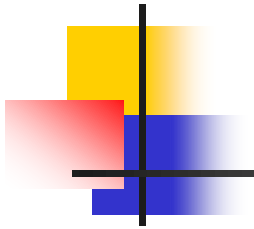
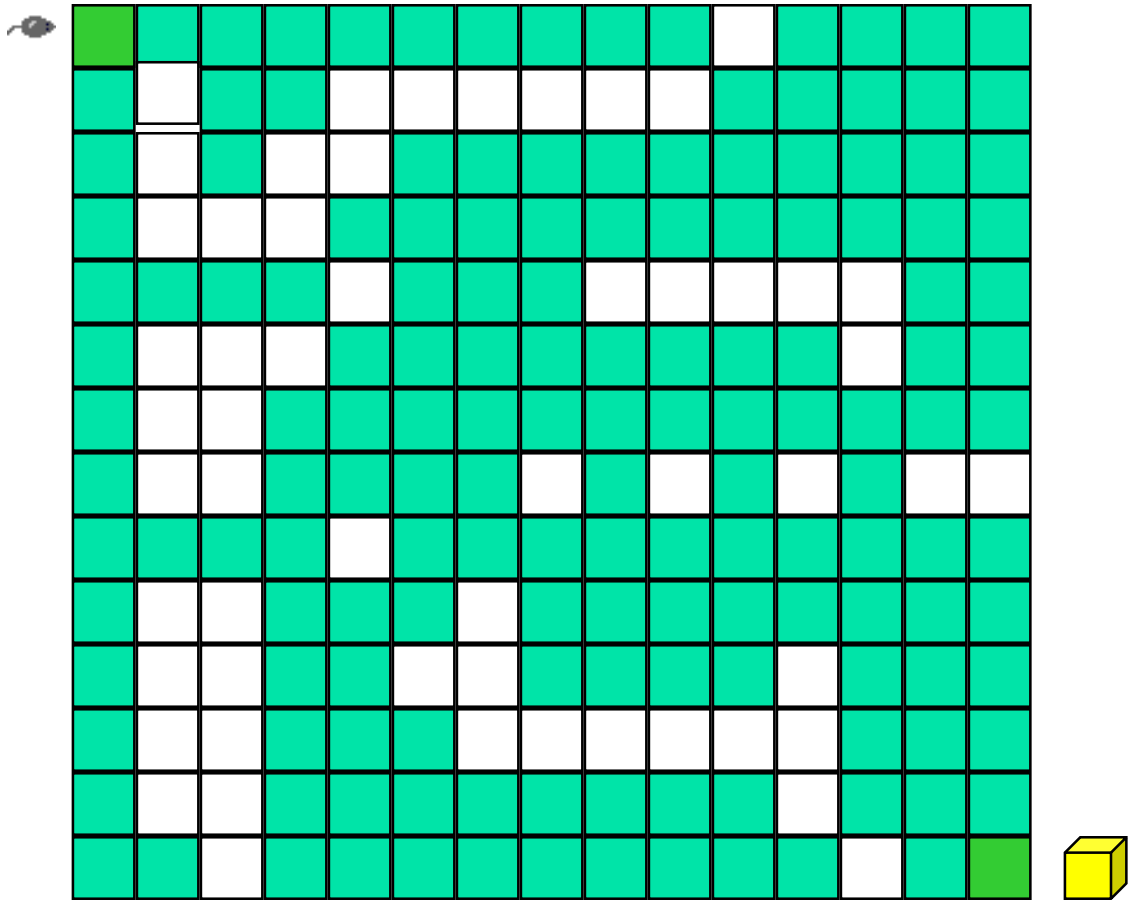
递归与回溯

- 对一个包含有许多结点，且每个结点有多个分支的问题，可先选择一个分支进行搜索。当搜索到某一结点，发现无法再继续搜索下去时，可以沿搜索路径回退到前一结点，沿另一分支继续搜索
- 如果回退之后没有其他选择，再沿搜索路径回退到更前结点，…。依次执行，直到搜索到问题的解，或搜索完全部可搜索的分支没有解存在为止
- 回溯法与分治法本质相同，可用递归求解

