Data Structures

Recursive Programming

Repeating a Code Segment

- There are two basic methods for repeating a code segment:
  - Loops (do-while, while, for, ...)
  - Recursion

- Recursion: a function calling itself
  - Simplifies code writing for the solutions to some types of problems.
  - Is an important advanced programmed technique and shortens the code when used appropriately.
  - Has a downside: The program may take longer to run.

Recursive Programming

- The basic approach is to define the problem to be solved in terms of simpler subproblems.
- The recursive function can solve the base case and returns the result if there is one.
- If the input problem is not the base case, the problem can be divided into simple parts, and the function calls itself for each part.
- Since recursive program writing is different from the usual style, it takes more practice to learn.

Example: Computing the Factorial

- Definition of the factorial function:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 3 \cdot 2 \cdot 1 & \text{if } n \geq 1 \end{cases}$$

$n! = n \cdot (n - 1) \cdot \cdots \cdot 3 \cdot 2 \cdot 1$

$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$

- We could also define the function in terms of itself:

Example: fact(5) = $5 \cdot (4 \cdot 3 \cdot 2 \cdot 1) = 5 \cdot$ fact(4)

$$f(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot f(n-1) & \text{else} \end{cases}$$

Factorial Function

```
int recursiveFactorial(int n) {
   int m;
   if (n == 0) return 1;   // base case
   else {                  // recursion step
      m = recursiveFactorial(n - 1);
      return n * m;
   }
}
```

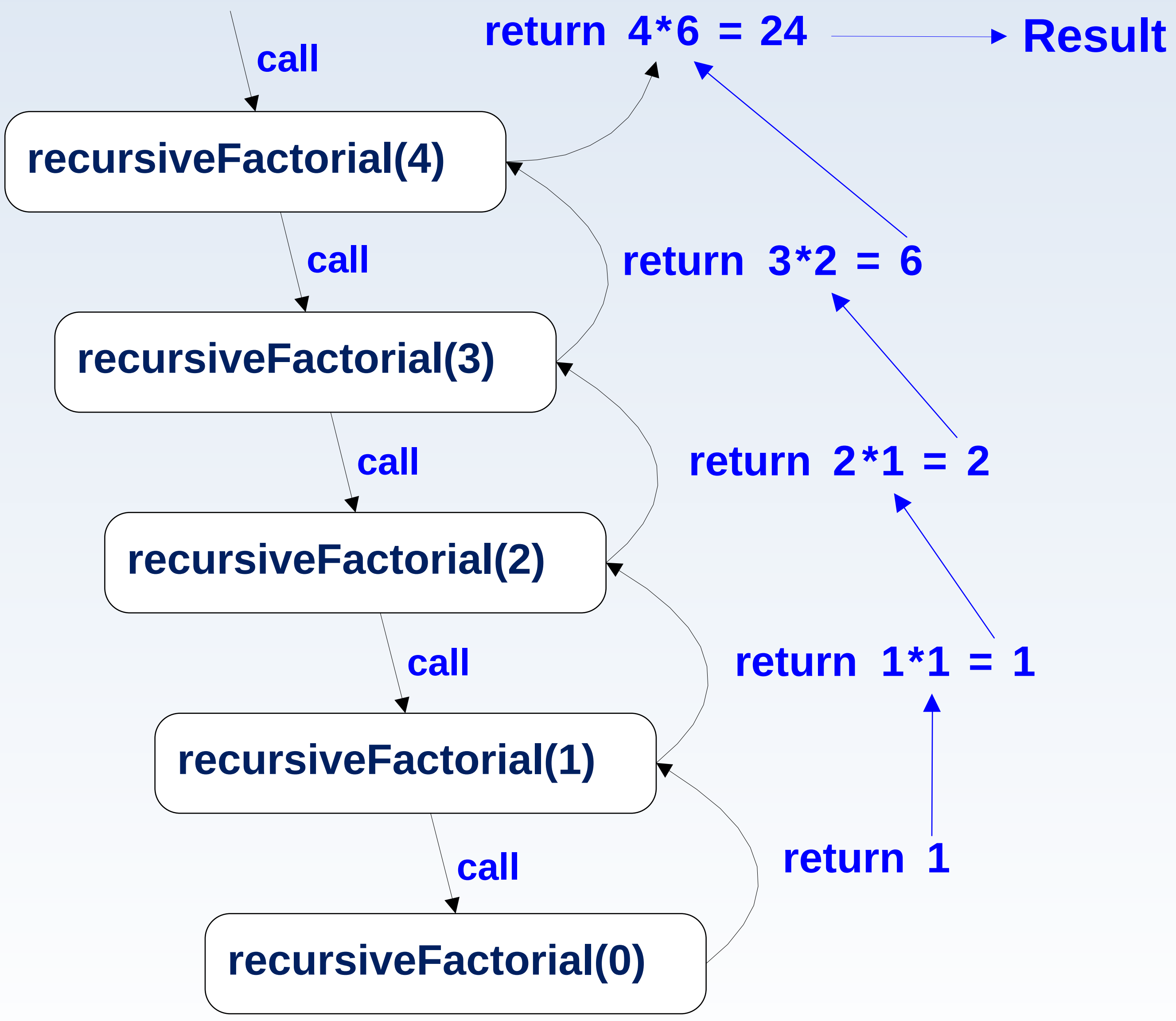
Factorial Function

```
int recursiveFactorial(int n) {
   int m;
   if (n == 0) return 1;   // base case
   else {                  // recursion step

      return n * recursiveFactorial(n - 1);

   }
}
```

