Tecniche di Programmazione Funzionale e Imperativa

Ivano Salvo

Alcune "perle" di stile e di programmazione

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

SAPIENZA
UNIVERSITÀ DI ROMA

Lezione **23**, 24 maggio 2021

Lezione 23a ~ 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

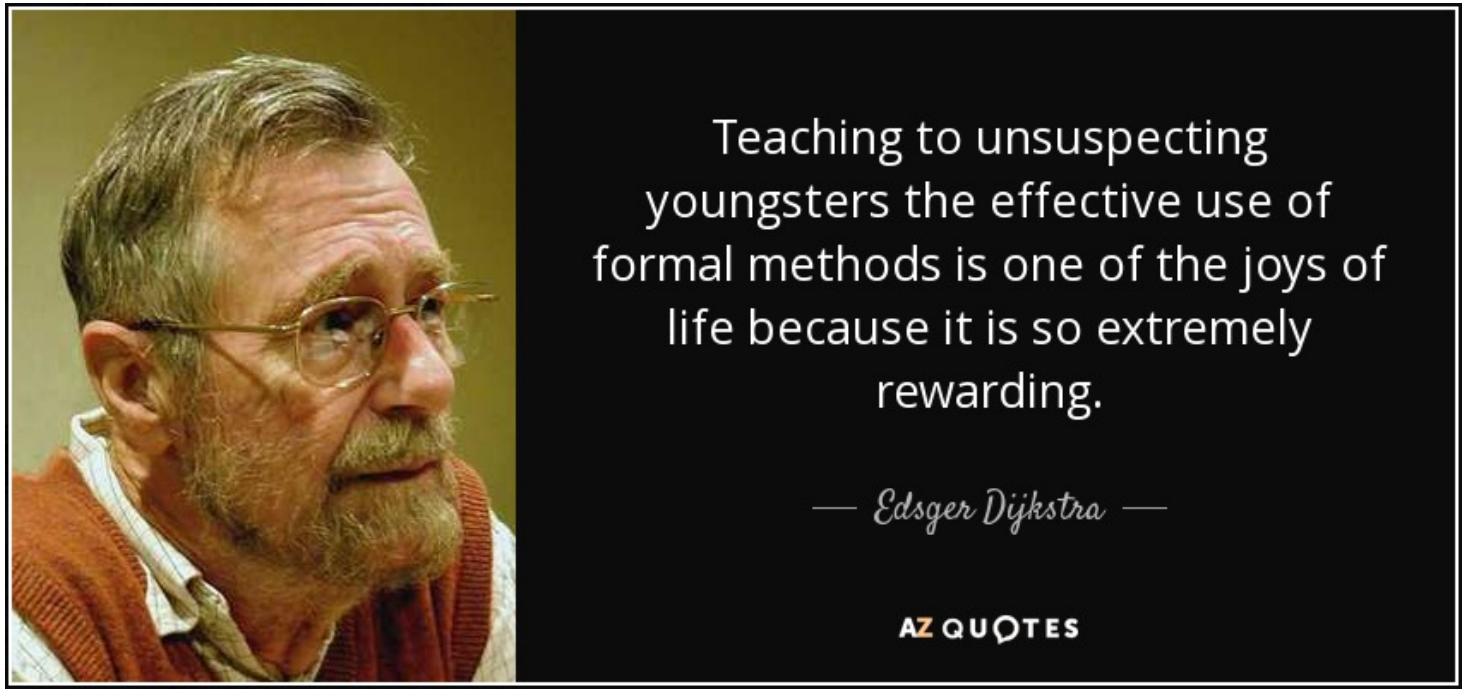
Dennis, a un tratto però è preso da un momento di sconforto e si lascia andare a una sonora imprecazione. Allora rientra in casa, apre con impazienza il suo inseparabile portatile e si mette di nuovo febbrilmente al lavoro: si è reso conto che la funzione `copiaRec` è inutile! In fondo, `mergeRec` di 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 23c:
Una lezione dal Maestro E.W. Dijkstra



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 ($0 \le k < n$) per cui:

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

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 $\mathbf{U} = \{U_i \mid 0 \le i < n\}$ e $\mathbf{V} = \{V_i \mid 0 \le i < n\}$ 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** $U^* = max\ \mathbf{U}$ e $V^* = max\ \mathbf{V}$ 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**.

