Tecniche di Programmazione Funzionale e Imperativa

Ivano Salvo

Strutture Dati Generiche
Alcune "perle" di programmazione C

Corso di Laurea in **Informatica**, III anno

SAPIENZA
UNIVERSITÀ DI ROMA

Lezione **25**, 27 maggio 2022

Lezione 25a:

Alberi binari

Alberi (binari e non)

Abbiamo visto in Haskell vari tipi di albero (binario e non) polimorfo. Ricordiamo brevemente alcune definizioni:

```
-- alberi con alberi vuoti:
data BinTree a = Empty |
            Node a (BinTree a)(BinTree a)

-- a partire dalle foglie, con etichette
-- solo sulle foglie:
data BinTree’ a = Leaf a |
            Node (BinTree a) (BinTree a)

-- alberi non necessariamente binari:
data Tree a = Fork [Tree a]
```

Ma in **C**?

Alberi: rappresentazione in C

Come per le liste vedremo prevalentemente una **rappresentazione a puntatori** (saprete che ci sono altre rappresentazioni)

Un albero non vuoto $r(L, R)$ contiene sempre l'elemento r (**radice** dell'albero) attaccato ai sotto-alberi L ed R (**sottoalbero destro** e **sinistro**).

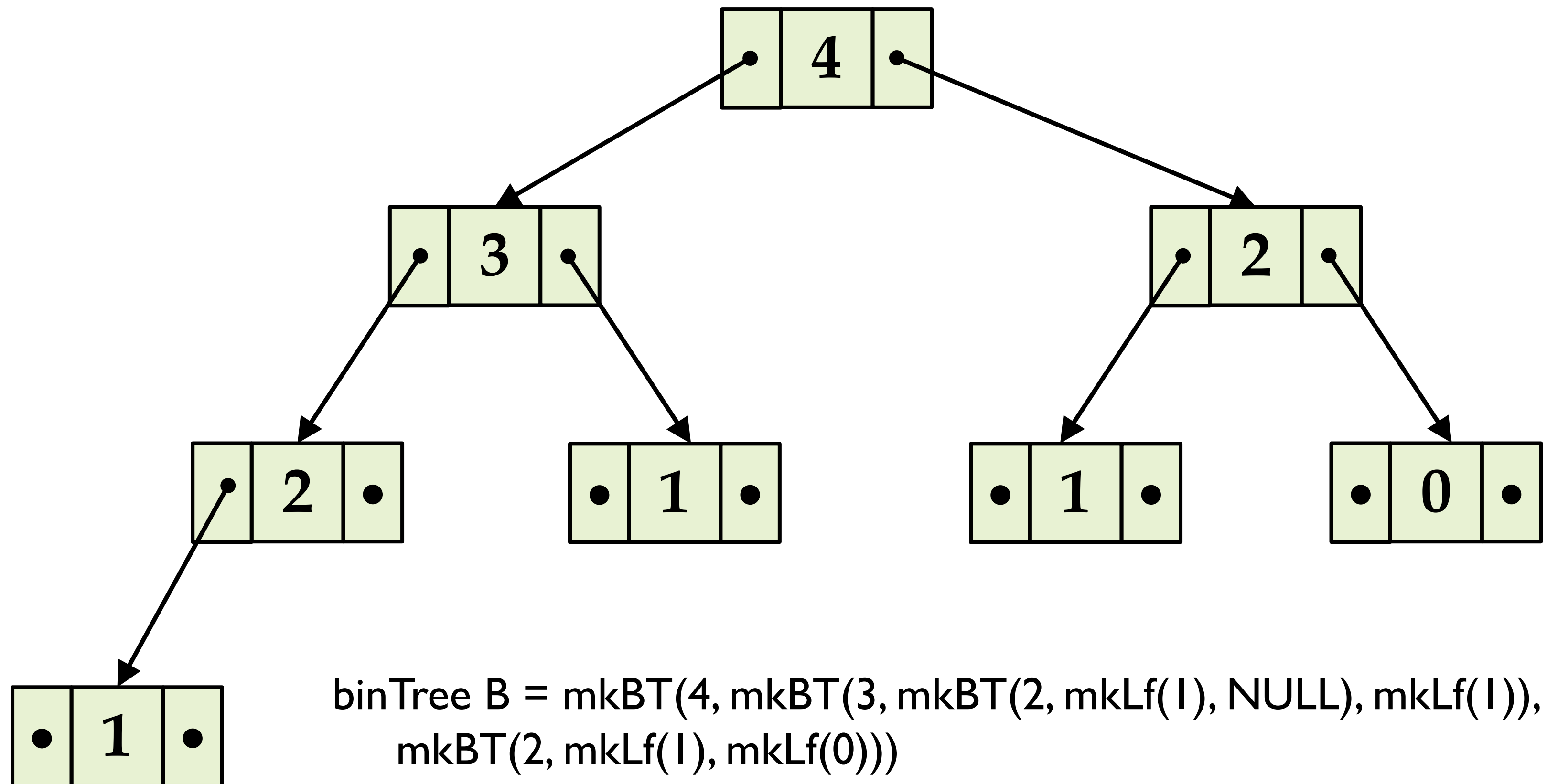
Come nel caso delle liste, useremo una struct per memorizzare un nodo dell'albero che consiste 1) di un campo **informazione** e 2) due campi puntatore ai **sottoalberi** (e anche qui i puntatori vengono usati per le definizioni ricorsive di tipo).

```
typedef
struct B {
   struct B * left;
   int info;
   struct B * right;
} binTreeNode;

typedef binTreeNode* binTree;
```

