# *Moderiamo le interazioni* 😏

Spegnete microfoni e telecamere.

Aprite il microfono per fare domande tra una parte e l'altra della lezione;

Mentre il docente parla, fate domande sulla chat. Io le ripeto ad alta voce e rispondo (se ne sono capace).

# *Informatica Generale*
# *Programmazione C*

*Ivano Salvo*

Corso di Laurea in Matematica

SAPIENZA
UNIVERSITÀ DI ROMA

Lezione **6**, 31 marzo 2020

# ...andare avanti... sempre!

la Repubblica Sabato, 28 marzo 2020 ROBINSON Resistere pagina 7

QUATTRO REGOLE DI AUTODISCIPLINA PER RESISTERE DIGNITOSAMENTE A QUESTO TEMPO.

Resistenza
di Zerocalcare

L'artista di Rebibbia già candidato al premio Strega, campione di vendite e vincitore dei massimi premi del fumetto attualmente sta lavorando a un nuovo graphic novel e a dei cartoni animati autoprodotti. Gli ultimi due sono sul tema coronavirus

① SOPRAVVIVI –

COSE DA FARE OGGI:
- MANGIA
- BEVI
- CESSO.

② NON DIVENTARE UN DELATORE PSICOPATICO.

PRONTO ESERCITO? HO VISTO IL MIO VICINO CHE USCIVA DI CASA HO POSTATO LA SUA FOTO SU FACEBOOK MA NON INTERVIENE NESSUNO.

DOVE SONO I CARRI ARMATI? PERCHÉ NESSUNO HA MONTATO UNA FORCA?

③ NON DIVENTARE UN CINICO MENEFREGHISTA

IO ESCO CON GLI AMICI CAZZO ME NE.

TANTO CREPANO I VECCHI

IO NON NE CONOSCO DI VECCHI

MI DAI LA PAGHETTA MA'?

DAJE FORTE.

④ NON ASSUEFARTI A QUALSIASI COSA.

TRACCIATEMI PURE TUTTO, TELEFONO, BRACCIALETTO ELETTRONICO, CHIP SOTTO PELLE COME I CANI, COME VE PARE.

TANTO DO' CAZZO DEVO ANNA'.

OK, MO' È EMERGENZA, PERÓ 'STA ROBA DEL CONTROLLO SOCIALE POI TOCCHERÀ TEMATIZZARLA.

VE PREGO, NON USCIMO DA 'STA QUARANTENA COME PERSONE (MOLTO) PIÙ ORRENDE DI QUELLO CHE GIÀ SIAMO. CERCHIAMO DI RIMANERE UN'ALTERNATIVA PREFERIBILE AL CORONAVIRUS.

# *Lezione 6a:*

# *Altri vettori in C*

# *Il problema del minimo intero libero*

## 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.*

# Possibili soluzioni: idee

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.

[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.

# Soluzione con ricerca iterata

```
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.

È sempre una buona idea definire **funzioni ausiliarie** (che risolvono sottoproblemi). In questo caso, `trova` ha un prototipo molto generale e restituisce anche l'indice in cui si trova l'elemento `x`. Noi non usiamo questa informazione.

# *Soluzione da Vero Programmatore C*

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

Il Vero Programmatore C, ovviamente passa il vettore come fosse un puntatore, e poi trova il modo di fare **tutto nella guardia del `while`**! L'esecuzione di `trova(++i, v, n, &p)` restituisce 1 se i è presente, 0 altrimenti.

Esco dal `while` quando sono al primo i che non è presente nel vettore. E quindi lo ritorno come risultato.

# *Imparando dagli studenti… (1)*

È sempre bello correggere soluzioni ingegnose a cui non avevo pensato. Eccone un paio.

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.

```
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;
   }
```

Osservate che se un certo *j* non compare nel vettore, si arriva in fondo e si esce perché *i*==*n*.

# *Imparando dagli studenti… (2)*

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`.