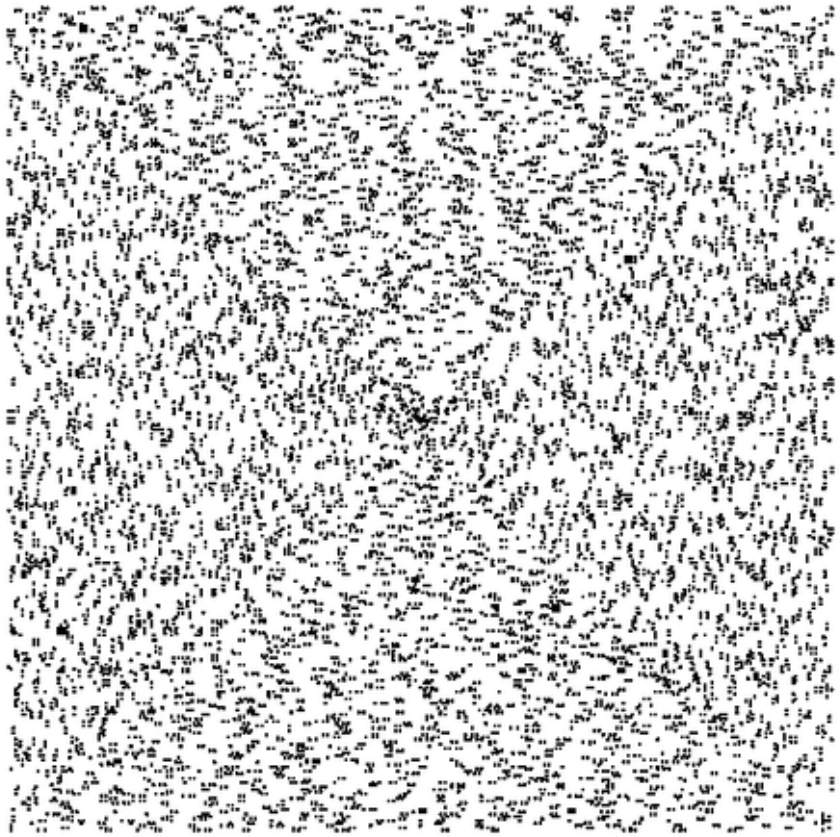
Dulcis in fundo: il programma

```
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 13a:
Ulam numbers



(1971), 249–257. These mysterious numbers, which were first defined by S. Ulam in *SIAM Review* **6** (1964), 348, have baffled number theorists for many years. The ratio

Ulam "generativo"

Abbiamo programmato un `**crivello di Ulam**': si provano tutti i naturali e si tengono solo quelli riconosciuti soddisfare i requisiti per essere il prossimo numero di Ulam. Un'alternativa è quella di **generare tutte le somme**.

Lavorando con array, è possibile tenere un array *s* che riporta in posizione *i* il numero di coppie di distinti numeri di Ulam che generano *i* come somma.

Attenzione: generare 1 sola volta le somme. Trovato u_n si possono incrementare tutti i valori s_j tali che $j = u_i+u_n$ per $i<n$.

Il problema più serio è **quante posizioni allocare per *s***: possiamo speculare sul fatto che $\lim_{n\to\infty} u_n/n$=13.52 e quindi allocare $27n$ posizioni (ricordiamo però che generemo le somme fino a $u_{n-1}+u_{n-2}$). **Con le liste questa idea è più elegante** [**Esercizio**]

constant, ≈ 21.6016. Calculations by Jud McCranie and the author for $U_n < 640000000$ indicate that the largest gap $U_n - U_{n-1}$ may occur between $U_{24576523} = 332250401$ and $U_{24576524} = 332251032$; the smallest gap $U_n - U_{n-1} = 1$ apparently occurs only when $U_n \in \{2, 3, 4, 48\}$. Certain small gaps like 6, 11, 14, and 16 have never been observed.]

Ulam: un programma "generativo"

