

第一节 计算

2016年7月15日 20:41

第一章 绪论

第一节 1.1.1 计算

计算是本课程直接研究对象和内容，也是研究的目的和最终目标；

计算的本质的内在规律，一般方法，典型技巧。

进行有效的，高效的计算，同时低廉的资源消耗。

1.1.2 绳索计算机

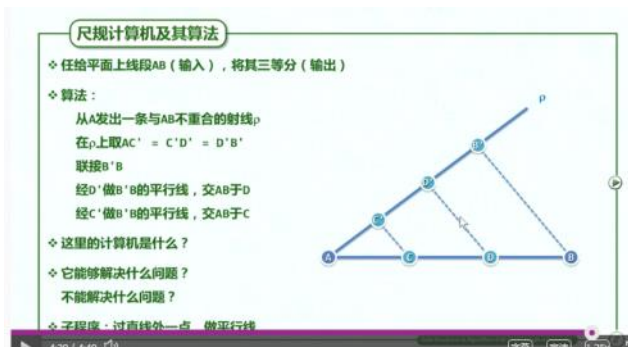
计算机：12条等长的绳索；计算：用绳索求解过直线外一点垂线的方法。



屏幕剪辑的捕获时间: 2016/7/15 21:00

1.1.3 尺规计算机

计算机：尺规；计算：做三等分点的作图过程；



屏幕剪辑的捕获时间: 2016/7/15 21:07

1.1.4 算法

算法：特定计算模型下，旨在解决特定问题的指令序列。

| | |
|---------|-----------------|
| 特点要素：输入 | 待处理信息（问题） |
| 输出 | 经处理信息（答案） |
| 正确性 | 可解决指定问题 |
| 确定性 | 语义明确 |
| 可行性 | 操作可实现，且在常数时间内完成 |
| 有穷性 | 基本操作次数 |

1.1.5 有穷性

程序！=算法（死循环，栈溢出）

$$Hailstone序列: \begin{cases} \{1\} & n \leq 1 \\ \{n\} \cup Hailstone(n/2) & n \text{ 为偶} \\ \{n\} \cup Hailstone(3n+1); & n \text{ 为奇} \end{cases}$$

1.1.6 好算法

效率：速度尽可能快；存储空间尽可能少；

正确，健壮（处理非法输入），可读

第二节 计算模型

2016年7月15日 22:13

1.2.1 性能测度

1.2.2 问题规模

算法分析：主要有两个方面：

正确性：算法功能与问题要求是否一致

成本：运行时间 + 所需存储空间

复杂度与问题规模有关，归纳可采用：划分等价类。

一般问题实例的规模，往往是决定计算成本的主要因素。（规模扩大，计算成本上升）

1.2.3 最坏情况

定义 $T(n)$ 为规模为 n 的实例的复杂度。（然而，同一问题等规模的不同实例计算成本不尽相同），所以有以下定义：

$T(n) = \max (T(P) \mid |P| = n)$ ；即选取出等规模中最坏的情况。

1.2.4 理想模型

排除不同算法对输入规模、类型的适应性，不同程序语言，不同编译器、体系结构，操作系统等具体因素的影响，从而抽象出一个理想的平台或模型，来评价算法。

1.2.5 图灵机

无限长的纸带（Tape），读写头（Head），有限种状态（State），有限的字母表（Alphabet）

1.2.6 图灵机的实例

（读写头当前状态，当前字符，改变后字符，左右移，改变后读写头状态，）



屏幕剪辑的捕获时间: 2016/7/17 20:32

1.2.7 RAM Random Access Machine 模型（随机存储器）



1.2.8 RAM 实例

RAM: Floor

◇ 功能：向下取整的除法， $0 \leq c$ ， $0 \leq d$

$\lfloor c/d \rfloor = \max \{ x \mid d \cdot x \leq c \}$
 $= \max \{ x \mid d \cdot x < 1 + c \}$