```
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;
}
```

Non appena c'è un buco, cresce i ma non min.

# *Allocando memoria...*

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

Per risolvere questo esercizio dobbiamo introdurre un nuovo concetto: i **vettori variabili**. E' possibile, dentro una funzione dichiarare un vettore con **un numero di elementi** che **dipende** ad esempio dal **parametro** di una funzione.

Non è C standard, ma ormai accettato da tutti i principali compilatori C.

**Attenzione**: tale vettore viene **allocato nel record di attivazione della funzione** e quindi de-allocato quando la funzione smette di eseguire!

# Soluzione lineare con vettore

Possiamo allocare un vettore $p$ di lunghezza $n$ e codificare con: $\begin{cases} p[i] = 1 \text{ se } i \text{ occorre in } v \\ p[i] = 0 \text{ se } i \text{ non occorre in } v \end{cases}$

Prima di cominciare non sappiamo nulla, e quindi il vettore $p$ va azzerato. Poi scorriamo $v$ e mettiamo $p[v[i]]$ a 1 per ogni $i$.

Infine, scorriamo $p$ e il primo indice i tale che $p[i]$=0 è il risultato cercato.

```
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;
}
```

*Attenzione a non uscire dai limiti degli indici di p*

Figura 4: Funzione `minFreeLin`

# Alcune osservazioni

*Limitazioni sui vettori variabili*

*Sempre meglio scomporre in funzioni*

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 $\{0, 1, 2, 3, 2^{64} - 1\}$?

*Sappiamo che il risultato è in [0,n]*

# Potendo modificare il vettore?

**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**.

# Soluzione Divide et Impera

**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* $[inf, sup]$.

[redacted] 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 $m \in [inf + s, sup]$. [redacted]

# Soluzione Divide et Impera

## Scelta del Perno:

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

## 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 $m = inf$.

# Soluzione Divide et Impera

```
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*

# Soluzione Divide et Impera

**Complessità:**

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).

**Ma dov'è il trucco?**

# *Lezione 6b:*

# *Allocazione di Memoria*

# *Generiamo un vettore…*

Immaginiamo di voler risolvere il seguente problema:

*"Scrivete una funzione C di prototipo*

```
int* generaPrimi(int n, int* k)
```

*che genera il vettore di tutti i numeri primi minori di n, e carica nell'intero *k il numero dei primi trovati"*

Potremmo essere tentati di scrivere qualcosa del tipo:

```
int* generaPrimi(int n, int* k){
   int v[n]; /* n è un po' troppo in effetti ☺ */
   … /* codice che carica v coi primi */
   return v;
}
```

Tuttavia sarebbe un **grave errore**. Il vettore v **è locale** a generaPrimi e viene **deallocato** quando la funzione termina.

# *Allocazione di memoria*

In C è possibile **allocare memoria dinamicamente**, durante l'esecuzione del programma. Tale memoria risiede in una zona di memoria della macchina **diversa dallo stack** di attivazione delle chiamate di funzione, detta **Heap**.

L'allocazione di memoria avviene attraverso chiamate a funzioni della libreria <stdlib.h>. Le più usate sono:

```
void* malloc(int n)
void* calloc(int k, int n)
```

Entrambe allocano un blocco di memoria di un certo numero di bytes (`n` nel caso di `malloc` e `k×n` nel caso di `calloc`) e **ritornano un puntatore** alla base di tale blocco.

Osservate che il tipo di ritorno è **`void *`**: si tratta di un curioso tipo puntatore che **è compatibile per assegnazione con tutti gli altri tipi puntatore**. Ciò è necessario affinché queste funzioni siano generali e possano tornare pointer di ogni tipo.