Alberi: rappresentazione in memoria

Con le definizioni date, gli alberi in memoria sono rappresentati come tante `perline' tenute insieme dai puntatori. Ecco l'albero descritto prima in memoria (dove • rappresenta il pointer NULL)



Detour: equivalenza di tipi

Forse qualcuno si è accorto che **da un punto di vista strutturale**, gli **alberi** sono del tutto **uguali alle liste doppiamente concatenate**: ogni nodo contiene un campo informazione e due campi puntatore.

Tuttavia è molto diverso come i campi puntatore vengono usati, a riprova che anche le **operazioni determinano un tipo**!

I linguaggi di programmazione hanno due forme di equivalenza tra tipi:

- **Equivalenza strutturale**: due tipi sono equivalenti se hanno la stessa struttura (ad esempio alberi e liste doppiamente concatenate).
- **Equivalenza per nome**: due tipi sono equivalenti se hanno lo stesso nome.

L'equivalenza per nome sottintende che se il programmatore definisce due tipi con la stessa struttura ma nomi diversi si protegge da sé stesso a usare in modo incoerente i tipi.

Alberi: funzioni costruttori in C

Anche nel caso degli alberi, useremo il puntatore NULL per rappresentare l'**albero vuoto #**.

Il costruttore canonico degli alberi di interi sarà una funzione binTree makeBT(int, binTree, binTree). Lo costruiamo in due passi, definendo anche un costruttore binTree makeLeaf(int).

```
binTree makeLeaf(int r){
   binTree B;
   B = (binTree)malloc(sizeof(binTreeNode));
   B->info = r;
   B->left = NULL;
   B->right= NULL;
   return B;
}
```

```
binTree makeTree(int r, binTree L,
                                 binTree R){
   binTree B = makeLeaf(r);
   B->left = L;
   B->right= R;
   return B;
}
```

Alberi: distruttori in C

È sufficiente un **unico distruttore**:

```
int isEmptyBT(binTree B, int* r,
                binTree *L, binTree *R){
    if (!B) return 1;
    *r = B->info;
    *L = B->left;
    *R = B->right;
    return 0;
}
```

… oppure una suite di 4 distruttori:

```
int isEmptyBT2(binTree B)
    if (!B) return 1;
    return 0;
}
```

```
binTree right(binTree B)
/* PREC: B!= # */
    return B->right;
}
```

```
binTree left(binTree B)
/* PREC: B!= # */
    return B->left;
}
```

```
int root(binTree B)
/* PREC: B!= # */
    return B->info;
}
```