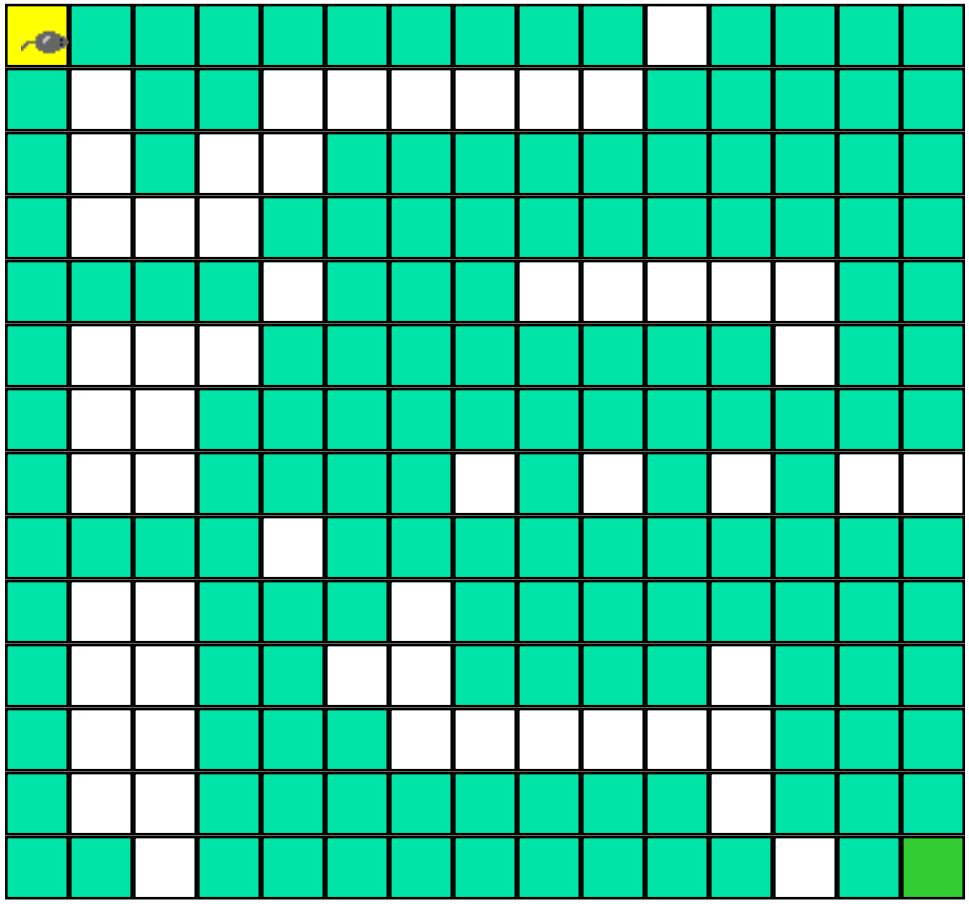


栈的应用：递归->迷宫问题

Rat In A Maze

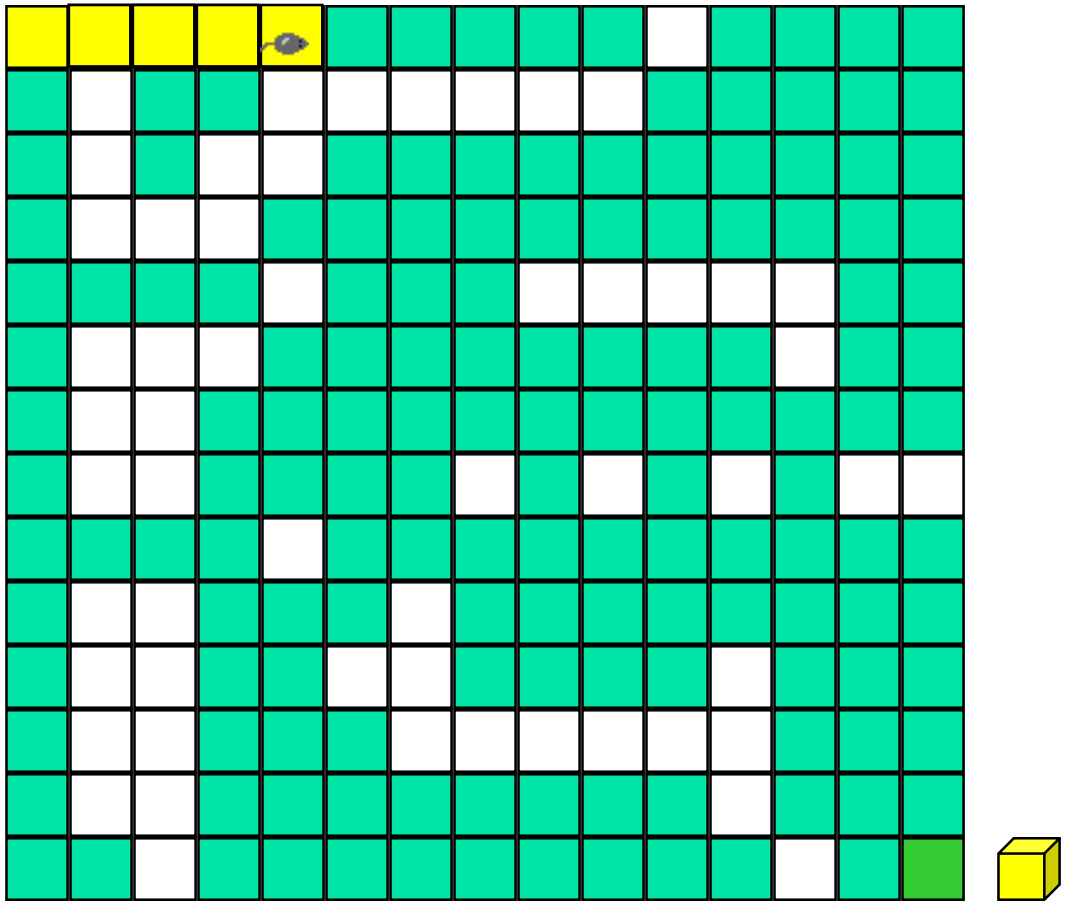


Rat In A Maze

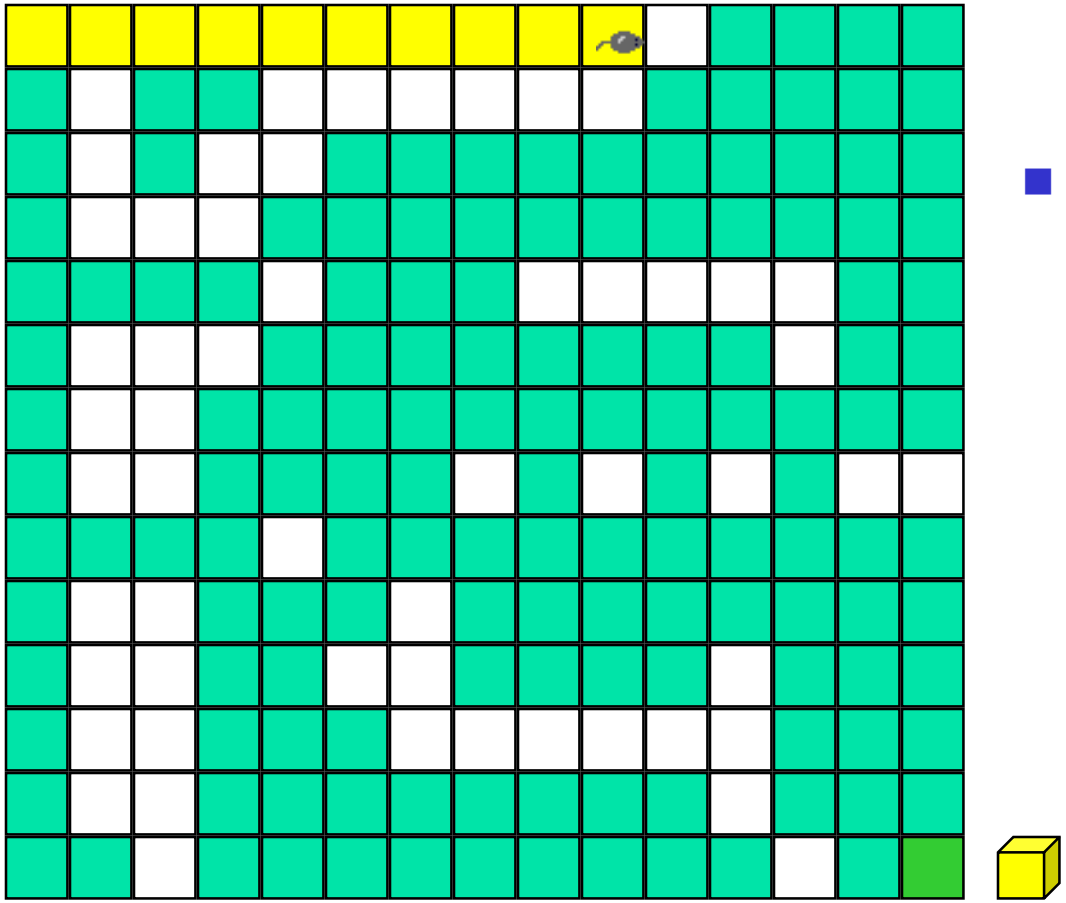


- Move order is: right, down, left, up
- Block positions to avoid revisit.

Rat In A Maze

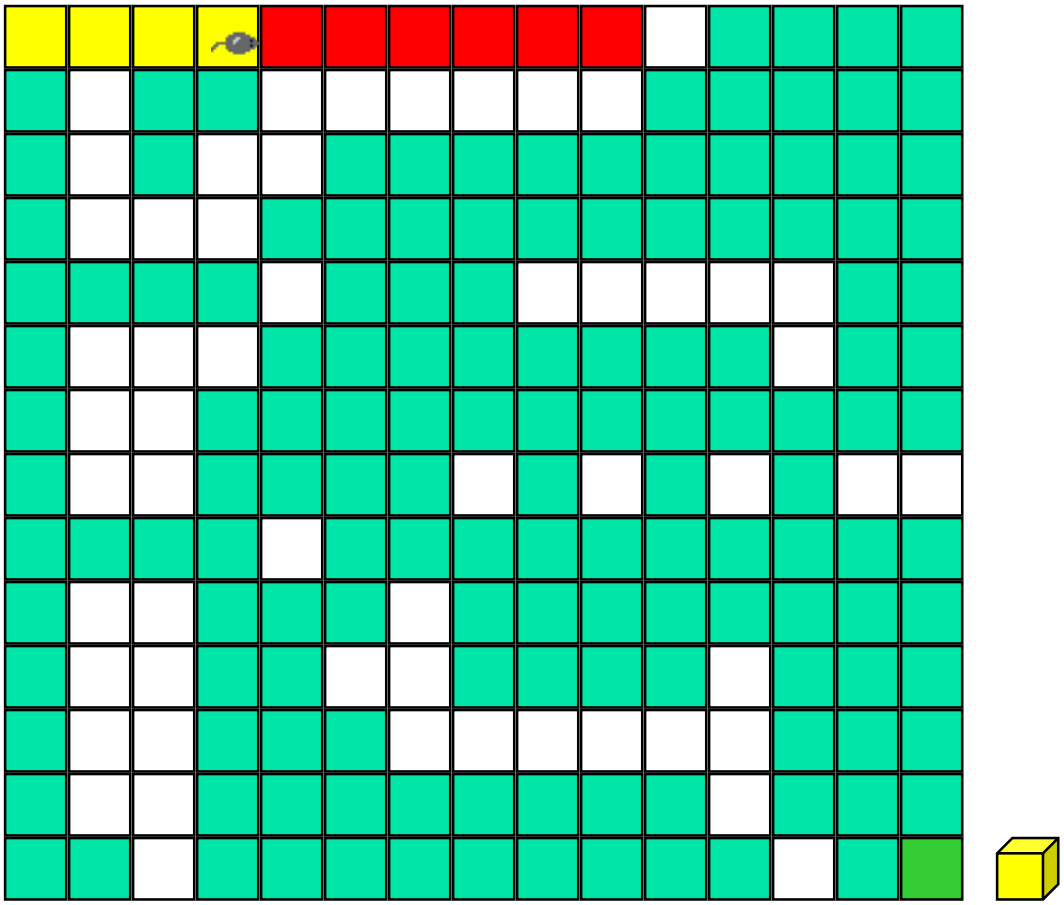


Rat In A Maze



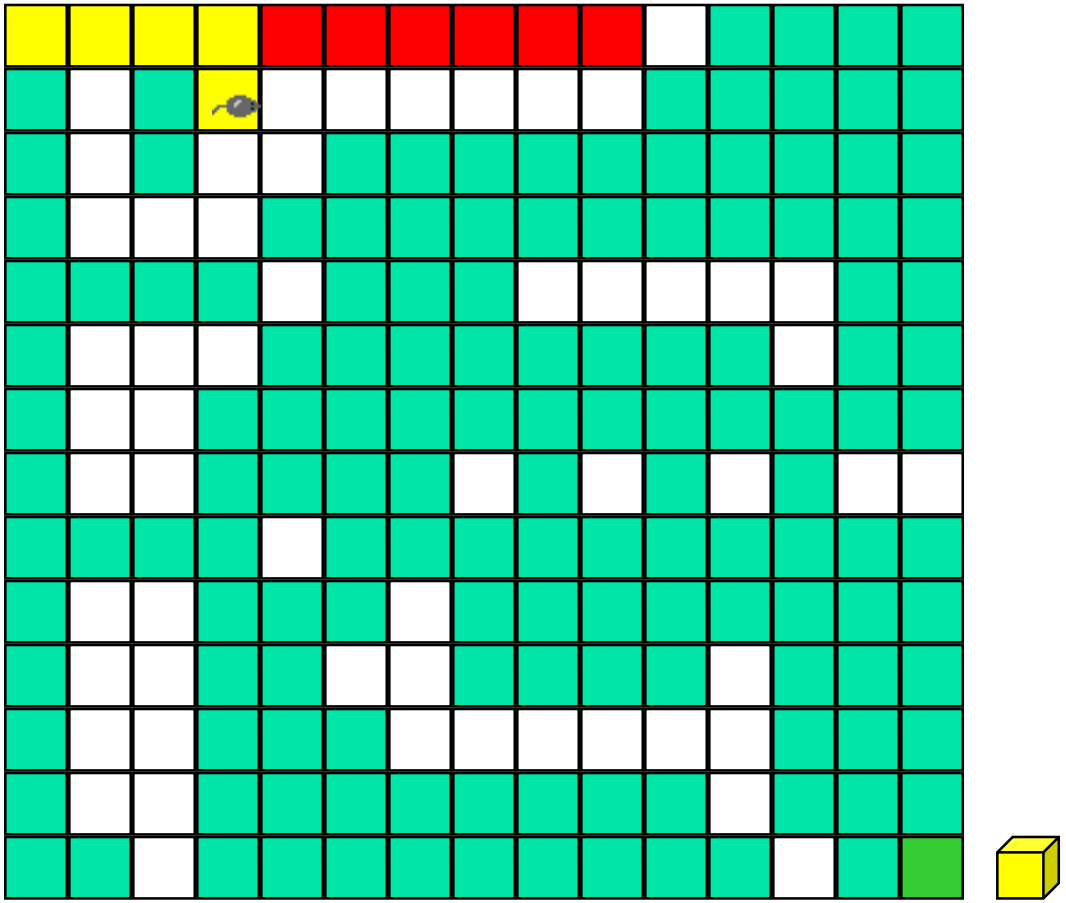
- Move backward until we reach a square from which a forward move is possible.

Rat In A Maze



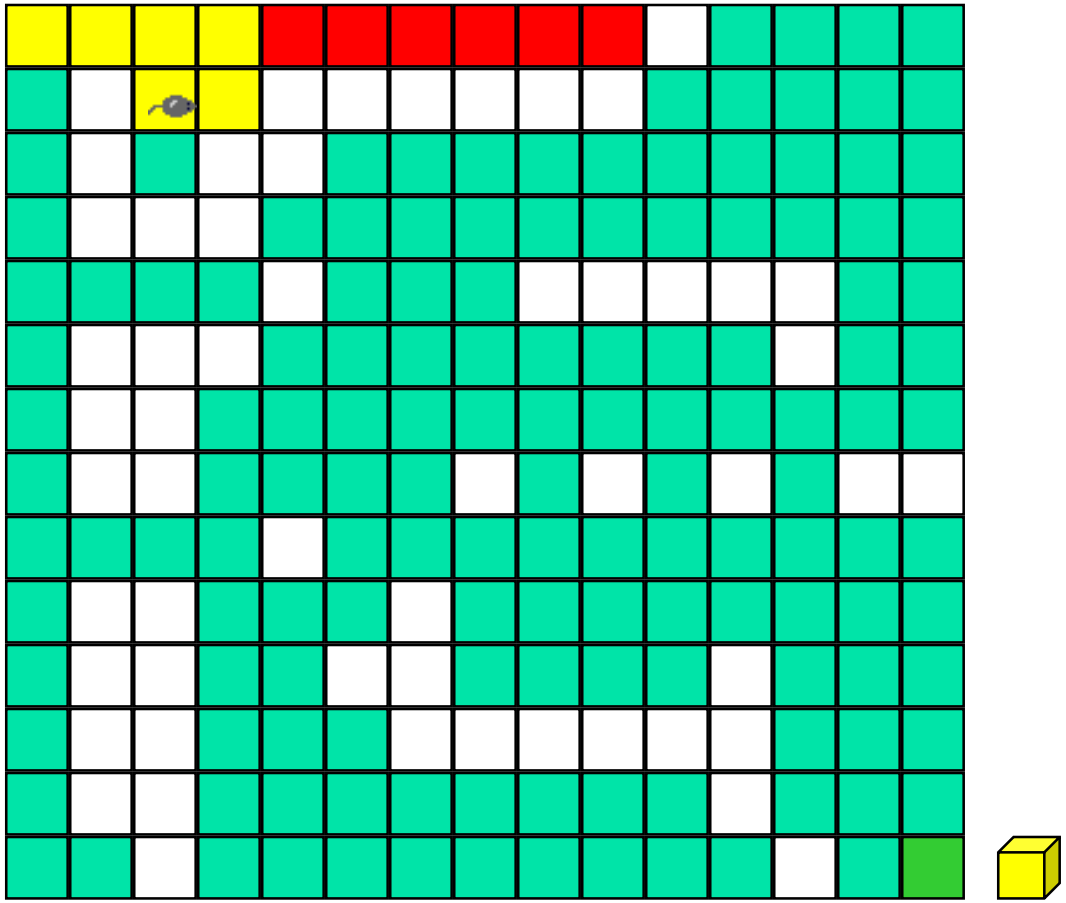
- Move down.

