```
{
```

Solution for task GAME

```
-------- --- ---- ----
The following simple observation leads to a solution of the task:
Since the game board initially contains an even number of elements arranged
in a sequence, when the first player moves one end of the sequence is in odd
and the other end is in even position. Therefore the first player can always
select element either from odd or even position.
The algorithm pre-process the contents of the initial board before staring the
game to compute the values OddSum and EvenSum, the sum of the elements in odd
positions and the sum of the elements in even positions, respectively.
If OddSum>=EvenSum then the first player always selects from odd position
and force the second player to select from even position. The case
OddSum<EvenSum treated similarly.
}

Program Game;
Uses Play;
Const
  MaxN=100;                { max size of the board }
Var
  N: Word;                 { size of the board }
  Board:Array[1..MaxN] Of Word; {contents of the board }
  Sum:Word;                { sum of the elements in the initial board }
  Sel:Word;                { sum of the elements selected by the first player }
  Odds:Boolean;

Procedure ReadInput;
{ Global output variables: N, Board, Sum }
Var InFile: Text;i:Word;
Begin
  Assign(InFile,'input.txt'); Reset(InFile);
  ReadLn(InFile,N);
  Sum:=0;
  For i:=1 To N Do Begin
    ReadLn(InFile,Board[i]);
    Sum:=Sum+Board[i];
  End;
  Close(InFile);
End;

Procedure Preprocess;
{ Global input variables: N, Board }
{ Global output variable: Odds }
Var i:Word;
  OddSum,EvenSum:Word;
Begin
  OddSum:=0;
  EvenSum:=0;
  For i:=1 To N Do
    If Odd(i) Then Inc(OddSum,Board[i])
              Else Inc(EvenSum,Board[i]);
  {end for i};
  Odds:=OddSum>=EvenSum;
End {Preprocess};

Procedure Playing;
{ Global input variable: N, Board, Odds }
{ Global output variable: Sel }
Var M,i:Word;
  C1,          { move of the first player }
```

```
  C2:Char;      { move of the second player }
  Head,         { position of the left end of the board }
  Tail:1..MaxN; { position of the right end of the board }
Begin
  Sel:=0;
  Head:=1; Tail:=N;
  M:=N Div 2;          { number of moves for one player}
  For i:=1 To M Do Begin
    If Odds Then Begin {select from the odd position}
      If Odd(Head) Then Begin
          Sel:=Sel+Board[Head];
          Inc(Head);
          C1:='L';
        End Else Begin
          Sel:=Sel+Board[Tail];
          Dec(Tail);
          C1:='R';
        End;
    End Else Begin     {select from the even position}
      If Odd(Head) Then Begin
        Sel:=Sel+Board[Tail];
        Dec(Tail);
        C1:='R';
      End Else Begin
        Sel:=Sel+Board[Head];
        Inc(Head);
        C1:='L';
      End;
    End {Odd-Even};
    MyMove(C1);        { perform the move }

    YourMove(C2);      { obtain the second player's move }
    If C2='L' Then Inc(Head)
              Else Dec(Tail);
  End{For i};
End;

Procedure WriteOut;
Var OutFile: Text;
Begin
  Assign(OutFile,'output.txt'); Rewrite(OutFile);
  WriteLn(OutFile,Sel,' ',Sum-Sel);
  Close(OutFile);
End;

Begin { Program }
  ReadInput;
  Preprocess;
  StartGame;
  Playing;
  WriteOut;
End.

{Scientific Committee IOI'96}
```

{

Solution of task JOBS

-------- -- ---- ----

Consider the following data structures.

```
Const
  MaxM=30;                  (* max number of machines *)
Type
  Operation='A'..'B';
  ProcTime=Array[Operation,1..MaxM] Of Word;
Var
  N:Longint;                  (* number of jobs *)
  M:Array[Operation] Of Word;(* M[op] is the number of machines of type op *)
  PTime: ProcTime;            (* PTime[op,m] is the processing time for machine
                                m of type op *)
```

Subtask A

It is obvious that an optimal schedule can be modified in such a way that each machine starts processing at time 0 and never idle until performing the operation on all jobs that are scheduled for that machine.
The maximal number of jobs that can be processed within time t by machine m of type Op is t div PTime[Op,m]. Therefore the minimal amount of time that is needed to perform operation Op on all N jobs is the least number t such that the sum

```
  (t Div PTime[Op,i]) (for i:=1 to M[Op])
```

is greater or equal to N.
The following algorithm computes the total processing time for operation Op in variable t:

```
    t:=0;
    Repeat
      Inc(t);
      Processed:=0;
      For i:=1 To M[Op] Do
        Processed:=Processed+(t Div PTime[Op,i]);
    Until Processed>=N;
```

Subtask B

It is obvious that the schedule for type A machines is the same as in case of subtask A.
Let TAB be the minimal amount of time that is necessary to perform both operations on all N jobs.
We may assume that each machine of type B finishes processing exactly at time TAB and never idle between executing two consecutive jobs. If this is not the case originally then we can modify the optimal schedule accordingly, since if a job is available at a time in the intermediate container then this job will be available later too.

Let d be the time when processing of the first job by a machine of type B starts according to the optimal schedule. Denote by TB the minimal amount of time that is necessary to perform single operation B on all N jobs.
By the same argument as in case of subtask A, we have that TAB=d+TB.
We already have an algorithm to compute the value TB, hence it remains to develop an algorithm which computes the delay time d.

Let Finish(Op,t) be the number of jobs that are finished at time t according to an optimal schedule for single operation Op.

