算法总复习

- 第一章 算法
  - 算法定义
  - 算法的特征
  - 算法的描述（伪代码）
  - 算法与程序的联系与区别
- 第二章 算法效率分析的基础
  - 算法的分析
  - 评估运行时间的方法
  - 时间复杂度的理论分析
  - 增长规模
  - 最好、最坏、平均时间复杂度
  - 算法分析框架
  - 渐进时间复杂度
  - 算法的比较
  - 非递归算法的时间复杂度分析
  - 递归算法的时间复杂度分析
  - **主定理**
- 第三章 蛮力法
  - 选择排序
  - 冒泡排序
  - 字符串匹配
  - 两点间距离
  - 多项式求值
  - 旅行家算法（TSP）npHard
  - 背包问题 npHard
  - 时间安排问题
  - np 困难问题
- 第四章 递归算法
  - 迭代与递归
  - C(n)的计算
- 第五章 分治法
  - 分治法的思想
  - 分治法三步骤
  - 平衡(balancing)子问题的思想
  - 大数相乘
  - 矩阵乘法
  - 二分搜索
  - 归并排序
  - 快速排序
  - 棋盘覆盖
- 第六章 减治法
  - 减治3种形式
  - 插入排序
  - 深度优先遍历
  - 回溯法
  - 广度优先遍历
  - 分支界限法
  - 拓扑排序
- 第七章 变治法
  - 分 减 变的区别：
  - 变治法三个方式：
  - 预排序
  - 高斯消去法
  - 堆和堆排序

算法总复习

第一章 算法

算法定义

算法是解决特定问题的一种特定**方法或过程。**是若干指令的有穷序列。

三要素：操作，控制结构，数据结构

算法的特征

- 输入
- 输出
- 确定性
- 有限性
- 可行性

算法的描述（伪代码）

是自然语言与程序语言的结合，精确而简洁。满足以下要求：

- 省略变量声明
- 用缩进表示while if for语句
- 赋值用<-

算法与程序的联系与区别

1. 区别：
    - 描述上：

      算法用自然语言，伪代码或流程图、程序设计语言、盒图、PAD图描述

      程序用特定的编程语言编码，用特定的机器执行
    - 执行上：

      算法：有限步骤

      程序：可以无限执行下去
    - 定义上:

      算法是解决问题的逐步的大纲或流程图

      程序是一个基于算法的问题的解决方案的实现代码
2. 联系：
    - 程序是算法用某种程序设计语言的具体实现
    - 程序和算法都是有限指令序列
    - 程序=算法+数据结构

第二章 算法效率分析的基础

算法的分析

- 运行时间
- 输入规模
- 增长趋势
- 最好/坏/平均效率

评估运行时间的方法

- 测量运行时间（不好，因为取决于电脑速度和程序质量）
- 测量每一个元素执行的次数（不好，太难了，没必要）
- 测量基础操作（Basic operation）的执行次数

基础操作：算法中最耗时的操作

时间复杂度的理论分析

通过确定基本操作的重复次数作为输入规模的函数来分析时间复杂度。

$$T(n) = t_{op}C(n)$$

其中T 为运行时间，n为输入规模，t_{op}为基本操作执行时间，C为基本操作的执行次数

增长规模

$$1 < log(n) < n < nlog(n) < n^2 < n^3 < 2^n < n!$$

最好、最坏、平均时间复杂度

- 最好：$C(n)$最小

- 最坏：$C(n)$最大
- 平均：不能直接取最好和最坏的平均，使用下面的公式：

$$T_{avg}(n) = \sum_{I \in D_n} P(I)T(N, I)$$

算法分析框架

时间效率$T(n)$是通过计算算法中执行的基本操作的数量来衡量的。空间效率$S(n)$由消耗的额外内存单元的数量来衡量。时间和空间效率都以输入大小的函数来度量。该框架的主要兴趣在于当其输入大小为无穷大时，算法的运行时间(空间)的增长顺序。对于相同大小的输入，某些算法的效率可能会有显著差异。对于这些算法，我们需要区分最坏情况、最好情况和平均情况的效率。

渐进时间复杂度

$O(x)$ 表示装入的x是大的一方，x>=y === y∈O(x)

$\Omega(x)$ 表示装入的x是小的一方，x>=y === x∈ Ω(y)

$\Theta(x)$ 表示相等

$$1 \in O(n)$$
$$n \in O(n^2)$$
$$n^3 \in \Omega(n^2)$$
$$2n^2 + 3n \in \Theta(n^2)$$