Prime funzioni su alberi in C

Vediamo l'implementazione in **C** di alcune semplici funzioni:

```
int nodes(binTree B){
   int r;
   binTree L, R;
   if (isEmptyBT(B, &r, &L, &R))
        return 0;
   return 1+nodes(L)+nodes(R);
}
```

```
int weight(binTree B){
   int r;
   binTree L, R;
   if (isEmptyBT(B, &r, &L, &R))
        return 0;
   return r+weight(L)+weight(R);
}
```

```
int depth(binTree B){
   int r;
   binTree L, R;
   if (isEmptyBT(B, &r, &L, &R))
        return -1;
   return 1+max(depth(L), depth(R));
}
```

Prime funzioni su alberi in C

Leggermente più laboriosa la funzione
`int equalsBT(binTree, binTree)`:

```
int equalsBT(binTree B1, binTree B2){
   int r1, r2, e1, e2;
   binTree L1, L2, R1, R2;
   e1 = isEmptyBT(B1, &r1, &L1, &R1));
   e2 = isEmptyBT(B2, &r2, &L2, &R2));
   if (e1 && e2) return 1;
      /* almeno uno è non vuoto */
   if (e1 || e2) return 0;
      /* entrambi sono non vuoti */
   return r1==r2 && equalsBT(L1, L2)
          && equalsBT(R1, R2);
}
```

Verificare `r1==r2` *è più a buon mercato*
quindi conviene metterlo prima,
sfruttando l'ordine di valutazione di `&&`

Lezione 25b:

Strutture Dati Generiche in C

Come gestire il polimorfismo in C?

Ovviamente un gran numero di funzioni sugli alberi **non dipendono** dal fatto che gli alberi contengano **valori interi** o di altro tipo (profondità, bilanciamento etc.).

Come è possibile avere **alberi generici** o **polimorfi**?

In **C** c'è un'opportunità data dal tipo **`void *`** che è compatibile per assegnazione con ogni altro tipo puntatore.

Quindi, è possibile definire strutture dati generici, mettendo nel campo informazione dei `void *`.

Si tratta di una tecnica del tutto analoga a quella usata ad esempio in **Java prima dell'introduzione dei Generics**: strutture dati generiche venivano implementate mantenendo nelle strutture dati, oggetti di tipo `Object`.

Alberi Generici in C

La rappresentazione è del tutto analoga a quella degli alberi appena visti.

Cambia solo il tipo del campo informazione, che stavolta sarà di tipo `void *`.

Attenzione: occorre modificare opportunamente le funzioni che ad esempio aggiungono elementi in un albero: in particolare, occorre sempre allocare memoria dinamica per le informazioni inserite in un albero.

```
typedef
struct BG {
   struct BG * left;
   void* info;
   struct BG * right;
} binTreeGNode;

typedef binTreeGNode* binTreeGen;
```

Tipi Generici in C: Limiti e Vantaggi

Ovviamente questo trucco ha alcuni punti deboli:

1. Non c'è **nessun controllo** sulle informazioni inserite in una struttura dati: il sistema dei tipi non controlla il tipo delle informazioni inserite;
2. Quando vado a estrarre informazioni, il programmatore deve assegnare il valore di tipo `void *` a una variabile di tipo T* per poterlo usare (con `T` tipo noto).

Al solito, come contraltare, guadagno in flessibilità, in particolare, in alcuni casi potrebbe essere "vantaggioso" o quantomeno "comodo" poter inserire valori di tipi non omogenei dentro una struttura dati.

Esempio: una pila generica potrebbe rappresentare la pila di sistema, dove ogni funzione ha un record di attivazione di tipo diverso rispetto alle altre funzioni.

Esempio: Pile Generiche (1)

Vediamo rapidamente l'esempio delle pile generiche.