The delay time d is the least number that satisfies the following condition:
  for every t, 0 <= t < TB at least Finish('B',TB-t) number of jobs are

```
  available in the intermediate container at time d+t.

We can check for a given d whether it satisfies the above condition. Therefore
the value d could be computed by starting d=1 and incrementing d while the
condition does not hold.
We have faster computation by using incremental method. This method works
as follows. The starting value for d is 1. Suppose that the above condition
holds for a given d and t values 0,...,ts. If the condition does not hold
for t value ts+1 then increase d until the condition holds.
This method is implemented in the program by the procedure Adjust.

}
Program Jobs;
Const
  MaxM=30;                    { max number of machines }
Type
  Operation='A'..'B';
  ProcTime=Array[Operation,1..MaxM] Of Word;
Var
  N:Longint;                  { number of jobs }
  M:Array[Operation] Of Word; { M[op] is the number of machines of type op }
  PTime: ProcTime;            { PTime[op,m] is the processing time for machine
                                m of type op }
  TA,          { the time needed to perform single operation A on all N jobs }
  TB: Longint;{ the time needed to perform single operation B on all N jobs }
  d :Longint;

Procedure ReadInput;
{ Global output variables: N, M, PTime }
  Var InFile: Text; i: Word;
  Begin
    Assign(InFile, 'input.txt'); Reset(InFile);
    ReadLn(InFile,N);
    ReadLn(InFile,M['A']);
    For i:=1 To M['A'] Do
      Read(InFile, PTime['A',i]);
    ReadLn(InFile);
    ReadLn(InFile,M['B']);
    For i:=1 To M['B'] Do
      Read(InFile, PTime['B',i]);

    Close(InFile);
  End {ReadInput};

Function Compute_Time(Op:Operation):Longint;
{Computes the minimal time that is needed to perform operation Op on N jobs}
{ Global input variables: M, PTime }
  Var t,Processed:Longint;
    i:Word;
  Begin
    t:=0;
    Repeat
      Inc(t);
      Processed:=0;
      For i:=1 To M[Op] Do
        Processed:=Processed+(t Div PTime[Op,i]);
    Until Processed>=N;
    Compute_Time:=t;
  End;{Compute_Time}

Function Finish(Op:Operation; t: Longint): Longint;
{ Finish(Op,t) is the number of jobs that are finished at time t
  according to the optimal schedule for single operation Op for N jobs. }
```

```
{ Global input variables: N, M, PTime }
  Var Res,UpTo: Longint;
    i: Word;
  Begin
    Res:=0;
    For i:=1 To M[Op] Do
      If (t Mod PTime[Op,i])=0 Then Inc(Res);
    { If the number of jobs that can be completed up to time t
      is more then N then decrease Res to the proper value. }
    UpTo:=0;
    For i:=1 To M[Op] Do UpTo:= UpTo+ (t-1) Div PTime[Op,i];
    If Upto >= N Then
      Res:= 0
    Else If Upto+Res>N Then
      Res:= N-UpTo;
    Finish:=Res;
  End {Finish};

Procedure Adjust;
{ Computes the delay time d when the first type B machine starts to work }
{ Global input variables: TA, TB }
{ Global output variables: d }
  Var Inter:Word;{ number of jobs in the intermediate container }
    t: Longint;
    JB:Word;
  Begin
    d:=1; t:=0; Inter:=0;
    While d+t<TA Do Begin
      Inter:=Inter+Finish('A',d+t);
      JB:=Finish('B',TB-t);    { # jobs starting at time d+t }
      While Inter<JB Do Begin  { while not enough jobs available }
        Inc(d);
        Inter:=Inter+Finish('A',d+t);
      End;
      Inter:=Inter-JB;
      Inc(t);
    End;
  End;{Adjust}

Procedure WriteOut(AnswerA,AnswerB:Longint);
  Var OutFile: Text;
  Begin
    Assign(OutFile, 'output.txt'); Rewrite(OutFile);
    WriteLn(OutFile, AnswerA);
    WriteLn(OutFile, AnswerB);
    Close(OutFile);
  End;{WriteOut}

Begin {Main}
  ReadInput;
  TA:= Compute_Time('A');
  TB:= Compute_Time('B');
  Adjust;
  WriteOut(TA, d+TB);
End.

{Scientific Committee for IOI'96}
```

{

Solution for task NET

-------- --- ---- ---

The network of schools can be represented by a directed graph whose vertices are the schools and (A, B) is an edge in the graph iff school B is in the distribution list of school A. Let us first reformulate the task using graph terminology.
We use the notation p->q if there is a (directed) path from p to q in a graph.
A set of vertices D of a graph G is called dominator set of G if for each vertex q there is a vertex p in D such that p->q.
Subtask A is to find a dominator set of G with minimal number of elements.
A set of vertices CD of G is called codominator set of G if for each vertex p there is a vertex q in CD such that p->q. A graph G is called strongly connected, if for all vertices p and q there is a path p->q and a path q->p in G.
Solution of subtask B is the minimal number of new edges that are necessary to make G strongly connected.
Let us denote the number of elements of a set S by |S|.
Let D be a minimal dominator set and CD be a minimal codominator set of G.

We shall prove that solution of subtask B is 0 if G is strongly connected, and Max(|D|, |CD|) otherwise.
The proof follows from the statements S1 and S2.
We can assume without loss of generality that |D|<=|CD|.

S1. If D is a one element set containing p and CD contains the elements q1,...,qk then introducing the new edges (q1, p), ... , (qk, p) makes G strongly connected.
Proof: Let u,v be arbitrary vertices of G. Then there is an element qi in CD such that u->qi, therefore u->qi->p->v is a path from u to v.

S2. If |D|>1 then there are vertices p in D and q in CD such that introducing the new edge (q, p) in G makes D-[p] a new dominator set and CD-[q] a new codominator set of G.
Proof: Since |D|>1 there are different vertices p1 and p2 in D, and there are different vertices q1 and q2 in CD such that p1->q1 and p2->q2. Then the new edge (p,q) will be (q1,p2). Indeed, any vertex u that was reachable from p2 by the path p2->u will be reachable from p1 by p1->q1->p2->u and for any path v->q1 there will be a path v->q1->p2->q2 in the new graph.