算法的比较

相除求极限，或洛必达法则求，0小于，c等于，无穷大于

非递归算法的时间复杂度分析

1. 确定输入规模n
2. 确定基础操作
3. 确定是否需要分最好/坏/平均情况讨论
4. 取得$C(n)$的值
5. 求值

递归算法的时间复杂度分析

1. 确定输入规模n
2. 确定基础操作
3. 确定是否需要分最好/坏/平均情况讨论
4. 为$C(n)$ 建立递归关系和初始条件
5. 求值

主定理

$T(n)$ 是个最终非递减函数

$T(n) = aT(n/b) + f(n)$,其中$n = b^k, k = 1, 2, \dots$

$T(1) = c$

其中 $a \geq 1, b \geq 2, c > 0$ 如果$f(n) \in \Theta(n^d)$，$d \geq 0$，那么

$$T(n) \in \Theta(n^d) \quad \text{当} a < b^d \text{时}$$
$$T(n) \in \Theta(n^d log(n)) \quad \text{当} a = b^d \text{时}$$
$$T(n) \in \Theta(n^{log_b(a)}) \quad \text{当} a > b^d \text{时}$$

第三章 蛮力法

选择排序

依次选择最小的放在队首。时间复杂n^2，交换次数n

冒泡排序

依次交换顺序。时间复杂n^2

字符串匹配

最坏时间复杂度nm， 平均时间复杂度$n+m => n$

两点间距离

时间复杂度n^2

多项式求值

旅行家算法（TSP）npHard

除起点外全排列，复杂度$(n-1)!$

背包问题 npHard

复杂度2^n

时间安排问题

不同工作交给不同人，求最佳安排，复杂度$n!$

np 困难问题

介绍**NP困难**之前要说到P问题和NP问题，**P问题**就是在多项式时间内可以被解决的问题，**NP问题**就是在多项式时间内可以被解决并验证其正确性的问题。 **NP困难**（**NP-hardness**, non-deterministic polynomial-time hardness）问题是计算复杂性理论中最重要的复杂性类之一。如果所有NP问题都可以多项式时间归约到某个问题，则称该问题为NP困难。

因为NP困难问题未必可以在多项式的时间内验证一个解的正确性（即不一定是NP问题），因此即使NP完全问题有多项式时间的解（P=NP），NP困难问题依然可能没有多项式时间的解。因此NP困难问题"至少与NP完全问题一样难"。

第四章 递归算法

迭代与递归

迭代是个循环算式，递归请务必标明递归的初始条件以及递归关系

$C(n)$的计算

- 减一法：$T(n) = T(n-1) + f(n)$
- 主定理：$T(n) = aT(n/b) + f(n), T(1) = c, f(n) \in n^d$

第五章 分治法

分治法的思想

将要求解的大规模问题分解成k个更小规模的子问题，对这k个问题分别求解，如果子问题规模仍不够小，再划分为k个子问题，如此递归的进行下去，直到问题规模足够小，容易求解为止。

子问题求解后，将小规模问题的解合并成为一个更大规模问题的解，自底向上逐步求出原来问题的解。

分治法三步骤

- 分解
- 求解
- 合并

平衡(balancing)子问题的思想

在用分治法设计算法时，最好使子问题的规模大 致相同。即将一个问题分成大小相等的 k 个子问题的处理方法是行之有效的。

大数相乘

蛮力法：时间复杂度n^2

分治法：

- 将两数都拆解为两部分，四个部分分别相乘再乘对应系数：$T(n) = 4T(n/2) + O(n)$
- 四个部分按照$(A+B)*(C+D)-AC-BD$求值：$T(n) = 3T(n/2) + O(n)$

