CHAPTER 6
EXHAUSTIVE SEARCH and BACKTRACKING

Introduction

Sometimes only the way to solve a problem is to try all the possibilities. This is always slow but better than nothing. Exhaustive Search systematically searches for a solution to a problem among all available options. Sometimes, in solving a problem, a situation arises to a place where there are different ways leading from the current position, none of which known to lead to a solution. In such a case, after trying one path without reaching any solution, we return to this crossroads and try to find a solution using another path. Backtracking speeds up the Exhaustive Search by pruning the paths that don't lead to any solutions.

In order to implement Exhaustive Search and backtracking programs, we use divide and conquer technique. The solutions are represented by vectors of values. When invoked, usually, the algorithm starts with an empty vector. At each stage it extends the partial vector with a new value. In some cases instead of a vector the solutions are represented by a matrix.

In this chapter first we will study **Exhaustive Search** and **Backtracking** one by one with some sample programs. It can be possible to make those programs more efficiently with other methods. We don't care it, because our aim is to teach Exhaustive Search and Backtracking. Next, in the Lab Tasks, you will find some classical problems to solve. We will give you the algorithms, then you will make the programs. Finally in the last part, Programming Projects, you will be exposed to some problems from different national and international programming contests. For those problems you must make the algorithms and the programs.

Exhaustive Search

Exhaustive Search systematically searches all the solutions for a problem. Suppose that there is a mouse in a maze. All the paths have been numbered in each intersection in the maze. Our mouse is in its initial position. It knows that somewhere in the maze there is a piece of cheese. Our mouse is starving. After thinking a while, it finds a method to search all the maze without searching the same place more than once. Fortunately, he has studied some programming. It starts to move from

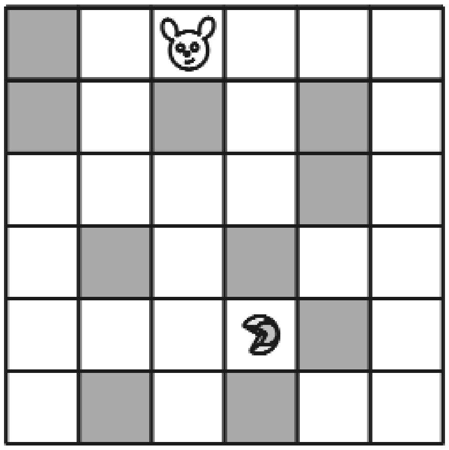


Figure 6.1: Where is the cheese?

its initial position to the first path and when it encounters an intersection, he tries the first path, and if it sticks somewhere, the mouse comes back to the previous intersection and tries the next untried path. Later in this chapter, we will make a program to help the mouse. Now let's see the implementation with some sample programs.

Example 1: Binary Strings.

Problem: Make a program to generate all the binary strings of n bits, where n is a positive integer.

Solution: There are 2^n solutions. We use a procedure to generate one single bit of the string and then we call it recursively to generate the next bit. At the first time we call the procedure for the first bit of the string. In the procedure body, we check if we have generated n bits so far. If we have n bits so we have a solution. We just display the solution. If we don't have n bits, first we assign 0 to the current position and call the procedure to generate the next position. After that, we assign 1 to the current position and call the procedure to generate the next position. In the program we use an array X[1..n] to represent the string. Call procedure binary(1).

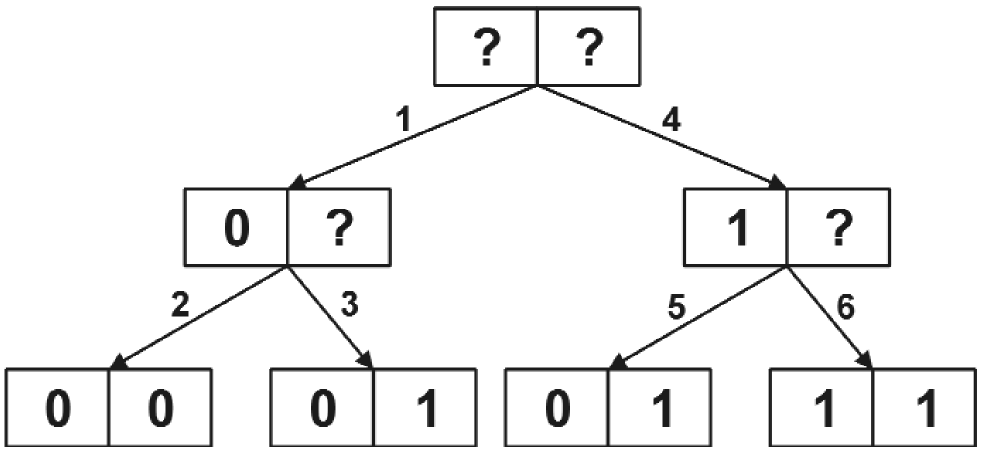


Figure 6.2: Generating binary strings of length 2. Leaf nodes represent the solutions. The numbers on the branches denote the flow of the program.

```
Procedure binary(m)
   If m>n then print
   Else
          X[m] := 0; binary(m+1);
          X[m] := 1; binary(m+1);

Program:

Program Bit_Strings;
var
    X : array[1..30] of byte;
    n : byte;
{-----------------------------------------}
Procedure Print;   {print a solution}
var
    i:byte;
begin
       for i:=1 to n do
            write(X[i],' ');
          writeln;
     end;
     {-------------------------------------}
     Procedure Binary(m:byte);
     begin
          if m>n then  print
          else
          begin
               X[m] := 0;    Binary(m+1);
               X[m] := 1;    Binary(m+1);
          end;
     end;
     {-------------------------------------}
     Begin
          write('n=?');
          readln(n);
          Binary(1);   {call the procedure for the first element of
     the array X}
          readln;
     End.
```

The procedure Binary can be modified as follow.

```
     {-------------------------------------}
     Procedure Binary(m:byte);
     var
       i:byte;
     begin
          if m>n then  print
          else
          for i:=0 to 1 do
          begin
               X[m] := i;    Binary(m+1);
          end;
     end;
     {-------------------------------------}
```

Output:

```
N=?2
0 0
0 1
1 0
1 1
```

Example 2: K-ary Strings.

Problem: Generate all the strings of n numbers drawn from 0.. k-1.

Solution: There are k^n solutions. We use the same algorithm used in the first example. Instead of 0 and 1, at this time we assign the values 0..k-1 to the each element of the array. Call procedure k_ary(1).

```
     Procedure kary(m)
```

```
   If m>n then print
   Else
         For i:=0 to k-1 do
X[m] := i; kary(m+1);
```

```
Program:

Program k_ary_strings;
var
   X : array[1..30] of byte;
   n,k : byte;
{-------------------------------------------------------}
Procedure Print;
var
   i:byte;
begin
     for i:=1 to n do
         write(X[i],' ');
     writeln;
end;
{-------------------------------------------------------}
Procedure k_ary(m:byte);
var
   i:byte;
begin
     if m>n then  print
     else
     for i:=0 to k-1 do
     begin
          X[m] := i;    k_ary(m+1);
     end;
end;
{-------------------------------------------------------}
Begin
     write('n and k=?');
     read(n,k);
     k_ary(1);
     readln;
End.
```

Output:

```
n and k=?2 3
0 0
0 1
0 2
1 0
1 1
1 2
2 0
2 1
2 2
```

Example 3: Permutations.

Problem: Generate all the permutations of the set S={1,2,...,n}.

Solution: There are n! solutions. We generate all the strings of n numbers drawn from 1.. n and then we display only the ones that represent the permutations. If a string of length n, doesn't have a duplicated value, it is a permutation. Our program generates n^n strings. Actually we need to generate only n! strings. So our algorithm is slower than it should be. In the next section we will write a faster version of the program permutations. Call procedure permutations(1).

```
Procedure permutations(m)
  If m>n then
       If IsPermutation then print;
       Else
            For i:=1 to n do
                 X[m] := i; binary(m+1);
```

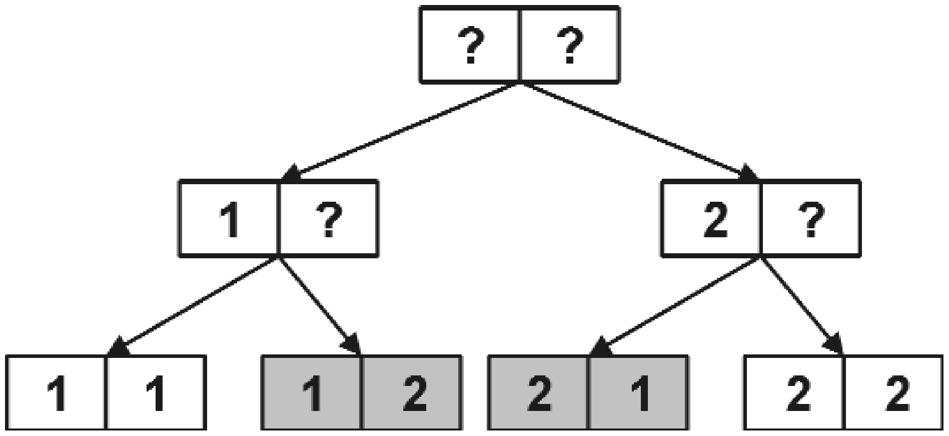


Figure 6.3: Generating the permutations of the set S = {1,2}. Shaded nodes represent the solutions.

Program:

```
Program Permutations;
var
   X : array[1..15] of byte;
   n : byte;
{------------------------------------------------------}
Function valid:boolean;
{return true if X represent a permutation, false otherwise}
var
   i,j : byte;
begin
     for i:=1 to n-1 do
          for j:=i+1 to n do
          if X[i] = X[j] then     {dublicated value}
          begin
               valid := false;
               exit;
          end;
     valid := true;
```

```
end;
{------------------------------------------------------}
Procedure Print;
var
   i:byte;
begin
     for i:=1 to n do
          write(X[i],' ');
     writeln;
end;
{------------------------------------------------------}
Procedure Permutations(m:byte);
var
   i:byte;
begin
     if m>n then
     begin
          if valid then print
     end
     else
     for i:=1 to n do
     begin
          X[m] := i;     Permutations(m+1);
     end;
end;
{------------------------------------------------------}
Begin
     repeat
           write('n=?');
           read(n);
     until n>=1;
     Permutations(1);
     readln;
End.
```

Output:

```
n=?3
1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1
```

In this program first we generate all the n bits strings, then we check each of them if it represents a permutation. For this purpose we use the function valid. So the function valid is called n^2 times. It is obvious that if we can write a faster version of the function valid, our program will run faster. In the function valid we search the ith element in the partial array X[i+1..n]. We make this search process n times. To make it faster we can keep the occurence of the elements of the array X in another array (that is the array used[1..n] in the next program) and in the function valid we can only check if there exist an element whose occurence is different from one.

<u>Program:</u>

```
Program Permutations;
var
   X : array[1..15] of byte;
   used : array[1..15] of byte;  {ith element keeps the
occurence of the value i}
   n,i : byte;
{-------------------------------------------------------}
Function valid:boolean;
var
   i,j : byte;
begin
     for i:=1 to n do
          if used[i]<>1 then {each element must be occured only
once}
          begin
               valid := false;
               exit;
          end;
     valid := true;
end;
{-------------------------------------------------------}
Procedure Print;
var
   i:byte;
begin
     for i:=1 to n do
          write(X[i],' ');
     writeln;
end;
{-------------------------------------------------------}
Procedure Permutation(m:byte);
var
   i:byte;
begin
     if m>n then
     begin
           if valid then print
     end
     else
     for i:=1 to n do
     begin
           X[m] := i;
           Inc(used[i]);        {increase the occurence of the
ith element}
           Permutation(m+1);
           Dec(used[i]);        {decrease the occurence of the
ith element}
     end;
end;
{-------------------------------------------------------}
Begin
     repeat
```

```
                 write('n=?');
                 read(n);
         until n>=1;
         for i:=1 to n do
               Visited[i] := 0; {as an initial occurence of each
element is 0}
         Permutation(1);
         readln;
End.
```

Output:

```
n=?3
1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1
```

Backtracking

Backtracking is a faster version of Exhaustive Search. When a situation arises to a place where there are different ways leading from the current position, in Exhaustive Search, we try all the paths; however, in Backtracking, we use a criteria and prune the paths that don't lead to a solution.

Example 1: Permutations

Problem: Generate all the permutations of the set S={1,2,...,n}.

Solution: We just modify the program above. Before we assign the value i to the mth element of the array X, we check if it has been used in the sub-array X[1..m-1]. If it has been used then we try the next value. In the program we use a boolean array used[1..n], where used[m] is true if the value m has been used.

Program:

```
Program Permutations;
var
    X : array[1..15] of byte;
    used:array[1..15] of boolean;
    i,n : byte;
{-----------------------------------------------------}
Procedure Print;
var
    i:byte;
begin
      for i:=1 to n do
            write(X[i],' ');
      writeln;
end;
```

```
{------------------------------------------------------}
Procedure Permutation(m:byte);
var
   i:byte;
begin
     if m>n then print
     else
     for i:=1 to n do
     begin
          if not used[i] then
          begin
               X[m] := i;
               used[i] := true;
               Permutation(m+1);
               used[i] := false;
          end;

     end;
end;
{------------------------------------------------------}
Begin
     repeat
           write('n=?');
           read(n);
     until n>=1;
     for i:=1 to n do
         used[i] := false;
     Permutation(1);
     readln;
End.
```

Output:

```
N=?3
1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1
```

Example 2: N Queens Problem.