It is obvious that a codominator set of a graph G is a dominator set of the transposed graph GT and conversely. Therefore we can compute the minimal codominator set of G by transposing G and then computing the minimal dominator set of the transposed graph.

The strategy for computing a minimal dominator set is the following.
(We use Pascal terminology for set operations)

```
  Dominated:=[];
  D:=[];
  While there is a p not in Dominated Do Begin
    Search(p);(* put all vertices in set Reachable that are reachable from p*)
    Dominated:=Dominated+Reachable;
    D:=D-Reachable; (* exclude all elements of D that are in Reachable *)
    Include p in D;
  End;
```

Evidently the set D that is produced by this algorithm is a dominator set.
Assume that D contains the vertices p1,...,pk and D is not minimal, i.e. there is a minimal dominator set Q that contains vertices q1,...ql, and l<k.
Since D is a dominator set and Q is a minimal dominator set it follows that for each qi in Q there is a unique pi in D such that qi is reachable from pi

by a path pi->qi. But every vertex is reachable from Q, therefore pk is also reachable from some vertex in Q, say qi->pk. We obtained that there is a path pi->qi->pk from pi to pk. The algorithm has executed both Search(pi) and Search(pk). Either Search(pi) or Search(pk) was executed first, the vertex pk was excluded from D because pk is reachable from pi. This contradicts to the assumption that D is not minimal.

The algorithm above can be modified to avoid set operations union (+) and difference (-). Indeed, when Search(p) is executed, we can include p in the set Dominated and exclude p from D.

```
}
Program Net;
Const
  MaxN=200;            { max number of schools }
Type
  GraphType=Array[1..MaxN,0..MaxN] Of 0..MaxN;
  VertexSet=Set Of 1..MaxN;
Var
  OutFile: Text;
  N :Word;             { the number of schools }
  G: GraphType;        { representation of the network with graph G; }
                       { G[p,0] is the number of edges outgoing from p }
                       { the outgoing edges from p: (p, G[p,i]) 1<=i<=G[p,0]) }
  Domin,               { dominator set }
  CoDomin: VertexSet;  { codominator set }
  NoDomins,            { number of dominator elements   }
  NoCoDomins: 0..MaxN;{ number of codominator elements }
  AnswerB: 0..MaxN;    { solution of subtask B }
  p: 0..MaxN;

Procedure ReadInput;
{ Global output variables: N, G }
  Var InFile: Text;
  i,p: Word;
  Begin
    Assign(InFile, 'input.txt'); Reset(InFile);
    ReadLn(InFile,N);

    For i:=1 To N Do
      G[i,0]:=0;
    For i:=1 To N Do Begin
      Read(InFile, p);
      While p<>0 Do Begin
        Inc(G[i,0]);
        G[i,G[i,0]]:=p;
        Read(InFile, p);
      End;
      ReadLn(InFile);
    End;

    Close(InFile);
  End { ReadInput };

Procedure ComputeDomin(Const G: GraphType; Var D: VertexSet);
{ Computes a minimal dominator set D of graph G }
{ Global input variables: N }
  Var
    Dominated, Reachable: Set of 1..MaxN;
    p: 1..MaxN;

  Procedure Search(p:Word);
    Var i: Word;
  Begin
```

```
    Exclude(D, p);
    Include(Dominated, p);
    For i:= 1 To G[p,0] Do
      If Not (G[p,i] in Reachable) Then Begin
        Include(Reachable,G[p,i]);
        Search(G[p,i]);
      End;
  End { Search };

  Begin { ComputeDomin }
    D:=[];
    Dominated:=[];
    For p:=1 To N Do
      If Not (p In Dominated) Then Begin
        Reachable:=[p];
        Search(p);
        Include(D, p);
      End;
  End { ComputeDomin };

Procedure ComputeCoDomin(Const G: GraphType; Var CD: VertexSet);
{ Computes a minimal codominator set D of graph G }
{ Global input variables: N }
  Var
    GT: GraphType;           { transposed graph of G }
    p,q: 1..MaxN; i:Word;

  Begin { ComputeCoDomin }
    For p:=1 To N Do
      GT[p,0]:=0;
    For p:=1 To N Do         { compute the transpose of the graph G in GT }
      For i:=1 To G[p,0] Do Begin
        q:=G[p,i];
        Inc(GT[q,0]); GT[q,GT[q,0]]:=p;
      End;
  ComputeDomin(GT, CD)        { computes CD, the dominator set of GT }
  End;{ ComputeCoDomin }

Begin { Program }
  ReadInput;
  ComputeDomin(G,Domin);
  ComputeCoDomin(G,CoDomin);

  NoDomins:=0;
  For p:=1 To N Do     { count the number of elements in the set Domin }
    If p In Domin Then Inc(NoDomins);

  NoCoDomins:=0;
  For p:=1 To N Do     { count the number of elements in the set CoDomin }
    If p In CoDomin Then Inc(NoCoDomins);

  If (Domin=[1]) And (CoDomin=[1])  { strongly connected }
    Then AnswerB:=0
    Else If NoDomins > NoCoDomins
      Then AnswerB:=NoDomins
      Else AnswerB:=NoCoDomins;

  Assign(OutFile, 'output.txt'); Rewrite(OutFile);
  WriteLn(OutFile, NoDomins);
  Writeln(OutFile, AnswerB);
  Close(OutFile);
End.
```

