程序设计实习（实验班-2023春）
经典算法设计思想：递归

授课教师：姜少峰
助教：张宇博 楼家宁

Email: shaofeng.jiang@pku.edu.cn

recursion:
see “recursion”

递归

- 递归是用f自身来定义函数/程序f的方法
- 比如Fibonacci数列

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

- 无循环依赖：f(a) = f(b), f(b) = f(c), f(c) = f(a)这种不是良定义的
- 边界条件充分：例如对Fibonacci数列需指定两个初值$f(1) = f(2) = 1$

递归的本质?

- 递归仅仅是一种表达方法，并不会产生额外的能力
- 即：任何递归算法也可以不用递归去实现，并且代价仅是常数倍
- 然而，该“表达”方法经常有利于算法的设计和表达，因此成为一种算法设计思想

重复计算的问题

```c
// 等差数列
int f(int n)
{
    if (n <= 1) return 1;
    return f(n - 1) + 1;
}
```

```c
// Fibonacci数列
int f(int n)
{
    if (n <= 2) return 1;
    return f(n - 1) + f(n - 2);
}
```

- 这两个关于f的程序，如果调用f(n)，程序的复杂度分别是多少（关于n）？
- 是线性的吗?

关于Fibonacci程序复杂度的分析

```c
// Fibonacci数列
int f(int n)
{
    if (n <= 2) return 1;
    return f(n - 1) + f(n - 2);
}
```

- 设g(n)代表上述f运行f(n)的计算次数

$$g(n) = \begin{cases} O(1) & n \leq 2 \\ g(n-1) + g(n-2) & \text{otherwise} \end{cases}$$

- 解得$g(n) = O(f(n)) \approx O(1.618^n)$

一般高效实现方法：记忆化（自顶向下）

- 直接面对重复计算的挑战：每个f(i)的值一旦计算就记录下来，以后只需要调取

```
// Fibonacci数列 -- 记忆化版本
int f(int n, int value[], bool evaluated[])
{
    if (n <= 2) return 1;
    if (evaluated[n]) return value[n];
    value[n] = f(n - 1, value, evaluated) + f(n - 2, value, evaluated);
    evaluated[n] = true;
    return value[n];
}
int main()
{
    // 需要预先开好存放值和访问记录信息的数组
    int value[100];
    bool evaluated[100];
    for (int i = 0; i < 100; i++) evaluated[i] = 0;
    cout << f(10, value, evaluated) << endl;
}
```

value[i]记录f(i)的值，evaluated[i]记录f(i)是否被计算/访问过

如果计算过，则直接取计算结果

注意递归调用依然是调用f（而不是尝试调用value）

记忆化方法的存储实现

- 刚刚的程序开的value/evaluated数组其实是一种字典
  - 字典：存key-value，用key索引
  - 这里key就是递归调用的参数n，value就是这个key对应的值f(n)

利用哈希表等字典数据结构

- 有时需要一般的字典数据结构而不是数组：要存的东西的域很大/稀疏/非int
- 有时可以把非int映射成int：例如n元集合的2^n个subset可以用n位二进制表示

每一位的0/1代表对应元素取/不取
例如10010代表选取第2、5位
可以使用位运算进行集合操作

标准库中的字典

- map：带order（支持查找前趋/后继等）
- unordered_map（速度更快，但是无序）

也可以采用迭代式实现

- 一种高效实现方法是找出一种求值顺序，使得求值过程不会重复
- 缺点：

例如有些递归定义在图等复杂结构上，
递推顺序很难预先确定依赖于输入数据

  - 代码改动大，需要具体问题具体分析
  - 很多时候依然需要开额外数组，无本质优化

```
// Fibonacci数列 -- 迭代版本
int f(int n)
{
    if (n <= 2) return 1;
    int an_1 = 1, an_2 = 1, an = 0;
    for (int i = 3; i <= n; i++)
    {
        an = an_1 + an_2;
        an_2 = an_1;
        an_1 = an;
    }
    return an;
}
```

