Stacks（2）
Stack Applications

College of Computer Science, CQU

Application 3
Converting Infix to RPN

- RPN?

  Reverse Polish Notation

  A notation for arithmetic expressions:

  operators are written after the operands.
- Infix notation: operators written between the operands
- Postfix notation (RPN): operators written after the operands
- Prefix notation : operators written before the operands
- Examples:

INFIX	RPN (POSTFIX)	PREFIX
A + B	A B +	+ A B
A * B + C	A B * C +	+ * A B C
A * (B + C)	A B C + *	* A + B C
A–(B –(C – D))	A B C D - - -	- A - B - C D
A – B – C – D	A B - C - D -	- - - A B C D

Application 3
Converting Infix to RPN

1. Initialize an empty stack of operators.

2. While no error has occurred and end of infix expression not reached

a. Get next Token in the infix expression.

b. If Token is

(i) a left parenthesis: Push it onto the stack.

(ii) a right parenthesis:Pop and display stack elements until a left parenthesis is encountered,but do not display it. (Error if stack empty with no left parenthesis found.)

(iii) an operator:If stack empty or Token has higher priority than top stack element, push Token on stack.(Left parenthesis in stack has lower priority than operators)

Otherwise, pop and display the top stack element; then repeat the comparison of Token with new top stack item.

(iv) an operand: Display it.

3. When end of infix expression reached, pop and display stack items until stack is empty.

算符间的优先关系:

θ_1 \ θ_2	+	-	*	/	(	)	#
+	>	>	<	<	<	>	>
-	>	>	<	<	<	>	>
*	>	>	>	>	<	>	>
/	>	>	>	>	<	>	>
(	<	<	<	<	<	≒	
)	>	>	>	>		>	>
#	<	<	<	<	<		=

Precede: 判定运算符栈的栈顶运算符θ_1与读入的运算符θ_2之间的优先关系的函数。

Operate: 进行二元运算aθb的函数.

Application 3
Converting Infix to RPN

- Example: ((A+B)*C)/(D-E)

Push (		
Push (		
Display A	A	
Push +		
Display B	AB	
Read)		
Pop +, Display +, Pop (	AB+	
Push *		
Display C	AB+C	
Read)		
Pop *, Display *, Pop (	AB+C*	// stack now empty
Push /		
Push (		
Display D	AB+C*D	
Push –		
Display E	AB+C*DE	
Read)		
Pop -, Display -, Pop (	AB+C*DE-	
Pop /, Display /	AB+C*DE-/	

Application 4
Evaluating RPN Expressions

- "By hand": Underlining technique:

1. Scan the expression from left to right to find an operator.
2. Locate ("underline") the last two preceding operands and combine them using this operator.
3. Repeat until the end of the expression is reached.

- Example:

2 3 4 + 5 6 - - *

→ 2 $\underline{3\ 4\ +}$ 5 6 - - *

→ 2 7 5 6 - - *

→ 2 7 $\underline{5\ 6\ -}$ - *

→ 2 7 -1 - *

→ 2 $\underline{7\ -1\ -}$ *

→ 2 8 * → $\underline{2\ 8\ *}$ → 16

Application 4
Evaluating RPN Expressions

- STACK ALGORITHM

  Receive:An RPN expression.

  Return: A stack whose top element is the value of RPN
  expression(unless an error occurred).

1. Initialize an empty stack.
2. Repeat the following until the end of the expression is encountered:

a. Get next token (constant, variable, arithmetic operator) in the RPN expression.

b. If token is an operand, push it onto the stack.

   If it is an operator, then

(i) Pop two values from the stack.

   If stack does not contain two items, error due to a malformed RPN Evaluation terminated

(ii) Apply the operator to these two values.

(iii) Push the resulting value back onto the stack.

3. When the end of expression encountered, its value is on top of the stack
   (and, in fact, must be the only value in the stack).

Application 4
Evaluating RPN Expressions

- Unary minus causes problems

Example:

5 3 - -

→ 5 3 - -

→ 5 -3 - → 8

5 3 - -

→ 5 3 - -

→ 2 - → -2

Use a different symbol:

5 3 ~ -

5 3 - ~

Application 4
Evaluating RPN Expressions

- Example: 2 3 4 + 5 6 - - *

Push 2

Push 3

Push 4

Read +

    Pop 4, Pop 3, 3 + 4 = 7

    Push 7

Push 5

Push 6

Read –

    Pop 6, Pop 5, 5 - 6 = -1

    Push -1

Read –

    Pop -1, Pop 7, 7 - -1 = 8

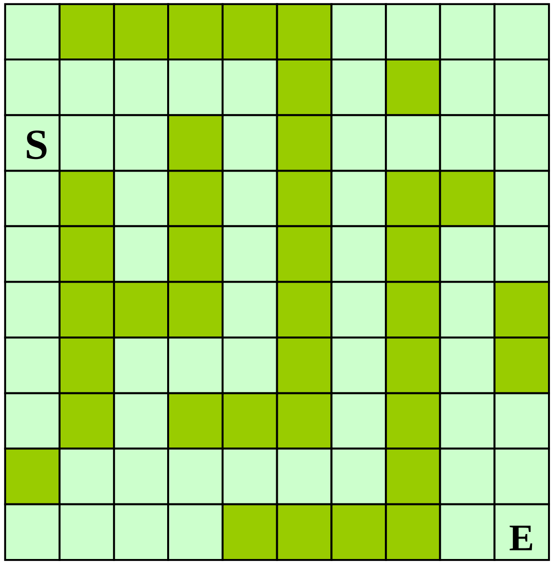
    Push 8

Read *

    Pop 8, Pop 2, 2 * 8 = 16

Application 5
Maze

- A maze is a rectangular area with an entrance and an exit.
- The interior of a maze contains walls or obstacles that one cannot walk through.
- We view the maze as broken up into equal size squares, some of which are part of the walls and the others are part of the hallways. Thus they are "open".
- One of the open squares is designated as the Start position and another as the Exit position.



Maze M

Application 5
Maze

- So we will use an enumerated type which will include constants OPEN and WALL.

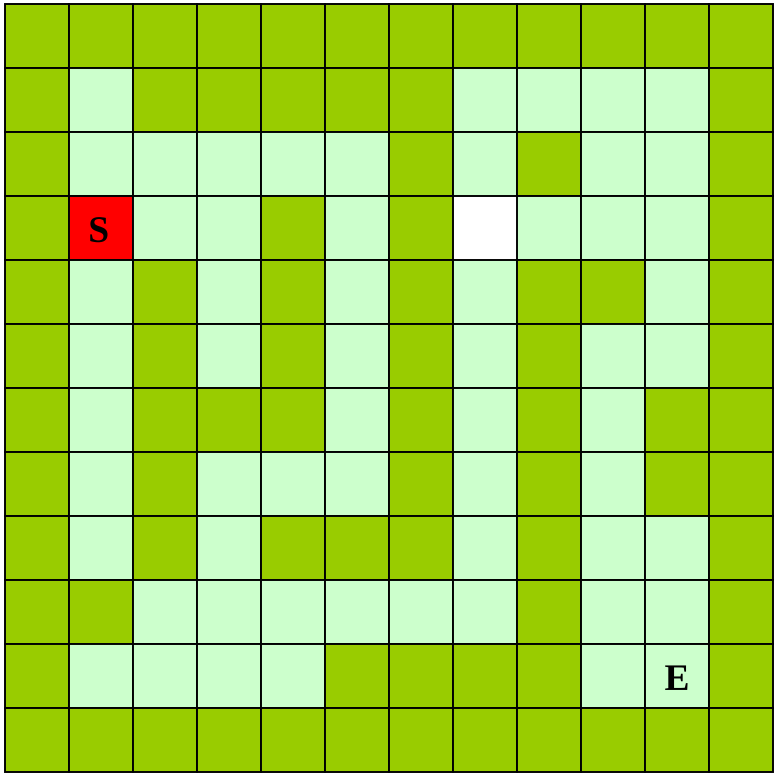
- To reflect our search method for finding an exit path, we will also have a

  third state: VISITED.

- In traversing the maze, we can only move forward into an OPEN position .

- Thus a given position has 4 possible neighboring positions in the directions east, south, west and north.

Searching the Example Maze

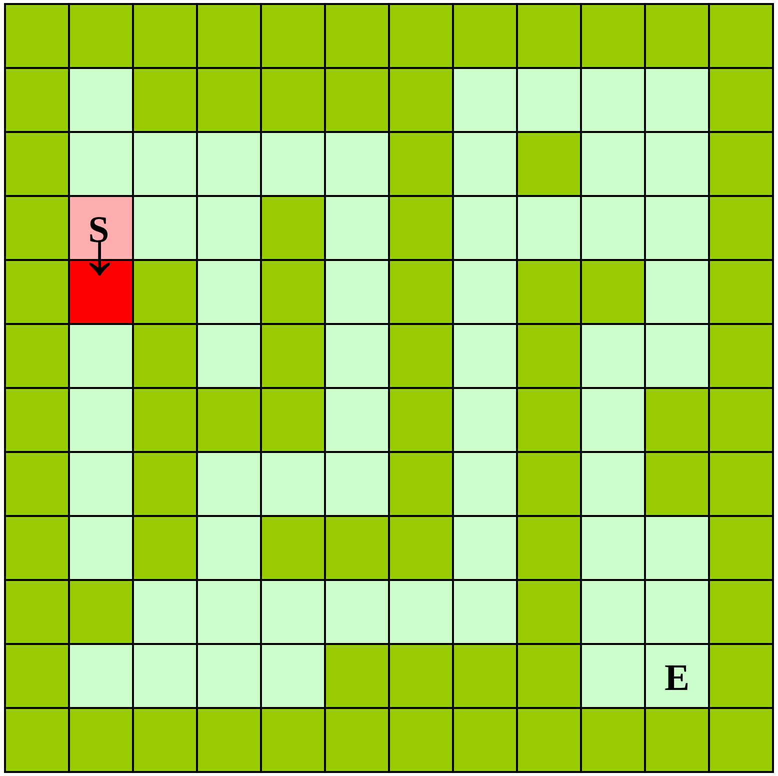


Current position 🟥

Visited and pending
(on current path) ▨

Visited and Done
(not on current path) 🟧

Searching the Example Maze



Arrows indicate the current path

Current position

🟥

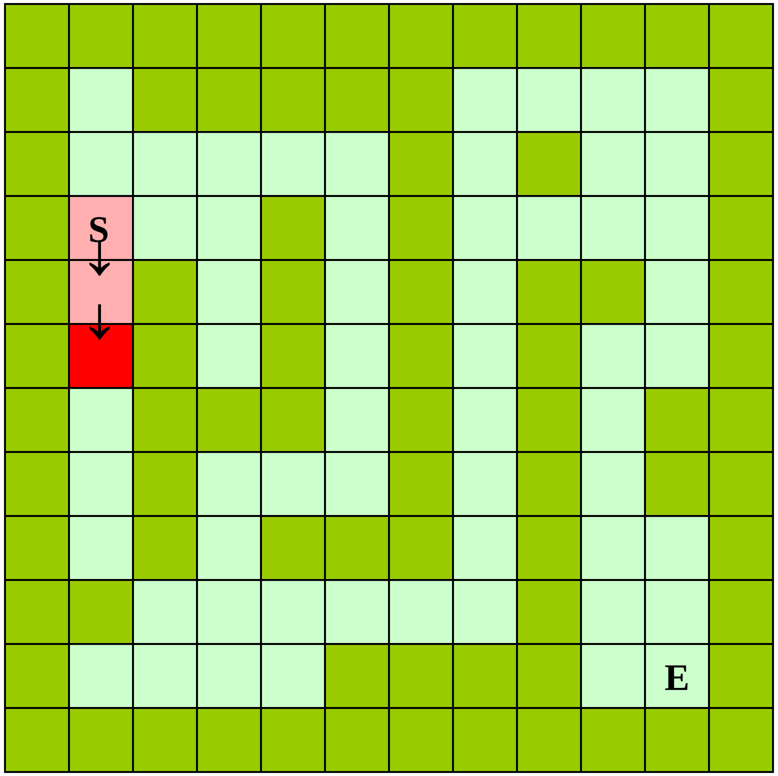
Visited and pending
(on current path)

◻

Visited and Done
(not on current path)

