Soluzioni 2 L'Angolo Degli Esercizi

corso di laurea in **Matematica**

Informatica Generale, **Ivano Salvo**

Esercizi lez. **7(c)** **[17/10/23]**

SAPIENZA
UNIVERSITÀ DI ROMA

Massimo Fattore Primo Ricorsivo

Esercizio: Massimo fattore primo

Supponiamo che il nostro esecutore sappia solo calcolare la funzione **`max`** (tra due numeri) e una funzione **`scomponi`**, con la seguente semantica:

"`scomponi(n)` restituisce sempre una coppia di numeri f_1 e f_2 tali che $f_1 \cdot f_2 = n$. Se n è **primo**, restituisce la coppia 1, n mentre se n è **composto** f_1 , $f_2 \neq 1$, ma non possiamo fare assunzioni su chi siano f_1 , f_2. Ad esempio `scomponi(24)` può ritornare una qualsiasi delle coppie (2, 12), (3, 8), (4, 6), (6, 4), (8, 3), (12, 2)."

- scrivere una funzione ricorsiva **`maxPrimo(n)`** che calcola il massimo fattore primo di n.
- **Corredare** il programma di opportune **asserzioni logiche**.
- Provare a scrivere una versione iterativa di `maxPrimo`. Argomentare sulle eventuali difficoltà.

Massimo fattore primo: Soluzione

Dobbiamo trovare una formulazione ricorsiva.

Chiaramente, se ***n* è primo**, ***n* = maxPrimo(*n*)**.

Quindi, i **numeri primi**, sono i nostri candidati **casi base**.

Se un numero n è composto, allora per ogni coppia p, q tale che $n = p \cdot q$, è chiaro che il massimo fattore di primo n sia…

… il massimo fattore primo di p o il massimo di q…

$$\begin{aligned} \text{mfp}(n) &= n && \text{se } n \text{ è primo} \\ \text{mfp}(n) &= \max(\text{mfp}(p), \text{mfp}(q)) && \text{se } n = p \cdot q \end{aligned}$$

Le occorrenze ricorsive di mfp si applicano ad argomenti più piccoli, se p, q ≠1

```
def maxPrimo(n):
    f1, f2 = scomponi(n)
    if f1==1 return f2
    return max(maxPrimo(f1), maxPrimo(f2))
```

Sequenza quasi Crescente

Problema 3: sequenza quasi crescente

Problema: una sequenza di un vettore $v[inf, sup)$ è **quasi crescente** se esiste **al più un indice** $j \in (inf, sup)$ tale che $v[j] < v[j-1]$.

Scrivere una funzione `k, l = maxSeqCresc(v)` che restituisce la **più lunga sequenza quasi crescente** di v, tornando l'**indice di inizio** k e la sua **lunghezza** l.

Idea (**sempre buona**): scrivere una funzione **`seqQC(v, i)`** che trova la lunghezza della sequenza quasi crescente a cominciare dall'indice i: è sufficiente trovare il secondo indice j per cui $v[j] > v[j-1]$.

A questo punto è sufficiente calcolare il **massimo di tutte le sequenze quasi crescenti**, calcolandone una per ogni indice j di v.

```
def seqQC(v, i):
    n, j, d = len(v), i+1, 0
    while d < 2 and j < n:
        if v[j]<v[j-1]: d = d+1
        if d<2: j = j+1
    return j-i
```

```
def maxSeqQC(v):
    n, j = len(v), 0
    b, l = 0, 0
    while j<n:
        lnew = seqQC(v, j)
        if lnew>l: l, b = lnew, j
        j = j+1
    return b, l
```

Seq. quasi crescente: miglioramenti

Chiaramente l'algoritmo visto ricomincia sempre daccapo ogni sequenza, e **ricalcola i suffissi di sequenze quasi crescenti già trovate**. Cosa fa, ad esempio, su un vettore ascendente? È $\mathcal{O}(n^2)$.

Per evitare ciò, **basta ripartire dal primo indice j in cui $v[j] > v[j\text{-}1]$**: questo può essere calcolato da **`seqQC`** e utilizzato da **`maxSeqQC`**.

