

1、递归

递归：函数（方法）直接或间接调用自身。是一种常用的编程技巧。

严格来说递归不算一种编程思想，其实最多算是一种编程技巧！！

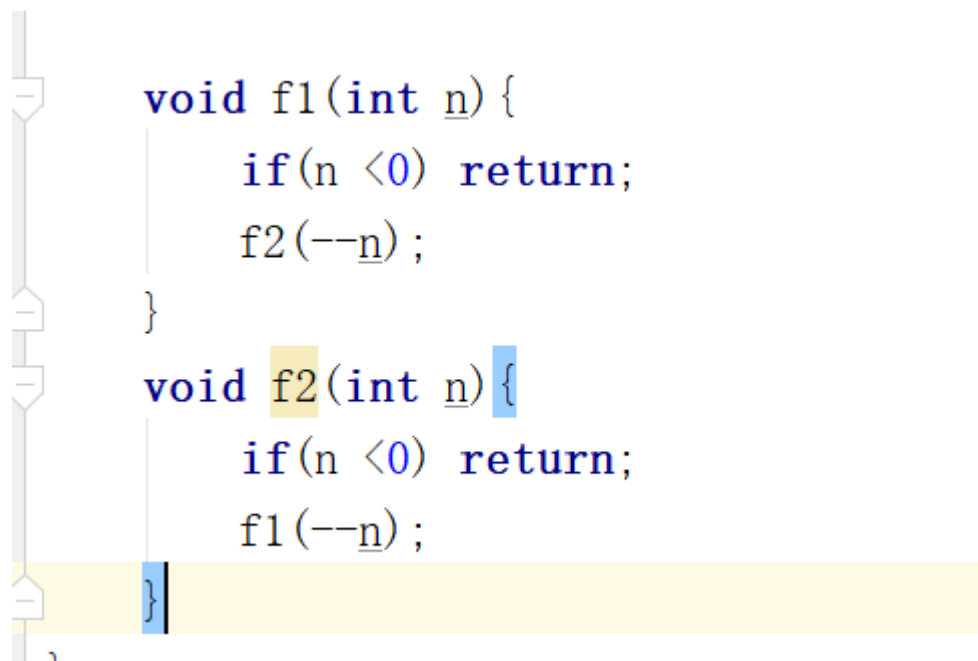
第 1 节 常见递归调用

1.1 编程递归现象

直接调用

```
/**
 * 计算数值累加
 * @param n
 * @return
 */
int sum(int n) {
    if(n <=1) return n;
    return n+sum(n-1);
}
```

间接调用



```

void f1(int n) {
    if(n < 0) return;
    f2(--n);
}

void f2(int n) {
    if(n < 0) return;
    f1(--n);
}

```


1.2 生活递归现象

- 从前有座山，山里有座庙，庙里有个老和尚，正在给小和尚讲故事呢！故事是什么呢？
 【从前有座山，山里有座庙，庙里有个老和尚，正在给小和尚讲故事呢！故事是什么呢？
 『从前有座山，山里有座庙，庙里有个老和尚，正在给小和尚讲故事呢！故事是什么呢？……』
 】
- 假设你在一个电影院，想知道自己坐在哪一排，但是前面人很多，
 - m1 懒得数，便问前一排的人 m2【你坐在哪一排？】，只要把 m2 的答案加一，就是 m1 的排数。
 - m2 懒得数，便问前一排的人 m3【你坐在哪一排？】，只要把 m3 的答案加一，就是 m2 的排数。
 - m3 懒得数，便问前一排的人 m4【你坐在哪一排？】，只要把 m4 的答案加一，就是 m3 的排数。
 - …… 直到问到最前面的一排，最后大家都知道自己在哪一排了

第 2 节 调用过程分析

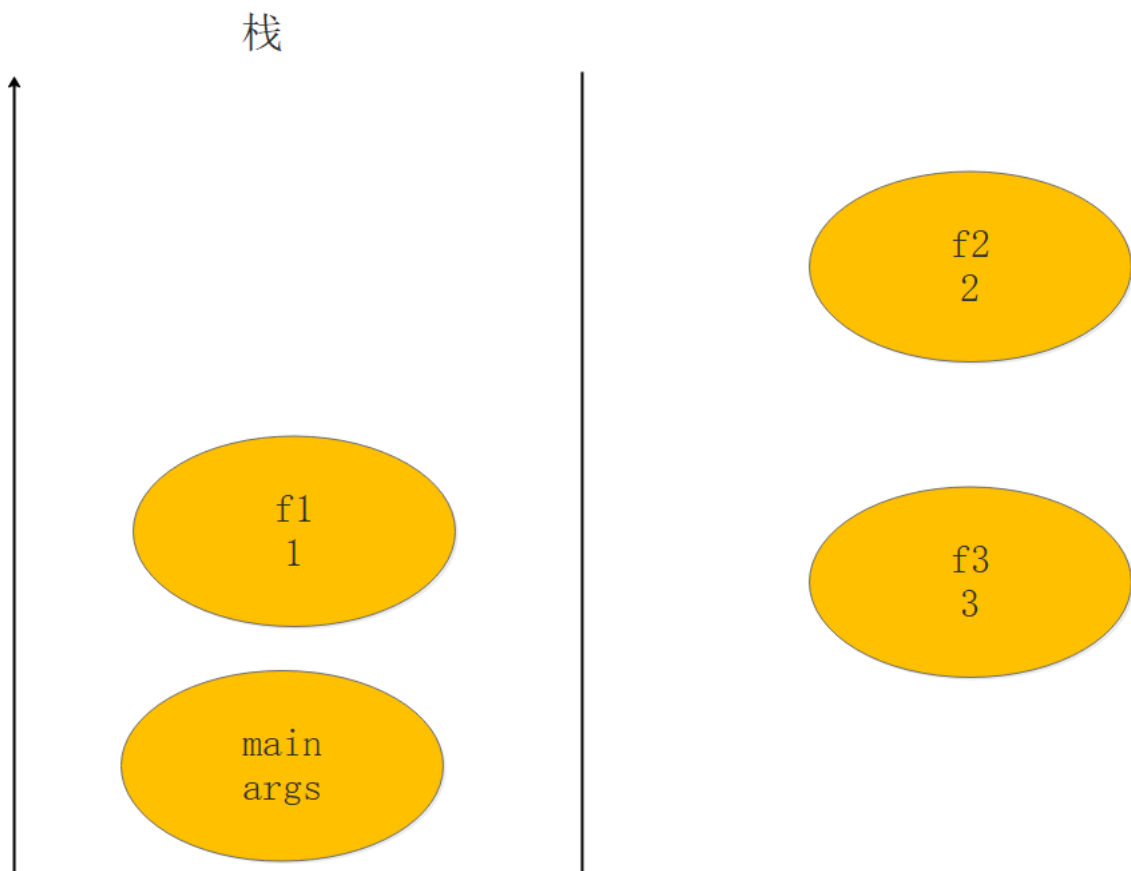
2.1 函数调用过程

- 普通函数调用过程



```
public static void main(String[] args) {  
    f1(n: 1);  
    f2(n: 2);  
}  
  
static void f1(int n) {  
}  
  
static void f2(int n) {  
    f3(n: 3);  
}  
  
static void f3(int n) {  
}
```

栈空间图示



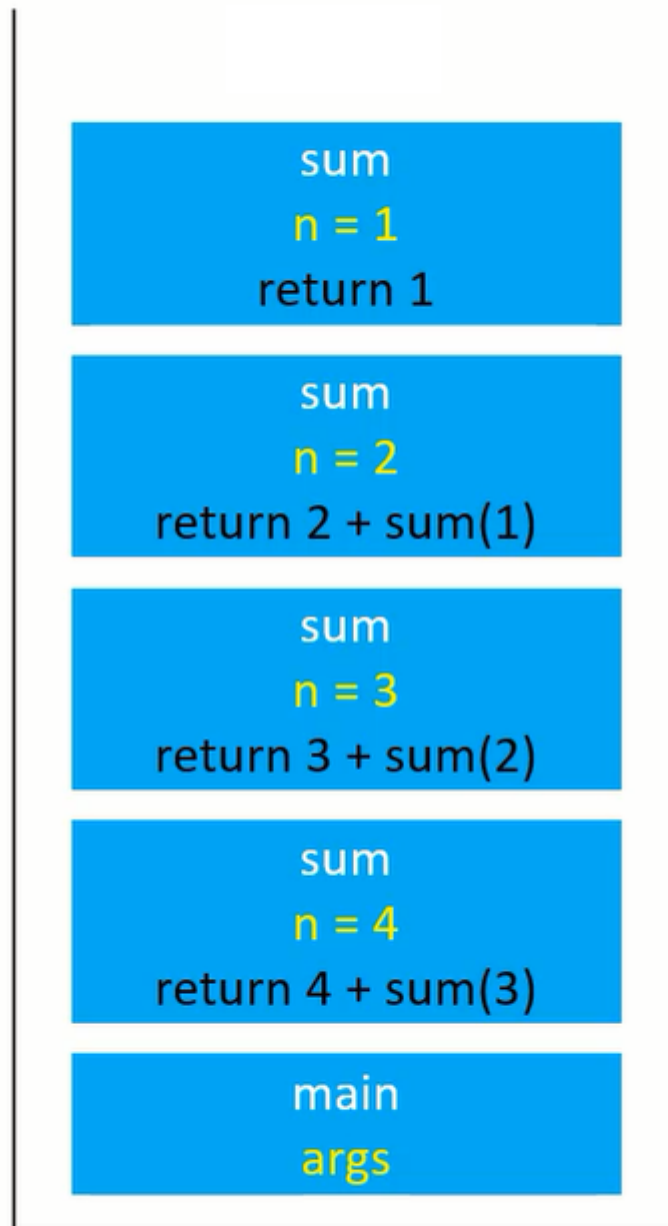
栈空间其实就是内存保存的是当前调用方法的一些局部变量！！

- 递归调用过程

```
public static void main(String[] args) {  
    sum(4);  
}  
static int sum(int n){  
    if(n <=1) return n;  
    return n+sum(n-1);  
}
```

栈空间示意图

栈



空间复杂度是 $O(n)$

为什么递归调用我们考虑空间复杂度，因为在执行最后一步之前，其余调用都没有释放！！所以是占用这栈的内存空间！！

备注：普通函数调用为何不讨论空间复杂度？（ $O(1)$ ）常数级别！！

Stack Overflow

如果递归调用过程一直没有终止，则栈空间会被一直占用并不断消耗宝贵的空间资源！！最终导致导致栈内存溢出 (Stack Overflow);

递归基

结合上面的分析，如果要进行递归调用，一定要考虑明确一个结束递归的条件，这个条件一般被称为边界条件或者递归基。

2.2 例题--累加和

计算 $1+2+3+4+\dots+(n-1)+n$ 的和, ($n > 0$)

- 方式1

```
public static void main(String[] args) {
    sum(4);
}

static int sum(int n){
    if(n <=1) return n;
    return n+sum(n-1);
}
```

分析复杂度

时间复杂度: $O(n)$

空间复杂度: $O(n)$

备注

空间复杂度: $O(n)$,先不讨论对堆空间的占用, 比如在方法中调用new 方式创建对象。

- 方式2

```
public static void main(String[] args) {
    sum(4);
}

static int sum(int n){
    int res=0;
    for (int i = 1; i <= n; i++) {
        res+=i;
    }
    return res;
}

static int sum1(int n){
    if(n <=1) return n;
    return n+sum(n-1);
}
```

复杂度分析

时间复杂度: $O(n)$

空间复杂度: $O(1)$

- 方式3

```
public static void main(String[] args) {  
    System.out.println(sum(4));  
    System.out.println(sum1(4));  
  
}  
static int sum(int n){  
    if(n <=1){ return n;}  
    return ((n+1)*n)/2;  
}
```

复杂度分析

时间复杂度: $O(1)$

空间复杂度: $O(1)$

总结

以上3种方式都能实现，但是从复杂度分析我们知道第三种方式显然复杂度更低，所以第三种是对前两种的优化和效率的提升！！

思考

递归效率不高，为何还使用递归？

使用递归往往不是为了求得最优解，是为了简化解问题的思路，代码会更简洁！！

2.3 评价算法

2.3.1 概述

算法的定义


算法是用来解决特定问题的一系列的执行步骤

对于同一个问题可以多种解决方式，也就是同一个问题可以有多种算法可以解决，那我们不禁要问：

- 不同的算法她们的效率一样吗？

答案：使用不同的算法解决同一个问题，效率可能相差非常大！！

2.3.2 如何评价算法质量



```
}  
  
static int sum(int n) {  
    if(n <=1) { return n;}  
    return ((n+1)*n)/2;  
}  
  
static int sum1(int n) {  
    int res=0;  
    for (int i = 1; i <= n; i++) {  
        res+=i;  
    }  
    return res;  
}
```

上面是计算累加和的案例，使用了两种不同的方式实现，我们如何评价两种方案的执行效率呢？

- 比较不同算法对同一组输入数据的处理时间，被称为事后统计法！！

自己编写时间计算工具测算不同算法执行时间


```

public static void main(String[] args) {
    Times.test(name: "sum", new Times.Task() {
        public void execute() {
            System.out.println(sum(n: 4000000));
        }
    });
    Times.test(name: "sum1", new Times.Task() {
        public void execute() {
            System.out.println(sum1(n: 4000000));
        }
    });
}

```

Times.java

```

package com.lg.utils;

import java.text.SimpleDateFormat;
import java.util.Date;

public class Times {
    private static final SimpleDateFormat fmt = new SimpleDateFormat("HH:mm:ss.SSS");

    public interface Task {
        void execute();
    }

    public static void test(String name, Task task) {
        if (task == null) return;
        name = (name == null) ? "" : ("[" + name + "]");
        System.out.println(name);
        System.out.println("开始: " + fmt.format(new Date()));
        long begin = System.currentTimeMillis();
        task.execute();
        long end = System.currentTimeMillis();
        System.out.println("结束: " + fmt.format(new Date()));
        double delta = (end - begin) / 1000.0;
        System.out.println("耗时: " + delta + "秒");
        System.out.println("-----");
    }
}

```

```

public static void main(String[] args) {
    Times.test(name: "sum", new Times.Task() {
        public void execute() {
            System.out.println(sum(n: 4000000));
        }
    });
    Times.test(name: "sum1", new Times.Task() {
        public void execute() {
            System.out.println(sum1(n: 4000000));
        }
    });
}

```

事后统计法缺点

- 执行时间取决于硬件以及运行时各种不确定的环境因素(开启很多软件与没开启很多软件)
- 必须编写测试代码
- 测试数据的选择比较难保证公正性(有些算法数据量小可能快，大了就不行，有些相反)

通常从以下维度来评估算法的优劣

- 正确性、可读性、健壮性（对不合理输入的反应能力和处理能力）
- 时间复杂度（Time complexity）：估算程序指令的执行次数（执行时间）
- 空间复杂度（Space complexity）：估算所需占用的存储空间

2.3.3 大O表示法(Big O)

大O表示法的定义：一般用大O表示法来描述算法复杂度，它表示的是算法对于数据规模 n 的复杂度(时间复杂度，空间复杂度)