◇ 算法：反复地从 $R[0] = 1 + c$ 中减去 $R[1] = d$
统计在下溢之前，所做减法的次数 x

0 $R[3] \leftarrow 1$ //increment

1 $R[0] \leftarrow R[0] + R[3]$ //c++

2 $R[0] \leftarrow R[0] - R[1]$ //c -= d

3 $R[2] \leftarrow R[2] + R[3]$ //x++

4 IF $R[0] > 0$ GOTO 2 //if c > 0 goto 2

5 $R[0] \leftarrow R[2] - R[3]$ //else x-- and

6 STOP //return R[0] $\leftarrow \lfloor c/d \rfloor$

| Step | IR | R[0] | R[1] | R[2] | R[3] |
|------|----|------|------|------|------|
| 0 | 0 | 12 | 5 | 0 | 0 |
| 1 | 1 | ^ | ^ | ^ | 1 |
| 2 | 2 | 13 | ^ | ^ | ^ |
| 3 | 3 | 8 | ^ | ^ | ^ |
| 4 | ^ | ^ | ^ | 1 | ^ |
| 5 | 2 | ^ | ^ | ^ | ^ |
| 6 | 3 | 3 | ^ | ^ | ^ |
| 7 | 4 | ^ | ^ | 2 | ^ |
| 8 | 2 | ^ | ^ | ^ | ^ |
| 9 | 3 | 0 | ^ | ^ | ^ |
| 10 | 4 | ^ | ^ | 3 | ^ |
| 11 | 5 | ^ | ^ | ^ | ^ |
| 12 | 6 | 2 | ^ | ^ | ^ |

