

数据结构与算法

DATA STRUCTURE

第十三讲 递归

胡浩栋

信息管理与工程学院

2017 - 2018 第一学期

课堂内容

- 作业三回顾
- 递归Recursion

Mergesort

```
void SortedIntArray::MergeSort(int start, int end)
{
    if (start >= end)
    {
        return;
    }

    int middle = start + (end - start) / 2;
    MergeSort(start, middle);
    MergeSort(middle+1, end);
    Merge(start, middle+1, end);
}
```

```
int SortedIntArray::PartitionRandom(int start, int end)
{
    int rand = GetRandom(start, end);
    swap((*this)[rand], (*this)[end]);
    return PartitionEnd(start, end);
}
```

```
void SortedIntArray::Merge(int start, int middle, int end)
{
    IntArray left(middle-start+1);
    for (int i=0; i < middle - start; i++)
    {
        left[i] = (*this)[start + i];
    }

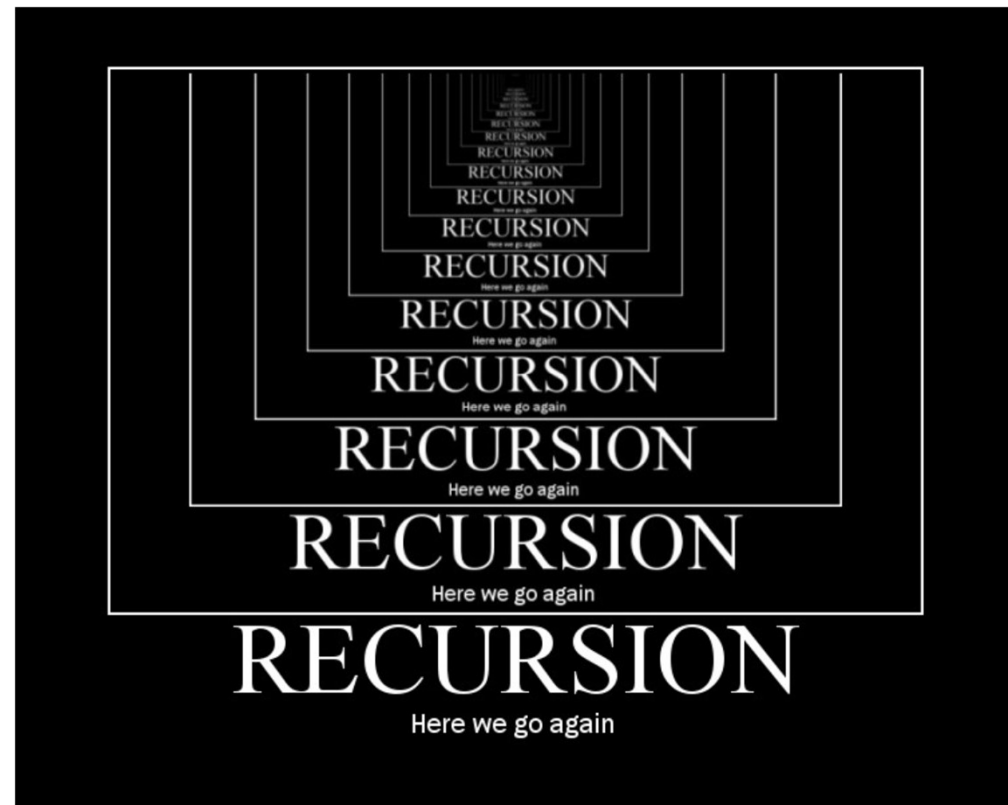
    left[middle-start] = numeric_limits<int>::max();

    IntArray right(end-middle+2);
    for (int i=0; i<end-middle+1; i++)
    {
        right[i] = (*this)[middle+i];
    }

    right[end-middle+1] = numeric_limits<int>::max();

    // Have two pointer to left array and right array.
    // move the smaller number of both arrays to merged array
    for (int k=start, i=0, j=0; k<=end; k++)
    {
        if (left[i] < right[j])
        {
            (*this)[k] = left[i];
            i++;
        }
        else
        {
            (*this)[k] = right[j];
            j++;
        }
    }
}
```

递归Recursion



递归的概念

- 从前有座山，山上有座庙，庙里和尚讲“从前有座。。。”。

```
int Func()  
{  
    return 1 + Func();  
}
```