示例

估算程序指令的执行次数，以分号为准，判断语句不算！！，

```

package com.lg.dynamic;

public class Demo {

```

```

/* 0 1 2 3 4 5
 * 0 1 1 2 3 5 8 13 ....
 */

// O(2^n)
public static int f1(int n) {
    if (n <= 1) return n;
    return f1(n - 1) + f1(n - 2);
}

// O(n)
public static int f2(int n) {
    if (n <= 1) return n;

    int first = 0;
    int second = 1;
    for (int i = 0; i < n - 1; i++) {
        int sum = first + second;
        first = second;
        second = sum;
    }
    return second;
}

public static int f3(int n) {
    if (n <= 1) return n;

    int first = 0;
    int second = 1;
    while (n-- > 1) {
        second += first;
        first = second - first;
    }
    return second;
}

public static void main(String[] args) {
    int n = 12;

    System.out.println(f2(n));
    System.out.println(f3(n));

    // TimeTool.check("f1", new Task() {
    //     public void execute() {
    //         System.out.println(f1(n));
    //     }
    // });

    // TimeTool.check("f2", new Task() {
    //     public void execute() {
    //         System.out.println(f2(n));
    //     }
    // });

```

```

//    });
}

public static void m1(int n) {
    // 汇编指令

    // 1
    if (n > 10) {
        System.out.println("n > 10");
    } else if (n > 5) { // 2
        System.out.println("n > 5");
    } else {
        System.out.println("n <= 5");
    }

    // 1 + 4 + 4 + 4
    //空间复杂度
    //只定义了一个变量i, 所以是o(1)
    for (int i = 0; i < 4; i++) {
        System.out.println("m");
    }

    // 140000
    // o(1)
    // o(1)
}

//空间复杂度 所以是o(1)
public static void m2(int n) {
    // o(n)
    // 1 + 3n
    for (int i = 0; i < n; i++) {
        System.out.println("m");
    }
}

public static void m3(int n) {
    // 1 + 2n + n * (1 + 3n)
    // 1 + 2n + n + 3n^2
    // 3n^2 + 3n + 1
    // o(n^2)

    // o(n)
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            System.out.println("m");
        }
    }
}

public static void m4(int n) {
    // 1 + 2n + n * (1 + 45)
    // 1 + 2n + 46n
}

```

```

// 48n + 1
// O(n)
for (int i = 0; i < n; i++) {
    for (int j = 0; j < 15; j++) {
        System.out.println("m");
    }
}

}

public static void m5(int n) {
    // 8 = 2^3
    // 16 = 2^4

    // 3 = log2(8)
    // 4 = log2(16)

    // 执行次数 = log2(n)
    // O(logn)
    while ((n = n / 2) > 0) {
        System.out.println("m");
    }
}

public static void m6(int n) {
    // log5(n)
    // O(logn)
    while ((n = n / 5) > 0) {
        System.out.println("m");
    }
}

public static void m7(int n) {
    // 1 + 2*log2(n) + log2(n) * (1 + 3n)

    // 1 + 3*log2(n) + 2 * nlog2(n)
    // O(nlogn)
    for (int i = 1; i < n; i = i * 2) {
        // 1 + 3n
        for (int j = 0; j < n; j++) {
            System.out.println("m");
        }
    }
}

// 空间复杂度是int[] array = new int[n]; O(n)
public static void m10(int n) {
    // O(n)
    int a = 10;
    int b = 20;
    int c = a + b;
    int[] array = new int[n];
    for (int i = 0; i < array.length; i++) {
        System.out.println(array[i] + c);
    }
}

```

```
}  
}  
}
```

以上估算完还是很复杂

大O表示法的规则：

- 忽略常数、系数、低阶
 - $9 \rightarrow O(1)$
 - $2n + 3 \rightarrow O(n)$
 - $n^2 + 2n + 6 \rightarrow O(n^2)$
 - $4n^3 + 3n^2 + 22n + 100 \rightarrow O(n^3)$
 - 对数阶一般省略底数 $\log_2^n = \log_2^9 * \log_9^n$ 所以 \log_2^n 、 \log_9^n 统称为 \log
 - 写法上, n^3 等价于 $n^{\wedge}3$
- 注意：大O表示法仅仅是一种粗略、近似的分析模型，是一种估算，能帮助我们快速了解一个算法的执行效率

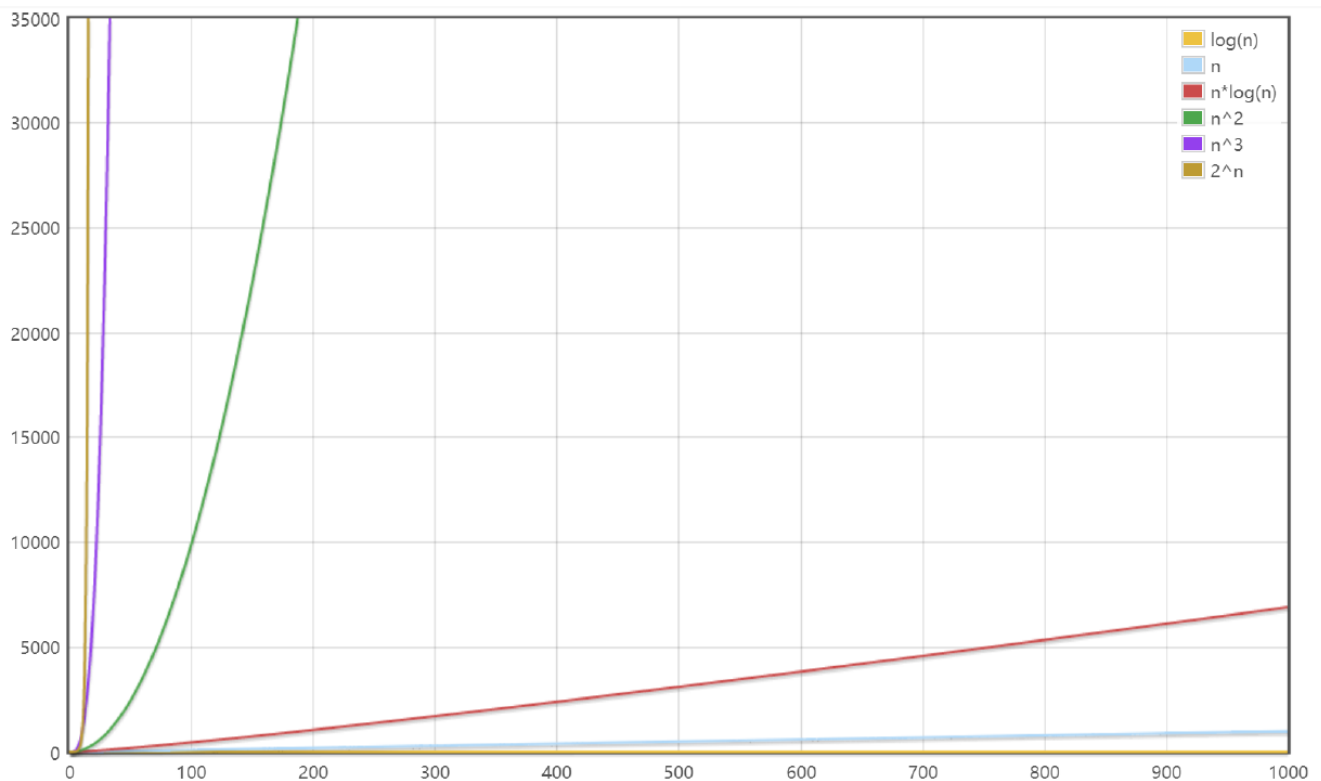
常见复杂度

执行次数	复杂度	术语
120	$O(1)$	常数阶
$3n + 3$	$O(n)$	线性阶
$10n^2 + 2n + 60$	$O(n^2)$	平方阶
$40\log_2 n + 250$	$O(\log n)$	对数阶
$30n + 3n\log_3 n + 150$	$O(n\log n)$	$n\log n$ 阶
$n^3 + 2n^2 + 2n + 10$	$O(n^3)$	立方阶
2^n	$O(2^n)$	指数阶

时间复杂度的优先级

$O(1) < O(\log n) < O(n) < O(n\log n) < O(n^2) < O(n^3) < O(2^n) < O(n!) < O(nn)$

复杂度曲线图



备注：图片来自网络

空间复杂度

空间复杂度是估算大概占用的存储空间

计算m1,m2,m10为例，最后总结：对于当前的算法程序来说我们更关注的时间复杂度，因为现在硬件资源都很充足！！

补充：注意大O表示法是估算，所谓的执行次数并不是我们的代码执行次数，因为代码最终都要转为汇编的指令，可能就不仅仅是我们代码的执行次数了。但是这种差异大家都一样而且是常数级别，所以不影响大O表示法的最终结果！！

2.3.4 估算斐波那契实现复杂度

```
// o(2^n):借助于调用过程
//o(n)
public static int f1(int n) {
    if (n <= 1) return n;
    return f1(n - 1) + f1(n - 2);
}

// o(n)
//o(1)
public static int f2(int n) {
    if (n <= 1) return n;

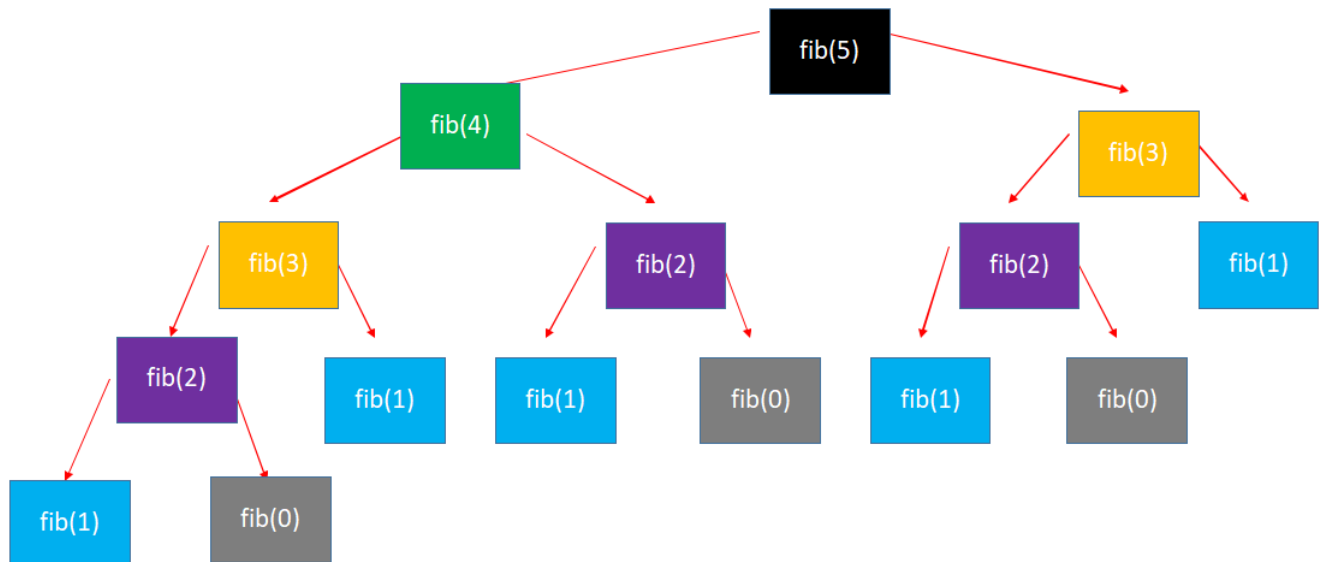
    int first = 0;
    int second = 1;
    for (int i = 0; i < n - 1; i++) {
        int sum = first + second;
        first = second;
    }
}
```

```

        second = sum;
    }
    return second;
}

```

fib递归调用过程分析



- $1 + 2 + 4 + 8 = 20 + 21 + 22 + 23 = 24 - 1 = 2^n - 1 = 0.5 * 2^n - 1$
- 时间复杂度是 $O(2^n)$

2.3.5 算法优化角度

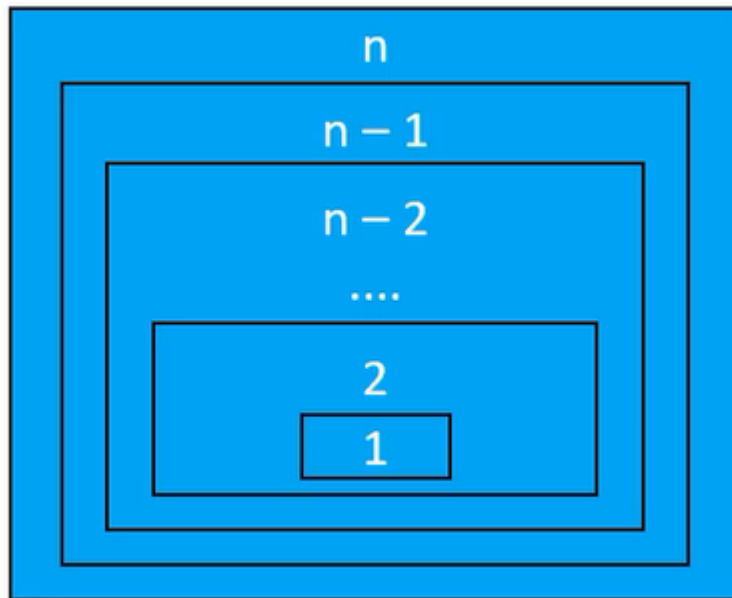
- 尽可能少的存储空间
- 尽可能少的执行时间
 - 根据情况，可以 空间换时间
 - 时间换 空间

补充: Leetcode

<https://leetcode.com/>
<https://leetcode-cn.com/>

第 3 节 递归基本思想

递归的基本思想就是拆解问题，通过下图理解递归的思想



- 拆分问题
 - 把规模大的问题编程规模较小的同类型问题
 - 规模较小的问题又不断变成规模更小的问题
 - 规模小到一定程度就可以直接得到结果
- 求解
 - 由最小规模问题的解推导出较大规模问题的解
 - 由较大规模问题的解不断推导出规模更大问题的解
 - 最后推导出原来问题的解

总结：

只要问题符合上述描述也就是可以拆解问题和求解，可以尝试使用递归解决！！

第4 节递归使用步骤与技巧

4.1 确定函数的功能

第一步先不要思考函数里面代码逻辑如何实现，先搞清楚这个函数的目的，完成什么事情？

4.2 确定子问题与原问题的关系

找到 $f(n)$ 与 $f(n-1)$ 的关系

$$f(n)=f(n-1)+n$$

4.3 明确递归基(边界条件)

- 递归的过程中，问题的规模在不断减小，当问题缩减到一定程度便可以直接得出它的解
- 寻找递归基，等价于：问题规模小到什么程度可以直接得出解？

```
/**
 * 计算1+2+3+4+5。。。+(n-1)+n的值 确定函数的功能
 * @param n
 * @return
 */
static int sum(int n) {
    if(n <=1) return n; 明确递归基
    return n+sum(n-1); 找到原问题与子问题的关系
}
```

第5节 递归练习

5.1 斐波那契数列

斐波那契数列：1、1、2、3、5、8、13、21、34、.....

