All Rooks Problem

Place N rooks on a N*N board so that no rook can take (check) another. Find all solutions.

A rook can attack along a row or a column. Let (x,y) denote the position of a rook. A solution to the placing N non-attacking rooks on board can be expressed as;
Find the set of pairs

$R = \{ (x,y) \mid 1 \le x \le n \ \& \ 1 \le y \le n\}$

such that no 2 pairs attack each. Two rooks are non-attacking if the lie on different rows and columns. i.e.

Non_Attack : $(i,j) \varepsilon R \ \& \ (k,l) \varepsilon R \Rightarrow i \neq k \ \& \ j \neq l$

A solution to the rook problem is such that only one rook lies in each row and only one rook lies in each column. Under these conditions we can represent a solution by an array R such that

if R@i = j then rook j is in row i,

i.e. a rook is in row i and column j, e.g. R = <<2,1,3,5,4>>
We can justify this mathematically.

Relation

In Maths, a Relation R on a set A is a set of ordered pairs,
$R = \{(x,y) \mid x \ \varepsilon \ A \ \& \ y \ \varepsilon \ A \ \& \ x$ is related to y by a property$\}$
i.e. R is a subset of AxA (Cartesian Product)

Note: (Ordered Pair Property)

$(x,y) = (u,v)$ iff $x=u \ \& \ y=v$

Function

A Relation F is a function iff each argument has at most one value.
i.e. $(x,y) \varepsilon F \ \& \ (u,v) \varepsilon F \ \& \ y \neq v \rightarrow x \neq u$

The solution R to the rooks problem is a function

i.e. Assume $(i,j) \varepsilon R \ \& \ (k,l) \varepsilon R \ \& \ j \neq l$, Show $i \neq k$

Pf:

$(i,j) \varepsilon R \ \& \ (k,l) \varepsilon R \ \& \ j \neq l$
{ A & B → A -- Boolean Algebra }
$(i,j) \varepsilon R \ \& \ (k,l) \varepsilon R$
{ Non_Attack property }
$i \neq k \ \& \ j \neq l$
{ Bool. Alg. }
$i \neq k$

End Pf:

Notation: Since R is function we can write

$(i,j) \varepsilon R$ as $R(i) = j$

or in Eiffel, R.item(i) = j.

Defn. Injective function; 1-1 function

F is injective iff $x \neq y \rightarrow F(x) \neq F(y)$

Show R is an Injective function.

Pf:

Assume $R(i) = j \ \&\ R(k) = l \ \&\ i \neq k$

Show $j \neq l$

$R(i) = j \ \&\ R(k) = l \ \&\ i \neq k$

{ Bool. Alg. }

$R(i) = j \ \&\ R(k) = l$

{ Non-Attack Property }

$i \neq k \ \&\ j \neq l$

{ Bool. Alg. }

$j \neq l$

End Pf:

In Conclusion

A solution to the rooks problem is an injective function onto the set {1..N} where the domain of R is also {1..N} and so R is a Bijective function, since #Dom(R) = #Ran(R) tf. R is permutation on {1..N}

To find All solutions of the Rooks Problem we need a program that will generate all permutations of {1..N}

```
class Gen_Perm
creation  make
feature
        p           :     ARRAY[INTEGER]
        used        :     ARRAY[BOOLEAN]

        All_Perms(k:INTEGER) is
                local
                        j : INTEGER
                do
                        if k > p.size then
                                Print_Perm
                        else
                                from
                                        j := 1
                                until
                                        j > p.size
                                loop
                                        if not used.item(j)  then
                                                p.put(j,k)
                                                used.put(True,j)
                                                All_Perms(k+1)
                                                used.put(False,j)
                                        end
                                        j := j+1
                                end -- loop
                        end
                end -- All_Perms
```

```
    make is
        do
            io.put_string("%N Enter size of Perm. ")
            io.read_integer
            !!p.make(1,io.last_integer)
            !!used.make(1, io.last_integer)
            io.put_string("%N The Permutations are: %N")
            All_Perms(1)
        end -- make

    Print_Perm is
        local
            k : INTEGER
        do
            from
                k := 1
            until
                k > p.size
            loop
                io.put_integer(p.item(k))
                io.putchar(' ')
                k := k+1
            end
            io.new_line
        end -- Print_Perm

end -- Gen_Perm
```

Derangements

A person writes n letters and addresses n envelopes. How many different ways are there of putting all the letters into the wrong envelopes.

The problem describes a permutation p s.t.