Note:

The base case is, at the same time, the termination condition of the function and should not be left out. Otherwise, there will be “infinite” recursion (analogous to the problem of an infinite loop in an iterative solution).

Exercise


call
return 4*6 = 24
Result
recursiveFactorial(4)
call
return 3*2 = 6
recursiveFactorial(3)
call
return 2*1 = 2
recursiveFactorial(2)
call
return 1*1 = 1
recursiveFactorial(1)
call
return 1
recursiveFactorial(0)

Is Recursion Always Useful?

- Let us write the same function iteratively:

```
int iterativeFactorial(int n) {
  int i, p;
  p = 1;
  for (i = 2; i <= n; i++)
    p *= i;
  return p;
}
```

- This solution wastes less space in memory and works faster.

Recursion Types

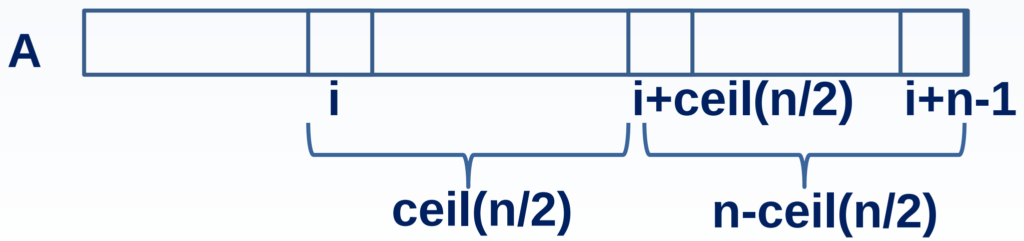
- Linear
  - Each time a function (method) is called, it makes at most one recursive call.
- Tail
  - Tail recursion occurs when a linearly recursive function makes its recursive call as its last step.
  - These types of recursive functions can be easily converted to iterative functions.
- Binary
  - If the function calls itself twice at a time (i.e., there are two recursive calls for each non-base case), the function uses binary recursion.

Binary Recursion Example

Let us write the function that sums elements of an array.

Algorithm:

```
BinarySum(A,i,n)
    Input:    array A, integers i and n
    Output: sum of n numbers in array A, starting with
            index i
if (n == 1) then
    return A[i]
else
    return BinarySum(A, i, ceil(n/2))
           + BinarySum(A, i+ceil(n/2), n-ceil(n/2))
```



BinarySum() Call Sequence

$$\begin{tikzcd}
 & & & BS(0, 7) \arrow[dll] \arrow[dr] & & \\
 & BS(0, 4) \arrow[dl] \arrow[dr] & & & BS(4, 3) \arrow[d] \arrow[dr] & \\
BS(0, 2) \arrow[d] \arrow[dr] & & BS(2, 2) \arrow[d] \arrow[dr] & & BS(4, 2) \arrow[d] \arrow[dr] & BS(6, 1) \\
BS(0,1) & BS(1,1) & BS(2,1) & BS(3,1) & BS(4,1) & BS(5,1)
\end{tikzcd}$$

- Binary recursion is generally used in algorithms designed with the "divide and conquer" method.
- In these types of algorithms, recursion may prove more effective than an iterative solution.