{

Solution for task SORT3

-------- --- ---- -----

The basic idea behind our solution is the following.
Compute first the number of appearances Na[x] for elements x=1, 2, 3 in the input sequence. Then the sorted sequence consists of Na[1] number of 1's followed by Na[2] number of 2's and then Na[3] number of 3's.
We say that an element x is in the place of y's if the current position of x equals the position of an element y in the sorted sequence. We use the abbreviation x:y to denote an element x in place of y's.
Next compute NEP[x,y], the number of elements x's in the place of y's for all x and y. Consider the following algorithm written in pseudo-code.

```
  NoCh:=0;
  While Not Sorted(S) Do Begin
    If there are x and y in each other's place Then Begin
      Inc(NoCh);
      Exchange x and y;
      Update NEP[x,y] and NEP[y,x];
    End Else Begin
      If (NEP[1,2]>0) And (NEP[3,1]>0) Then Begin
        Exchange a pair of elements 3:1 and 1:2 ;
        Update NEP[1,2] And NEP[3,1];
        Inc(NoCh);
      End;
      If (NEP[2,1]>0) And (NEP[1,3]>0) Then Begin
        Exchange a pair of elements 2:1 and 1:3 ;
        Update NEP[2,1] And NEP[1,3];
        Inc(NoCh);
      End;
    End;
  End;
```

First we show that the number of exchange operations performed by the algorithm can be given by the expression

```
     Ch(S)=Min(NEP[1,2], NEP[2,1])+
           Min(NEP[1,3], NEP[3,1])+
           Min(NEP[2,3], NEP[3,2])+
           2*Abs(NEP[1,2]-NEP[2,1])
```

After performing Min(NEP[x,y], NEP[y,x]) exchange operations for all x<>y the resulting sequence contains NEP[1,2]-NEP[2,1] number of elements 1:2, 2:3, 3:1 if NEP[1,2]>NEP[2,1] and 1:3, 2:1, 3:2 if NEP[1,2]<NEP[2,1]. In the first case the algorithm makes exchange 2:1 and 1:3 which results an element 2 in place of 3, therefore in the next iteration an exchange of 2:3 and 3:2 will be performed. The second case is similar. We conclude that the expression Ch(S) is correct for the number of exchange operations performed by the algorithm.

Let us denote by OCh(S) the minimal number of exchange operations needed to make the sequence S sorted. We shall prove that Ch(S)=OCh(S). The proof is by induction on the value OCh(S).
If OCh(S)=0 or 1 then the statement obviously holds.
Assume that Och(S)=Ch(S) for all S if OCh(S)<k, for some k>1.
Let S be a sequence and OCh(S)=k.

Consider an optimal sequence of exchange operations that makes S sorted.
Assume that the first exchange operation exchanges elements x1:y1 and x2:y2 (or x1:y2 and x1:y1) and denote by S' the resulting sequence.
We distinguish the following two cases:

```
  C1: x1=y2 and x2=y1 or
      NEP[1,2]>NEP[2,1] and x1=1, y1=2, x2=3, y2=1 or
                            x1=1, y1=2, x2=2, y2=3 or
                            x1=3, y1=1, x2=2, y2=3 or
      NEP[1,2]>NEP[2,1] and x1=2, y1=1, x2=3, y2=2 or
                            x1=2, y1=1, x2=1, y2=3 or
                            x1=3, y1=2, x2=1, y2=3 or
  C2: all other combinations for x1, y1, x2, y2.

We can verify by a routine calculation that Ch(S')=Ch(S)-1 in case C1 and
Ch(S')>=Ch(S) in case C2. In case C1 we obtain by the inductive hypotheses
that the algorithm performs Ch(S)=Ch(S')+1=OCh(S')+1=OCh(S) exchange
operations.
Case C2 contradicts to the optimality condition, therefore an optimal
sequence of exchange operations can only start with an exchange specified in
case C1.

In order to develop efficient algorithm that constructs a sequence of exchange
operations, we introduce the array First, that First[x,y] always contains
the first position of x in place of y's. First[x,y] is computed by the
preprocess procedure and is updated after each exchange operation.

}
Program Sort3;
Const
  MaxN =1000;                  { max number of elements to sort }
Type
  ElemType = 1..3;
  ArrayType= Array[1..MaxN] Of ElemType;
  Matrix   = Array[ElemType, ElemType] of Integer;
Var
  N : Word;                    { number of elements to sort             }
  S : ArrayType;               { array of elements to sort              }
  Na: Array[ElemType] Of Word;{ Na[x] is the number of x's in the input }
  NEP : Matrix;                { NEP[x,y] is the number of x's          }
                               { in place of y's                        }
  First: Matrix;               { First[x,y] is the first position of x  }
                               { in place of y's                        }
  NoCh: Word;                  { number of exchange operations          }
  OutFile: Text;

Procedure ReadInput;
{ Global output variables: N, S, Na }
  Var
    InFile: Text;
    i,j: Word;
  Begin
    Assign(InFile, 'input.txt');
    Reset(Infile);

    ReadLn(InFile, N);
    For i:=1 To 3 Do Na[i]:=0;
    For i:=1 To N Do Begin
      ReadLn(InFile,S[i]);
      Inc(Na[S[i]]);
    End;
    Close(InFile);
  End { ReadInput };

Function Min(X,Y: Word):Word;
  Begin
    If X<Y Then Min:=X
           ELse Min:=Y
```

```
  End { Min };

Procedure Preprocess;
{ Global input variables: N, S, Na }
{ Global output variables: NEP, First }
  Var i,j,M:Word;
  Begin
    For i:=1 To 3 Do Begin
      For j:=1 To 3 Do Begin
        NEP[i,j]:=0; First[i,j]:=0
      End { For j };
    End { For i };

    For i:=1 To N Do Begin
      If i<=Na[1] Then Begin                     { S[i] is in place of 1's    }
        If NEP[S[i],1]=0 Then First[S[i],1]:=i; { first S[i] in place of 1's }
        Inc(NEP[S[i],1]);
      End Else If i<=Na[1]+Na[2] Then Begin     { S[i] is in place of 2's    }
        If NEP[S[i],2]=0 Then First[S[i],2]:=i; { first S[i] in place of 2's }
        Inc(NEP[S[i],2]);
      End Else Begin                             { S[i] is in place of 3's    }
        If NEP[S[i],3]=0 Then First[S[i],3]:=i; { first S[i] in place of 3's }
        Inc(NEP[S[i],3])
      End;
    End { For i };
    NoCh:= Min(NEP[1,2], NEP[2,1])+              { subtask A }
           Min(NEP[1,3], NEP[3,1])+
           Min(NEP[2,3], NEP[3,2])+
           2*Abs(NEP[1,2]-NEP[2,1]);
  End;{ Preprocess }

  Procedure Next(i1,i2:Byte);
  { Global input-output variables: First, NEP }
    Begin
      Dec(NEP[i1,i2]);
      If NEP[i1,i2]>0 Then Begin
        Repeat
          Inc(First[i1,i2]);
        Until S[First[i1,i2]]=i1;
      End;
    End { Next };

Procedure Pairs;
  Var M,i,x,y :Word;
  Begin
    For x:=1 To 3 Do
      For y:=x+1 To 3 Do Begin
        M:=Min(NEP[x,y], NEP[y,x]);
        For i:=1 To M Do Begin
          WriteLn(OutFile, First[x,y],' ',First[y,x]);
          Next(x,y); Next(y,x);
        End;
      End;
   End { Pairs };

Procedure Triples;
  Var M,i: Word;
  Begin
    If NEP[1,2] > 0 Then Begin
      M:=NEP[1,2];
      For i:=1 To M Do Begin
        WriteLn(OutFile, First[3,1],' ',First[1,2]);
        WriteLn(OutFile, First[1,2],' ',First[2,3]);
```

```
        Next(3,1); Next(1,2); Next(2,3);
      End;
    End Else Begin
      M:=NEP[2,1];
      For i:=1 To M Do Begin
        WriteLn(OutFile, First[2,1],' ',First[1,3]);
        WriteLn(OutFile, First[1,3],' ',First[3,2]);
        Next(2,1); Next(3,2); Next(1,3);
      End;
    End;
  End;

Begin { Program }
  ReadInput;

  Preprocess;

  Assign(OutFile, 'output.txt');
  Rewrite(OutFile);
  WriteLn(OutFile,NoCh);

  Pairs;
  Triples;

  Close(OutFile);
End.

{ Scientific Committee IOI'96 }
```