Rat In A Maze



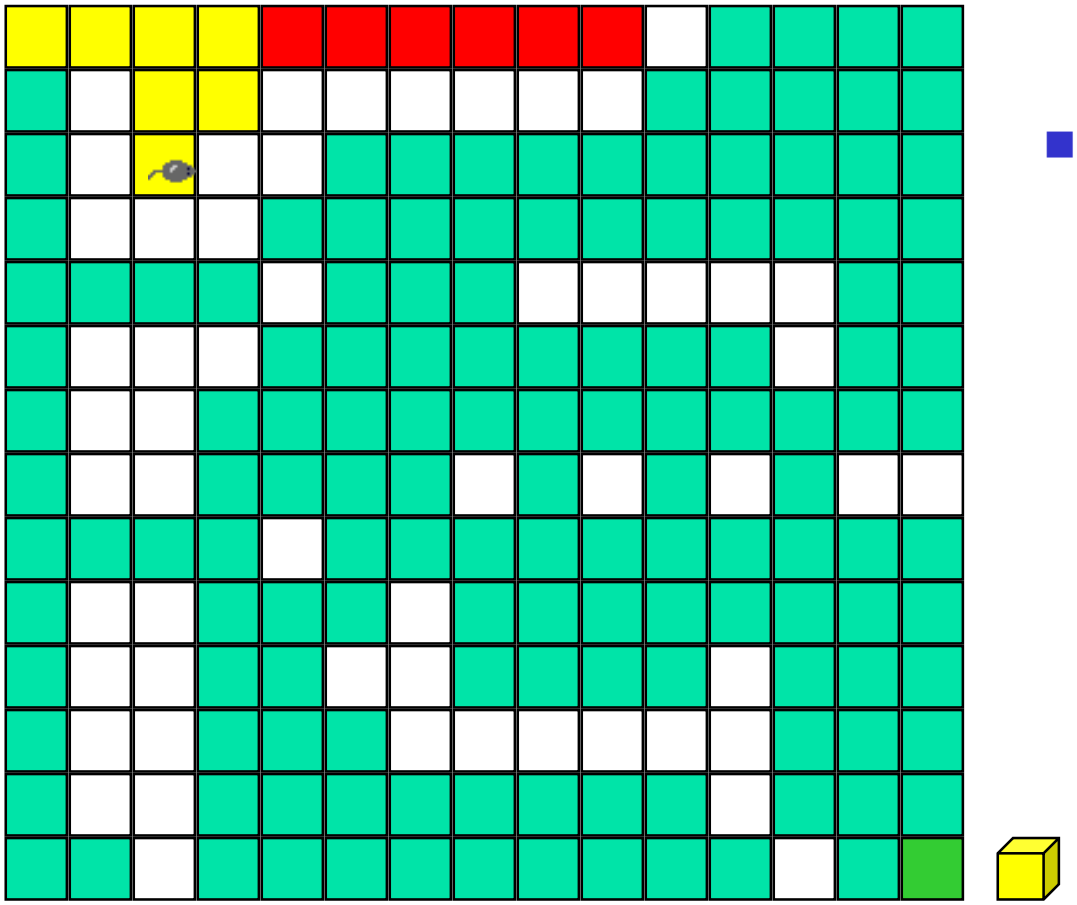
- Move left.

Rat In A Maze



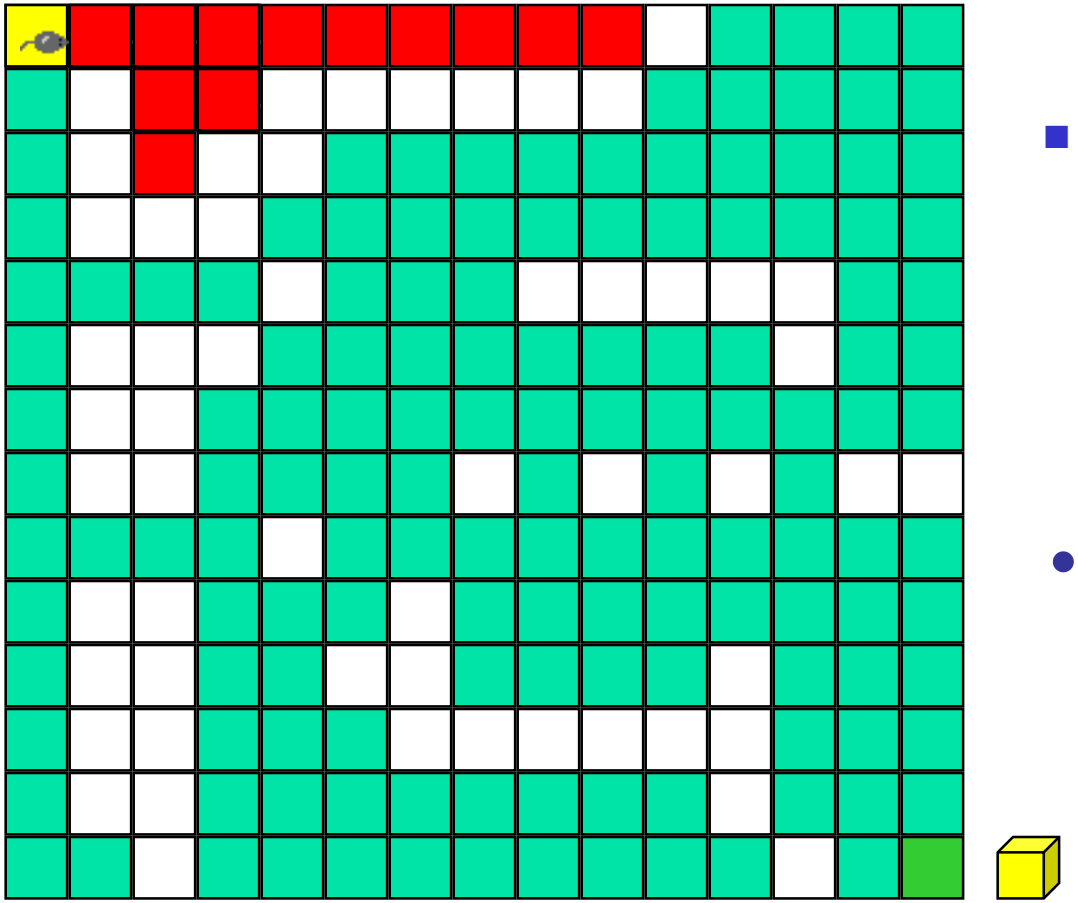
- Move down.

Rat In A Maze



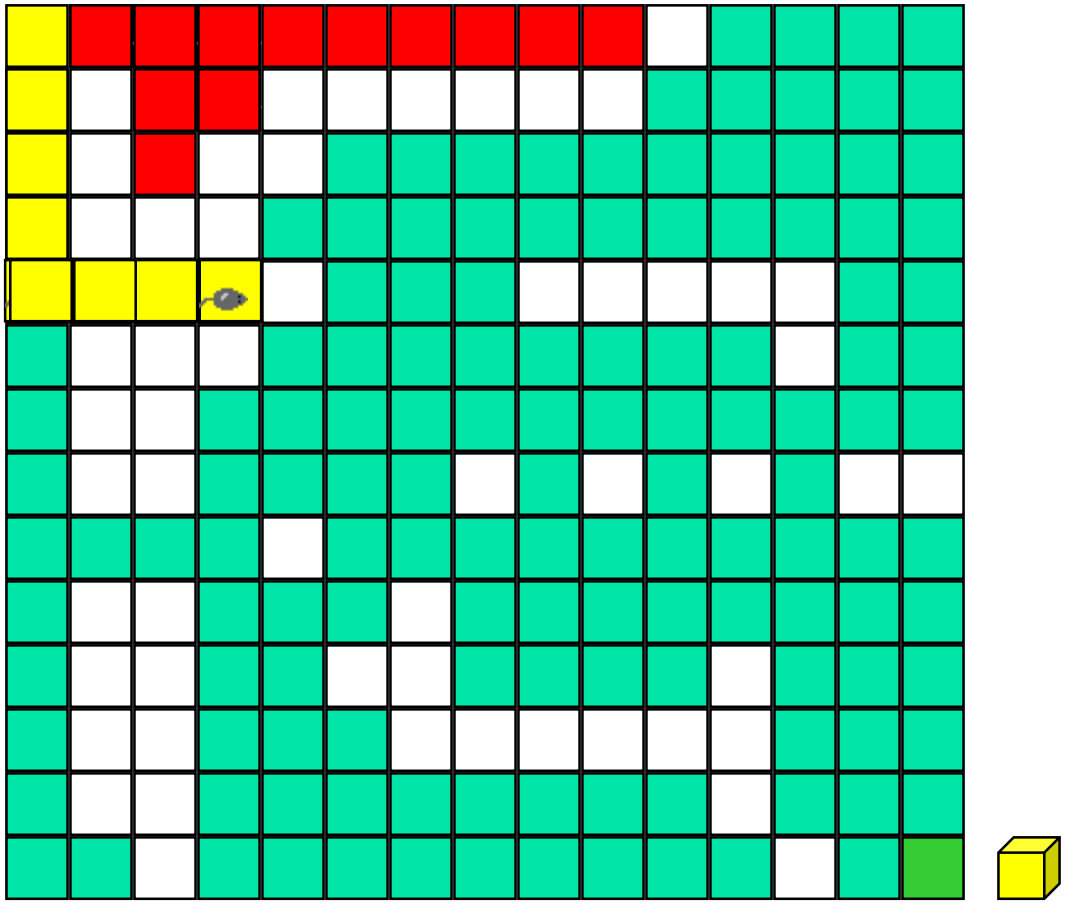
- Move backward until we reach a square from which a forward move is possible.