$F(1)=1, F(2)=1, F(n)=F(n-1)+F(n-2) (n \geq 3)$

- 编写一个函数求第 n 项斐波那契数

```
public static int f1(int n) {
    if (n <= 1) return n;
    return f1(n - 1) + f1(n - 2);
}
```

根据递推式 $T_n = T(n-1) + T(n-2) + O(1)$ ，可得知

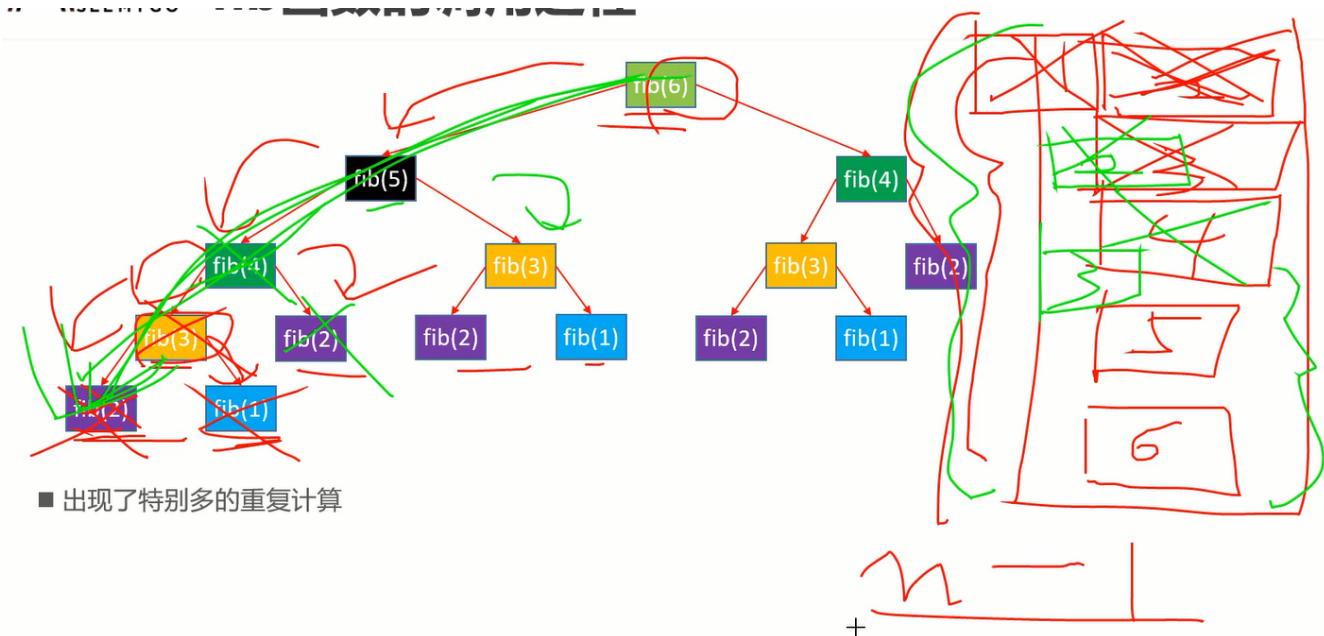
- 时间复杂度： $O(2^n)$
- 空间复杂度： $O(n)$
- 递归调用的空间复杂度 = 递归深度 * 每次调用所需的辅助空间

是不是所有递归调用都是 $O(n)$ 级别的空间复杂度呢？

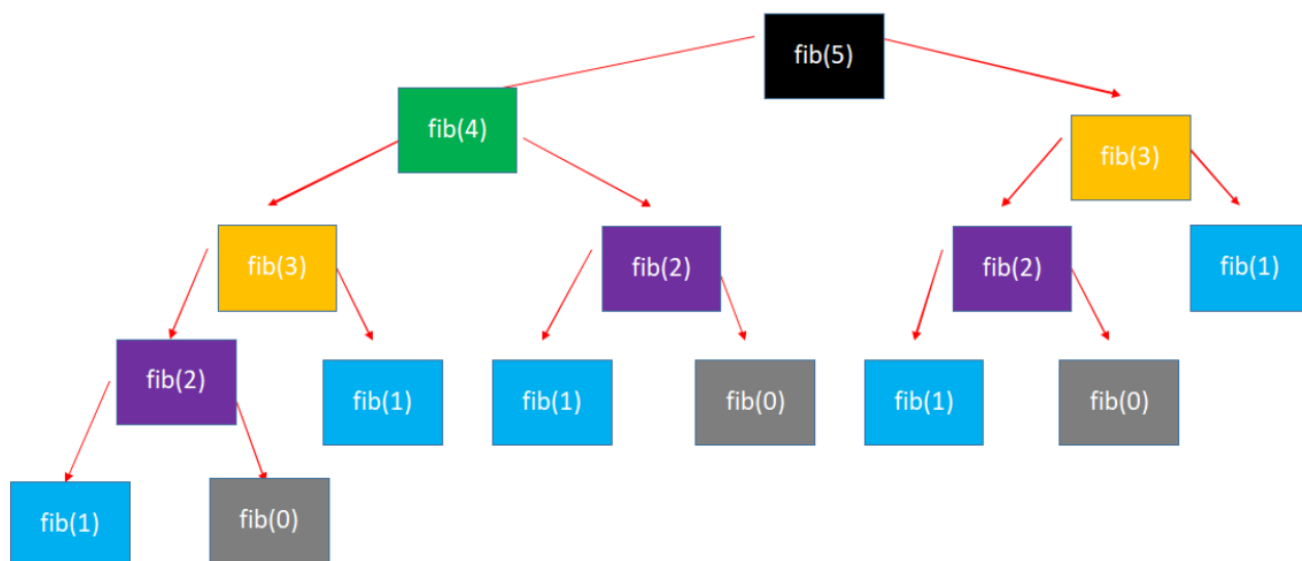
```
public static int f1(int n) {
    if (n <= 1) return n;
    return f1(n-1) + f1(n-2);
}
```

左边的先执行完才会开始右边的

递归深度借助调用过程



1、优化1 (使用数组)



时间复杂度高的原因：出现了很多重复计算，可以认为是一种自顶向下的调用过程！！

优化方案：避免掉重复计算，之前计算过的值不要再次计算！！

思路：使用数组存放之前的计算结果，正好利用索引！！

```
package com.lg.reursion;

public class Demo1 {
    public static void main(String[] args) {
        system.out.println(f1(1));
    }

    static int f(int n){
        if(n <=2){ return 1;}
    }
}
```

```

        return f(n-1)+f(n-2);
    }

    /**
     * 利用数组存储之前计算的值
     * @return
     */
    static int f1(int n){
        if(n <=2){ return 1;}
        int[] arr=new int[n+1]; //+1是为了方便索引对应，无其它含义
        //初始化值
        arr[1]=arr[2]=1;
        return f2(n-1,arr)+f2(n-2,arr);
    }

    static int f2(int n,int[] arr){

        //计算时判断数组中是否已经有值了，如果有不重复计算直接获取
        if(arr[n]==0){
            arr[n]=f2(n-1,arr)+f2(n-2,arr);
        }
        return arr[n];
    }
}

```

复杂度分析

时间复杂度：O(n)

空间复杂度：O(n)

2、优化2(去除递归)

分析使用数组的调用过程，发现只有第一次的时候需要计算出来，后续其实都无需再计算，同时发现另外一个问题：

递归调用是自顶向下，如果改为自下向上调用是否可以呢？

再次理解原问题与子问题的关系

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

如果先计算出小问题然后再累加求出大问题的解是否可以呢？答案是可以的！！

```

/**
 * 去除递归调用
 * @return
 */
static int f1(int n) {
    if(n <=2) { return 1;}
    int[] arr=new int[n+1];//+1是为了方便索引对应，无其它含义
    arr[1]=arr[2]=1;
    for (int i = 3; i <=n; i++) {
        arr[i]=arr[i-1]+arr[i-2];
    }
    return arr[n];
}

```

```

/**
 * 去除递归调用
 * @return
 */
static int f1(int n){
    if(n <=2){ return 1;}
    int[] arr=new int[n+1];//+1是为了方便索引对应，无其它含义
    arr[1]=arr[2]=1;
    for (int i = 3; i <=n; i++) {
        arr[i]=arr[i-1]+arr[i-2];
    }
    return arr[n];
}

```

复杂度分析

时间复杂度: $O(n)$

空间复杂度: $O(n)$

3、优化3(借助变量)

分析第二版代码的计算过程，会发现在计算arr[i]的值时只需要知道前面两个的值即可，其余值已经没有了意义，所以数组的大部分空间很多时候都是没有意义的！！

解决方式：去掉数组使用2个变量记录i前面的两个值即可！！

代码实现

```

package com.lg.recursion;

public class Demo3 {
    public static void main(String[] args) {
        System.out.println(f1(3));
        System.out.println(f(3));
    }

    static int f(int n){
        if(n <=2){ return 1;}
        return f(n-1)+f(n-2);
    }

    /**
     * 去除递归调用
     * @return
     */
    static int f1(int n){
        if(n <=2){ return 1;}
        int first=1;
        int second=1;
        for (int i = 3; i <=n; i++) {

            second=first+second;
            first=second-first;
        }
        return second;
    }
}

```

复杂度分析

时间复杂度：O(n)

空间复杂度：O(1)

4、优化4(公式)

斐波那契数列有个线性代数解法：特征方程

$$F(n) = \frac{1}{\sqrt{5}} \left[\left(\frac{1 + \sqrt{5}}{2} \right)^n - \left(\frac{1 - \sqrt{5}}{2} \right)^n \right].$$

代码实现

```

package com.lg.recursion;

```

```

public class Demo4 {
    public static void main(String[] args) {
        System.out.println(f1(5));
        System.out.println(f(5));

    }

    static int f(int n){
        if(n <=2){ return 1;}
        return f(n-1)+f(n-2);
    }

    /**
     * 去除递归调用
     * @return
     */
    static int f1(int n){
        double c=Math.sqrt(5);
        return (int)((Math.pow((1+c)/2,n )-Math.pow((1-c)/2,n ))/c);
    }

}

```

复杂度分析：时间复杂度、空间复杂度取决于 pow 函数（至少可以低至 $O(\log n)$ ）

备注：注意并不是所有算法都有公式可以直接使用，这是个特例！！

5.2 跳台阶

题目：楼梯有 n 阶台阶，上楼可以一步上 1 阶，也可以一步上 2 阶，走完 n 阶台阶共有多少种不同的走法？



步骤一

定义一个函数 $f(n)$

函数的功能：函数的功能就是返回 n 级台阶共多少种走法；

步骤二

确定原问题与子问题的解

- 假设第一次走 1 阶，还剩 $n - 1$ 阶，共 $f(n - 1)$ 种走法
- 假设第一次走 2 阶，还剩 $n - 2$ 阶，共 $f(n - 2)$ 种走法

同理反过来推导也可以的！！

所以 $f(n)=f(n-1)+f(n-2)$ (眼熟？斐波那契)

步骤三

初始值

$f(1)=1$

$f(2)=2$

与斐波那契数列非常相似，只是初始值不同，大家实现下这个需求，优化思路也是一样！！

5.3 汉诺塔 (Hanoi)

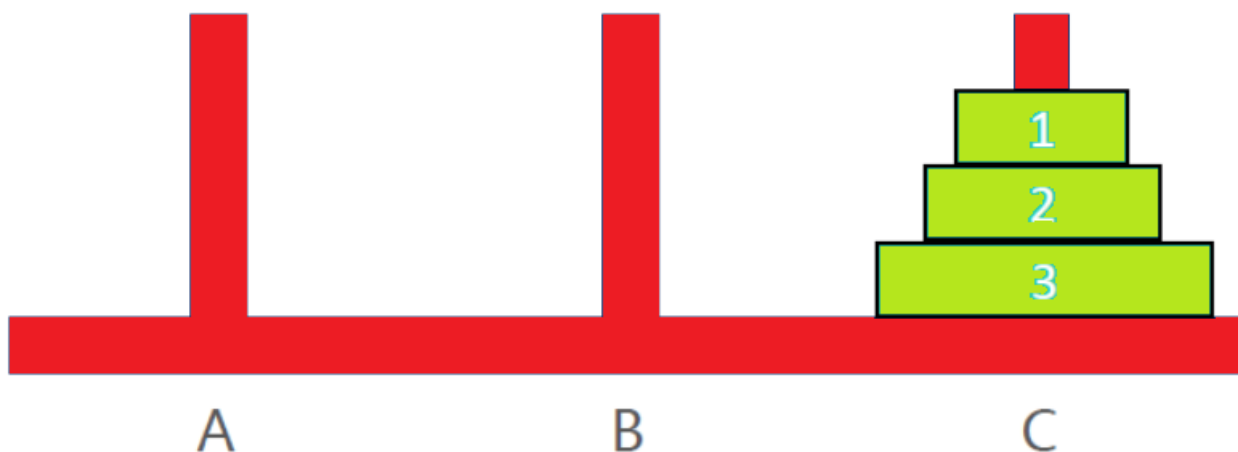
1、题目

编程实现把 A 的 n 个盘子移动到 C (盘子编号是 $[1, n]$)



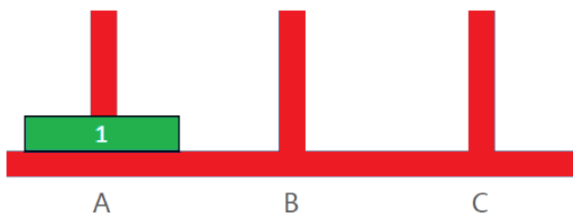
移动要求：

- 每次只能移动 1 个盘子
- 大盘子必须放在小盘子下面(挪动过程中 也是如此)

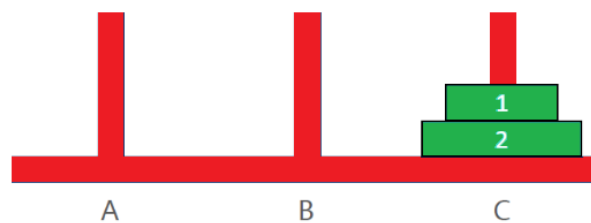
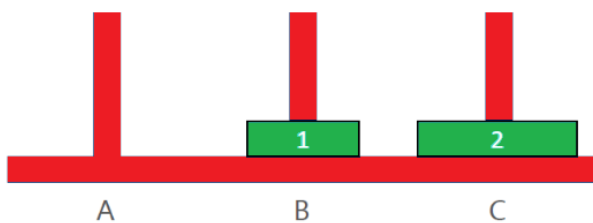
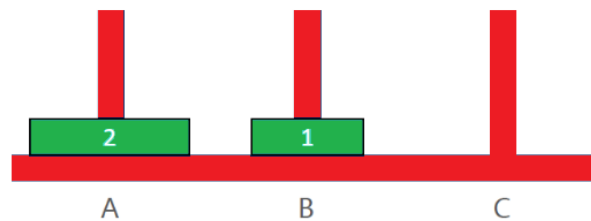


2、问题分析

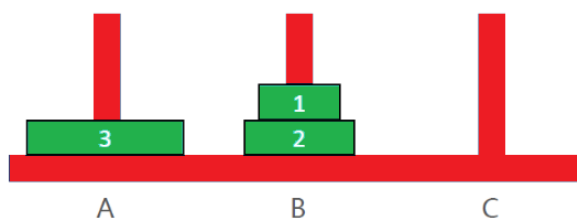
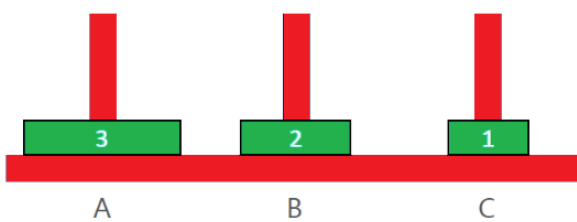
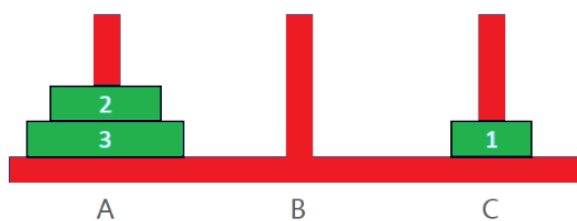
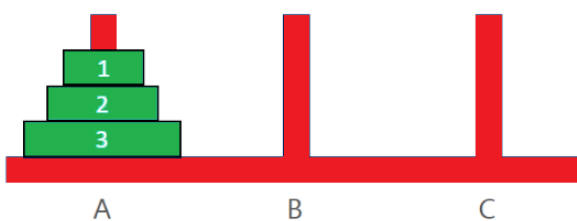
挪动1个盘子

