Erano poi così difficili questi Homework?

Ivano Salvo
Sapienza Università di Roma

Anno Accademico 2011-12

1 HW1.3: Partizioni

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:

```
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);
}
```

Il programma scritto sopra, benchè assolutamente corretto dal punto di vista della soluzione all'homework, soffre di una certa inefficienza dovuta al fatto che va spesso a ricalcolare valori già calcolati, allo stesso modo dei programmi ricorsivi che calcolano i numeri di fibonacci o i coefficienti binomiali. Per risolvere questa inefficienza, sono possibili varie strade, che comportano un investimento in memoria.

Una strada standard che permette di mantenere la semplice idea ricorsiva è quella della cosidetta *memoization*, che consiste nel salvarsi i risultati già calcolati ed evitare quindi di andare a ricalcolarli quando non necessario.

Ad esempio, è possibile allocare una matrice $n \times n$ e salvare i risultati per poi ripescarli. Ecco una semplice soluzione naïf (qui si fa vedere come si può usare un

vettore come una matrice, e si scopre perchè il C vuole il numero di colonne quando si passa una matrice come parametro, nel nostro caso m):

```
int partEff(int n, int k, int r[], int m){
     int p;
     int i;

     if (n<k) return 0;
     if (r[(n-1)*m+k]!=0) return r[(n-1)*m+k];
     if (n==k) {
          r[(n-1)*m+k]=1;
          return 1;
          }
     p=partEff(n-k, k,r,m)+partEff(n, k+1,r,m);
     r[(n-1)*m+k]=p;
     return p;
}
```

A questo punto, la strada per la scrittura di un programma completamente iterativo è aperta, e consiste nel calcolare direttamente la matrice `r[n][k]` nella parte in cui $k \leq n$. L'unica difficoltà consiste nel trovare l'ordine giusto, per evitare di andare a calcolare $\mathsf{part}(n, k)$ prima di aver calcolato le informazioni necessarie.

```
int partIter(int n){
     int p1,p2;
     int i,j,k;
     int p[n+1][n+1];

     for (i=0; i<=n; i++)
         for (j=0; j<=n; j++) p[i][j]=0;
        /* carico con 1 la diagonale principale */
     for (i=1; i<=n; i++) p[i][i]=1;
        /* procedo per diagonali discendenti */
     for (k=2; k<=n; k++){
         i=k; j=1;
         while (i<=n) {
             if (i-j<j) p1=0; else p1=p[i-j][j];
             if (i<j+1) p2=0; else p2=p[i][j+1];
             p[i][j]=p1+p2;
             i++;j++;
         }
     }
     return p[n][1];
}
```

Ovviamente tutto ciò richiede allocare una matrice di dimensione n^2. Ulteriori ottimizzazioni sono attese!

2 HW2.1: Mastemind

Ho sempre trovato questo esercizio più insidioso di quanto non sembri a prima vista. La difficoltà consiste nel rendersi conto che essendo ammesse ripetizioni di elementi nel codice da indovinare e nel tentativo, occorre evitare di "ricontare" due volte sia lo stesso elemento del codice, sia lo stesso elemento del tentativo.

Marcare gli elementi già usati si può fare in molti modi diversi. È meglio usare un vettore ausiliario perchè usualmente il programmino che calcola i risultati di un tentativo è semplicemente un sottoprogramma del programma che interattivamente gioca a mastermind con voi, e quindi non si devono modificare i vettori originari. Tuttavia, ai fini della soluzione dell'esercizio proposto nell'homework, ciò non fa differenza, e anche marcare i numeri già usati sostituendoli con valori "impossibili" (tipo numeri negativi o maggiori di 10) è una soluzione pienamente legittima. Nella soluzione da me proposta verrà usato un vettore ausiliario `gu` (già usato) i cui elementi saranno caricati con i valori 1 e 0.

La prima funzione calcola gli elementi *giusti al posto giusto* (ripulendo correttamente il vettore `gu`), mentre la seconda calcola gli elementi *giusti al posto sbagliato*. Dato che gli elementi del codice vengono analizzati uno sola volta, e si esce subito dal ciclo qualora essi producano un risultato (grazie al `break`), non occorre marcarli.

```
int gpg(int c[], int t[], int gu[]){
   int s=0;
   int i;

   for (i=0; i<NCIFRE; i++)
   if (t[i]==c[i]){
      s++;
      gu[i]=1;
   } else gu[i]=0;
   return s;
}

int gps(int c[], int t[], int gu[]){
   int i,j;
   int b=0;

   for (i=0; i<NCIFRE; i++)
      if (gu[i]!=1)
         for (j=0; j<NCIFRE; j++)
            if (t[j]==c[i] && i!=j && !gu[j]){
               b++;
               gu[j]=1;
               break;
            }
   return b;
}
```