🟫

Searching the Example Maze



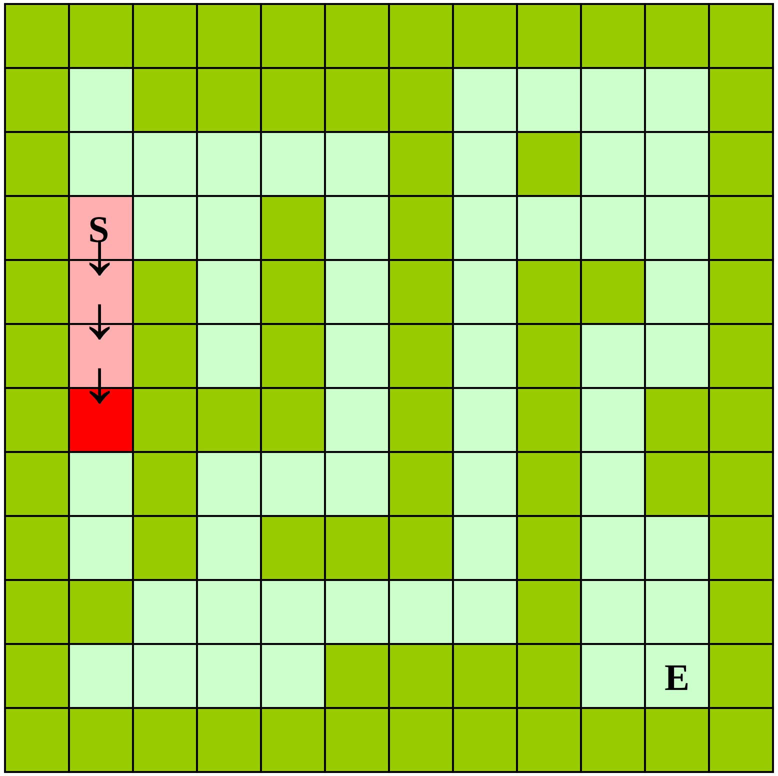
Arrows indicate the current path

Current position 🟥

Visited and pending (on current path) ▨

Visited and Done (not on current path) 🟫

Searching the Example Maze



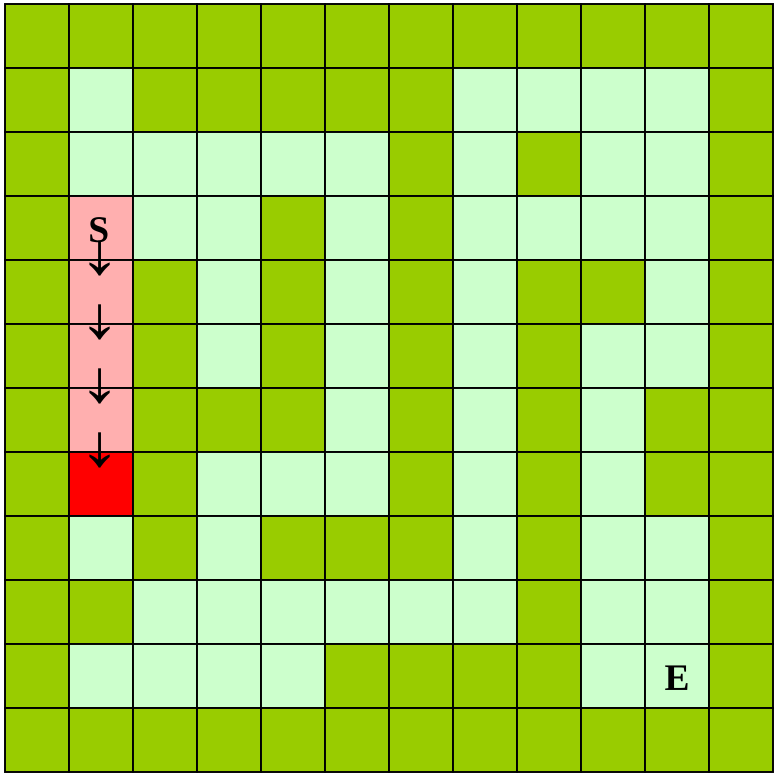
Arrows indicate the current path

Current position 🟥

Visited and pending (on current path) ▨

Visited and Done (not on current path) 🟫

Searching the Example Maze



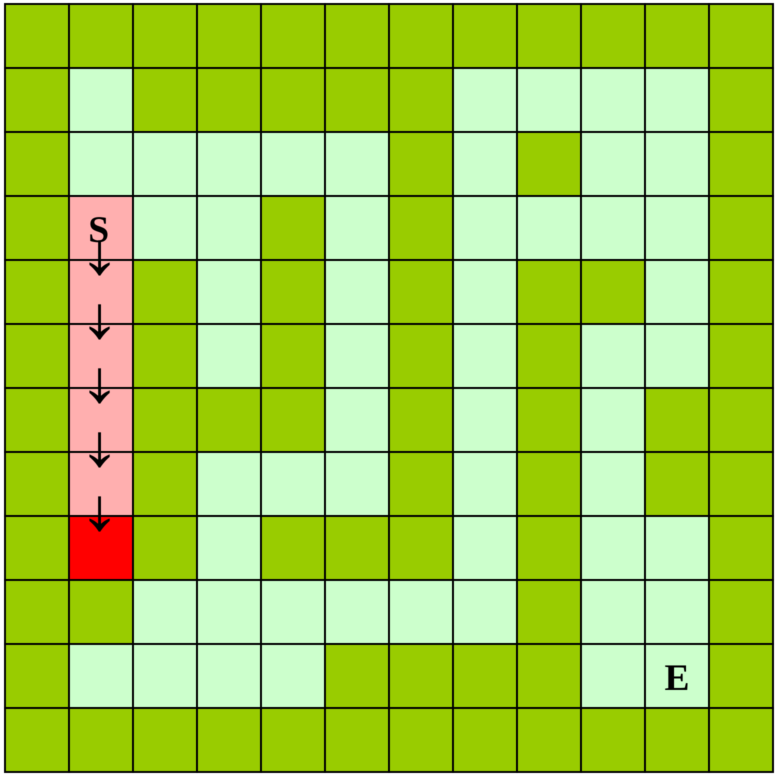
Arrows indicate the current path

Current position 🟥

Visited and pending (on current path) ▨

Visited and Done (not on current path) 🟫

Searching the Example Maze



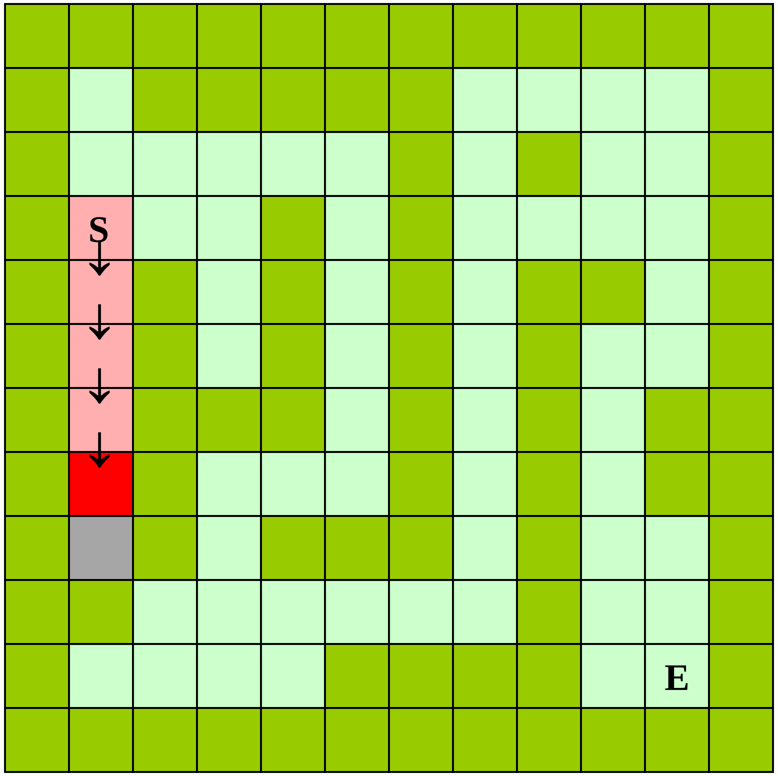
Arrows indicate the current path

Current position 🟥

Visited and pending (on current path) ▨

Visited and Done (not on current path) 🟧

Searching the Example Maze



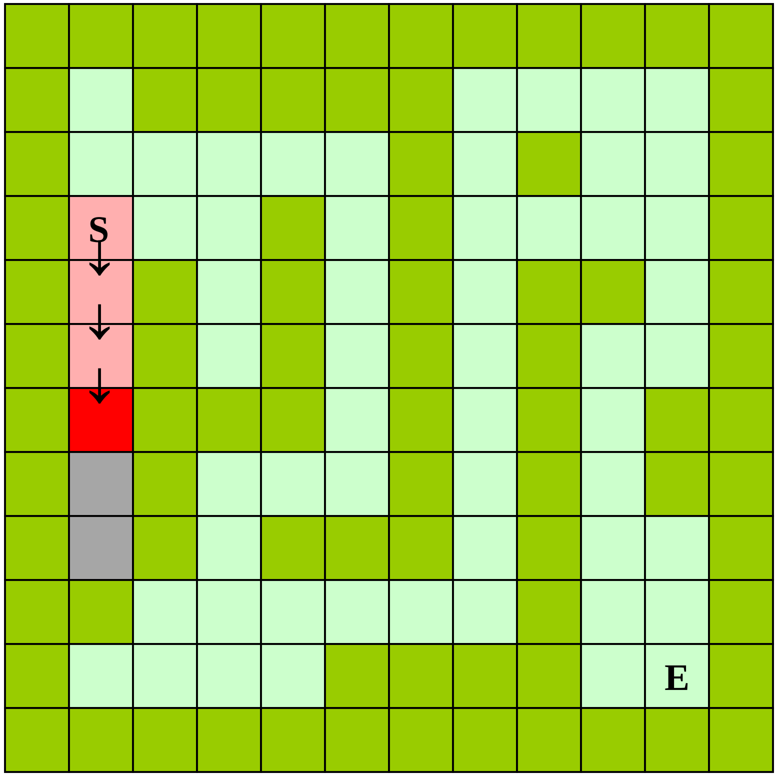
Arrows indicate the current path

Current position 🟥

Visited and pending
(on current path) 🟥

Visited and Done
(not on current path) 🟫

Searching the Example Maze



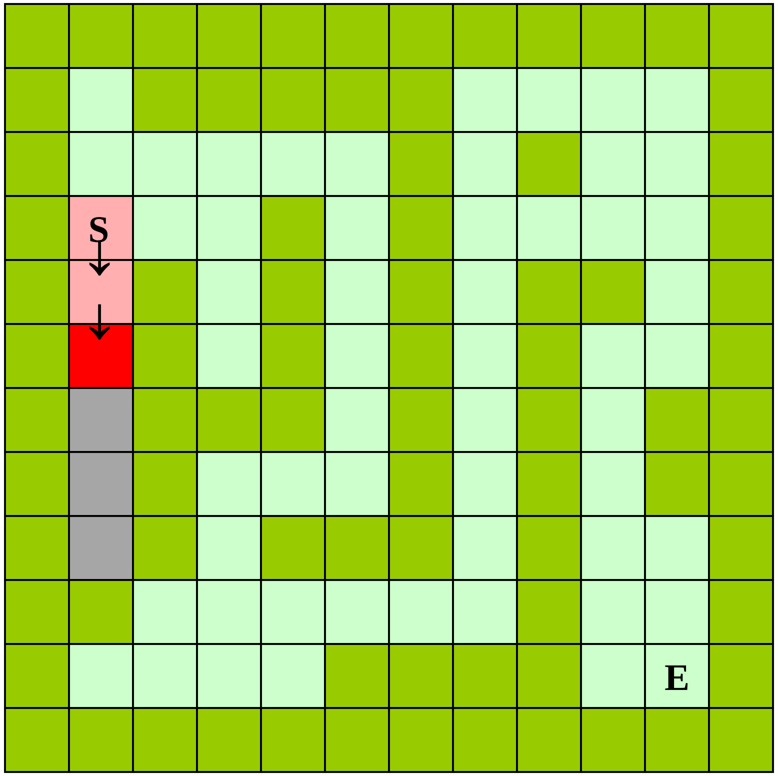
Arrows indicate the current path

Current position 🟥

Visited and pending (on current path) 🟥

Visited and Done (not on current path) 🟫

Searching the Example Maze



Arrows indicate the current path

Current position

🟥

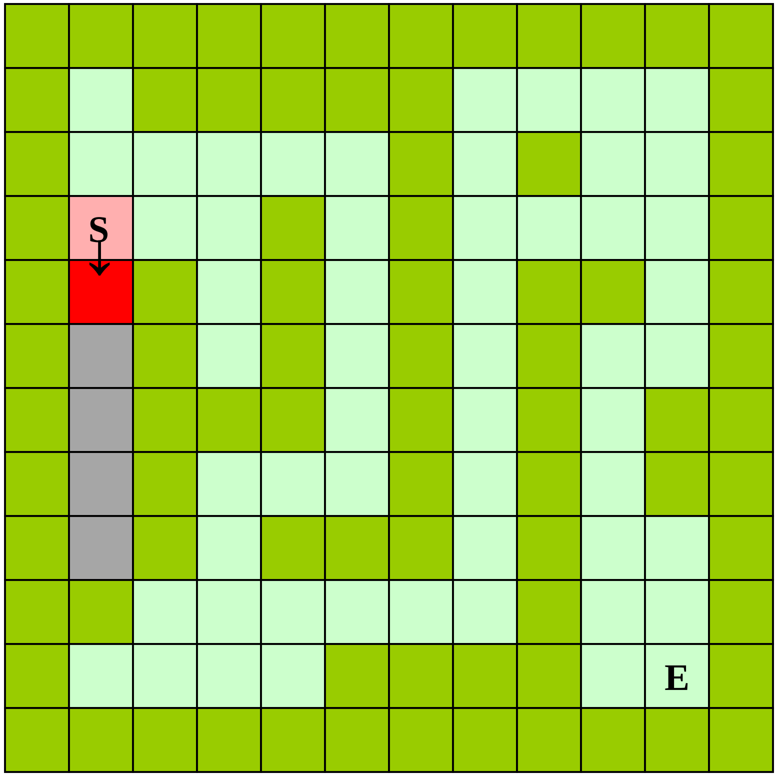
Visited and pending (on current path)