Implementiamo le Pile, usando una lista semplice.

```
typedef struct S {
    void * val;
    struct S * next;
} Snode;

typedef struct {
    Snode* top;
    int numElem;
} stackDescriptor;

typedef stackDescriptor *stack;
```

Esempio: Pile Generiche (2)

Vediamo le operazioni della pila.

```
stack createEmptyStack(){
   stack S;
   S = malloc(sizeof(stackDescriptor));
       /* la cima della pila viene inizializzata a NULL  */
   S->top = NULL;
       /* numero di elementi a 0  */
   S->numElem = 0;
   return S;
}
```

```
int isEmpty(stack S){
       /* equivalente a return S->top==NULL */
   return S->numElem==0;
   }
```

```
void pop(stack S){
   Snode *tmp;
       /* si mantiene l'invariante del tipo di dato */
   S->numElem--;
       /* salva il putatore al primo elemento per la free */
   tmp = S->top;
       /* modifica il puntatore al top della pila */
    S->top = (S->top)->next;
    free(tmp);
}
```

```
void* top(stack S){
   return (S->top)->elem;
}
```

Esempio: Pile Generiche (3)

Vediamo le operazioni della pila.

```
void push(stack S, void* el){
   Snode *tmp;
      /* alloca memoria per inserire un nuovo nodo */
   tmp = malloc(sizeof(Qnode));
       /* il link del nuovo nodo punta al vecchio top */
   tmp->next = S->top;
   tmp->elem = el;
   S->top = tmp;
      /* si mantiene l'invariante del tipo di dato */
   S->numElem++;
}
```

Uso delle Pile Generiche (2)

Consideriamo il problema della Torre di Hanoi, mantenendo lo stato del gioco in tre pile di dischi.

Inoltre, useremo le stesse pile per produrre un programma iterativo che elimina la ricorsione simulando esplicitamente la pila di sistema.

Cominciamo con le torri del gioco.

```
void inizializzaTorri(stack torri[], int n){
    int *x;
    int i;

    for (i=0; i<3; i++)
        torri[i]=createEmptyStack();
    for (i=n; i>0; i--){
          x = malloc(sizeof(int));
          *x=i;
          push(torri[0], x);
        }
}
```

Sarebbe un errore da principianti fare **`push(torri[0], &i)`**

Uso delle Pile Generiche (2)

Mostriamo la versione ricorsiva e la mossa di un disco.

```
void hanoi(int sorg, int aux, int dest, int n, stack torri[]){
/* inizio */
   if (n==1) move(sorg, dest, torri);
      else {
         hanoi(sorg, dest, aux, n-1, torri);
/* mezzo */
         move(sorg, dest, torri);
         hanoi(aux, sorg, dest, n-1, torri);
/* fine */
      }
 }
```

```
void move(int sorg, int dest, stack torri[]){
    push(torri[dest], top(torri[sorg]));
    pop(torri[sorg]);
    printTorri(torri);
}
```

Lo stato delle torri può essere necessario ad es. per una versione grafica

Uso delle Pile Generiche (3)

```
void hanoiIterOpt(stack torri[]){
    hanoiAR2 *newAR, *AR;
    stack controlStack;
    int aux, sorg, dest, nd, w;
    controlStack = createStack(2048);
    sorg=0; aux=1; dest=2; nd=howManyStack(torri[0]);
```

```
while (1) {
      if (nd == 1) {
         move(sorg, dest, torri);
         if (!isEmpty(controlStack)) {
             AR = top(controlStack);
             pop(controlStack);
             sorg = AR->sorg; dest = AR->dest;
             aux = AR->aux; nd = AR->nd;
         } else break;
       } else {
         newAR = createAR2(aux, sorg, dest, nd - 1);
         push(controlStack, newAR);
         newAR = createAR2(sorg, aux, dest, 1);
         push(controlStack, newAR);
         nd--; w=aux; aux=dest; dest=w;
       } /* end else */
 }/* end while */
```

Lezione 25c ~ Intermezzo

Un week-end al SortPub (aka: Esercizi di Stile)