Problem: Find all the placements for n queens on an NxN chess board so that no two queens threaten each other.

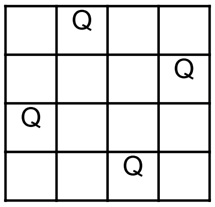


Figure 6.5

Solution: This is one of the classical backtracking problems. You know that two queens threaten each other if they are on the same row, on the same column or on the same diagonal. It is clear that all the queens must be on separate rows, so we can represent the chess board with the vector X[1..N]. X[i] indicates the column number of the queen in row i. This representation guarantees that all the queens are on separate rows. The placement in the picture is X[1] = 2, X[2] = 4, X[3] = 4 and X[4] = 3.

The problem states that, we must generate all the permutations - of the set S={1,2,..,n} - that represent N queens on an NxN board without attacking one another. In the program, while we are generating all the permutations, we will stop whenever one queen threatens another. For this purpose we will write the boolean function DoesAttack. DoesAttack returns True if the new queen threatens one or some of the old queens. Ohterwise it returns False. How can we check if two queens theaten each other? We need to check only if they are on the same column or on the same diagonal. Suppose that first queen is on the ith column and second queen is on the jth column:
If i=j, they are on the same column.
If abs(i-j) = abs(X[i] - X[j]), they are on the same diagonal, where abs stands for absolute value. Call procedure Queens(1).

```
Procedure Queens(m:byte);
Begin
  If m>n then print
  Else
       For i:=1 to n do
            X[m]:=i;
            If Not DoesAttack(m) then Queens(m+1)
```

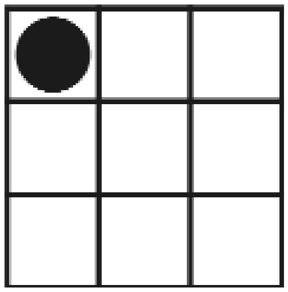


Figure 6.6a: Replace the first queen to the first unguarded square in the first row.

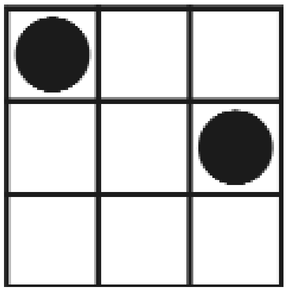


Figure 6.6b: Replace the second queen to the first unguarded square in the second row.

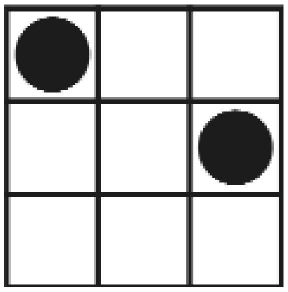


Figure 6.6c: Replace the third queen to the first unguarded square in the third row. There is no such a square. Backtrack and replace the second queen to the next unguard ed square in the second row. There is no such a square. Backtrack one more time and replace the first queen to the next unguarded square in the first row.

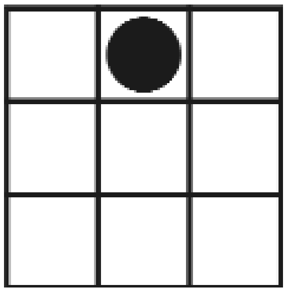


Figure 6.6d: **Replace the second queen to the first unguarded square in the second row. There is no such a square. Backtrack and replace the first queen to the next unguarded square in the first row.**

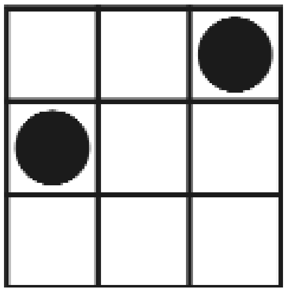


Figure 6.6e: **Replace the second queen to the first unguarded square in the second row.**

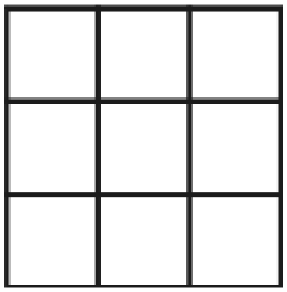


Figure 6.6f: **Replace the third queen to the first unguarded square in the third row. There is no such a square. Backtrack and replace the second queen to the next unguarded square in second row. There is no such a square. Backtrack and replace the first queen to the next unguarded square in the first row. There is no such a square. So there is no solution for an 3X3 board.**

Program:

```
Program N_Queens;
const max = 10;
var
   x : array[1..max] of byte;
   n : byte;
{----------------------------------------------------------}
procedure print;
var
   j:byte;
begin
     for j:=1 to n do
          write(j,':',X[j],'   ');
     writeln;
     readln;
end;
{----------------------------------------------------------}
function DoesAttack (k:byte) :boolean;
var
```

```
    i : byte;
begin
     for i:=1 to k-1 do
               if (x[i] = x[k]) or                  {in the
same column}
                  (k - i = abs(x[k] - x[i]))       {in the same
diagonal}
               then
               begin
                    DoesAttack := True;
                    exit;
               end;
     DoesAttack := False;
end;
{----------------------------------------------------------}
procedure Queens(m:byte);
var
   i:byte;
begin
     if m>n then Print
     else
          for i:=1 to n do
          begin
               x[m] := i;
               if Not DoesAttack(m) then  Queens(m+1)
          end;
end;
{----------------------------------------------------------}
Begin
     write('n = ?');
     readln(n);
     Queens(1);
end.
```

<u>Output:</u>

```
N = ?4
1:2 2:4 3:1 4:3
1:3 2:1 3:4 4:2
```

Example 3: Combinations

<u>Problem:</u> Generate all the combinations of n things chosen r at a time.

<u>Solution:</u> We must choose r objects from n. The order does not matter, (1,2) and (2,1) are the same combination. We can solve this problem with some modifications of the permutation program. In the permutation program we displayed n elements of the array X, now we need to display r elements. Furthermore we need to add another condition to avoid generating the same combination more than once. For this purpose we can generate the element of the array X[1..r] in ascending order. So that each element of X is bigger then previous one. In the program, take the declaration of X (X[0..r]) into consideration. Call procedure combinations(1).

```
Procedure combinations(m:byte);
```

```
If m>r then print
Else
     For i:=X[m-1]+1 to n do
               X[m] := i;
               combinations[m+1];
```

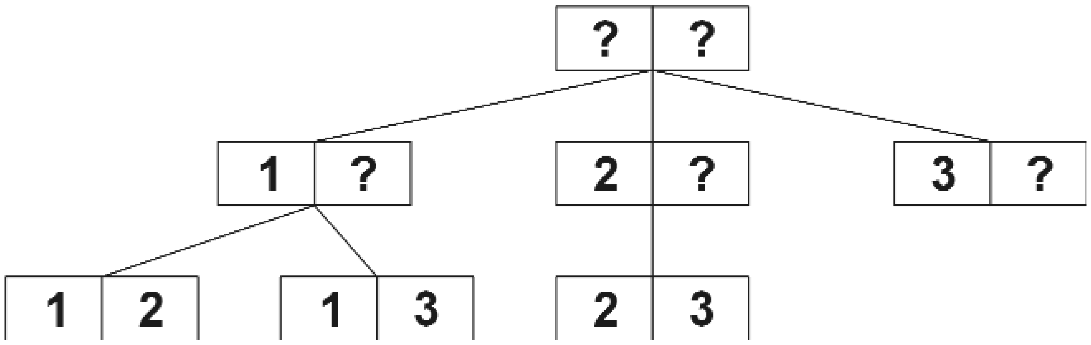


Figure 6.7: Generating combinations of 3 items chosen 2 at a time.

Program:

```
Program Gen_Combinations;
var
   X : array[0..30] of byte;
   n,r : byte;
{-------------------------------------------------------}
Procedure Print;
var
   i:byte;
begin
     for i:=1 to r do
          write(X[i],' ');
     writeln;
end;
{-------------------------------------------------------}
Procedure combinations(m:byte);
var
   i:byte;
begin
     if m>r then print
     else
     for i:=X[m-1]+1 to n do
          begin
               X[m] := i;
               combinations(m+1);

          end;
end;
{-------------------------------------------------------}
Begin
     write('n and r=? [r <= n]');
     read(n,r);
     combinations(1);
End.
```

Output:

```
N and r=? [r <= n]3 2
1 2
1 3
2 3
```

Example 4: Knapsack Problem.

Problem: A set of items S={1,2,..,n} is given, where item i has size si and a knapsack capacity C. Find all the subsets S' Ì S that uses the full capacity of the knapsack. Items cannot be broken into smaller pieces.

Solution: This is a kind of combination problem. We have N items and we must choose r of them at a time, where r varies between 1 and N. When we are adding an unused item to the subset S' (that is X[0..n] in the program), we must take care of the knapsack capacity, we mustn't exceed it. In the program the procedure knapsack has two parameters: m and k. m is the index of the next element of X to be filled, k is the sum of the sizes of items that have been selected. If k equal to the knapsack capacity, we have a solution, we display the selected items. If k is bigger than the capacity, it cannot be a part of a solution, we discard the last item in X and try the next candidate item, if k is less than capacity, we may have a partial solution, we keep the last item and continue with the next candidate item. Call procedure knapsack(1,0).

```
Procedure knapsack(m,k:byte);
  If k=c then
print(m-1)
  Else if k<c then
       For i:=1 to n do
            If i > X[m-1] then
                 X[m] := i;
                 Knapsack(m+1,k+sizes[i]);
```

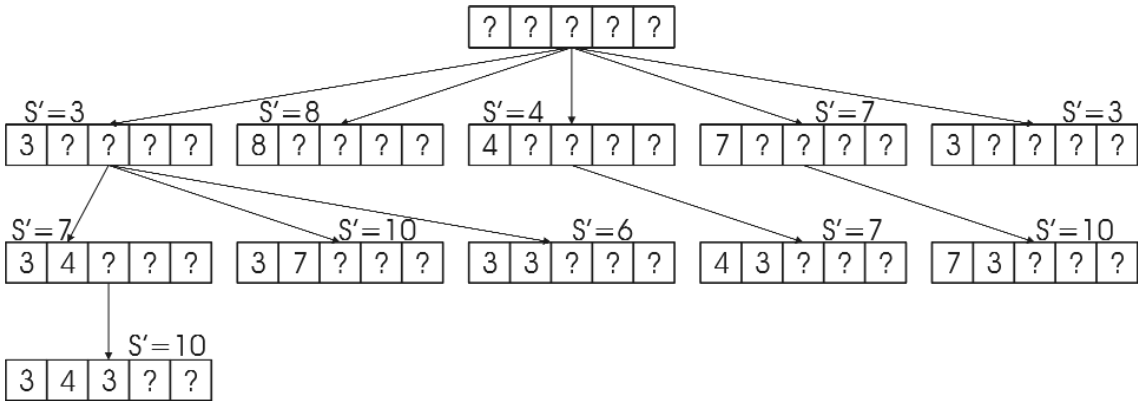


Figure 6.8: Finding the solution of the Knapsack Problem for 5 items sizes of 3,8,4,7,3 and the knapsack capacity 10. s' denotes total size of the selected items. The nodes represent the solutions, where s' = 10.

Program:

```
Program OneZero_Knapsack;
const max = 20;
var
   X,Sizes : array[0..max] of byte;
   i,n,c:integer;
{------------------------------------------------------}
Procedure Print(k:byte);
{displays indeces of the selected items}
var
   i:byte;
begin
     for i:=1 to k do
          write(X[i],' ');
     writeln;
end;
{------------------------------------------------------}
Procedure knapsack(m,k:byte);
var
   i:byte;
begin
     if k = c then
        Print(m-1)
     else if k < c then
          for i:=1 to n do
               if (i > X[m-1]) then
               begin
                    X[m] := i;
                    knapsack(m+1,k+sizes[i]);
               end;
end;
{------------------------------------------------------}
Begin
     write('Number of the items=? ');
     readln(n);
     write('Capacity of the knapsack=? ');
     readln(c);
     for i:=1 to n do
     begin
          write('size of the item #',i,'=?');
          readln(sizes[i]);
     end;
     knapsack(1,0);
End.
```

Output:

```
Number of the items =?5
Capacity of the knapsack =?10
Size of the item #1=?3
Size of the item #2=?8
Size of the item #3=?4
Size of the item #4=?7
Size of the item #5=?3
1 3 5
1 4
4 5
```

Laboratory Tasks

Task 1: Faster Permutations.

Problem: Generate all the permutations of the set S={1,2,...,n}.