传达的一个重要的概念

4

若能够整除，是否需要-1？（需要，c+1）

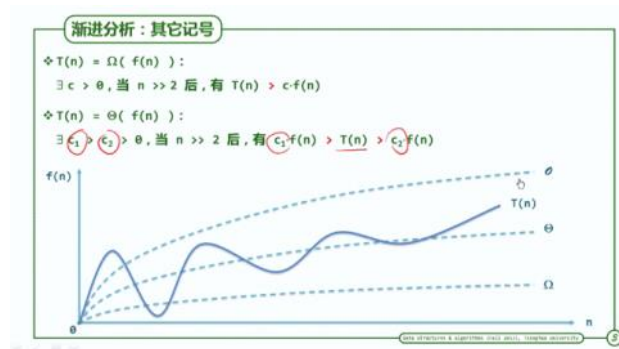
第三节 大O记号

2016年7月17日 21:02

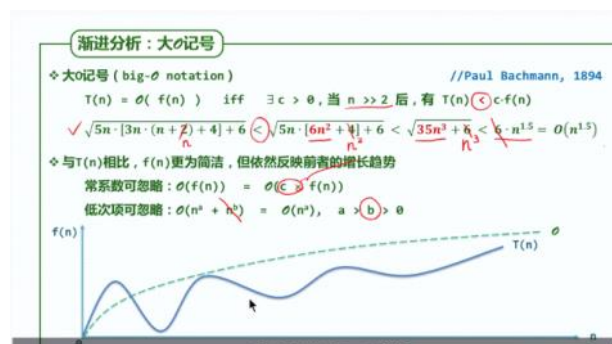
1.3.1 主流长远

评价DSA时，不要拘泥于暂时变化。观察主要的、长远的变化趋势

1.3.2



屏幕剪辑的捕获时间: 2016/7/17 21:26

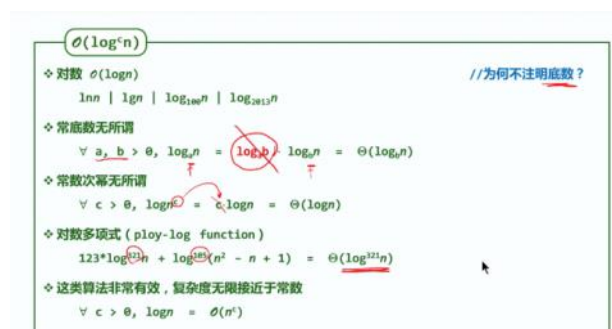


屏幕剪辑的捕获时间: 2016/7/17 21:26

1.3.3 高效解

常数复杂度

对数多项式 (对数) 复杂度



屏幕剪辑的捕获时间: 2016/7/17 21:36

1.3.4 有效解

多项式复杂度

1.3.5 难解

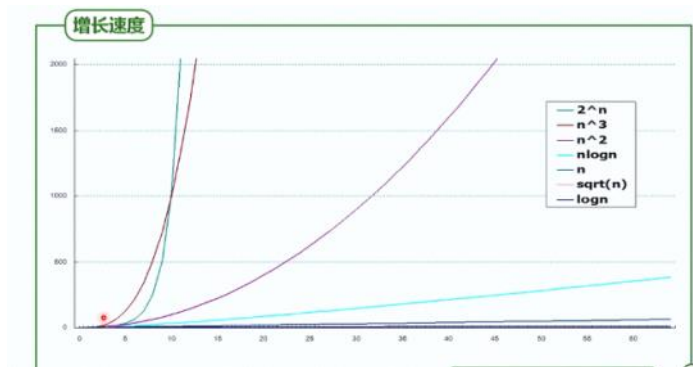
指数复杂度（通常被认为是无效算法）

指数复杂度的算法通常容易看出，但要加以改进为多项式复杂度，通常很困难。

1.3.6 例子 2-subset 子集

NP-complete（NPC）

1.3.7 增长速度



屏幕剪辑的捕获时间: 2016/7/17 22:03

第一节 算法分析

2016年7月21日 19:44

2.1.1 算法分析

C++等高级语言的基本指令，均等效于常数条RAM基本指令；在渐进意义下，二者大体相当。

分支转向: goto //算法灵魂；出于结构化考虑，被隐蔽了

迭代循环: for()、while()... //本质为“if + goto”

调用 + 递归 //本质也是goto

复杂度分析主要方法

迭代: 级数求和

递归: 递归跟踪 + 递归方程

猜测 + 验证

2.1.2 级数

级数

❖ **算数级数: 与末项平方同阶**

$$T(n) = 1 + 2 + \dots + n = n(n+1)/2 = O(n^2)$$

❖ **幂方级数: 比幂次高出一阶:**

$$\sum_{k=0}^n k^d \approx \int_0^n x^{d+1} dx = \frac{1}{d+1} x^{d+1} \Big|_0^n = \frac{1}{d+1} n^{d+1} = O(n^{d+1})$$

$T_2(n) = 1^2 + 2^2 + 3^2 + \dots + n^2 = n(n+1)(2n+1)/6 = O(n^3)$

$T_3(n) = 1^3 + 2^3 + 3^3 + \dots + n^3 = n^2(n+1)^2/4 = O(n^4)$

$T_4(n) = 1^4 + 2^4 + 3^4 + \dots + n^4 = n(n+1)(2n+1)(3n^2+3n-1)/30 = O(n^5)$

...

❖ **几何级数 ($a > 1$): 与末项同阶**

$$T_a(n) = a^0 + a^1 + \dots + a^n = (a^{n+1} - 1)/(a - 1) = O(a^n)$$

$1 + 2 + 4 + \dots + 2^n = 2^{n+1} - 1 = O(2^{n+1}) = O(2^n)$

2的n次方 同样对应于2的n次方

屏幕剪辑的捕获时间: 2016/7/21 20:03

级数

❖ **收敛级数**

$$1/1/2 + 1/2/3 + 1/3/4 + \dots + 1/(n-1)/n = 1 - 1/n = O(1)$$
$$1 + 1/2^2 + \dots + 1/n^2 < 1 + 1/2^2 + \dots = \pi^2/6 = O(1)$$
$$1/3 + 1/7 + 1/8 + 1/15 + 1/24 + 1/26 + 1/31 + 1/35 + \dots = 1 = O(1)$$

❖ 有必要讨论这类级数吗?