Ogni sequenza crescente viene vista **al massimo due volte**, quindi otteniamo un algoritmo $\mathcal{O}(2n)$, mentre il problema è chiaramente $\mathbf{\Omega}(n)$: gli ottimizzatori però hanno **ancora spazio di manovra**!

```
def seqQCSmart(v, i):
    n, j, d = len(v), i+1, 0
    while d < 2 and j<n:
        if v[j]<v[j-1]:
            d=d+1
            if d==1: m=j
        if d<2: j = j+1
    if d<1: m=j--se Asc(v[i,n))
    return m, j-i
```

```
def maxSeqQCSmart(v):
    n, j = len(v), i+1, 0
    b, l = 0, 0
    while j<n:
        m, lnew = seqQCSmart(v, j)
        if lnew>l: l, b = lnew, j
        j = m
    return b, l
```

sequenza quasi crescente: esempi

Vediamo brevemente l'esecuzione dei due programmi su 3 casi, semplicemente tracciando le sequenze quasi crescenti considerate.

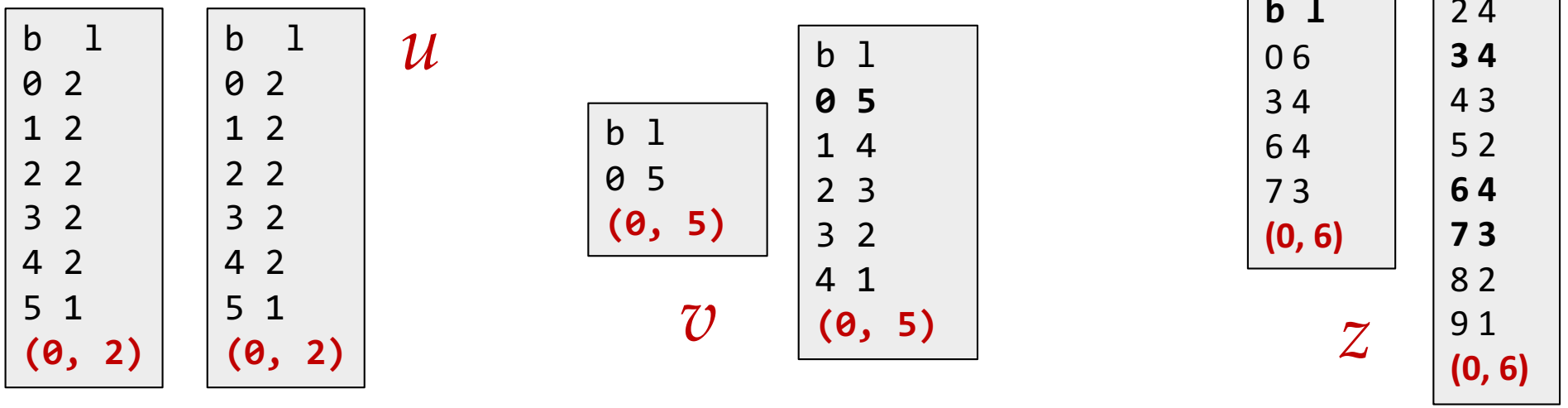
Consideriamo 3 vettori:

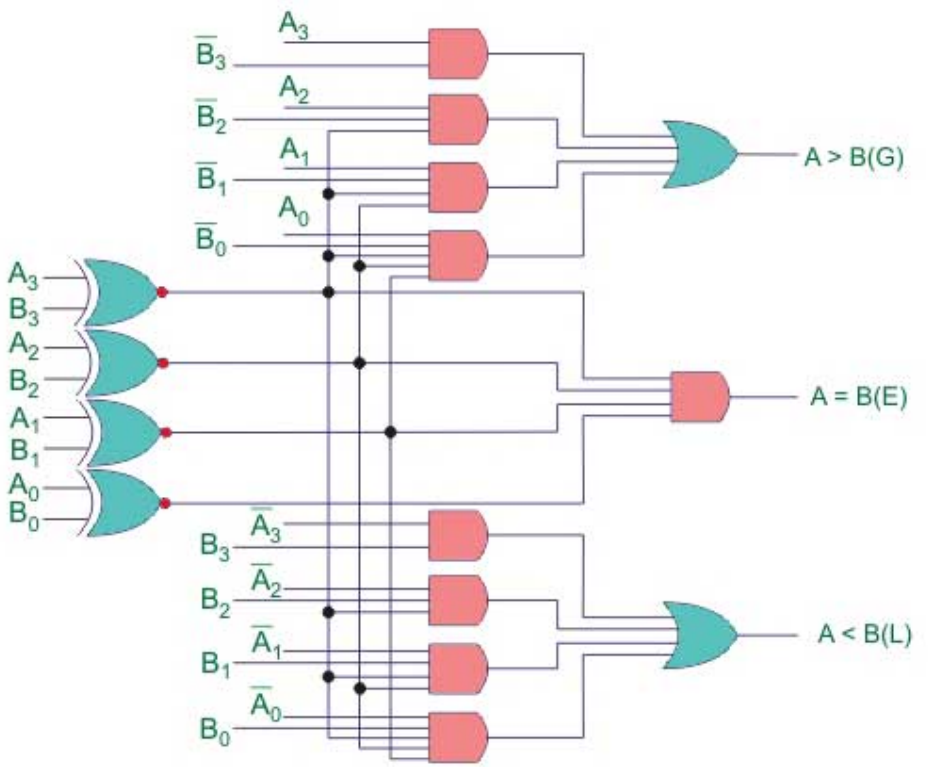
$u = \{6, 5, 4, 3, 2, 1\}$, $v = \{1, 2, 3, 4, 5\}$ e $z = \{\mathbf{1}, 2, 3, \mathbf{1}, 4, 6, \mathbf{2}, \mathbf{1}, 8, 9\}$