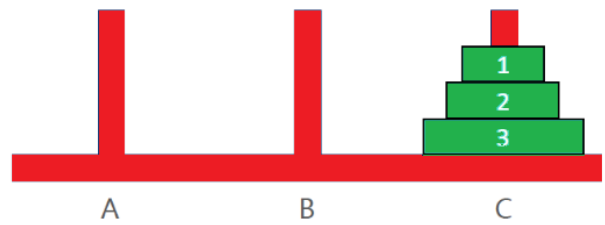
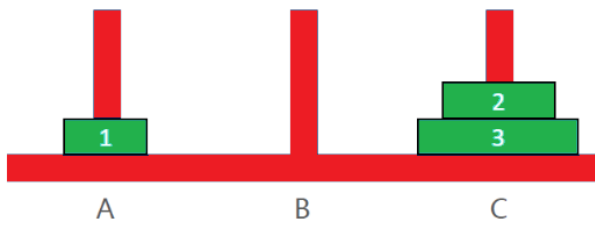
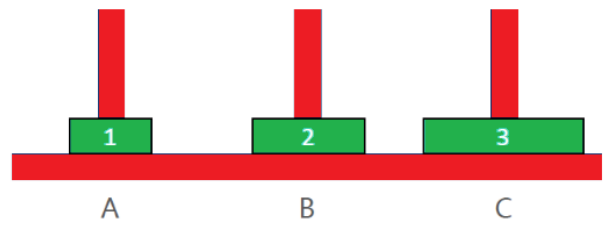
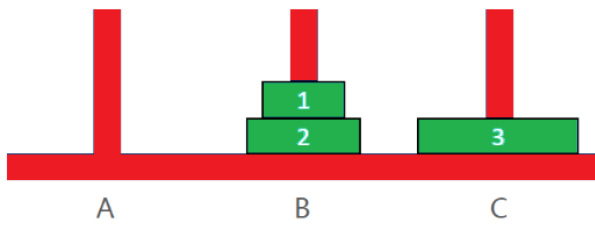


挪动2个盘子



挪动3个盘子

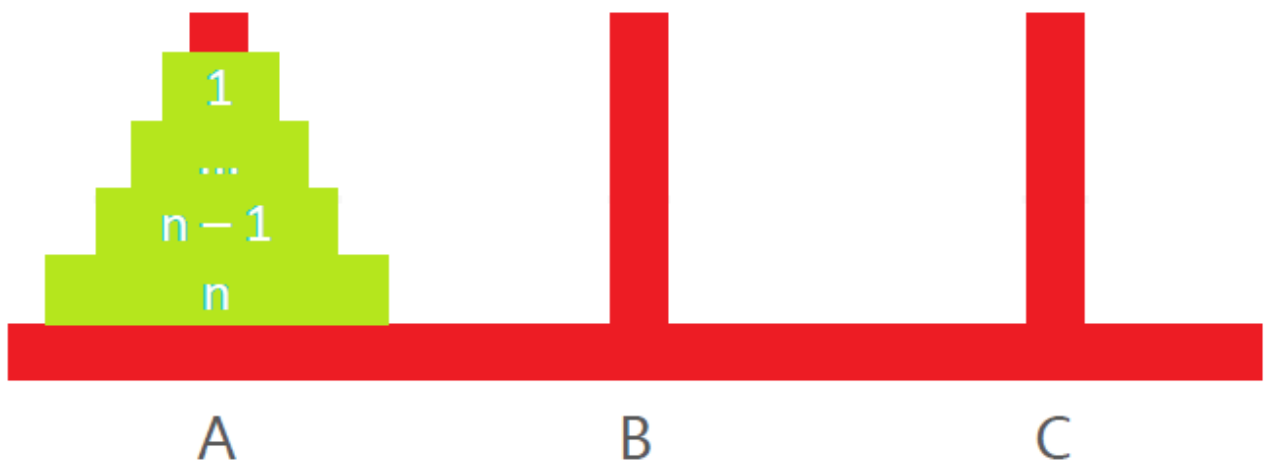


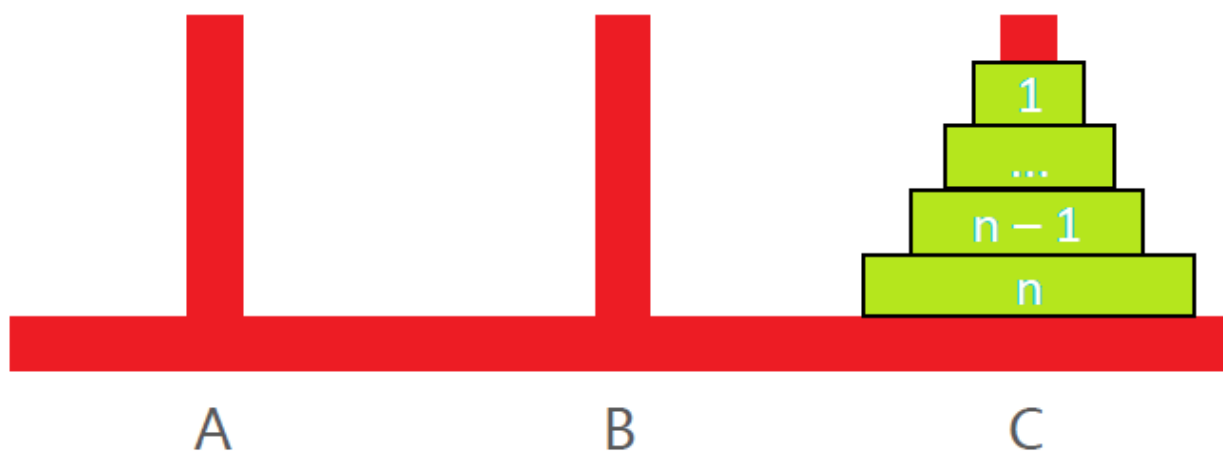
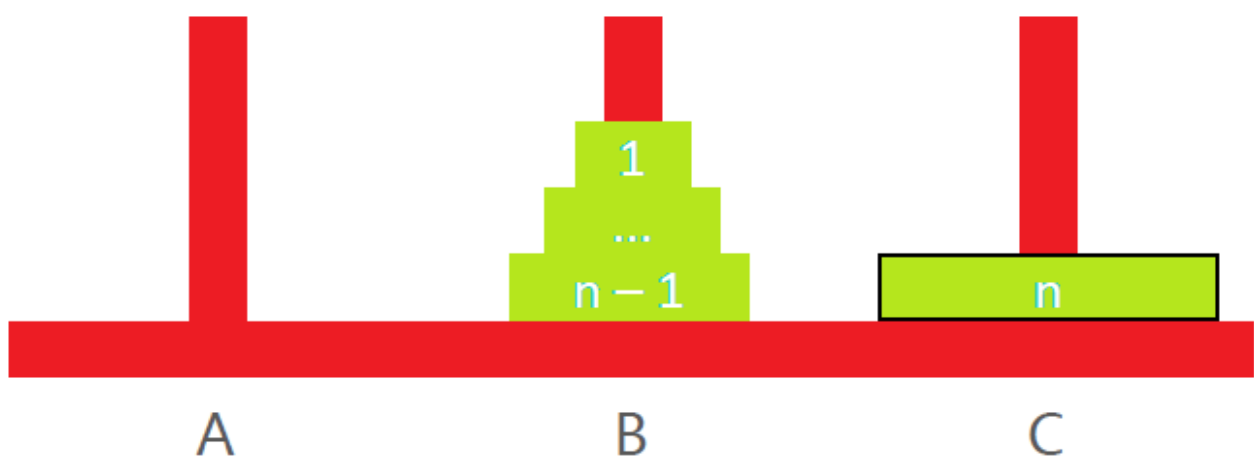
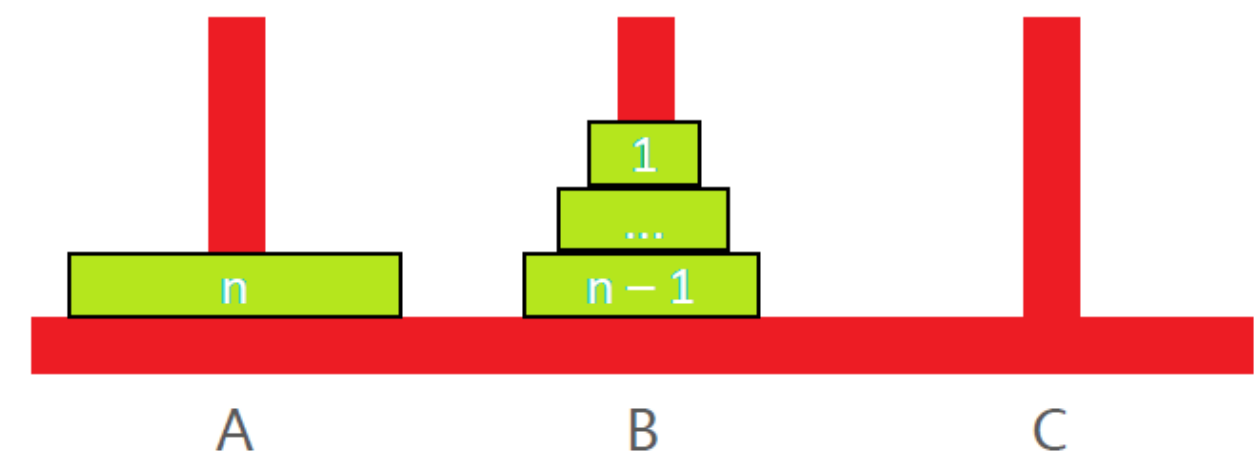


必须将 $n-1$ 个盘子移动到B柱上！！

- 如果只有一个盘子， $n=1$ 时，直接将盘子从A移动到C；
- 如果不止一个盘子， $n \geq 2$ 时，其实可以分为三步
 - 将 $n-1$ 个盘子从A移动到B
 - 将编号为 n 的盘子从A移动到C
 - 将 $n-1$ 个盘子从B移动到C

规律：第一步和第三部是相同的！！





代码实现

```
package com.lg.hanoi;  
  
public class Hanoi {
```

```

public static void main(String[] args) {
    String a="A";
    String b="B";
    String c="C";

    moveHanoi(3, a, b, c);
}
/**
 *
 * 将n个碟子从A挪动到C,B是中间柱子
 */
static void moveHanoi(int n, String a, String b, String c) {

    //确定递归基, 判断代码中值的边界, 小技巧
    if (n == 1) {
        printInfo(n, a, c);
        return;
    }
    //1 将n-1个盘子从A移动到B
    moveHanoi(n - 1, a, c, b);
    //2 将编号为n的盘子从A移动到C
    printInfo(n, a, c);
    //3 将n-1个盘子从B移动到C
    moveHanoi(n - 1, b, a, c);
}

static void printInfo(int num, String from, String to) {
    System.out.println("将编号为" + num + "号盘子, 从" + from + "移动到" + to);
}
}

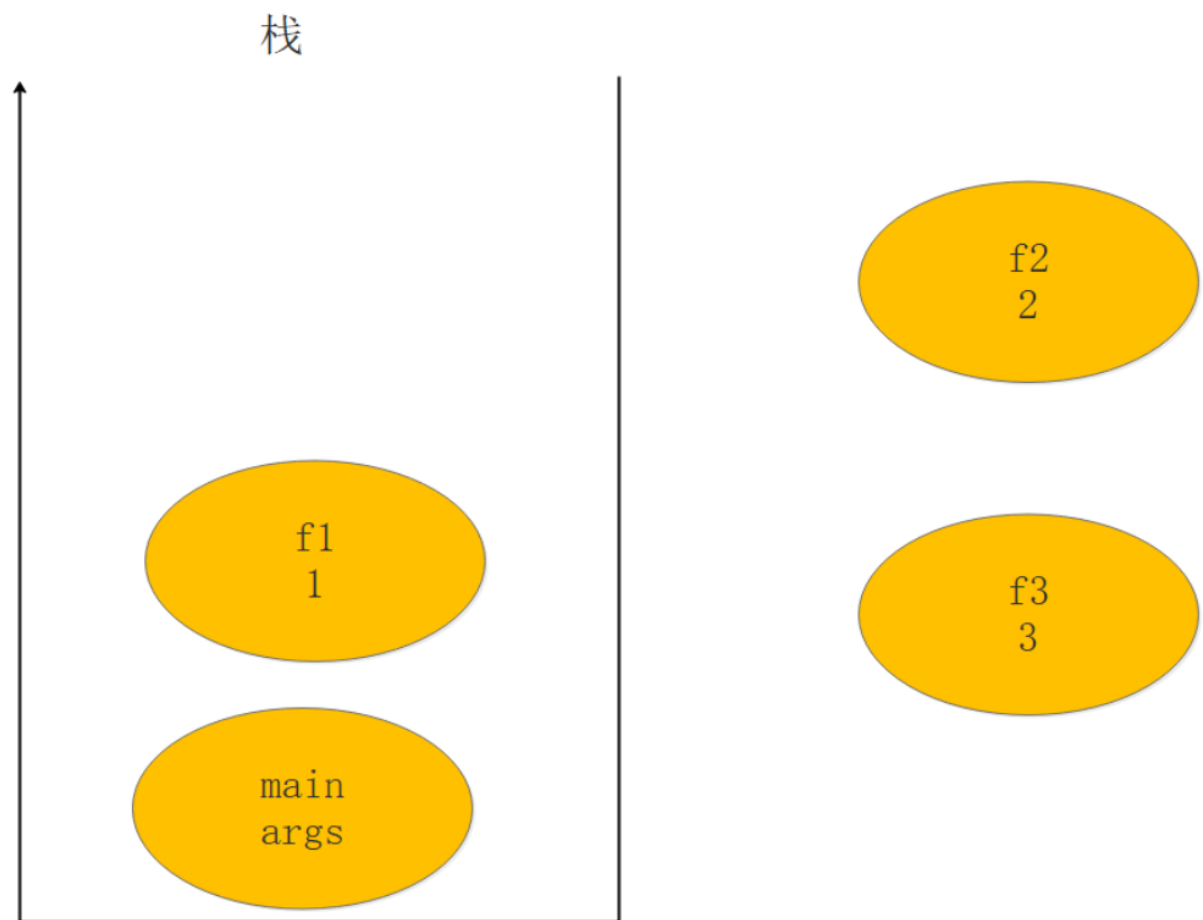
```

复杂度分析

时间复杂度: $O(2^n)$

空间复杂度: $O(n)$

第 6 节 递归总结



- 如果递归调用深度较大，调用次数过多的时候，会占用较多的空间，有可能导致栈溢出(Stack Overflow)
- 递归调用有时候存在大量重复调用，性能不好

所以我们可以考虑将递归转为非递归(迭代);