难道，基本操作次数、存储单元数可能是分数？某种意义上！

$(1-\lambda) \cdot [1 + 2\lambda + 3\lambda^2 + 4\lambda^3 + \dots] = 1/(1-\lambda) = O(1)$, $0 < \lambda < 1$ //几何分布

❖ 可能未必收敛，然而长度有限

$$h(n) = 1 + 1/2 + 1/3 + \dots + 1/n = \Theta(\log n) \quad //调和级数$$
$$\log 1 + \log 2 + \log 3 + \dots + \log n = \log(n!) = \Theta(n \log n) \quad //对数级数$$

它的界可以估计为logn 这是个确界

屏幕剪辑的捕获时间: 2016/7/21 20:05

2.1.3 循环

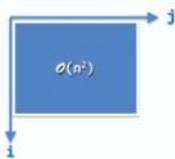
循环 vs. 级数

```

❖ for (int i = 0; i < n; i++)
  for (int j = 0; j < n; j++)
    O1Operation(i, j);

```

算术级数：
 $\sum_{i=0}^{n-1} n = n + n + \dots + n = n * n = O(n^2)$

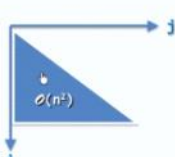


```

❖ for (int i = 0; i < n; i++)
  for (int j = 0; j < i; j++)
    O1Operation(i, j);

```

算术级数：
 $\sum_{i=0}^{n-1} i = 0 + 1 + \dots + (n-1) = \frac{n(n-1)}{2} = O(n^2)$



屏幕剪辑的捕获时间: 2016/7/21 20:10

循环 vs. 级数

```

❖ for (int i = 0; i < n; i++)
  for (int j = 0; j < i; j += 2013)
    O1Operation(i, j);

```

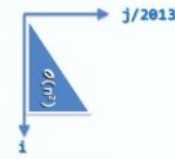
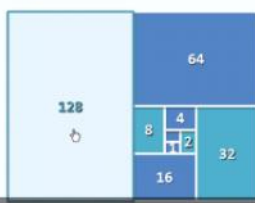
算术级数：...

```

❖ for (int i = 1; i < n; i <= 1)
  for (int j = 0; j < i; j++)
    O1Operation(i, j);

```

几何级数：
 $1 + 2 + 4 + \dots + 2^{\lfloor \log_2(n-1) \rfloor}$
 $= \sum_{k=0}^{\lfloor \log_2(n-1) \rfloor} 2^k \quad (\text{let } k = \log_2 i)$
 $= 2^{\lceil \log_2 n \rceil} - 1 = O(n)$

你已经知道 它的复杂度是多少了

屏幕剪辑的捕获时间: 2016/7/21 20:15

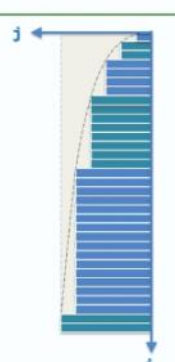
循环 vs. 级数

```

❖ for (int i = 0; i <= n; i++)
  for (int j = 1; j < i; j *= 2)
    O1Operation(i, j);

```

几何级数： $\sum_{k=0}^n \lceil \log_2 i \rceil = O(n \log n)$
 $(i = 0, 1, 2, 3 \sim 4, 5 \sim 8, 9 \sim 16, \dots)$
 $= 0 + 0 + 1 + 2*2 + 3*4 + 4*8 + \dots$
 $= \sum_{k=0}^{\lceil \log_2 n \rceil} (k * 2^{k-1})$
 $= O(\log n * 2^{\log n}) \quad (\text{CM page\#33})$



屏幕剪辑的捕获时间: 2016/7/21 20:24

2.1.4 非极端元素 + 起泡排序

起泡排序

- 问题：给定n个整数，将它们按（非降）序排列
- 观察：有序/无序序列中，任意/总有一对相邻元素顺序/逆序
- 扫描交换：依次比较每一对相邻元素，如有必要，交换之
若整整扫描都没有进行交换，则排序完成；否则，再做一趟扫描交换

```

void bubblesort(int A[], int n) { //第二章将进一步改进
    for (bool sorted = false; sorted = !sorted; n--) //逐趟扫描交换，直至完全有序
        for (int i = 1; i < n; i++) //自左向右，逐对检查A[0, n)内各相邻元素
            if (A[i-1] > A[i]) { //若逆序，则
                swap(A[i-1], A[i]); //令其互换，同时
                sorted = false; //清除（全局）有序标志
            }
    }