➡ $1 + (1 + (1 + (1 + \dots + (1 + Func()))))$

- 一个函数直接或间接地调用自身，是为直接或间接递归
 - (1) 递：在过程或函数里调用自身
 - (2) 归：必须有一个明确的结束条件

上面的例子只有递，没有归

递归一般用于解决三类问题

- 数据的定义是按递归定义的
 - Fibonacci函数。 $F(n) = F(n - 1) + F(n - 2)$
 - n 的阶乘。 $Factorial(n) = Factorial(n - 1) \times n$
- 问题解法按递归实现
 - 汉诺塔, Mergesort: 分治法思想
 - 回溯: 数独, 迷宫, 八皇后问题
- 数据的结构形式是按递归定义的
 - 树的遍历
 - 图的搜索

递归的优缺点

- 优点
 - 把大问题分解位小问题后求解
 - 容易理解问题本质
 - 实现简单
- 缺点
 - 比循环运行效率低
 - 递归层数太深，会造成stack overflow
 - 有时候更难理解

递归算法模板

```
int Recursion()  
{  
    if (Base Case)  
    {  
        return value;  
    }  
  
    Break problem into subproblems  
    while (have subproblem)  
    {  
        Call Recursion() on each subproblem  
    }  
    Merge results of subproblems.  
}
```

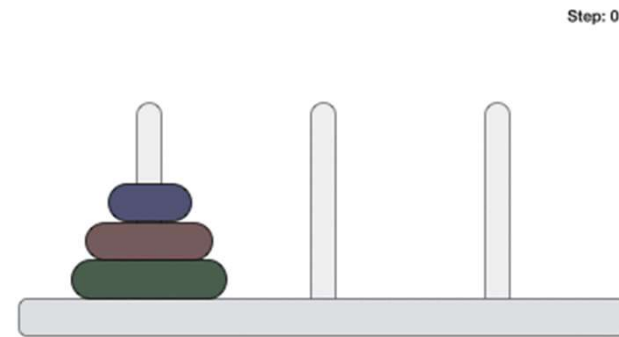
Fibonacci函数和 n 的阶乘的递归算法都是类似的



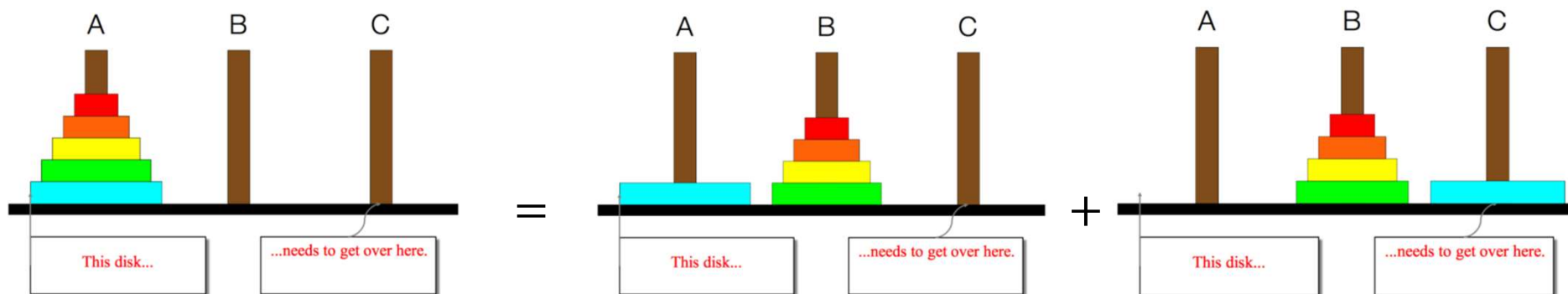
```
int Count(int n)  
{  
    if (n <= 1)  
    {  
        return 0;  
    }  
    return 1 + Count(n-1);  
}
```


