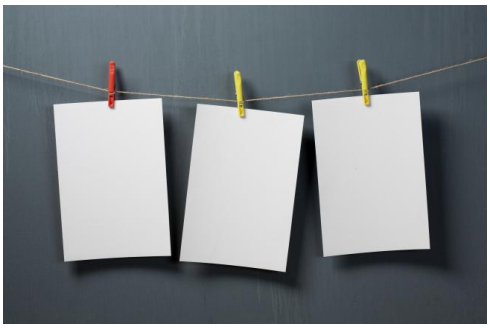
数据结构作业报告：第一次作业

算法分析与排序问题

课程
姓名
专业班级
学号
邮箱
提交日期



目录

数据结构作业报告：第一次作业

算法分析与排序问题

此报告为第一次数据结构课程作业报告，内容包含作业题目的程序实现、结果展示与算法分析。作业中程序代码采用 Java 语言实现并在 Windows 环境下测试，语言及运行时环境参数如下：

处理器及内存	CPU: Intel Core i7-6700HQ @2.60GHz RAM: 8.00 GB
运行操作系统	Windows 10 Pro x64 (Version: 10.0.10586)
Java 运行环境	Java(TM) SE Runtime Environment (build 1.8.0_74-b02)

1 Fibonacci 数列求值

1.1 作业题目

> *将课本 P22 页的 2.3 题中提供的两个程序分别实现。完成如下内容：*
> *1) 给出 n 从 1 到 90,两个程序分别计算所需要的时间，将这些时间结果用图形的方式展示出来。*
> *2) 尝试用渐进性分析方法给出两个算法的时间增长率，结合 1）的结果给出总结。*

1.2 程序实现

分别采用迭代与递归两种方式实现 Fibonacci 数列的计算。为便于测试，使不同的计算方法实现了相同的接口，实现代码如下。

- 迭代法计算 Fibonacci 数列数值：

```
public class FibonacciCalculatorIterativeImpl implements FibonacciCalculator {
    @Override
    public long calc(int n) {
        if (n <= 0) throw new IllegalArgumentException("The index of Fibonacci
Number must be a positive integer");
        if (n >= 93)
            throw new ArithmeticException("The 93rd and up Fibonacci Numbers
cannot be hold in long type");
        long cur = 1L, prev = 1L;
        for (int i = 3; i <= n; i++) {
            cur = prev + cur;
            prev = cur - prev;
```

```
        }
        return cur;
    }
}
```

- 递归法计算 Fibonacci 数列数值

```
public class FibonacciCalculatorRecursiveImpl implements FibonacciCalculator {
    @Override
    public long calc(int n) {
        if (n <= 0) throw new IllegalArgumentException("The index of Fibonacci
Number must be a positive integer");
        if (n >= 93)
            throw new ArithmeticException("The 93rd and up Fibonacci Numbers
cannot be hold in long type");

        if (n == 1 || n == 2) {
            return 1L;
        }
        return calc(n - 1) + calc(n - 2);
    }
}
```

- 递归法计算 Fibonacci 数列数值（使用“记忆”优化）

```
public class FibonacciCalculatorRecursiveWithMemoImpl implements
FibonacciCalculator {

    private static HashMap<Integer, Long> memory = new HashMap<>();

    @Override
    public long calc(int i) {
        if (i <= 0) throw new IllegalArgumentException("The index of Fibonacci
Number must be a positive integer");
        if (i >= 93)
            throw new ArithmeticException("The 93rd and up Fibonacci Numbers
cannot be hold in long type");
        if(memory.containsKey(i)) return memory.get(i);
        long result = i <= 2 ? 1L : calc(i - 1) + calc(i - 2);
        memory.put(i, result);
        return result;
    }
}
```

1.3 程序运行结果与时间统计

运行并统计上述各方式计算 Fibonacci 数列第 n 项数值运行时间，得到下面运行结果。其中直接采用递归法测量运行时间时，当计算的项数较大时，n 由于运行时间过长，无法测量结果，但在该运行过程中，时间增长规律近似遵循与 Fibonacci 数列数值的增长，从而通过代数计算估计了当 n 取值较大时的运行时间，在曲线图（图 1-1**Error! Reference source not found.**）中已标注。

部分 n 值下，上述三种不同方式计算 Fibonacci 数列的 Java 程序在计算时所用时间如下表所示：

表 1-1 Fibonacci 数列求值运行时间表[1]

n	5	10	15	20	25	50	80
迭代法	0.001186	0.001185	0.00158	0.001185	0.001185	0.854519	0.001975
递归法	0.00079	0.004741	0.046617	0.270618	0.749038	80.63 秒	2.973 年（估）[2]
递归法（缓存）	0.00079	0.001185	0.015407	0.00079	0.002765	0.00158	0.001185