```

下面通过这个例子介绍

屏幕剪辑的捕获时间: 2016/7/21 20:50

非极端元素：有些算法，不论规模n有多大，可能需要执行时间是一个常数。

2.1.5 起泡排序正确性说明

起泡排序

- 问题：该算法必然会结束？至多需迭代多少趟？
- 不变性：经k轮扫描交换后，最大的k个元素必然就位
- 单调性：经k轮扫描交换后，问题规模缩减至n-k
- 正确性：经至多n趟扫描后，算法必然终止，且能给出正确解答

屏幕剪辑的捕获时间: 2016/7/21 20:57

2.1.6 封地估算-1

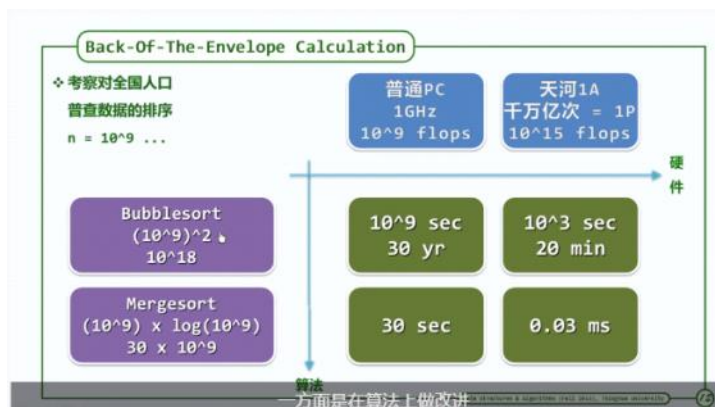
Some history

2.1.7 封地估算-2

- 1天 = 24hr x 60min x 60sec
≈ 25 x 4000 = 10⁵ sec
- 1生 = 1世纪
= 100yr x 365 = 3 x 10⁴ day = 3 x 10⁹ sec
- “为祖国健康工作五十年” ≈ 1.6 x 10⁹ sec
- “三生三世” ≈ 300 yr = 10¹⁰ = (1 googol)^(1/10) sec
- 宇宙大爆炸至今 = 10²¹ = 10 x (10¹⁰)² sec