汉诺塔 Tower of Hanoi

1. 每次只能移动一个
2. 只有顶部的可以被移动
3. 只有小的才能放到大 的上面



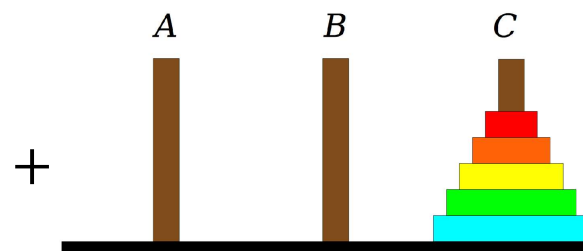
递归移动汉诺塔



```
void Hanoi(int n, int A, int B, int C)
{
    if (n == 0)
    {
        return;
    }

    Hanoi(n-1, A, C, B);
    cout << "move " << A << " to " << C << endl;
    Hanoi(n-1, B, A, C);
}
```

$$\begin{aligned} T(n) &= T(n-1) + 1 + T(n-1) \\ &= 2T(n-1) + 1 \\ &= 2^n - 1 \end{aligned}$$

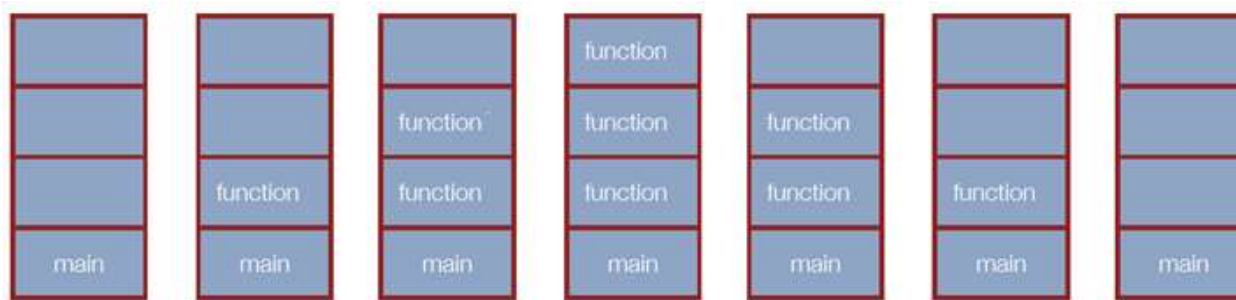


思考题：迭代方式如何实现？

递归转化为非递归

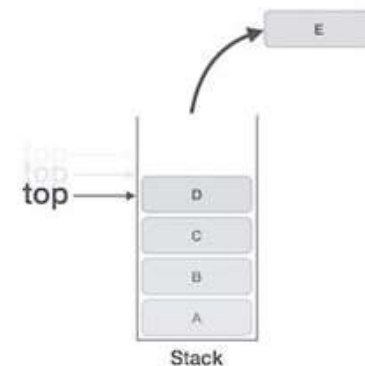
- 递归方法是一种很有效很自然的分析和描述问题的方法
- 但递归算法运算效率低，而且栈空间有限
- 有时先用递归的思想分析和描述问题，然后转化成非递归的算法
- 非递归算法有时候不好理解