在 n 值取值为 1 至 92 的范围中，不同计算方法求值所需时间变化如下曲线图[3]所示：

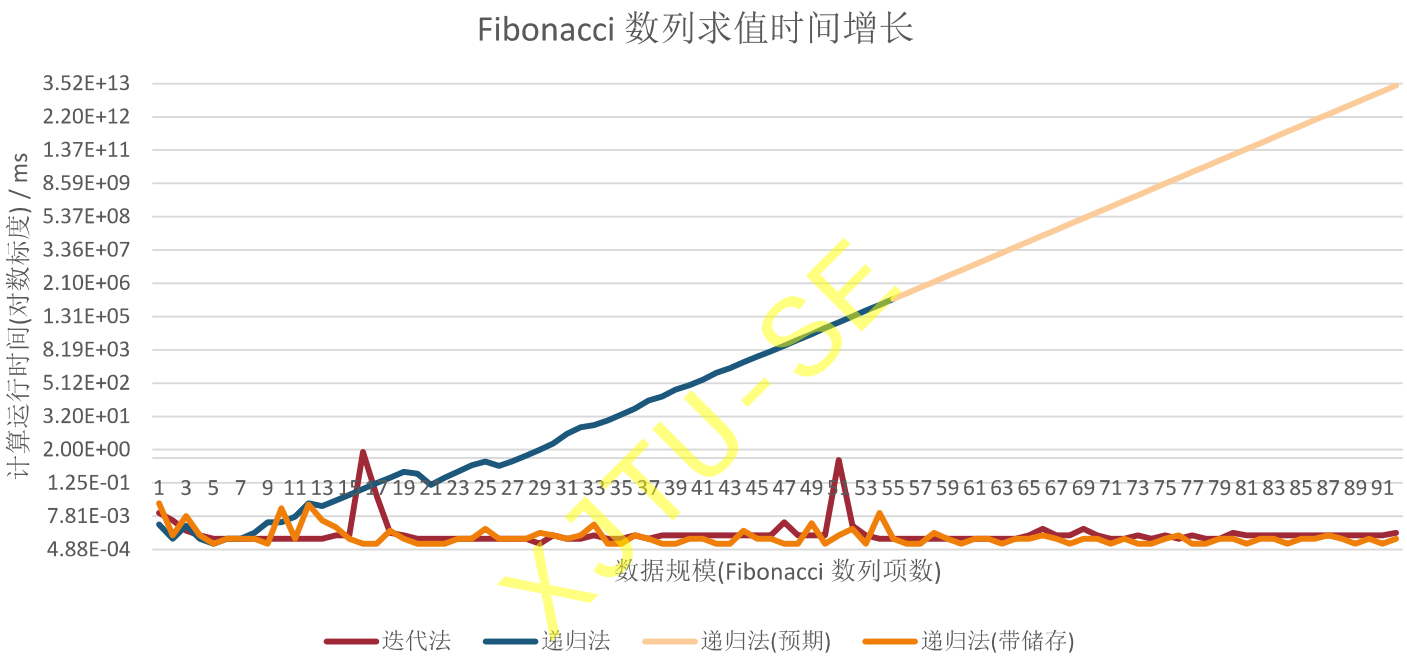


图 1-1 Fibonacci 数列求值时间增长

值得指出的是，递归法运行时间非常慢，在上曲线图中，显示的估计值部分，当 n 取值为 92 时，预计的运行时间为$3.02 \times 10^{13}\ ms$，相当于 957.6 年，接近一个世纪。

[1] 表中数值未作标注均为毫秒，时间统计采用 Java 中 System.nanoTime 方法进行测量，该方法返回值单位为纳秒，此处转换为毫秒。

[2] 由于时间限制，该项运行时间无法直接测量，此处结果显示为根据 n 值较小时数据用 Fibonacci 数列增长的方式进行估计所得。

[3] 同上，对于 n 大于 55 时，计算结果使用估计值进行绘图，并用不同颜色加以区分。

1.4 算法分析

对于上述程序，当采用迭代法时，时间复杂度为$O(n)$，当采取递归法时，时间复杂度将达到$O(2^n)$[1]，而采用带有存储的递归法时，在最糟糕的情况下完全没有存储需，则要存储前 n 项结果，时间复杂度为$O(n)$，当在最好的情况下所需的数值已经得到缓存，只需要读取即可，时间复杂度将为$O(1)$。

在图 1-1 中，通过使用对数标度，不同算法的差距十分明显，其中递归法的速度非常缓慢，对于指数式增长的算法很难应用于 n 稍大的场合。带有缓存的递归法与迭代法时间消耗均很少，实际测量中环境因素将会对计算结果产生较大的影响。

2 数据搜索

2.1 作业题目

> *完成课本 P45 页的 3.12（项目设计）中的 3.2 题。完成如下内容：*
> *1) 按题目要求给出运行时间结果的图形表示；*
> *2). 观察 1），在什么情形下顺序搜索要快于二分搜索？尝试给出你认为的原因。*

2.2 程序实现

顺序查找与二分查找实现均较为简单，均可以采用单个循环的方式进行。其中二分查找要求数组为有序状态，否则采用调用 JDK 中 Arrays.sort 的当时进行排序。代码实现如下[2]：

二分查找[3]：

```
public class SearchBinaryImpl implements Search {
    @Override
    public int search(int[] array, int valueToSearch) {
        Arrays.sort(array);
        return searchSorted(array, valueToSearch);
    }

    @Override
    public int searchSorted(int[] sortedArray, int valueToSearch) {
        int l = 0, u = sortedArray.length, t;
        while(l <= u){
```

[1] 关于这一点，一方面可以由在对数标度的曲线图中以线性直线形式增长的时间复杂度看出，另一方面，数学上可以推导求出 Fibonacci 数列任一项数值满足一个指数函数，而分析该递归调用，每次求第 n 项结果时总需要先求得 n-1 项和 n-2 项的结果，从而时间复杂度上也满足于 Fibonacci 数列数值的关系，亦即服从指数关系增长。

[2] 为便于测试，此处代码在实现时同样采用实现接口的方式。

[3] 对于二分查找中，对于有序数组的特殊效率，可以通过调用 searchSorted 方法已减少一次对于数组有序性的检查，提高效率。