Correzione Primo Esonero, Esercizio 2

Ivano Salvo – Sapienza Università di Roma

Anno Accademico 2017-18

Esercizio 2: *Considerare il problema di trovare il* minimo *intero* non presente *in un vettore di interi non negativi* distinti. *Ad esempio, nel vettore:*

$$\{3, 8, 4, 5, 11, 15, 9, 2, 6, 0, 1\}$$

il minimo intero non presente è il 7.

Punto 1 *Scrivere una funzione* `C` *di prototipo* `int minFree(int v[], int n)` *che restituisca il minimo intero non presente in* `v`. *La funzione* `minFree` non deve *modificare il vettore* `v` *e* non può *allocare strutture dati di dimensione dipendente da* n. *Valutare, anche informalmente, la complessità della funzione.*

Non potendo allocare memoria, né modificare il vettore, l'unica soluzione parrebbe (ma ci sarà una sorpresa nel finale!) cercare il primo numero non presente, iterando una ricerca o un calcolo di un minimo. Più precisamente:

1. provare a vedere se un qualunque numero sta in v partendo da 0 e continuando con 1,2,... fino a quando non se ne trova uno assente (banalmente questo numero esiste, essendo il vettore finito e i naturali infiniti, ma si può essere più precisi osservando, come evidenziato nella traccia del Punto **2**, che questo numero è necessariamente compreso nell'intervallo $[0, n]$[1];

2. calcolare i minimi successivi m del vettore (cioè il minimo *maggiore* del minimo precedente pm) e non appena $m > pm + 1$, dedurre che il minimo numero non presente in v è proprio $pm + 1$ (pm va inizializzata a -1 per il primo ciclo).

Dovrebbe essere abbastanza intuitivo che entrambe le procedure considerate sono $\mathcal{O}(n^2)$, con n lunghezza del vettore.

Percorriamo la prima strada scrivendo la funzione in Fig. 1. Osserviamo che `trova` ha il prototipo più consono alla sua natura, caricando nell'ultimo parametro l'indice dell'elemento eventualmente trovato, ma a noi questo risultato qui non serve.

Osservate anche che la condizione `i<n` non serve a nulla, perchè so che prima o poi un numero assente verrà trovato e quindi si uscirà dal `for` di `minFree` per effetto

[1] non è possibile mai definire una funzione suriettiva da un insieme di cardinalità n in un insieme di cardinalità m se $n < m$ (o iniettiva se $m > n$). Questa proprietà viene spesso chiamata *principio dei buchi di piccionaia* o, in inglese, *pidgeon-hole principle* e curiosamente gioca un ruolo chiave in diverse dimostrazioni di Informatica Teorica.

```
int trova(int x, int v[], int n, int *p){
   for (int i=0; i<n; i++)
      if (v[i]==x) {*p=i; return 1;}
   return 0;
}

int minFree(int v[], int n){
   int p;
   for (int i=0; i<n; i++)
      if (!trova(i,v,n,&p)) return i;
   return n;
}
```

Figura 1: Funzione che non alloca memoria, né modifica il vettore.

del `return i`. Potremo scrivere allora del codice da vero programmatore `C` nella funzione `minFreeVPC` (vedi Fig. 2, in cui viene fatto tutto dalla guardia del `while` (il corpo del ciclo è vuoto!).

```
int minFreeVPC(int *v, int n){
   int p, i=-1;
   while (trova(++i,v,n,&p));
   return i;
   }
```

Figura 2: Funzione che non alloca memoria, né modifica il vettore. Versione VPC

Ho peraltro visto molte soluzioni, molte delle quali ingegnose e piacevoli da leggere, il più delle volte senza una funzione ausiliaria per cercare un numero nel vettore. Tra tutte vorrei segnalarne un paio in Fig. 3:

1. `minFreeNonAnnidata` usa un solo ciclo. Colgo l'occasione per stimolare gli irriducibili riduzionisti a dimostrare che ogni programma (diciamo iterativo) può essere scritto senza cicli annidati. Non è usualmente una buona idea, intendiamoci, in quanto spesso porta a programmi poco leggibili. Qui risulta elegante.

2. `minFreeIMin` sfrutta il fatto che la variabile `min` cresce insieme alla variabile dei potenziali numeri liberi `i` (che viene sempre incrementata). Non appena la variabile `min` non viene incrementata, significa che non ho trovato il valore *min* in *v* e quindi *min* è il primo numero "libero" in `v`.

Punto 2: *Osservando che il minimo intero non presente in* `v` deve *necessariamente essere un numero nell'intervallo* $[0, n]$*, dove* n *è la lunghezza del vettore, fornire una funzione* `int minFreeLin(int v[], int n)` *di complessità lineare in*

```
int minFreeNonAnnidata(int v[], int n){
   int i=0;
   while (i<n && j<n)
    if (v[i]==j) { i=0; j++;}
              else i++;
   return j;
   }

int minFreeIMin(int v[], int n){
   int min=0;
   int i=0;
   while (min==i){
      for (j=1; j<n; j++)
         if (min==a[j]) min++;
      i++;
   }
   return min;
}
```

Figura 3: Divertenti variazioni sul tema.

n che risolve lo stesso problema avendo la libertà di allocare un vettore di opportuna lunghezza locale alla funzione `minFreeLin`.