由主定理得时间复杂度分别为$O(n^2), O(n^{log_2 3})$

第二种方式成功降低了时间复杂度

矩阵乘法

蛮力：时间复杂度$O(n^3)$

分治：

- 正常分为四块，设置8个值
- 按照规律，设置7个值

由主定理得时间复杂度分别为$O(n^3), O(n^{log_2 7})$

第二种方式成功降低了时间复杂度

二分搜索

最好：$O(1)$

最坏：$O(logn)$ => $logn+1$

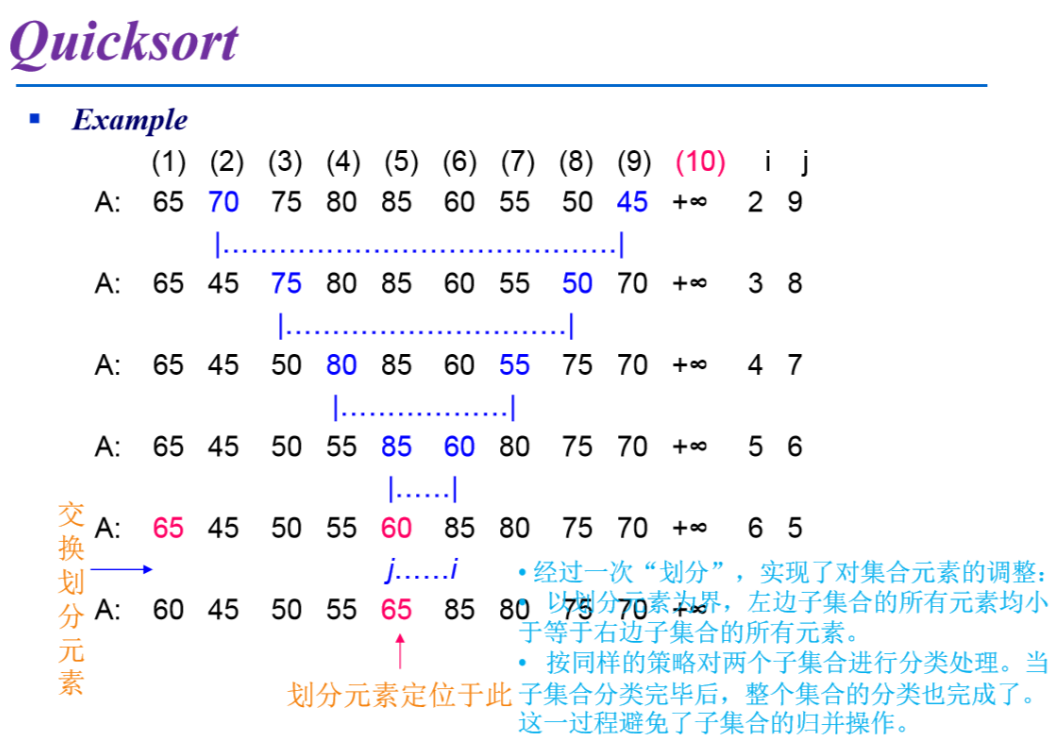
平均：$O(log(n+1)-1)$

逐个查找：$p(n+1)/2 + n(p-1)$

归并排序

复杂度：$O(nlogn)$

快速排序



最好：$C(b)=2C(b/2)+\Theta(n)$ 复杂度:$O(nlogn)$

最坏：$C(b)=C(b-1)+\Theta(n)$ 复杂度：$O(n^2)$

平均: $C(n)=(1/n)\sum_{s=0}^{n-1}[(n+1)+(C(s)+C(n-1-s))]$

复杂度：$O(nlogn)$

改进：

随机选取划分元素，使划分比较对称

棋盘覆盖

不断覆盖，每次将棋盘划分成四个小棋盘。

设n=2^k $T(k)=4T(k-1)+1$

复杂度：$T(k)=O(4^k)$　$O(n^2)$

第六章 减治法

减治3种形式

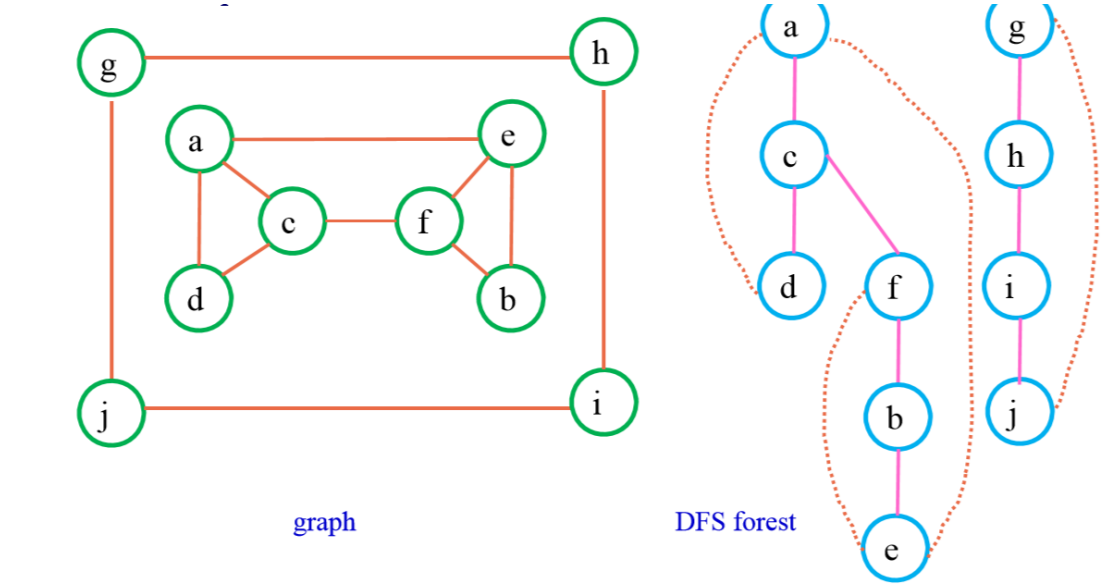
- 减一个常量
- 减一个常量因子
- 减去的规模是可变的

插入排序

- 减常量1

- 最坏：$\Theta(n^2)$
- 最好：$\Theta(n)$
- 平均：$\Theta(n^2/4) = \Theta(n^2)$

深度优先遍历

- 使用栈存储访问过的结点，弹出时说明它已成为死胡同
- 

回溯法

- 回溯法是一个既带有系统性又带有跳跃性的搜索算法；
- 它在包含问题的所有解的解空间树中，按照深度优先的策略，从根结点出发搜索解空间树。——**系统性**
- 算法搜索至解空间树的任一结点时，判断该结点为根的子树是否包含问题的解，如果肯定不包含，则跳过以该结点为根的子树的搜索，逐层向其祖先结点回溯。否则，进入该子树，继续深度优先的策略进行搜索。——**跳跃性**
- 这种以深度优先的方式系统地搜索问题的解得算法称为回溯法，它适用于解一些组合数较大的问题