动态规划

- 动态规划 = 把问题写成递推式，然后用记忆化/迭代方法求解
- 相对于Fibonacci等数列的递推式，动态规划通常解决最优化（最大/最小）问题

这个“动态”一开始指代的是需要考虑非常多状态，从基态到终态演进来求解问题

- 动态规划 = dynamic programming （DP）

不是编程，而是“规划”/“优化”的意思
类似的有linear programming

典型例子：最长上升子序列 LIS

- 输入是n个整数$a_1, \ldots, a_n$，找一个最长的（未必连续）单调上升子序列
- 设$f(i)$代表以a_i为起点的最长上升子列长度
- $f(i) = 1 + \max_{j \geq i : a_j > a_i} f(j)$, 边界条件$f(n) = 1$，最后解是 $\max_{1 \leq i \leq n} f(i)$
- 如果要输出f(i)对应的解：找到使得递推式取等号的j，从j递归

代码实现

```
// 初始化 value = -1
int LCS(int a[], int i, int n, int value[])
{
    if (i == n - 1) value[i] = 1;
    if (value[i] != -1) return value[i];
    value[i] = 1;
    for (int j = i; j < n; j++)
    {
        if (a[j] > a[i])
        {
            int t = LCS(a, j, n, value);
            value[i] = max(value[i], t + 1);
        }
    }
    return value[i];
}
```

```
void output_solution(int a[], int value[], int i, int n)
{
    cout << i << endl;
    if (value[i] != 1)
    {
        for (int j = i; j < n; j++)
        {
            if (a[j] > a[i] && value[i] == value[j] + 1)
            {
                output_solution(a, value, j, n);
                break;
            }
        }
    }
}
```

取等号

编辑距离

- 考虑两个长度（至多）是n的字符串S和T
- $f(i,j)$表示将$(S_1,\ldots,S_i)$变成$(T_1,\ldots,T_j)$需经过最少多少如下三种之一的操作：
  - 插入一个字符，删除一个字符，替换一个字符

$$f(i,j)=\begin{cases} f(i-1,j-1) & S_i=T_j \\ \min\begin{cases} f(i-1,j)+1 \\ f(i,j-1)+1 \\ f(i-1,j-1)+1 \end{cases} & S_i\neq T_j \end{cases}$$

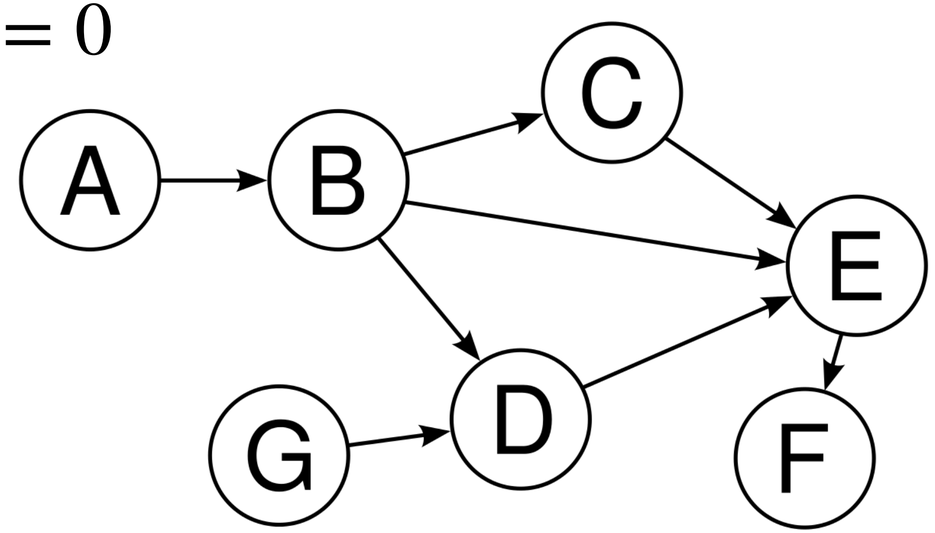
初值$f(i,0),f(0,j)$

有向无环图求（单源）最短路

- 给定有向无环图G(V, E)，边权函数$w : E \to \mathbb{N}$，以及起点u
- 对于节点v，$f(v)$代表起点u到v的最短路长度
- $f(v) = \min_{v' \in V:(v',v) \in E} \{f(v') + w(v', v)\}$初值$f(u) = 0$
- 如何计数最短路条数?