递归调用时call stack

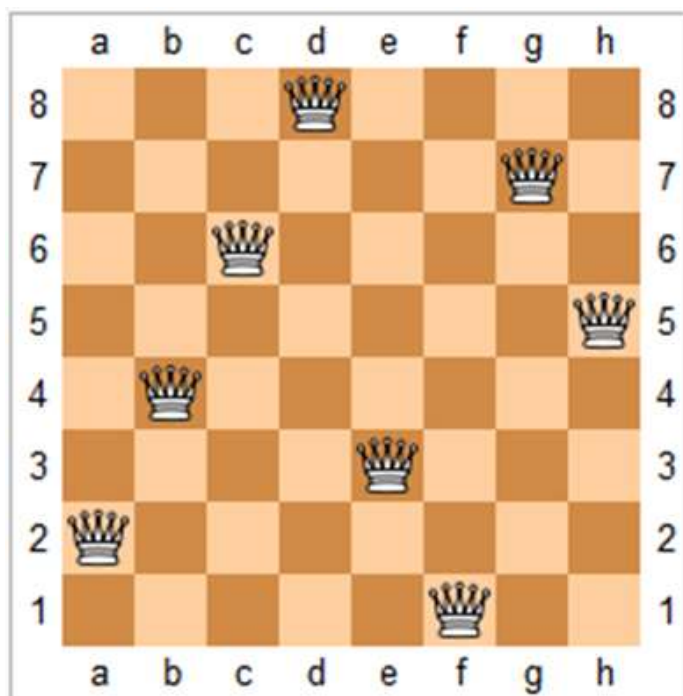


就是后进先出的栈，
可以借助栈来实现非递归过程

原理上讲，所有递归都是可以消除的，
代价就是可能自己要维护一个栈，过程不容易理解



八皇后问题

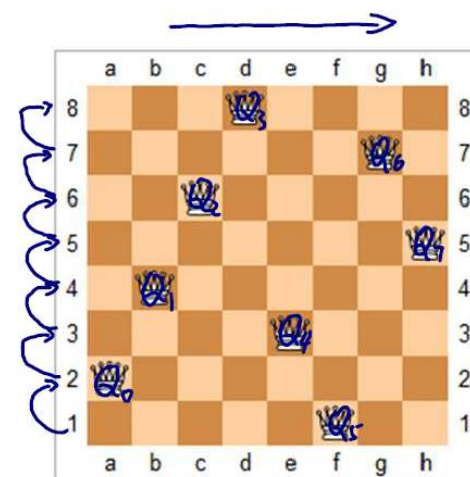


回溯递归算法

假定 Q_i 是Queen在 i 列可以放的位置（行号）

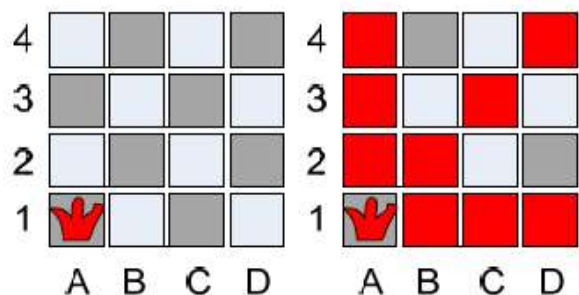
假定递归函数是从 i 列往后的Q可行放法

- 从第 i 列开始，试着把 Q_i 从第一行开始放，
 1. 如果已经有攻击了，就试下一行
 2. 如果安全，就递归放置后面的Queen。
- 如果后面的Queen不能安全放，重复1) 和2) ，直到有一个合适的位置
- 如果当前 i 列的所有行数都有攻击，那么返回不能放。