Solution: In the backtracking solution, we attempt to add all the candidates to the partial solution one by one , check all of them if they can be part of a solution and prune the ones that don't lead to a solution. A better idea is to get a new permutation by changing two members of an existing permutation systematically. In other word, selecting one element and recursively permuting the remaining elements. First initialize X[i] = i for all 1<= i<=n. This is the first permutation. Select an element in X[1..n] and recursively permute the remaining elements.

$$\text{Perm}(x_1, \dots, x_n) = \begin{cases} \text{Xi, Perm(x1, ..., xi-1, xi+1) if n>1} \\ \text{Xn if n=1} \end{cases}$$

Call the procedure permutation(1)

```
Procedure permutation(m:byte);
  If m=n then print
  Else
       For i:=m to n do
            Swap(X[m],X[i])
            Permutation(m+1);
            Swap(X[m],X[i]);
```

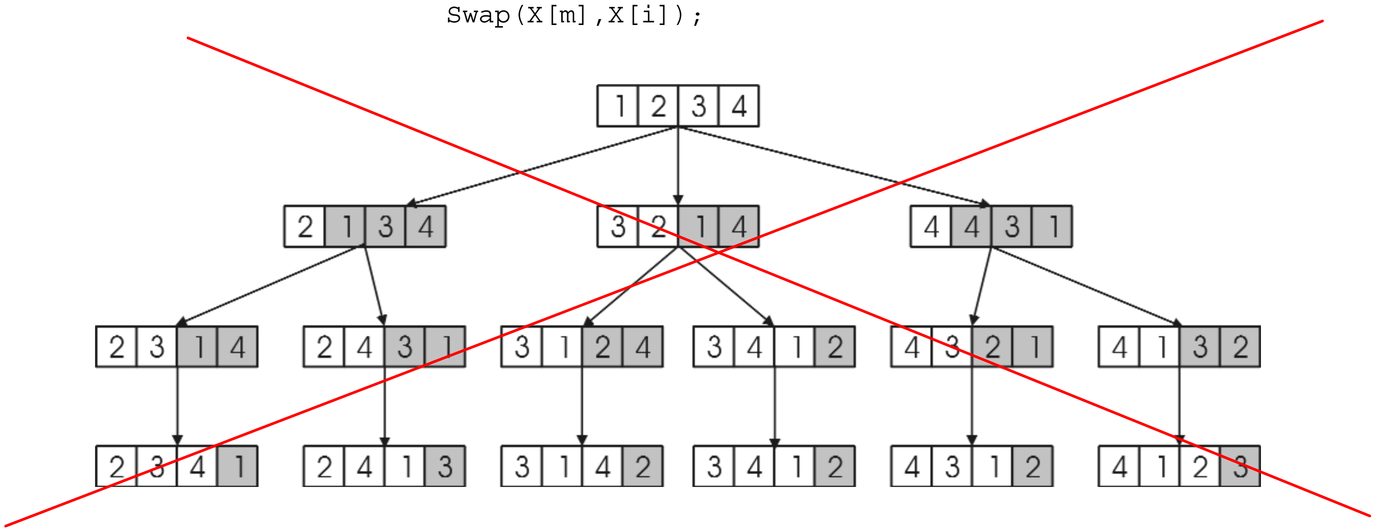


Figure 6.9 Generating permutations for the set S = {1,2,3}. Leaf nodes represent the permutations. Shaded digits represent the subsets that will be processed.

Task 2: Faster Combinations

Problem: Generate all the combinations of n things chosen r at a time.

Solution: In the first version of the program combinations we generated the combinations in increasing order so we never attempted to use same item twice. In this way we assigned the minimum possible value to the each element of X as a first value. But we didn't do the same thing for the final value. In the tree illustration of the execution of the program you can see some uncompleted branches in picture 6. We modify the procedure combinations so that the elements of X will not get the values that don't lead to a solution. Call procedure combinations(1).

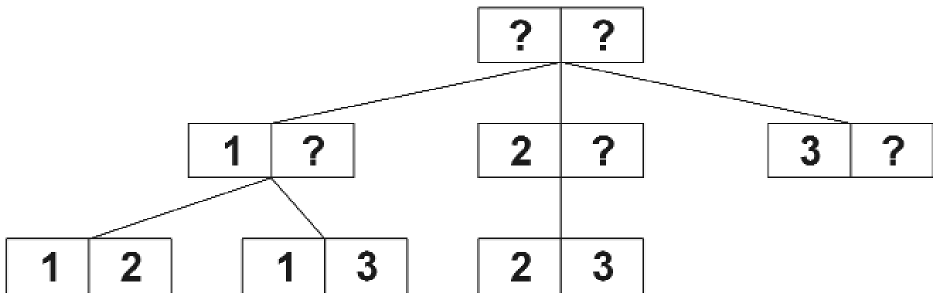


Figure 6.10: Generating combinations of 3 items chosen 2 at a time.

```
Procedure combinations(m:byte);
  If m>r then print
  Else
       For i:= X[m-1] + 1 to n-r+m do
            X[m] := i;
            Combinations(m+1);
```

Task 3: Knapsack - II

Problem: A set of items S={1,2,..,n} is given, where item i has size si, and value vi and a knapsack capacity C. Find the subsets S' Ì S that maximizes the value of SiÎS'vi subjects to SiÎS'si£ C. Items can not be broken into smaller pieces.

Solution: In the first version of the knapsack problem we were interested in the sizes of the items. In this version, we are interested in sizes and values of the items. We must maximize the total values of the selected items and we mustn't exceed the knapsack capacity but we don't need to fill it up. To solve this problem with backtracking, we must generate all the combinations where sum of the sizes is less than or equal to knapsack capacity and another item cannot be added without exceeding the capacity. The combination which has the biggest total value is the solution. Actually when the knapsack capacity is not so big there is an efficient dynamic programming algorithm but it is out of scope of the book.

Task 4: Subsets

Problem: An n-elemented set, S={1,2,..,n} is given. Generate all the subsets of S.

Solution: This is another combination problem. We have n items and we must generate all the combinations chosen any number of the items at a time. Procedure subsets generates all the subset except empty set. Call the procedure SubSets(1).

```
Procedure subsets(m:byte);
  For i:= X[m-1]+1 to n do
       X[m]  := i;
       Print(m);             {print the first m element of X}
       SubSets(m+1);
```

Task 5: Partitions

Problem: Generate all the partitions of N, where N⊂N_+. For N=3 the partitions are: (1 1 1), (1 2), (3).

Solution: In the partitions, we may have duplicated values so that each element of X is equal to or bigger than previous one. Whenever we add a new value to X, we extract it from N. When n gets 0, we have a solution. Before calling the program X[0] must get 1. Call the procedure Partitions(1,n).

```
Procedure partitions(m,k:byte);
  If k=0 then print(m-1)
  Else
       For i:= X[m-1] to k do
       X[m]:= i;
       Partitions(m+1,k-i);
```

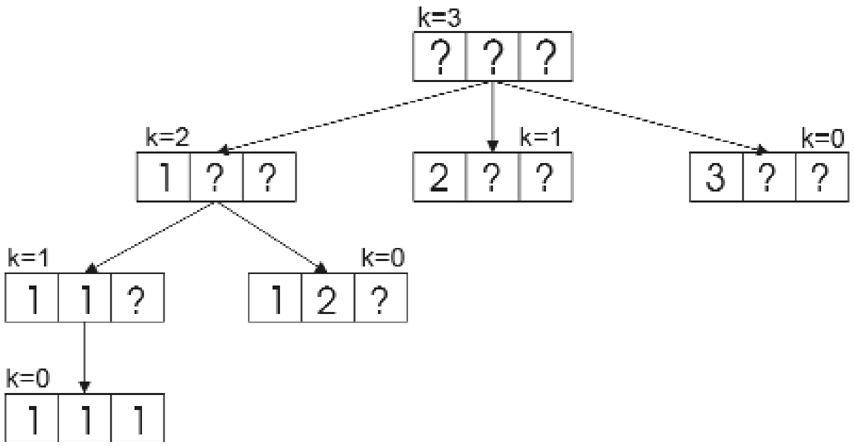


Figure 6.11: Generating partitions of 3. k denotes the remainder of the number will be processed. The nodes represent the solutions where k = 0.

Task 6: Parenthesis

Problem: N pairs of parenthesis are given. Generate all the correct arrangements of them, where N⊂ N_+. For N=2 the output is: (()), ()().

Solution: It is a kind of permutation problem. We can generate all the permutations then discard the ones that are not solutions. But this is not a good algorithm. Because we waste a lot of time for the permutations that are not the solutions. We may directly generate the solutions. Any close parentheses must match an open parentheses and number of the open parentheses mustn't be more than N. Call procedure parentheses(1,0,0).

```
Procedure parentheses(m,open,close:byte);
{m is the next element, open is number of the open parentheses,
close number of the close parentheses so far}
  If m>n then print
  Else
       If open < n then
            X[m] := '(';
            Parentheses(m+1, open+1, close);
       If close < open then
            X[m] := ')';
            Parentheses(m+1, open, close+1);
```

It is clear that always X[1] = '(' and X[n]=')'. Before calling the procedure, you can assign '(' to X[1] and ')' to X[2] and you can omit them in the procedure.

Task 7: Traveling Salesman

Problem: The Traveling Salesman Problem (TSP) assumes a set of N cities, and a salesman who needs to visit each city exactly once and return to the base city at the end. The solution would provide a route of minimal length. In the picture 15, the route (1 -> 4 -> 2 -> 3 -> 1) is the shortest one, and its length 35.

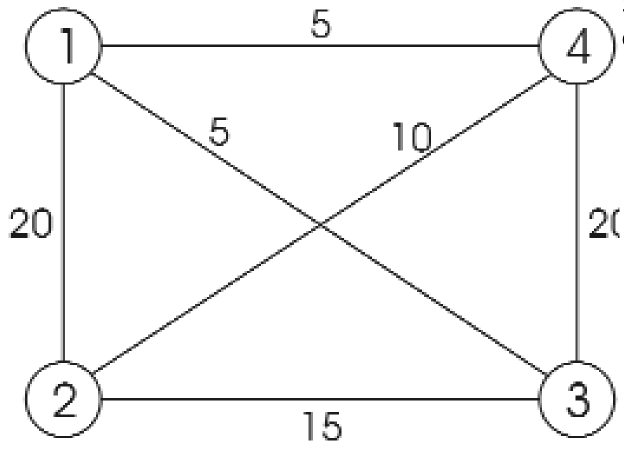


Figure 6.12: Four cities, connections and distances between them.

Solution: TSP is one of the most known programming problems. There aren't no algorithm more efficient than backtracking to solve this problem for an exact solution. TSP is a kind of permutation problem. The backtracking algorithm should search a vector of cities (v1, v2,...,vn) which represents the best route. So the program must generate all the permutations of the cities and keeps the one which has the minimal length. To make the program faster, we can prune if the current length in any step reached the minimal length found so far.

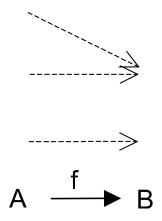
Task 8: Set covering

Problem: A set of subsets S = {S1,S2,..Sm} of the universal set U = {1,2,...,n} is given. What is the smallest subset T of S such that union of the subsets is equal to U.
Solution: We must get the universal set with the minimum number of the subsets. Set covering is a kind of combination problem. The best solution can be one subset. So first we will

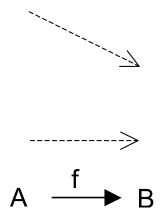
try to cover the universal set only with one of the subsets. For this purpose we will choose all the subsets one by one, until we cover the universal set or there is no subset any more. If we can not cover the universal set with one subset we will try to cover it with two subsets and so on. Each subset we add to the partial solution must cover at least one member of the universal set. If there are some subsets that are subsets of other subsets we can eliminate them at advance.

Task 9: Functions (Surjection, Injection, Bijection)

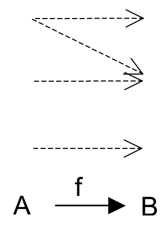
A function f: A -> B is a way of relating every item in a set A with precisely one item in a set B. In particular, the item a Î A is related to an item in B denoted f(a). The set A is called the domain of the function, and the set B is called the range of the function. The image is the set of values in the domain that the function f actually takes on. The image is always a subset of the range.



a function

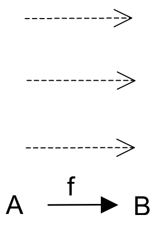


not a function

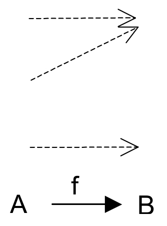


not a function

Figures 6.13 (a,b,c,d)

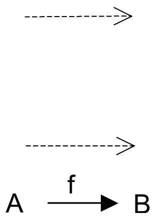


a surjection

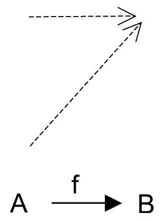


not a surjection

A function f: A -> B is a surjection if for all a Î B, there exists an a Î A such that f(a)= b. In other words, a function f is a surjection if it takes on every value in its range. That is, the image of f is the entire range of f. A surjection function is also sometimes called an onto function.