先算f，然后$g(v) = \sum_{v':(v',v) \in E,\ f(v) = f(v') + w(v',v)} g(v')$

可以用于计数一般DP问题的方案数



TSP

- 输入$\mathbb{R}^d$上n个点，求将所有点连接起来的最短回路
- 暴力：$O(n!)$，因为任何一个回路可以对应于一个排列

TSP的动态规划精确算法 (Held-Karp, 1962)

复杂度： $O(2^n \cdot n^2)$

- 假设回路从1号点开始和结束
- 对于$S \subseteq \{2,\ldots,n\}$和$u \notin S$且$u \neq 1$，设$f(S,u)$为以1为起点，访问S所有点，最后落在u点的最短路径长度
- 设某个$S = \{s_1, \ldots, s_k\}$，那么$f(S,u) = \min\limits_{1 \le j \le k} \{f(S \backslash \{s_j\}, s_j) + \mathrm{dist}(s_j, u)\}$
- 设$T = \{2,\ldots,n\}$。最后解是$\min\limits_{2 \le i \le n} \{f(T \backslash \{i\}, i) + \mathrm{dist}(i,1)\}$
- 优化：S可以编码成一个n位的二进制数，对于比较小的n可以在int32/int64内

* $(1+\epsilon)$-近似2维TSP（Arora, 1998）概述

Gödel Prize哥德尔奖

- PTAS = polynomial time approximation scheme
  - 对任何$0 < \epsilon < 1$，有$(1+\epsilon)$-近似的多项式时间算法（可以是$n^{1/\epsilon}$）

- 先证明：存在一个结构简单的TSP，称为STSP，使得$\text{STSP} \leq (1+\epsilon)\text{TSP}$
- 用动态规划找到最优的STSP

事实上STSP是定义在一个随机几何结构上的，并且

$\Pr[\text{STSP} \leq (1+\epsilon)\text{TSP}] \geq 0.1$

经典递归算法

快速幂

- 给定正整数$a \leq n$，求a^n（假设所有算术运算计单位代价）
- 朴素：依次计算$a, a^2, \ldots, a^n$，总共$O(n)$时间
- 递归：$O(\log n)$时间

事实上这才是“线性时间”，因为输入规模是$O(\log n)$的

$$a^n = (\underbrace{a \times \ldots \times a}_{n/2})^2 \quad \Rightarrow \quad f(n) = \begin{cases} f(n/2)^2 & \text{even } n \\ f((n-1)/2)^2 \cdot a & \text{odd } n > 1 \\ a & n = 1 \end{cases}$$

快速幂与Fibonacci数列

- Fibonacci数列：$a_n = \begin{cases} a_{n-1} + a_{n-2} & n \geq 3 \\ 1 & n = 1,2 \end{cases}$
- 观察：$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} a_{n-2} \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} a_{n-1} \\ a_n \end{bmatrix}$
- 设$A = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$，则对$n \geq 3$，$\begin{bmatrix} a_{n-1} \\ a_n \end{bmatrix} = A^{n-2} \begin{bmatrix} 1 \\ 1 \end{bmatrix}$
- 用快速幂求A^{n-2}可以$O(\log n)$时间得到结果