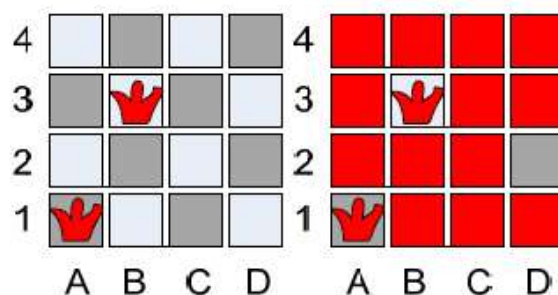


八皇后问题

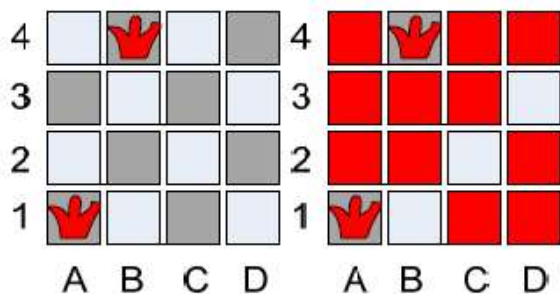
4-Queen求解过程



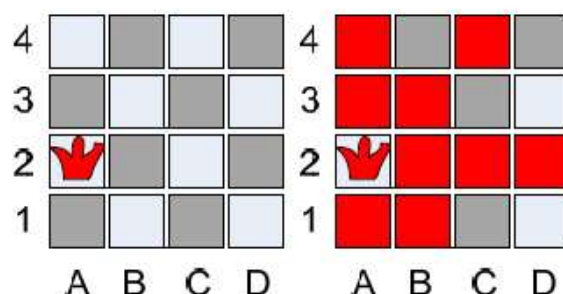
1. 起始状态，从第一列开始， Q_1 放在第一行



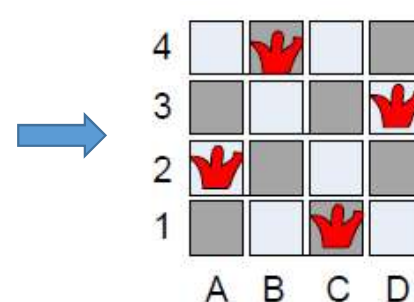
2. 继续第二列， Q_2 只能放第三行，但第三列不能再放



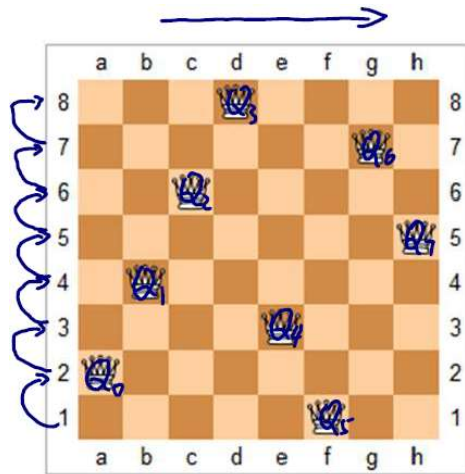
3. 回溯 Q_2 到第四行，但接着会发现 Q_4 有问题



4. 回溯至 Q_1 ，尝试第二行； Q_2 有一个选择； Q_3 ， Q_4 都能放下



递归实现



八皇后问题

```
bool SolveNQ(int board[][N], int col)
{
    /* base case: If all queens are placed
       then return true */
    if (col >= N)
    {
        return true;
    }

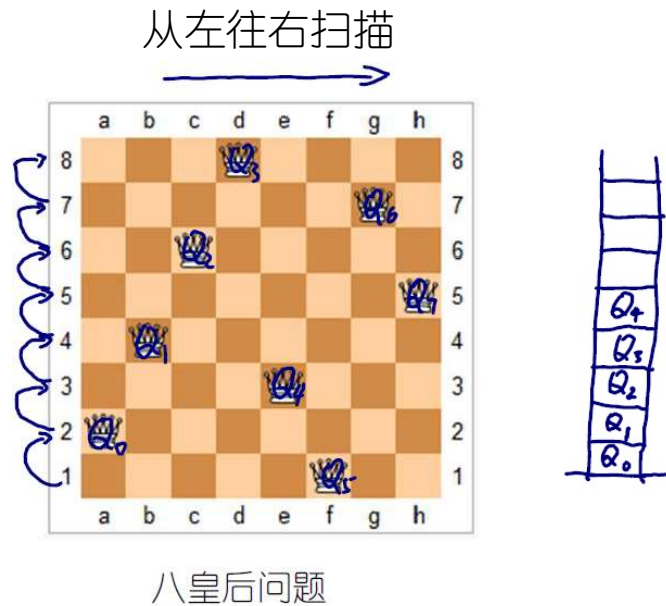
    /* Consider this column and try placing
       this queen in all rows one by one */
    for (int row = 0; row < N; row++)
    {
        /* Check if queen can be placed on
           board[i][col] */
        if (isSafe(board, row, col) )
        {
            /* Place this queen in board[row][col] */
            board[row][col] = 1;

            /* recur to place rest of the queens */
            if (SolveNQ(board, col + 1) )
            {
                return true;
            }

            /* If placing queen in board[row][col]
               doesn't lead to a solution, then
               remove queen from board[row][col] */
            board[row][col] = 0; // BACKTRACK
        }
    }

    return false;
}
```


迭代实现



```
bool SolveNQ_Iterative(int board[][N], int N)
{
    stack<int> stk;
    for (int col = 0; col < N; col++)
    {
        /* Consider this col and try placing
           this queen in all rows one by one */
        for (int row = 0; row < N; row++)
        {
            /* Place this queen in board[row][col] */
            board[row][col] = 1;

            /* Check if queen can be placed on board[row][col] */
            if (IsSafe(board, row, col) )
            {
                stk.push(row * 100 + col);
                break;
            }
            // It is not safe, remove queen at current row
            board[row][col] = 0;

            // This is backtrace steps, if all rows are not safe.
            // Restore row/col to previous state and retry
            while (row >= N-1)
            {
                if (stk.empty())
                {
                    return false;
                }
                row = stk.top() / 100;
                col = stk.top() % 100;
                stk.pop();
                board[row][col] = 0;
            }
        }
    }

    return true;
}
```