递归百分百可以转为非递归！！

什么是非递归？

```
public static void main(String[] args) {  
    sum(4);  
}  
//递归  
static int sum(int n){  
    int res=0;  
    for (int i = 1; i <= n; i++) {  
        res+=i;  
    }  
    return res;  
}  
//非递归, 迭代  
static int sum1(int n){  
    if(n <=1) return n;  
    return n+sum(n-1);  
}
```

```
}
```

递归转为非递归的另一种方法：

在某些时候可以考虑重复使用一组相同的变量来保存每个栈帧的内容

```
static void log(int n) {  
    for (int i = 1; i <= n; i++) {  
        System.out.println(i + 10);  
    }  
}
```

如果之前的是递归调用，那现在就变成了非递归，使用一次循环即搞定了。

空间复杂度：由 $O(n)$ 变为了 $O(1)$

递归转非递归两种方式：

- 自己使用栈
- 考虑重复利用一组相同的变量

2、贪心

2.1 概述

贪心策略，也被称为贪婪策略。

什么贪心策略？

每一步都采取当前状态下最优的选择（局部最优解），从而希望推导出全局最优解

2.2 案例一

最优装载问题-海盗运货

海盗们截获了一艘装满各种各样古董的货船，每一件古董都价值连城，一旦打碎就失去了珍宝的价值 海盗船的载重量为 W ，每件古董的重量为 w_i ，海盗们怎么做才能把尽可能多数量的古董装上海盗船？比如 W 为 30， w_i 分别为 3、5、4、10、7、14、2、11

使用贪心策略解决

选择货物标准：每一次都优先选择重量最小的古董！！

1、选择重量为 2 的古董，剩重量 28 2、选择重量为 3 的古董，剩重量 25 3、选择重量为 4 的古董，剩重量 21 4、选择重量为 5 的古董，剩重量 16 5、选择重量为 7 的古董，剩重量 9

结论：按照以上顺序装载货物，并最多装载5件古董！！

代码实现：

```
package com.lg.pirate;
```

```

import java.util.Arrays;

/**
 * 海盗们截获了一艘装满各种各样古董的货船，每一件古董都价值连城，一旦打碎就失去了珍宝的价值
 * 海盗船的载重量为 w，每件古董的重量为 wi，海盗们怎么做才能把尽可能多数量的古董装上海盗船？
 * 比如 w 为 30，wi 分别为 3、5、4、10、7、14、2、11
 */
public class PirateDemo {

    public static void main(String[] args) {
        piratePackage();
    }

    static void piratePackage() {
        //定义载重容量
        int capacity = 30;
        //货物
        int[] goods = {3, 5, 4, 10, 7, 14, 2, 11}; //无序，先进行排序
        Arrays.sort(goods); //升序，由小到大
        //每一次选择当前重量最小的货物装运
        // for (int i = 0; i < goods.length; i++) {
        //     System.out.println(goods[i]);
        // }
        // 2,3,4,5,7,10,11,14
        //定义已装载的重量
        int weight = 0;
        int numbs = 0;
        for (int i = 0; i < goods.length && weight < capacity; i++) {
            int newweight = weight + goods[i];
            if (newweight <= 30) {
                System.out.println(goods[i]); //打印下当前装载的是哪个货物
                weight = newweight;
                numbs++; //累计装载的货物数量
            }
        }

        System.out.println("当前装载了" + numbs + "件古董!!!");
    }
}

```

2.3 案例二

零钱兑换

假设有 25 分、5 分、1 分的硬币，现要找给客户 41 分的零钱，如何办到硬币个数最少？

贪心策略：

每一步都优先选择面值最大的硬币

具体步骤

- 选择 25 分的硬币，剩 16 分
- 选择 5 分的硬币，剩 11 分
- 选择 5 分的硬币，剩 6 分
- 选择 5 分的硬币，剩 1 分
- 选择 1 分的硬币

最终的解是 1 枚 25 分、3 枚 5 分、1 枚 1 分的硬币，共 5 枚硬币

代码实现

```
package com.lg.coins;

import java.util.Arrays;
import java.util.Comparator;

/**
 * 假设有 25 分、5 分、1 分的硬币，现要找给客户 41 分的零钱，如何办到硬币个数最少？
 * - 选择 25 分的硬币，剩 16 分
 * - 选择 5 分的硬币，剩 11 分
 * - 选择 5 分的硬币，剩 6 分
 * - 选择 5 分的硬币，剩 1 分
 * - 选择 1 分的硬币
 */
public class CoinsDemo {

    // 定义一个找零钱的方法
    // static void changeCoins(){
    //     //准备硬币的面值
    //     int[] faces={25,1,5};
    //     //总金额
    //     int money=41;
    //     //记录硬币的个数
    //     int coinsNum=0;
    //     //每次选择硬币面值最大的
    //     Arrays.sort(faces);//默认是升序
    //     for (int i = faces.length-1; i >=0; i--) { //从最高面值循环
    //         //判断当前金额与当前面值的关系
    //         if(money<faces[i]){
    //             continue;
    //         }
    //         money-=faces[i]; //同一个面值可以重复使用
    //         coinsNum++;
    //         i=faces.length-1;
    //     }
    // }

    //定义一个找零钱的方法
    static void changeCoins() {
        //准备硬币的面值
```

```

Integer[] faces = {25, 1, 5};
//总金额
int money = 41;
//记录硬币的个数
int coinsNum = 0;
//每次选择硬币面值最大的
//      Arrays.sort(faces, new Comparator<Integer>() {
//          public int compare(Integer o1, Integer o2) {
//              return o2-o1;
//          }
//      });//默认是升序,排序使用降序
Arrays.sort(faces, (Integer c1, Integer c2) -> {
    return c2 - c1;
});//降序,

//指针
int i = 0;
while (i < faces.length) {
    //判断当前金额与当前面值的关系
    if (money < faces[i]) {
        i++;
        continue;
    }

    money -= faces[i]; //同一个面值可以重复使用
    System.out.println(faces[i]);
    coinsNum++;
}

System.out.println("一共选择了" + coinsNum + "个硬币!!");
}

public static void main(String[] args) {
    changeCoins();
}
}

```

修改题目

假设有 25 分、20分， 5 分、1 分的硬币，现要找给客户 41 分的零钱，如何办到硬币个数最少？

贪心策略：

每一步都优先选择面值最大的硬币

具体步骤

- 选择 25 分的硬币，剩 16 分
- 选择 5 分的硬币，剩 11 分
- 选择 5 分的硬币，剩 6 分
- 选择 5 分的硬币，剩 1 分

- 选择 1 分的硬币

最终的解是 1 枚 25 分、3 枚 5 分、1 枚 1 分的硬币，共 5 枚硬币

但是本题的最优解是：2 枚 20 分、1 枚 1 分的硬币，共 3 枚硬币！！

贪心策略的优缺点：

- 优点
简单、高效、不需要穷举所有可能，通常作为其他算法的辅助算法来使用
- 缺点
目光短浅，不从整体上考虑其他可能，每一步只采取局部最优解，不会对比其他可能性，因此贪心很少情况能获得最优解。

使用动态规划算法来解决！！

2.4 案例三

0-1背包问题

有 n 件物品和一个最大承重为 W 的背包，每件物品的重量是 w_i 、价值是 v_i

- 在保证总重量不超过 W 的前提下，将哪几件物品装入背包，可以使得背包的总价值最大？
- 注意：每个物品只有 1 件，也就是每个物品只能选择 0 件或者 1 件，因此这类问题也被称为 0-1 背包问题

1、价值主导：优先选择价值最高的物品放进背包 2、重量主导：优先选择重量最轻的物品放进背包 3、价值密度主导：优先选择价值密度最高的物品放进背包（价值密度 = 价值 ÷ 重量）

以下是物品列表：

物品序号	1	2	3	4	5	6	7
重量	35	30	60	50	40	10	25
价值	10	40	30	50	35	40	30
价值密度	0.29	1.33	0.5	1.0	0.88	4.0	1.2

假设背包总载重量是 150，将哪几件物品装入背包，可以使得背包的总价值最大？

代码实现

```
package com.lg.packsack;

import java.util.Arrays;
import java.util.Comparator;
```

```

/**
 * 假设背包总载重量是150，将哪几件物品装入背包，可以使得背包的总价值最大？
 */
public class PackSack {

    public static void main(String[] args) {
        //准备货物
        Item[] items = {
            new Item(35, 10),
            new Item(30, 40),
            new Item(60, 30),
            new Item(50, 50),
            new Item(40, 35),
            new Item(10, 40),
            new Item(25, 30),
        };

        /**
         *
         *
         * 1、价值主导：优先选择价值最高的物品放进背包
         * 2、重量主导：优先选择重量最轻的物品放进背包
         * 3、价值密度主导：优先选择价值密度最高的物品放进背包（价值密度 = 价值 ÷ 重量）
         */

        //价值主导
        System.out.println("价值主导的选择结果：");
        packSackMethod(items, new Comparator<Item>() {
            @Override
            public int compare(Item o1, Item o2) {
                return o2.getValue() - o1.getValue();
            }
        });

        //重量主导
        System.out.println("重量主导的选择结果：");
        packSackMethod(items, new Comparator<Item>() {
            @Override
            public int compare(Item o1, Item o2) {
                return o1.getWeight() - o2.getWeight();
            }
        });

        //价值密度主导
        System.out.println("价值密度主导的选择结果：");
        packSackMethod(items, new Comparator<Item>() {
            @Override
            public int compare(Item o1, Item o2) {
                return Double.compare(o2.getValueDen(), o1.getValueDen());
            }
        });
    }
}

```

//贪心策略0-1背包 价值最大化

```
static void packSackMethod(Item[] items, Comparator<Item> comp) {  
    //按照要求对货物进行排序  
    Arrays.sort(items, comp);  
    //遍历数组，取出策略要求价值最大的物品  
    int capacity = 150;  
    int itemNum = 0;  
    //已装载的重量  
    int weight = 0;  
    //初始总价值  
    int value = 0;  
    for (int i = 0; i < items.length; i++) {  
        int newweight = weight + items[i].getweight();  
        if (newweight <= capacity) {  
            itemNum++;  
            value += items[i].getvalue();  
            weight = newweight;  
            //打印物品  
            System.out.println(items[i]);  
        }  
    }  
  
    System.out.println("共选择了" + itemNum + "件物品!!");  
    System.out.println("总价值" + value);  
    System.out.println("总重量是"+weight);  
  
    System.out.println("-----");  
}
```

3、分治

3.1 概述

分治，就是分而治之。

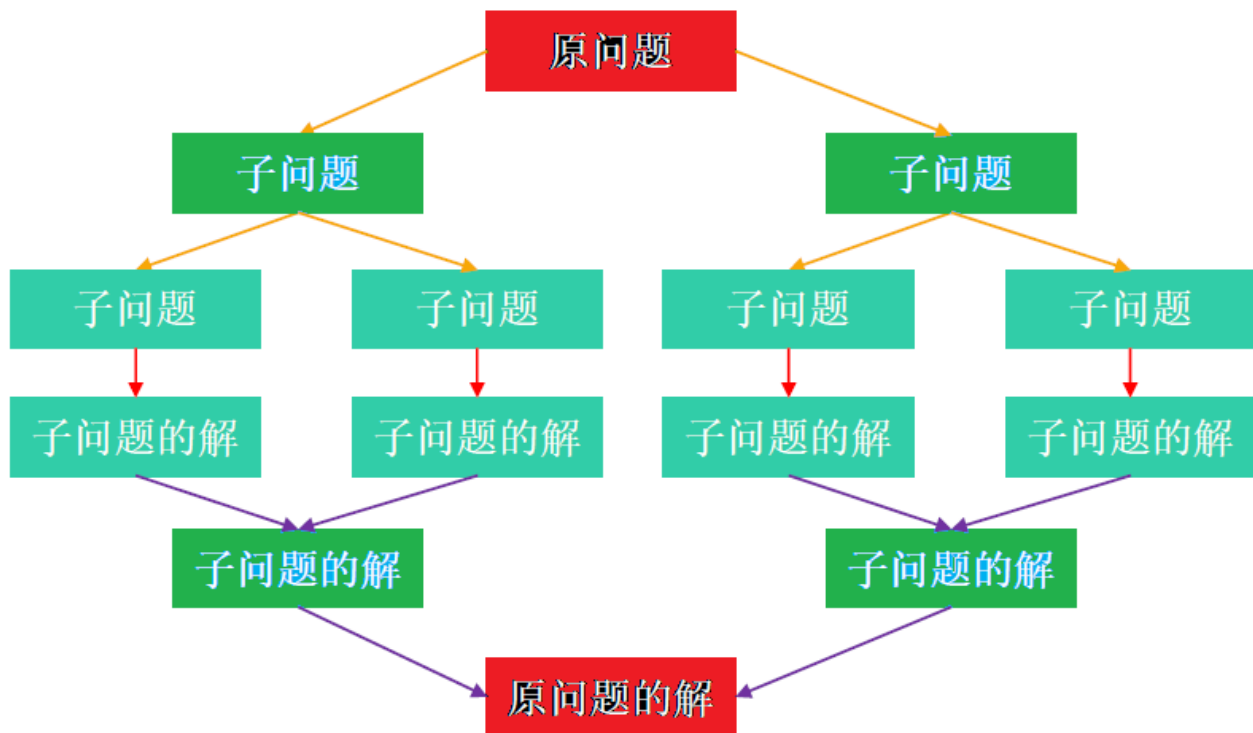
分治的一般步骤：

- 1、将原问题分解成若干个规模较小的子问题（子问题和原问题的结构一样，只是规模不一样）
- 2、子问题又不断分解成规模更小的子问题，直到不能再分解（直到可以轻易计算出子问题的解）
- 3、利用子问题的解推导出原问题的解

所以：分治思想非常适合使用递归实现！！

注意：分治的适用场景中必须要求子问题之间是相互独立的，如果子问题之间不独立则需要使用动态规划实现！！

分治思想图示



3.2 案例一

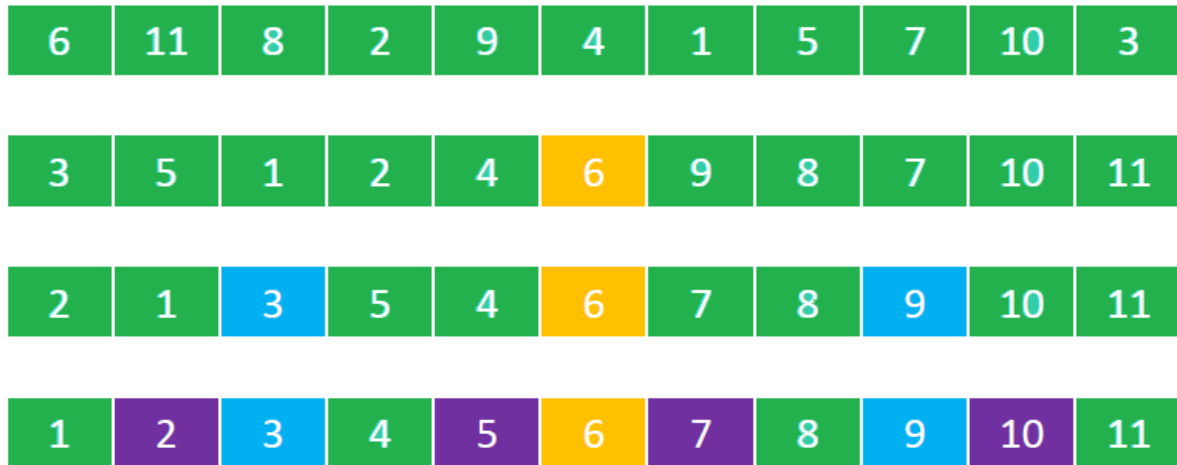
快速排序(Quick Sort)