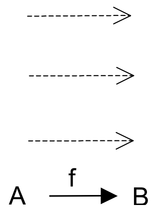
an injection



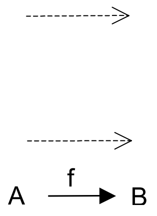
not an injection

A function f: A -> B is an injection if for all a, a'∈A, (a<>a') => (f(a)<>f(a')). Informally, f is an injection if no two items in A are mapped to the same item in B. Note that f need not take on every value in its range; the only restriction is that f cannot take on the same value twice. Injective functions are also called one-to-one.

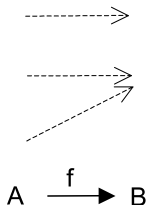
A function f: A -> B is bijection if it is both a surjective and an injection. Sometimes a bijection is also called one-to-one correspondence or



a bijection



not a bijection
(not onto)



not a bijection
(not one-to-one)

one-to-one and onto function.

Problem: In this task you should make three separate programs. Two sets are given: set A= {1,2,...,n} and set B = {1,2,..,m} .

a. Generate all the surjection functions f: A -> B, where n>=m.
b. Generate all the injection functions f: A -> B, where n<=m.
c. Generate all the bijection functions f: A -> B, where n=m.

Solution:

a. Generating Surjection Functions : We can generate all the surjection functions f:A->B by modifying the permutation program. In the permutation program we avoided the duplicated values. We can have duplicated values in generating surjection functions. After we assign the n-th value of vector X[1..n] we check if X contains all the members of set B (that is, the image is equal to the range). If it contains then we have a solution and we display it. In the program, X[i] indicates a member of set B where i indicates a member of set A. In the program Image[1..m] is a vector to keep the occurrences of the members of set B in the vector X. Call the procedure Surjection(1).

```
Procedure Surjection(k:integer);
     if k>n then
          if  IsImageEqualRange then print
     else
         for i:= 1 to m do
              X[k]  := i;
              Inc(Image[i]);
              Surjection(k+1);
              Dec(Image[i]);
```

In this program we exhaustively search all the ways and print the ones that represent the solutions. Before we call the procedure surjection recursively, we can use a criteria to test if a new call leads to a solution. If it doesn't lead to a solution, we can omit it (pruning). In this way our program runs faster. For this test, in a variable (counter), we can keep how many different members of set B are in the vector X. If the condition (n-k >= m-counter) is satisfied, then new procedure calling leads to a solution. Before calling the procedure, initialize the counter to 0.

```
Procedure Surjection(k:integer);
     if k>n then
          if  IsImageEqualRange then print
     else
         for i:= 1 to m do
              X[k]  := i;
            If image[i] = 0 then inc(counter);
              Inc(Image[i]);
              If (n-k >= m-counter) then Surjection(k+1);
              Dec(Image[i]);
            If image[i] = 0 then dec(counter)
```

b. Generating Injection Functions : We can generate all the injection functions f:A->B by modifying the permutation program. Like in permutations we must avoid having duplicated values in the vector X[1..n]. The only difference is that the elements of the vector X may get the values in the range [1..m], instead of [1..n]. To avoid the duplicated values we keep the list of the used members of B in the boolean vector visited[1..m]. Initialize the vector visited to false. Call the procedure Injection(1).

```
Procedure Injection(k:integer);
     if k>n then print
     else
         for i:= 1 to m do
              if not vizited[i] then
                   X[k] := i;
                   vizited[i] := true;
                   Injection(k+1);
                   vizited[i] := false;
```

c. Generating Bijection Functions : Generating bijection functions is the same as generating permutations. The procedure Injection generates all the bijection functions when m=n. So you don't need to make a new program, you can use the same program to generate all the injection and bijection functions

Task 10: Maze

Problem: Given an NxN maze. Find all the path between source and the target positions. The maze is a in the grid format and you can move horizontaly or vertically.

Solution: It is time to help our mouse. Suppose that maze is represented with a two dimentional array X[1..n,1..n]. The matrix X is full of 0s and -1s. 0 represents a space and -1 represents an obstacle in the maze. The mouse may move only in the spaces. To solve this problem first we should number the four directions. Let's number the directions as in picture 12.b. In each crossreads the mouse moves the first untried direction if it is not an obstacle. As soon as it reaches the target position, we display one solution and contine for the next solution. To keep the path, where the mouse came to the target, we number the squares it stepped, starting with one. Call the procedure Maze(xs,ys,1) where xs,ys indicates the source position and 1 indicates the first step.

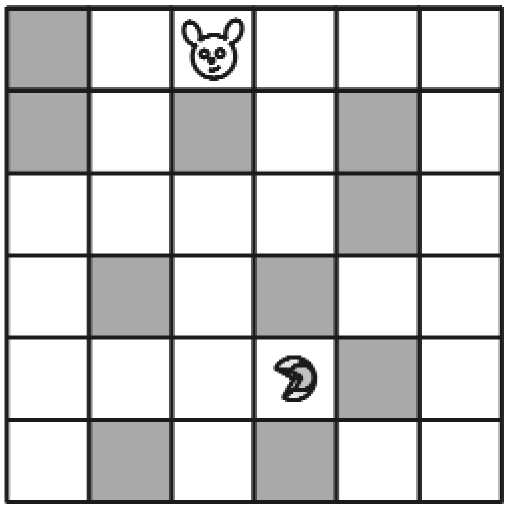


Figure 6.14a: A maze.

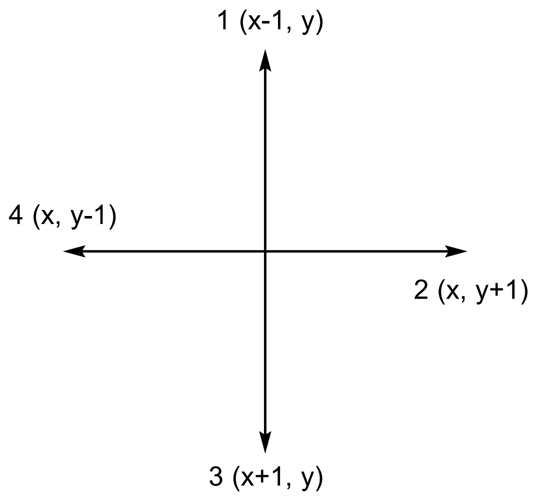


Figure 6.14b: Four directions

```
Procedure Maze(x,y,step);
  If x,y is the target position print X
  Else
```

```
        If x,y is in the maze and x,y is not an obstacle then
             X[x,y] := step;
             Maze(x-1,y,step+1);
             Maze(x,y+1,step+1);
             Maze(x+1,y,step+1);
             Maze(x,y-1,step+1);
             X[x,y] := 0;
```

Task 14: Knigth tour

Problem: Given an NxN chessboard. A knight starts to walk from its initial position and steps all the squares succesively only once. Generate all the possible paths.

Solution: In the maze problem from a current square we may reach to the four new squares. The knight may reach eight new squares. We modify the maze program a little. Call the procedure knight(xs,ys,1) where xs,ys indicates the initial position and 1 indicates the first step.

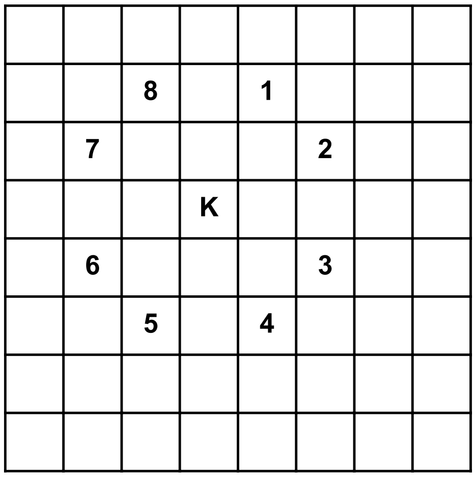


Figure 15: The moves of the knight.

```
procedure knight(i,j,pass : shortint);
     if i in [1..size] and j in [1..n] and board[i,j]=0 then
          board[i,j] := pass;
          if pass = size*size then print; {we have a solution}
          knight (i-2,j+1,pass+1);
          knight (i-1,j+2,pass+1);
          knight (i+1,j+2,pass+1);
          knight (i+2,j+1,pass+1);
          knight (i+2,j-1,pass+1);
          knight (i+1,j-2,pass+1);
          knight (i-1,j-2,pass+1);
          knight (i-2,j-1,pass+1);
          board [i,j] := 0;
```

Programming Projects

Project 1: Safe Breaker (Canadian Olympiads 1996)

Input file: safe.in
Output file: safe.out

We are observing someone playing a game similar to Mastermind (TM). The object of this game is to find a secret code by intelligent guesswork, assisted by some clues. In this case the secret code is a 4-digit number in the inclusive range from 0000 to 9999, say "3321". The player makes a first random guess, say "1223" and then, as for each of the future guesses, gets a clue telling how close the guess is. A clue consists of two numbers: the number of correct digits (in this case, one: the "2" at the third position) and the additional number of digits guessed correctly but in the wrong place (in this case, two: the "1" and the "2"). The clue would in this case be: "1/2". For the guess "1110", the clue would be "0/1", since there are no correct digits and only one misplaced digit. (Notice that there is only one digit 1 misplaced.)
Write a program that, given a set of guesses and corresponding clues, tries to find the secret code.

Input specification
The first line of input specifies the number of test cases (N) your program has to process. Each test case consists of a first line containing the number of guesses G (0 <= G <= 10), and G subsequent lines consisting of exactly 8 characters: a code of four digits, a blank, a digit indicating the number of correct digits, a "/", and a digit indicating the number of correct but misplaced digits.

Output specification
For each test case, the output contains a single line saying either:
impossible if there is no code consistent with all guesses.
n , where n is the secret code, if there is exactly one code consistent with all guesses.
indeterminate if there is more than one code which is consistent with all guesses.

Sample input
```
4
6
9793 0/1
2384 0/2
6264 0/1
3383 1/0
2795 0/0
0218 1/0
1
1234 4/0
1
1234 2/2
2
6428 3/0
1357 3/0
```

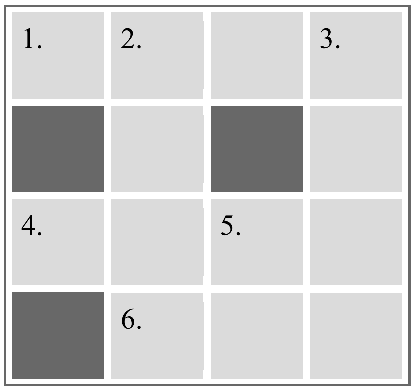
Sample output
```
3411
1234
indeterminate
```

impossible

Project 2: Cross-Number Puzzle (Canadian Olympiads 1998)

Input file: none
Output file (part 1): perfect.out
Output file (part 2): cube.out

Solve the following three problems. Solutions to the first two problems are to be generated using a computer. The solution to the third problem may be done by hand, but must be entered into the grid provided in the information/answer form.
1. Write a program to print the perfect numbers between 1000 and 9999 inclusive. A perfect number is a positive integer which is equal to the sum of its proper divisors. A proper divisor is any divisor less than the number itself. For example, 6 is a perfect number since 1 + 2 + 3 = 6. The numbers should be printed one per line.
2. Write a program to generate all integers between 100 and 999 inclusive which are equal to the sum of the cubes of their digits. The numbers are to be printed one per line.
3. Use the output from parts 1 and 2, and any other programs you care to write, to solve the following cross-number puzzle.



Clues:
ACROSS:
1. A 4-digit perfect number.
4. A 4-digit palindrome. (A palindrome is a number which reads the same right to left as it does left to right. For example: 1221.)
6. A 3-digit number that is equal to the sum of the cubes of its digits.

DOWN:
2. A pair of twin primes. (Primes which differ by 2; for example, 3 and 5 are twin primes.)
3. A 4-digit perfect square.
5. A prime number.

Project 3: Letter Arithmetic (Canadian Olympiads 1999)

Input file : letter.in
Output file : letter.out
A popular form of pencil game is to use letters to represent digits in a mathematical statement. An example is
    SEND

```
 +MORE
 -----
 MONEY
```

which represents

```
   9567
  +1085
  -----
  10652
```

Your task is to read in sets of three "words" and assign unique digits to the letters in such a way as to make the sum of the first two words equal to the third word.
The input file begins with a line containing a positive integer n which is the number of test data sets contained in the file. Each data set consists of three lines, each of which contains one word with the third word being the sum of the first two. The words will contain no more than 20 upper case letters.
The output file is to consist of n sets of lines each containing the numeric representation of each word in the corresponding test data set. There will be exactly one correct solution for each data set. Leave an empty line after the output for each data set.

Sample Input

```
2
SEND
MORE
MONEY
MEND
COPE
CONEY
```

Output for Sample Input

```
9567
1085
10652

9567
1085
10652
```

Project 4: Squares (From British Olympiads 1995)

Given a 5x5 grid, the numbers 1 to 25 are to be placed without repetition in the grid. The placement rule is that given the number k occupying co-ordinates (x, y), 1 <= k < 25, the number k+1 can be placed in any unoccupied square on the board with the following co-ordinates (z, w):
(z, w) = (x +/- 3, y)
(z, w) = (x, y +/- 3)
(z, w) = (x +/- 2, y +/- 2)
For example, given the starting position of 1 at (1, 2), the number 2 can be placed in any of
the squares marked with a *.