在三生三世中的0.1秒

屏幕剪辑的捕获时间: 2016/7/21 21:11



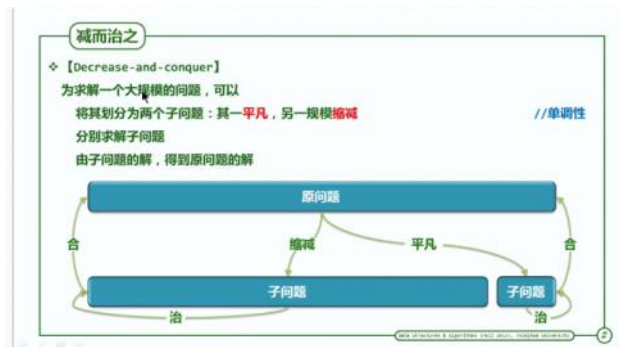
屏幕剪辑的捕获时间: 2016/7/21 21:11

第二节 迭代与递归

2016年8月7日 18:30

2.2.1 迭代与递归

2.2.2 减而治之

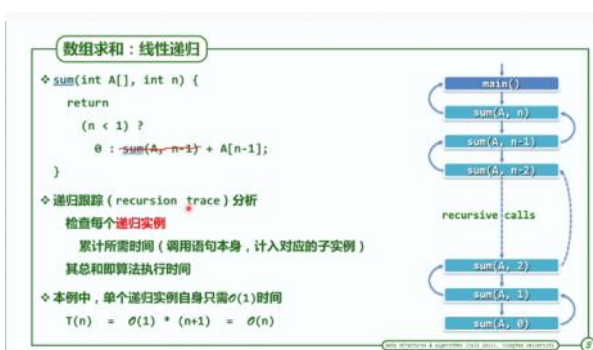


屏幕剪辑的捕获时间: 2016/8/7 20:57

2.2.3 递归跟踪

【线性递归】

检查每个递归实例累计所需时间（调用语句本身，计入相应的子实例），其中和即算法执行时间。



屏幕剪辑的捕获时间: 2016/8/7 21:15

2.2.4 递归方程

递归方程：根据算法结构得出一个递推式，进而通过求解方程得出显式的复杂度的解。

递归基：“递归基”是递归函数的一种平凡情况，只有有递归基，递归才不会一直进行下去。

2.2.5 数组倒置



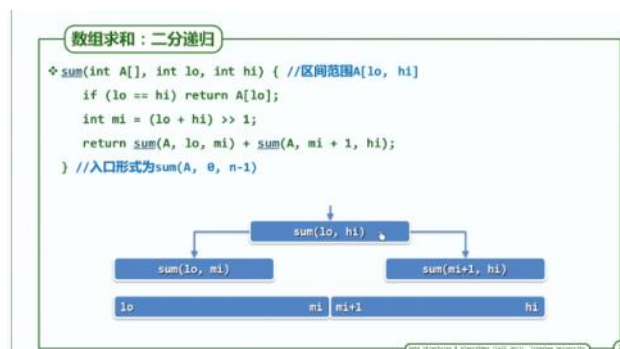
屏幕剪辑的捕获时间: 2016/8/7 21:34

2.2.6 分而治之



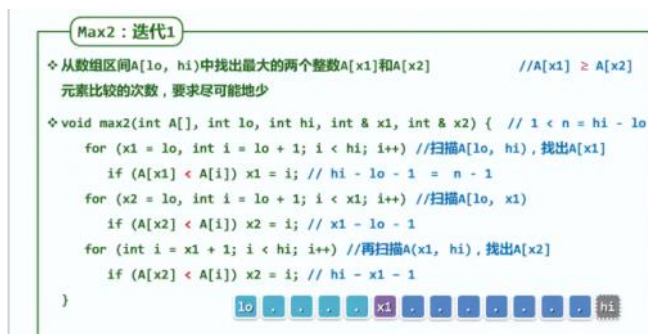
屏幕剪辑的捕获时间: 2016/8/7 21:35

2.2.7 二分递归：数组求和 二叉树图，类似于二分裂。



屏幕剪辑的捕获时间: 2016/8/7 21:48

2.2.8 二分递归：MAX2



屏幕剪辑的捕获时间: 2016/8/7 21:58

迭代 2

```

void max2(int A[], int lo, int hi, int & x1, int & x2) {
    if (A[x1 = lo] < A[x2 = lo + 1]) swap(x1, x2);
    for (int i = lo + 2; i < hi; i++)
        if (A[x2] < A[i])
            if (A[x1] < A[x2 = i])
                swap(x1, x2);
}

