2011-03-21 作業解答

1. 輸入兩數並輸出其最小公倍數。

The Least Common Multiple (LCM) is the product of two numbers divided by their Greatest Common Divisor (GCD), which can be evaluated recursively.

C++:

```
#include <iostream>
using namespace std;

unsigned gcd(unsigned a, unsigned b)
{
	if (a % b) return gcd(b, a % b);
	return b;
}

int main()
{
	cout << "input a b: ";
	unsigned a;
	unsigned b;
	cin >> a >> b;
	cout << "the LCM of " << a << " and " << b
	     << " is " << a * b / gcd(a, b) << '\n';
	return 0;
}
```

Python:

```
#!/usr/bin/env python
def gcd(x, y):
	if x % y: return gcd(y, x % y)
	return y
print "input a b:",
[a, b] = map(int, raw_input().split(' '))
print "the LCM of %d and %d is %d" % (a, b, a * b / gcd(a, b))
```

2. 輸入 n 個名稱並輸出其所有 2^n 個組合。

Using a recursive thinking, the task of producing all combinations of n items can be broken up into withholding one item while producing all combinations of the rest n-1 items plus producing the same combinations of the n-1 items while adding the withheld item to the mix of each of the produced combination.

C++:

```
#include <iostream>
using namespace std;

string composed_output;

void show_combinations(string it[], size_t n)
// show combinations of 'n' items in the array 'it'
{
	if (!n) { // no more items to hold, dump the output
		cout << "["
		     << (composed_output.size() ? composed_output : " ")
		     << "]\n";
		return;
	}
	show_combinations(it + 1, n - 1); // withhold 1st, do the rest
	string save_str = composed_output; // save composed_output
```

```
        // now add the withheld to all mixes
        if (composed_output.size()) composed_output += ", ";
        composed_output += * it; // show the 1st item
        show_combinations(it + 1, n - 1); // do the rest 2 ~ n items again
        composed_output = save_str; // restore composed_output
  }

  int main()
  {
        cout << "How many items? ";
        unsigned nitem;
        cin >> nitem;
        string names[nitem];
        for (int i = 0; i < nitem; i ++) {
              cout << "item " << i + 1 << ": ";
              cin >> names[i];
        }
        show_combinations(names, nitem); // call the recursive function
        return 0;
  }
```

On the other hand, we know combinations is related to binomial expansion. Consider the product expansion (1+x)*(1+y)*(1+z) = 1+x+y+z+x*y+x*z+y*z+x*y*z. Each term in the expansion corresponds to a combination of x, y, and z because to form a term on the right hand side, we need to decide between 1 and the variable for each factor on the left hand side. If we replace 1 by the empty set {}; the variables by sets of single items, e.g., {apple}; and the '*' operator by the union operator between sets, we get the terms to represent all the combinations of the items in the product. This actually can be done very concisely in Python.
Python:

```
  #!/usr/bin/env python
  print 'Number of items:',
  n = int(raw_input())
  l = []
  for i in range(0,n):
        print 'Item %d:' % (i + 1),
        l.append(raw_input())
  union_product = lambda x,y: [a + b for a in x for b in y]
  for i in reduce(union_product,[[[],[x]] for x in l]):
        print i
```