```
int ulam(int n){
   int u[n+1];
   u[0]=1; u[1]=2;

   int *s = (int*) calloc(27*n,sizeof(int));
   int nu=2;

   for (int i=1; i<n; i++){
         /* aggiorna le somme con u[i] */
      for (int j=i-1; j>=0; j--)
                  s[u[j]+u[i]]++;
         /* cerca il prox */
      while (s[nu]!=1) nu++;
         /* carica il nuovo Ulam number
          * sposta l'inizio nuova ricerca*/
      u[i+1]=nu++;
   }
   return u[n];
}
```

Esercizio per gli impavidi [Knuth]

Diverse persone hanno dedicato la vita a dare la caccia agli Ulam numbers e alle loro sfuggevoli proprietà. Sul fronte computazionale, citerei Philip Gibbs, Jude McCranie… e un informatico d'eccezione: **Donald E. Knuth**.

I virtuosi e/o gli impavidi possono provare a implementare quest'idea, basata su operazioni bit a bit (in C, sono &, | etc.) **ma sostanzialmente simile al programma `generativo':**

141. Suppose we've computed bits $a = a_0 a_1 \ldots a_{2m-1}$ and $b = b_0 b_1 \ldots b_{2m-1}$ such that

$a_s = [s = 1$ or $s = 2$ or s is a sum of distinct Ulam numbers $\le m$ in exactly one way$]$,

$b_s = [s$ is a sum of distinct Ulam numbers $\le m$ in more than one way$]$,

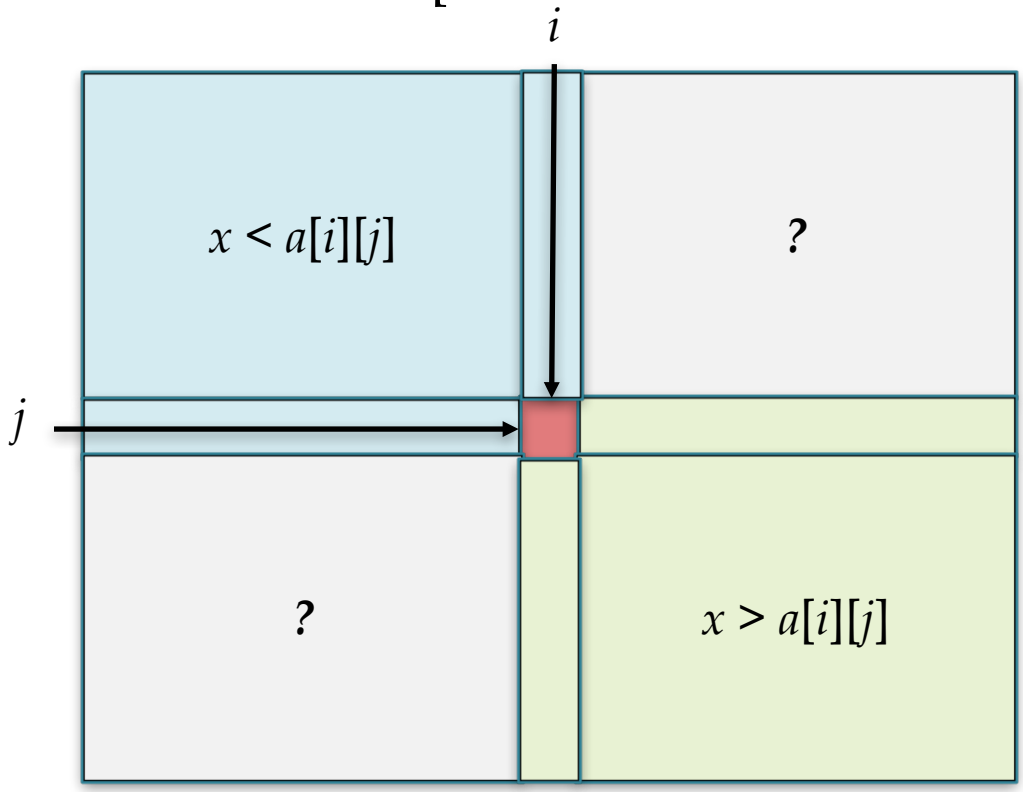
for some integer $m = U_n \ge 2$. For example, when $m = n = 2$ we have $a = 0111$ and $b = 0000$. Then $\{s \mid s \le m$ and $a_s = 1\} = \{U_1, \ldots, U_n\}$; and $U_{n+1} = \min\{s \mid s > m$ and $a_s = 1\}$. (Notice that $a_s = 1$ when $s = U_{n-1} + U_n$.) The following simple bitwise operations preserve these conditions: $n \leftarrow n + 1$, $m \leftarrow U_n$, and