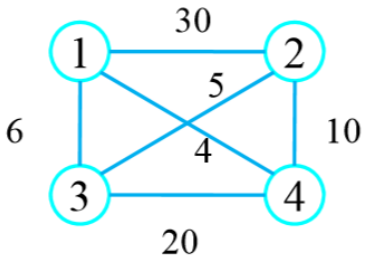
概念：解向量，解空间，约束条件（显式，隐式），活结点，死结点，当前扩展结点。

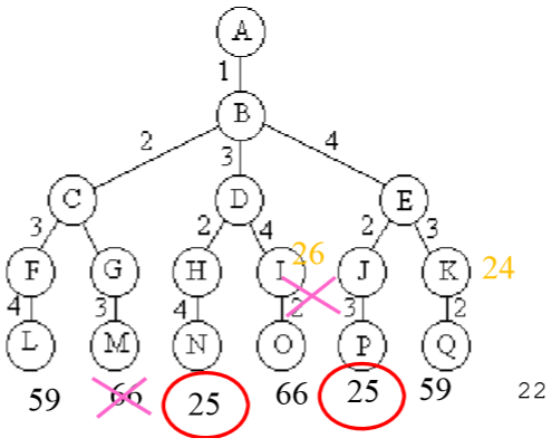
解题步骤：

- 定义解空间
- 确定易于搜索的解空间结构
- 深优搜索解空间，过程中用剪枝函数避免无效搜索

常见解空间树：

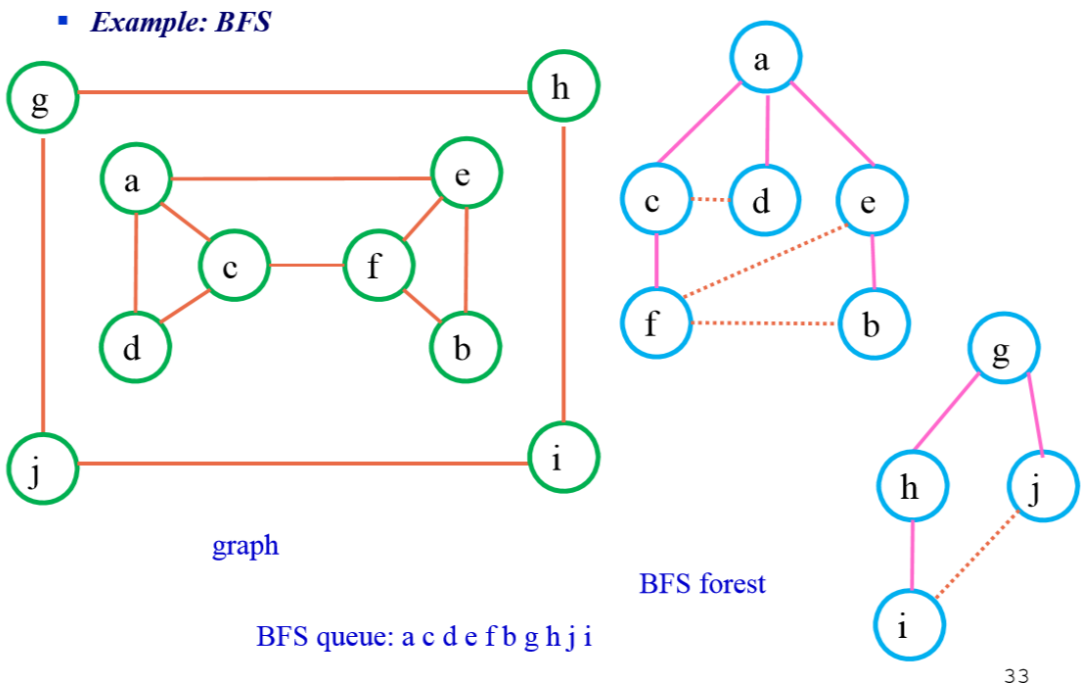
- 子集树（背包问题$O(2^n)$）
- 排列树（TSP问题$O(n!)$）





广度优先遍历

- 使用先进先出队列存储访问的结点，访问时，对应结点出队列，并且把它的相邻结点加入到队列中

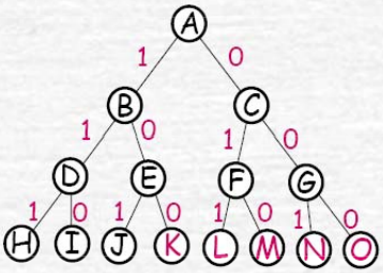


分支界限法

- 分支限界法常以广度优先或以最小耗费（最大效益）优先的方式搜索问题的解空间树，裁剪那些不能得到最优解的子树以提高搜索效率。
- 搜索策略是在扩展结点处先生成其所有儿子结点（分支），然后再从当前的活结点表中选择下一个扩展结 点。为了有效地选择下一个扩展结点，以加速搜索的进程， 在每一活结点处，计算一个函数值（优先值），并根据这些已计算出的函数值，从当前活结点表中选择一个最有利的结点作为扩展结点，使搜索朝着解空间树上有最优解的分支推进，以便尽快地找出一个最优解。
- **与回溯法区别**：
  - 求解目标不同（回溯找所有解，分支尽快找出一个解）
  - 搜索方法不同（深优vs广优（优先级搜索））
  - 对扩展结点的扩展方式不同（分支法中，一个节点只有一个成为扩展结点的机会，一次性给出所有儿子）
  - 存储空间要求不同（分支界限法要求的空间比回溯法大得多）