Permutation

- Example: permutations of the set {1, 2, 3} :

1 2 3	1 3 2
2 1 3	2 3 1
3 1 2	3 2 1

- To generate these permutations, we could use:

```
for (i1 = 1; i1 <= 3; i1++) {
  for (i2 = 1; i2 <= 3; i2++) {
    if (i2 != i1) {
       for (i3 = 1; i3 <= 3; i3++) {
           if ( (i3 != i1) && (i3 != i2) ) {
              cout << i1 << “ ”<< i2 << “ ”<< i3 << endl;
           }
       }
    }
  }
}
```

- However, this code can only generate ternary (3-ary) permutations. How do we generate n-ary permutations?

Recursive Solution for Permutation

```
void perm(int *list, int k, int m) {
  int i;
  if (k == m) { // just print out list
       for (i = 0; i <= m; i++) {
              cout << list[i] << " ";
       }
       cout << endl;
  }
  else {
       for (i = k; i <= m; i++){
              swap(&list[k], &list[i]);
              perm(list, k + 1, m);
              swap(&list[k], &list[i]);
       }
  }
}
```

list[0..x] -> an array storing the data that can be printed to the screen
k: starting index
m: ending index
First call: perm(list, 0, x)
Ex: perm(list,0,3) for an array of size 4 elements

```
void swap(int *a, int *b) {
       int temp = *a;
       *a = *b;
       *b = temp;
}
```

Recursive Solution for Permutation

- At each step, a position is held constant, and the others are permuted.
- For example, when writing the permutation of (1 2 3 4), these numbers are placed into the first position in order, and the remaining three positions are written as a permutation.

1 … permutations that start with 1
2 … permutations that start with 2
3 …
4 …

- Thus, the problem is divided into subproblems.

Eight Queens Problem

- A classical problem
- Goal: Place 8 queens on a 8 X 8 chessboard such that they cannot attack each other
- Two queens cannot be located on the
  - Same row
  - Same column
  - Same diagonal
- A solution for a 5 X 5 board is shown on the right.
  - At the top of every column, the number of the row where the queen is located is shown.

5	3	1	4	2
		q		
				q
	q			
			q	
q				

Solution to the Eight Queens Problem

- The problem has more than one solution.
- Since only one queen may be located on a row, the potential solutions may be found by generating permutations of the row numbers shown on the example board.
  - All permutations may not be solutions.
  - However, all solutions are among these permutations.

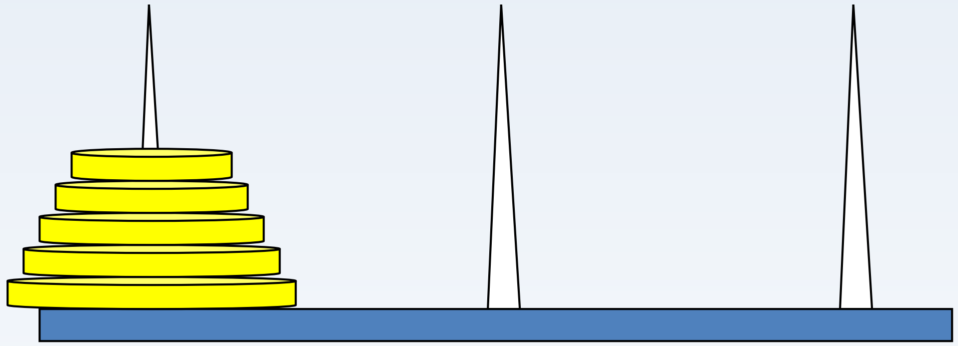
4	1	3	5	2
	q			
				q
		q		
q				
			q	

Solution to the Eight Queens Problem

- A permutation generator with the addition of "diagonal attack" checks could be used to solve the n queens problem.