Avendo la possibilità di allocare un vettore p di lunghezza n possiamo marcare la presenza del numero i ponendo $p_i = 1$. Questa scansione costa n e poi, sempre con una scansione lineare, possiamo cercare il primo elemento nullo del vettore p. Semplice ed efficace. Vediamo il programma in Fig. 4.

Anche questa idea ammetteva innumerevoli variazioni sul tema. Non ho molto gradito (ma senza punizioni), le soluzioni che necessitavano di un trattamento particolare ad esempio del numero 0 (questo per coloro che hanno posto $p_{v_i} = v_i \ldots$ a quel punto, la cosa non funzionava per p_0). Osserviamo solamente due cose:

1. la dichiarazione `int p[n]={0};` non viene accettata dai compilatori con vettori di lunghezza *variabile*. Non ho tolto punti per questo "errore". Comunque, poco male, possiamo invocare una funzione `azzera` (meglio di azzerare il vettore in loco con un ciclo `for` per motivi di semplicità del codice – anche qui questioni che non modificano il punteggio ottenuto).

2. Più importante, ricordarsi che prima di settare p_{v_i} ad 1 è necessario assicurarsi che $v_i \leq n$. Attenzione sempre alla *zona proibita*!

Errore Interessante: Al solito, alcuni errori sono particolarmente interessanti. Qualcuno, noncurante dei saggi consigli della traccia, ha voluto calcolare il valore $m = \max_i v_i$ e poi definire il vettore p con m elementi. Questo evita il controllo $v_i \leq n$, ma cosa accade se il vettore v, contenesse ad esempio i valori $\{0, 1, 2, 3, 2^{64} - 1\}$?

```
int minFreeLin(int v[], int n){
   int p[n+1];

   azzera(p, n+1);
   for (int i=0; i<n; i++)
      if (v[i]<=n) p[v[i]]=1;

   for (int i=0; i<n+1; i++)
      if (!p[i]) return i;
}
```

Figura 4: Funzione `minFreeLin`

Punto 3: *Non potendo allocare memoria come al punto 1, ma potendo modificare il vettore, in quale altro modo è possibile accelerare la funzione scritta al punto 1? [non è richiesto codice per questo punto]*

Qui occorreva semplicemente dire che una volta riordinato il vettore (investendo $n \log n$ passi), il minimo intero non presente è il primo indice i per cui $v_i > i$, oppure n se per ogni $i \in [0, n).v_i = i$. Chiaramente $\mathcal{O}(n \log n) + \mathcal{O}(n) = \mathcal{O}(n \log n)$.

A rigore, *mergeSort* non va tuttavia bene (qualcuno lo ha osservato!) perchè non è un algoritmo *in place* e necessita di allocare memoria (un vettore ausiliario) durante la fase di fusione (onore a chi ha fatto questa osservazione, ma niente punizioni per chi ha detto di usare *mergeSort*).

I raffinati hanno poi cercato l'intero mancante con una ricerca binaria. Finezza che non cambia la complessità $n \log n$. Ma come si fa ad adattare la ricerca binaria per trovare il minimo numero mancante? Un delicato esercizio di antipasto, assolutamente da fare, per preparare il cervello alla soluzione del Punto **4**.

Punto 4: *Adattando opportunamente l'algoritmo quickSort (in particolare la funzione partiziona), scrivere una funzione* C *basata su* divide et impera *ricorsiva di complessità lineare che non alloca memoria, ma modifica* `v`*, senza necessariamente riordinarlo tutto.* SUGG: *scrivere una funzione ausiliaria ricorsiva di prototipo:*

```
int minFreeAux(int v[], int inf, int sup)
```

che viene chiamata rispettando la precondizione che il minimo numero libero sia nell'intervallo $[inf, sup]$*.*

Chiamiamo m il minimo intero mancante e cominciamo con l'osservare che se le precondizioni sono soddisfatte, cerco nella porzione di vettore $v_{inf}, v_{inf+1}, \ldots, v_{sup-1}$ e so che $m \in [inf, sup]$. Osserviamo anche che in una ipotetica prima chiamata della funzione con parametri 0 ed n, le precondizioni sono soddisfatte per quanto osservato nella traccia del punto **2**. Inoltre ho i seguenti casi base:

• in una chiamata con $inf + 1 = sup$, ho solo v_{inf} nella porzione di vettore da analizzare e quindi deve essere che se $v_{inf} = inf$, allora $m = inf + 1$, mentre se $v_{inf} > inf$ allora $m = inf$;