1			*	
		*		
*				

Your program should repeatedly offer 3 options: A - Problem A; B - Problem B; X - exit program.

Problem A

You are given a starting position for the number 1. Print to the screen the list of all valid squares in which the number 2 can be placed according to the above rule. The order of this list is unimportant. Append a copy of this to the file 'BIO95R2.OUT'. A sample copy of this output is given below.

Input

```
Starting square in format x y : 1 2
```

Output

```
Valid positions starting from (1, 2):
(1, 5)
(4, 2)
(3, 4)
```
blank line

Problem B

You are again given a starting position in which the number 1 is placed. Evaluate and display the number of valid combinations which result in a complete grid. This means a grid in which all pieces 2 to 25 have been placed from the given starting position following the above rule. You should also give one example complete board. A copy of your output should be appended to the file 'BIO95R2.OUT'.

Input

```
Starting square in format x y : 1 2
```

Output

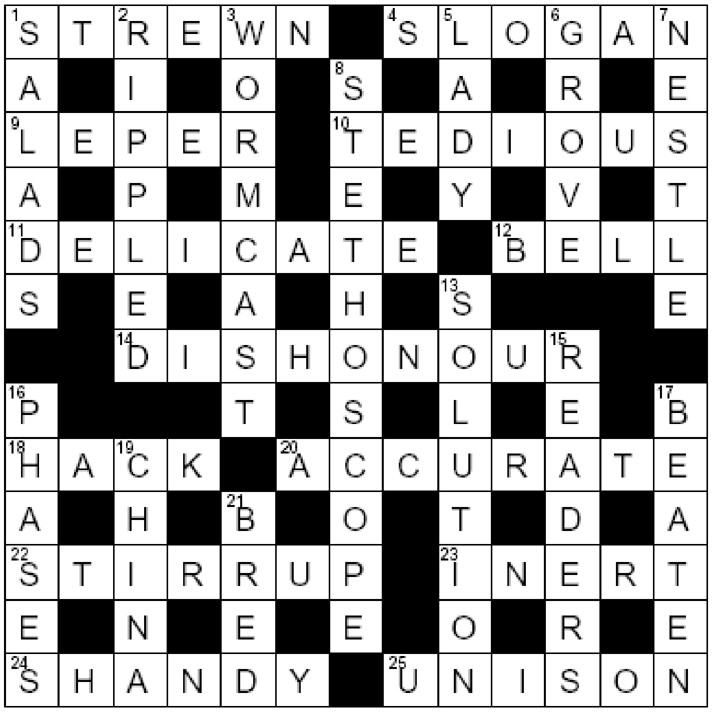
```
Total number of valid combinations starting from (1, 2): 548
Example grid:
16 24 10 17 25
1  13 21 2  12
9  18 5  8  19
15 23 11 14 22
4  7  20 3  6
```
blank line

Project 4: Reverse Crossword (From British Olympiads 1997)

A conventional crossword is written on a rectangular grid of `black' and `white' squares. The white squares are ultimately filled with letters which, read across or down, make words corresponding to given clues. Across and down clues are listed
separately. They are ordered based on the position in the grid of the first letter of their solution; left to right then up to down. An example grid, completed with solution words, is shown below. Some squares have also been given numbers. If you list the across words in the normal crossword order, the numbers for their starting squares will be in increasing order. Similarly for the down words. (see figure below)
You are to write a program which will produce the crossword grid, with the solution words filled in. You will be given the size of a square grid, and sorted lists of the across words and down words, but you will not be given the position of the `black' squares.
The first line of the input will contain 3 numbers; the first is the size of the grid (< 20), the second the number of across words, and the third the number of down words. There will then follow (one on each line) the across words in normal crossword order, followed by the

down words in normal crossword order. Your output should consist of the crossword grid completed with the solution words, such that :

- All the words that are listed should appear in the solution, going in the correct direction (across or down) and in the correct order.
- There should be no other words visible. A word is any combination of 2 or more letters across or down. Diagonally adjacent letters do not count.
- The grid of `black' and `white' squares must have 180 ffi rotational symmetry. If you do not believe there is a solution you should print out `Impossible'.



1 S T 2 R E 3 W N 4 S 5 L O 6 G A 7 N
A I O 8 S A R E
9 L E P E R 10 T E D I O U S
A P M E Y V T
11 D E L I C A T E 12 B E L L
S E A H 13 S E
14 D I S H O N O U 15 R
16 P T S L E 17 B
18 H A 19 C K 20 A C C U R A T E
A H 21 B O T D A
22 S T I R R U P 23 I N E R T
E N E E O R E
24 S H A N D Y 25 U N I S O N

Sample Input

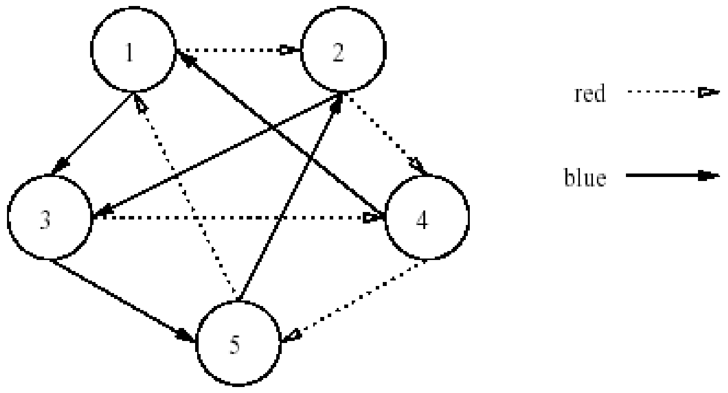
```
13 13 13
STREWN
SLOGAN
LEPER
TEDIOUS
DELICATE
BELL
DISHONOUR
HACK
ACCURATE
STIRRUP
INERT
SHANDY
UNISON
SALADS
RIPPLED
WORMCAST
LADY
GROVE
NESTLE
STETHOSCOPE
SOLUTION
READERS
PHASES
BEATEN
CHINA
BRED
```

Sample Output

```
STREWN.SLOGAN
A.I.O.S.A.R.E
LEPER.TEDIOUS
A.P.M.E.Y.V.T
DELICATE.BELL
S.E.A.H.S...E
..DISHONOUR..
P...T.S.L.E.B
HACK.ACCURATE
A.H.B.O.T.D.A
STIRRUP.INERT
E.N.E.E.O.R.E
SHANDY.UNISON
```

Project 4: Coloured Routes (From British Olympiads 1998)

A groups of towns is connected together by a rail network. To help travel aroundthe network some routes have been given colours; each station has one `red' route leaving it, and one `blue' route. Routes have an associated direction, so a route from A to B does not necessarily mean there is a corresponding route from B to A. You are trying to give directions to a friend, so that you can meet them at a station of your choosing. Unfortunately you do not know which station your friend will start from, and due to their identical layouts they cannot be distinguished. To direct your friend, you need to give them a sequence of coloured routes to take, so that they will finish at the your chosen station irrespective of where they start. Due to limitations on the rail tickets and the placement of barriers, it is not enough that they pass through this station before the end of the sequence.
Write a program which reads in details of the rail network, and returns a sequence of coloured routes. To make life simple for your friend, you should return the shortest route possible.
The first line of your input will be an integer 2 x 16, specifying how many towns are connected in the network. There will then follow x lines of two integers; the first integer on the ith of these lines is the destination of the red route from town i, and the second the destination of the blue route. Towns are numbered from 1 to x.
You should output the length of the shortest possible route sequence, followed by an example sequence, using r and b to denote the two types of route, and the town your friend will finish in.
If there is no possible route sequence you should just output Impossible.



Sample Input
```
5
2 3
4 3
4 5
5 1
1 2
```

Sample Output
```
6
rbrrbb
3
```

Project 4: Dominoes (From British Olympiads 1998)

A domino is a 21 rectangle with 0,1,2,3,4,5 or 6 written on each 11 half. A set of dominoes consists of the 28 unique dominoes that can be constructed. A set of dominoes has been fitted into an 87 grid. You will be given the values on each square, but not the boundary of the dominoes. Your task is to determine where the dominoes are positioned.
Input will consist of seven rows of eight numbers, representing the values on the grid. For each possible arrangement of dominoes you should output the layout, and finally print the number of solutions you have found.

If you do not believe there is a solution you should print out `Impossible'.

Sample Input
```
3 5 1 2 6 3 5 6
0 3 0 4 6 1 0 4
```

```
5 6 6 3 0 6 0 4
5 6 5 3 1 0 4 1
2 2 2 2 0 3 2 1
1 1 4 1 3 4 0 5
4 5 3 6 2 5 2 4
```

<u>Sample Output</u>

```
3 5 1-2 6-3 5-6
| |
0 3 0 4-6 1 0 4
    |     | | |
5 6 6 3 0 6 0 4
| |   | |
5 6 5 3 1 0-4 1
|       |
2-2 2 2-0 3-2 1

1 1 4 1-3 4 0-5
| | |     |
4 5 3 6-2 5 2-4

3-5 1-2 6-3 5-6

0-3 0 4-6 1 0 4
|         | | |
5 6 6 3 0 6 0 4
| |   | |
5 6 5 3 1 0-4 1
|       |
2-2 2 2-0 3-2 1

1 1 4 1-3 4 0-5
| | |     |
4 5 3 6-2 5 2-4
2 solutions found
```

Project 8 Magic (Singapore Olympiads)

A magic square is an N × N matrix such that
Every entry of the matrix is an integer between 1 and N2 inclusively.
The entries of the matrix are distinct.
The N row sums, the N column sums, and the two main diagonal sums are all equal.

For example,

8	3	14
1	5	9
6	7	2

Is a 3 × 3 matrix that is a magic square. The three row sums are 8+3+4, 1+5+9, 6+7+2; the three column sums are 8+1+6, 3+5+7, 4+9+2; the two main diagonal sums are 8+5+2, 4+5+6. All these sums are 15.

You are to find out if the remaining entries of a partially filled N × N matrix can be completed so that the matrix becomes a magic square.

Example 1: The partially filled matrix

	1		24	
		8		
	9			
	10	21		
				11

can be completed to become

2	1	18	24	20
25	23	8	4	5
16	9	12	13	15
3	10	21	17	14
19	22	6	7	11

It can be checked the five row sums, the five column sums, and the two main diagonal sums are all 65. Furthermore, all entries are distinct with values from 1 to 25 inclusively. Thus the given partially filled matrix can become a magic square.

Requirements
Read the input file MAGIC.IN to obtain the size of the matrix and the values of the filled entries.
Check if the partially filled matrix can be completed to become a magic square.
Write the word “yes” or “no” to the output file MAGIC.OUT accordingly.

Input, Output Files: MAGIC.IN, MAGIC.OUT
The first line of the input file consists of two integers: the first integer N (2 <= N <= 5) is the number of rows (or columns) of the partially filled matrix, the second integer E is the number of the filled entries of the partially filled matrix. Each of the remaining E lines of the input file consists of three integers with a space between two adjacent integers: the row index R (1 <= R <= N), the column index C (1 <= C <= N), and the value V (1 <= V <= N2) of the filled entry. All the V’s are distinct.

The output file contains only one word: “yes” if the given matrix can be completed to become a magic square, “no” otherwise.

The input and output files of Example 1 are:

```
MAGIC.IN      MAGIC.OUT
5 7           yes
1 4 24
4 2 10
```

```
5 5 11
2 3 8
3 2 9
1 2 1
4 3 21
```

Project 9 Rectangles (Bulgarian Olympiads 1996)

Given are no more than 50 rectangles. Is it possible to cover some square using the rectangles by rotating and shifting them? It is not allowed to overlape rectangles as well to remain gaps on the square. Every rectangle must participate in the covering and no part of any rectangle should go out of the square.

The input file INPUT2.TXT contains several sets of test data. Each row of the file consists of two nonnegative integers meaning the length and the width of the subsequent rectangle. Every data set ends by a pair of zeros.

Output file OUTPUT2.TXT must contain one number in each its row for every data set and its value must be equal to the length of the side of the square, if the covering is possible, and zero, otherwise.

Project 10 Strings (Bulgarian Olympiads 1997)

A string may contain only 3 type of characters: A, B and C. Write a program, that finds a string containing N characters (3 <= N <= 97), so that the string does not contain any two identical adjacent substrings.

Input data: File INPUT5.TXT containing one integer N.

Output: File OUTPUT5.TXT must contain only one row with the output string.

An Example:

INPUT5.TXT	OUTPUT5.TXT
7	ABACBAB

Project 11 Servicing Stations (Bulgarian Olympiads 1999)

A company offers personal computers for sale in N towns (3 <= N <= 35). The towns are denoted by 1, 2, ..., N. There are direct routes connecting M pairs from among these towns. The company decides to build servicing stations in several towns, so that for any town X, there would be a station located either in X or in some immediately neighbouring town of X. Write a program BAS.EXE for finding out the minumum number of stations, which the company has to build, so that the above condition holds.
The input data are read from text file BAS.INP. In the first row of this file, there are written number N of towns and number M of pairs of towns directly connected each other. The integers N and M are separated by a space. Every one of the next M rows contains a pair of

connected towns, one pair per row. The pair consists of two integers for town's numbers, separated by a space.
The output must be written into a text file BAS.OUT with a single row containing the obtained minimum.