- 什么叫排序?
 - 排序前: 3,1,6,9,2,5,8,4,7
 - 排序后: 1,2,3,4,5,6,7,8,9 (升序) 或者 9,8,7,6,5,4,3,2,1 (降序)

1、概述

快速排序是1960年由查尔斯·安东尼·理查德·霍尔 (Charles Antony Richard Hoare, 缩写为C. A. R. Hoare) 提出, 简称为东尼·霍尔 (Tony Hoare)

2、Quick Sort 分析

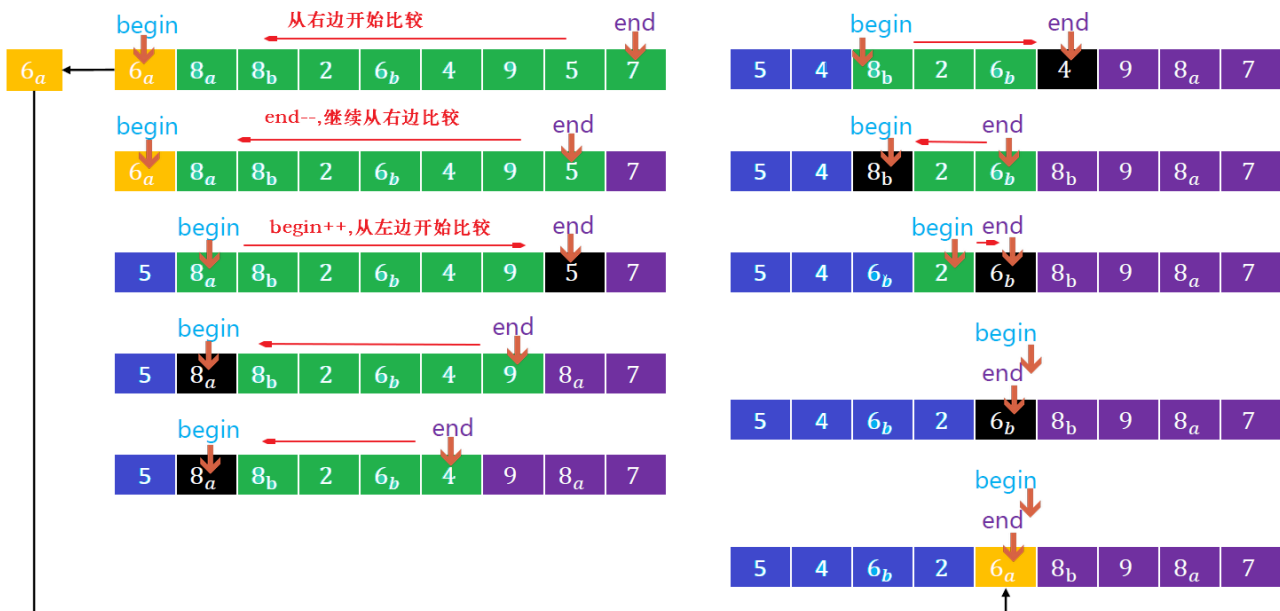


- 第一步
- 从数组中选择一个轴点元素（Pivot element），一般选择0位置元素为轴点元素
- 第二步
 - 利用Pivot将数组分割成2个子序列
 - 将小于 Pivot的元素放在Pivot前面（左侧） 将大于 Pivot的元素放在Pivot后面（右侧） 等于Pivot的元素放哪边都可以(暂定放在左边)
- 第三步
 - 对子数组进行第一步，第二步操作，直到不能再分割(子数组中只有一个元素)

Quick Sort的本质：

不断地将每一个元素都转换成轴点元素！！

3、Quick Sort实现思路



begin指针会不会大于end?

begin最终会等于end!!

只要begin==end说明这次的轴点元素已经选择成功!!

4、代码实现

```
package com.lg.sort;

import com.lg.utils.NumberTools;

import java.util.Arrays;

public class QuickSort {

    public Integer[] arr;

    public void sort(Integer[] array) {
        if (array == null || array.length < 2) return;
        this.arr = array;
        for (int i = 0; i < 20; i++) {
            System.out.print(arr[i]+"_");
        }
        long begin = System.currentTimeMillis();
        sort(0,array.length);
        long time = System.currentTimeMillis() - begin;
        System.out.println("花费时间为: "+time/1000.0+"s ");
        for (int i = 0; i < 20; i++) {
            System.out.print(arr[i]+"_");
        }
        long begin1 = System.currentTimeMillis();
        Arrays.sort(arr);
        long time1 = System.currentTimeMillis() - begin1;
        System.out.println("花费时间为: "+time1/1000.0+"s ");
    }
}
```

```

        System.out.println("-----");
        for (int i = 0; i < 20; i++) {
            System.out.print(arr[i]+"-");
        }
    }

    /**
     * 对 [begin, end) 范围的元素进行快速排序
     * @param begin
     * @param end
     */
    public void sort(int begin, int end) {
        if (end - begin < 2) return;

        // 确定轴点位置 O(n)
        int mid = pivotIndex(begin, end);
        // 对子序列进行快速排序
        sort(begin, mid);
        sort(mid + 1, end);
    }

    /**
     * 构造出 [begin, end) 范围的轴点元素
     * @return 轴点元素的最终位置
     */
    public int pivotIndex(int begin, int end) {

        // 备份begin位置的元素
        Integer pivot = arr[begin];
        // end指向最后一个元素
        end--;

        while (begin < end) {
            while (begin < end) {
                if (cmp(pivot, arr[end]) < 0) { // 右边元素 > 轴点元素
                    end--;
                } else { // 右边元素 <= 轴点元素
                    arr[begin++] = arr[end];
                    break;
                }
            }
            while (begin < end) {
                if (cmp(pivot, arr[begin]) > 0) { // 左边元素 < 轴点元素
                    begin++;
                } else { // 左边元素 >= 轴点元素
                    arr[end--] = arr[begin];
                    break;
                }
            }
        }
    }
}

```

```

        // 将轴点元素放入最终的位置
        arr[begin] = pivot;
        // 返回轴点元素的位置
        return begin;
    }

    /*
     * 返回值等于0, 代表 array[i1] == array[i2]
     * 返回值小于0, 代表 array[i1] < array[i2]
     * 返回值大于0, 代表 array[i1] > array[i2]
     */
    private int cmp(Integer i1, Integer i2) {
        return i1.compareTo(i2);
    }

    public static void main(String[] args) {
        new QuickSort().sort(NumberTools.random(10000, 1, 1000));
    }
}

```

生成数组工具类

```

package com.lg.utils;

public class NumberTools {

    public static Integer[] random(int count, int min, int max) {
        if (count <= 0 || min > max) return null;
        Integer[] array = new Integer[count];
        int delta = max - min + 1;
        for (int i = 0; i < count; i++) {
            array[i] = min + (int) (Math.random() * delta);
        }
        return array;
    }
}

```

5、Quick Sort复杂度以及稳定性分析

常见递推式与复杂度

时间递推式	复杂度
$T(n) = T(n/2) + O(1)$	$O(\log n)$
$T(n) = T(n-1) + O(1)$	$O(n)$
$T(n) = T(n/2) + O(n)$	$O(n)$
$T(n) = 2 * T(n/2) + O(1)$	$O(n)$
$T(n) = 2 * T(n/2) + O(n)$	$O(n \log n)$
$T(n) = T(n-1) + O(n)$	$O(n^2)$
$T(n) = 2 * T(n-1) + O(1)$	$O(2^n)$
$T(n) = 2 * T(n-1) + O(n)$	$O(2^n)$

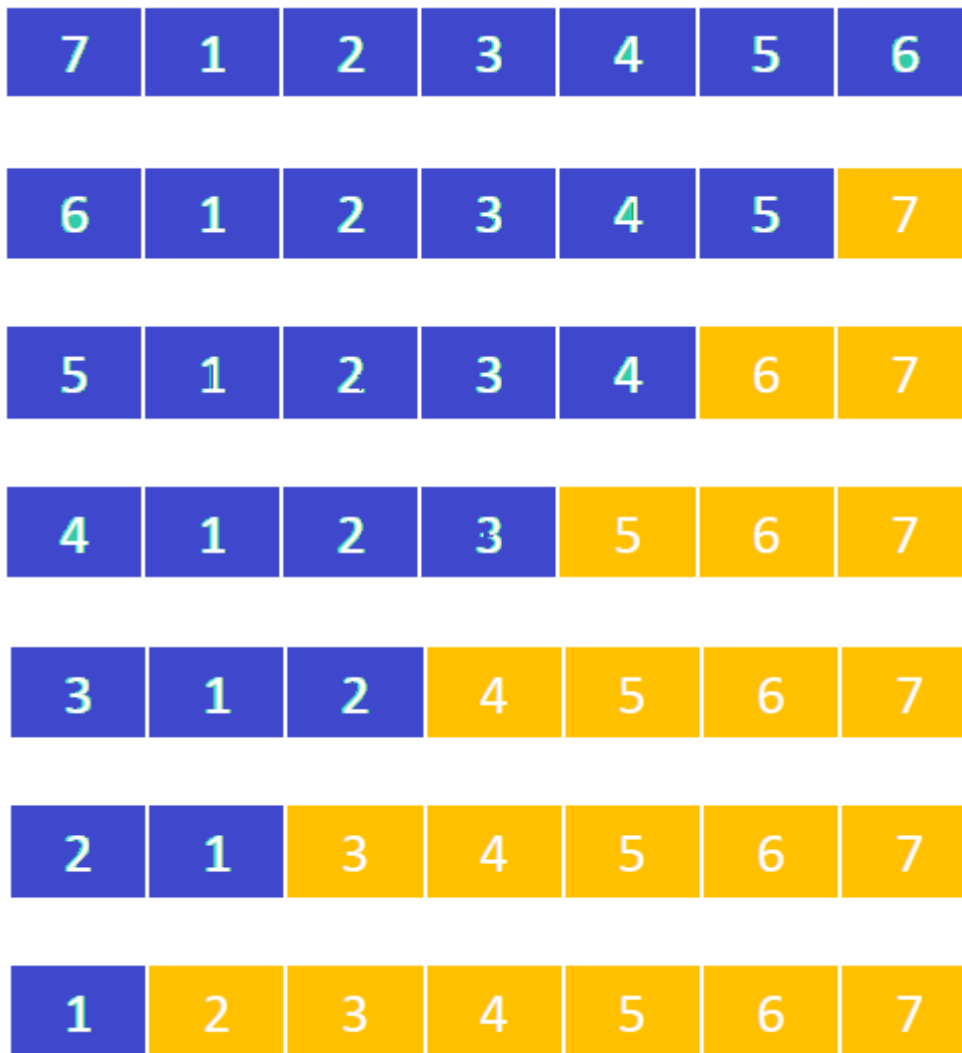
- 时间复杂度

- 最坏情况:

$$T(n) = T(n - 1) + O(n) = O(n^2)$$

- 最好情况:

$$T(n) = 2 * T(n/2) + O(n) = O(n \log n)$$



- 空间复杂度

由于递归调用，每次类似折半效果所以空间复杂度是 $O(\log n)$

排序算法的稳定性

对于排序算法还有一个评价指标就是稳定性！！

什么是排序算法的稳定性？

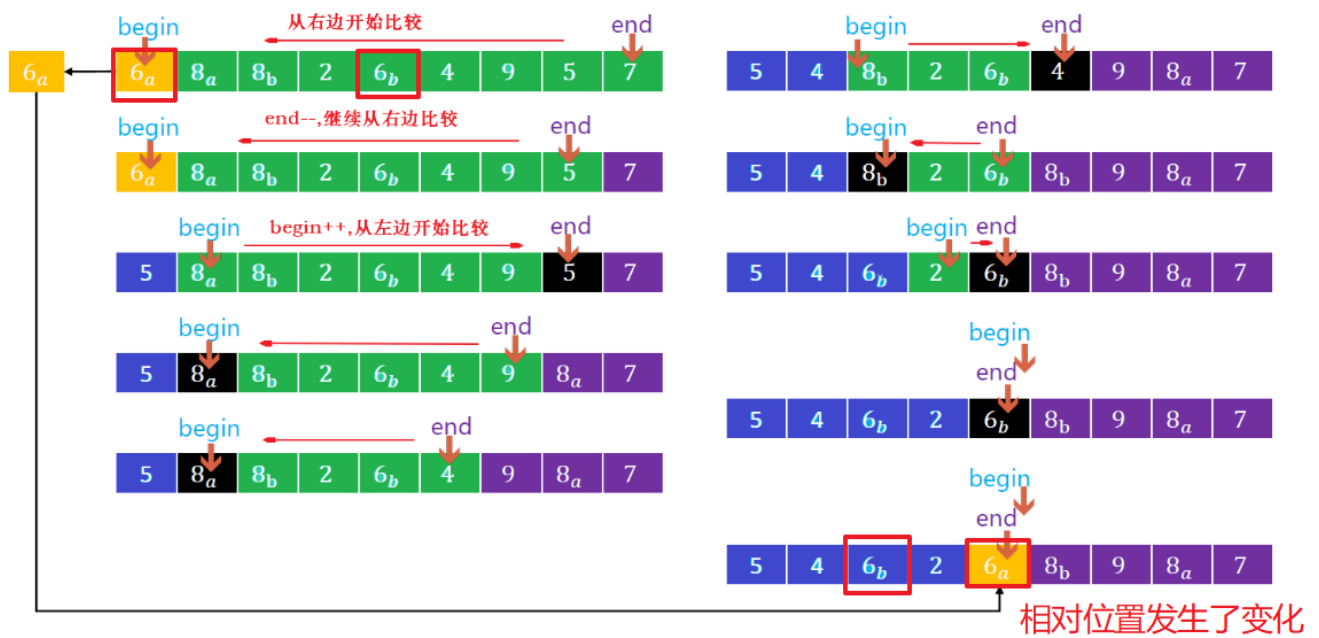
如果相等的2个元素，在排序前后的相对位置保持不变，则该算法是稳定的排序算法！！

举例：

排序前：5, 1, 2a, 4, 7, 2b

稳定的排序：1, 2a, 2b, 4, 5, 7

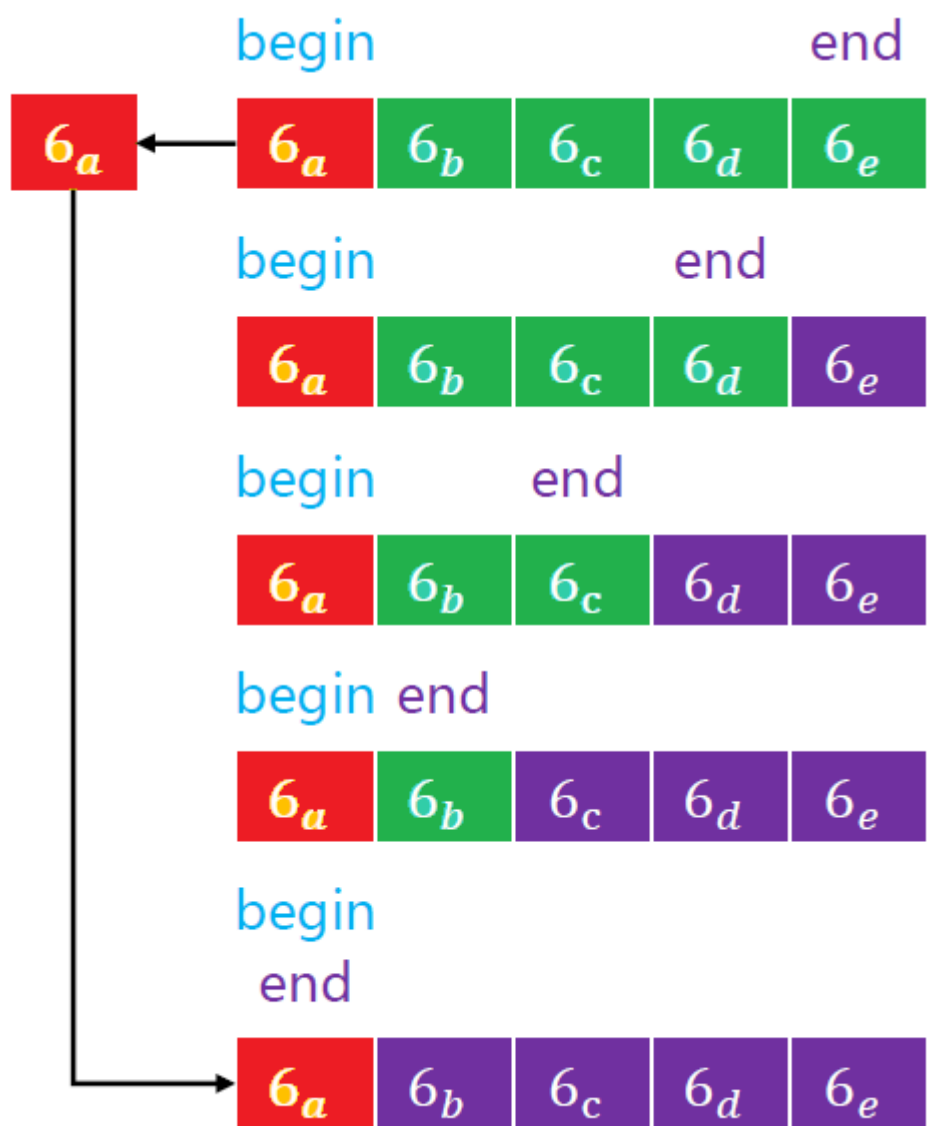
不稳定的排序：1, 2b, 2a, 4, 5, 7



对于数值类型的排序而言，算法的稳定性没有意义，但是对于自定义对象的排序，排序算法的稳定性是会影响最终结果的！！

元素与轴点元素相等

上述代码实现中，对于与轴点相等的元素选择了移动到左边而不是原地不动，如果发生元素与轴点元素相等的情况，选择原地不动会出现什么情况？



如果按照上述标准，不移动与轴点相等的元素，是有可能把数组分割为不均匀的子数组，进而导致产生最坏的时间复杂度 $O(n^2)$ 。

补充：参考代码中的实现，对于与轴点相等元素进行了移动并不是简单移动到左边而是每次移动后会调换方向，所以即使所有元素全部相等也可以利用轴点元素将序列分割成 2 个均匀的子序列。

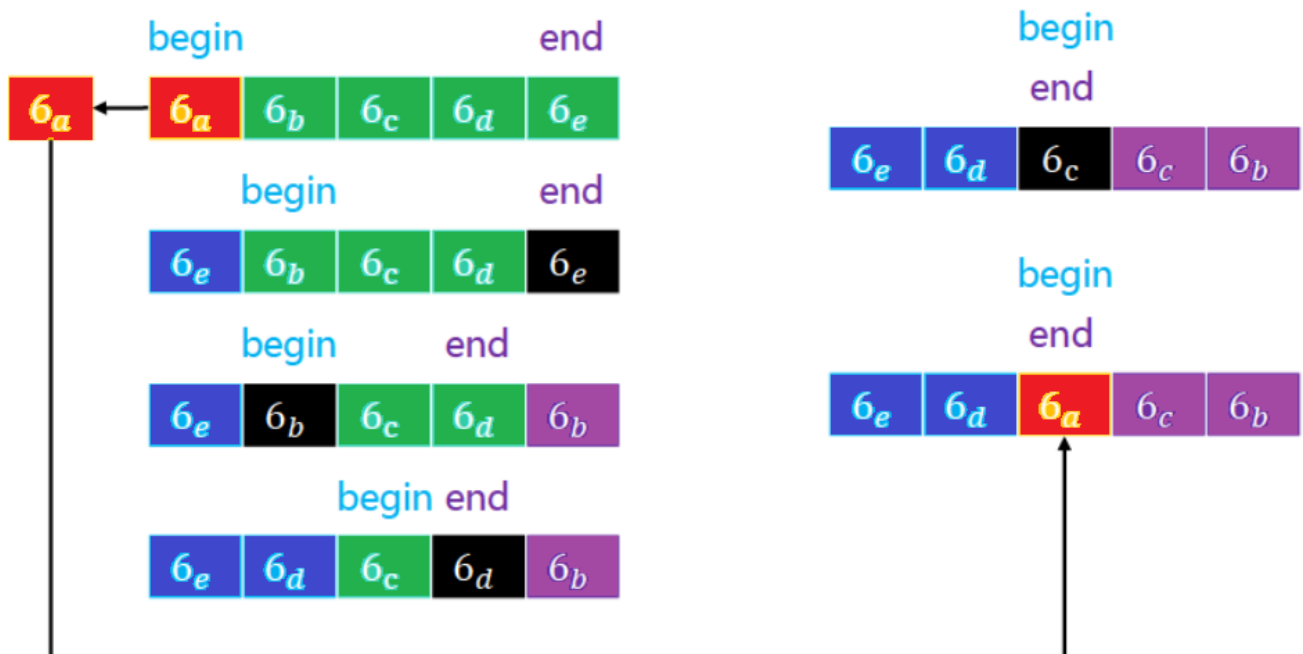
```

// end指向最后一个元素
end--;

while (begin < end) {
    while (begin < end) {
        if (cmp(pivot, arr[end]) < 0) { // 右边元素 > 轴点元素
            end--;
        } else { // 右边元素 <= 轴点元素
            arr[begin++] = arr[end];
            break;
        }
    }
    while (begin < end) {
        if (cmp(pivot, arr[begin]) > 0) { // 左边元素 < 轴点元素
            begin++;
        } else { // 左边元素 >= 轴点元素
            arr[end--] = arr[begin];
            break;
        }
    }
}
}

```

左右互换



作业

如何降低快速排序排序最坏时间复杂度产生的概率？

4、动态规划

4.1 概述

动态规划，简称为DP。

动态规划是求解最优化问题的一种常用策略。

动态规划使用步骤

- 递归（自顶向下，出现了重叠子问题）
- 记忆化（自顶向下,备忘录）
- 递推（自底向上，循环）

4.1.1重要概念

Dynamic Programming is a method for solving a complex problem by breaking it down into a collection of simpler subproblems, solving each of those subproblems just once, and storing their solutions.

--维基百科

中文翻译：

- 1、将复杂的原问题拆解成若干个简单的子问题
- 2、每个子问题仅仅解决1次，并保存它们的解
- 3、最后推导出原问题的解

动态规划策略可以解决哪些问题？

这类问题通常具有两个特点

- 最优化问题(最优子结构问题)：通过求解子问题的最优解，可以获得原问题的最优解
- 无后效性

补充：无后效性是指某阶段的状态一旦确定后，后续状态的演变不再受此前各状态及决策的影响（未来与过去无关）；在推导后面阶段的状态时，只关心前面阶段的具体状态值，不关心这个状态是怎么一步步推导出！！

无后效性

(0,0)				
		(i, j-1)		
	(i-1, j)	(i, j)		
				(4,4)

- 从起点 $(0, 0)$ 走到终点 $(4, 4)$ 一共有多少种走法？只能向右、向下走
- 假设 $dp(i, j)$ 是从 $(0, 0)$ 走到 (i, j) 的走法
 - $dp(i, 0) = dp(0, j) = 1$
 - $dp(i, j) = dp(i, j - 1) + dp(i - 1, j)$

总结

推导 $dp(i, j)$ 时只需要用到 $dp(i, j - 1)$ 、 $dp(i - 1, j)$ 的值 不需要关心 $dp(i, j - 1)$ 、 $dp(i - 1, j)$ 的值是怎么求出来的

有后效性

(0,0)				
		(i, j-1)		
	(i-1, j)	(i, j)		
				(4,4)

规则：可以向左、向右、向上、向下走，并且同一个格子不能走 2 次

有后效性：

- $dp(i, j)$ 下一步要怎么走，还要关心上一步是怎么来的
- 还需要考虑 $dp(i, j - 1)$ 、 $dp(i - 1, j)$ 是怎么来的

4.2 使用步骤

- 定义状态
状态指的是原问题，子问题的解，例如 $dp(i)$
- 设定初始状态
问题的边界，比如设置 $dp(0)$ 的值
- 确定状态转移方程
确定 $dp(i)$ 和 $dp(i - 1)$ 的关系

4.3 案例一

leetcode_322_零钱兑换: <https://leetcode-cn.com/problems/coin-change/>

假设有25分、20分、5分、1分的硬币，现要找给客户41分的零钱，如何办到硬币个数最少？

此前贪心策略求解结果是5枚硬币，并非是最优解！！

思路:

- 定义状态

dp(n):凑到 n 分需要的最少硬币个数

- 设定初始状态

dp(25)=dp(20)=dp(5)=dp(1)=1

- 确定状态转移方程

- 如果第 1 次选择了 25 分的硬币, 那么 $dp(n) = dp(n - 25) + 1$
- 如果第 1 次选择了 20 分的硬币, 那么 $dp(n) = dp(n - 20) + 1$
- 如果第 1 次选择了 5 分的硬币, 那么 $dp(n) = dp(n - 5) + 1$
- 如果第 1 次选择了 1 分的硬币, 那么 $dp(n) = dp(n - 1) + 1$
- 所以 $dp(n) = \min \{ dp(n - 25), dp(n - 20), dp(n - 5), dp(n - 1) \} + 1$

代码实现

```
package com.lg.dp;

/**
 * 使用动态规划求解最少硬币个数
 */
public class CoinsDemo {
    //1 定义一个状态, 方法, 凑够n分的最少硬币个数
    //dp(n):凑够n分的最少硬币个数
    static int coinChange(int n){
        //递归基
        if(n < 1){ return Integer.MAX_VALUE;}
        //2 确定初始状态
        if(n==1 || n==5 || n==20 || n==25){ return 1;}
        //3、确定状态的转移方程: dp(n)与dp(n-1)的关系,
        /**
         * 如果第一次选择了1分; coinChange(n)=coinChange(n-1)+1
         * 如果第一次选择了5分; coinChange(n)=coinChange(n-5)+1
         * 如果第一次选择了20分; coinChange(n)=coinChange(n-20)+1
         * 如果第一次选择了25分; coinChange(n)=coinChange(n-25)+1
         * 以上四种情况选择哪一种呢? 应该选择的是硬币个数最少的这种
         */
        final int min1 = Math.min(coinChange(n - 1), coinChange(n - 5));
        final int min2 = Math.min(coinChange(n - 20), coinChange(n - 25));
        return Math.min(min1, min2) + 1;
    }

    public static void main(String[] args) {
        System.out.println(coinChange(41));
    }
}
```

这种递归方式会存在大量重复计算, 时间复杂度是比较高的!!

优化一

使用数组保存计算过的子问题的解，避免重复计算！！

代码实现

```
package com.lg.dp;

/**
 * 使用动态规划求解最少硬币个数,使用备忘录方式,记忆化搜索
 * 大体思路
 * 把计算过的结果记录下来,每个子问题的解仅仅解决一次,使用数组保存子问题的解
 */
public class CoinsDemo2 {
    /**
     * 定义找零钱方法
     */
    static int coins(int n){
        //过滤不合理的值
        if(n <1){ return -1;}
        //定义一个数组保存子问题的解, dp(n)...dp(2),dp(1)
        int[] dp=new int[n+1]; //数组索引从0开始,只是为了与dp(n)位置对应,数组初始化值都是0
        //初始状态准备好
        dp[1]=dp[5]=dp[20]=dp[25]=1;
        return coins(n,dp); //调用coins(n,dp)
    }

    static int coins(int n, int[] dp) {
        //缺少递归基
        if(n <1){ return Integer.MAX_VALUE;}
        /**
         * 子问题的解保存在dp数组中,如果已经计算过则直接取出,否则需要计算,并最终保存到dp数组中
         */

        if(dp[n]==0){
            //说明dp(n)没有被计算,需要计算
            int min1 = Math.min(coins(n - 1,dp), coins(n - 5,dp));
            int min2 = Math.min(coins(n - 20,dp), coins(n - 25,dp));
            dp[n]= Math.min(min1, min2) + 1;
        }

        return dp[n];
    }

    public static void main(String[] args) {
        System.out.println(coins(41));
    }
}
```

优化二

使用递推方式实现,从小计算到大。

代码实现

```
package com.lg.dp;

/**
 * 使用动态规划求解最少硬币个数,使用递推方式, 由小到大
 * 大体思路
 * 从小计算到到, dp(1),dp(2)...dp(n)
 */
public class CoinsDemo3 {
    public static void main(String[] args) {
        System.out.println(coins(0));
    }
    /**
     递推方式实现找零钱-动态规划版本
     */

    static int coins(int n) { //对应dp(n)

        //过滤掉不合理的值
        if (n < 1) {
            //说明钱数不合理
            return -1;
        }
        //使用一个数组盛放子问题的解
        int[] dp = new int[n + 1];
        //从小问题计算直到计算出大问题的解
        for (int i = 1; i <= n; i++) {
            //
            int min = Integer.MAX_VALUE;
            //
            min = Math.min(min, dp[i - 1]);
            int min = dp[i - 1];
            if (i >= 5) min = Math.min(min, dp[i - 5]);
            if (i >= 20) min = Math.min(min, dp[i - 20]);
            if (i >= 25) min = Math.min(min, dp[i - 25]);
            dp[i] = min + 1;
        }
        return dp[n];
    }
}
```

打印硬币的面值



代码实现

```
package com.lg.dp;

/**
 * 使用动态规划求解最少硬币个数,使用递推方式, 由小到大
 * 大体思路
 * 从小计算到到, dp(1),dp(2)...dp(n)
 */
public class CoinsDemo3 {
    public static void main(String[] args) {
        System.out.println(coins(6));
    }
    /**
     递推方式实现找零钱-动态规划版本
     */

    static int coins(int n) { //对应dp(n)

        //过滤掉不合理的值
        if (n < 1) {
            //说明钱数不合理
            return -1;
        }
        //使用一个数组盛放子问题的解
        int[] dp = new int[n + 1];
        //准备一个数组记录下选择了哪些面值
        int[] selected = new int[dp.length];
        //从小问题计算直到计算出大问题的解
        for (int i = 1; i <= n; i++) {
            //
            int min = Integer.MAX_VALUE;
            //
            min = Math.min(min, dp[i - 1]);
            int min = dp[i - 1]; //当前i分选择硬币的最后一枚面值大小
            selected[i] = 1;
            if (i >= 5 && dp[i - 5] < min) {
                min = dp[i - 5];
                selected[i] = 5;
            }
            if (i >= 20 && dp[i - 20] < min) {
                min = dp[i - 20];
                selected[i] = 20;
            }
            if (i >= 25 && dp[i - 25] < min) {
                min = dp[i - 25];
                selected[i] = 25;
            }
            dp[i] = min + 1;
        }

        printSelected(selected, n);
        return dp[n];
    }
}
```



```

//打印selected数组中的面值信息
static void printSelected(int[] selected, int n) {
    while (n > 0) {
//        selected[n] //第一枚硬币，下一枚硬币的索引是当前索引减去取出的面值
        System.out.print(selected[n] + "-");
        n -= selected[n];
    }
    System.out.println();
}
}