Towers of Hanoi

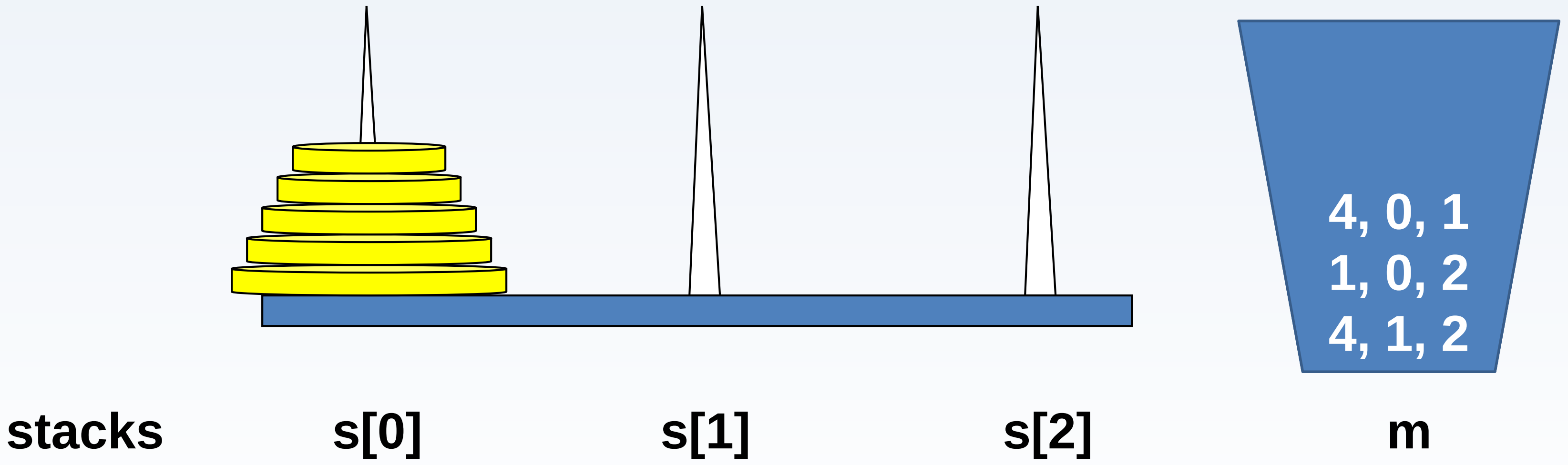
- The Towers of Hanoi problem can easily be solved using recursion.



stacks **s[0]** **s[1]** **s[2]**

Hanoi Iterative Solution (reminder)

- As long as the move stack is full, we pop a move (n, source, destination) from this stack, and push these moves onto the stack:
    - (n - 1, temp, destination)
    - (1, source, destination)
    - (n - 1, source, temp) (what has to be done first is at the top of the stack)