$$(a_m \ldots a_{2m-1}, b_m \ldots b_{2m-1}) \leftarrow ((a_m \ldots a_{2m-1} \oplus a_0 \ldots a_{m-1}) \,\&\, \overline{b_m \ldots b_{2m-1}},$$
$$(a_m \ldots a_{2m-1} \,\&\, a_0 \ldots a_{m-1}) \mid b_m \ldots b_{2m-1}),$$

Problemi simili: ricerca su matrice

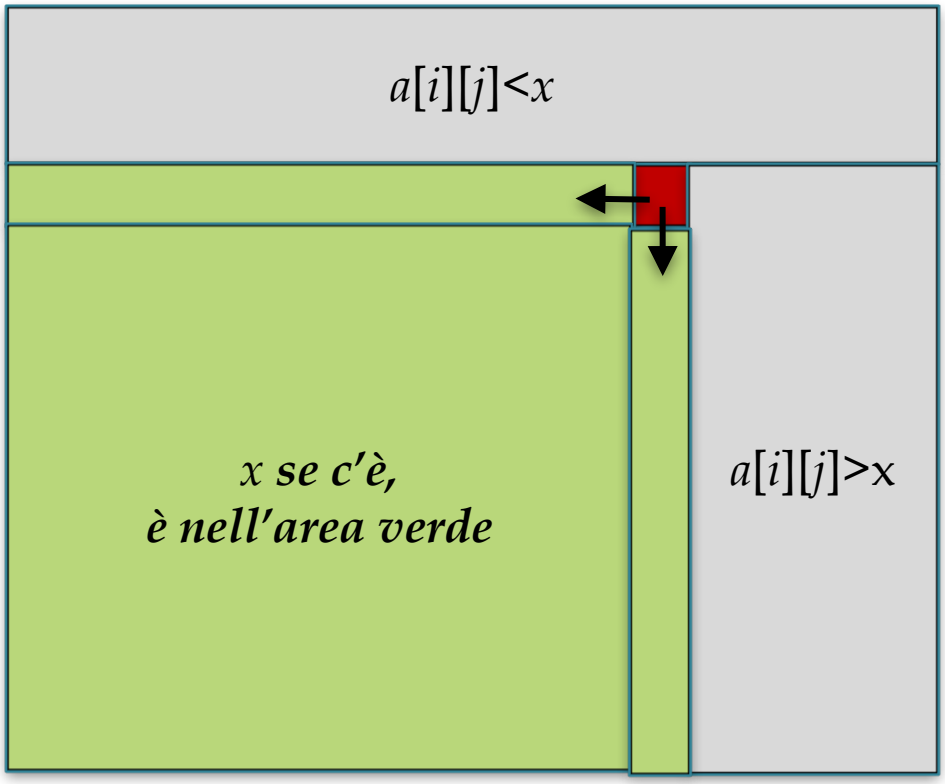
Una matrice è ordinata per righe e colonne se $a_{i,j} \leq a_{i+1,j}$ e $a_{i,j} \leq a_{i,j+1}$

Idea 1: usare una tecnica **divide et impera**: ad **ogni passo posso scartare ¼ della matrice**. [**Esercizio**: calcolare la complessità].



Ricerca su matrice: geodetica

Idea 2: comincio dall'angolo in alto a destra… Se $x > a[i][j]$ vado verso il basso (**posso escludere tutta la riga**), se $x < a[i][j]$ vado a sinistra (**posso escludere la colonna**). **Complessità**: $\mathcal{O}(m+n)$



Programma: Geodetica

```
int ricMatOrd(int** a, int m, int n, int x,
                        int *x, int *y){
   int i=0;
   int j=n-1;

   while (i<m && j>=0){
    /* INV: x\in a[i..m-1][0..j]
     * TERM: j – i
     */
      if (a[i][j]>x) i++;
      else if (a[i][j]<x) j--;
      else {
         *x = i;
         *y = j;
         return 1;
      } /* end else */
   } /* end while */
   return 0;
}
```

Lezione 23

That's all Folks!

Grazie per l'attenzione…

…Domande?