▨

Visited and Done (not on current path)

🟫

Searching the Example Maze



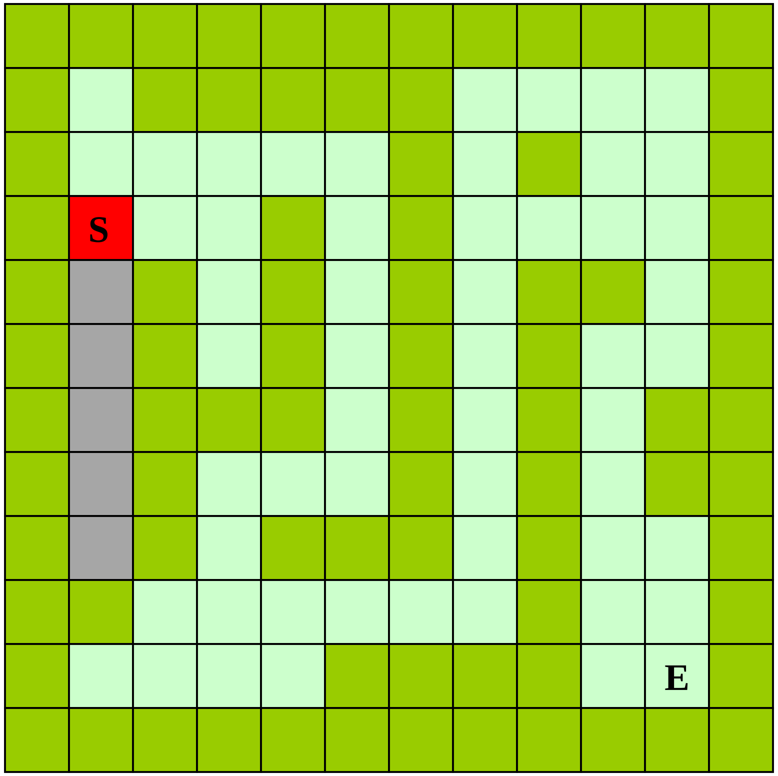
Arrows indicate the current path

Current position 🟥

Visited and pending (on current path) ■

Visited and Done (not on current path) 🟫

Searching the Example Maze



Arrows indicate the current path

Current position

🟥

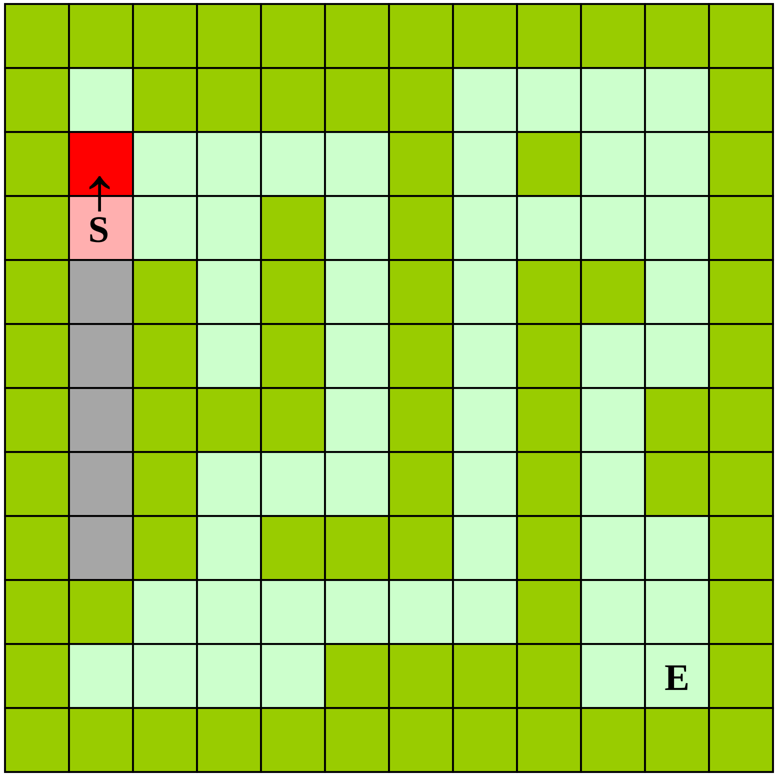
Visited and pending
(on current path)

■

Visited and Done
(not on current path)

🟫

Searching the Example Maze



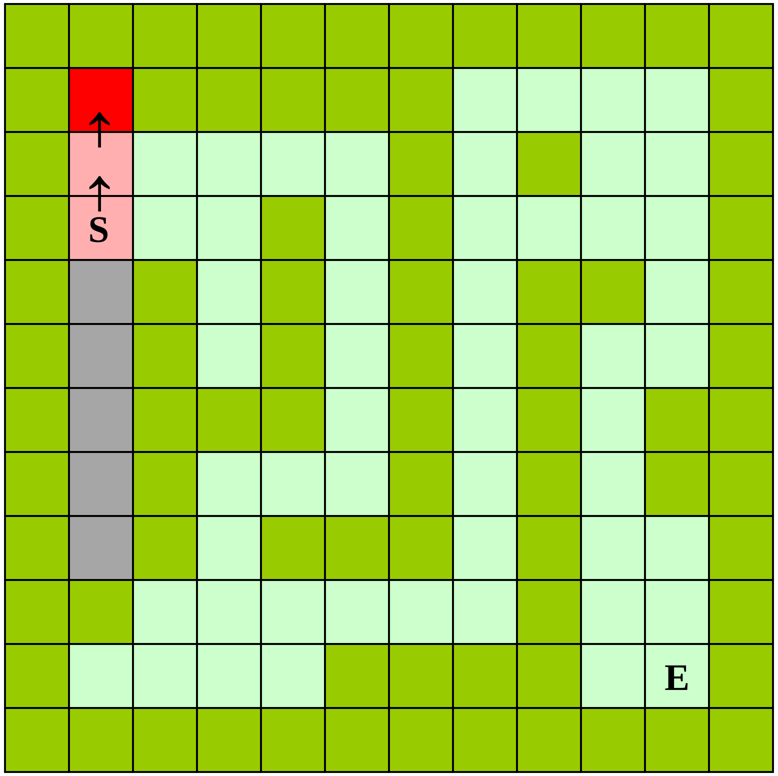
Arrows indicate the current path

Current position 🟥

Visited and pending (on current path) ▨

Visited and Done (not on current path) 🟫

Searching the Example Maze



Arrows indicate the current path

Current position

🟥

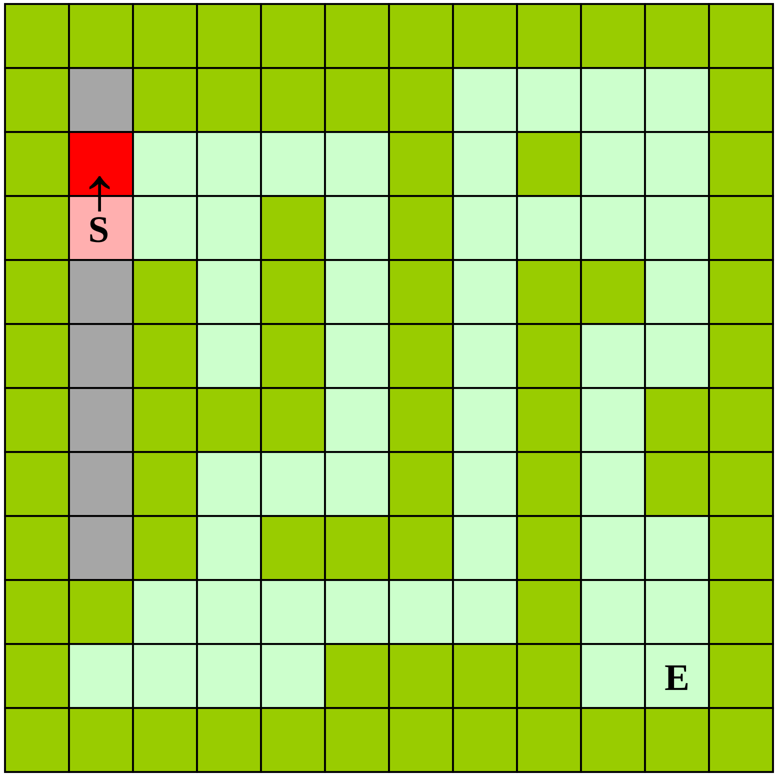
Visited and pending
(on current path)

🟪

Visited and Done
(not on current path)

🟧

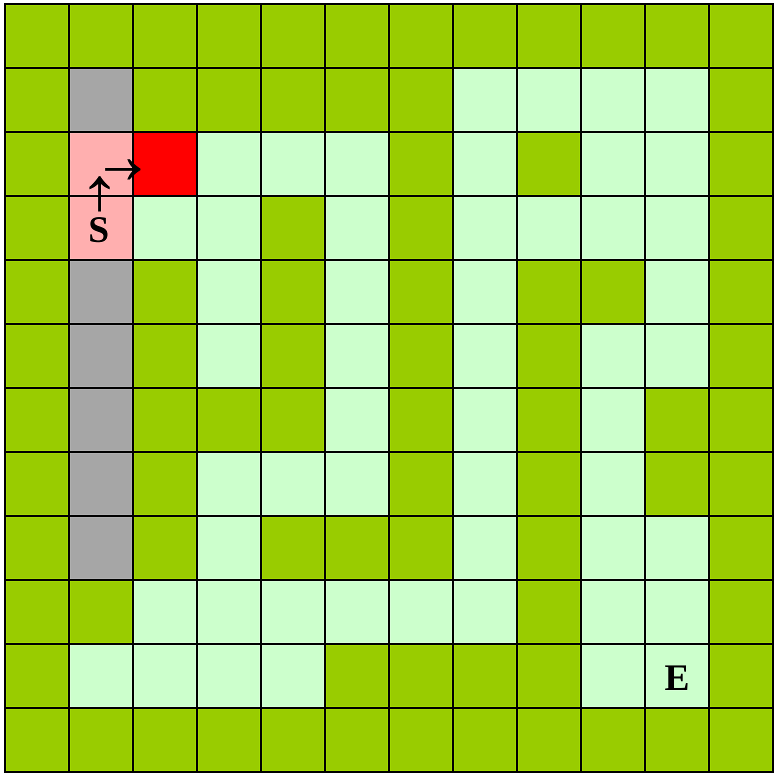
Searching the Example Maze



Arrows indicate the current path

Current position	Visited and pending (on current path)	Visited and Done (not on current path)
🟥	□	🟫

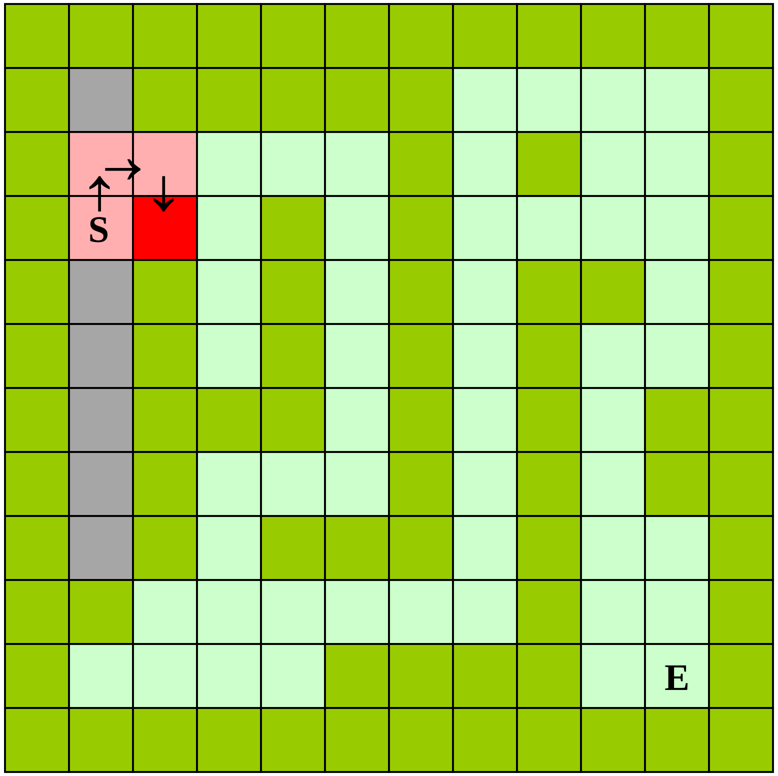
Searching the Example Maze



Arrows indicate the current path

Current position	Visited and pending (on current path)	Visited and Done (not on current path)
🟥	▨	🟫