- 两种常见方法：
  - 队列式（FIFO）
  - 优先队列（代价最小或效益最大）分支界限法：可以采用**堆**

- ③ BFS搜索（FIFO队列）

扩展结点	活结点	队列(可行结点)	可行解(叶结点)	解值
A	B,C	BC		
B	D,E(D死结点)	CE		
C	F,G	EFG		
E	J,K(J死结点)	FG	K	40
F	L,M	G	L,M	50,25
G	N,O	ϕ	N,O	25,0



∴ 最优解为L，即(0,1,1)；解值为50

拓扑排序

按照深度优先算法出栈顺序反向或者按照无入度结点顺序排序

第七章 变治法

把问题变换为更简单，易求解的问题。

分 减 变的区别：

分治：解多个子问题，合并为原问题的解

减治：解一个子问题，扩展成为原问题的解

变治：把问题实例变得更容易求解

变治法三个方式：

- 实例化简（预排序法，高斯消去法）
- 改变表现（AVL树，多路查找树）
- 问题化简（lcm）

预排序

- ***Selection Sort*** $\Theta(n^2)$
- ***Bubble Sort*** $\Theta(n^2)$
- ***Insertion Sort*** $C_{worst}(n)=\frac{(n-1)n}{2}$ $C_{best}(n)=n-1$ $C_{avg}(n)\approx\frac{n^2}{4}$
- ***Mergesort*** $\Theta(n \log n)$
- ***Quicksort*** $C_w(n)=\Theta(n^2)$ $C_b(n)=\Theta(n \log n)$ $C_{avg}(n)=O(n \log n)$