3 HW2.2: Lucchetto

Posto che la parte più difficile si è rivelata l'acquisizione dell'input... ecco comunque un algoritmo per determinare il lucchetto, una volta ottenute correttamente le parole in input. Siccome devo trovare il lucchetto massimale, comincio a cercarlo dalla prima lettera della prima parola $a[0]$. Per ogni lettera $a[i]$ della prima parola verifico se la sequenza che comincia in $a[i]$ è uguale alla sequenza iniziale della seconda parola b. Questo viene fatto con un ciclo con una variabile contatore `k`, all'interno del quale è sempre verificato l'invariante $\forall j : 0 \leq j < k.a[i+j] = b[j]$. Scopro di avere effettivamente trovato un suffisso della prima parola uguale ad un prefisso della seconda solo se sono arrivato alla fine della prima parola.

```
void inputw(char a[], int* m){
   int j=0;

   while (1) {
       scanf("\n%c",&a[j]);
       if (a[j]=='*') break;
       j++;
       }
   *m=j;
}

void copia(char a[], int n, char b[]){
   int i;
   for (i=0; i<n; i++) b[i]=a[i];
}

void lucchetto(char a[], int m, char b[], int n, char c[], int* p){
   int i, j, k;
   for (i=0; i<m; i++) {
       for (k=0; a[i+k]==b[k] && i+k<m && k<n ; k++);
          /* esco non appena ho trovato un lucchetto */
       if (i+k==m) break;
    }
       /* verifico se non ho trovato nessun lucchetto */
    if (i==m) k=0;
       /* costruisco la parola risultato */
    copia(a, i, c);
    copia(&b[k], n-k, &c[i]);
       /* determino la lunghezza della parola risultato */
    *p=i+n-k;
}
```

Con questo armamentario di funzioni il programma principale risulta alquanto semplice:

```
int main(){
   int m, n, p;
   char w[100];
   char y[100];
   char z[100];

   inputw(w, &m);
   inputw(y, &n);
   lucchetto(w, m, y, n, z, &p);
   printw(z, p);
   printf("\n");
   return 0;
}
```

4 HW3.1: Cammini Minimi del Cavallo

La soluzione corretta a questo problema segue la seguente procedura: prima si determinano *tutte* le caselle distanti 1, poi *tutte* le caselle distanti 2 (che sono ovviamente quelle raggiungibili in un passo dalle caselle distanti 1) e così via finchè non ci sono nuove caselle raggiungibili (questa condizione si può facilmente verificare vedendo se in una certa passata non sono state trovate nuove caselle).

Viceversa, soluzioni ricorsive che vanno *in profondità* (cioè vanno a calcolare le caselle distanti $n + 1$ prima di aver calcolato tutte quelle distanti n) o producono risultati scorretti, oppure necessitano di precise cure (sostituire il valore in una casella se si transita nuovamente con una distanza minore di quella già calcolata) che ne rendono la programmazione problematica. Una soluzione alla portata di tutti può essere la seguente:

```
void camminiMinimiCavallo(int r, int c, int  n) {
   int i, j, p, f;

   for (i = 0; i < n; i++)
       for (j = 0; j <  n; j++) S[i][j] = -1;

   S[r][c] = 0;
   p = 0;
   f = 1;
   while (f) {
       f=0;
       for (i = 0; i < n; i++)
           for (j = 0; j < n; j++)
               if (S[i][j] == p)
                  if (prossimeMosseCavallo(i, j, n, p+1)) f=1;
   p++;
   } /* end while */
}
```

Ad ogni iterazione del `while` si determina l'insieme di tutte le caselle distanti $p+1$ cercando prima nella matrice le caselle già marcate distanti p e andando a calcolare le successive con la funzione `prossimeMosseCavallo` (Fig. 1) che restituirà 1 se ha trovato almeno una nuova casella libera. Di conseguenza, la variabile `f` assumerà il valore 0 ogni qualvolta alla fine di una iterazione del `while` esterno, nessuna nuova casella raggiungibile è stata trovata.

La soluzione appena vista, però, non brilla nè per efficienza, nè per eleganza. Per quanto riguarda l'efficienza, il suo principale problema è che scorre l'intera matrice ad ogni iterazione del ciclo `while` (che viene eseguito tante volte quant'è la distanza massima dalla casella di partenza del cavallo). È più opportuno usare una coda in cui inserire inizialmente la casella di partenza, poi le nuove caselle raggiunte. Ad ogni passo si etrae una casella dalla coda e si inseriscono le nuove caselle raggiunte da quella casella. Tra l'altro, all'interno della coda, le caselle saranno sempre parzialmente ordinate rispetto alla distanza dalla casella iniziale.