$p(i) = j \quad \equiv \quad$ letter i is put into envelope j

How many permutations are there of 1..n s.t. $p(i) \neq i$.
i.e. How many perms have no fixed point.
i.e. How many perms have no 1-cycle

Def^n Derangement

A Derangement of 1..n is a perm p s.t. $p(i) \neq i$.
i.e. p has no 1-cycle.

e.g. n=3; |perms(3)| = 6 <u>Note</u>: |perms(n)| = n!

The perm 1,3,2 describes the range of the function perm,
i.e. p(1) = 1, p(2) = 3 and p(3) = 2

<u>perms</u>	<u>cycle Not^n</u>	
1,2,3	(1)(2)(3)	-- Id
1,3,2	(1)(2 3)	
2,1,3	(1 2)(3)	
2,3,1	(1 2 3)	
3,1,2	(1 3 2)	
3,2,1	(1 3)(2)	

Derange(3) = { (1 2 3), (1 3 2) } -- cycle not^n

Let D(n) = |Derange(n)| -- number of derangements

D(1) = 0; D(2) = 1; D(3) = 2

Defining a function for D(n)

Consider the set of derangements of 1..n. A perm p is written as p(1),p(2) ... p(n). The set of derangements can be partitioned in n-1 subsets according to which of 2,3,...,n is in first position i.e. which of 2,3,..,n equals p(1). Each of these subsets contain the same number of elements.

e.g. n=4, D(4) = 9

Derange(4) = { 2 1 4 3, 2 3 4 1, 2 4 1 3,
3 1 4 2, 3 4 1 2, 3 4 2 1,
4 1 2 3, 4 3 1 2, 4 3 2 1 }

tf. (therefore) $D(n) = (n-1)d(n)$

where
$d(n)$ = # derangements with, say, 2 as the first element.

Such a derangement has the form

$2, p(2), p(3), \ldots, p(n)$ -- $p(i) \neq i$

e.g. $n = 4$,

2 1 4 3, 2 3 4 1, 2 4 1 3

These $d(n)$ derangements can be further partitioned into 2 subsets according as $p(2) = 1$ or $p(2) \neq 1$.

Let $d'(n)$ = # derangements of the form

$2, 1, p(3), p(4), \ldots, p(n)$ -- $p(i) \neq i$

e.g. $n=4$

2 1 4 3

and $d''(n)$ = # derangements of the form

$2, p(2), p(3), \ldots, p(n)$ where $p(2) \neq 1$ and $p(i) \neq i$

e.g. $n=4$

2 3 4 1, 2 4 1 3

Since $d(n) = d'(n) + d''(n)$

$D(n) = (n-1)(d'(n) + d''(n))$

We have $d'(n)$ = #derangements

s.t. $p(1)=2$ and $p(2)=1$ and $p(i) \neq i$ for $i>2$

tf. $d'(n) = D(n-2)$

We have $d''(n)$ = #derangements, p, s.t.

$p(1) = 2$, $p(2) \neq 1$ and $p(i) \neq i$

tf. $d''(n) = D(n-1)$

tf. $D(n) = (n-1)(D(n-2) + D(n-1))$, for $n>2$

$D(1) = 0$,

$D(2) = 1$.

Another Recursive Algorithm for D(n)

For $n>2$, $\quad D(n) = (n-1)(D(n-2) + D(n-1))$

$= (n-1)D(n-2) + (n-1)D(n-1)$

$= n\,D(n-1) - D(n-1) + (n-1)D(n-2)$

tf. $D(n) - n\,D(n-1) = -[D(n-1) - (n-1)D(n-2)]$

$= (-1)^2\,[D(n-2) - (n-2)D(n-3)]$

. . .

by induction $\quad = (-1)^k\,[D(n-k) - (n-k)D(n-(k+1))]$

for $k=n-2$ and so $k+1=n-1$ and $n-k = 2$

$= (-1)^{n-2}\,[D(2) - 2\,D(1)]$

$= (-1)^{n-2}$ since $D(2) = 1$ and $D(1) = 0$

tf. $\quad D(n) = n\,D(n-1) + (-1)^{n-2}$, for $n>2$
but this is also true for $n=2$ as