Hanoi Recursive Solution

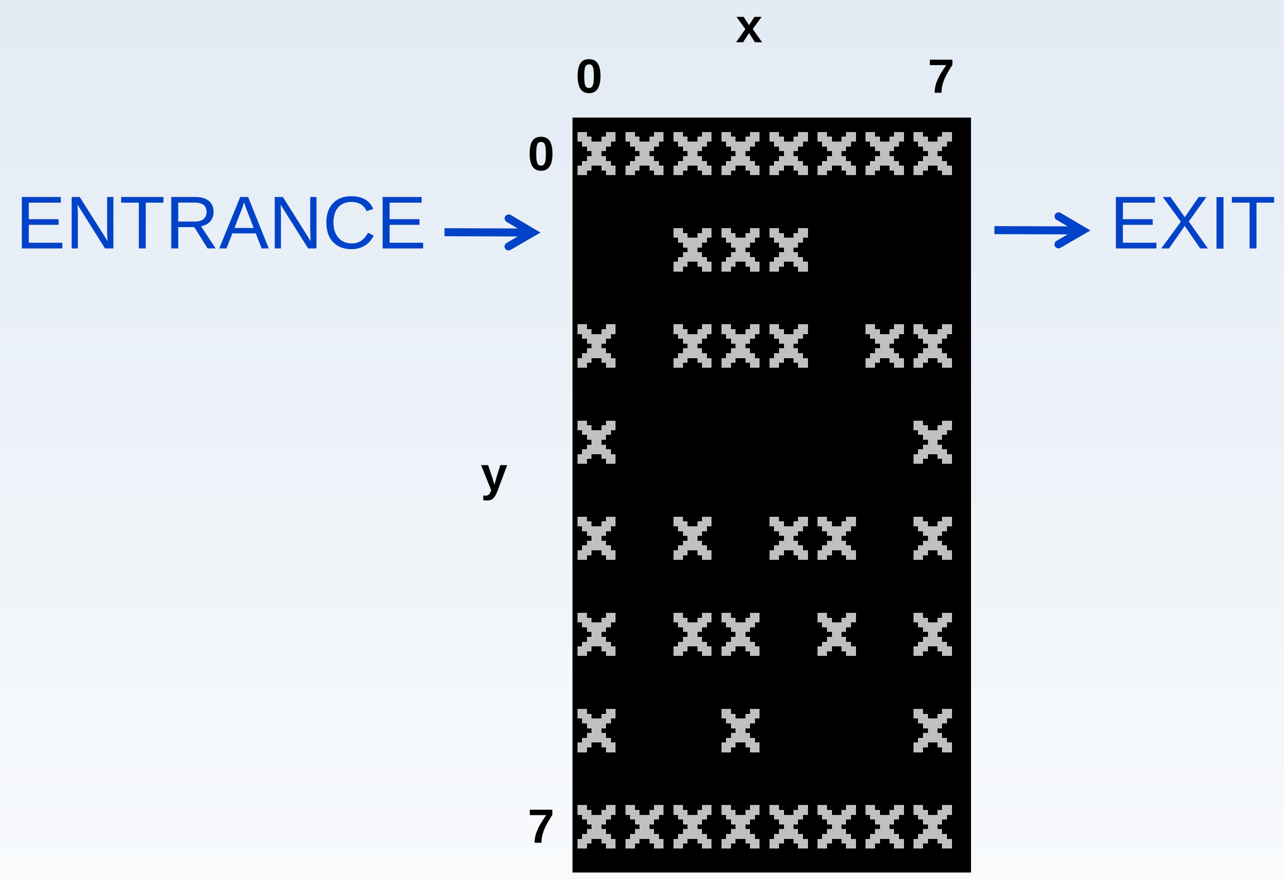
```
void Hanoi_recursive(int n, int source, int destination, int temp) {
   if (n >= 1) {
         Hanoi_recursive(n - 1, source, temp, destination);
         s[destination].push( s[source].pop() );
         Hanoi_recursive(n - 1, temp, destination, source);
   }
}

Hanoi_recursive(5, 0, 2, 1);
```

Finding a Path in a Labyrinth

- The problem of finding a path in a labyrinth, which we had previously solved using a stack, could also be solved by writing a recursive function.

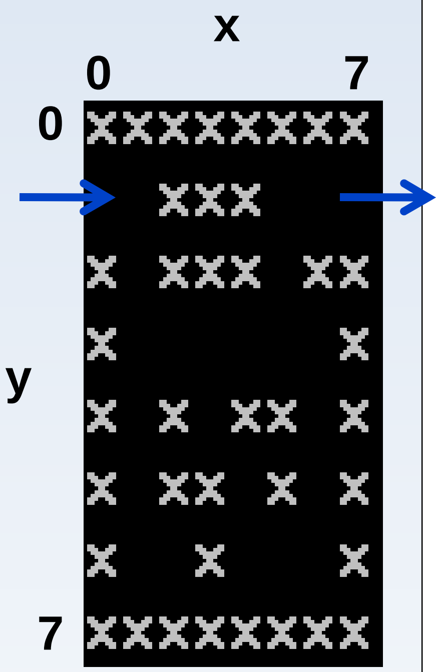
```
int main(){
   char lab[8][8] = {{'x','x','x','x','x','x','x','x'},
                     {' ',' ','x','x','x',' ',' ',' '},
                     {'x',' ','x','x','x',' ','x','x'},
                     {'x',' ',' ',' ',' ',' ',' ','x'},
                     {'x',' ','x',' ','x','x',' ','x'},
                     {'x',' ','x','x',' ','x',' ','x'},
                     {'x',' ',' ','x',' ',' ',' ','x'},
                     {'x','x','x','x','x','x','x','x'}
   };
   if ( find_path(lab, entrance.y, entrance.x, LEFT) )
        cout << "PATH found“ << endl;

   printlab(lab);
   return EXIT_SUCCESS;
}
```