Rat In A Maze



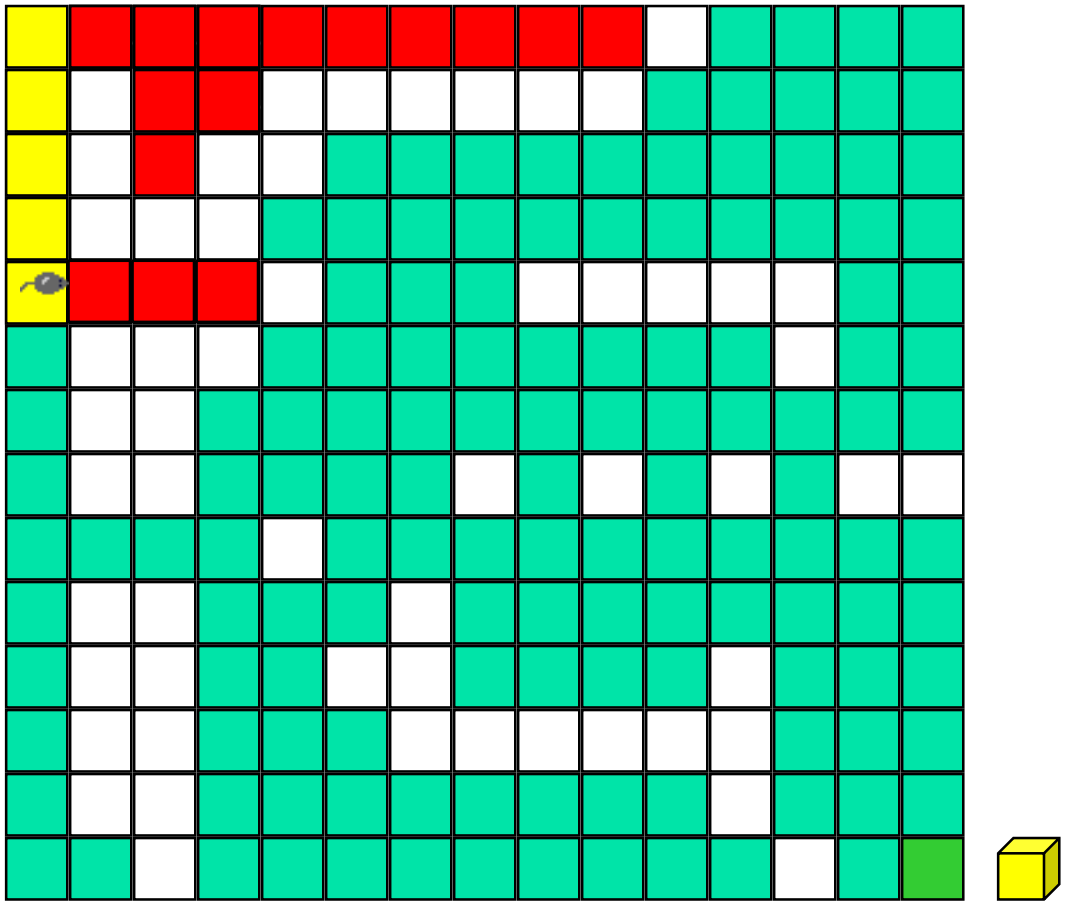
- Move backward until we reach a square from which a forward move is possible.
- Move downward.

Rat In A Maze



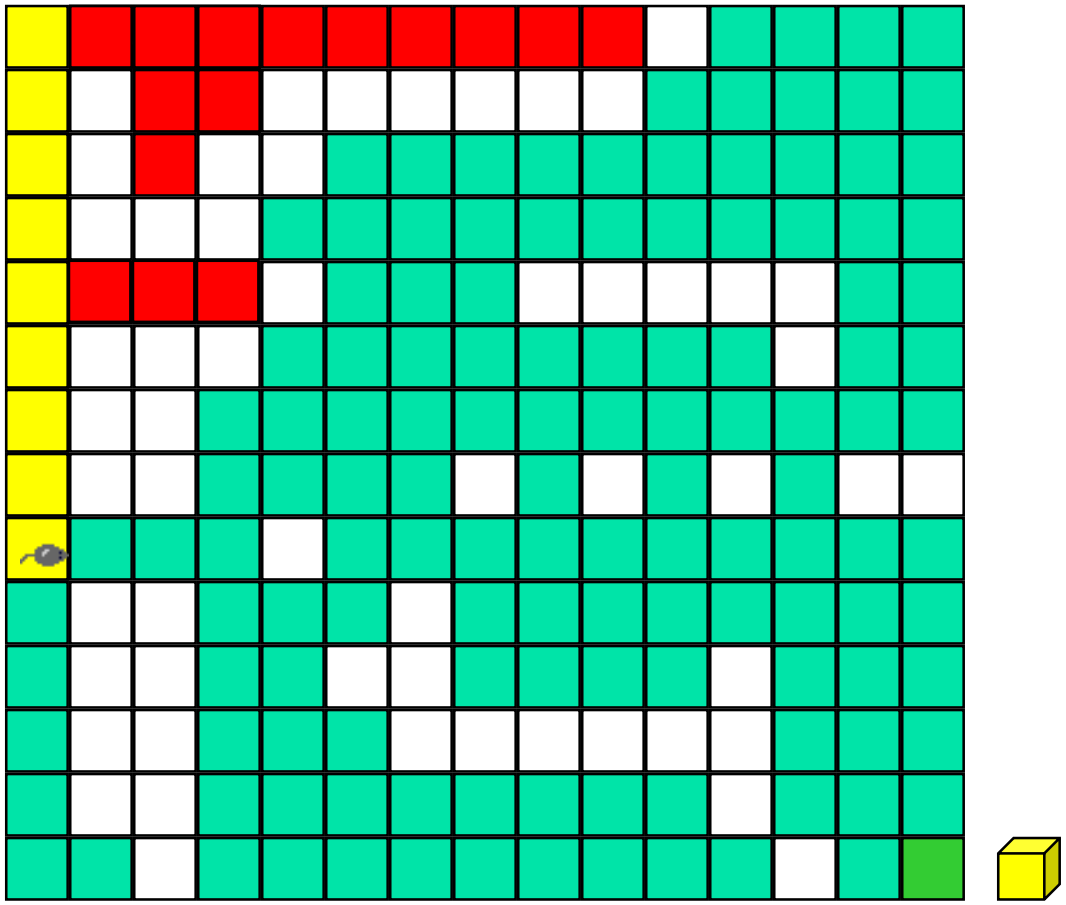
- Move right.
- Backtrack.

Rat In A Maze



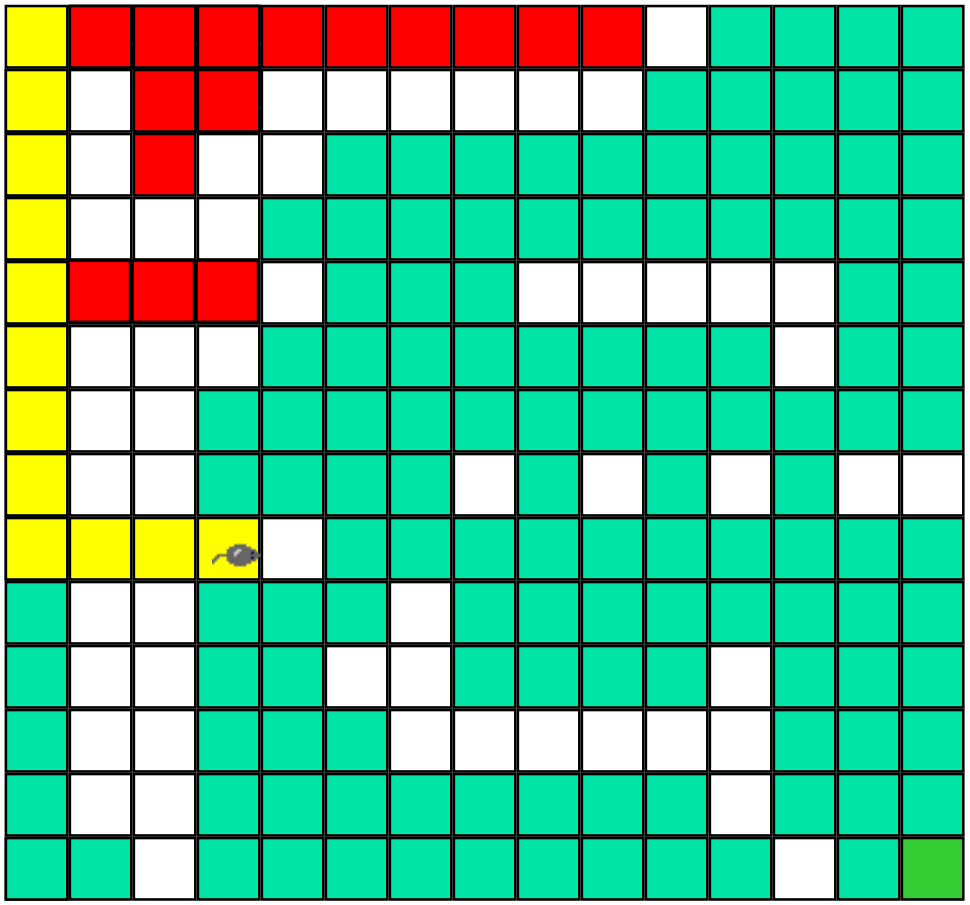
- Move downward.