$D(2) = 1$

$= 2\,D(1) + 1$

tf

$$D(n) = n\,D(n-1) + (-1)^n, \quad \text{for } n>1 \quad \text{-- } (-1)^{n-2} = (-1)^n$$

and $\quad D(1) = 0$

A Non-Recursive Algorithm

Again for n>1,

$$
\begin{aligned}
D(n) &= n\,D(n-1) + (-1)^n \\
&= n[D(n-1) + \frac{(-1)^n}{n}] \\
&= n[(n-1)D(n-2) + (-1)^{n-1} + \frac{(-1)^n}{n}] \\
&= n(n-1)[\,D(n-2) + \ldots + \frac{(-1)^n}{n(n-1)}]
\end{aligned}
$$

by induction:

$$= n(n-1)..(n-k)[\,D(n-(k+1)) + \ldots + \frac{(-1)^n}{n(n-1)..(n-k)}]$$

for k=n-2 and so 2=n-k

$$= n(n-1)..2[\,D(1)) + \ .\ .\ . + \frac{(-1)^n}{n(n-1)..2}]$$

$$\text{tf.}\quad D(n) = n!(1 - \frac{1}{1!} + \frac{1}{2!} - \ldots + \frac{(-1)^n}{n!})$$

<u>Note</u> 1. $1 - \frac{1}{1!} = 0$ -- $D(1) = 0$

<u>Note</u> 2.

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} - \ldots + \frac{x^n}{n!} + \ldots$$

tf.

$$
\begin{aligned}
\frac{1}{e} &= e^{-1} \\
&= 1 - \frac{1}{1!} + \frac{1}{2!} - \ldots + \frac{(-1)^n}{n!} + \ldots
\end{aligned}
$$

<u>End</u>.

$$\text{Since}\quad \sum_{n=0}^{\infty} x_n = \lim_{n\to\infty}\sum_{n=0}^{n} x_n$$

$$
\begin{aligned}
\frac{1}{e} &= \sum_{k=0}^{\infty} \frac{(-1)^k}{k!} \\
&= \lim_{n\to\infty}\sum_{k=0}^{n} \frac{(-1)^k}{k!} \\
&= \lim_{n\to\infty}\frac{D(n)}{n!}
\end{aligned}
$$

From above

$$\frac{n!}{e} = n!\,(1 - \frac{1}{1!} + \frac{1}{2!} - \ldots + \frac{(-1)^n}{n!} + \frac{(-1)^{n+1}}{(n+1)!} + \ldots)$$

$$= D(n) + n!\,(\frac{(-1)^{n+1}}{(n+1)!} + \ldots)$$

tf. $D(n) = \frac{n!}{e} - \frac{(-1)^{n+1}}{(n+1)}$

i.e. $D(n) = \left[\frac{n!}{e}\right]$ -- the nearest integer, n>1

e.g. n = 7

$$D(7) = \left[\frac{7!}{e}\right]$$

$$= \left[\frac{5040}{2.718}\right]$$

$$= 1854$$

Note: $\frac{1}{e} = \frac{D(7)}{7!} + \frac{(-1)^8}{8!}$

$$= \frac{1854}{5040} + 0.0000248 +$$

$$= 0.36785 + 0.0000248 + \ldots$$

$$\approx 0.367944 \quad (= \frac{1}{e} \text{ to 6 decimal places})$$

$\frac{D(7)}{7!}$ approximates $\frac{1}{e}$ correct to 4 decimal places.

Mis-Addressed Letter

There are n! ways of putting the n letters into n envelopes of which D(n) are completely
mis-addressed. The probability of getting at least one letter into the correct envelope is

$$1 - \frac{D(n)}{n!} \quad \text{but} \quad \frac{D(n)}{n!} \approx \frac{1}{e}$$

tf. Prob. of getting at least one letter getting into correct envelpe is

$$1 - \frac{1}{e} \approx 0.63$$

i.e. there is a 63% (i.e. 2 out of 3) chance of getting at least one right.

Table of values for #Derangements

n	Der(n)	n!
1	0	1
2	1	2
3	2	6
4	9	24
5	44	120
6	265	720
7	1,854	5,040
8	14,833	40,320
9	133,496	362,880
10	1,334,961	3,628,800

$$e \approx \frac{87}{32} \approx \frac{878}{323} \approx 2.71828182845$$

$$\frac{1}{e} \approx \frac{32}{87} \approx \frac{323}{878} \approx 0.36787944$$

$D(n) = \left[\frac{n!}{e}\right]$ -- the nearest integer, "Rounding"

e.g. $$D(4) = \left[\frac{24}{e}\right]$$

$$= \left[\frac{24*32}{87}\right] = \left[\frac{768}{87}\right]$$

$$= [8.8]$$

$$= 9$$

Iterative Eiffel functions for D(n).