# Uso di malloc e calloc

Se voglio allocare memoria per un singolo intero dovrò eseguire un'istruzione tipo:

```
int* m = (int *) malloc (sizeof(int))
```

Osserviamo un po' di cose:

`(int *)` si chiama **coercion**: `malloc` torna un `void*` ma io lo assegno a una variabile `int *`: in C classico questo è assolutamente normale (un puntatore è sempre un puntatore), tuttavia i compilatori moderni tendono ad avvertire il programmatore che sta facendo qualcosa di strano. Mettendo la coercion, "tranquillizziamo" il compilatore.

`sizeof` è una pseudofunzione che **torna il numero di byte necessario per memorizzare un dato di un certo tipo**. **Evitate sempre chiamate tipo: `malloc(42)`**, ma usate sempre `sizeof`: questo rende i programmi correttamente eseguibili su macchine diverse che hanno diverse dimensioni dei dati.

# *Allocazione di un vettore dinamico*

Un vettore contenente `n` elementi di un certo tipo `T`, può essere allocato usando indifferentemente una delle seguenti istruzioni:

```
T* v = (T *) malloc (n*sizeof(T))
T* v = (T *) calloc (n, sizeof(T))
```

L'unica differenza è che **`calloc` azzera la memoria allocata**. Era quindi poco amata dai Veri Programmatori C, in quanto "più lenta": al giorno d'oggi tutto ciò è (praticamente) ininfluente sui tempi di calcolo.

A questo punto, possiamo usare `v` come un qualsiasi vettore, e possiamo anche usare l'operatore `[]` per accedere ai singoli elementi.

# *Esempio: Crivello di Eratostene*

Scriviamo ora la funzione `int* generaPrimi(int n, int* k)` usando l'algoritmo del crivello di Eratostene (nella dispesa **D4** questo algoritmo è presentato con vettori statici).

Esiste un antichissimo metodo (forse uno dei primi algoritmi di cui si abbia conoscenza) per generare tutti i numeri primi da 1 ad $n$, noto come *Crivello di Eratostene*, che risale al III secolo avanti Cristo. Il metodo si può scrivere informalmente come segue:

> *"si scrivono tutti i numeri naturali da 1 a n. Si comincia da 2 e si cancellano tutti i suoi multipli 4, 6, 8, 10, ... fino a n. Si prende il primo numero non cancellato, il 3, e si cancellano tutti i suoi multipli 6, 9, 12, 15, ... Il prossimo numero non cancellato ora è il 5 e si cancellano i suoi multipli, e così via. Alla fine, seguendo questo procedimento, i numeri non cancellati rimasti sono tutti i numeri primi tra 1 e n."*

# *Esempio: Crivello di Eratostene*

Useremo un vettore $p$ con $n$ elementi con l'idea che $p[i]=1$ se $p[i]$ è primo e 0 altrimenti. Dopodiché useremo $p$ per costruire il vettore di primi. Assumendo che il vettore p sia inizializzato con tutti 0, possiamo usare la funzione che trovate nelle dispense per implementare l'algoritmo di Eratostene.

```
void crivello(int p[], int n){
 /* PREC: forall i in [0,n). p[i]=1
  * POST: forall i in [0,n).
              p[i]<=>primo(i)
  */

  for (int i=2; i*i<n; i++)
     if (p[i]) {
       for (int j=i*i; j<=n; j+=2*i)
          p[j]=0;
       }
  }
```

# *Generare i Primi*

Osservate che uso un **vettore variabile** *p* (**non servirà più finita la funzione**) mentre **allochiamo dinamicamente** il vettore dei primi che **devo restituire come risultato**.

Devo contare i primi per allocare v e caricare il loro numero in `*k`

```
int* generaPrimi(int n, int *k){
    int p[n+1];
    inizializza(p,n+1,1);
        /* mette tutti gli elementi a 1 */
    crivello(p,n+1);
    *k = contaPrimi(p, n+1);
        /* conto gli 1 in p, per allocare v */
    int* v = calloc(*k, sizeof(int));
    caricaPrimi(v, p, n);
    return v;
}
```

```
void caricaPrimi(int v[], int p[], int n){
    int j=0;
    for (int i=2; i<n; i++)
        if (p[i]) v[j++]=i;
}
```

# *Osservazione importante*

Il vettore *p*, viene deallocato quando finisce di eseguire generaPrimi.