```

♦ 最好情况, $1 + (n - 2) * 1 = n - 1$ ✓
 ♦ 最坏情况, $1 + (n - 2) * 2 = 2n - 3$ ✗

屏幕剪辑的捕获时间: 2016/8/7 22:01

2.2.9 MAX2: 二分迭代

Max2: 递归 + 分治

```

void max2(int A[], int lo, int hi, int & x1, int & x2) {
    if (lo + 2 == hi) { /* ... */; return; } // T(2) = 1
    if (lo + 3 == hi) { /* ... */; return; } // T(3) <= 3
    int mi = (lo + hi) / 2; // divide
    int x1L, x2L; max2(A, lo, mi, x1L, x2L);
    int x1R, x2R; max2(A, mi, hi, x1R, x2R);
    if (A[x1L] > A[x1R]) {
        x1 = x1L; x2 = (A[x2L] > A[x1R]) ? x2L : x1R;
    } else {
        x1 = x1R; x2 = (A[x1L] > A[x2R]) ? x1L : x2R;
    } // 1 + 1 = 2
} // T(n) = 2 * T(n/2) + 2 <= 5n/3 - 2

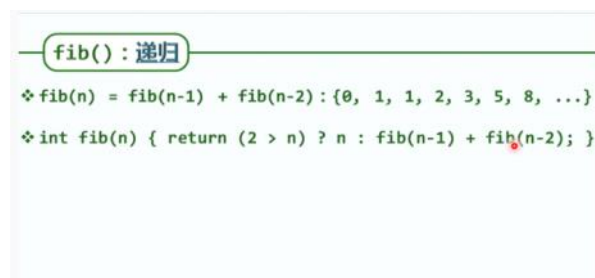
```

屏幕剪辑的捕获时间: 2016/8/7 22:14

第三节 动态规划

2016年8月13日 10:32

2.3.1 动态规划

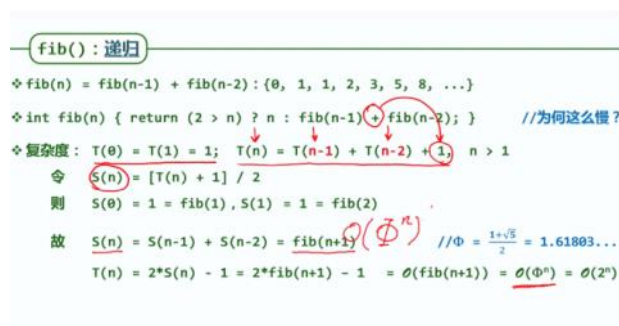


屏幕剪辑的捕获时间: 2016/8/13 10:48

动态规划可以理解为，
通过递归，找出了算法的本质，并且给出一个初步的解之后
再将其等效地转化为迭代的形式

对于斐波那契数列，我们采用简单的递归方法，解决问题所用时间非常之长，效率非常低。

2.3.2 FIB():递推方程



屏幕剪辑的捕获时间: 2016/8/13 16:24

斐波那契数列的通项公式:

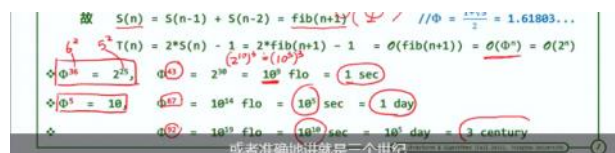
$$a_n = \frac{1}{\sqrt{5}} \left[\left(\frac{1 + \sqrt{5}}{2} \right)^n - \left(\frac{1 - \sqrt{5}}{2} \right)^n \right]$$

由复杂度 $T(n)$ 的运算，构造出 $S(n)$ 。而 $S(n)$ 恰恰是斐波那契数列的第 $n+1$ 项。由此计算复杂度。

对于斐波那契数列通项公式计算，参考

1. [线性递归数列的特征方程](#)
2. [斐波那契数列的通项公式求法](#)

2.3.3 FIB():封地估算



2.3.4 FIB(): 递归跟踪

应用实例

考察一个上台阶的过程

在某一个楼梯

允许你每次走一步

也可以每次走两步

那么请问上到某一级

比如说第6级台阶的时候

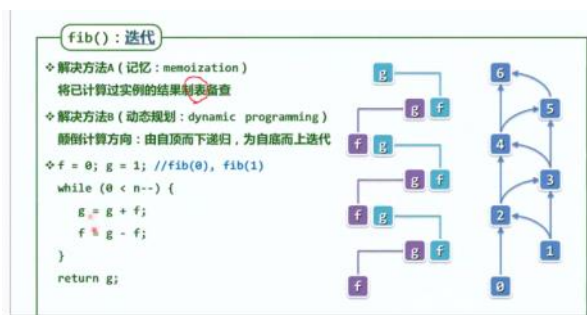
总共有多少种走法?