Searching the Example Maze



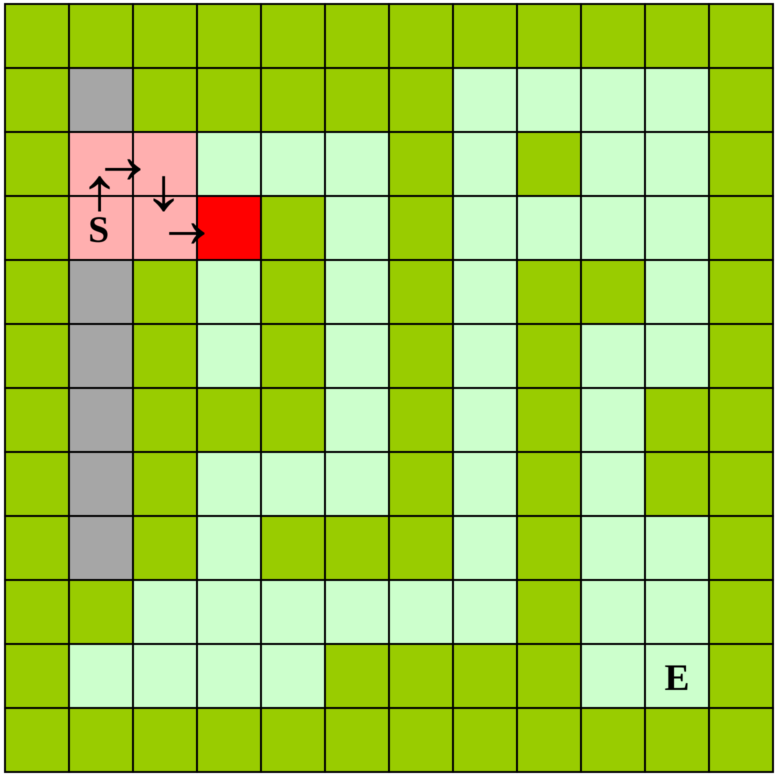
Arrows indicate the current path

Current position
🟥

Visited and pending
(on current path)
🟥

Visited and Done
(not on current path)
🟧

Searching the Example Maze



Arrows indicate the current path

Current position

🟥

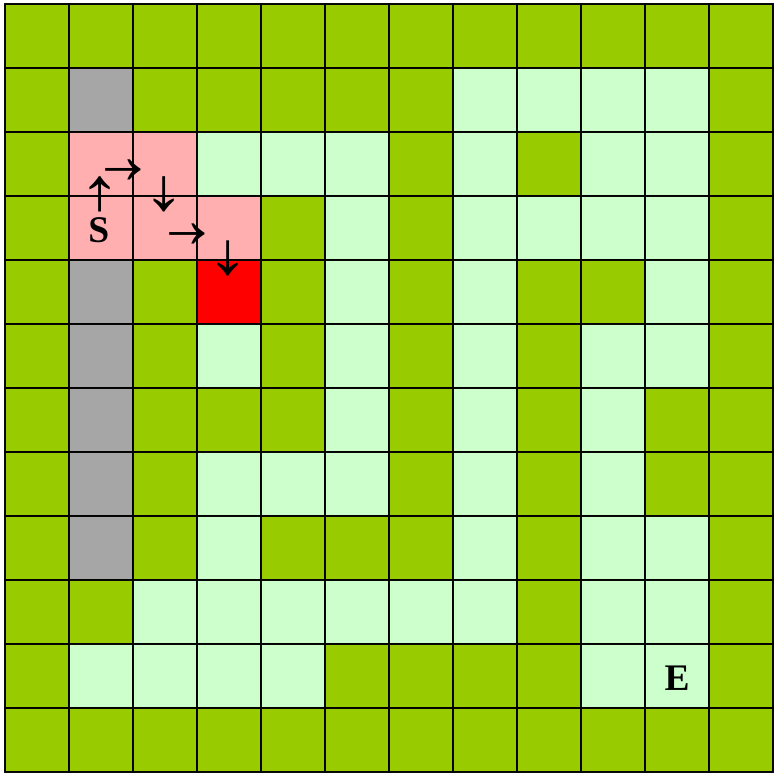
Visited and pending
(on current path)

▨

Visited and Done
(not on current path)

🟫

Searching the Example Maze



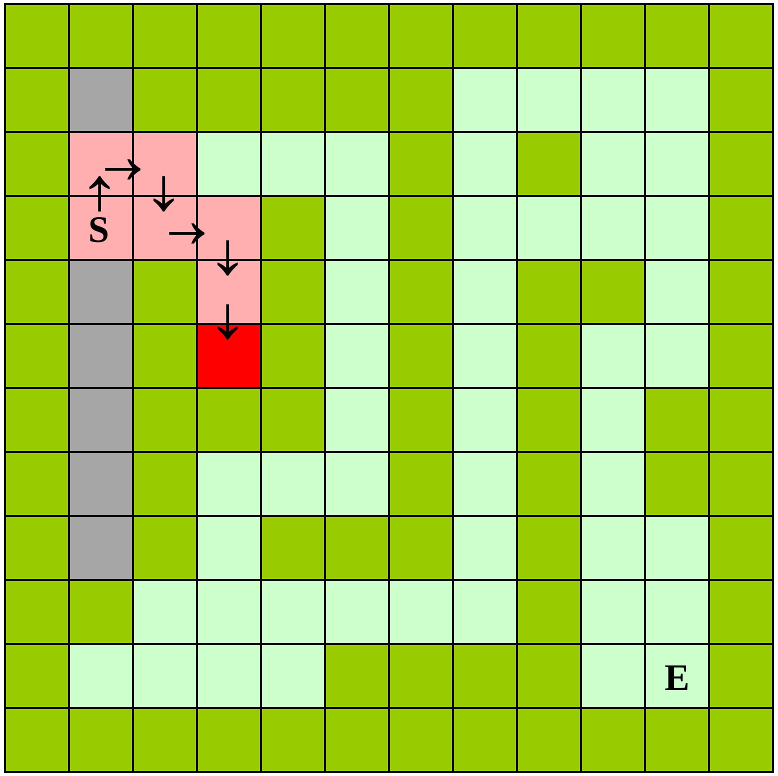
Arrows indicate the current path

Current position 🟥

Visited and pending (on current path) ▨

Visited and Done (not on current path) 🟫

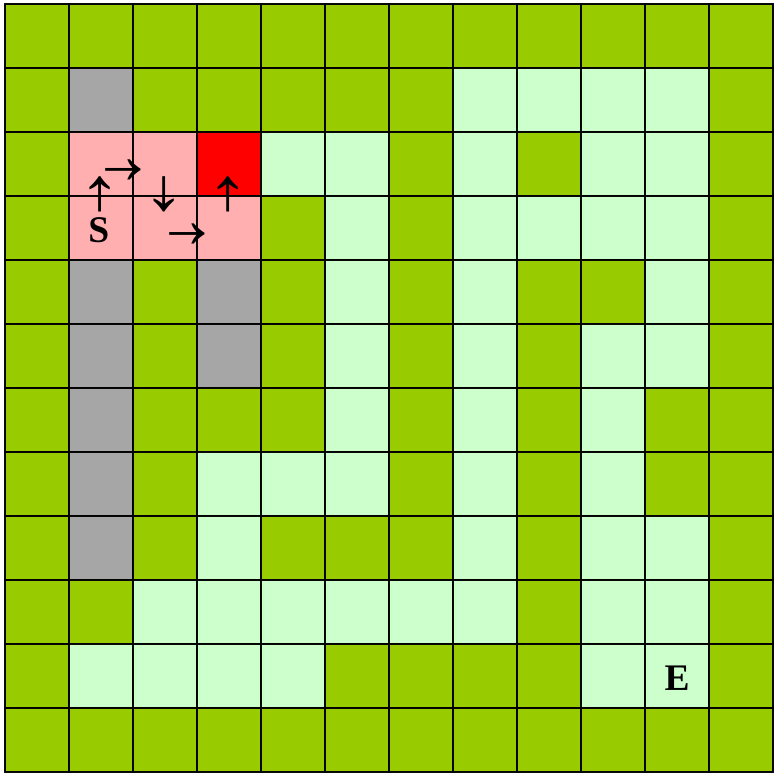
Searching the Example Maze



Arrows indicate the current path

Current position	Visited and pending (on current path)	Visited and Done (not on current path)
🟥	▨	🟫

Searching the Example Maze



Three steps combined

Arrows indicate the current path

Current position

🟥

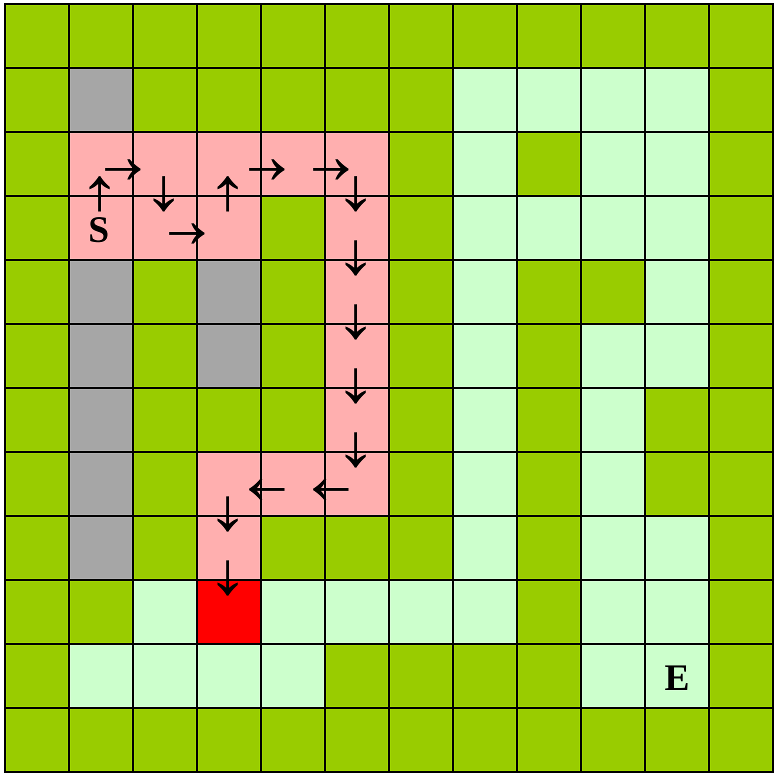
Visited and pending (on current path)

▨

Visited and Done (not on current path)

🟫

Searching the Example Maze



Several steps combined

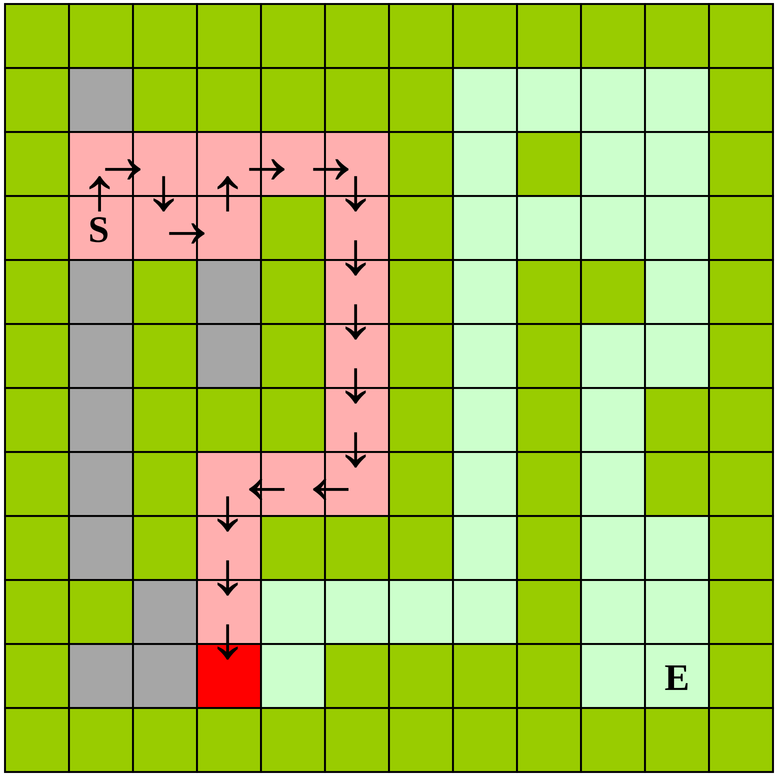
Arrows indicate the current path

Current position 🟥

Visited and pending (on current path) ◻

Visited and Done (not on current path) 🟫

Searching the Example Maze

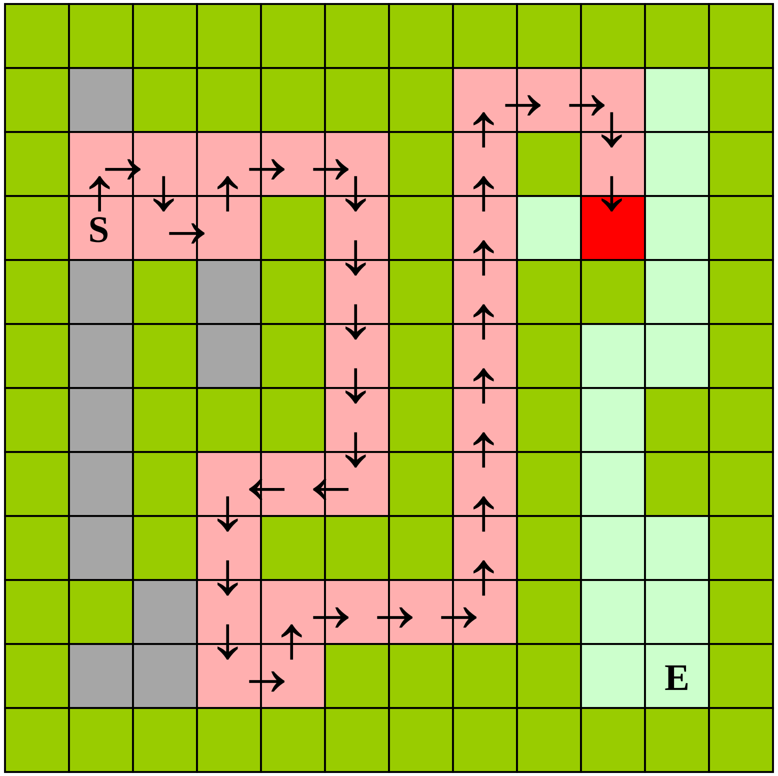


Several steps combined

Arrows indicate the current path

Current position	Visited and pending (on current path)	Visited and Done (not on current path)
🟥	▨	🟧