{

Solution of task PREFIX

```
-------- -- ---- ------
Let S be a sequence of letters and let P be a set of primitives.
Denote by Suff(S,P) the set of sequences v such that the following
two conditions hold:
  (1) v is a prefix of a primitive in P
  (2) S=uv for some u.

(For two sequences of letters u and v we denote the concatenation of u and v
by uv.)
It is obvious that S can be composed from primitives in P iff the empty
sequence is in Suff(S,P). Moreover, S has an extension u on the right that
makes Su a composition of primitives in P iff Suff(S,P) is non-empty.
Therefore the following algorithm gives a solution to the task if DataFile
contains the sequence to be examined.

  Res:=0;
  S:=empty; NoS:=1;
  ReadLn(DataFile,X);
  Slength:=1;
  While (X<>'.') And (NoS>0) Do Begin
    Append X to the end of S;
    Q:=Suff(S,P);
    NoS:=number of elements of Q;
    If the empty sequence is element of Q Then Res:=Slength;
    ReadLn(DataFile,X);
    Inc(Slength);
  End;
  WriteOut(Res);

One of the obvious problem with this algorithm is that the datafile is too
large to read into the memory (unless you know how to use the machine's
extra memory).
Fortunately, it is not necessary to read the whole sequence into memory.
Let us observe that Suff(Sx,P) for a sequence S and a letter x can be
computed from the set Suff(S,P). Indeed, the following algorithm satisfies
the requirement that if Q=Suff(S,P) holds before executing Next(Q,X)
then after the execution Q=Suff(SX,P) will hold.

Procedure Next(Q,X);
  Begin
    Q1:=empty;
    Forall u in Q Do Begin
      If ux is a prefix of some primitives in P Then
        Begin
          include ux in Q1;
          If ux is equal to a primitive in P
            Then include the empty sequence in Q1
        End;
    End;
    Q:=Q1;
  End;

In order to refine the algorithm Next we have to answer for the questions:
  - Is a sequence ux a prefix of some primitive in P?
  - Is a sequence u equal to a primitive in P?

Consider the following data structure for the set of primitives P.
Const
  MaxN=100;                    (* maximum possible number of primitives *)
```

```
  MaxL=20;                       (* maximum possible length of primitives *)
Var
  P:Array[1..MaxN,1..MaxL] Of Char;                  (* array of primitives *)
  L:Array[1..MaxN] Of Word;                    (* length of the primitives *)

Let us represent a sequence u which is a prefix of a primitive in P by the
pair (i,j), such that the prefix of P[i] consisting of the first j letters
of the primitive P[i] equals u, and i is the least such index for u.
Note that the empty sequence is represented by the pair (1,0).

We can pre-process the set of primitives to build a transition table T:

  T[i,j,x] is 0 if there is no primitive in P with prefix P[i][1..j]x,
  otherwise the least index k such that P[i][1..j]x is a prefix of P[k].
  (P[i][1..j] denotes the sequence of letters consisting of the first j
  elements of the primitive P[i].)

In other words, if a sequence u is represented by the pair (i,j) then the
sequence ux is a prefix of a primitive in P iff T[i,j,x]>0, and in this case
ux is a prefix of P[T[i,j,x]] and is represented by the pair (T[i,j,x],j+1).
The procedure BuildTable computes the transition table T and builds the array
Full as well. Full[i,j] is true iff the sequence represented by (i,j) is equal
to a primitive in P.
We can easily implement the algorithm Next(Q,X) using the arrays T and Full.

}
Program Prefix;
Const
  MaxN=100;                        { maximum possible number of primitives }
  MaxL=20;                         { maximum possible length of primitives }
Var
  DataFile:Text;                    { file for the sequence to be examined }
  P:Array[1..MaxN,1..MaxL] Of Char;                  { array of primitives }
  L:Array[1..MaxN] Of Word;                    { length of the primitives }
  T:Array[1..MaxN,0..MaxL,'A'..'Z'] Of Byte;            { transition table }
  N,                                                { number of primitives }
  ML:Word;                          { max of the length of the primitives }
  Res:Longint;                             { length of the longest prefix }
  Full:Array[1..MaxN,1..MaxL] Of Boolean;

Type
  State=Array[1..MaxL+1] Of Record
                              i,j:Byte;
                            End;
Procedure Init;
Var M,i,j:Word;
  InFile:Text;
Begin
  Assign(InFile,'input.txt'); Reset(InFile);
  ReadLn(InFile,N);
  ML:=0;
  For i:=1 To N Do Begin
    ReadLn(InFile,L[i]);
    If L[i]>ML Then ML:=L[i];