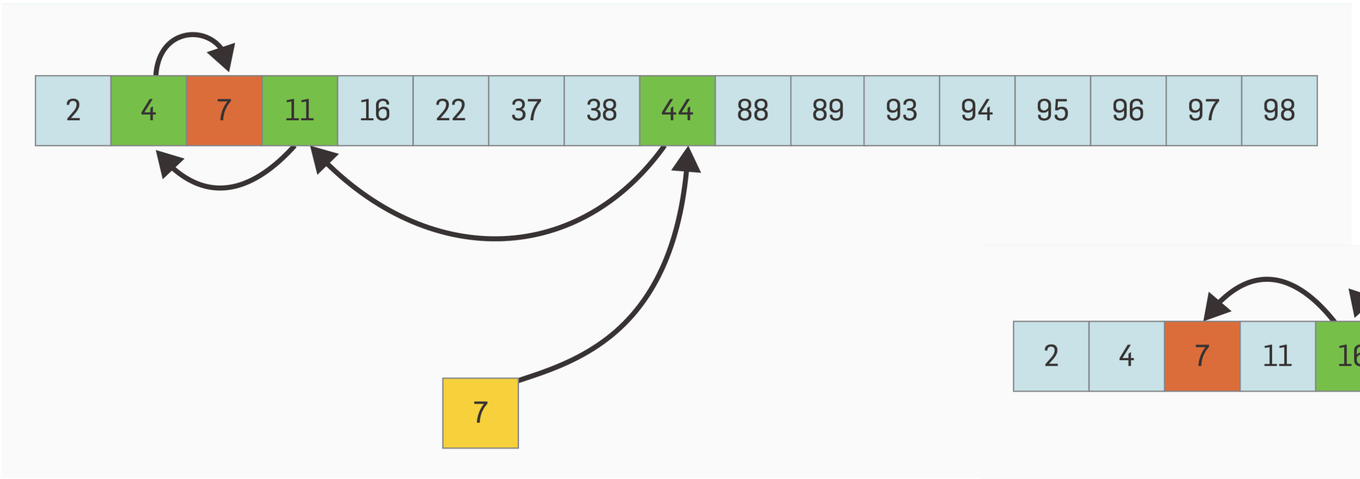
二分查找

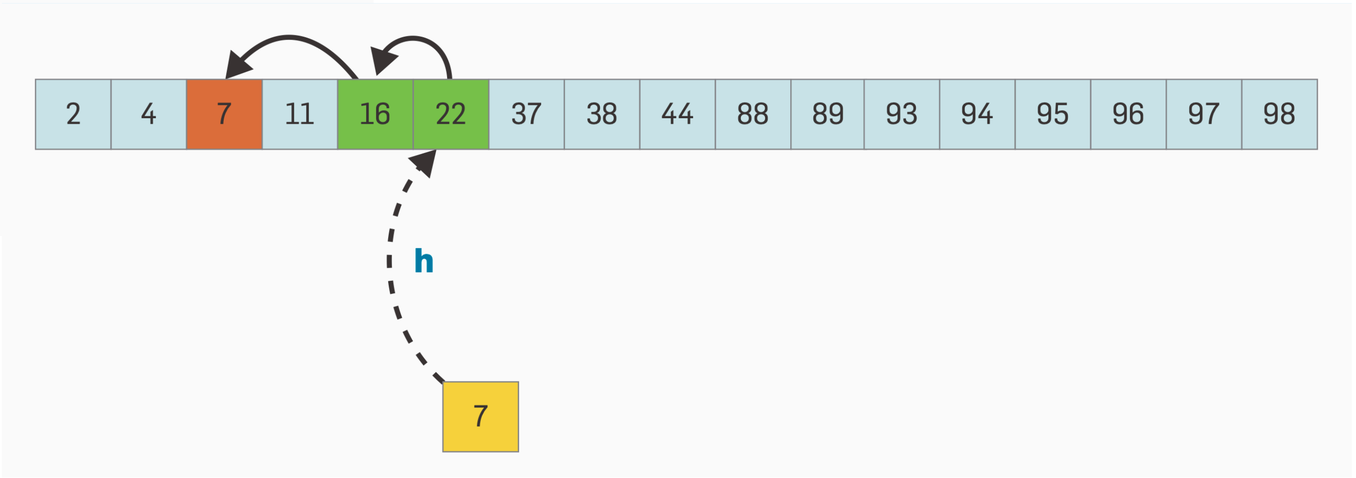
- 输入是升序数组$a_1, \ldots, a_n$以及一个关键字k，输出满足$a_i = k$的i
- 更进一步：输出满足$a_i \geq k$的最小i
- 复杂度$O(\log n)$

```
// 搜索闭区间[low, high]
int search(int a[], int low, int high, int k)
{
    if (low == high) return low;
    int mid = (low + high) / 2;
    if (a[mid] >= k)
        return search(a, low, mid, k);
    return search(a, mid + 1, high, k);
}
```

* 带预测信息的二分查找

- 另给出一个下标h代表对于最后答案i的猜测，要求$O(\log(|h-i|))$复杂度





* 带预测信息的算法设计分析框架（LV框架）
(Lykouris-Vassilvitskii, JACM' 21)