尾递归

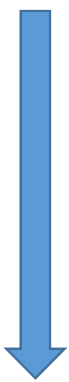
- 递归调用在函数的尾部
- 尾递归很容易转化为循环
- 右边例子是不是尾递归？

```
int Factorial(int n)
{
    if (n <= 1)
    {
        return 1;
    }
    return n * Factorial(n-1);
}
```

尾递归

改写尾递归，需要把所有用到的内部变量改写成函数的参数。
下面的例子，阶乘函数 **Factorial** 需要用到一个中间变量 **total**，
把这个中间变量改写成函数的参数。

factorial(5, 1)
factorial(4, 5)
factorial(3, 20)
factorial(2, 60)
factorial(1, 120)
120

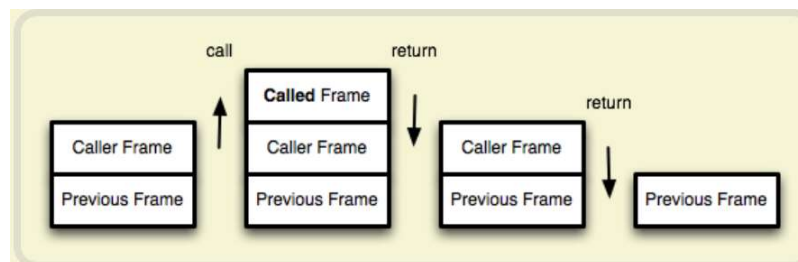
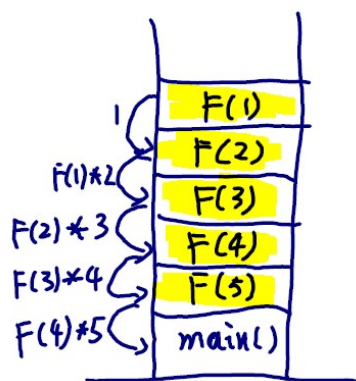


```
int FactorialTail(int n, int total = 1)
{
    if (n <= 1)
    {
        return total;
    }
    return FactorialTail(n-1, n * total);
}
```

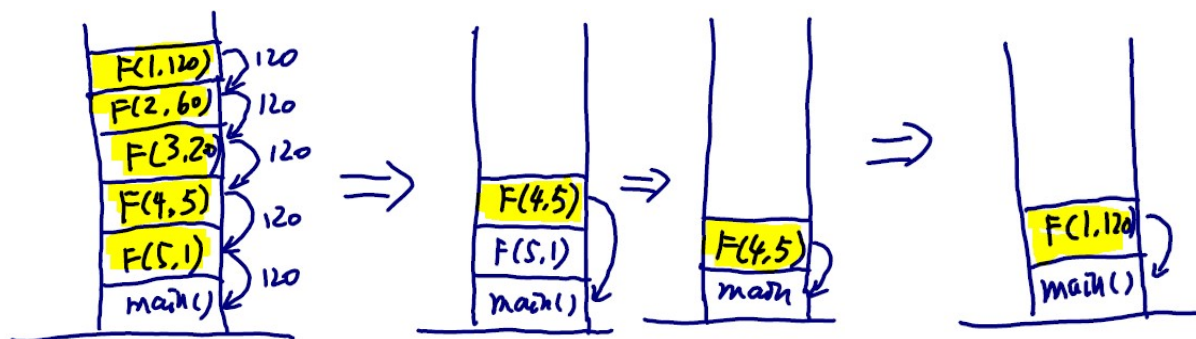
```
int FactorialIterative(int n)
{
    int total = 1;
    for (int i = n; i >= 1; i--)
    {
        total = i * total;
    }
    return total;
}
```

尾递归栈调用优化

- 正常递归



- 尾递归



Quicksort

```
void SortedIntArray::QuickSort(int start, int end)
{
    if (start >= end)
    {
        return;
    }

    int pivot = PartitionEnd(start, end);
    QuickSort(start, pivot - 1);
    QuickSort(pivot + 1, end);
}
```

Inplace, $O(1)$ 额外空间

并不是真正的 $O(1)$ 空间，因为还有call stack
最糟情况也需要 $O(n)$ 栈空间，而且不是堆空间

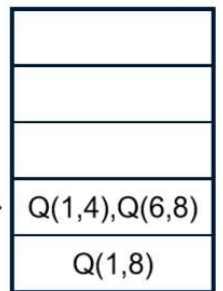
1	3	7	8	2	6	4	5
---	---	---	---	---	---	---	---

