro, allora non possono stare entrambi in S (altrimenti sarebbero entrambe sorvegliate direttamente); I corridoi bui sono archi che collegano due nodi entrambi al di fuori di S.

2. Dato un insieme di stanze S, l'algoritmo deve controllare le due condizioni elencate sopra. Si può visitare l'albero in preordine e, per ogni nodo visitato v:
   - se $v \notin s$ si deve verificare se il padre o uno dei figli lo è; se si verifica questo caso S non è un impianto (è violata la proprietà i);
   - se $v \in S$ si deve verificare che nessuno dei figli è in S, altrimenti S non è un impianto (è violata la proprietà ii).

3. Una semplice algoritmo per costruire un impianto è la seguente: si posiziona una telecamera nella radice, si "saltano" i figli, si mette una telecamera in ciascun nipote, e così via, ovvero si mettono le telecamere solo nei livelli dispari dell'albero. Operativamente si può procedere con una visita ricorsiva

in cui, tra gli argomenti delle chiamate ricorsive, si passa anche l'informazione relativa alla distanza del nodo dalla radice oppure sulla parità del livello in cui si trova il nodo.

L'algoritmo non produce in generale un impianto ottimale. Questo succede ad esempio nel caso del deposito in figura, in cui si otterrebbe l'impianto formato dai nodi S_1, S_2, S_3, S_4, che non è ottimale.

4. Questo problema è risolubile efficientemente ricorrendo alla programmazione dinamica.

   Consideriamo un qualunque nodo x e immaginiamo di conoscere il valore di tutti gli impianti ottimali dei sottoalberi di x. Come possiamo calcolare il valore ottimale dell'impianto per l'albero con radice x? Ci sono due possibilità: o x fa parte dell'impianto ottimale, oppure no. Nel secondo caso, l'impianto ottimale per l'albero con radice in x si ottiene unendo gli impianti ottimali per i sottoalberi con radice nei figli di x. Nel primo caso, invece, l'impianto ottimale per l'albero con radice in x è formato da x e dagli impianti ottimali per gli alberi aventi per radice tutti i nipoti di x. Quindi, indicando con $I(x)$ il valore dell'impianto ottimale peril sottoalbero radicato in x, possiamo scrivere questa relazione di ricorrenza:

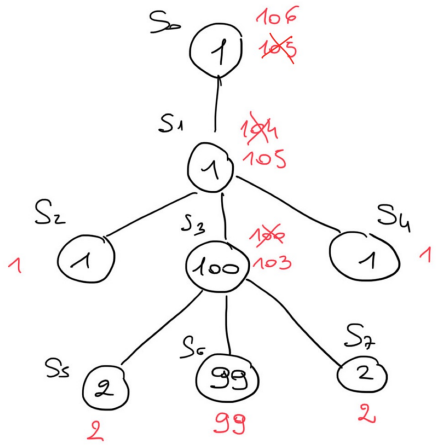
$$I(x) = max\left(\sum_{y \text{ figlio di } x} I(y), v(x) + \sum_{z \text{ nipote di } x} I(z),\right)$$

   Il caso base si ha sulle foglie: in questi casi il valore dell'impianto ottimale è dato dal valore della foglia stessa.

   I valori $I(x)$ possono essere calcolati e memorizzati in ciascun nodo partendo dalle foglie e risalendo, quindi usando una visita in ordine posticipato.

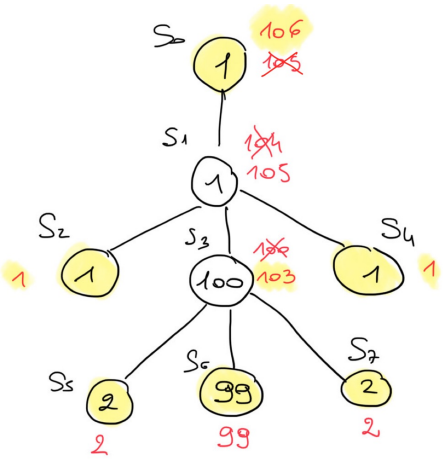
   Notate che con questo algoritmo stiamo calcolando i valori massimi ma non stiamo ancora posizionando le telecamere (come faremmo se seguissimo un approccio *greedy*) poiché la valutazione se mettere o meno una telecamera in un nodo x non viene fatta quando si calcola il valore $I(x)$ poiché dipende anche dai valori calcolati poi sul padre e sul nonno.

   Nella figura seguente è mostrato per ogni nodo x il valore $I(x)$ che si ottiene con questo algoritmo; per i nodi che non sono foglie sono indicati due numeri, corrispondenti ai due casi in cui si mette la telecamera o meno; quello barrato è il minore dei due, $I(x)$ è il rimanente.



   Una volta conclusa la visita, e dunque memorizzati i valori $I(x)$ di ogni nodo, possiamo stabilire dove posizionare le telecamere ripartendo dalla radice e scendendo, quindi con una visita in ordine anticipato. Si mette una telecamera nella radice se e solo se $I(x)$ è maggiore di $\sum_{y \text{ figlio di } x} I(y)$. Poi, si prosegue scendendo; per i nodi i cui padri hanno una telecamera non si fa nulla, per gli altri si stabilisce se mettere una telecamera oppure no usando la stessa condizione usata per la radice.

   Nella figura qui sotto sono indicati i nodi che vengono selezionati con questo algoritmo, nel caso del deposito dell'esempio.


S0
1
106
105
S1
1
104
105
S2
1
1
S3
100
102
103
S4
1
1
S5
2
2
S6
99
99
S7
2
2