Rat In A Maze



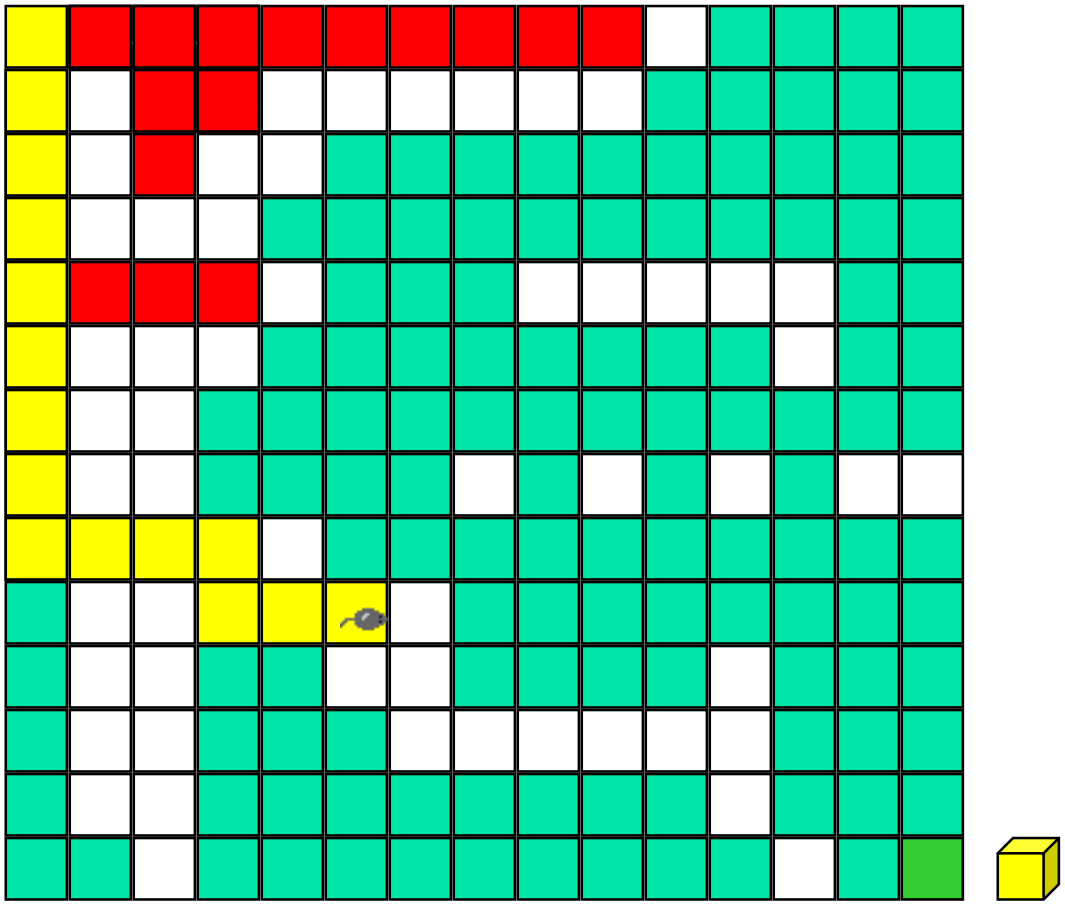
- Move right.

Rat In A Maze



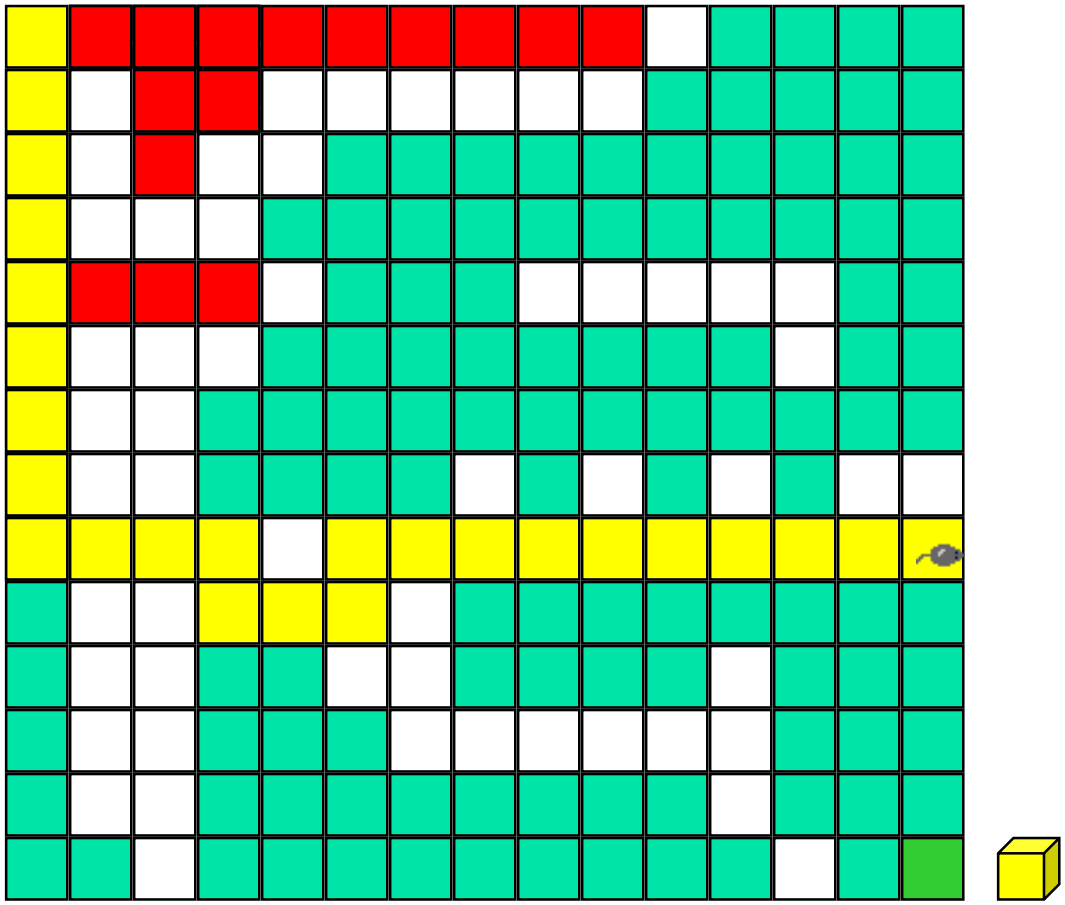
- Move one down and then right.

Rat In A Maze



- Move one up and then right.

Rat In A Maze



- Move down to exit and eat cheese.

Standing… Wondering…

- Move forward whenever **possible**
  - No wall & not visited
- Move back ---- HOW?
  - Remember the footprints
  - NEXT possible move from previous position
- Storage?
  - STACK

Path from maze entry to current position operates as a stack!

It’s a LONG life …

- How to continue this misery?
  - Whenever exist a possible move from previous positions
  - Whenever the stack is not empty

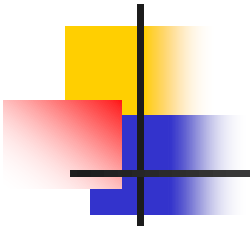
To Do: A Mazing Problem

Problem: find a path from the entrance to the exit of a maze.

entrance

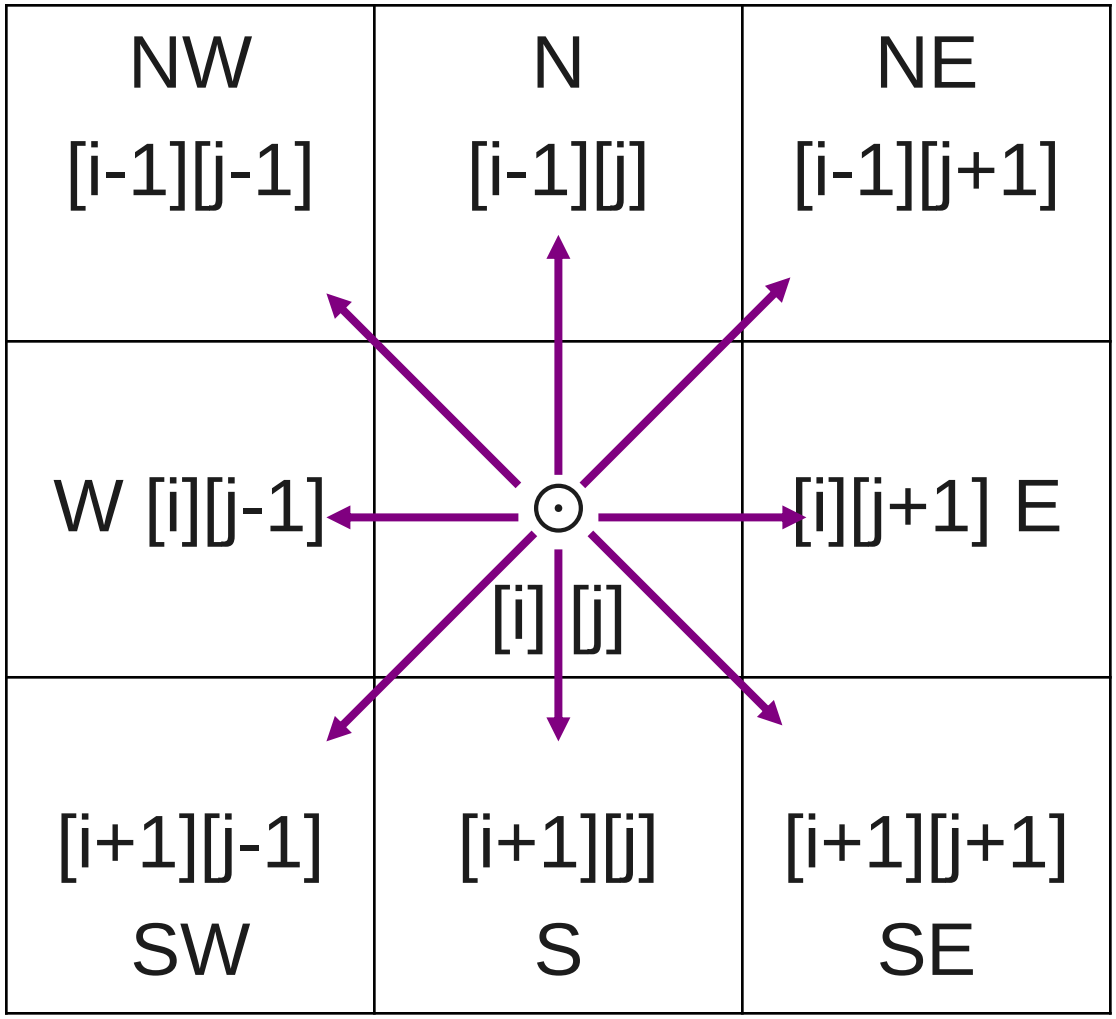
0	1	0	0	1	1	0	1	1
0	**0**	**0**	1	0	0	1	1	1
0	1	0	**0**	1	1	1	0	1
1	1	0	**0**	1	**0**	**0**	1	0
1	0	0	0	**0**	1	1	**0**	1
0	0	1	1	0	0	**0**	1	1
0	1	0	0	1	1	**0**	**0**	**0**