Funzione merge: madrelingua Pascal

```
void mergePascal(int a[], int inf, int medio, int sup){
  /* PREC: forall i. inf<=i<med. a[i]<=a[i+1] &
   *       forall i. med<=i<sup. a[i]<=a[i+1]
   * POST: forall i. inf<=i<sup. a[i]<=a[i+1]
   */
     int i=inf;
     int j=medio;
     int k=0;
     int c[sup-inf];

     while (i < medio && j < sup) {
         if (a[i] <= a[j]) {
            c[k] = a[i];
            i = i+1;
           } else {
               c[k] = a[j];
               j = j+1;
         } /* endif */
         k = k+1;
     } /* endwhile */
     if (i<medio)
       for (; i<medio; i++) { c[k]=a[i]; k=k+1; }
     if (j<sup)
       for (; j<sup; j++) { c[k]=a[j]; k=k+1; }
     copia(c, k, a, inf, sup);
}
```

Innanzitutto osserviamo che ha fatto la fusione di due pezzi dello stesso vettore dentro un vettore ausiliario. Del resto, è in questa forma che a lui serve nell'algoritmo *MergeSort* (vedi Fig. 1). E gli sarebbe difficile (in un linguaggio di famiglia ALGOL), all'interno di *MergeSort* usare una funzione *merge* più generale, che fonde due vettori ordinati qualsiasi, se a lui serve fondere due porzioni dello stesso vettore. Non avendo

Oltre a questo, scrive ben 6 caratteri per incrementare una variabile contatore (spesso 9 o più a causa della generosità con cui è aduso distribuire spazi tra un operatore infisso e i suoi operandi) laddove un programmatore C ne userebbe al più 4 con l'incremento postfisso.

C'è ancora qualcosa che lo disturba. I due `if` finali sono evidentemente inutili. Sono sussunti dalle guardie dei `for`. Un ultimo sorriso interiore mentre termina di scrivere: "programmare in Pascal atrofizza il cervello. . . ".

Pseudocodice nelle dispense di…

```
Funzione Fondi (A: vettore; indice_primo, indice_medio, indice_ultimo: intero)
1       i ← indice_primo;
2       j ← indice_medio + 1;
3       k ← 1;
4       while ((i ≤ indice_medio) and (j ≤ indice_ultimo))
5               if (A[i] < A[j])
6                       B[k] ← A[i]
7                       i ← i + 1
8               else
9                       B[k] ← A[j]
10                      j ← j + 1
11              k ← k + 1
12      while (i ≤ indice_medio) //il primo sottovettore non è terminato
13              B[k] ← A[i]
14              i ← i + 1
15              k ← k + 1
16      while (j ≤ indice_ultimo) //il secondo sottovettore non è terminato
17              B[k] ← A[j]
18              j ← j + 1
19              k ← k + 1
20      ricopia B[1..k-1] su A[indice_primo..indice_ultimo]
21      return
```

Merge VeroProgrammatoreC (sciolto)

```
void merge(int a[], int m, int b[], int n, int c[]){
    int i=0;
    int j=0;
    int k=0;

    while (i<m && j<n)
          if (a[i]<=b[j]) c[k++]=a[i++];
                     else c[k++]=b[j++];
    while (i<m) c[k++]=a[i++];
    while (j<n) c[k++]=b[j++];
}
```

Il pomeriggio, senza strafare, e senza usare l'artimetica dei puntatori che il suo amico Niklaus, madrelingua PASCAL, non capirebbe, scrive la funzione in Fig. 3. Innanzitutto, scrive una funzione più generale, che fonde due vettori qualsiasi. Tanto lui, nella *MergeSort* la potrà comunque chiamare con `merge(&a[inf], c-inf, &a[c], sup-inf, int c[]);` e poi chiamare la funzione `copia`.

Subito dopo, compatta tutti gli incrementi degli indici dentro le assegnazioni tra elementi dell'array. In fondo vanno incrementati proprio quelli coinvolti nelle assegnazioni. Immagina già l'obiezione dell'amico-nemico: "ma è un caso fortunato". Lui pensa con il suo spirito pratico tipico dei Veri Programmatori C: "sarà pure fortuna, ma di solito è così. . . ".