在搜索，计算中间值，检查元素唯一性前可以对元素进行预排序，可以将时间复杂度由$O(n^2)$降低至$O(nlogn)$

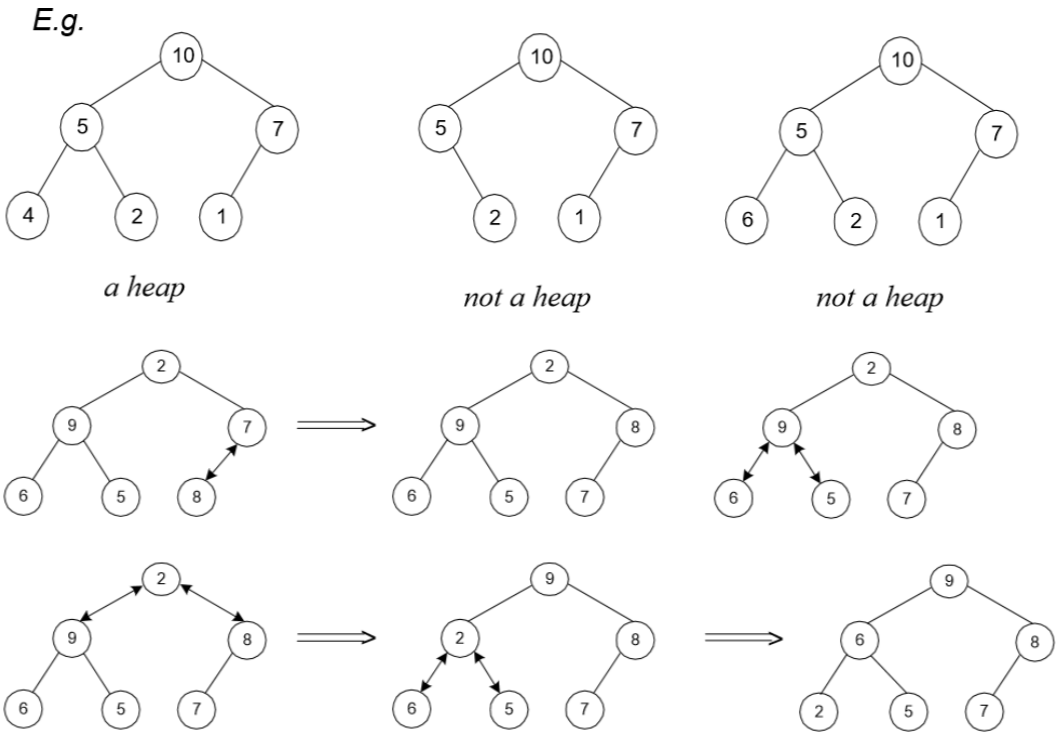
高斯消去法

线性代数知识

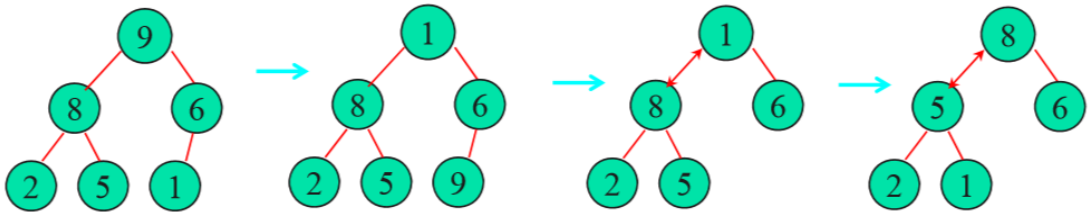
初等变换：时间复杂度$O(n^3)$

反向替代：时间复杂度$O(n^2)$

堆和堆排序



提示：最大堆和大的比，最小堆和小的比



根删除法

霍纳法则

多项式乘法更改为嵌套相乘

$$E.g.: p(x) = 2x^4 - x^3 + 3x^2 + x - 5 = x(2x^3 - x^2 + 3x + 1) - 5 =$$
$$= x(x(2x^2 - x + 3) + 1) - 5 = x(x(x(2x - 1) + 3) + 1) - 5$$