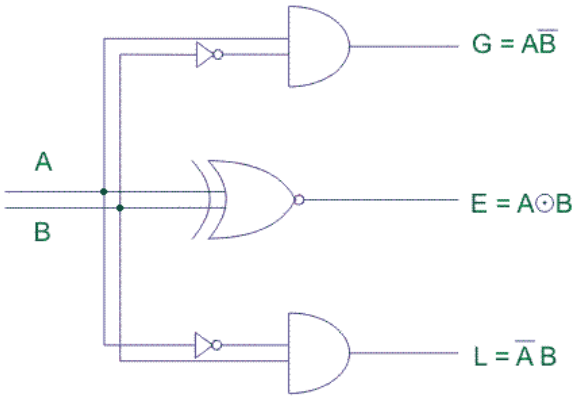
u è decrescente: le sequenze quasi crescenti sono tutti i segmenti di lunghezza 2. I due algoritmi hanno lo stesso comportamento.

v è crescente: il 2do algoritmo esamina solo l'unica sequenza crescente. Il primo vede anche tutti i suffissi.

Anche su z, il 2do algoritmo esamina un sottoinsieme di sequenze.







Comparatore binario

Problema 2: comparatore

Problema 2: Indichiamo con $\underline{a}$ l'intero è rappresentato da un **vettore** ***a*** di lunghezza k **di cifre binarie** con la cifra **più significativa** in posizione k - 1.

1. Scrivere una funzione `u, m = confronta(a, b, k)` che confronta **due vettori di cifre binarie** a e b, e ritorna **1, #** se $\underline{a} = \underline{b}$, **0, 1** se $\underline{a} > \underline{b}$, e **0, 0** se $\underline{a} < \underline{b}$.
2. Valutare la **complessità**, indicando il caso **ottimo** e **pessimo**.
3. ★Valutare la complessità del caso **medio**, assumendo le sequenze siano distribuite uniformemente (se uscisse una sommatoria, basta impostarla).

```
def confronta(a, b, k):
    for j = k-1 downto 0:
        if a[j]>b[j]: return 0,1
        if a[j]<b[j]: return 0,0
    return 1,42 -- 😉
```

Caso ottimo: $a[k\text{-}1] \neq b[k\text{-}1]$, si esce subito dal ciclo.