Contaminazioni

Come trarre ispirazione da una specifica ricorsiva su sequenze per la merge? In fondo merge non usa i vettori nella loro piena gloria, ma li **processa sequenzialmente**, come una sequenza…

$$\mathsf{merge}(a_1 \cdot s_1, a_2 \cdot s_2) = \begin{cases} a_1 \cdot \mathsf{merge}(s_1, a_2 \cdot s_2) & \text{se } a_1 \leq a_2 \\ a_2 \cdot \mathsf{merge}(a_1 \cdot s_1, s_2) & \text{se } a_2 < a_1 \end{cases}$$

$$\mathsf{merge}(\langle\,\rangle, s) = \mathsf{merge}(s, \langle\,\rangle) = s$$

I vettori C **possono essere passati con un pointer all'inizio**, mentre la lunghezza ci può dire se sono terminati o meno. In questo caso usiamo due interi `ra` **e** `rb` **che ci dicono quanti sono gli elementi che mancano alla fine**.

Di fatto, usiamo dei **vettori come fossero sequenze.**

Merge ricorsiva VPC

```
void mergeRecVPC(int* a, int ra,
                   int* b, int rb, int* c){
   if (!ra) copiaRec(b, rb, c);
   else if (!rb) copiaRec(a, ra, c);
   else {
      if (*a<=*b){
         *c++=*a++;
         ra--;
      } else { *c++=*b++;
               rb--;
             }
      mergeRecVPC(a, ra, b, rb, c);
      } /* end else */
   }
```

```
void copiaRec(int* a, int ra, int* c){
   if (ra){
         *c++=*a++;
         copiaRec(a, ra-1, c);
      }
}
```

Merge ricorsiva VPC senza copia

Dennis, a un tratto però è preso da un momento di sconforto e si lascia andare a una sonora imprecazione. Allora rientra in casa, apre con impazienza il suo inseparabile portatile e si mette di nuovo febbrilmente al lavoro: si è reso conto che la funzione `copiaRec` è inutile! In fondo, `mergeRec` di suo, fa già molto di più. Ecco il suo super-compresso programma in Fig. 6.

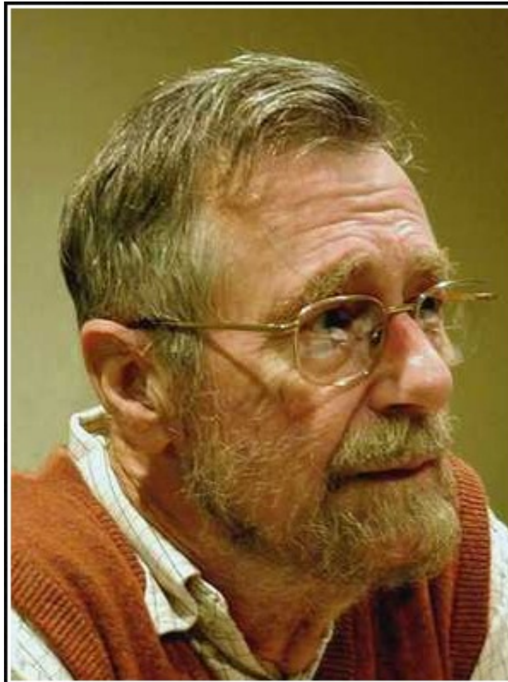
```
void mergeRecVPC(int* a, int ra,
                         int* b, int rb, int* c){
   if (!ra && !rb) return;
   if (!rb || (ra && *a<=*b)){
         *c++=*a++;
         ra--;
      } else {
         *c++=*b++;
         rb--;
      }
   mergeRecVPC(a, ra, b, rb, c);
}
```

Questo programma specula 2 volte (!!) sull'ordine di valutazione di || e **&&**

Si rammarica un po' perché la fusione di vettori venerdì prossimo al *SortPub* sarà già fuori moda e non potrà farsi bello col suo programma. Ora però non gli interessa più: contemplare il suo piccolo capolavoro lo appaga completamente!