Searching the Example Maze



Several steps combined

Arrows indicate the current path

Current position

🟥

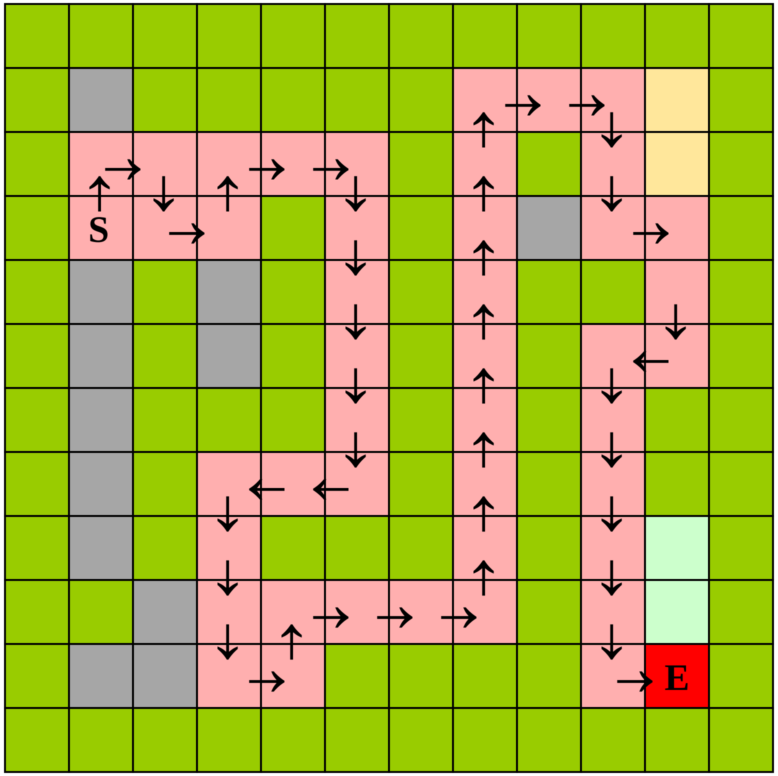
Visited and pending
(on current path)

▨

Visited and Done
(not on current path)

🟫

Searching the Example Maze



Several steps combined

Arrows indicate the current path

Current position 🟥

Visited and pending (on current path) ▨

Visited and Done (not on current path) ⬛

Application 5
Maze： Search Method

- First we mark the Start position as VISITED and make it the current position.
- When exploring from a current position P, we look for an open neighbor in a fixed pattern: east, south, west and north.
- When an OPEN neighbor Q is found, we mark it as visited.
- In this case we say that Q was entered from P and that P is “pending”: we have not finished exploring from P.
- We first check to see if Q is the Exit position.
- If so, the search concludes successfully. In this case, we print out the path from the Start to the Exit.
- If not, we continue exploring from position Q.

Application 5
Maze： Search Method

- If all four neighbors of Q are either WALL or VISITED, we return to the position P from which Q was first visited and continue exploring from P.

- That is, we “back out” from Q to P. This is the only way we can move to a previously visited position.

- Once we back out from Q, we will never return. Also, Q will not be on the path.

- If we return to the Start position and find no OPEN neighbors, the search concludes unsuccessfully: there is no path from the Start to the Exit.

Application 5

用"穷举求解"的方法解迷宫的规则：

1. 从入口进入迷宫之后，不管在迷宫的哪一个位置上，都是先往东走，如果走得通就继续往东走，如果在某个位置上往东走不通的话，就依次试探往南、往西和往北方向，从一个走得通的方向继续往前直到出口为止；

2. 如果在某个位置上四个方向都走不通的话，就退回到前一个位置，换一个方向再试，如果这个位置已经没有方向可试了就再退一步，如果所有已经走过的位置的四个方向都试探过了，一直退到起始点都没有走通，那就说明这个迷宫根本没有通道；

3. 所谓"走不通"不单是指遇到"障碍物挡路"，还有"已经走过的路不能重复走第二次"，它包括"曾经走过而没有走通的路"。显然为了保证在任何位置上都能沿原路退回，需要用一个"后进先出"的结构即栈来保存从入口到当前位置的路径。并且在走出出口之后，栈中保存的正是一条从入口到出口的路径。

动画演示1　　动画演示2

Application 5
Maze

- We now consider how to carry out the algorithm described previously.
- In particular, how do we keep track of the position from which the current position was entered?
- Consider this: Suppose we go from S to P, then P to Q, then Q to R and R to T, then at position T find no OPEN neighbors.
- Then we back out to R. Suppose R has no OPEN neighbors. Then we should back out to Q.
- If Q has no OPEN neighbors, we back out to P and then to S.
- So the Path progresses like this:
- S
- S→P
- S →P →Q
- S →P→Q →R
- S →P→Q →R →T
- S →P→Q →R
- S →P→Q
- S →P
- S

So we will maintain a stack of Positions.
When we first visit a Position, we push it on the stack.
To back out from a Position, we pop the top of the stack and continue exploring from there.

#	#	#	#	#	#	#	#	#	#
#	→	↓	#	$	$	$	#		#
#		↓	#	$	$	$	#		#
#		←	$	$	#	#			#
#		#	#	#				#	#
#				#				#	#
#		#				#			#
#	#	#	#	#		#	#		#
#								Θ	#
#	#	#	#	#	#	#	#	#	#

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

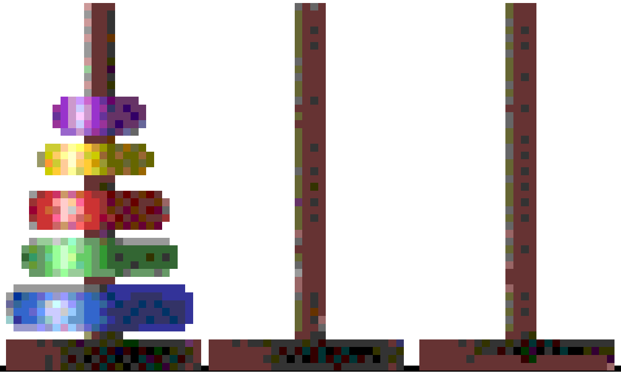
Application 6
Towers of Hanoi

- A Legend ： The Towers of Hanoi

In the great temple of Brahma in Benares, on a brass plate under the dome that marks the center of the world, there are 64 disks of pure gold that the priests carry one at a time between these diamond needles according to Brahma's immutable law: No disk may be placed on a smaller disk. In the begging of the world all 64 disks formed the Tower of Brahma on one needle. Now, however, the process of transfer of the tower from one needle to another is in mid course. When the last disk is finally in place, once again forming the Tower of Brahma but on a different needle, then will come the end of the world and all will turn to dust.

Application 6
The Towers of Hanoi

- GIVEN: three poles
- a set of discs on the first pole, discs of different sizes, the smallest discs at the top
- GOAL: move all the discs from the left pole to the right one.
- CONDITIONS: only one disc may be moved at a time.
- A disc can be placed either on an empty pole or on top of a larger disc.



假设

1、如果①号杆上只有1个盘子，把盘子移到③号杆，只需要移动几次?

2、如果①号杆上有2个盘子，把盘子移到③号杆，最少移动几次?怎样移?

移动规则如下：

（1）每次只能移动一个盘子;

（2）大盘子不能放到小盘子上面。

“河内塔问题”