预设：

- 算法可以得到一个不可信的预测，并且误差是η

算法并不知道η

一致性：

- 当$\eta \to 0$时算法表现的接近于最优

例如二分查找中复杂度变成O(1)

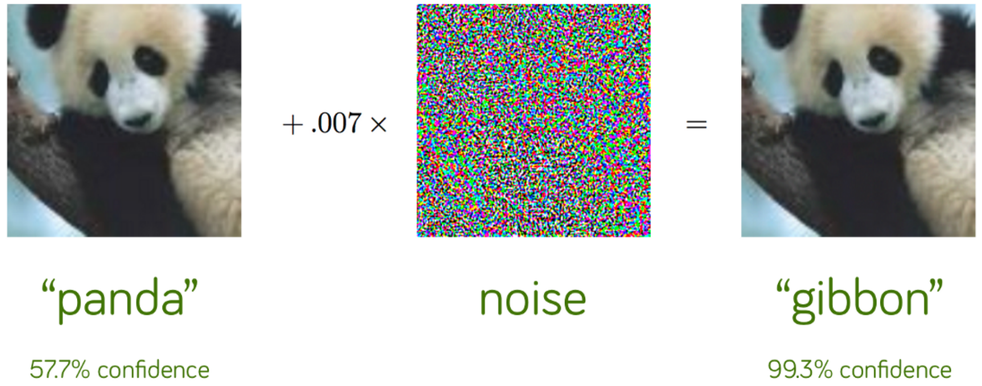
鲁棒性：

- 当$\eta \to \infty$时算法依然（几乎）达到不利用预测时一致的最坏情况表现

二分查找中即使h预测完全错误，仍可达到$O(\log n)$复杂度

* 考虑LV框架的意义

机器学习的鲁棒性问题



LV框架提供了一种直接面对下游应用，绕过鲁棒性问题的思路

分治思想

分治思想

- 将原问题划分为若干个子问题，每个子问题独立解决，将子问题答案汇总

- 与动态规划的区别：一般不需要记忆化，追求平衡划分以限制递归次数

快速排序

- 输入$A = \{a_1, \ldots, a_n\}$（假定互不相等），输出一个序列，对应A从小到大排序
- 随机选取$1 \leq p \leq n$作为pivot
- 设$A_{<p} := \{a_i \in A : a_i < a_p\}$，类似定义$A_{\geq p}$
- 返回$\text{quicksort}(A_{<p}) \oplus \text{quicksort}(A_{\geq p})$

⊕代表拼接

- $\mathbb{E}[|A_{<p}|] = \frac{1}{n}\sum_{i=1}^{n} i \leq \frac{n}{2}$

实际实现可以用median trick多sample几个p，然后选a_p居于中位的p（median trick），这样可以较小代价迅速缩小子问题规模

* Dual-pivot快速排序

- dual-pivot：选2个随机pivot的快速排序，将数组分成三个部分
  - 设pivot $A_{p_1} \leq A_{p_2}$，则分成$A_{<p_1}$，$A_{p_1 p_2}$，$A_{>p_2}$
- 对于$O(n \log n)$的总复杂度没有帮助
- 但是一些精心实现可以利用CPU的缓存等达到比single-pivot更好的实际效果（Vladimir Yaroslavskiy, 2009），并且从Java 7开始就是其标准库排序实现

一些研究表明，dual-pivot的总CPU工作量反而更大了，但是dual-pivot带来的更少的cache miss显著降低了IO代价

* Multi-pivot快速排序与分布式排序算法

- 现代分布式计算的典型设定：
  - 数据n个整数，分布式存储在$O(n^{1-\alpha})$个机器上，每个机器有$s = O(n^{\alpha})$空间
  - 每轮：每个机器可以收发$O(s)$的数据，并且在本地做一些运算
  - 设计的算法需要让轮数尽量小
- Single-pivot快速排序可以在$O(\log n)$轮实现

* 基于Multi-pivot快排的O(1)轮算法

- 考虑特殊情况$s = O(\sqrt{n})$的每机器存储
- 基于Multi-pivot的分布式快排：
  - 1号机器生成$\sqrt{n}$个随机pivot，然后广播给所有机器
  - 这些pivot将所有n个元素均匀分成了$\sqrt{n}$条长度是$\sqrt{n}$的段，每段对应1台机器
  - 每台机器将自己的数据根据与pivot的大小关系发到对应的段/机器
  - 每台机器直接本地排序

如何在O(1)轮做广播?