Finding a Path in a Labyrinth



Recursive Path Finding Function

```
bool find_path(char lab[8][8], int y, int x, int camefrom) {
   lab[y][x] = 'o';
   if ( x == lab_exit.x && y == lab_exit.y )
         return true;
   printlab(lab);
   if ( x > 0 && lab[y][x - 1] != 'x' && camefrom != LEFT )
         if ( find_path(lab, y, x - 1, RIGHT) ) // go left
                return true;
   if ( y < 7 && lab[y + 1][x] != 'x' && camefrom != DOWN )
         if ( find_path(lab, y + 1, x, UP) ) // go down
                return true;
   if ( y > 0 && lab[y - 1][x] != 'x' && camefrom != UP )
         if ( find_path(lab, y - 1, x, DOWN)) // go up
                return true;
   if ( x < 7 && lab[y][x + 1] != 'x' && camefrom != RIGHT)
         if ( find_path(lab, y, x + 1, LEFT) ) // go right
                return true;
   lab[y][x] = ' '; // delete incorrect paths
   printlab(lab); // display the return from the incorrect paths as well
   return false;
}
```



Finding a Path in a Labyrinth

1
```
xxxxxxxx
o xxx
x xxx xx
x      x
x x xx x
x xx x x
x  x   x
xxxxxxxx
```

2
```
xxxxxxxx
ooxxx
x xxx xx
x      x
x x xx x
x xx x x
x  x   x
xxxxxxxx
```

3
```
xxxxxxxx
ooxxx
xoxxx xx
x      x
x x xx x
x xx x x
x  x   x
xxxxxxxx
```

4
```
xxxxxxxx
ooxxx
xoxxx xx
xo     x
x x xx x
x xx x x
x  x   x
xxxxxxxx
```

5
```
xxxxxxxx
ooxxx
xoxxx xx
xo     x
xox xx x
x xx x x
x  x   x
xxxxxxxx
```

6
```
xxxxxxxx
ooxxx
xoxxx xx
xo     x
xox xx x
xoxx x x
x  x   x
xxxxxxxx
```

7
```
xxxxxxxx
ooxxx
xoxxx xx
xo     x
xox xx x
xoxx x x
xo x   x
xxxxxxxx
```

8
```
xxxxxxxx
ooxxx
xoxxx xx
xo     x
xox xx x
xoxx x x
xoox   x
xxxxxxxx
```

9
```
xxxxxxxx
ooxxx
xoxxx xx
xo     x
xox xx x
xoxx x x
xo x   x
xxxxxxxx
```

10
```
xxxxxxxx
ooxxx
xoxxx xx
xo     x
xox xx x
xoxx x x
x  x   x
xxxxxxxx
```

Finding a Path in a Labyrinth

11
```
xxxxxxxx
ooxxx
xoxxx xx
xo     x
xox xx x
x xx x x
x  x   x
xxxxxxxx
```

12
```
xxxxxxxx
ooxxx
xoxxx xx
xo     x
x x xx x
x xx x x
x  x   x
xxxxxxxx
```

13
```
xxxxxxxx
ooxxx
xoxxx xx
xoo    x
x x xx x
x xx x x
x  x   x
xxxxxxxx
```

14
```
xxxxxxxx
ooxxx
xoxxx xx
xooo   x
x x xx x
x xx x x
x  x   x
xxxxxxxx
```

15
```
xxxxxxxx
ooxxx
xoxxx xx
xooo   x
x xoxx x
x xx x x
x  x   x
xxxxxxxx
```

16
```
xxxxxxxx
ooxxx
xoxxx xx
xooo   x
x x xx x
x xx x x
x  x   x
xxxxxxxx
```

17
```
xxxxxxxx
ooxxx
xoxxx xx
xoooo  x
x x xx x
x xx x x
x  x   x
xxxxxxxx
```

18
```
xxxxxxxx
ooxxx
xoxxx xx
xooooo x
x x xx x
x xx x x
x  x   x
xxxxxxxx
```

19
```
xxxxxxxx
ooxxx
xoxxxoxx
xooooo x
x x xx x
x xx x x
x  x   x
xxxxxxxx
```

20
```
xxxxxxxx
ooxxxo
xoxxxoxx
xooooo x
x x xx x
x xx x x
x  x   x
xxxxxxxx
```

21
```
xxxxxxxx
ooxxxoo
xoxxxoxx
xooooo x
x x xx x
x xx x x
x  x   x
xxxxxxxx
```

22
```
xxxxxxxx
ooxxxooo
xoxxxoxx
xooooo x
x x xx x
x xx x x
x  x   x
xxxxxxxx
```