Un piccolo classico: minimo intero libero

corso di laurea in **Matematica**

Informatica Generale, Lezione **14(c)**

Ivano Salvo

SAPIENZA
UNIVERSITÀ DI ROMA

Problema: minimo intero libero

Problema: Immaginiamo di avere un vettore u di n **numeri naturali distinti**. Si chiede di trovare il **minimo naturale non presente** in u.

Esempio: consideriamo il vettore u = [3, 8, 4, 5, 11, 15, 9, 2, 6, 0, 1]. Il minimo intero libero è **7**.

➧ **Osservazione importante** (che sarà utile in varie soluzioni che presenterò): il minimo intero libero è necessariamente un numero **nell'intervallo chiuso [0, *n*]**.

Infatti, il vettore contiene n naturali, mentre l'intervallo $[0, n]$ ne contiene n+1. Per il **principio dei buchi di piccionaia** non è possibile che v contenga tutti i valori in $[0, n]$. Alla peggio, il minimo intero libero è quindi n.

Esercizio 1: **senza allocare memoria** e **senza modificare** v scrivere una funzione che risolve il problema.

"**Senza allocare memoria**" significa **senza avvalersi di strutture dati di dimensione proporzionale ad** n, dove n è la dimensione del problema (in questo caso la lunghezza di v).

Quindi potete usare una **quantità costante di memoria** ($\Theta(1)$, ma non $\Theta(n)$ oppure $\Theta(\log n)$).

Soluzioni quadratiche

Non **potendo allocare memoria** e **non potendo modificare il vettore**, possiamo **solo dare soluzioni quadratiche**, anche se si può fare seguendo molte strade.

La più semplice è **cercare nel vettore tutti i naturali in ordine**: 0, 1, 2, 3, …: il primo che non viene trovato è il minimo intero libero. È essenzialmente una ricerca di ricerche.

È facile usando una ricerca lineare…

In particolare, la "**ricerca** del numero assente" esterna, **terminerà** necessariamente dopo **al più *n* passi**, per l'osservazione vista prima.

… ma deliziosa **senza usare funzioni** e senza **cicli annidati** (liberamente ispirata a un compito di uno studente)

```
def minFree(v):
    i = 0
    while cerca(v, i) ≥ 0:
        i = i + 1
    return i
```

```
def minFree(v):
    i, j, n = 0, 0, len(v)
    while i ≤ n:
        if v[i] == j:
            i, j = 0, j+1
        else: i=i+1
    return i
```

Allocando memoria…

Allocando memoria, è possibile usare un'idea **analoga a `countingSort`** (in versione 0/1).

Scorrendo una sola volta il vettore v, si carica un **vettore** p che **codifica la presenza** o meno di ogni **intero in** $[0, n)$. Per quanto visto all'inizio, p deve avere dimensione n+1, circa come v.

Scorrendo il vettore v con una variabile j, possiamo mantenere l'invariante $p[i]=\text{TRUE} \Leftrightarrow i \in v[0, j)$.

Questo ci dice come inizializzare p: all'inizio tutti FALSE. Alla **fine** sarà soddisfatta l'asserzione $p[i]=\text{TRUE} \Leftrightarrow i \in v[0, n) \equiv i \in v$ e quindi il **minimo intero mancante** è $m = \mathbf{min}\{\, j \mid p[i] = \text{FALSE} \,\}$. Tanto il caricamento di p che la ricerca di m è $\Theta(n)$.

```
def minFreeMem(v):
    n = len(v)
    alloca(p, n)
    inizializza(p, False)
    for i=0 to n-1:
        if v[i] ≤ n: p[v[i]]=True
    return cerca(p, False)
```

È "sbagliato" fare i sofisticati e pensare di allocare max v elementi

È fondamentale non scrivere fuori dai limiti di un vettore

Modificando, ma non allocando

Potendo **modificare** il vettore (ma **non potendo allocare memoria**), una soluzione ovvia è ordinare v e poi cercare il primo elemento di v in cui $v[i]+1 < v[i]$, con qualche fastidio per trattare i casi che il minimo intero mancante sia proprio 0 oppure n.

La complessità è $\Theta(n \log n)$ [**ordinamento ottimo**] + $\Theta(n)$ [**ricerca**] = $\Theta(n \log n)$.

Qualche studente scrupoloso osservò che **`mergeSort` non va bene**, perché **`mergeSort`** **alloca memoria** per la fusione. E neanche **`countingSort`**, per lo stesso motivo.

Altri studenti scrupolosi proposero una **ricerca binaria**, ma non si cambia la complessità asintotica, dominata dall'ordinamento.

Tuttavia si può fare **molto meglio**.

A ben pensarci, è **una specie di 1-mediana sull'insieme $\mathbb{N} \setminus v$…**
… cioè sui mancanti.

Partiziona, mon amour

C'è una deliziosa soluzione **divide et impera**, che sfrutta ancora una volte le virtù di **`partiziona`**.

Partizioniamo v con rispetto al valore $n/2$: se il risultato è $m < n/2$ significa che il minimo intero libero sta **nella parte sinistra** (ho meno elementi di $n/2$ a sinistra) **altrimenti sta nella parte destra**.

Un bel mix tra ricerca binaria e quickSort. Siccome vado solo in una delle due metà, la complessità è $T(n) < T(n/2) + \theta(n)$, la cui soluzione **è lineare** (è n (1 + ½ + ¼ + …) cioè la corsa di Achille!

In questo caso, **le partizioni sbilanciate vanno** persino **bene**, perché vado a cercare sempre nella più piccola! **Notare la precondizione**!

Da cui il **caso base**.!

```
def minFree(u, inf, sup):
# REQ: x ∈ [inf, sup]
    if sup - inf = 1: # caso base
        if u[inf]>inf: return inf
            else: return inf+1
    p = puntoMedio(inf, sup)
    m = partiziona(u, inf, sup, p) # valore perno
    if m < p: return minFree(u, inf, p-1)
    return minFree(u, p, sup)
```

Finale a sorpresa

Un arguto studente (dopo il compito) osservò che era possibile codificare il vettore p in v, **senza** bisogno di **allocare memoria** e **modificando temporanemente** v.

Ecco l'idea: se $v[j] < n$ allora pongo $v[v[j]] = -v[v[j]]$.

L'**indice del primo valore positivo** in v sarà il **minimo intero mancante**. Una volta trovato, semplicemente si rimettono apposto i negativi e si fanno tornare positivi.

Gran bella idea. Ma **è vero che non allochiamo memoria**?

Strictu senso sì, ma sfruttiamo l'idea di aver sprecato bit, codificando **`meno informazione' di quanto possibile**, avendo utilizzato un tipo intero con negativi per codificare solo positivi.

La stessa idea funziona se sappiamo che usiamo i numeri da 0 a 2^{31} mentre noi usiamo 32 bit: è sufficiente porre a 1 il primo bit.

Morale: abbiamo "allocato" memoria perché **abbiamo codificato più informazione** (ecco di cosa parla la **Teoria dell'Informazione**)