An example:

```
Input:
8 12
1 2
1 6
1 8
2 3
2 6
3 4
3 5
4 5
4 7
5 6
6 7
6 8
Output:
2
```

Project 12 Formula - 9 (Balkan Olympiads 1999)

The Formula-9 series consists of R car races taking place over one season (1 <= R <= 5). All D drivers competing in the series run in every race (2 <= D <= 8). In a single race, only the drivers taking the top T places (1 <= T <= min(4,D)) gain points; the others get zeros. The driver with the maximum total points at the end of the series wins the Formula-9 trophy of that season (although in this question we are not concerned about who wins the trophy). Given the final result (i.e. total points of all D drivers at the end of the series), your task is to determine the number of all possible ways of reaching this result. A possible way of reaching the final result can be represented by a "score table" whose columns represents the drivers and whose rows represent the races. More precisely, the d'th element of the r'th row is the points gained by the driver d in the r'th race (1 <= d <= D, 1 <= r <= R). Notice that the sum of the column d is the total points gained by the driver d by the end of the series (1 <= d <= D). Thus your task can also be stated as one of determining the number of all score tables whose column sums are the same as the given final result.

Input
The input file has three lines. The first line contains three integers in following order: the number of drivers (D), the number of races (R) and the number of top finishers (T), each separated by a blank (1 <= R <= 5, 2 <= D <= 8, 1 <= T <= min(4,D)). The second line holds a sequence of T distinct positive integers <= 15 in descending order, each separated by a blank. These are the points to be awarded to the top finishers, starting from the first place and ending at the T'th place. The third line contains the result of the series given as D non-negative integers, each separated by a blank. The d'th integer on the third line is the final score of the driver d. The input is a text file named f9.inp.

Output
The output file must contain a single line having a non-negative integer, which is the number of possible ways of reaching the given final result. The output file must be a text file

named f9.out.

Example

f9.inp:
4 3 3
7 4 1
9 9 14 4

f9.out:
6

The 6 possible score tables (not to appear on the output):

1 7 0 4	4 1 7 0	4 1 7 0	7 1 0 4	1 4 7 0	1 4 7 0
4 1 7 0	1 7 0 4	4 1 7 0	1 4 7 0	7 1 0 4	1 4 7 0
4 1 7 0	4 1 7 0	1 7 0 4	1 4 7 0	1 4 7 0	7 1 0 4

Project 13 Game on chessboard (Central European Olympiads 1998)

On the chessboard of size 4x4 there are 8 white and 8 black stones, i.e. one stone on each field. Such a configuration of stones is called a game position. Two stones are adjacent if they are on fields with a common side (i.e. they are adjacent in either horizontal or vertical direction but not diagonal). It means that each stone has at most 4 neighbours. The only legal move in our game is exchanging any two adjacent stones. Your task is to find the shortest sequence of moves transforming a given starting game position into a given final game position.

Input:
The starting game position is described in the first 4 lines of input file GAME.IN. There are 4 symbols in each line, which define the colour of each stone in the row from the left to the right. The lines describe the rows of the chessboard from the top to the bottom. Symbol `0' means a white stone and symbol `1' a black one. There is no space symbol separating the symbols in the line. The fifth line is empty. The next 4 following lines describe the final game position in the same way.

Output:
The first line of output file GAME.OUT contains the number of the moves. The following lines describe the sequence of moves during the game. One line describes one move and it contains 4 positive integers R_1 C_1 R_2 C_2 separated by single spaces. These are the coordinates of the adjacent fields for the move, i.e. fields [R_1,C_1] and [R_2,C_2], where R_1 (or R_2) is the number of the row and C_1 (or C_2) is the number of the column. The rows on the chessboard are numbered from 1 (top row) to 4 (bottom row) and the columns are numbered from 1 (the leftmost column) to 4 (the rightmost one) as well (i.e. the coordinates of the left upper field are [1,1]). If there are multiple shortest sequences of moves transforming the starting position to the final position, you can output any one of them.

Example:
GAME.IN

```
1111
0000
1110
0010

1010
0101
1010
0101
```

Project 14 Orders (Central European Olympiads 1998)

The stores manager has sorted all kinds of goods in an alphabetical order of their labels. All the kinds having labels starting with the same letter are stored in the same warehouse (i.e. in the same building) labelled with this letter. During the day the stores manager receives and books the orders of goods which are to be delivered from the store. Each order requires only one kind of goods. The stores manager processes the requests in the order of their booking.
You know in advance all the orders which will have to be processed by the stores manager today, but you do not know their booking order. Compute all possible ways of the visits of warehouses for the stores manager to settle all the demands piece after piece during the day.

Input:
Input file ORDERS.IN contains a single line with all labels of the requested goods (in random order). Each kind of goods is represented by the starting letter of its label. Only small letters of the English alphabet are used. The number of orders doesn't exceed 200.

Output:
Output file ORDERS.OUT will contain all possible orderings in which the stores manager may visit his warehouses. Every warehouse is represented by a single small letter of the English alphabet -- the starting letter of the label of the goods. Each ordering of warehouses is written in the output file only once on a separate line and all the lines containing orderings have to be sorted in an alphabetical order (see the example). No output will exceed 2 megabytes.

Example:
ORDERS.IN
```
bbjd
```

ORDERS.OUT
```
bbdj
bbjd
bdbj
bdjb
bjbd
bjdb
dbbj
dbjb
djbb
jbbd
jbdb
jdbb
```

Project 15 Hamilton's Robot (IOI 1992)

On a plane there are given N positions P1, P2, ..., PN with integer coordinates (X1,Y1), (X2,Y2), ..., (XN,YN).

A robot should move through all these positions starting at P1. It should come to each position only once with the exception of P1 which also has to be the position at the end of the tour.

There are constraints on the robot's movements. It can only move along straight lines. From P1 it can start in any direction. Reaching one of the Pi, before moving on to another position it must turn 90 degrees either to the left or to the right.

A robot program consists of five types of statements:

1. "ORIENTATION Xk Yk"	: usable as the first statement only.The robot turns to the direction of the position Pk (k between 2 and N).
2. "MOVE-TO Xj Yj"	: if the robot can reach Pj without changing its current orientation, then it moves to the position Pj (j between 1 and N). Otherwise the statement is not executable.
3. "TURN-LEFT"	: the robot changes its orientation 90 degrees to the left.
4. "TURN-RIGHT"	: the robot changes its orientation 90 degrees to the right.
5. "STOP"2	: deactivates the robot. This is the necessary last statement of each robot program.

Problem statement

Implement a program that does the following:
1. Read the value of N and the coordinates for N given positions from an ASCII input file (see Example) and display the data on the screen.
2. Develop a robot program for a valid tour through all positions (as defined above) if one exists.
3. If there is no possible tour, the robot program must consist just of the "STOP"-statement.
4. Display on the screen, whether a tour is possible or not and, if there exists one, its length (rounded, 2 digits after the decimal point). The length of a tour the sum of the lengths of the straight line pieces.
5. Write the robot program to an ASCII output file exactly as is shown in Example.

Tehnical constraints
Constraint-1: Put your solution program into an ASCII text file named
"C:\IOI\DAY-2\421-PROG.xxx". Extension .xxx is:
- .BAS for BASIC programs, .C for C programs,
- .LCN for LOGO programs, .PAS for PASCAL programs.
Constraint-2: The name of the ASCII input file for reading the
positions from must be "C:\IOI\DAY-2\421-ROBO.IN".
Constraint-3: The name of the ASCII output file for writing the robot
program to must be "C:\IOI\DAY-2\421-ROBO.OU".
Constraint-4: Program must reject inputs where N is less than 4 or
greater than 16, without trying to find a tour!

Example(s)
Input: An input file contains in the first line the value for N and in the following N lines the X and Y coordinates of the selected positions, for example:

```
4
2 -2
0 2
-1 -1
3 1
```

Output: For these 4 positions one shortest robot program with length = 12.65 is:

```
ORIENTATION 3 1
MOVE-TO 3 1
TURN-LEFT
MOVE-TO 0 2
TURN-LEFT
MOVE-TO -1 -1
TURN-LEFT
MOVE-TO 2 -2
STOP
```

Credits

Read input data correctly from every file and display it	5 points
Algorithm for computing a valid tour ok	30 points
Generated robot program syntactically correct, if tour does not exist	10 points
Generated robot program syntactically correct, if tour does exist	15 points
Screen display gives all required information	5 points
Displayed length of computed tour correct	10 points
Robot program correctly written to a file	10 points
Technical constraints obeyed	15 points
maximal	100 points

Project 16 The Clocks (IOI 1994)

```
|-------|   |-------|   |-------|
|       |   |       |   |   |   |
|---O   |   |---O   |   |   O   |
|       |   |       |   |       |
|-------|   |-------|   |-------|
    A           B           C

|-------|   |-------|   |-------|
|       |   |       |   |       |
|   O   |   |   O   |   |   O   |
|   |   |   |   |   |   |   |   |
|-------|   |-------|   |-------|
    D           E           F

|-------|   |-------|   |-------|
|       |   |       |   |       |
|   O   |   |   O---|   |   O   |
|   |   |   |       |   |   |   |
|-------|   |-------|   |-------|  (Figure 1)
    G           H           I
```

There are nine clocks in a 3*3 array (figure 1). The goal is to return all the dials to 12 o'clock with as few moves as possible. There are nine different allowed ways to turn the dials on the clocks. Each such way is called a move. Select for each move a number 1 to 9. That number will turn the dials 90' (degrees) clockwise on those clocks which are affected according to figure 2 below.

Input Data
Read nine numbers from the INPUT.TXT file. These numbers give the start positions of the dials. 0=12 o'clock, 1=3 o'clock, 2=6 o'clock, 3=9 o'clock. The example in figure 1 gives the following input data file:

```
3 3 0
2 2 2
2 1 2
```

Output Data
Write to the OUTPUT.TXT file a shortest sequence of moves (numbers), which returns all the dials to 12 o'clock. In case there are many solutions, only one is required. In our example the OUTPUT.TXT file could look as follows:
5849

Move	Affected clocks
1	ABDE
2	ABC
3	BCEF
4	ADG
5	BDEFH
6	CFI
7	DEGH
8	GHI
9	EFHI

Example of Method
Each number represents a time accoring to following table:
0 = 12 o'clock
1 = 3 o'clock
2 = 6 o'clock
3 = 9 o'clock

```
3 3 0        3 0 0        3 0 0         0 0 0        0 0 0
2 2 2  5->   3 3 3  8->   3 3 3  4 ->   0 3 3  9->   0 0 0
2 1 2        2 2 2        3 3 3         0 3 3        0 0 0
```

Project 17 Packing Rectangles (IOI 1995)

Four rectangles are given. Find the smallest enclosing (new) rectangle into which these four may be fitted without overlapping. By smallest rectangle we mean the one with the smallest area.
All four rectangles should have their sides parallel to the corresponding sides of the enclosing rectangle.
There may exist several different enclosing rectangles fulfilling the requirements, all with the same area. You have to produce all such enclosing rectangles.

Input Data
The input file INPUT.TXT consists of four lines. Each line describes one given rectangle by

two positive integers: the lengths of the sides of the rectangle. Each side of a rectangle is at least 1 and at most 50.

Output Data
The output file OUTPUT.TXT should contain one line more than the number of solutions. The first line contains a single integer: the minimum area of the enclosing rectangles (Subtask A). Each of the following lines contains one solution described by two numbers p and q with p<=q (Subtask B). These lines must be sorted in ascending order of p, and must all be different.

Example Input and Output

INPUT.TXT	OUTPUT.TXT
1 2	40
2 3	4 10
3 4	5 8
4 5	

Project 18 Party Lamps (IOI 1998)

To brighten up the gala dinner of the IOI'98 we have a set of N coloured lamps numbered from 1 to N. The lamps are connected to four buttons:
button 1 -- when this button is pressed, all the lamps change their state: those that are ON are turned OFF and those that are OFF are turned ON.
button 2 -- changes the state of all the odd numbered lamps.
button 3 -- changes the state of all the even numbered lamps.
button 4 -- changes the state of the lamps whose number is of the form 3K+1 (with K ³0), i.e., 1,4,7,...
There is a counter C which records the total number of button presses. When the party starts, all the lamps are ON and the counter C is set to zero.

Task
You are given the value of counter C and information on the final state of some of the lamps. Write a program to determine all the possible final configurations of the N lamps that are consistent with the given information, without repetitions.

Input Data
The file named PARTY.IN with four lines, describing the number N of lamps available, the number C of button presses, and the state of some of the lamps in the final configuration. The first line contains the number N and the second line the final value of counter C. The third line lists the lamp numbers you are informed to be ON in the final configuration, separated by one space and terminated by the integer -1. The fourth line lists the lamp numbers you are informed to be OFF in the final configuration, separated by one space and terminated by the integer -1.

