Informatica per Statistica
Riassunto della lezione del 09/11/2012

Igor Melatti

Alcuni algoritmi per matrici

- Una *matrice* è una struttura dati in grado di memorizzare una matrice con m righe ed n colonne in $I^{m \times n}$, dove I è un opportuno insieme (supporremo nel seguito che sia numerico, ma non è necessario)
- Viene rappresentata in un'opportuna area di memoria, organizzata tipicamente concatenando una dopo l'altra le righe che compongono la matrice stessa
  - ad esempio, la matrice $M = \begin{pmatrix} 3 & 4 & 5 \\ 13 & 14 & 15 \end{pmatrix}$ è rappresentata come una sequenza $\langle 3, 4, 5, 13, 14, 15 \rangle$
  - per sapere dove inizia ogni riga, e dove finisce l'intera matrice, ci sono 2 informazioni ulteriori: cols$[M]$ e rows$[M]$
  - per accedere all'elemento che si trova sulla colonna j della riga i di M, si scrive $M[i][j]$ (in matematica, si scrive usualmente $M_{i.j}$)
- Problema della somma di matrici (ad esempio, di interi)

  Input: due matrici A e B in $\mathbb{Z}^{m \times n}$

  Output: una matrice C t.c. $C_{i,j} = A_{i,j} + B_{i,j}$ per ogni $i \in [1, m]$ e $j \in [1, n]$
- Dimensione dell'input: numero di righe m e numero di colonne n
- L'algoritmo che risolve il problema dato è in Figura 1
  - come prima cosa, controlla che le dimensioni siano corrette (si possono sommare due matrici solo se le dimensioni coincidono)
  - **esercizio**: scrivere un unico `if` e un unico `return` per la condizione di errore
  - dopodiché scorre per intero sia A che B, costruendo di volta in volta il risultato C

```
1 SommaMatrici(A, B)
2   if (rows[A] ≠ rows[B])
3     return DIM_ERROR
4   if (cols[A] ≠ cols[B])
5     return DIM_ERROR
6   for i ← 1 to rows[A]
7     for j ← 1 to cols[A]
8       C[i][j] ← A[i][j] + B[i][j]
9   return C
```

Figure 1: Somma tra matrici

  - per farlo, occorrono due cicli: uno su i che individua le righe, e uno su j che all'interno della riga i individua una colonna

- La complessità è ovviamente $\Theta(nm)$, risultante dal doppio ciclo for (uno di lunghezza n, l'altro di lunghezza m)
  - da notare che non ci sono casi migliori o peggiori (a meno di considerare un input con due matrici non delle stesse dimensioni...)
- La Figura 2 mostra *un* modo di realizzare in C le matrici
  - molto simile agli array, solo che ci sono due dimensioni
  - l'inizializzazione può essere fatta staticamente indicando i valori sulle varie righe della matrice (linee 17 e 18 in Figura 2)
  - **esercizio**: scrivere il corpo della funzione `stampa_matrice`, che stampa sul monitor i valori della matrice (una linea di schermo per ogni riga della matrice)
  - **esercizio**: scrivere pseudocodice e codice di una funzione che cerca un elemento all'interno di una matrice, e ritorna la riga sulla quale lo trova, o -1 altrimenti
  - **esercizio**: scrivere pseudocodice e codice di una funzione che cerca un elemento all'interno di una matrice, e ritorna la colonna sulla quale lo trova, o -1 altrimenti
  - **esercizio (difficile)**: scrivere pseudocodice e codice di una funzione che cerca un elemento all'interno di una matrice, e ritorna la riga e la colonna sulla quale lo trova, o -1, -1 altrimenti
    - usare due variabili globali per ritornare il risultato
  - **esercizio (difficile)**: scrivere pseudocodice e codice di una funzione che calcola il prodotto di matrici

```c
#define NUM_COLS 3
#define NUM_ROWS 2

void SommaMatrici(int A[NUM_ROWS][NUM_COLS], int B[NUM_ROWS][NUM_COLS],
                  int C[NUM_ROWS][NUM_COLS])
{
  unsigned i, j;

  for (i = 0; i < NUM_ROWS; i++) {
    for (j = 0; j < NUM_COLS; j++)
      C[i][j] = A[i][j] + B[i][j];
  }
}

int main()
{
  int A[NUM_ROWS][NUM_COLS] = {{2, 3, 4}, {12, 13, 14}};
  int B[NUM_ROWS][NUM_COLS] = {{22, 23, 24}, {32, 33, 34}};
  int C[NUM_ROWS][NUM_COLS];
  SommaMatrici(A, B, C);
  stampa_matrice(C);
}
```