    For j:=1 To L[i] Do Read(InFile,P[i][j]);
    ReadLn(InFile);
  End;
  Close(InFile);
  Assign(DataFile,'data.txt'); Reset(DataFile);
End;

Procedure BuildTable;
{Global input variables: N, ML, P }
```

```
{Global output variables: T, Full }
Var
  i,i1,j,k:Word;
  X:Char;
Begin
  For i:=1 To N Do  {initialize the array Full}
    For j:=1 To ML Do Full[i,j]:=False;
  For i:=1 To N Do          { compute T[i,0,x] }
    For X:='A' To 'Z' Do Begin
      k:=1;
      While (k<=N) And (P[k][1]<>X) Do Inc(k);
      If (k<=N) Then Begin
        T[i,0,X]:=k;
        Full[k,1]:=Full[k,1] Or (L[i]=1) And (P[i][1]=X);
      End Else
        T[i,0,X]:=0;
    End;
  For j:=1 To ML Do Begin
    For i:=1 To N Do Begin
      For X:='A' To 'Z' Do Begin { compute T[i,j,X] }
        If j>L[i] Then Begin
          T[i,j,X]:=0;
        End Else Begin
          i1:=T[i,j-1,P[i][j]];
          k:=1;
          While (k<=N) And
                Not ((j+1<=L[k]) And (P[k][j+1]=X) And (i1=T[k,j-1,P[k][j]]))
            Do Inc(k);
          If (k<=N) Then Begin
            T[i,j,X]:=k;
            Full[k,j+1]:=Full[k,j+1] Or (L[i]=j+1);
          End Else
            T[i,j,X]:=0;
        End;
      End {for 'A'..'Z'};
    End {for i};
  End {for j};
End {BuildTable};

Procedure Next(Var NoS:Word; Var Q:State; X:Char; Var Complete:Boolean);
{Input: NoS is the number of prefixes in Suff(S,P),
        (Q[1].i,Q[1].j),...,(Q[NoS].i,Q[NoS].j) are the representatives of
        the prefixes in Suff(S,P),
        X is the actual element of the sequence to be examined.
 Output:NoS is the number of prefixes in Suff(SX,P),
        (Q[1].i,Q[1].j),...,(Q[NoS].i,Q[NoS].j) are the representatives of
        the prefixes in Suff(SX,P),
        Complete is True iff the empty sequence is in Q.
}
  Var i,j,ii,newi,newj:word;
  Begin
    ii:=0; Complete:= False;
    For i:=1 To NoS Do Begin                           { compute next state }
      newi:=T[Q[i].i,Q[i].j,X]; newj:=Q[i].j+1;
      If newi>0 Then Begin
        Inc(ii);
        Q[ii].i:=newi; Q[ii].j:=newj;
        Complete:=Complete Or Full[newi,newj];
      End;
    End;
    If Complete Then Begin
      Inc(ii); Q[ii].i:=1;Q[ii].j:=0;             {include the empty string}
    End;
```

```
    NoS:=ii;
  End {Next};

Procedure Process;
{Global input variables: DataFile }
{Global output variables: Res }
Var
  X:Char;                                { the actual element of the sequence }
  Q:State;                           { set of prefixes of primitives that are
                                          suffixes of the sequence red so far }
  NoS:Word;                                    { number of the elements of Q }
  Slength:Longint;                        { length of the sequence red so far }
  Complete:Boolean;
Begin
  NoS:=1; Q[1].i:=1; Q[1].j:=0;                                { initialize Q }
  Res:=0; Slength:=1;
  ReadLn(DataFile,X);
  While (X<>'.') And (NoS>0) Do Begin
    Next(NoS,Q,X,Complete);
    If Complete Then Res:=Slength;
    ReadLn(DataFile,X);
    Inc(Slength);
  End {While};
  Close(DataFile);
End {Process};

Procedure WriteOut(Res:Longint);
  Var OutFile:Text;
Begin
  Assign(OutFile,'output.txt'); Rewrite(OutFile);
  WriteLn(OutFile,Res);
  Close(OutFile);
End;

Begin
  Init;
  BuildTable;
  Process;
  WriteOut(Res);
End.

{ Scientific Committee IOI'96 }
```

{

Solution of task MAGIC

-------- -- ---- -----

The following algorithm written in pseudocode generates all configurations that can be obtained by applying a sequence of basic transformations to the initial configuration.

```
  Make the set Generated empty;
  Make the set Disp empty;
  Include the configuration Ini in Disp;
  While Disp is not empty Do Begin
    take an element P out of Disp;
    for all basic transformation C Do Begin
      let Q be the configuration obtained by applying C to P;
      If Q is not in the set Generated Then Begin
        include Q in the set Generated;
        include Q in the set Disp;
      End
    End
  End;
```

We can stop searching if the configuration Q is the target.
Let us first investigate the implementation of the operations on the set Generated.
The number of all configurations is 8!=40320. It is too large to store the configurations in an array. We can overcome this problem by using an bijective function Rank that maps a configuration into a number in the range 0..8!-1. We can obtain such function by defining Rank(Q) as the number of configurations that precedes Q according to the lexicographic ordering of the permutations of the numbers 1..8.