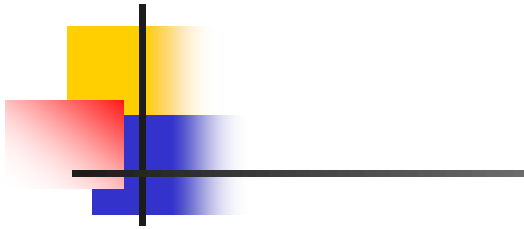
exit



Representation:

- **maze[i][j], $1 \le i \le m$, $1 \le j \le p$.**
- **1--- blocked, 0 --- open.**
- **the entrance: maze[1][1], the exit: maze[m][p].**
- **current point: [i][j].**
- **boarder of 1's, so a maze[m+2][p+2].**
- **8 possible moves: N, NE, E, SE, S, SW, W and NW.**


NW
[i-1][j-1]
N
[i-1][j]
NE
[i-1][j+1]
W [i][j-1]
[i][j]
[i][j+1] E
[i+1][j-1]
SW
[i+1][j]
S
[i+1][j+1]
SE



To predefine the 8 moves:

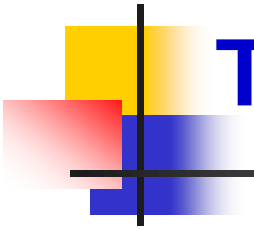
```
struct  offsets
{
    int a,b;
};

enum directions {N, NE, E, SE, S, SW, W, NW};

offsets move[8];
```

Table of moves

q	move[q].a	move[q].b
N	-1	0
NE	-1	1
E	0	1
SE	1	1
S	1	0
SW	1	-1
W	0	-1
NW	-1	-1

Thus, from [i][j] to [g][h] in SW direction:

g=i+move[SW].a; h=j+move[SW].b;



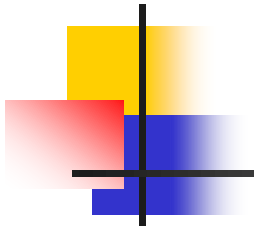
The basic idea

Given current position [i][j] and 8 directions to go, we pick one direction d, get the new position [g][h].

If [g][h] is the goal, success.

If [g][h] is a legal position, save [i][j] and d+1 in a stack in case we take a false path and need to try another direction, and [g][h] becomes the new current position.

Repeat until either success or every possibility is tried.



In order to prevent us from going down the same path twice:

use another array mark[m+2][p+2], which is initially 0.

Mark[i][j] is set to 1 once the position is visited.

First pass:

Initialize stack to the maze entrance coordinates and direction east**;**

while (stack is not empty)

{

(i, j, dir)=coordinates and direction from top of stack**;**

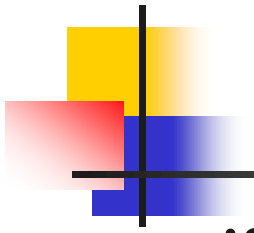
pop the stack**;**

while (there are more moves from (i, j))

{

(g, h)= coordinates of next move **;**

if ((g==m) **&&** (h==p)) success**;**

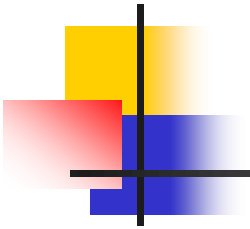


```
        if ((!maze[g][h]) && (!mark[g][h])) // legal and not visited
        {
            mark[g][h]=1;
            dir=next direction to try;
            push (i, j, dir) to stack;
            (i, j, dir) = (g, h, N);
         }
    }
}
cout << "No path in maze."<< endl;
```

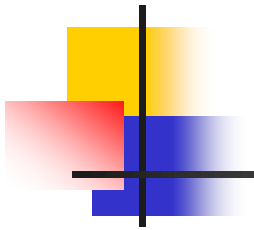
We need a stack of items:

```
struct Items {
        int x, y, dir;
 };
```

```
void path(const int m, const int p)
{ //Output a path (if any) in the maze; maze[0][i] = maze[m+1][i]
  // = maze[j][0] = maze[j][p+1] = 1, 0 ≤ i ≤ p+1, 0 ≤ j ≤ m+1.
    // start at (1,1)
    mark[1][1]=1;
    Stack<Items> stack(m*p);
    Items temp(1, 1, E);
    stack.Push(temp);
    while (!stack.IsEmpty())
    {
          temp= stack.Top();
        Stack.Pop();
        int i=temp.x; int j=temp.y; int d=temp.dir;
```



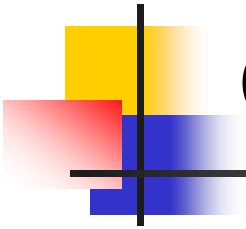
```
while (d<8)
{
    int g=i+move[d].a; int h=j+move[d].b;
    if ((g==m) && (h==p)) { // reached exit
        // output path
         cout <<stack;
         cout << i<<" "<< j<<" "<<d<< endl; // last two
         cout << m<<" "<< p<< endl;          // points
         return;
       }
```



```
            if ((!maze[g][h]) && (!mark[g][h])) { //new position
               mark[g][h]=1;
               temp.x=i; temp.y=j; temp.dir=d+1;
               stack.Push(temp);
               i=g ; j=h ; d=N;   // move to (g, h)
             }
              else d++;  // try next direction
        }
    }
 cout << "No path in maze."<< endl;
}
```

Stacks and Queues

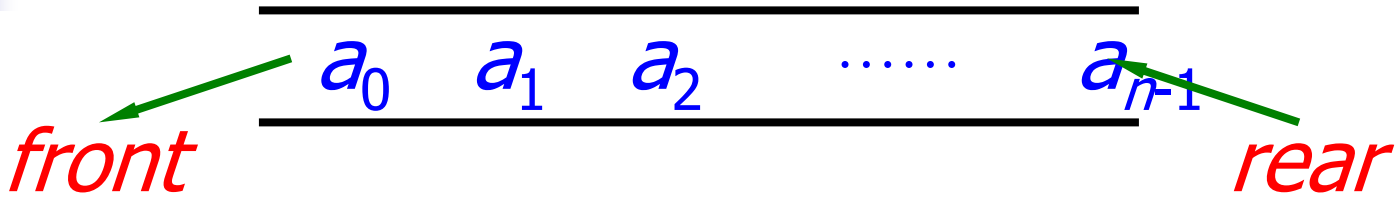
- 栈(Stack)
  - 基本概念
  - 顺序存储结构
  - 链式存储结构
  - 应用
- 队列（Queue）
  - -- 基本概念
  - – 顺序存储结构
  - – 链式存储结构
  - – 应用



Queue

- Queue is
  - an ordered list.
  - in insertion known as push.
  - in remove known as pop.
  - take place at different ends.
  - new elements are added at the rear.
  - old elements are remove from the front.
- Queue is also known as a FIFO list
  - FIFO: First-In-First-Out

Queue



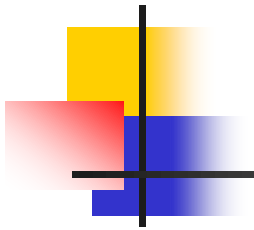
- 定义
  - 队列是只允许在一端删除，在另一端插入的线性表
  - 允许删除的一端叫做队头(front)，允许插入的一端叫做队尾(rear)。
- 特性
  - 先进先出(FIFO, First In First Out)

ADT of Queue

```
template <class E>
class Queue {
public:
    Queue() { };              //构造函数
    ~Queue() { };             //析构函数
    virtual bool EnQueue(E x) = 0;          //进队列
    virtual bool DeQueue(E& x) = 0;         //出队列
    virtual bool getFront(E& x) = 0;        //取队头
    virtual bool IsEmpty() const = 0;       //判队列空
    virtual bool IsFull() const = 0;        //判队列满
};
```

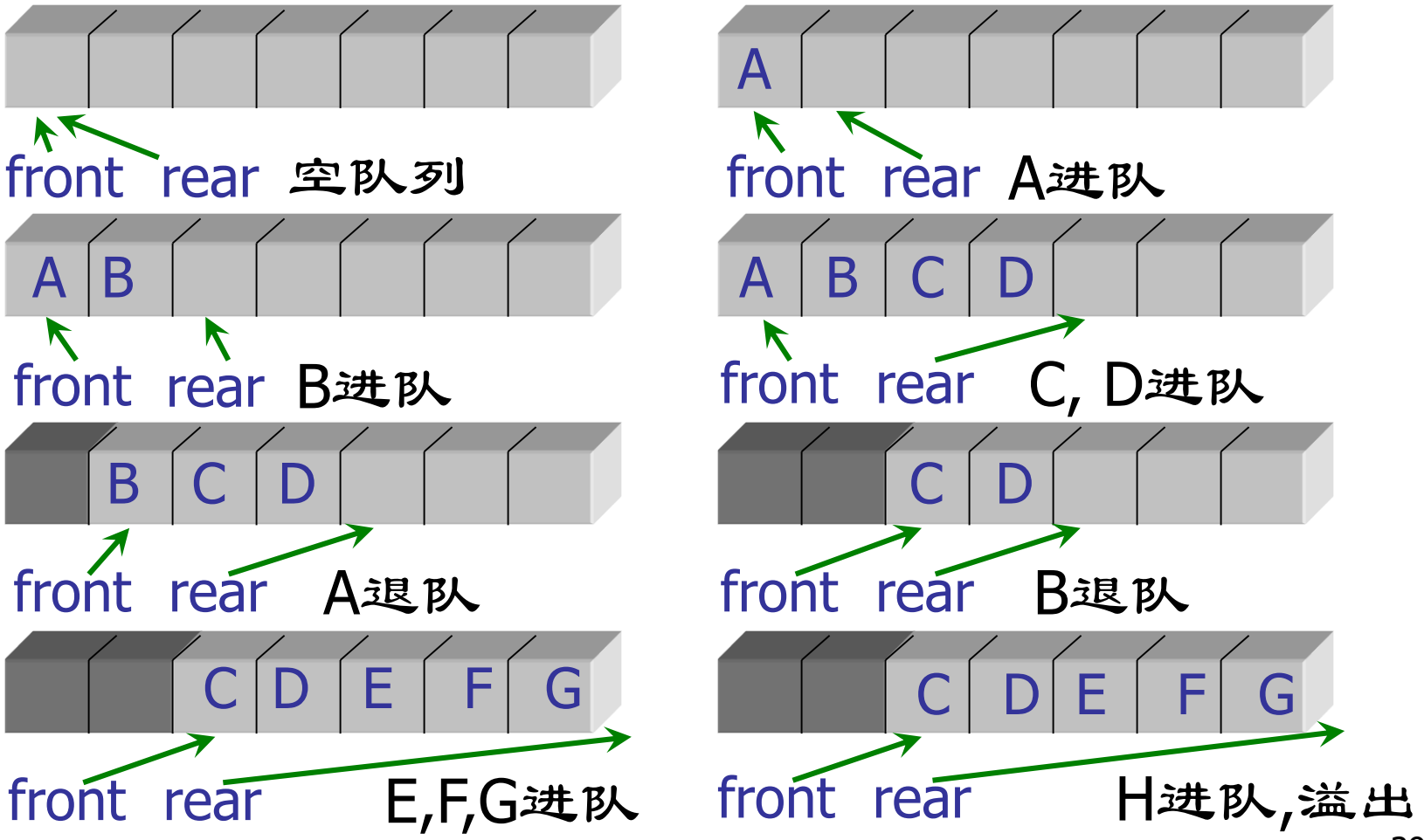
队列的数组存储表示——顺序队列

```
#include <assert.h>
#include <iostream.h>
#include "Queue.h"
template <class E>
class SeqQueue : public Queue<E> {  //队列类定义
protected:
     int rear, front;                    //队尾与队头指针
     E *elements;                        //队列存放数组
     int maxSize;                        //队列最大容量
public:
    SeqQueue(int sz = 10);     //构造函数
```



```
    ~SeqQueue()
      {  delete[ ] elements; }  //析构函数
    bool EnQueue(E x);          //新元素进队列
    bool DeQueue(E& x);         //退出队头元素
    bool getFront(E& x);        //取队头元素值
    void makeEmpty()
      {  front = rear = 0; }
    bool IsEmpty() const
      {  return front == rear; }
    bool IsFull() const
      { return ((rear+1)% maxSize == front); }
    int getSize() const
      { return (rear–front+maxSize) % maxSize; }
};
```