Per quanto riguarda l'eleganza, è possibile compattare enormemente il codice della funzione `prossimeMosseCavallo`, costruendosi a priori due vettori di 8 elementi che rappresentano le possibili nuove posizioni (ovviamente relativamente a una casella corrente). Vediamo questa nuova soluzione. Viene usata una coda, ma volendo fare le cose con le mani si poteva usare un array (ogni casella può essere inserita una sola volta nella coda, per cui l'array può essere sovradimensionato con il numero delle caselle della scacchiera). Osservate che ora, la funzione che cerca le nuove caselle raggiungibili (`nextMoves`), consiste ora solo di un ciclo `for`.

```
typedef struct P{
   int x;
   int y;
} pos;

int dirx[8]={-2,-1, 1, 2, 2, 1,-1,-2};
int diry[8]={ 1, 2, 2, 1,-1,-2,-2,-1};

void nextMoves(int x, int y, int n, queue q){
   int i;
   int nx, ny;

   for (i=0; i<8; i++) {
       nx=x+dirx[i]; ny=y+diry[i];
       if (nx>=0 && nx<n && ny>=0 && ny<n)
           if (s[nx][ny]==-1) {
               s[nx][ny]=s[x][y]+1;
               insert(q, nx, ny);
           }
   }
}
```

```
void minPathsKnight(int x, int y, int n){
   queue q;
   pos p;
   initQ(q);
   s[x][y]=0;
   insert(q, x, y);
   while (!isEmpty(q)){
       p=extract(q);
       nextMoves(p.x, p.y, n, q);
       }
}
```

5 HW3.2: Scacco di Regina

Per determinare se la regina dà scacco al re su una scacchiera vuota, è sufficiente verificare se la regina occupa la stessa colonna, la stessa riga o la stessa diagonale del re. Avendo la posizione rx, ry del Re, e dx, dy della donna, ciò si può verificare facilmente con la seguente procedura:

```
int scacco(int rx, int ry, int dx, int dy){
   if (rx==dx || ry == dy || rx+ry == dx+dy || rx-ry == dx-dy)
      return 1;
   else return 0;
}
```

A questo punto, nel caso il re non sia sotto scacco, è sufficiente contare le caselle raggiungibili dalla donna che darebbero scacco. Queste caselle si trovano percorrendo le 8 possibili direzioni lungo cui si muove la donna. Per evitare di scrivere 8 cicli, è possibile usare lo stesso trucco usato nel programma del cavallo, definendo dei vettori che indicano le direzioni: ad esempio la coppia $[0, 1]$ indica la direzione "colonna verso il basso", mentre $[-1, 1]$ diagonale verso basso-sinistra. Concludiamo la nostra fatica con la funzione completa:

```
int legale(int x, int y, int n){
   if (x>=0 && y>=0 && x<n && y<n) return 1 else return 0;
}

int contaScacco(int rx, int ry, int dx, int dy, int n){
   int x, y, i, j, s=0;
   if (scacco(rx, ry, dx, dy)) return 0;
   for (i=0; i<8; i++)
       for (j=1; ; j++) {
           x=dx+j*dirx[i]; y=dy+j*diry[i];
           if (legale(x,y,n)) s+=scacco(rx,ry,x,y);
              else break;
       }
   return s;
}
```

Buon homework 4!

```
int prossimeMosseCavallo(int r, int c, int n, int p) {
   int t=0;
   if (r - 2 >= 0) {
       if (c - 1 >= 0 && S[r-2][c-1] == -1) {
              S[r-2][c-1] = p; t=1;
          }
       if (c + 1 < n && S[r-2][c+1] == -1) {
              S[r-2][c+1] = p; t=1;
          }
   }
   if (r + 2 < n) {
       if (c - 1 >= 0 && S[r+2][c-1] == -1) {
              S[r+2][c-1] = p; t=1;
          }
       if (c + 1 <  n && S[r+2][c+1] == -1) {
              S[r+2][c+1] = p; t=1;
          }
   }
   if (c - 2 >= 0) {
       if (r - 1 >= 0 && S[r-1][c-2] == -1) {
               S[r-1][c-2] = p; t=1;
          }
       if (r + 1 <  n && S[r+1][c-2] == -1) {
               S[r+1][c-2] = p; t=1;
          }
   }
   if (c + 2 <  n) {
       if (r - 1 >= 0 && S[r-1][c+2] == -1) {
              S[r-1][c+2] = p; t=1;
          }
       if (r + 1 <  n && S[r+1][c+2] == -1) {
              S[r+1][c+2] = p; t=1;
          }
    }
  return t;
}
```

Figura 1: Funzione che trova le caselle raggiungibili dal cavallo in una mossa