Let us observe that each basic transformation C has a unique inverse in the sense that for any configuration Q if C transforms Q to P then its inverse transforms P to Q and conversely, if the inverse of C transforms P to Q then C transforms Q to P. The inverses of the basic transformations are:

  A: A itself,
  B: single left circular shifting,
  C: single anti-clockwise rotation of the middle four squares.

If the configuration Q is obtained in the algorithm by applying the basic transformation C to P then there is a sequence of basic transformations that transforms the initial configuration to Q whose last element is C. If we know Q and C then we can compute P by applying the inverse of C to Q.
Consider the array Last: Array[0..8!-1] Of Char. We use the array Last for two purposes. Last[Rank(Q)]=' ' iff the configuration Q has not been generated. If Q is obtained during the generation by applying C to a configuration P then Last[Rank(Q)] is set to C. Following the link provided by the inverse transformation we can compose the whole sequence of basic transformations for the target configuration T.

```
    S:='';             (* string S is set to empty      *)
    While T <> Ini Do Begin
      X:=Last[Rank(T)];
      S:=X+S;          (* append X to the left end of S *)
      Apply_1(T,X,P);  (* Apply the inverse of X to T   *)
      T:=P;            (* link to backward              *)
    End (* While *);
```

We implement the dispenser Disp as a queue, i.e. by first-in first-out policy; items come out in the order of their insertion.

A simple experiment will show that the maximal length of the sequences produced by the algorithm is 22.

The queue implementation of the dispenser provides optimal solution in the sense that for each configuration T the algorithm produces the shortest sequence of basic transformations.
We prove by induction on the length of the optimal solution.
Let us denote by l(T) the length of the sequence generated by the algorithm for configuration T.

Suppose that during the execution of the algorithm the queue contains the configurations T1,...,Tk where T1 is the head. Then the following two conditions hold:

```
  1) l(T1)<=...<=l(Tk)
  2) l(Tk)<=l(T1)+1
```

The proof is by induction on the number of queue operations.
Initially the statement holds because only the initial configuration is in the queue and l(Ini)=0. If the head T1 is dequeued, then the new head is T2. But then we have l(Tk)<=l(T1)+1<=l(T2)+1, and the remaining inequalities are unaffected. Consider the case when T1 is dequeued and the new configuration Q which is obtained by applying C to T1 is enqueued.
Then l(Q)=l(T1)+1, therefore the inequalities

```
  l(Tk)<=l(T1)+1=l(Q) and l(Q)=l(T1)+1<=l(T2)+1
```

hold by 1) and 2).

It follows from the previous statement that if the configurations inserted into the queue over the course of the algorithm in the order T1,...,Tn then l(T1)<=...<=l(Tn).

Let T(k) denote the set of all configurations Q that the minimal length of the sequence of basic transformations that transform Ini to Q is k. The proof of the optimality of the algorithm follows by induction on k.