An Example of Queue

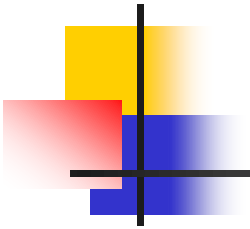


Principles of EnQueue and DeQueue

- When EnQueue
  - 先将新元素按 rear 指示位置加入，
  - 再将队尾指针加一： rear = rear + 1，
  - 队尾指针指向实际队尾的后一位置。
- When DeQueue
  - 先按队头指针指示位置取出元素，
  - 再将队头指针进一: front = front + 1，
  - 队头指针指向实际队头位置。

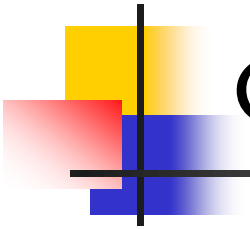
Principles of EnQueue and DeQueue

- 队满时
  - 进队将溢出出错（假溢出）
- 队空时
  - 出队将队空处理
- 如何解决假溢出问题？
- 解决假溢出的办法之一：将队列元素存放数组首尾相接，形成循环（环形）队列



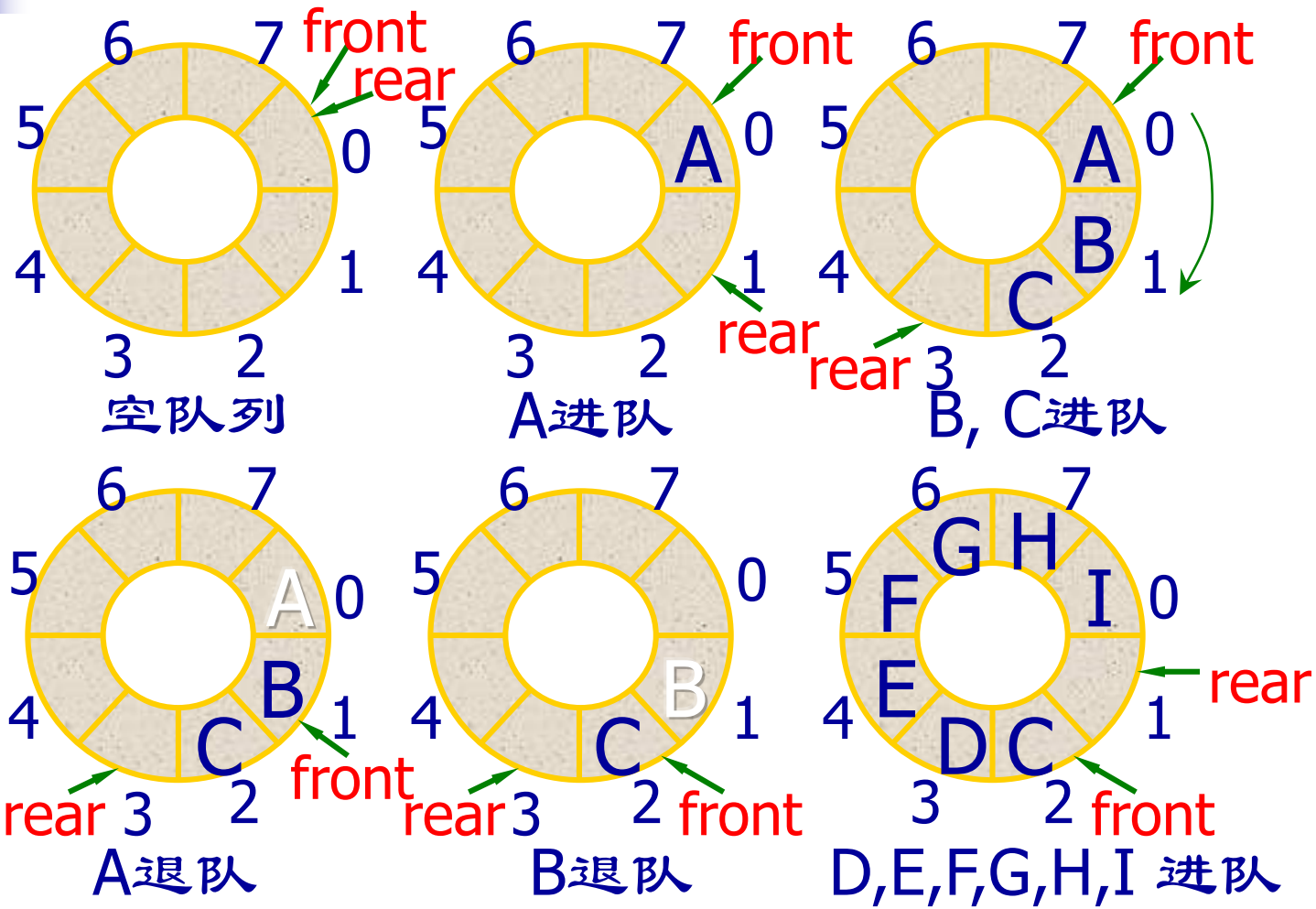
Circular Queue

- 队列存放数组被当作首尾相接的表处理
- 队尾指针加1时，从maxSize−1直接进到0
- 可用语言的取模(余数)运算实现



Circular Queue

- 队头指针进1: front = (front+1) % maxSize;
- 队尾指针进1: rear = (rear+1) % maxSize;
- 队列初始化：front = rear = 0;
- 队空条件：front == rear;
- 队满条件：(rear+1) % maxSize == front

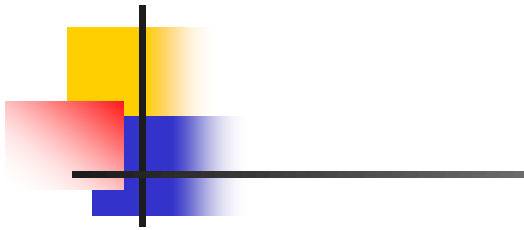

front
rear
空队列
A
A进队
B
C
rear
B, C进队
A退队
B退队
G
H
F
I
E
D
C
D,E,F,G,H,I 进队

循环队列操作的定义

```
void MakeEmpty() { front = rear = 0; }
int IsEmpty() const { return front == rear; }
int IsFull() const
    { return  (rear+1) % maxSize == front; }
```

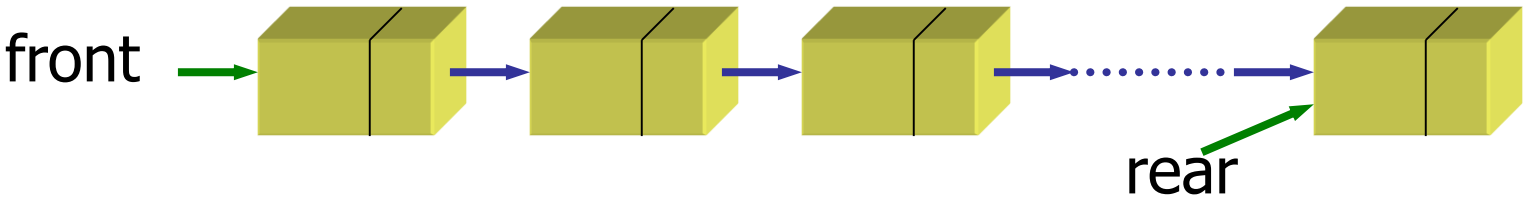
```
template <class E>
SeqQueue<E>::SeqQueue(int sz)
   : front(0), rear(0), maxSize(sz) {     //构造函数
     elements = new E[maxSize];
     assert ( elements != NULL );
};
```

```
template <class E>
bool SeqQueue<E>::EnQueue(E x) {
//若队列不满,则将x插入到该队列队尾,否则返回
    if (IsFull() == true) return false;
    elements[rear] = x;                  //先存入
    rear = (rear+1) % maxSize;           //尾指针加一
    return true;
}

template <class E>
bool SeqQueue<E>::DeQueue(E& x) {
//若队列不空则函数退队头元素并返回其值
    if (IsEmpty() == true) return false;
     x = elements[front];                //先取队头
    front = (front+1) % maxSize;         //再队头指针加一
    return true;
}
```



```
template <class E>
bool SeqQueue<E>::getFront(E& x) const {
//若队列不空则函数返回该队列队头元素的值
    if (IsEmpty() == true) return false;    //队列空
    x = elements[front];                    //返回队头元素
    return true;
};
```

队列的链接存储表示 — 链式队列



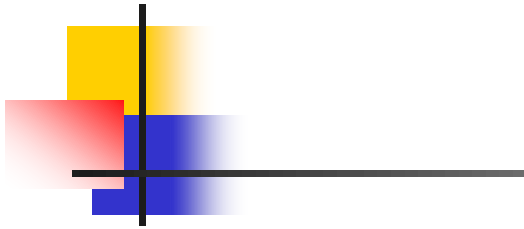
- 队头在链头，队尾在链尾
- 链式队列在进队时无队满问题，但有队空问题
- 队空条件为 front == NULL