To evaluate p(x) at x=3

coefficients	2	-1	3	1	-5
x=3	2	3*2+(-1) = 5	3*5+3=18	3*18+1=55	3*55+(-5)=160

乘法n次加法n次

最小公倍数（lcm）

使用了问题化简法

$$lcm(m,n) = m * n / gcd(m,n)$$

线性规划

问题化简

建立数学模型解决问题，找界限点， 比如背包问题，一次函数问题

画图化简

比如狼羊草问题

第八章 动态规划

与分治算法区别和联系

分治法是指将问题划分成一些独立地子问题，递归地求解各子问题，然后合并子问题的解而得到原问题的解。与此不同，动态规划适用于子问题独立且重叠的情况，也就是各子问题包含公共的子子问题。在这种情况下，若用分治法则会做许多不必要的工作，即重复地求解公共的子问题。动态规划算法对每个子子问题只求解一次，将其结果保存在一张表中，从而避免每次遇到各个子问题时重新计算答案。

动态规划特征：

适合采用动态规划方法的最优化问题中的两个要素：最优子结构和重叠子问题。

- 最优子结构：如果问题的一个最优解中包含了子问题的最优解，则该问题具有最优子结构。
- 重叠子问题：适用于动态规划求解的最优化问题必须具有的第二个要素是子问题的空间要很小，也就是用来求解原问题的递归算法课反复地解同样的子问题，而不是总在产生新的子问题。对两个子问题来说，如果它们确实是相同的子问题，只是作为不同问题的子问题出现的话，则它们是重叠的。

算法设计步骤

- 描述最优解的结构
- 递归定义最优解的值
- 按自底向上的方式计算最优解的值
- 由计算出的结果构造一个最优解

计算二项式系数

$$C(n,0) = C(n,n) = 1$$
$$C(n,k) = C(n-1,k-1) + C(n-1,k)$$

	0	1	2	3	4	5	……	k-1	k
0	1								
1	1	1							
2	1	2	1						
3	1	3	3	1					
4	1	4	6	4	1				
5	1	5	10	10	5	1			
k	1								1
…									
n-1	1							C(n-1,k-1)	C(n-1,k)
n	1								C(n,k)

$$A(n,k) = k(k-1)/2 + n(n-k) \in \Theta(nk)$$

动态矩阵乘法

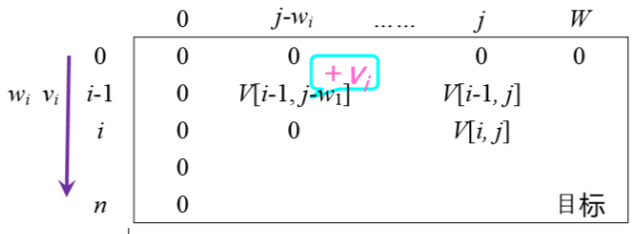
矩阵乘法代价为$A_{pq} * A_{qr}$对应代价为pqr

所以给矩阵连乘中间加入括号，使得代价变小

$$m[i,j] = \begin{cases} 0 & i = j \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\} & i < j \end{cases}$$

0-1背包问题

Ex. recurrence from the first item

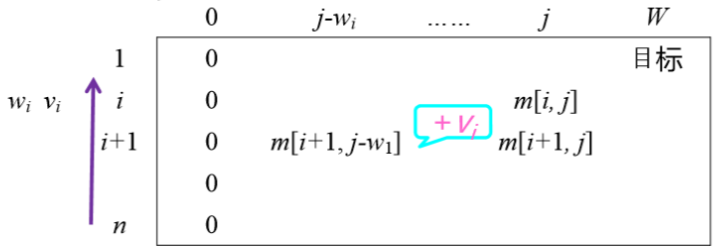


item	weight	value
1	2	12¥
2	1	10¥
3	3	20¥
4	2	15¥

	i	0	1	2	3	4	5			
	0	0	0	0	0	0	0	$V(i-1,j-w_1)+v_1$		$V(i-1,j)$
w_1=2. v_1=12	1	0	0	12	12	12	12	$V(i-1,j-w_2)+v_2$	$V(i-1,j)$	$V(i,j)$
w_2=1. v_2=10	2	0	10	12	22	22	22		$V(i,j)$	
w_3=3 v_3=20	3	0	10	12	22	30	32			
w_4=2. v_4=15	4	0	10	15	25	30	37			

Composition of an optimal solution, through tracing back the computations of the last entry V[4,5]
V[4,5]≠*V*[3,5], *item* 4 *is included in an optimal solution, with an optimal subset for V*[3,3];
V[3,3] =*V*[2,3], *item* 3 *not included in an optimal subset,*
V[2,3] ≠*V*[1,3], *item* 2 *is included in an optimal subset*
V[1,2] ≠*V*[0,2], *item* 1 *is included in an optimal subset . So, optimal solution is* {1,1,0,1}, *i.e. item{1,2,4}*