The elements of T(1) are those configurations that can be obtained by applying the basic transformations to Ini. The statement obviously holds for k=1. Assume that the statement holds for all l<k. Let Q be a configuration in T(k). Then there is a configuration P in T(k-1) and a basic transformation C such that Q can be obtained by applying C to P. By the inductive hypotheses, l(P)=k-1. P was inserted into the queue when it was generated by the algorithm. Consider the time when the algorithm dequeues P from the queue and checks whether Q is already generated. If Q is new then the algorithm generates Q and hence l(Q)=l(P)+1=k. If Q was already generated then l(Q)<=l(P)=k-1 by the monotonicity property, which is a contradiction.

```
}

Program Magic;
Const
  Size=8;     { Size of the sheet }
  M   =40320; { =Size! }
Type
  Trans =Array[1..Size] Of 1..Size;
  Config=Array[1..Size] Of 1..Size;
Const
  BT  :Array['A'..'C'] Of Trans=((8,7,6,5,4,3,2,1), { basic transformations }
                                 (4,1,2,3,6,7,8,5),
                                 (1,7,2,4,5,3,6,8));
  BT_1:Array['A'..'C'] Of Trans=((8,7,6,5,4,3,2,1), { inverses of the basic }
                                 (2,3,4,1,8,5,6,7), { transformations       }
                                 (1,3,6,4,5,7,2,8));
  Ini :Config=(1,2,3,4,5,6,7,8);                { the initial configuration }
Var
  T    :Config;                                  { the target configuration }
```

```
  Answer:String;           { the solution sequence of basic transformations }
  Fact :Array[0..Size] Of Longint;             { array of factorial values }
  Last :Array[0..M] Of Char;
               { Last[Rank(T)] is the last character of a sequence of basic }
          { transformations that transforms the initial configuration to T. }
                      { If Last[Rank(T)]=' ' then T has not been generated. }

Procedure ReadInput;
{ Global output variable: T }
  Var InFile:Text;
    i:Word;
  Begin
    Assign(InFile,'input.txt'); Reset(Infile);
    For i:=1 To Size Do Read(Infile,T[i]);
    Close(Infile);
  End { ReadInput };

Procedure ComputeFact;
{ Computes the factorial values }
  Var i:Word;
  Begin
    Fact[1]:=1;Fact[0]:=1;
    For i:=2 To Size Do
      Fact[i]:=i*Fact[i-1];
  End;

Function Rank(Const P:Config): Word;
{ Rank(P) is the number of permutations that precedes P }
{ according to the lexicographic ordering.              }
  { Global input variables: Size, Fact }
  Var Res,l,i,j:Word;
  Begin
    Res:=0;
    For i:=1 To Size Do Begin
      l:=0;              { l is the number of elements of P in positions }
                         { 1..i-1 that are less than P[j]                }
      For j:=1 To i-1 Do
        If P[j]<P[i] Then Inc(l);
      { Keeping fixed the first i-1 elements of P there can be (P[i]-1-l) }
      { numbers that are less than P[i] in position i in permutations.   }
      { The number of permutations Q such that the first i-1 elements    }
      { are the same as in P but Q precedes P in the lexicographic       }
      { ordering is (P[i]-1-l)*Fact[Size-i].                             }
      Res:=Res+(P[i]-1-l)*Fact[Size-i];
    End { For };
    Rank:=Res;
  End { Rank };

Procedure Apply(Const T:Config; X:Char; Var R:Config);
{ R is obtained by applying the basic transformation X }
{ to the configuration T                               }
  Var i:Word;
  Begin
    For i:=1 To Size Do R[i]:=T[BT[X][i]];
  End { Apply };

Procedure Apply_1(Const T:Config; X:Char; Var R:Config);
{ R is obtained by applying the inverse of the basic }
{ transformation X to the configuration T            }
  Var i:Word;
  Begin
    For i:=1 To Size Do R[i]:=T[BT_1[X][i]];
  End {Apply_1};
```

```
Function Equal(Const R,T:Config): Boolean;
{ Checks equality of the configurations R and T }
  Var i:Word;
  Begin
    i:=1;
    While (i<=Size) And (R[i]=T[i]) Do Inc(i);
    Equal:= i>Size;
  End { Equal };

Procedure Generate(Const T: Config);
{ Generates a sequence of basic transformations that transforms the     }
{ initial configuration to T. Last[Rank(T)] will be the last element of }
{ the sequence.                                                         }

{ Global input-output variable: Last }
  Const
    Qs=7000;  { Queue size }
  Var
    Queue:Array[0..Qs-1] Of Config;
    NotFound:Boolean;
    Head,Tail:Word; { head and tail of the queue }
    R,S: Config;
    X: Char;

  Procedure InitGener;
  Var i:Word;
  Begin
    For i:=0 To M Do Last[i]:=' '; { initialize }
    Last[0]:='.';                  { 0=Rank(Ini), sentinel }
  End;

  Procedure InitQueue;
  { initialize the queue }
  Begin
    Head:=0; Tail:=1;
    Queue[0]:=Ini;                 { put Ini into the queue }
  End { InitQueue} ;

  Procedure Enqueue(Const Q:Config);
  Begin
    Queue[Tail]:=Q;
    Inc(Tail); If Tail=Qs Then Tail:=0;
  End { Enqueue };

  Procedure Dequeue(Var Q:Config);
  Begin
    Q:=Queue[Head];
    Inc(Head); If Head=Qs Then Head:=0;
  End { Dequeue };

  Function NotMember(Const Q:Config; X:Char):Boolean;
  { Checks membership of Q in the set of generated configurations.          }
  { If it is not generated then marks it as generated by setting the value }
  {  of Last[Rank(Q)] to X.                                                }
  { Global input-output variable: Last }
  Var RankQ:Word;
  Begin
    RankQ:=Rank(Q);
    If Last[RankQ]=' ' Then Begin
      NotMember:=True;
      Last[RankQ]:=X;
    End Else
```

```
      NotMember:=False;
  End { NotMember };

  Begin { Generate }
    InitGener;
    InitQueue;
    NotFound:=True;
    While NotFound Do Begin
      Dequeue(R);
      For X:='A' To 'C' Do Begin            { apply all basic }
        Apply(R, X, S);                     { transformations to R }
        If NotMember(S,X) Then Begin        { S is a new configuration }
          If Equal(T,S) Then Begin          { T=R*C decomposition found }
            NotFound:= False;
            Break;                          { exit the loop }
          End;
          Enqueue(S);
        End { If new tr. };
      End { For j };
    End { While };
  End { Generate };

Procedure Compose(Const T: Config; Var S:String);
{ Composes the sequence of basic transformations from the array Last }
{ following the link provided by the inverse transformation.         }
  { Global input variable: Last }
  Var
    RankQ:Word; X:Char;
    P,Q  : Config;
  Begin
    Q:=T;
    RankQ:=Rank(Q);
    S:='';
    While RankQ <> 0 Do Begin { while Q<>Ini                  }
      X:=Last[RankQ];
      S:=X+S;               { append X to the left end of S }
      Apply_1(Q,X,P);       { Apply the inverse of X to Q   }
      Q:=P;                 { link to backward              }
      RankQ:=Rank(Q);
    End { While };
  End { Compose };

Procedure WriteOut;
 { Global input variable: Answer }
  Var OutFile:Text;
    L,i:Word;
  Begin
    Assign(OutFile,'output.txt'); Rewrite(OutFile);
    L:=Length(Answer);
    WriteLn(OutFile,L);
    For i:=1 To L Do  WriteLn(OutFile,Answer[i]);
    Close(OutFile);
  End { WriteOut };

Begin { Program }
  ReadInput;
  ComputeFact;
  Generate(T);
  Compose(T, Answer);
  WriteOut;
End.
```