↓ after partition

1	3	2	4	5	6	7	8
---	---	---	---	---	---	---	---

Qsort(A,1,4) Qsort(A,6,8)

STACK



Quicksort尾递归优化

```
void SortedIntArray::QuickSort_Better(int start, int end)
{
    while (start < end)
    {
        int pivot = PartitionEnd(start, end);
        QuickSort(start, pivot - 1);

        start = pivot + 1;
    }
}
```

1	2	3	4	5	6	7	8
---	---	---	---	---	---	---	---

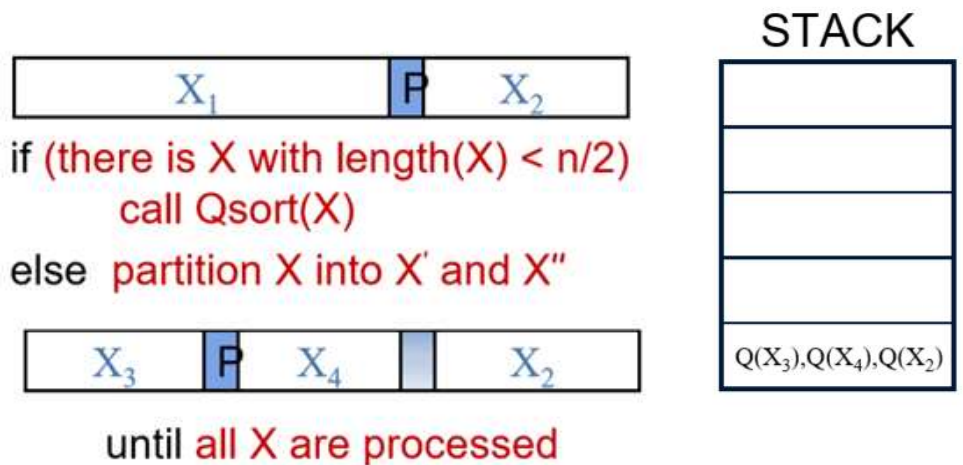
STACK size =
 $O(n)$ entries

STACK
Q(1,1)
...
Q(1,6)
Q(1,7)
Q(1,8)

Quicksort尾递归优化

```
void SortedIntArray::QuickSort_Best(int start, int end)
{
    while (start < end)
    {
        int pivot = PartitionEnd(start, end);

        if (pivot - start < end - pivot)
        {
            QuickSort(start, pivot - 1);
            start = pivot + 1;
        }
        else
        {
            QuickSort(pivot + 1, end);
            end = pivot - 1;
        }
    }
}
```



保证最差情况下需要 $O(\log_2 n)$ 栈空间

Quicksort迭代实现

- 用栈来模拟递归调用
- 占用了 $O(n)$ 空间，不过可以是堆空间
- 去掉了递归，效率更高

```
void SortedIntArray::QuickSort_iterative(int start, int end)
{
    // Create an auxiliary stack
    stack<int> stk;

    // push initial values of start and end to stack
    stk.push(start);
    stk.push(end);

    // Keep popping from stack while is not empty
    while (!stk.empty())
    {
        // Pop end and start
        end = stk.top();
        stk.pop();
        start = stk.top();
        stk.pop();

        // Set pivot element at its correct position
        // in sorted array
        int pivot = PartitionEnd(start, end);

        // If there are elements on left side of pivot,
        // then push left side to stack
        if (pivot - 1 > start)
        {
            stk.push(start);
            stk.push(pivot - 1);
        }

        // If there are elements on right side of pivot,
        // then push right side to stack
        if (pivot + 1 < end)
        {
            stk.push(pivot + 1);
            stk.push(end);
        }
    }
}
```


总结

- 递归能使本质问题浮现
- 有不少情况下，用递归更好
- 得注意递归的问题，栈空间问题会制约递归的规模
- 一般来说，如果非递归方式不是特别麻烦的话，最好采用非递归方式，效率更高，规模更大
- 后面要讲的树的遍历，需要用迭代的方法，递归的规模不能解数目大的树。

Q&A

Thanks!