有①号、②号、③号三根杆子，你能借助②号杆把①号杆上的3个盘子移到③号杆而不改变盘子的上下顺序吗？最少移动多少次？

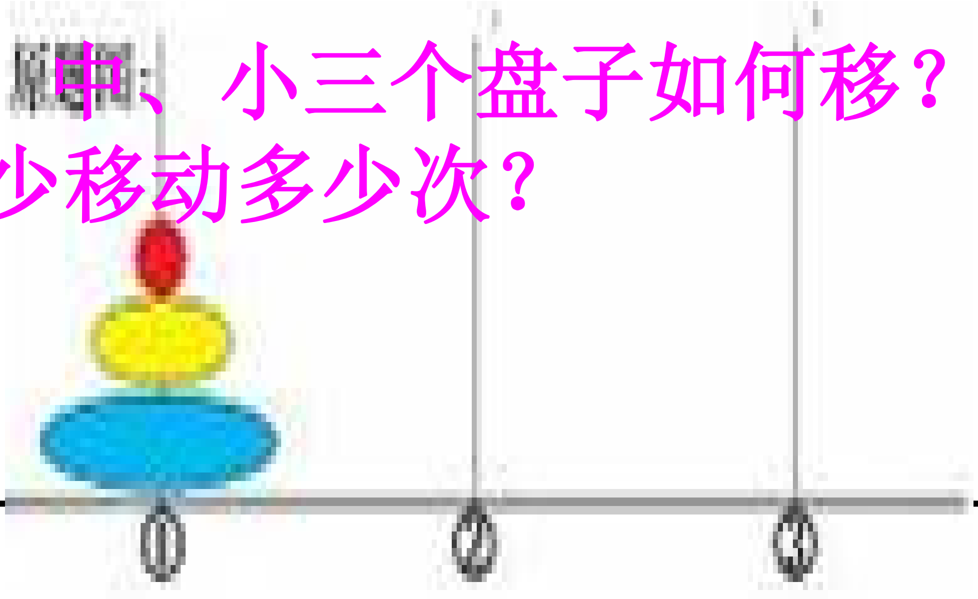
移动规则如下：

（1）每次只能移动一个盘子；

（2）大盘子不能放到小盘子上面。

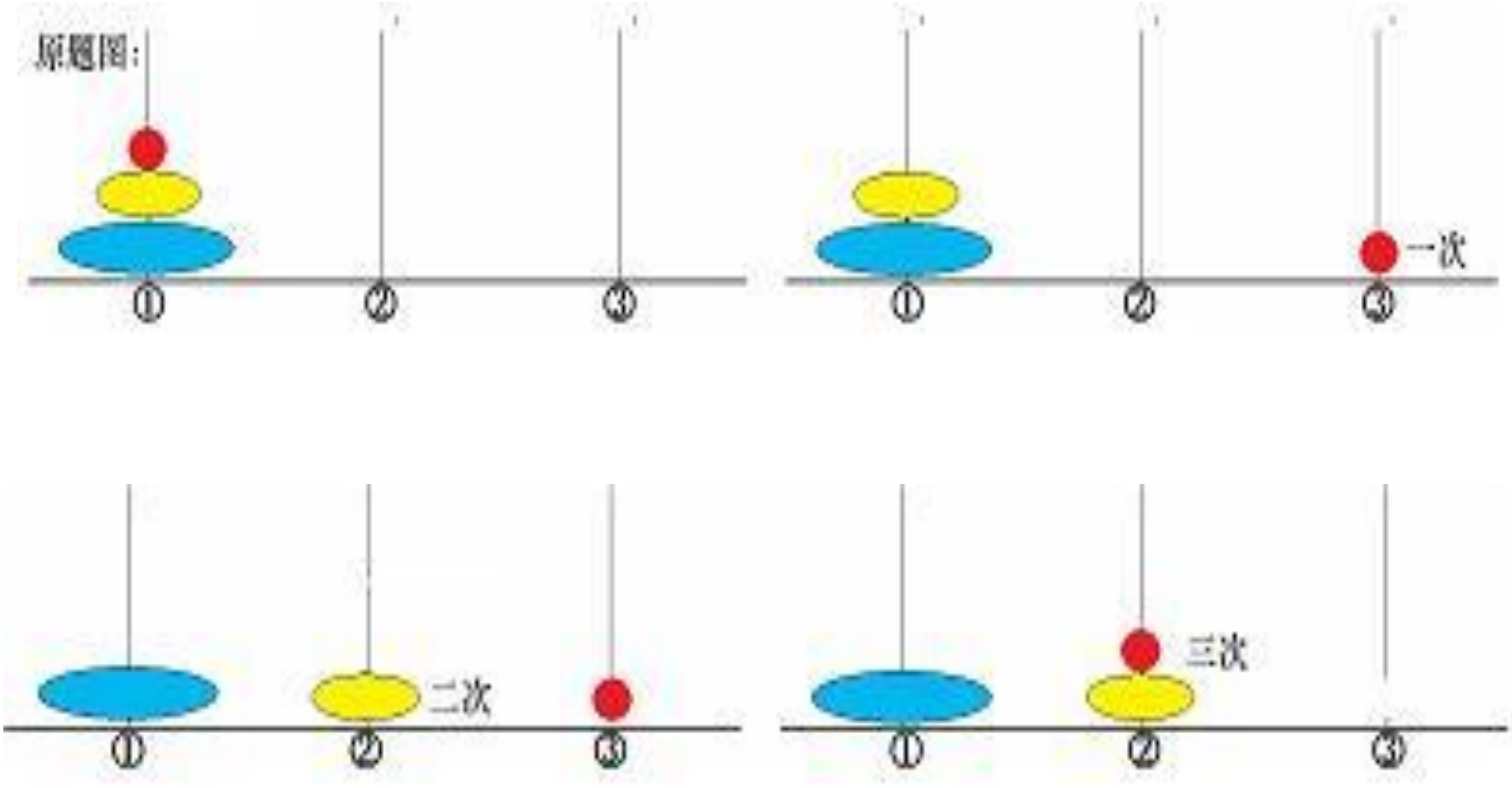
讨论：大、中、小三个盘子如何移？

最少移动多少次？

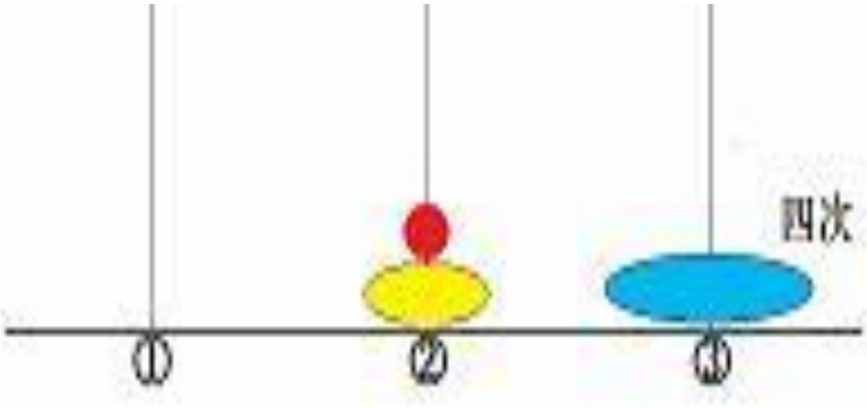


三个盘子的移动图解：三个盘子的移动只有两种移动方法：如果第一次移动时，把最小红盘子放到③号杆上是优选法。如下：

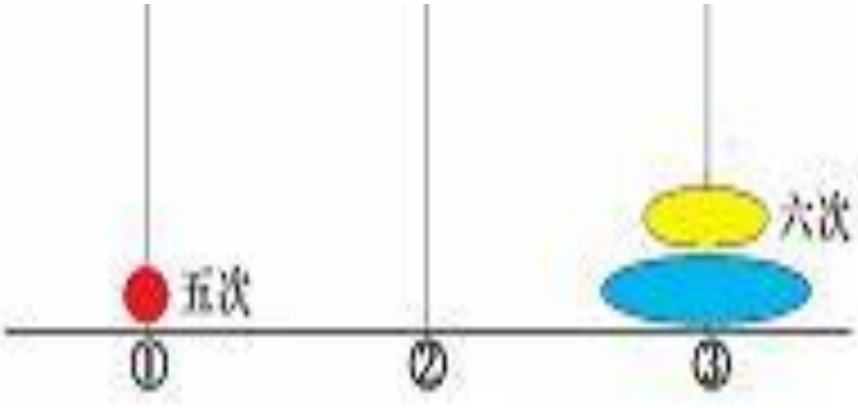
- （一）原题图：　　　　（二）移动第一次：



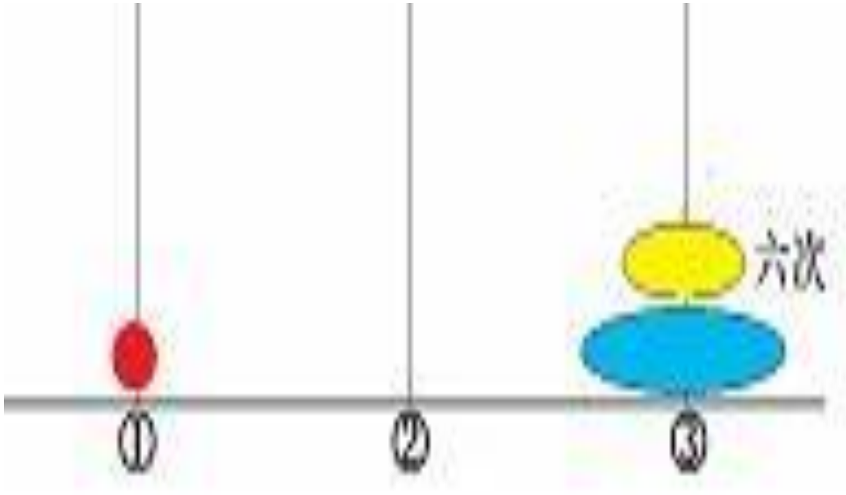
（五）移动第四次：



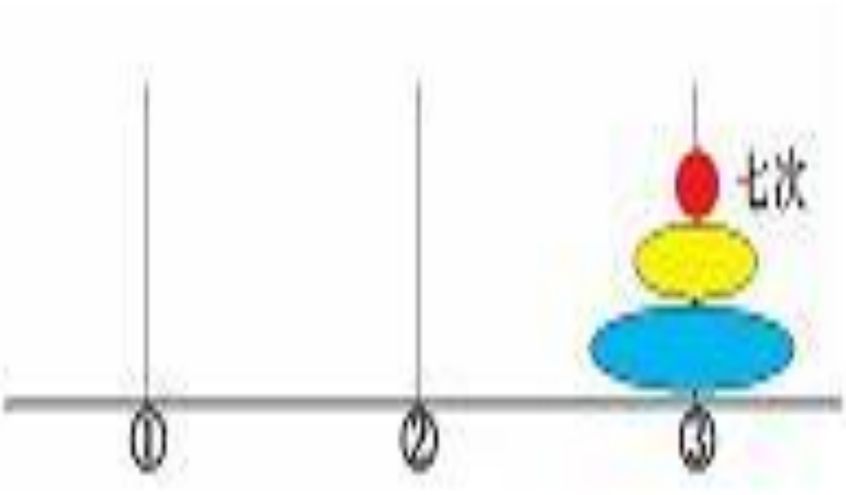
（六）移动第五次：



（七）移动第六次：



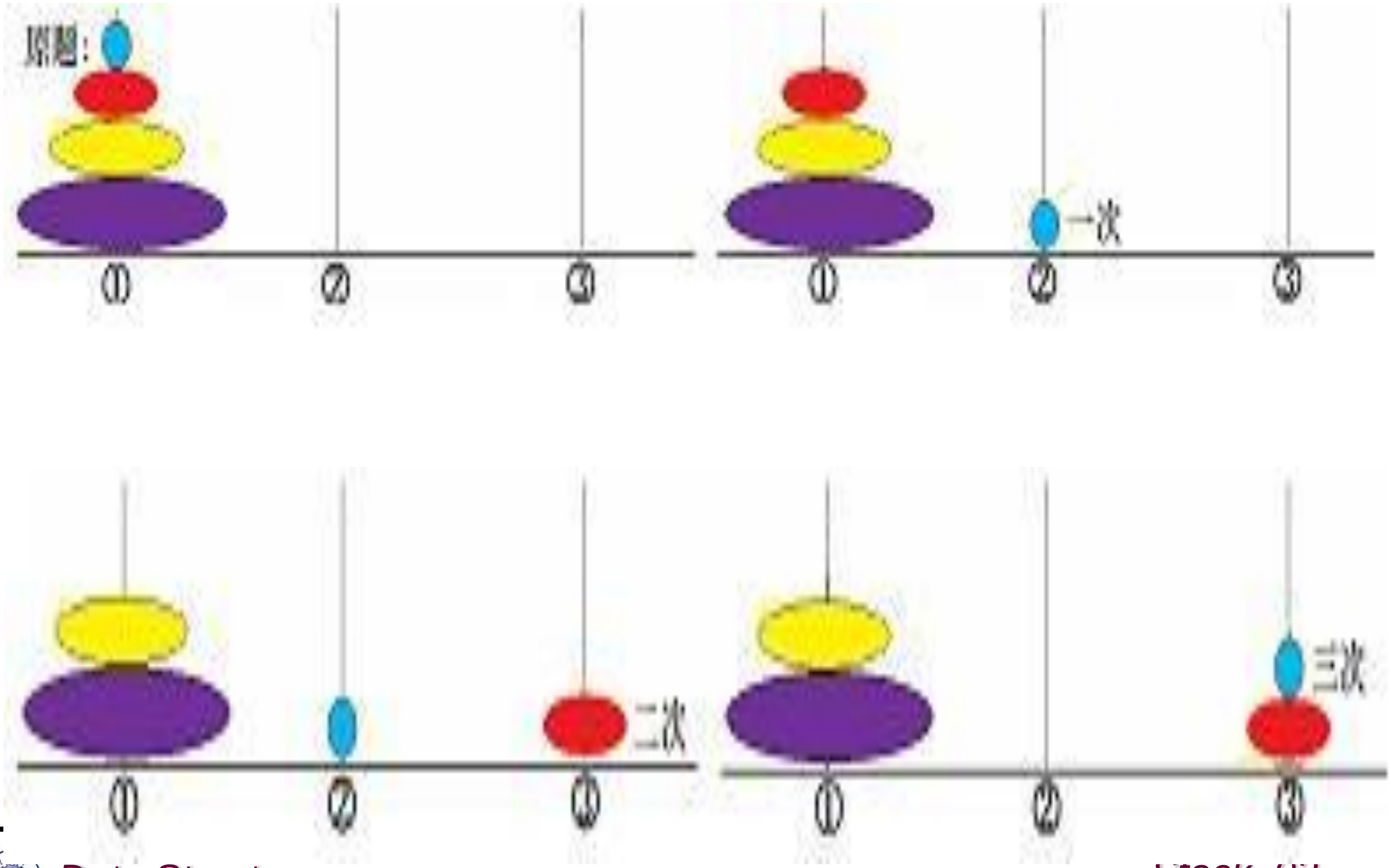
（八）移动第七次：



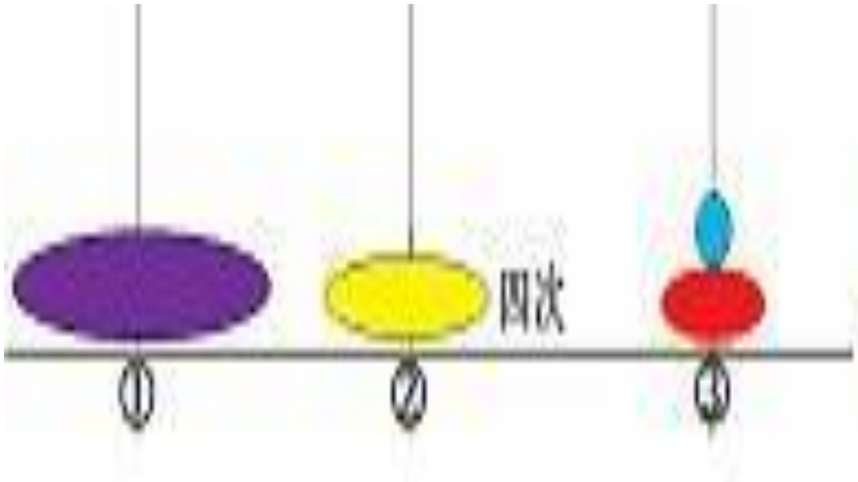
四个盘子的移动图解：

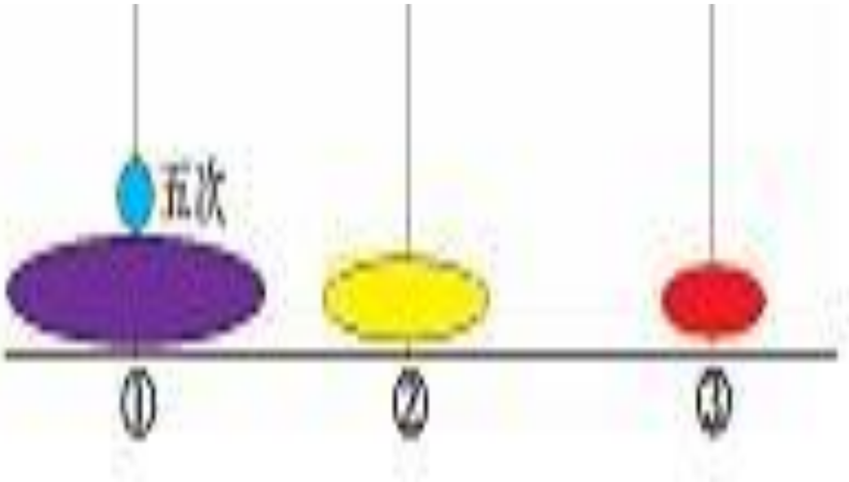
四个盘子：开始第一个盘子要放在②号杆上：

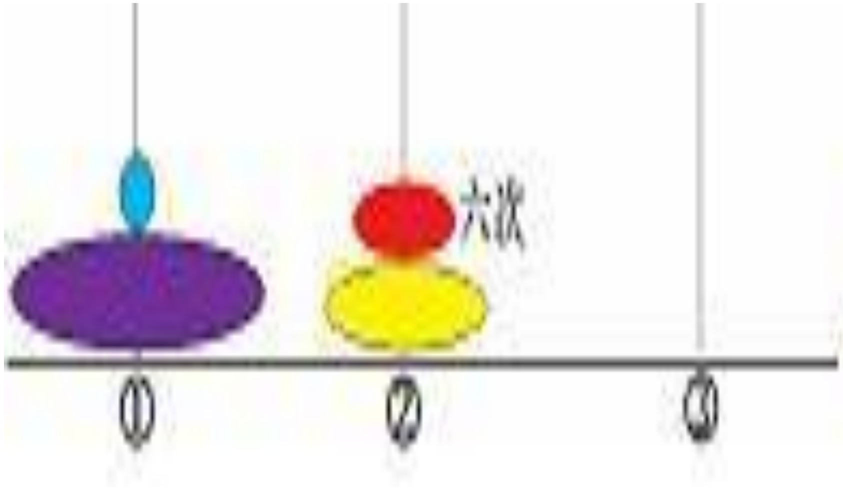
- （一）原题图：　　　　　　（二）第一次移动：

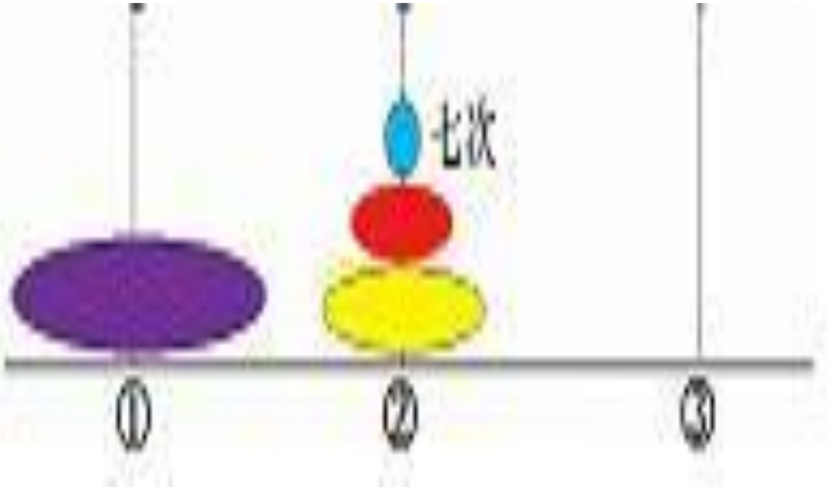


- （五）第四次移动：   （六）第五次移动：

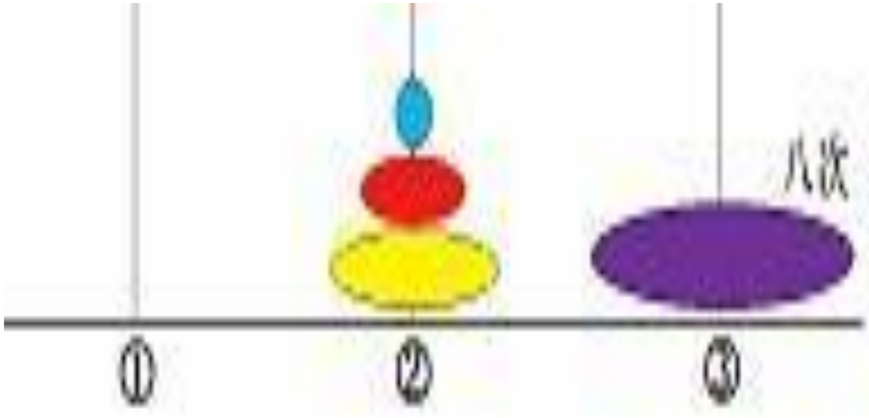




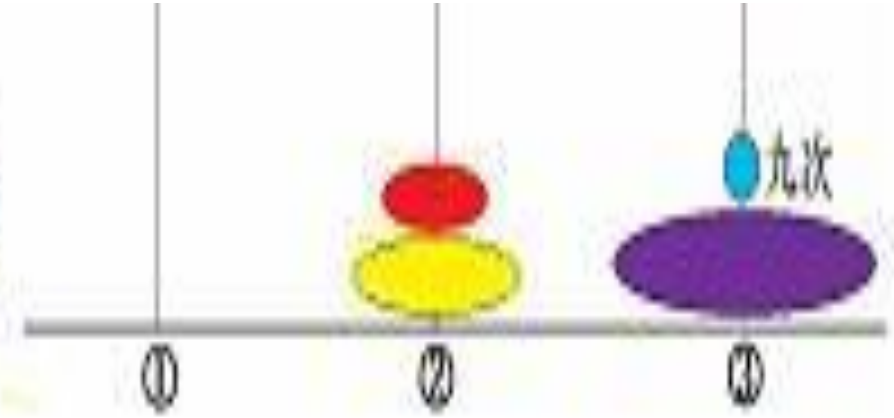


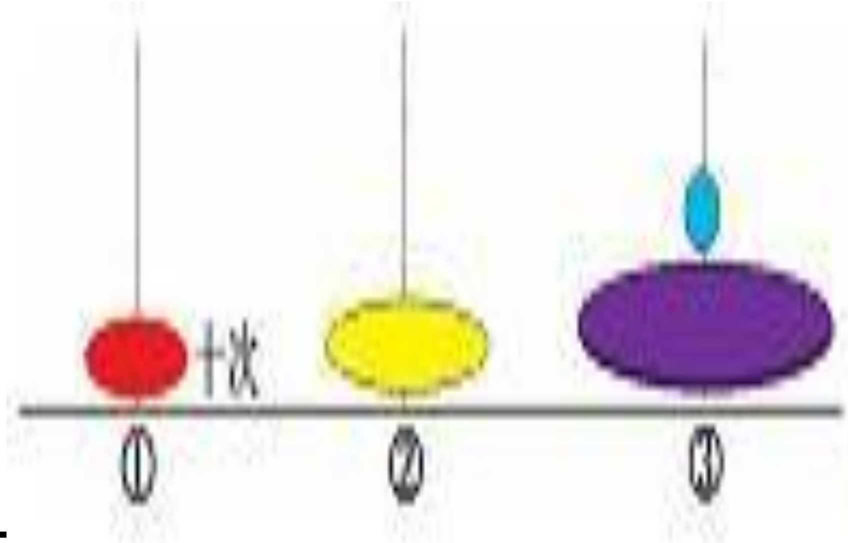


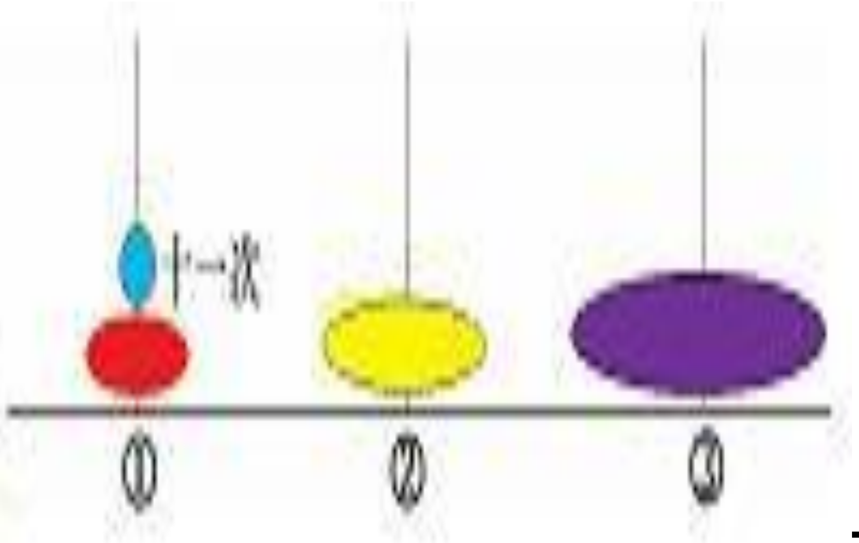
- （九）第八次移动：



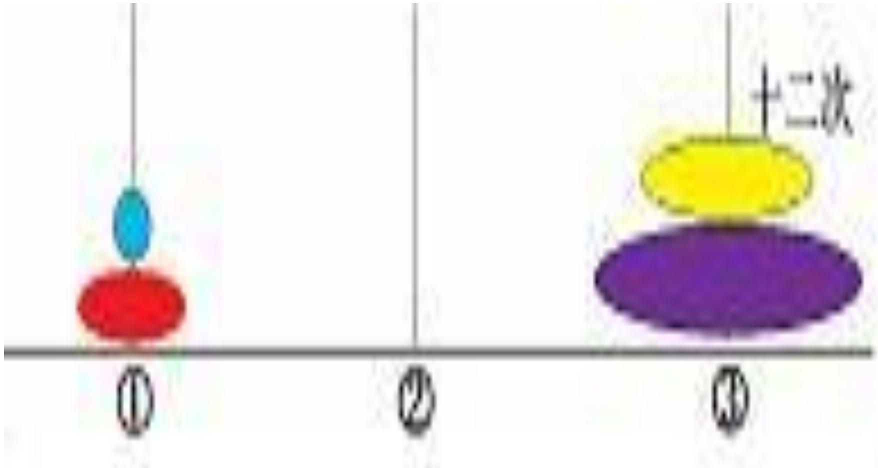
（十）第九次移动：



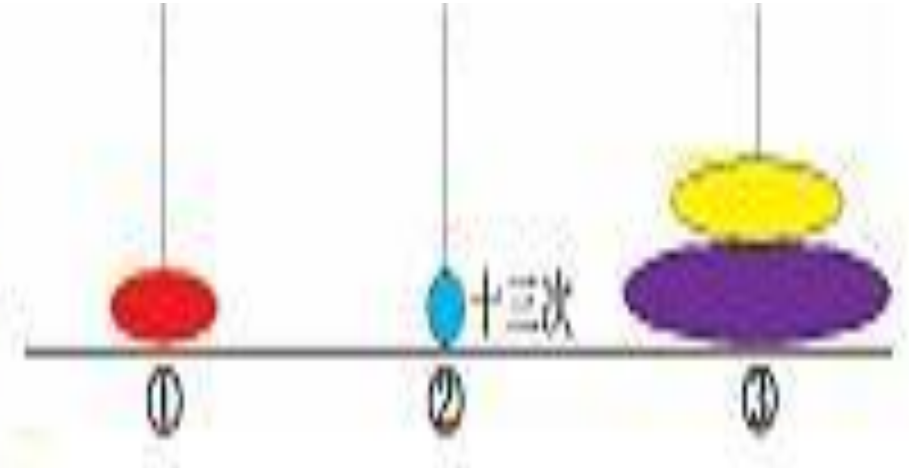




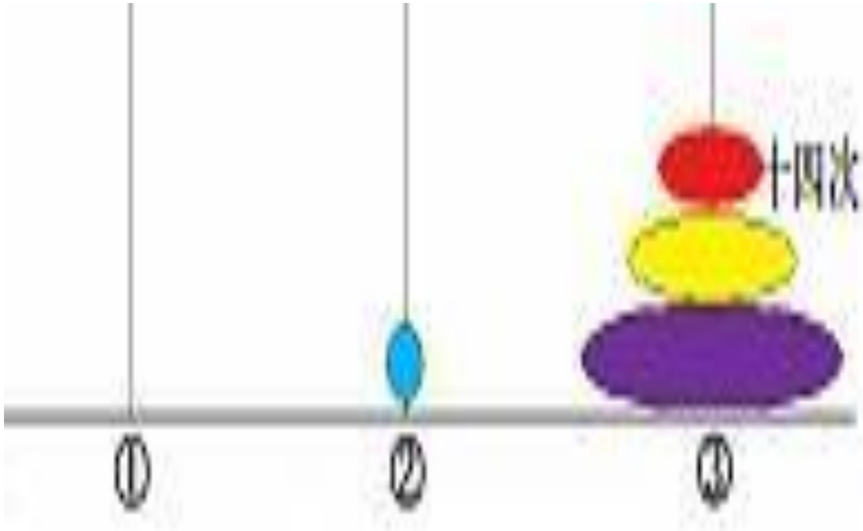
（十三）第十二次移动



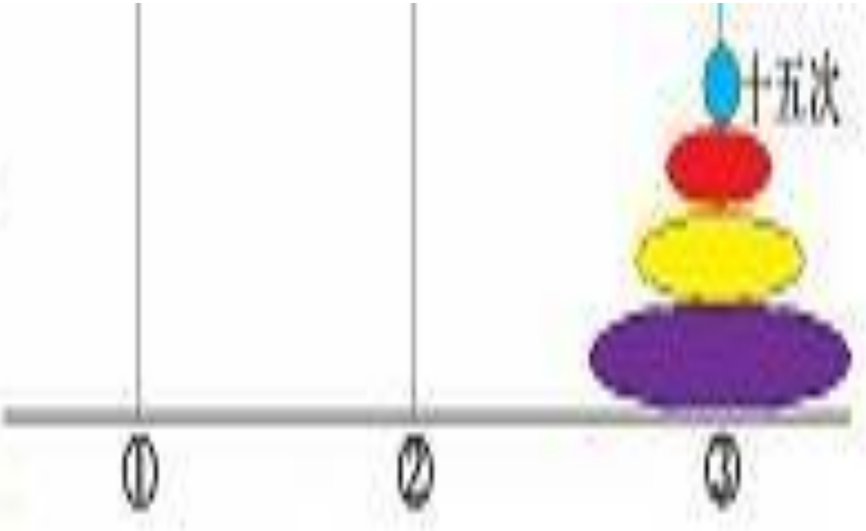
（十四）第十三次移动：



（十五）第十四次移动



（十六）第十五次移动：



Application 6
The Towers of Hanoi

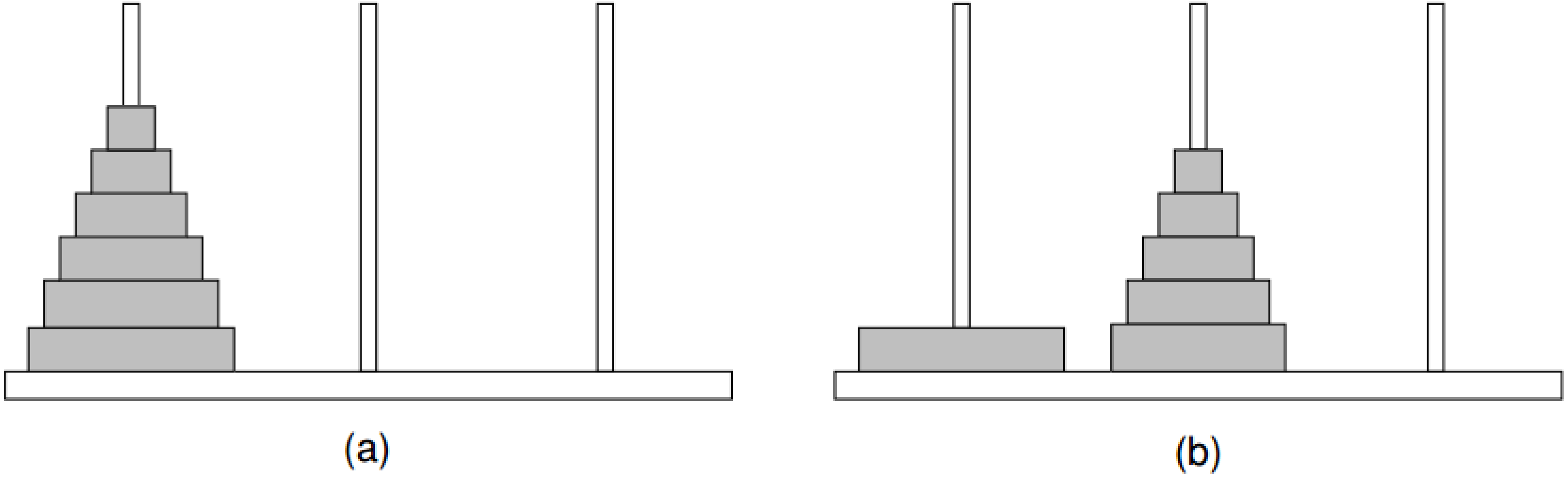
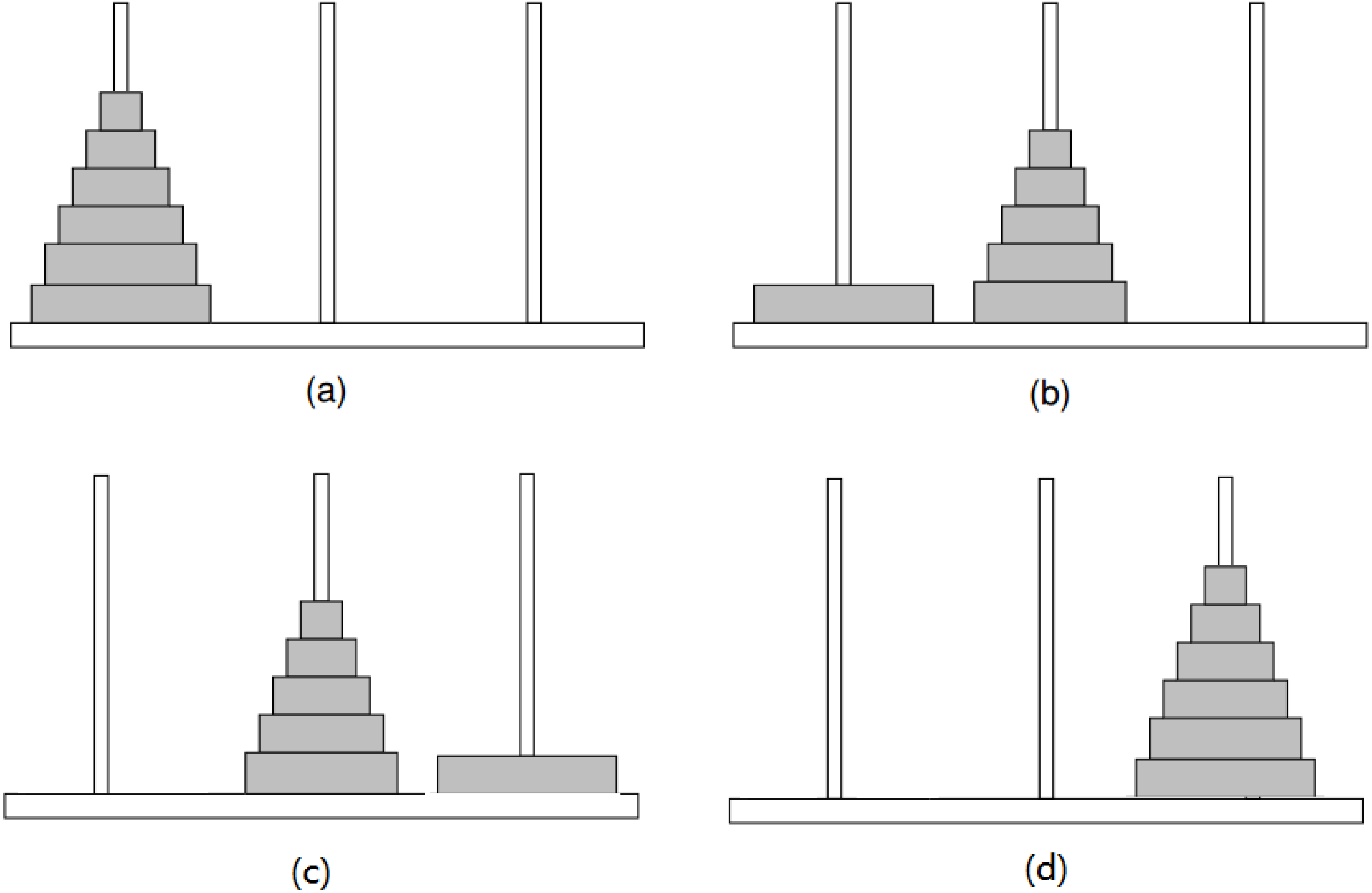
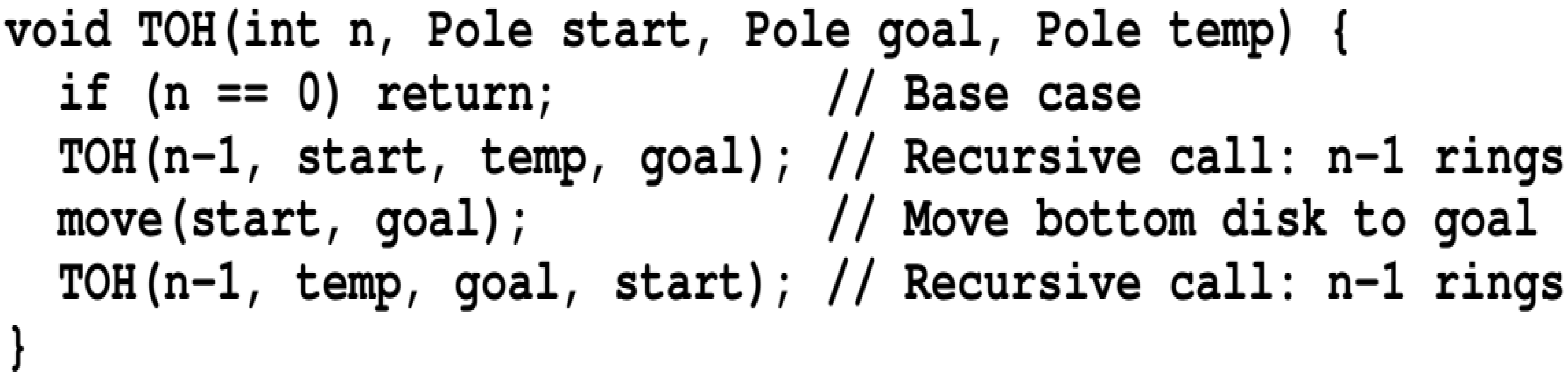


Figure 2.2 Towers of Hanoi example. (a) The initial conditions for a problem with six rings. (b) A necessary intermediate step on the road to a solution.


(a)
(b)
(c)
(d)

Application 6

Towers of Hanoi – Recursive Solution