Figure 2: Operazioni su una matrice

I limiti dei tipi

- Si consideri la funzione C alla Figura 3

```
1 unsigned long fattoriale(unsigned n)
2 {
3   unsigned long res = 1;
4   unsigned i;
5
6   for (i = 1; i <= n; i++)
7     res *= i; /* esattamente la stessa cosa che scrivere res = res*i */
8   return res;
9 }
```

Figure 3: Fattoriale iterativo

- Notare che il risultato è un `unsigned long`, ovvero ci si aspetta che il risultato potrebbe essere troppo grande per essere memorizzato su un `unsigned` "normale"
- Tuttavia, su molte macchine (almeno su quelle cosiddette "a 32 bit"), `unsigned long` ed `unsigned` sono uguali, ovvero entrambi a 32 bit (4 bytes)
- Sulle macchine più potenti, "a 64 bit", gli `unsigned long` (così come gli `int long` o `long`) sono invece a 8 bytes, ovvero 64 bit, quindi effettivamente averlo dichiarato come `long` aiuta
  - nota: per gli interi, il C non dice quanti bytes devono usare i compilatori, ma solo che i `long` devono essere almeno tanto grandi quanto i non-`long` (che a loro volta devono essere almeno tanto grandi quanto gli `short`)
  - per sapere quanti bytes un compilatore alloca per un certo tipo, si può invocare l'operatore `sizeof`
  - si può scrivere sia `sizeof(int)` o `sizeof(long)` che `sizeof(v)`, con `v` variabile
  - per i `float` e i `double`, invece, c'è lo standard IEEE 754: 32 bits per i primi, 64 per i secondi su (quasi?) tutti i compilatori
- In realtà, occorrerebbe dichiarare `res` come `double`: così facendo, si sacrifica un po' di precisione ma si può avere un'idea degli ordini di grandezza
- Una versione con `unsigned res` sarebbe giusta fino a 12!; una con `unsigned long res` è giusta fino a 20!, una con `float res` va approssimativamente bene fino a 34!, una con `double res` va approssimativamente bene fino a 170!

- Si può scrivere un programma per vedere effettivamente tutto ciò
- Punto di partenza: il numero di bits necessari per memorizzare un numero N è $\lfloor \log_2(N) \rfloor + 1$
  - numero di cifre decimali: $\lfloor \log_{10}(N) \rfloor + 1$
  - numero di cifre in una qualche base b: $\lfloor \log_b(N) \rfloor + 1$
- Se $N = n!$, allora si ha che il numero di bits necessari è $\lfloor \log_2(n!) \rfloor + 1 = \lfloor \log_2(\prod_{i=1}^{n} i) \rfloor + 1 = \lfloor \sum_{i=1}^{n} \log_2(i) \rfloor + 1$, dove l'ultimo passaggio vale per le proprietà dei logaritmi
  - $\log(ab) = \log(a) + \log(b)$
- Esiste, in C, una libreria di funzioni matematiche (accessibile con `#include <math.h>`, ma poi occorre compilare aggiungendo al `gcc` l'opzione `-lm`)
- Tra le varie funzioni in `math.h`, c'è il logaritmo, ma solo in base e (*logaritmo naturale*)
  - quindi la funzione `log(d)` prende in input un `double d` e ritorna un `double l` t.c. `l` è il logaritmo naturale di `d`
- Altra proprietà dei logaritmi: $\log_a(b) = \frac{\log_c(b)}{\log_c(a)}$; basta usare questa formula con $c = e$
- Si può quindi scrivere un programma come `fattoriale.c`
- **Esercizio**: modificare `fattoriale.c` in modo che avvisi che si è superato il limite nelle varie funzioni
  - ad esempio, se si chiama `ul_fattoriale(13)` su una macchina a 32 bits, deve ritornare un errore (è accettabile che ritorni 0)
  - suggerimento: controllare che il numero di bits richiesti sia sufficiente usando `sizeof`