Given the Specification of D(n) as

$$D(1) = 0 \quad D(2) = 1$$

$$D(n) = (n-1)(D(n-1) + D(n-2)), \ \ n>2$$

We can write the following iterative Eiffel function:

```
Der (n:INTEGER) : INTEGER is
        require
            Pos: n>0
        local
             prev, pres, next, k : INTEGER
        do
            prev := 0
            pres := 1
            from
                k := 2
             invariant
                pres = D(k)
                prev = D(k-1) and k >1
            until
                k = n
            loop
                next := k*(pres + prev)
                prev := pres
                pres := next
                k := k+1
            end
                result := pres
            ensure
                Post: result = D(n)
      end—Der
```

From the recursive definition of D(n) as

$$D(1) = 0$$
$$D(n) = n * D(n-1) + (-1)^n$$

we can write this in Eiffel as the function

```
der(n : INTEGER):INTEGER is
        require
                Pre_der: n > 0
        do
                if n = 1 then
                        result := 0
                else
                        if even(n) then
                                result := n * der(n-1) + 1
                        else
                                result := n * der(n-1) - 1
                        end
                end
        end—der


even(n:INTEGER) : BOOLEAN is
        do
                result := n \\ 2 = 0
        end -- even
```

also an iterative version as

```
der_iter(n : INTEGER):INTEGER is
        require
            Pre_der_iter: n > 0
        local
            r,k,i : INTEGER
        do

            if n = 1 then
                result := 0

            else—n > 1

                from
                    r := 1
                    k := 2
                    i := 1

                invariant

                    inv:  r = der(k) and i = (-1)^k

                until
                    k = n
                loop
                    k := k+1
                    i := -i
                    r := k*r + i
                end

                result := r

            end

    end—der_iter
```

Generate All Derangements of 1..N

Def^n. **Derangement**

A Derangement of 1..n is a permutation p
s.t. $p(i) \neq i$.

Let us write the perm. p in terms of its range,
i.e. if p(1) = 3, p(2) = 1 and p(3) = 2 then we can write this as p = (3,1,2)
i.e. p = (p(1), p(2), p(3))

A derangement d of 1..N is permutation
(d(1), d(2), d(3), ... d(N)) where

$d(1) \neq 1,\ d(2) \neq 2,\ d(3) \neq 3,\ \ldots,\ d(N) \neq N$

<u>Perms</u>:	1,2,3	1,3,2	2,1,3	2,3,1	3,1,2	3,2,1
<u>Derangements</u>				2,3,1	3,1,2	

The set of derangements of 1..3 = { (2,3,1), (3,1,2) }

The #derangements of 1..N = $\left[\dfrac{N!}{e} \right]$ -- nearest integer

To generate all the derangement we adapt the procedure for All_Perms.

```
All_Ders(k:INTEGER) is
    local
        j : INTEGER
    do
        if k > p.size then
            Print_Perm
        else
            from
                j := 1
            until
                j > p.size
            loop
                if not used.item(j) and j /= k  then
                    p.put(j,k)
                    used.put(True,j)
                    All_Ders(k+1)
                    used.put(False,j)
                end
                j := j+1
            end
        end
    end -- All_Ders
```

The Class for Generating Derangements

We can take advantage of the class GEN_PERM to construct a class for GEN_DER. To do this we make a simple use of inheritance. Let the class GEN_DER inherit all the attributes and features of GEN_PERM except that we will redefine the critical procedure All_Perms.

This use of inheritance saves us rewriting common routines and is more like 'including' the file for GEN_PERM. rather than true inheritance

Note:
It is not possible to redefine and rename at the same time.

Combinations: Choose k items from N items

Generate all combinations of 3 items from 5 items.

1, 2, 3	1, 2, 4	1, 2, 5	1, 3, 4	1, 3, 5	1,4,5
2, 3, 4	2, 3, 5	2, 4, 5			
3, 4, 5					

The number of way of choosing k from N is given by $\binom{N}{k}$ "N choose k"

Consider $\binom{N}{k}$

In choosing, we have

1st	choice: any from	N
2nd	"	N -1
3rd	"	N - 2
....		...
kth	"	N - (k-1)

i.e. total = N *(N-1)*(N-2)* ... *(N-(k-1)

but choosing say 1,2,3 is the same as choosing 2,1,3.
There are 3! ways of arranging 1,2,3
i.e. there are k! ways of arranging or permuting k things.
In the above we have choosen k! times too many and so

$$\binom{N}{k} = \frac{N*(N-1)..N-(k-1)}{k!}$$

$$= \frac{N!}{k!*(N-k)!}$$

Note:

$$\binom{N}{N-k} = \binom{N}{k}$$

In generating combinations we generate those perms of size k that are ordered or sorted,
e.g. in the above, we have 1, 3, 5 and never 1, 5, 3.