```
void TOH(int n, Pole start, Pole goal, Pole temp) {
  if (n == 0) return;             // Base case
  TOH(n-1, start, temp, goal); // Recursive call: n-1 rings
  move(start, goal);              // Move bottom disk to goal
  TOH(n-1, temp, goal, start); // Recursive call: n-1 rings
}
```

Application 6

Towers of Hanoi – Recursive Solution

Example 4.3 The **TOH** function shown in Figure 2.2 makes two recursive calls: one to move *n* – 1 rings off the bottom ring, and another to move these *n* – 1 rings back to the goal pole. We can eliminate the recursion by using a stack to store a representation of the three operations that **TOH** must perform: two recursive calls and a move operation. To do so, we must first come up with a representation of the various operations, implemented as a class whose objects will be stored on the stack.

Application 6
Towers of Hanoi – Recursive Solution

```
// Operation choices: DOMOVE will move a disk
// DOTOH corresponds to a recursive call
enum TOHop { DOMOVE, DOTOH };
class TOHobj {  // An operation object
public:
  TOHop op;    // This operation type
  int num;       // How many disks
  Pole start, goal, tmp;     // Define pole order
  // DOTOH operation constructor
  TOHobj(int n, Pole s, Pole g, Pole t) {
      op = DOTOH; num = n;
      start = s; goal = g; tmp = t;
  }
```

Application 6

Towers of Hanoi – Recursive Solution

```
// DOMOVE operation constructor
TOHobj(Pole s, Pole g)
 { op = DOMOVE; start = s; goal = g; }
};
 void TOH(int n, Pole start, Pole goal, Pole tmp,
 Stack<TOHobj*>& S) {