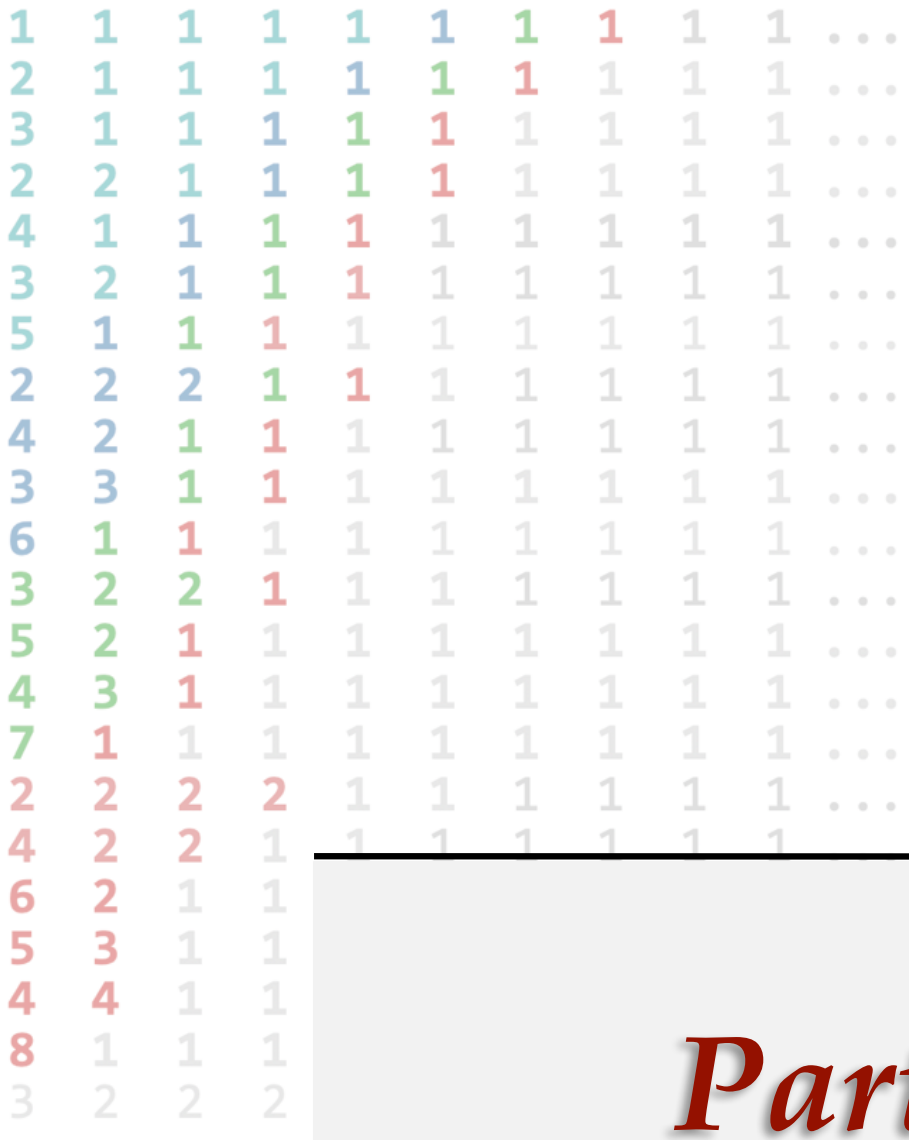
Caso pessimo: $\underline{a} = \underline{b}$ si fanno k iterazioni. Quindi $\mathcal{O}(k)$.

Caso medio: per ogni i, c'è la probabilità ½ che le due cifre binarie siano uguali e ½ che siano diverse. La probabilità di fare i cicli è quindi $1/2^i$. La complessità media è quindi $\sum_{i \in [0,\, k)} i \cdot 1/2^i \leq \sum_{i \in [0,\, \infty)} i \cdot 1/2^i =$

(Wikipedia, serie di potenze) $= \frac{1/2}{(1-1/2)^2} = 2 = \Theta(1)$.

$$\sum_{i=1}^{\infty} i x^i = \frac{x}{(1-x)^2}$$

Somma infinita (per $|x| < 1$)


1 1 1 1 1 1 1 1 1 1 ...
2 1 1 1 1 1 1 1 1 1 ...
3 1 1 1 1 1 1 1 1 1 ...
2 2 1 1 1 1 1 1 1 1 ...
4 1 1 1 1 1 1 1 1 1 ...
3 2 1 1 1 1 1 1 1 1 ...
5 1 1 1 1 1 1 1 1 1 ...
2 2 2 1 1 1 1 1 1 1 ...
4 2 1 1 1 1 1 1 1 1 ...
3 3 1 1 1 1 1 1 1 1 ...
6 1 1 1 1 1 1 1 1 1 ...
3 2 2 1 1 1 1 1 1 1 ...
5 2 1 1 1 1 1 1 1 1 ...
4 3 1 1 1 1 1 1 1 1 ...
7 1 1 1 1 1 1 1 1 1 ...
2 2 2 2 1 1 1 1 1 1 ...
4 2 2 1 1 1 1 1 1 1
6 2 1 1
5 3 1 1
4 4 1 1
8 1 1 1
3 2 2 2

Partizioni

Che fine hanno fatto le partizioni?