链式队列类的定义

```
#include <iostream.h>
#include "Queue.h"
template <class E>
struct QueueNode {                          //队列结点类定义
private:
    E data;                                  //队列结点数据
    QueueNode<E> *link;                     //结点链指针
public:
    QueueNode(E d = 0, QueueNode<E>
      *next = NULL) : data(d), link(next) { }
};
```

```
template <class E>
class LinkedQueue {
private:
     QueueNode<E> *front, *rear;   //队头、队尾指针
public:
    LinkedQueue() : rear(NULL), front(NULL) { }
    ~LinkedQueue();
    bool EnQueue(E x);
    bool DeQueue(E& x);
    bool GetFront(E& x);
    void MakeEmpty();              //实现与~Queue()同
    bool IsEmpty() const { return  front == NULL; }
};
```

```
template <class E>
LinkedQueue<E>::~LinkedQueue() {      //析构函数
    QueueNode<E> *p;
    while (front != NULL) {           //逐个结点释放
        p = front;  front = front->link;  delete p;
    }
}
template <class E>
bool LinkedQueue<E>::GetFront(E& x) {
//若队列不空，则函数返回队头元素的值
    if (IsEmpty() == true) return false;
    x = front->data;  return true;
}
```



```
template <class E>
bool LinkedQueue<E>::EnQueue(E x) {
//将新元素X插入到队列的队尾
      if (front == NULL)  {            //创建第一个结点
          front = rear = new QueueNode<E> (x);
         if (front == NULL) return false; }       //分配失败
    else {                                  //队列不空, 插入
         rear->link = new QueueNode<E> (x);
         if (rear->link == NULL) return false;
         rear = rear->link;
    }
    return true;
}
```

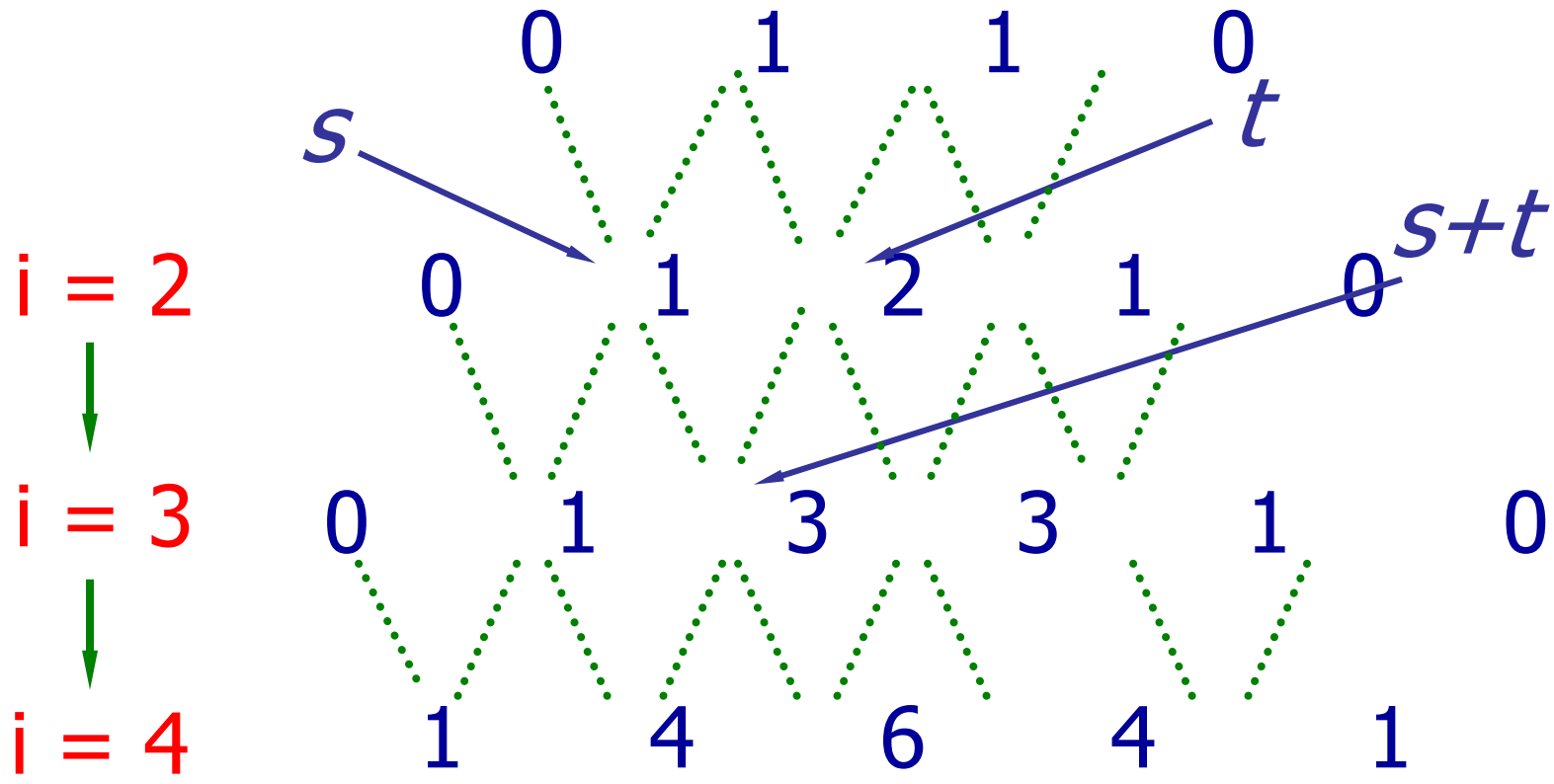
```
template <class E>
//如果队列不空，将队头结点从链式队列中删去
bool LinkedQueue<E>::DeQueue(E& x) {
    if (IsEmpty() == true) return false;        //判队空
    QueueNode<E> *p = front;
    x = front->data;  front = front->link;
    delete p;
   return true;
}
```

队列的应用：打印杨辉三角形

- 算法逐行打印二项展开式 $(a + b)^i$ 的系数：
  杨辉三角形 (Pascal's triangle)

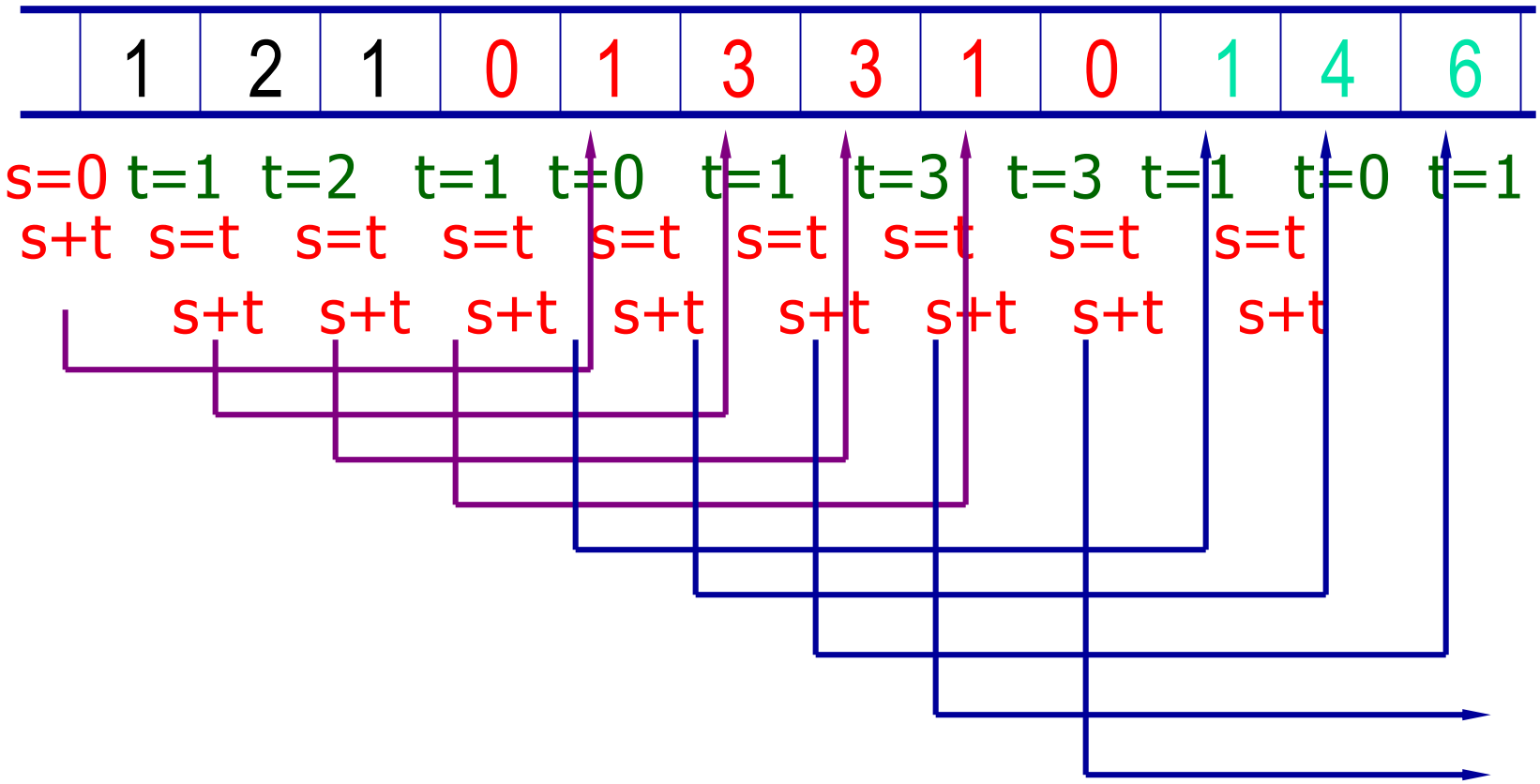
					1		1						i = 1
				1		2		1					2
			1		3		3		1				3
		1		4		6		4		1			4
	1		5		10		10		5		1		5
1		6		15		20		15		6		1	6

分析第 i 行元素与第 i+1行元素的关系



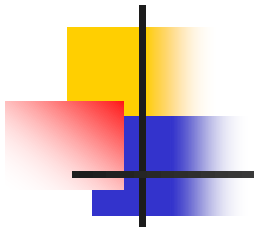
从前一行的数据可以计算下一行的数据

从第 i 行数据计算并存放第 i+1 行数据



The Algorithm with Queue

```
#include <stdio.h>
#include <iostream.h>
#include "queue.h"

void YANGHVI(int n)
{
    Queue q(n+1);                    //队列初始化
    q.MakeEmpty();
    q.EnQueue(1);  q.EnQueue(1);     //第1行的系数
    int s = 0, t;
```



```
        for (int i = 1; i <= n-1; i++) {          //逐行计算
          cout << endl;
          q.EnQueue(0);          //分隔各行
          for (int j = 1; j <= i+2; j++) {     //下一行
              q.DeQueue(t);
              q.EnQueue(s + t);
              s = t;
              if (j != i+2) cout << s << ' ';
          }
    }
}
```