   S.push(new TOHobj(n, start, goal, tmp)); // Initial
   TOHobj* t;
   while (S.length() > 0) {    // Grab next task
      t = S.pop();
      if (t->op == DOMOVE)   // Do a move
          move(t->start, t->goal);
      else if (t->num > 0) {
           // Store (in reverse) 3 recursive statements
```

Application 6

Towers of Hanoi – Recursive Solution

```
    int num = t->num;
    Pole tmp = t->tmp; Pole goal = t->goal;
    Pole start = t->start;
    S.push(new TOHobj(num-1, tmp, goal, start));
    S.push(new TOHobj(start, goal));
    S.push(new TOHobj(num-1, start, tmp, goal));
}
    delete t; // Must delete the TOHobj we made
}
}
```

Is the End of the World Approaching?

- Problem complexity 2^n
- 64 gold discs
- Given 1 move a second

64个金环，众僧们要移动

$$\underbrace{2\times2\times2\times\ldots\times2}_{\text{64个2}}-1$$

＝1844 6744 0737 0951 1615（次）

读作：一千八百四十四京
六千七百四十四兆
零七百三十七亿
零九百五十一万
一千六百一十五

数级：
个级
万级
亿级
兆级
京级
垓级
⋮

假如僧侣们每秒钟移动一次金片，夜以继日废寝忘食地照这样干下去，需要干多少年？

- 一年有多少秒？
- （60×60×24×365）秒
- 需要多少年？
- **1844 6744 0737 0951 1615÷(60×60×24×365) ≈5846**亿年

→ About 5846亿 years until the end of the world ☺

太阳的寿命最多还有100～150亿年

Exercise

- 简单背包问题(Knapstack Problem):
- 有一个背包，能盛放的物品总体积为T，设有N件物品，其体积分别为w1,w2,…,wn.希望从N件物品中选择若干件物品，所选物品的体积之和恰能放入该背包，即所选物品的体积之和等于T。
- 例如：当T=10，各件物品的体积{1，8，4，3，5，2}时，可找到下列4组解：（1，4，3，2）
  （1，4，5）
  （8，2）
  （3，5，2）。
- 请写出递归算法和非递归算法。

Summary

- ADT stack operations have a last-in, first-out (LIFO) behavior
- Stack applications
- A strong relationship exists between recursion and stacks

Reading Materials

- 《数据结构（ C 语言版）》，严蔚敏，吴伟民编著，清华大学出版社，1997年第1版,P50-58

End

Thank you for listening!