最后的结果是一个分布式存储的有序数组

* 关于刚刚提到的模型

- 叫做massively parallel computing，是MapReduce框架的理论对应
  - 数据n个整数，分布式存储在$O(n^{1-\alpha})$个机器上，每个机器有$s = O(n^{\alpha})$空间
  - 每轮：每个机器可以收发$O(s)$的数据，并且在本地做一些运算
  - 设计的算法需要让轮数尽量小

* 分治的重点：平衡划分

- 树的balanced separator：树$T(V, E)$上的一节点u，使得去掉u后的连通分量大小均不超过$2/3 \mid V \mid$
- 已知：给定n点的树，存在一个$O(n)$时间算法可以找到这样的separator点
- 应用：回答树上两点间的（最短路）距离
  - 类似快速排序的划分：每次找到这个separator，进行划分，共$O(\log n)$层

* 树上利用Balanced Separator

特别地，最短路也经过u

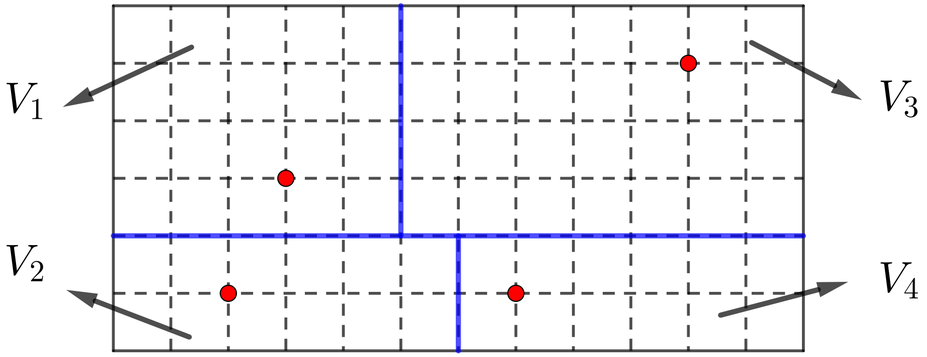
- separator的性质：u是separator，两个去u后不同分量的点互达必然经过u
- 因此可以预处理并存储到u的最短路（递归进行）
- 要想求x y两点的最短路：
  - 找到第一次separate x和y的separator点u
  - 找到x和y分属的T_1, T_2分量（去掉u及之前递归步骤的separator后的）
  - 返回$\mathrm{dist}(x, u) + \mathrm{dist}(u, y)$

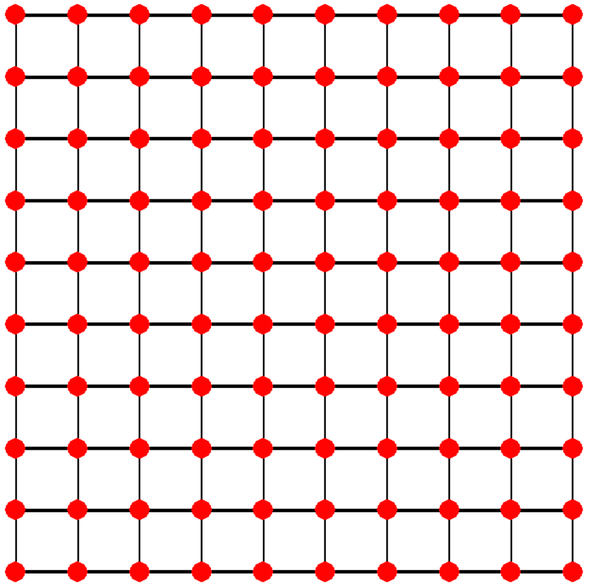
* 平面图的Balanced Separator

平面图

- 存在$\sqrt{n}$大小的vertex separator
- $O(1)$大小的shortest-path separator

Mikkel Thorup, JACM 04





- 可以利用这些separator分治，例如做快速最短路查询

练习题

- http://cssyb.openjudge.cn/recursion23/
- 选做，不计分数