Ricordate il problema delle **partizioni di *n***? Vediamo le partizioni di 6 opportunamente ordinate (lessicograficamente) e colorate.

[1, 1, 1, 1, 1, 1], [1, 1, 1, 1, 2], [1, 1, 1, 3], [1, 1, 2, 2],
[1, 1, 4], [1, 2, 3], [1, 5], [2, 4], [3, 3], [6]

Le **verdi** sono 1 seguito dalle partizioni di 5. Ma le **blu** cosa sono?

Sono le partizioni di 6 che non usano 1! Da questo posso trarre un'**idea ricorsiva**.

- **Esercizio**: **generare le partizioni**, partendo dal programma ricorsivo sotto (la prima chiamata mette *k* nella partizione che si sta contando, l'altra niente).
- **Sfida**: scrivere la funzione **`prossimaPart`** (per **generarle in ordine** come fatto per permutazioni e combinazioni)

```
def part(n):
    return partAux(n, 1)
```

```
def partAux(n, k):
# conta partizioni di n che usano k come minimo numero
    if n==k: return 1 # c'è 1 partizione di n con min. k
    if n <k: return 0 # non c'è nessuna nessuna partizione
    return partAux(n-k, k) + partAux(n, k+1)
```

Conteggio partizioni iterativo

Scrivere un programma iterativo "**nativo**" per le partizioni potrebbe essere molto difficile.

Molto più facile, scrivere la versione "**bottom-up**" di quello ricorsivo. Aiutandosi con una matrice **p**, caricare nella matrice tutti i valori di **partAux(n, k)** in $p[n][k]$ partendo dai casi base ($n = k$) ed eseguire a ritroso i calcoli di **partAux** (o se volete, i calcoli che essa esegue "tornando" dalle chiamate ricorsive.

Questa strategia ha il vantaggio di **evitare di ricalcolare** più volte gli stessi valori come avviene nella funzione ricorsiva.

```
def partIter(n):
    p = allocMatZero(n+1, n+1)
    for i = 0 to n: p[i][i] = 1 #casi base
    for i = 1 to n:
        for j = i-1 downto 1: #all'indietro
            p[i][j] = p[i-j][j] + p[i][j+1]
    return p[n][1]
```

```
1,  0,  0, 0, 0, 0, 0, 0, 0, 0, 0
0,  1,  0, 0, 0, 0, 0, 0, 0, 0, 0
0,  2,  1, 0, 0, 0, 0, 0, 0, 0, 0
0,  3,  1, 1, 0, 0, 0, 0, 0, 0, 0
0,  5,  2, 1, 1, 0, 0, 0, 0, 0, 0
0,  7,  2, 1, 1, 1, 0, 0, 0, 0, 0
0, 11,  4, 2, 1, 1, 1, 0, 0, 0, 0
0, 15,  4, 2, 1, 1, 1, 1, 0, 0, 0
0, 22,  7, 3, 2, 1, 1, 1, 1, 0, 0
0, 30,  8, 4, 2, 1, 1, 1, 1, 1, 0
0, 42, 12, 5, 3, 2, 1, 1, 1, 1, 1
```

Generazione delle partizioni

Quando si generano strutture combinatorie, ho sempre almeno 3 strategie da seguire:

- **usare il programma che le conta** (ad esempio nelle partizioni, la chiamata ricorsiva `partAux(n-k, k)` significa che ho messo un *k* nella partizione che sto contando e `return 1` significa che la partizione in costruzione è completa;
- Stabilire **un ordine**, scrivere una funzione `nextPart(c)` che trova la prossima partizione ed eseguirla finché non si raggiunge l'ultima;
- **codificare** ciascuna partizione con un numero naturale, scrivere le funzioni di codifica/decodifica e semplicemente **stampare le decodifiche** dei naturali fino al numero di oggetti da generare.

That's all Folks!

corso di laurea in **Matematica**

Informatica Generale, **Ivano Salvo**

Esercizi lez. **7(a)** **[17/10/23]**

SAPIENZA
UNIVERSITÀ DI ROMA