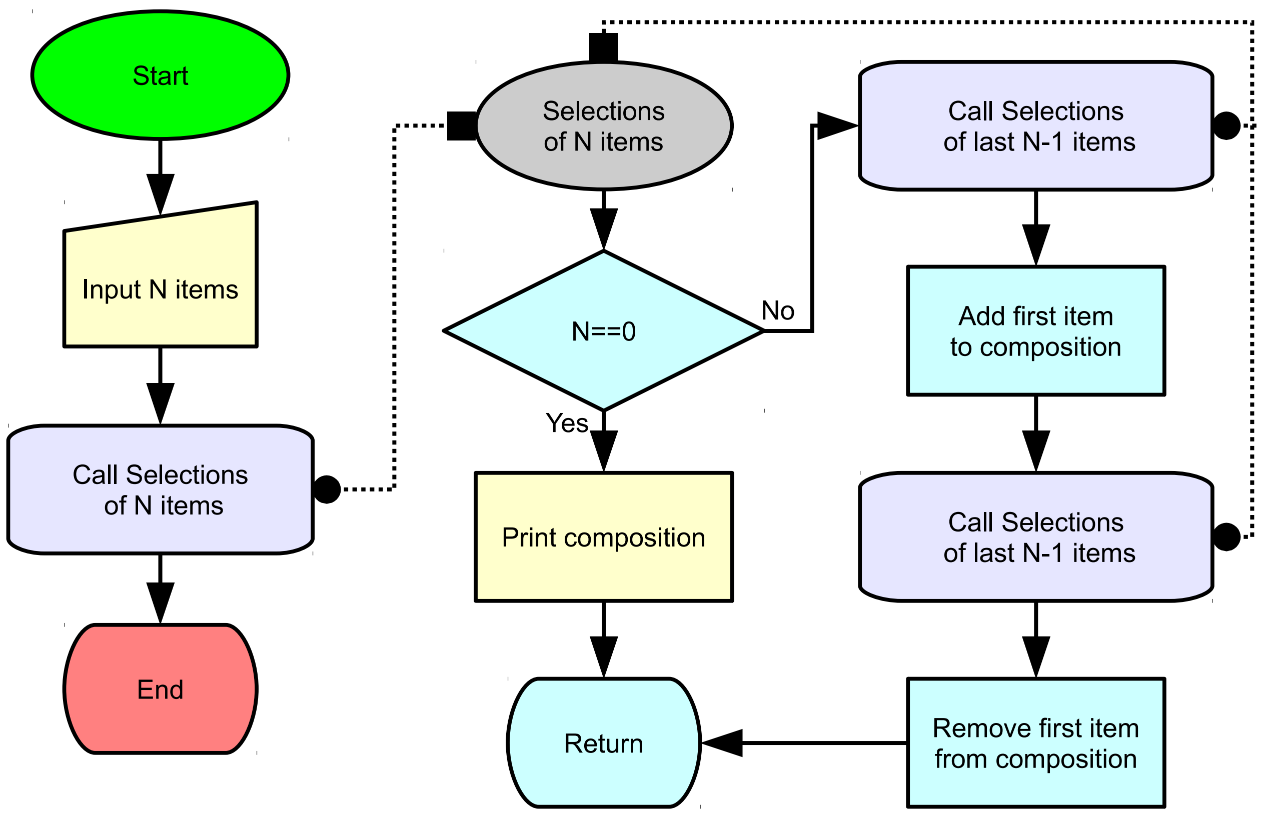
A very popular way to find all combinations from the set of n items is to map the inclusion of each item to a digit of a n-digit binary number, where 1 means the item is included and 0 means it is not. One can then run through all numbers from 0 to 2^n-1 to produce all possible combinations.
C++:

```
  #include <iostream>
  using namespace std;
  int main()
  {
        cout << "How many items? ";
        unsigned nitem;
        cin >> nitem;
        string names[nitem];
        for (int i = 0; i < nitem; i ++) {
              cout << "item " << i + 1 << ": ";
              cin >> names[i];
        }
        for (unsigned n = 0; n < (1 << nitem); n ++) {
              cout << '[';
              bool first = true;
              for (unsigned b = 1 << nitem, i = 0; b >>= 1; i++) if (n & b)
  {
                    if (first) first = false;
                    else cout << ',';
                    cout << names[i];
              }
              if (first) cout << ' ';
              cout << "]\n";
        }
        return 0;
  }
```

Sometimes, it is also desirable to order the combinations in the number of items each combination contains. Without doing post processing sort, it becomes somewhat tricky to enumerate over the combinations. This can be a good brain exercise even though it is not directly relevant to the course.

3. 繪出你在 2.所用的流程圖。（手繪的請寄掃描檔）

It is actually uncommon to express recursion in a flowchart. However, the goal is simply to provide a visual representation of the algorithm used and there are more than one way of doing this. See http://www.csc.liv.ac.uk/~frans/OldLectures/COMP101/week9/recursion.html for some insight on this.


Start
Input N items
Call Selections of N items
End
Selections of N items
N==0
No
Yes
Print composition
Return
Call Selections of last N-1 items
Add first item to composition
Call Selections of last N-1 items
Remove first item from composition

4. 實作河內塔問題的遞迴解。

From a recursive view, moving a stack of n disks from rod A to rod C amounts to moving the stack of n-1 disks from A to B; moving the n-th disk from A to C; then moving the n-1 stack from B to C.
C++:

```
#include <iostream>
using namespace std;
char rods[] = {'A', 'B', 'C'};

void move(size_t fr, size_t to, size_t n)
{
      if (n) {
            move(fr, 3 - fr - to, n - 1);
            cout << rods[fr] << "->" << rods[to] << '\n';
            move(3 - fr - to, to, n - 1);
      }
}
int main()
{
      cout << "Moving from A to C\n";
      cout << "How many disks? ";
      size_t n;
      cin >> n;
      move(0, 2, n);
      return 0;
}
```

A Python version would look remarkably identical except we could use

```
mw = (set('ABC') - set(fr + to)).pop()
```

to figure out the midway rod.