Lezione 25d:

Una lezione dal Maestro E.W. Dijkstra



Teaching to unsuspecting youngsters the effective use of formal methods is one of the joys of life because it is so extremely rewarding.

— Edsger Dijkstra —

AZ QUOTES

Uguaglianza a meno di shift

Scrivere un programma che verifica se due vettori u e v di n elementi sono uguali a meno di shift.

Dobbiamo verificare se esiste k ($0 \leq k < n$) per cui:

$$u_i = v_{i+k \bmod n} \text{ per ogni } i \ (0 \leq i < n).$$

Il problema ammette **un'ovvia soluzione quadratica**, che consiste nel provare tutti i k e verificare l'uguaglianza dei due vettori visti come circolari:

```
int equalsShift(int u[], int v[], int n, int *s){
    int i;
    for(int k=0; k<n; k++){
        i=0;
        while(i<n && u[i]==v[(i+k)%n]) i++;
        if (i==n){*s=k; return 1;}
    }
    return 0;
}
```

caso pessimo: $SA_0 = \underbrace{000\ldots0}_{n-1}1$ e $SB_0 = \underbrace{000\ldots0}_{n-2}11$).

Lavorare per una specifica "migliore"

Riscriviamo la **specifica** in modo da renderla **simmetrica** (scriveremo sempre v_i per $v_{i \bmod n}$):

$$\exists i, j \forall k.\ u_{i+k} = v_{j+k}$$

e indichiamo con U_i (resp. V_i) la sequenza ottenuta leggendo il vettore u (resp. v) in modo circolare, cioè:

$$U_i = u_i\, u_{i+1} \ldots u_{n-1} u_0 u_1 \ldots u_{i-1} \qquad V_i = v_i\, v_{i+1} \ldots v_{n-1} v_0 v_1 \ldots v_{i-1}$$

e possiamo infine scrivere la specifica in modo compatto:

$$\exists i, j.\ U_i = V_j$$

Ovviamente, testare tutte le coppie U_i e V_j porta a un programma quadratico, sulla falsariga del precedente.

Come spesso accade, **trovare un ordine nella ricerca** permette di recuperare il lavoro…

Andiamo con ordine…

Consideriamo l'insieme delle sequenze **U** $= \{U_i \mid 0 \leq i < n\}$ e **V** $= \{V_i \mid 0 \leq i < n\}$ equipaggiate con **l'ordine lessicografico** (quello del dizionario, per intenderci).

Ovviamente, $\exists\, i, j.\ U_i = V_j$ implica che che **U** e **V** sono **lo stesso insieme di sequenze**, e in particolare essendo **l'ordine lessicografico** un **ordine totale**, se $\exists\, i, j.\ U_i = V_j$ avrò anche che **i massimi** $U^* = max$ **U** e $V^* = max$ **V** sono uguali.

L'esplorazione di trovare due sequenze uguali, non sarà più completamente brute-force, ma **guidata dall'ordine lessicografico**, sempre verso sequenze maggiori, allo scopo di verificare quale delle seguenti tre proprietà sia soddisfatta:

$$\exists\, i\ U_i > V^* \qquad \exists\, i\ V_i > U^* \qquad \exists\, i, j\ U_i = V_j$$

e tornando 0 nei primi due casi e 1 nel terzo.

…verso il lieto fine

Consideriamo il seguente frammento di programma:

```
while(h<n && u[(i+h)%n==v[(j+h)%n]) h++;
```

Se usciamo da questo ciclo perché h=n, ovviamente abbiamo finito, scoprendo che $U_i = V_j$.

Viceversa, sappiamo che ci sono h elementi iniziali comuni tra U_i e V_j. Potremo ripartire con un'altra coppia di indici, ma questo (tenendo fermo i o j) porterebbe a un programma quadratico equivalente a quello visto prima.