Sample Input:
```
10
1
-1
7 -1
```
In this case, there are 10 lamps and only one button has been pressed. Lamp 7 is OFF in the final configuration.

Output Data
The file PARTY.OUT must contain all the possible final configurations (without repetitions) of all the lamps. Each possible configuration must be written on a different line. Configurations

may be listed in arbitrary order. Each line has N characters, where the first character represents the state of lamp 1 and the last character represents the state of lamp N. A 0 (zero) stands for a lamp that is OFF, and a 1 (one) stands for a lamp that is ON.

Sample Output:
```
0000000000
0110110110
0101010101
```
In this case, there are three possible final configurations: either all lamps are OFF; or lamps 1, 4, 7, 10 are OFF, and lamps 2, 3, 5, 6, 8, 9 are ON; or lamps 1, 3, 5, 7, 9 are OFF, and lamps 2, 4, 6, 8, 10 are ON.

Constraints
The parameters N and C are constrained by:
10 <= N <= 100
1 <= C <= 10000
The number of lamps you are informed to be ON, in the final configuration, is less than or equal to 2. The number of lamps you are informed to be OFF, in the final configuration, is less than or equal to 2.
There is at least one possible final configuration for each input test file.

Project 19 Depot (IOI 2001)

A Finnish high technology company has a big rectangular depot. The depot has a worker and a manager. The sides of the depot, in the order around it, are called left, top, right and bottom. The depot area is divided into equal-sized squares by dividing the area into rows and columns. The rows are numbered starting from the top with integers 1,2,... and the columns are numbered starting from the left with integers 1,2,... The depot has containers, which are used to store invaluable technological devices. The containers have distinct identification numbers. Each container occupies one square. The depot is so big, that the number of containers ever to arrive is smaller than the number of rows and smaller than the number of columns. The containers are not removed from the depot, but sometimes a new container arrives. The entry to the depot is at the top left corner. The worker has arranged the containers around the top left corner of the depot in such a way that he will be able to find them by their identification numbers. He uses the following method.
Suppose that the identification number of the next container to be inserted is k (container k, for short). The worker travels the first row starting from the left and looks for the first container with identification number larger than k. If no such container is found, then container k is placed immediately after the rightmost of the containers previously in the row. If such a container l is found, then container l is replaced by container k, and l is inserted to the following row using the same method. If the worker reaches a row having no containers, the container is placed in the leftmost square of that row.

Suppose that containers 3,4,9,2,5,1 have arrived to the depot in this order. Then the placement of the containers at the depot is as follows.
```
1 4 5
2 9
3
```

The manager comes to the worker and they have the following dialogue:
Manager: Did container 5 arrive before container 4?
Worker: No, that is impossible.
Manager: Oh, so you can tell the arrival order of the containers by their placement.

Worker: Generally not. For instance, the containers now in the depot could have arrived in the order 3,2,1,4,9,5 or in the order 3,2,1,9,4,5 or in one of 14 other orders.
As the manager does not want to show that the worker seems much smarter, he goes away. You are to help the manager and write a program which, given a container placement, computes all possible orders in which they might have arrived.

Input
The input file name is depot.in. The first line contains one integer R: the number of rows with containers in them. The following R lines contain information about rows 1,...,R starting from the top as follows. First on each of those lines is an integer M: the number of containers in that row. Following that, there are M integers on the line: the identification numbers of the containers in the row starting from the left. All container identification numbers I satisfy 1 ¡Ü I ¡Ü 50. Let N be the number of containers in the depot, then 1 ¡Ü N ¡Ü 13.

Output
The output file name is depot.out. The output file contains as many lines as there are possible arrival orders. Each of these lines contains N integers: the identification numbers of the containers in the potential arrival order described by that line. All lines describe an arrival order not described in any other line.

Example

Example 1:

depot.in

```
3
3 1 4 5
2 2 9
1 3
```

depot.out

```
3 2 1 4 9 5
3 2 1 9 4 5
3 4 2 1 9 5
3 2 4 1 9 5
3 2 9 1 4 5
3 9 2 1 4 5
3 4 2 9 1 5
3 4 9 2 1 5
3 2 4 9 1 5
3 2 9 4 1 5
3 9 2 4 1 5
3 4 2 9 5 1
3 4 9 2 5 1
3 2 4 9 5 1
3 2 9 4 5 1
3 9 2 4 5 1
```

Example 2:

depot.in

```
2
2 1 2
1 3
```

depot.out

```
3 1 2
1 3 2
```

Scoring
If the output file contains impossible orders or no orders at all, your score is 0 for that test case. Otherwise the score for a test case is computed as follows. If the output file

contains all possible orders exactly once, your score is 4. If the output file contains at least half of the possible orders and each of them exactly once, your score is 2. If the output file contains less than half of the possible orders or some of them appear more than once, your score is 1.

Project 20 Puzzle (ACM 1993)

A children’s puzzle that was popular 30 years ago consisted of a 5¥5 frame which contained 24 small squares of equal size. A unique letter of the alphabet was printed on each small square. Since there were only 24 squares within the frame, the frame also contained an empty position which was the same size as a small square. A square could be moved into that empty position if it were immediately to the right, to the left, above, or below the empty position. The object of the puzzle was to slide squares into the empty position so that the frame displayed the letters in alphabetical order.
The illustration below represents a puzzle in its original configuration and in its configuration after the following sequence of 6 moves:
1) The square above the empty position moves.
2) The square to the right of the empty position moves.
3) The square to the right of the empty position moves.
4) The square below the empty position moves.
5) The square below the empty position moves.
6) The square to the left of the empty position moves.

T	R	G	S	J
X	D	O	K	I
M	▨	V	L	N
W	P	A	B	E
U	Q	H	C	F

Original puzzle configuration.

T	R	G	S	J
X	O	K	L	I
M	D	V	B	N
W	P	▨	A	E
U	Q	H	C	F

Puzzle configuration after the sequence of described moves.

Input
Input for your program consists of several puzzles. Each is described by its initial configuration and the sequence of moves on the puzzle. The first 5 lines of each puzzle description are the starting configuration. Subsequent lines give the sequence of moves.
The first line of the frame display corresponds to the top line of squares in the puzzle. The other lines follow in order. The empty position in a frame is indicated by a blank. Each display line contains exactly 5 characters, beginning with the character on the leftmost square (or a blank if the leftmost square is actually the empty frame position). The display lines will correspond to a legitimate puzzle.
The sequence of moves is represented by a sequence of As, Bs, Rs, and Ls to denote which square moves into the empty position. A denotes that the square above the empty position moves; B denotes that the square below the empty position moves; L denotes that

the square to the left of the empty position moves; R denotes that the squareto the right of the empty position moves. It is possible that there is an illegal move, even when it is represented by one of the 4 move characters. If an illegal move occurs, the puzzle is considered to have no final configuration.
This sequence of moves may be spread over several lines, but it always ends in the digit 0.
The end of data is denoted by the character Z.

Output
Output for each puzzle begins with an appropriately labeled number (Puzzle #1, Puzzle #2, etc.). If the puzzle has no final configuration, then a message to that effect should follow. Otherwise that final configuration should be displayed.
Format each line for a final configuration so that there is a single blank character between two adjacent letters.
Treat the empty square the same as a letter. For example, if the blank is an interior position, then it will appear as a sequence of 3 blanks—one to separate it from the square to the left, one for the empty position itself, and one to separate it from the square to the right.
Separate output from different puzzle records by at least one blank line.
Sample input and the corresponding correct output are shown below. The first record corresponds to the puzzle illustrated below.

Sample input

```
TRGSJ
XDOKI
M VLN
WPABE
UQHCF
ARRBBL0
ABCDE
FGHIJ
KLMNO
PQRS
TUVWX
AAA
LLLL0
ABCDE
FGHIJ
KLMNO
PQRS
TUVWX
AAAAABBRRRLL0
Z
```

Output for the Sample Input

```
Puzzle #1:
T R G S J
X O K L  I
M D V B N
W P   A E
0  U Q H C

Puzzle #2:
  A  B  C  D
F G  H  I   E
K L  M  N   J
P Q R  S   O
T U  V  W  X

Puzzle #3:
This puzzle has no final
configuration.
```

Project 21 Trade On Verweggistan (ACM 1999)

Since the days of Peter Stuyvesant and Abel Tasman, Dutch merchants have been traveling all over the world to buy and sell goods. Once there was some trade on Verweggistan, but it ended after a short time. After reading this story you will understand why.
At that time Verweggistan was quite popular, because it was the only place in the world where people knew how to make a ‘prul’. The end of the trade on Verweggistan meant the end of the trade in pruls (or ‘prullen’, as the Dutch plural said), and very few people nowadays know what a prul actually is.

Pruls were manufactured in workyards. Whenever a prul was finished it was packed in a box, which was then placed on top of the pile of previously produced pruls. On the side of each box the price was written. The price depended on the time it took to manufacture the prul. If all went well, a prul would cost one or two florins, but on a bad day the price could easily rise to 15 florins or more. This had nothing to do with quality; all pruls had the same value.
In those days pruls sold for 10 florins each in Holland. Transportation costs were negligible since the pruls were taken as extra on ships that would sail anyway. When a Dutch merchant went to Verweggistan, he had a clear purpose: buy pruls, sell them in Holland, and maximize his profits. Unfortunately, the Verweggistan way of trading pruls made this more complicated than one would think.
One would expect that merchants would simply buy the cheapest pruls, and the pruls that cost more than 10 florins would remain unsold. Unfortunately, all workyards on Verweggistan sold their pruls in a particular order. The box on top of the pile was sold first, then the second one from the top, and so on. So even if the fifth box from the top was the cheapest one, a merchant would have to buy the other four boxes above to obtain it.
As you can imagine, this made it quite difficult for the merchants to maximize their profits by buying the right set of pruls. Not having computers to help with optimization, they quickly lost interest in trading pruls at all.
In this problem, you are given the description of several workyard piles. You have to calculate the maximum profit a merchant can obtain by buying pruls from the piles according to the restrictions given above. In addition, you have to determine the number of pruls he has to buy to achieve this profit.

<u>Input</u>
The input describes several test cases. The first line of input for each test case contains a single integer Z, the number of workyards in the test case (1 £ Z £ 50).
This is followed by Z lines, each describing a pile of pruls. The first number in each line is the number E of boxes in the pile (0 £ E £ 20). Following it are E positive integers, indicating the prices (in florins) of the pruls in the stack, given from top to bottom.
The input is terminated by a description starting with Z = 0. This description should not be processed.

<u>Output</u>
For each test case, print the case number (1, 2, …). Then print two lines, the first containing the maximum profit the merchant can achieve. The second line should specify the number of pruls the merchant has to buy to obtain this profit. If this number is not uniquely determined, print the possible values in increasing order. If there are more than ten possible values, print only the 10 smallest.

Display a blank line between test cases.

Sample Input

```
1
6 12 3 10 7 16 5
2
5 7 3 11 9 10
9 1 2 3 4 10 16 10 4 16
0
```

Output for the sample input

```
Workyards 1
Maximum profit is 8.
Number of pruls to buy: 4

Workyards 2
Maximum profit is 40.
Number of pruls to buy: 6 7 8 9 10 12 13
```

Project 22 Crossword Puzzle (ACM 2001)

Your brilliant but absent-minded uncle believes he has solved a difficult crossword puzzle but has misplaced the solution. He needs your help to reconstruct the solution from a list that contains all the words in the solution, plus one extra word that is not part of the solution. Your program must solve the puzzle and print the extra word.
The crossword puzzle is represented by a grid with ten squares on each side. Figure 1 shows the top left corner of a puzzle. The puzzle has a certain number of “slots” where a word can be placed. Each slot is represented by the row and column number of the square where the slot begins, and the direction in which the slot extends from its initial square (“across” or “down”). The length of each slot is not specified. The puzzle has a list of candidate words, all but one of which is used in solving the puzzle.

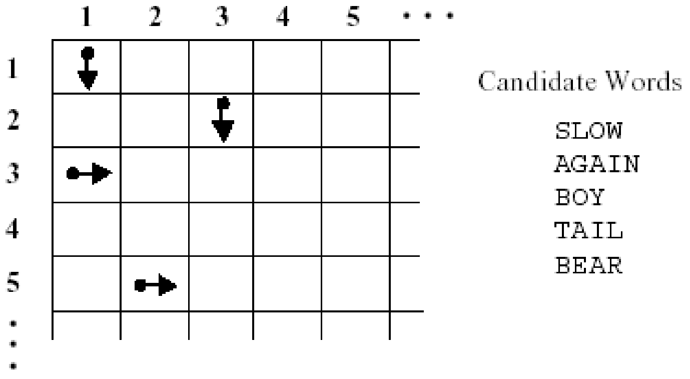


Figure 1: Corner of Example Puzzle

Figure 2 shows a solution to the example puzzle in Figure 1. In a valid solution, each slot is filled with a candidate word. Every maximal horizontal or vertical sequence of two or more letters must be a word in the input. Any candidate word can be used in any slot as long as the word fits in the puzzle and does not conflict with any other word. In the example, all the candidate words are used except the word “BOY”.