［隨意題］以非遞迴方式解出河內塔問題

To do it non-recursively, we can observe that there are limited choices at each move during the process. The smallest disk is always on top of one of the three rods. If we are not moving the smallest disk, there is only one way to move. Also, there is no point of making consecutive moves of the smallest disk. So, we will have to move the smallest disk for every other move. Observing the solution, we see the smallest disk is moving in a fixed cyclic direction, i.e., A → B → C → A → B → C. Combining all these consideration, we can narrow down to the right move for each step. The implementation includes an ASCII art output as illustrated below.

```
     I          I          I
     I          I          I
     I          I          I
     I          #         ###
   #####     #######   #########
=================================

     I          I          I
     I          I          I
     I          I          I
    ###         #          I
   #####     #######   #########
=================================

     I          I          I
     I          I          I
     #          I          I
    ###         I          I
   #####     #######   #########
=================================
```

C++:

```cpp
#include <iostream>
#include <vector>
using namespace std;
size_t n;
vector<int> rods[3];
void output_stacks()
{
      cout << '\n';
      for (size_t i = n; i; i --) {
            for (size_t r = 0; r < 3; r ++) {
                  if (rods[r].size() <= i) {
                        cout << string(n, ' ') << 'I';
                        cout << string(n, ' ');
                  }
                  else {
                        size_t d = rods[r][i];
                        cout << string(n - d, ' ');
                        cout << string(d * 2 + 1, '#');
                        cout << string(n - d, ' ');
                  }
            }
            cout << '\n';
      }
      cout << string(3 * (2 * n + 1), '=') << '\n';
}
void move(size_t f, size_t t)
{
      rods[t].push_back(rods[f].back());
      rods[f].pop_back();
}
int main()
{
      cout << "How many disks? ";
      cin >> n;
      for (size_t i = 0; i < 3; i ++) rods[i].push_back(n);
      for (size_t i = n; i --;) rods[0].push_back(i);
      output_stacks();
      bool m_small = true;
      size_t small = 0;
      for (size_t i = 1 << n; -- i;) {
            size_t s1 = (small + 1) % 3;
            size_t s2 = (small + 2) % 3;
            if (m_small) {
                  size_t s = (s1 + n % 2) % 3;
                  move(small, s);
                  small = s;
            }
            else {
                  if (rods[s1].back() > rods[s2].back()) move(s2, s1);
                  else move(s1, s2);
            }
            output_stacks();
            m_small = ! m_small;
      }
      return 0;
}
```

5. 線性代數：由檔案輸入所有係數 a 及 c，輸出 x 的解（假定 det(a)不為零）。

a11 x1+a12 x2+a13 x3 = c1
a21 x1+a22 x2+a23 x3 = c2
a31 x1+a32 x2+a33 x3 = c3

To solve this system of equations, we can invert the matrix a and have the vector c multiplied by it to get the vector x. We just use the brute force coding for the 3 by 3 matrix.
C++:

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
      ifstream f_i("input.txt");
      double a[3][3];
      double c[3];
      for (size_t i = 0; i < 3; i ++)
      for (size_t j = 0; j < 3; j ++) f_i >> a[i][j];
      for (size_t i = 0; i < 3; i ++) f_i >> c[i];
      double det = a[0][0] * a[1][1] * a[2][2]
            + a[0][1] * a[1][2] * a[2][0]
            + a[0][2] * a[1][0] * a[2][1]
            - a[0][0] * a[1][2] * a[2][1]
            - a[0][1] * a[1][0] * a[2][2]
            - a[0][2] * a[1][1] * a[2][0];
      double b[3][3];
      b[0][0] = a[1][1] * a[2][2] - a[1][2] * a[2][1];
      b[0][1] = a[1][2] * a[2][0] - a[1][0] * a[2][2];
      b[0][2] = a[1][0] * a[2][1] - a[1][1] * a[2][0];
      b[1][0] = a[2][1] * a[0][2] - a[2][2] * a[0][1];
      b[1][1] = a[2][2] * a[0][0] - a[2][0] * a[0][2];
      b[1][2] = a[2][0] * a[0][1] - a[2][1] * a[0][0];
      b[2][0] = a[0][1] * a[1][2] - a[0][2] * a[1][1];
      b[2][1] = a[0][2] * a[1][0] - a[0][0] * a[1][2];
      b[2][2] = a[0][0] * a[1][1] - a[0][1] * a[1][0];
      double x[3];
      x[0] = (b[0][0] * c[0] + b[0][1] * c[1] + b[0][2] * c[2]) / det;
      x[1] = (b[1][0] * c[0] + b[1][1] * c[1] + b[1][2] * c[2]) / det;
      x[2] = (b[2][0] * c[0] + b[2][1] * c[1] + b[2][2] * c[2]) / det;
      cout << "(x1,x2,x3)=(" << x[0] << ',' << x[1] << ',' << x[2] <<
")\n";
      return 0;
}
```