Ma cosa ci dice $u_{i+h} < v_{j+h}$ nelle sequenze ordinate? Ci dice che $U_i < V_j \leq V^*$. In realtà, avendo verificato h uguaglianze sappiamo anche che $U_{i+k} < V_{j+k} \leq V^*$ per ogni $0 \leq k < h$. Ovviamente, anche $U_{i+h} < V_{j+h} \leq V^*$. Possiamo di conseguenza ripartire confrontando U_{i+h+1} con V_j **sperando di salire nell'ordine lessicografico…**

Simmetricamente $u_{i+h} > v_{j+h}$ possiamo saltare a confrontare U_i con V_{j+h+1} **scartando sequenze sicuramente non massime**.

Lieto fine: asserzioni logiche

Riassumendo, nel nostro ciclo sarà sempre vero che:

$P[i, j, h] \equiv \forall k \in [0, h).u_{i+k} = v_{j+k}$: che asserisce che i due vettori coincidono nell'intervallo $[0, h)$: questo viene mantenuto vero perché quando trovo elementi uguali incremento h, mentre riazzero h, quando trovo elementi diversi.

$Q_U[i, k] \equiv \forall k \in [0, i).\ U_k < V^*$: tutte le sequenze **già esaminate** nel vettore u sono minori del massima V*: questo perché quando si incrementa i abbiamo $u_{i+h} < v_{j+h}$.

$Q_V[i, k] \equiv \forall k \in [0, j).\ V_k < U^*$: tutte le sequenze **già esaminate** nel vettore v sono minori della massima U*: questo perché quando si incrementa j abbiamo $u_{i+h} > v_{j+h}$.

Terminazione: $3n-(i+j+h)$ Ho 3 forme di modifica di i, j e h:

1) $i'=i$, $j'=j$ e $h'=h+1$ implica che $i'+j'+h'=i+j+h+1$;

2) $i'=i+h\mathbf{+1}$, $j'=j$ e $h'=0$ implica che $i'+j'+h'=i+j+h+1$;

3) $i'=i$, $j'=j+h\mathbf{+1}$ e $h'=0$ implica che $i'+j'+h'=i+j+h+1$;

Osservazione: il +1 **è essenziale** ☺ e si fanno al più $3n$ **iterazioni**.

Dulcis in fundo: il programma

```
int equalsShift(int u[], int v[], int n, int *s){
   int i=0;
   int j=0;
   int h=0;
   int x, y;

   while(i<n && j<n && h<n){
      x = (i+h)%n;
      y = (j+h)%n;
      if (u[x]==v[y]) h++;
          else{ if (u[x] < v[y]) i=i+h+1;
                   else j=j+h+1;
                h=0;
              }
   }

   if (h < n) return 0;
   *s = abs(i-j);
   return 1;
}
```

Lezione 25e:

Piccole note pratiche

Suddivisione programmi C (1)

Usualmente le definizioni dei tipi e i prototipi di alcune funzioni relative a una struttura dati vengono messi in un file .h

eulero.h

```
1    #include<stdlib.h>
2    #include<stdio.h>
3
4    typedef
5        struct{
6            short unsigned int prec;
7            short unsigned int succ;
8         } Pair;
9
```

Suddivisione programmi C (2)

Usualmente il codice C di tali funzioni viene scritto in un file .c

eulero.c

```c
#include<stdlib.h>
#include<stdio.h>
#include "eulero.h"

void init(Pair * p, int n){
    for (int i=2; i<n; i++){
        p[i].succ=1;
        p[i].prec=1;
    }
}

void cancella(Pair * p, int j, int n){


}

Pair * eulerSieve(int n){
    int* v=calloc(n,sizeof(int));
```

Osservate **#include"eulero.h"**

Suddivisione programmi C (3)

Alla fine si compila tutto insieme:

```
gcc –o eulero eulero.c main.c
```

(gli `#include “eulero.h”` vanno messi ovunque si chiamino funzioni il cui prototipo è definite in `eulero.h`)

Ogni file dovrebbe **compilare separatamente** con il comando:

```
gcc –c eulero.c
gcc –c main.c
```

che producono file **non eseguibili** `eulero.o` e `main.o`.

Lezione 25

That's all Folks!

Grazie per l'attenzione…

…Domande?