We can regard 1, 3, 5 as a representative of all the arrangements or perms of 1, 3, 5

```
All_Combs(i,N,k,Start:INTEGER) is
    local
        j : INTEGER
    do
        if i > k then
            Print_Comb(k)
        else
            from
                j := Start
            until
                j > N
            loop
                comb.put(j,i)
                All_Combs(i+1,N,k,j+1)
                j := j+1
            end
        end
    end -- All_Combs
```

Methods for Generating Permutations

We are viewing a permutation as an ARRAY[INTEGER], where the indexing is from 1..N and the values are 1..N and the array has the property of being bi-jective.

Backtracking

This is the method used above in All_Perms. The permutation are generated in 'Lexicographical' order
i.e. in 'increasing size'.

From	1, 2, 3, ... , N
	....
To	N, N-1, ... , 2, 1

Recursion (Horowitz & Sahni)

```
make is
    local
        p : ARRAY[INTEGER]
        i : INTEGER
    do
        io.put_string("%N Enter size of Perm. ")
        io.get_int
        !!p.make(1,io.last_int)
        from
            i := 1
        until
            i > p.count
        loop
            p.put(i,i)
            i := i+1
        end
        All_Perms_HS(P,1)
    end -- make
```

```
All_Perms_HS(A0:ARRAY[INTEGER], k:INTEGER) is
    local
        A : ARRAY[INTEGER]
        j, it : INTEGER
    do
        if k = A0.count then
            Print_Perm(A0)
        else
            !!A.make(1, A0.count)
            A.copy(A0)
            from
                j := k
            until
                j > A.count
            loop
                it := A.item(j)
                A.put(A.item(k), j)
                A.put(it,k)
                All_Perms_HS(A, k+1)
                j := j+1
            end
        end
    end -- All_Perms_HS
```

```
class Gen_Perm_HS

creation
        make

feature


make is

        local
                p : ARRAY[INTEGER]
                i,n : INTEGER
        do
                io.put_string(
                        "%NSize of Perm? ")
                io.read_integer
                n := io.last_integer
                !!p.make(1,n)
                from
                        i := 1
                until
                        i > p.count
                loop
                        p.put(i,i)
                        i := i+1
                end
                io.put_string(
                                "%N Perms =%N")
                All_Perms_HS(P,1)

        end -- make


Print_Perm1(p :ARRAY[INTEGER]) is
        local
                k : INTEGER
        do
                from
                        k := 1
                until
                        k > p.size
                loop
                        io.putint(p.item(k))
                        io.putchar(' ')
                        k := k+1
                end
                io.new_line
         end -- Print_Perm1
```

```
class Gen_Perm

creation
        make
feature

p : ARRAY[INTEGER]
used:ARRAY[BOOLEAN]


make is
        local
                n : INTEGER

        do

                io.put_string(
                        "%NSize of Perm?")

                io.read_integer

                n := io.last_integer

                !!p.make(1,n)

                !!used.make(1,n)

                io.put_string(
                "%N Perms  = %N")

                All_Perms(1)

end -- make


Print_Perm0 is
        local
                k : INTEGER
        do
                from
                        k := 1
                until
                        k > p.size
                loop
                        io.putint(p.item(k))
                        io.putchar(' ')
                        k := k+1
                end
                io.new_line
        end -- Print_Perm0
```

```
All_Perms_HS(
      A0:ARRAY[INTEGER],
      k : INTEGER) is

 local

      A : ARRAY[INTEGER]
      j, it : INTEGER

 do

      if k = A0.count then
            Print_Perm1(A0)

      else

            !!A.make(1,A0.count)
            A.copy(A0)
            from
                  j := k
            until
                  j > A.count
            loop
                  it := A.item(j)
                  A.put(A.item(k), j)
                  A.put(it,k)
                  All_Perms_HS(A,k+1)
                  j := j+1
            end
      end

 end -- All_Perms_HS


end -- Gen_Perm_HS
```

```
All_Perms(k:INTEGER) is

 local

      j : INTEGER

 do

      if k > p.size then
            Print_Perm0

      else

       from
            j := 1
       until
            j > p.size
       loop

            if not used.item(j) then
                  p.put(j,k)
                  used.put(True,j)
                  All_Perms(k+1)
                  used.put(False,j)
            end
            j := j+1
       end

  end

end -- All_Perms


end -- Gen_Perm
```