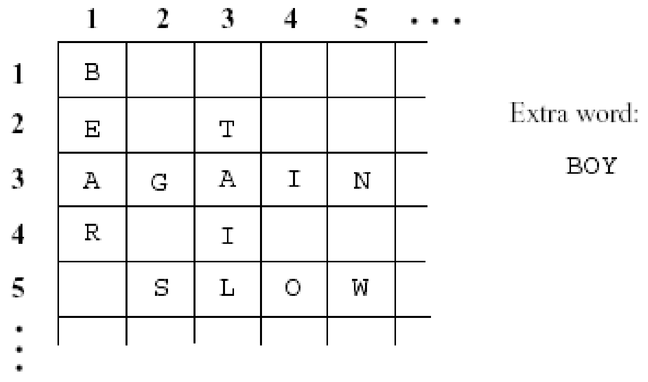


Figure 2: Example Solution

Input
The input data consist of one or more test cases each describing a puzzle trial. The first input line in each test case contains a positive integer N that represents the number of slots in the puzzle. This line is followed by N lines, each containing the row number and column number of a square where a slot begins, followed by the letter ‘A’ (if the slot is “Across”) or ‘D’ (if the slot is “Down”). The next N + 1 input lines contain candidate words that can be used in the puzzle solution.
The final test case is followed by a line containing the number zero.

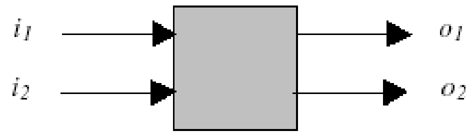
Output

For each trial, print the trial number followed by the word that is not used in the puzzle solution, using the format in the example output. Observe the following rules:
1) Print a blank line after each trial.
2) If your uncle has made a mistake and the puzzle has no solution using the given words, print the word “Impossible”. For example, if Trial 2 has no solution, you should print “Trial 2: Impossible”.
3) If the puzzle can be solved in more than one way, print each word that can be omitted from a valid solution. The words can be printed in any order but each word must be printed only once. For example, if Trial 3 has a solution that omits the word DOG and two solutions that omit the word CAT, you should print “Trial 3: DOG CAT” or “Trial 3: CAT DOG”.

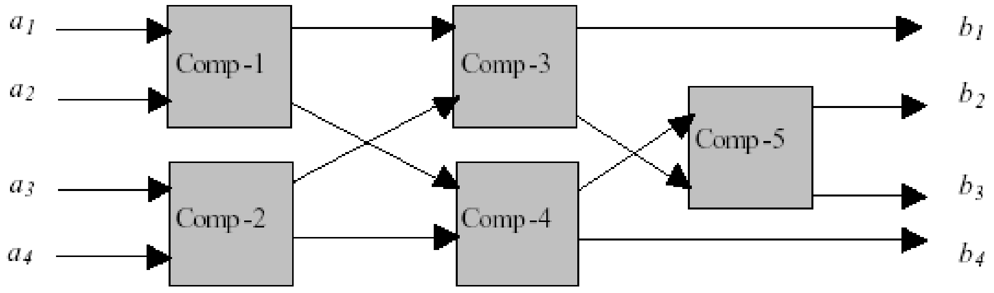
Sample input	Output for the sample input
4 1 1 D 2 3 D 3 1 A 5 2 A SLOW AGAIN BOY TAIL BEAR 0	Trial 1: BOY

Project 23 Professor Monotonic's Newworks (ACM 2001)

Professor Monotonic has been experimenting with comparison networks, each of which includes a number of twoinput, two-output comparators. A comparator, as illustrated below, will compare the values on its inputs, i1 and i2, and place them on the outputs, o1 and o2, so that o1 £ o2 regardless of the relationship between the input values.



A comparison network has n inputs a1,a2,…,an and n outputs b1,b2,…,bn. Each of the two inputs to a comparator is either connected to one of the network’s n inputs or connected to the output of another comparator. Each of the two outputs from a comparator is either connected to one of the network’s n outputs or is connected to the input of another comparator. A



graph of the interconnections of comparators must be acyclic. The illustration below shows a comparison network with four inputs, four outputs, and five comparators.

In operation, the network's inputs are applied and the comparators perform their functions. Of course a comparator cannot operate until both of its inputs are available. Assuming a comparator requires one unit of time to operate, this sample network will require three units of time to produce its outputs. Comp -1 and Comp -2 operate in parallel, as do Comp-3 and Comp -4. Comp-5 cannot operate until Comp -3 and Comp -4 have comp leted their work. Professor Monotonic needs help in determining which proposed comparison networks are also sorting networks, and how long they will take to perform their task. A sorting network is a comparison network for which the outputs are monotonically increasing regardless of the input values. The example above is a sorting network, since for all possible input values the output values will have the relation b1 £ b2 £ b3 £ b4.

Input

The professor will provide a description of each comparison network to be examined. Each description will begin with a line containing values for n (the number of inputs) and k (the number of comparators). These values satisfy 1 = n = 12 and 0 = k = 150. This is followed by zero or more non-empty lines, each containing at most 15 pairs of comparator inputs. The source of the input to each comparator is given by a pair of integers i and j. Each of these specifies either the subscript of a network input that is input to the comparator (that is, ai or aj), or the corresponding output of a preceding comparator.
The outputs of a comparator are numbered the same as its inputs (in other words, if the comparator's inputs are i and j, the corresponding outputs are also labeled i and j). The order in which these pairs appear is significant, and affects the order in which the comparators operate. If two pairs contain an integer in common, the order of the corresponding comparators in the network is determined by the order of the pairs in the list. For example, consider the input data for the example shown:

```
4 5
1 2 3 4 1 3
2 4 2 3
```

This indicates there will be four input values and five comparators in the network. The first comparator (Comp -1) will receive its input values from network inputs a1 and a2. The second comp arator (Comp -2) will receive its input values from network inputs a3 and a4. The third comparator (Comp -3) will receive its first input from the first output of Comp -1, and will receive its second input from the first output of Comp -2. Similarly, the fourth comparator (Comp-4) will receive its first input from the second output of Comp -1, and will receive its second input from the second output of Comp -2. Finally, the fifth comparator (Comp -5) will receive its first input from the first output of Comp-4, and will receive its second input from the second output of Comp -3. The outputs b1,b2,…,bn are taken from the first output of Comp -3, the first output of Comp -5, the second output of Comp -5, and the second output of Comp-4, respectively.
A pair of zeros will follow the input data for the last network.

Output

For each input case, display the case number (cases are numbered sequentially starting with 1), an indication of whether the network is a sorting network or not, and the number of time units required for the network to operate (regardless of whether it is a sorting network or not).

Sample input	Output for the sample input

4 5
1 2 3 4 1 3
2 4 2 3
8 0
3 3
1 2 2 3 1 2
0 0
	Case 1 is a sorting network and operates in 3 time units.
	Case 2 is not a sorting network and operates in 0 time units.
	Case 3 is a sorting network and operates in 3 time units.

Project 24 Balloons In A Box (ACM 2002)

You must write a program that simulates placing spherical balloons into a rectangular box. The simulation scenario is as follows. Imagine that you are given a rectangular box and a set of points. Each point represents a position where you might place a balloon. To place a balloon at a point, center it at the point and inflate the balloon until it touches a side of the box or a previously placed balloon. You may not use a point that is outside the box or inside a previously placed balloon. However, you may use the points in any order you like, and you need not use every point. Your objective is to place balloons in the box in an order that maximizes the total volume occupied by the balloons. You are required to calculate the volume within the box that is not enclosed by the balloons.

Input
The input consists of several test cases. The first line of each test case contains a single integer n that indicates the number of points in the set (1 ¡Ü n ¡Ü 6). The second line contains three integers that represent the (x, y, z) integer coordinates of a corner of the box, and the third line contains the (x, y, z) integer coordinates of the opposite corner of the box. The next n lines of the test case contain three integers each, representing the (x, y, z) coordinates of the points in the set. The box has non-zero length in each dimension and its sides are parallel to the coordinate axes.
The input is terminated by the number zero on a line by itself.

Output
For each test case print one line of output consisting of the test case number followed by the volume of the box not occupied by balloons. Round the volume to the nearest integer. Follow the format in the sample output given below.

Place a blank line after the output of each test case.

Sample input

```
0 0 0
10 10 10
3 3 3
7 7 7
0
```

Output for the sample input

```
Box 1: 774
```

Project 25 Dessert (US February Open 2002)

FJ has a new rule about the cows lining up for dinner. Not only must the N (3 <= N <= 15) cows line up for dinner in order, but they must place a napkin between each pair of cows with a "+", "-", or "." on it. In order to earn their dessert, the cow numbers and the napkins must form a numerical expression that evaluates to 0. The napkin with a "." enables the cows to build bigger numbers. Consider this equation for seven cows:

1 - 2 . 3 - 4 . 5 + 6 . 7

This means 1-23-45+67, which evaluates to 0. You job is to assist the cows in getting dessert.
(Note: "... 10 . 11 ...") will use the number 1011 in its calculation.)

Input format:
One line with a single integer, N

Sample input (file dessert.in):
```
7
```

Output format:
One line of output for each of the first 20 possible expressions -- then a line with a single integer that is the total number of possible answers. Each expression line has the general format of number, space, napkin, space, number, space, napkin, etc. etc. The output order is lexicographic, with "+" coming before "-" coming before ".". If fewer than 20 expressions can be formed, print all of the expressions.

Sample output (file dessert.out):
```
1 + 2 - 3 + 4 - 5 - 6 + 7
1 + 2 - 3 - 4 + 5 + 6 - 7
1 - 2 + 3 + 4 - 5 + 6 - 7
1 - 2 - 3 - 4 - 5 + 6 + 7
1 - 2 . 3 + 4 + 5 + 6 + 7
1 - 2 . 3 - 4 . 5 + 6 . 7
6
```

Project 26 Cryptarithm (US Fall Open 2001)

A cryptarithm is a number puzzle where many or all of the digits are missing and are replaced by asterisks. Each star represents some digit. Digits may be duplicated. The leading digit may not be a zero.

The cryptarithm below has many solutions:

```
      * * *
  x     * *
 -------------
    * * * *
  * * * *
 -------------
  * * * * *
```

Here is one sample solution:

```
   775
    33
 ------
  2325
 2325
 ------
 25575
```

Write a program that reads in a set of digits and finds all solutions to the puzzle using only those digits. You will only print the number of solutions along with any one of the solutions -- shown as the two numbers that are multiplied together and their product (you must ensure that all the other intermediate results are correct even though they are not printed).

Input format::
Line 1: One integer (1 <= N <= 10) that tells how many digits are
on the subsequent line.
Line 2: N single digits that represent the set of possible digits to
use in the answer.

Sample input (file crypt.in):
```
5
2 3 5 7 9
```

Output format:
Line 1: One integer that tells how many solutions are possible
Line 2: Three integers that represent any single solution by giving the first number, the second number, and the product.

Sample output (file crypt.out):
```
4
775 33 25575
```

Project 27 Sum25 (US Winter Open 2001)

Write a program that inputs 7 digits (0 <= each digit <= 9) and counts the number of ways one can extract a set of digits to sum to 25. The digit `0' means 10, not 0.

This input:
```
5 0 1 5 4 3 7
```

yields five different sums to 25:
```
5 0       3 7
5   1 5 4 3 7
5 0 1 5 4
  0 1   4 3 7
  0   5   3 7
```

Input format
A single line with seven single digit integers.

Sample input (file sum25.in):
```
5 0 1 5 4 3 7
```

Output format
A single line that tells the number of subsets that sum to 25.

Sample output (file sum25.out):
```
5
```

Project 28 Mother's Milk (US Winter Open 2001)

Farmer John has three milking buckets of capacity A, B, and C liters. Each of the numbers A, B, and C is an integer from 1 through 20, inclusive. Initially, buckets A and B are empty while bucket C is full of milk. Sometimes, FJ pours milk from one bucket to another until the second bucket is filled or the first bucket is empty. Once begun, a pour must be completed, of course. Being thrifty, no milk may be tossed out.

Write a program to help FJ determine how many liters of milk he can leave in bucket C when he begins with three buckets as above, pours milk among the buckets for a while, and then notes that bucket A is empty.

Input format
A single line with the three integers A, B, and C.

Sample input (file milk.in):
```
8 9 10
```

Output format
A single line that lists all the possible number of liters of milk that can be in bucket C when bucket A is empty. One space between numbers, please. No extra spaces on end of the line.

Sample output (file milk.out):
```
1 2 8 9 10
```

References

· **PROBLEMS ON ALGORITHMS, IAN PARBERRY**
· **The ALGORITHM Design Manual, STEVEN S. SKIENA**
· **Fundamentals of Data Structures in Pascal, Ellis Horowitz, Sartaj Sahni**
· **DATA STRUCTURES & PROGRAM DESIGN IN C, ROBERT KRUSE, C.L. TONDO, BRUCE LEUNG**
· **Limbjul Pascal Algorithmi funamentali, DOINA RANCEA**
· **CULUGERE DE PROBLEME PASCAL, ADRIAN ATANASIU, RODICA PINTEA**

ay_osman@yahoo.com