(这个数字就是对应的,

比如 第6阶的话

就是第6项Fibonacci数)

2.3.5 FIB(): 迭代

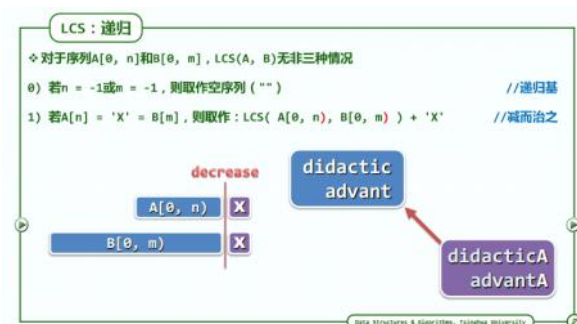


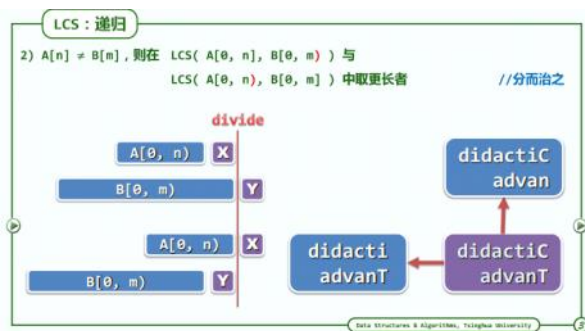
2.3.6 LCS: 最长公共子序列

子序列: 由序列中若干字符, 按原相对次序构成。

最长公共子序列: 可能有多个, 可能有歧义 (重复字符)

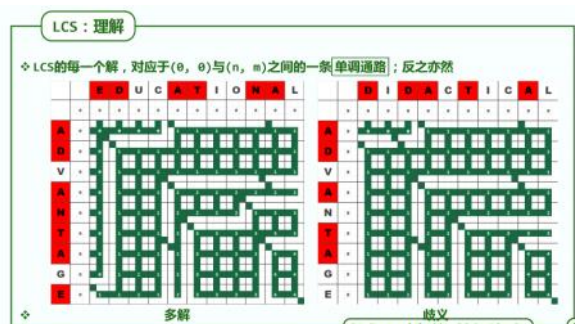
2.3.7 LCS: 递归





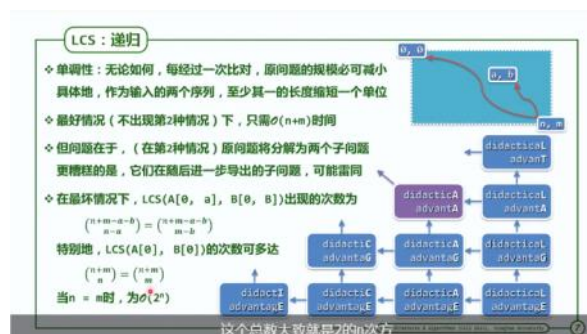
屏幕剪辑的捕获时间: 2016/8/14 17:20

2.3.8 LCS: 理解



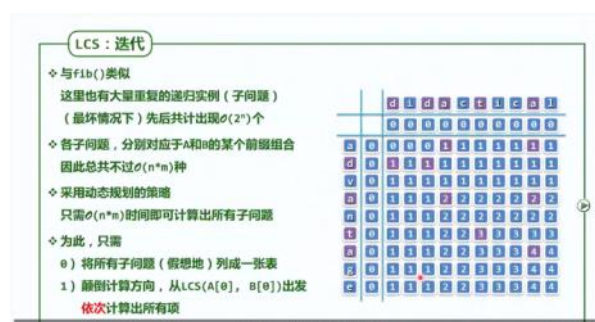
屏幕剪辑的捕获时间: 2016/8/14 21:15

2.3.9 LCS: 复杂度



屏幕剪辑的捕获时间: 2016/8/14 21:21

2.3.10 LCS: 动态规划



解决LCS问题，同样可以参照斐波那契数列的解决办法，改变考虑方向，化递归为迭代。

递归：设计出可行且正确的解

动态规划：消除重复计算，提高效率