• in una chiamata con $inf = sup$ necessariamente $m = inf$.

```
void swapV(int v[], int i, int j){
    int h = v[i];
    v[i] = v[j];
    v[j] = h;
    }

int minFreeAux(int v[], int inf, int sup){
 /* PREC: f \in [inf,sup] */
   int m = (inf + sup) / 2;

   if (inf==sup) return inf;
   if (sup==inf+1)
     if (v[inf]==inf) return inf+1;
                 else return inf;
   int i=inf;
   int s=sup-1;
   while (i<s){
       while (v[i]<=m) i++;
       while (v[s]>=m) s--;
       if (i<s) swapV(v,i,s);
   }
   if (i<m) return minFreeAux(v,inf,i+1);
       else return minFreeAux(v,i+1,sup);
}

int minFreeDeI(int v[], int n){
   return minFreeAux(v,0,n);
   }
```

Figura 5: Funzione lineare basata su *Divide et Impera*

Abbiamo fatto metà del lavoro. Supponete ora di conoscere un indice s del vettore per cui tutti gli elementi a sinistra di s siano minori di tutti gli elementi a destra di s. Chiamando $S = \{v_i \mid inf \leq i < s\}$ e $D = \{v_i \mid s \leq i < sup\}$ avremo che per ogni $x \in S$ e $y \in D$, necessariamente $x < y$. Se avete suddiviso il vettore seguendo la tecnica di partizionamento di *quickSort*, conoscete anche il valore p del perno che soddisfa alla condizione $x \leq p < y$, ancora con $x \in S$ e $y \in D$.

Inoltre, ragionando sulle cardinalità di S e D sapete se m sta in S o in D: infatti se $|S| < p - inf$, allora $m \in [inf, inf + s]$, altrimenti $m \in [inf + s, sup]$. A questo punto

abbiamo ricostruito le precondizioni o sulla parte sinistra o sulla parte destra (e, a differenza di *quickSort*, qui occorre analizzarne una sola delle due, analogamente ad una ricerca binaria, appunto).

Resta da decidere come scegliere il perno: non occorre che sia un elemento del vettore e la scelta ideale è il punto medio $(inf + sup)/2$. Così facendo, se $|D| < (inf + sup)/2$ andrò a cercare a destra, altrimenti a sinistra, ma in quel caso allora avremo $|S| < (inf + sup)/2$. Morale: in entrambi i casi ho almeno dimezzato la dimensione della porzione da analizzare (curiosamente, in questo caso, il caso pessimo è il dimezzamento bilanciato, perchè andrò sempre a scegliere la porzione più piccola per cercare il minimo intero libero). Quindi la complessità dell'algoritmo è maggiorata da $n + n/2 + n/4 + \ldots = n + n(1/2 + 1/4 + \ldots)$. La serie tra parentesi lasciava perplesso Zenone di Elea riflettendo sul fatto se Achille avrebbe potuto raggiungere la tartaruga, ma noi sappiamo che converge a 1 e quindi tutto a $2n$ (ovviamente si può impostare una relazione di ricorrenza).

Tutto ciò conduce alla funzione in Fig. 5, che è un curioso miscuglio tra *quickSort* e ricerca binaria.

Addendum Un arguto studente ha osservato (dopo il compito) come sia possibile scrivere una funzione lineare *senza* allocare memoria e *senza* modificare il vettore. Usando in sostanza la tecnica che risolve il punto **2**, riuscendo a codificare l'informazione nel vettore di appoggio p dentro v in modo "reversibile": essendo il vettore v di interi non negativi (assumiamo per semplicità che lo 0 non sia in v, se c'è sommiamo 1 a tutti gli elementi di v), posso porre v_i a $-v_i$ per codificare il fatto di sapere che i è un elemento di v. A quel punto il minimo intero non presente, sarà l'indice del primo elemento di v positivo! Prima di tornare, risistemiamo le cose, riportando i valori negativi al loro valore assoluto (sottraendo 1 per invertire l'operazione necessaria a sbarazzarsi dell'eventuale 0 presente).

È ovviamente un trucco basato sull'abilità di codificare più "informazione" dentro v. Chiaramente questa possibilità è dovuta al fatto che usando gli `int` per memorizzare solo numeri positivi, "spreco" tutte le combinazioni di bit che codificano numeri negativi. Quindi, in realtà, "moralmente" questa soluzione "alloca" memoria, esattamente come la soluzione del punto **2**, aumentando la "quantità di informazione" memorizzata da v.

Potremmo applicarla anche se il vettore fosse di `unsigned int`, ma sempre a patto di sapere di avere dei valori "liberi". Per esempio, supponete che di usare 64 bit per memorizzare un intero senza segno e supponete di sapere che v contenga come valore massimo $2^{32} - 1$: moltiplicando tutti i numeri per 2 rimaniamo dentro l'intervallo $[0, 2^{64} - 1]$ e marchiamo il fatto che i sta in v_i sottraendo 1 a v_i: allora il minimo intero non presente, sarà l'indice del primo elemento di v pari! Ancora una volta, è facile risistemare le cose (in tempo lineare) prima di tornare al chiamante.

Beh, ora avete capito di cosa si interessa la Teoria dell'Informazione.