Se avessimo allocato *p* usando malloc o calloc **sarebbe rimasta impegnata la memoria occupata da *p***, senza peraltro poterlo usare nuovamente, in quanto verrebbe **perso il riferimento a *p*** (locale alla funzione generaPrimi).

Tale memoria sarebbe diventata **garbage** (cioè spazzatura).

Attenzione, quindi. Esiste la funzione **`free(void *)`** per liberare la memoria allocata con malloc o calloc.

```
int* generaPrimi(int n, int *k){
    int p[n+1];
    inizializza(p,n+1,1);
        /* mette tutti gli elementi a 1 */
    crivello(p,n+1);
    *k = contaPrimi(p, n+1);
        /* conto gli 1 in p, per allocare v */
    int* v = calloc(*k, sizeof(int));
    caricaPrimi(v, p, n);
    return v;
}
```

# *Lezione 6c:*

# *Un aiutino sull'Homework*

# *Aiutino sull'Homework: partizioni*

[Homework **2012**, dispensa **S1**]

**Esercizio 3** Dato un intero strettamente positivo $n$, definiamo *partizione* di $n$ una sequenza di valori strettamente positivi $m_1 \leq m_2 \leq \ldots \leq m_k$ tale che $\sum_{i=1}^{k} m_i = n$.

Scrivere un programma C che letto in input un numero intero positivo $n$ calcoli il numero $p(n)$ delle partizioni di $n$. Attenzione, che per le definizioni date, due sequenze di valori che differiscono solo per l'ordine sono la stessa partizione.

ESEMPIO: Le partizioni del numero 5 sono i multiinsiemi $[5], [1,4], [2,3], [1,1,3], [1,2,2], [1,1,1,2], [1,1,1,1,1]$. Pertanto, sull'input 5 il vostro programma dovrebbe rispondere 7.

# Soluzione ricorsiva

[Homework **2012**, dispensa **S1**]

Una efficace idea ricorsiva per determinare il numero delle partizioni di un numero $n$ consiste nel considerare che le partizioni di $n$ possono essere divise ad esempio tra quelle che contengono l'1 come minimo numero e quelle che non lo contengono. Più in generale conviene considerare la funzione $\mathsf{part}(n, k)$ che restituisce il numero di partizioni che si possono fare di $n$ usando $k$ come minimo numero. A quel punto, dovrebbe essere chiara la relazione: $\mathsf{part}(n, k) = \mathsf{part}(n-k, k)+\mathsf{part}(n, k+1)$. Questa relazione dà immediatamente un programma ricorsivo. La terminazione è data dal fatto che la quantità $n - k$ decresce ad ogni chiamata ricorsiva, mentre i casi base sono dati da $n = k$ (nel qual caso il numero di partizioni possibili è 1) e $n < k$ (nel qual caso il numero di partizioni possibili è 0). Ecco quindi il programma:

*Uso k nella partizione, mi resta da partizionare n-k*

*Non uso k, devo partizionare n, usandok+1 come minimo numero*

# *Soluzione ricorsiva*

[Homework **2012**, dispensa **S1**]

```
int partAux(int n, int k){
     if (n<k) return 0;
     if (n==k) return 1;
     return partAux(n-k, k)+partAux(n, k+1);
}

int part(int n){
     return partAux(n,1);
}
```

**Esercizio 1**: provate a usare la logica di questo programma per stampare tutte le partizioni di un numero.

**Esercizio 2**: calcolate quante chiamate ricorsive occorrono. Scrivere l'albero delle chiamate ricorsive.

# *Lezione 6*

# *That's all Folks…*

# *…Domande?*