```

通用方案

代码实现

```

static int coinChange(int[] coins, int amount) {
    //过滤掉不合理值
    if (amount < 1 || coins == null || coins.length == 0) return 0;
    //使用递推方式
    int[] dp = new int[amount + 1];
    //遍历钱
    for (int i = 1; i <= amount; i++) {
        int min = Integer.MAX_VALUE;
        //遍历面值
        for (int face : coins) {
            //需要比较i与面值的大小关系
            if (i < face || dp[i-face] < 0) {
                continue;
            }
            min = Math.min(dp[i - face], min);
        }
        if (min == Integer.MAX_VALUE) {
            //面值不符
            dp[i] = -1;
        } else {
            dp[i] = min + 1;
        }
    }
    return dp[amount];
}
}

```

4.4 案例二

最大的连续子序列和

题目：

给定一个长度为 n 的整数序列，求它的最大连续子序列和

-2、1、-3、4、-1、2、1、-5、4 的最大连续子序列和是 $4 + (-1) + 2 + 1 = 6$

- 定义状态

$dp(i)$ 是以 $nums[i]$ 结尾的最大连续子序列和 ($nums$ 是整个序列)

- 初始状态

$dp(0)=nums[0]$

- 状态转移方程

如果以 $nums[0]$ -2 结尾，则最大连续子序列是 -2，所以 $dp(0) = -2$

如果以 $nums[1]$ 1 结尾，则最大连续子序列是 1，所以 $dp(1) = 1$

如果以 $nums[2]$ -3 结尾，则最大连续子序列是 1、-3，所以 $dp(2) = dp(1) + (-3) = -2$

如果以 $nums[3]$ 4 结尾，则最大连续子序列是 4，所以 $dp(3) = 4$

如果以 $nums[4]$ -1 结尾，则最大连续子序列是 4、-1，所以 $dp(4) = dp(3) + (-1) = 3$

如果以 $nums[5]$ 2 结尾，则最大连续子序列是 4、-1、2，所以 $dp(5) = dp(4) + 2 = 5$

如果以 $nums[6]$ 1 结尾，则最大连续子序列是 4、-1、2、1，所以 $dp(6) = dp(5) + 1 = 6$

如果以 $nums[7]$ -5 结尾，则最大连续子序列是 4、-1、2、1、-5，所以 $dp(7) = dp(6) + (-5) = 1$

如果以 $nums[8]$ 4 结尾，则最大连续子序列是 4、-1、2、1、-5、4，所以 $dp(8) = dp(7) + 4 = 5$

- 如果 $dp(i-1) \leq 0$ ，那么 $dp(i) = nums[i]$

- 如果 $dp(i-1) > 0$ ，那么 $dp(i) = dp(i-1) + nums[i]$

最终解：

最大连续子序列和是所有 $dp(i)$ 中的最大值 $\max \{ dp(i) \}$ ， $i \in [0, nums.length)$

代码

```
package com.lg.maxsubarray;

public class MaxSubArray {
    public static void main(String[] args) {
        int[] a= new int[]{-1,2};
        System.out.println(maxsubArray(a));
    }

    static int maxsubArray(int[] nums){
        if(nums==null || nums.length==0){
            return 0;
        }
        int[] dp=new int[nums.length];
        dp[0]=nums[0];

        int max=dp[0];
        //开始遍历循环
        for(int i =1;i<nums.length;i++){
            if(dp[i-1] <=0){
```

```

        dp[i]=nums[i];
    }else{
        dp[i]=dp[i-1]+nums[i];
    }
    max=Math.max(max, dp[i]);
}
return max;
}
}

```

优化

使用变量替代数组

代码实现

```

package com.lg.dp;

public class MaxSubArray2 {
    public static void main(String[] args) {
        int[] arr = new int[]{1, -2, 3, 4};
        System.out.println(getMaxSub(arr));
    }

    //定义状态: dp(i):dp(i) 是以 nums[i] 结尾的最大连续子序列和 (nums是整个序列)
    //使用递推方式
    static int getMaxSub(int[] nums) {

        //过滤不合理值
        if (nums == null || nums.length == 0) {
            return 0;
        }
        //准备一个变量
        int dp =nums[0] ;
        //遍历nums,找到每一个最大的连续子序列和
        int max = dp;
        for (int i = 1; i < nums.length; i++) {
            if(dp >0){
                //下一个连续子序列和要加上这个遏制
                dp=dp+nums[i];
            }else{
                dp=nums[i];
            }
            max = Math.max(dp, max);
        }

        return max;
    }
}

```

4.5 案例三

最长公共子序列

最长公共子序列 (Longest Common Subsequence, LCS)

leetcode_1143_最长公共子序列: <https://leetcode-cn.com/problems/longest-common-subsequence/>

计算两个序列的最长公共子序列长度

[1, 3, 5, 9, 10] 和 [1, 4, 9, 10] 的最长公共子序列是 [1, 9, 10], 长度为 3

1、思路分析

假设 2 个序列分别是 nums1、nums2

- $i \in [0, \text{nums1.length}]$
- $j \in [0, \text{nums2.length}]$

1.1 定义状态方程

假设 $dp(i, j)$ 是【nums1 前 i 个元素】与【nums2 前 j 个元素】的最长公共子序列长度

1.2 定义初始值

$dp(i, 0)$ 、 $dp(0, j)$ 初始值均为 0

1.3 定义状态转移方程

- 假设 $\text{nums1}[i - 1] = \text{nums2}[j - 1]$, 那么 $dp(i, j) = dp(i - 1, j - 1) + 1$
- 假设 $\text{nums1}[i - 1] \neq \text{nums2}[j - 1]$, 那么 $dp(i, j) = \max \{ dp(i - 1, j), dp(i, j - 1) \}$

2、第一版

递归实现

```
package com.lg.dp;

/*
计算最长公共子序列
*/
public class LCS {
    public static void main(String[] args) {
        int[] arr1={1,4,5,9,10};
        int[] arr2={1,4,9,10};
        System.out.println( getLCS(arr1,arr2 ));
    }
    //定义一个方法接收两个数组, 返回两个数组的最长公共子序列
    static int getLCS(int[] arr1, int[] arr2) {
        //过滤掉不合理的值
        if (arr1 == null || arr2 == null || arr1.length == 0 || arr2.length == 0) {
            return 0;
        }
    }
}
```

```

        return lcs(arr1, arr1.length, arr2, arr2.length);
    }

    //定义状态方程
    static int lcs(int[] arr1, int i, int[] arr2, int j) {
        //定义初始值
        if (i == 0 || j == 0) {
            return 0;
        }
        //如果arr1[i-1]=nums2[j-1]
        if (arr1[i - 1] == arr2[j - 1]) {
            return lcs(arr1, i - 1, arr2, j - 1) + 1;
        } else {
            return Math.max(lcs(arr1, i - 1, arr2, j), lcs(arr1, i, arr2, j - 1));
        }
    }
}

```

空间复杂度: $O(k)$, $k = \min\{n, m\}$, n 、 m 是 2 个序列的长度 时间复杂度: $O(2^n)$, 当 $n = m$ 时

3、第二版

使用递推实现

代码实现

```

package com.lg.dp;

/*
计算最长公共子序列
*/
public class LCS2 {
    public static void main(String[] args) {
        int[] arr1 = {1, 3, 5, 9, 10};
        int[] arr2 = {1, 4, 9, 10};
        System.out.println(getLCS(arr1, arr2));
    }

    //定义一个方法接收两个数组, 返回两个数组的最长公共子序列
    static int getLCS(int[] arr1, int[] arr2) {
        //过滤掉不合理的值
        if (arr1 == null || arr2 == null || arr1.length == 0 || arr2.length == 0) {
            return 0;
        }
        //准备二维一个数组记录下子问题的解, 使用递推方式自下而上计算
        int[][] dp = new int[arr1.length + 1][arr2.length + 1];
        for (int i = 1; i <= arr1.length; i++) {
            for (int j = 1; j <= arr2.length; j++) {
                //判断是否相等
                if (arr1[i - 1] == arr2[j - 1]) {
                    dp[i][j] = dp[i - 1][j - 1] + 1;
                } else {

```

```

        dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
    }
}
return dp[arr1.length][arr2.length];
}
}

```

空间复杂度: $O(n * m)$ 时间复杂度: $O(n * m)$

非递归实现分析

		A B C B D A B							
i \ j		0	1	2	3	4	5	6	7
0		0	0	0	0	0	0	0	0
B	1	0	0	1	1	1	1	1	1
D	2	0	0	1	1	1	2	2	2
C	3	0	0	1	2	2	2	2	2
A	4	0	1	1	2	2	2	3	3
B	5	0	1	2	2	3	3	3	4
A	6	0	1	2	2	3	3	4	4

(备注: 图片来自网络)

4、第三版

使用一维数组优化

代码实现

```

package com.lg.dp;

/**
 * 计算最长公共子序列
 */
public class LCS3 {
    public static void main(String[] args) {
        int[] arr1 = {1, 4, 5, 9, 10};
        int[] arr2 = {1, 4, 9, 10};
        System.out.println(getLCS(arr1, arr2));
    }
}

```

```

//定义一个方法接收两个数组，返回两个数组的最长公共子序列
static int getLCS(int[] arr1, int[] arr2) {
    //过滤掉不合理的值
    if (arr1 == null || arr2 == null || arr1.length == 0 || arr2.length == 0) {
        return 0;
    }
    //准备二维一个数组记录下子问题的解，使用递推方式自下而上计算
    int[] dp = new int[arr2.length + 1];
    for (int i = 1; i <= arr1.length; i++) {
        int tmp=0;
        for (int j = 1; j <= arr2.length; j++) {
            //判断是否相等
            int leftTop=tmp;
            tmp=dp[j];
            if (arr1[i - 1] == arr2[j - 1]) {
                dp[j] = leftTop + 1;
            } else {
                dp[j] = Math.max(dp[j], dp[j - 1]);
            }
        }
    }
    return dp[arr2.length];
}
}

```

空间复杂度优化到O(n)级别！！

5、案例四

0-1背包问题

有 n 件物品和一个最大承重为 W 的背包，每件物品的重量是 w_i 、价值是 v_i

- 在保证总重量不超过 W 的前提下，将哪几件物品装入背包，可以使得背包的总价值最大？
- 注意：每个物品只有 1 件，也就是每个物品只能选择 0 件或者 1 件，因此这类问题也被称为 0-1背包问题

动态规划步骤

设定 values 是价值数组，weights 是重量数组

```

int[] values= {6,3,5,4,6};
int[] weights={2,2,6,5,4};
int capacity=10;

```

- 定义状态方程

$dp(i, j)$ 是最大承重为 j 、有前 i 件物品可选 时的最大总价值， $i \in [0, n]$, $j \in [0, W]$

- 初始状态

$dp(i, 0)$ 、 $dp(0, j)$ 初始值均为 0

- 状态转移方程

$dp(i, j) = dp(i-1, j-1)$

如果只剩最后一件物品时，有两种情况

- 不选择该物品： $dp(i, j) = dp(i-1, j)$
- 选择该物品： $dp(i, j) = values[i] + dp(i-1, j - weights[i])$

$dp(i, j)$ 返回的是最大总价值

$\max(dp(i-1, j), values[i] + dp(i-1, j - weights[i]))$

- 如果 $j < weights[i - 1]$ ，那么 $dp(i, j) = dp(i - 1, j)$
- 如果 $j \geq weights[i - 1]$ ，那么 $dp(i, j) = \max \{ dp(i - 1, j), dp(i - 1, j - weights[i - 1]) + values[i - 1] \}$

代码实现

```
package com.lg.dp;

/**
 * 使用动态规划解决01背包问题
 */
public class PackSack {

    public static void main(String[] args) {
        int[] values = {6, 3, 5, 4, 6};
        int[] weights = {2, 2, 6, 5, 4};
        int capacity = 10;
        System.out.println(getMaxVal(values, weights, capacity));
    }

    static int getMaxVal(int[] values, int[] weights, int capacity) {
        //过滤掉不合理的值
        if (values == null || values.length == 0) {
            return 0;
        }
        if (weights == null || weights.length == 0) {
            return 0;
        }
        if (capacity < 0) {
            return 0;
        }
        //使用递推方式:dp(i,j):最大承重为j,有前i件物品可选时的最大总价值
        int[][] dp = new int[values.length + 1][capacity + 1]; //数组初始化默认值就是0
        //遍历
        for (int i = 1; i <= values.length; i++) {
            for (int j = 1; j <= capacity; j++) {
                //翻译状态转移方程
                if (j < weights[i - 1]) {
                    dp[i][j] = dp[i - 1][j];
                } else {

```



```

        dp[i][j] = Math.max(dp[i - 1][j], dp[i - 1][j - weights[i - 1]] +
values[i - 1]);
    }
}
}
return dp[values.length][capacity];
}
}

```

递推实现思路分析

		j											
		i	0	1	2	3	4	5	6	7	8	9	10
v=6, w=2	0	0	0	0	0	0	0	0	0	0	0	0	0
	1	0	0	6	6	6	6	6	6	6	6	6	6
v=3, w=2	2	0	0	6	6	9	9	9	9	9	9	9	9
v=5, w=6	3	0	0	6	6	9	9	9	9	11	11	14	14
v=4, w=5	4	0	0	6	6	9	9	9	10	11	13	14	14
v=6, w=4	5	0	0	6	6	9	9	12	12	15	15	15	15

第二版

使用一维数组进行优化。

```

package com.lg.dp;

import java.rmi.ConnectIOException;

/**
 * 使用动态规划解决01背包问题,使用一维数组优化
 */
public class PackSack1 {

    public static void main(String[] args) {
        int[] values = {6, 3, 5, 4, 6};
        int[] weights = {2, 2, 6, 5, 4};
        int capacity = 10;
        System.out.println(getMaxVal(values, weights, capacity));
    }

    static int getMaxVal(int[] values, int[] weights, int capacity) {
        //过滤掉不合理的值
        if (values == null || values.length == 0) {
            return 0;
        }
        if (weights == null || weights.length == 0) {
            return 0;
        }
        if (capacity < 0) {

```

```

        return 0;
    }
    //使用递推方式:dp(i,j):最大承重为j,有前i件物品可选时的最大总价值
    int[] dp = new int[capacity + 1]; //数组初始化默认值就是0
    //遍历
    for (int i = 1; i <= values.length; i++) {
        //改变为从右向左执行
        for (int j = capacity; j >= weights[i-1]; j--) {
            //翻译状态转移方程
            dp[j] = Math.max(dp[j], dp[j - weights[i - 1]] + values[i - 1]);
        }
    }
    return dp[capacity];
}
}

```