Ex. recurrence from the last item



item	weight	value
1	2	12¥
2	1	10¥
3	3	20¥
4	2	15¥

	i	0	1	2	3	4	5			
w_1=2. v_1=12	1	0	10	15	25	30	37			
w_2=1. v_2=10	2	0	10	15	25	30	35		$m(i, j)$	
w_3=3 v_3=20	3	0	0	15	20	20	35	$m(i+1, j-w_2)+v_2$	$m(i+1, j)$	$m(i, j)$
w_4=2. v_4=15	4	0	0	15	15	15	15	$m(i+1, j-w_3)+v_3$		$m(i+1, j)$

m[1,5]≠m[2,5], item 1 is included in an optimal solution, with an optimal subset for m[2,3]

m[2,3] ≠ m[3,3], item 2 is included in an optimal subset ,

m[3,2] = m[4,2], item 3 not included in an optimal subset ,

m[4,2] ≠ 0, item 4 is included in an optimal subset . So, optimal solution is {1,1,0,1}*, i.e. item{1,2,4}*

多阶段决策问题

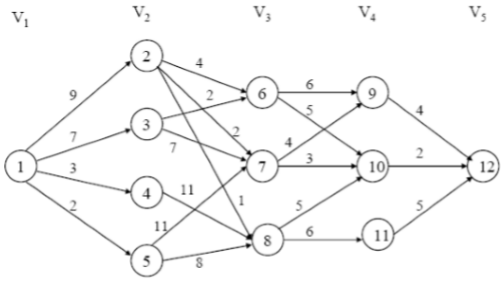
- 最优决策序列
- 最优化原理：
  - 过程的最优决策序列具有如下性质：无论过程的初始状态和初始决策是什么，其余的决策都必须相对于初始决策所产生的状态构成一个最优决策序列。
  - 利用动态规划求解问题的前提
    - 证明问题满足最优性原理

      如果对所求解问题证明满足最优性原理，则说明用动态规划方法有可能解决该问题
    - 获得问题状态的递推关系式

      获得各阶段间的递推关系式是解决问题的关键。

多段图问题

Multistage Decision Pr



- 向前递推结果

第4段 COST(4,9) = c(9,12) = 4
COST(4,10) = c(10,12) = 2
COST(4,11) = c(11,12) = 5

第3段 COST(3,6) = min{6+COST(4,9),5+COST(4,10)} = 7
COST(3,7) = min{4+COST(4,9),3+COST(4,10)} = 5
COST(3,8) = min{5+COST(4,10),6+COST(4,11)} = 7

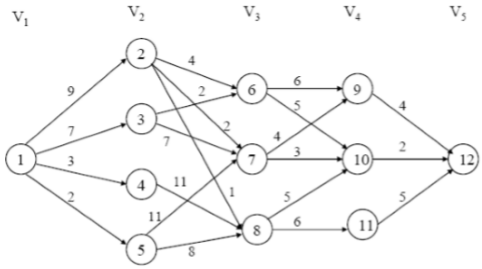
第2段 COST(2,2) = min{4+COST(3,6) , 2+COST(3,7), 1+COST(3,8)} = 7
COST(2,3) = 9
COST(2,4) = 18
COST(2,5) = 15

第1段 COST(1,1) = min{9+COST(2,2),7+COST(2,3),
3+COST(2,4),2+COST(2,5)} = 16

S到t的最小成本路径的成本 = 16

复杂度$O(n+e)$

Multistage Decision Pr



- 向后递推结果

第2段 BCOST(2,2) = 9
BCOST(2,3) = 7
BCOST(2,4) = 3
BCOST(2,5) = 2

第3段 BCOST(3,6) = min{BCOST(2,2)+4,BCOST(2,3)+2} = 9
BCOST(3,7) = min{BCOST(2,2)+2,BCOST(2,3)+7,BCOST(2,5)+11} = 11
BCOST(3,8) = min{BCOST(2,4)+11, BCOST(2,5)+8} = 10

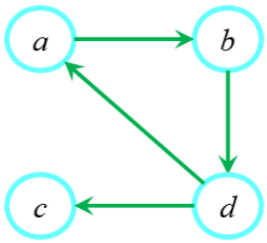
第4段 BCOST(4,9) = min{BCOST(3,6)+6,BCOST(3,7)+4} = 15
BCOST(4,10) = min{BCOST(3,6)+5, BCOST(3,7)+3,BCOST(3,8)+5}= 14
BCOST(4,11) = min{BCOST(3,8)+6} = 16

第5段 BCOST(5,12) = min{BCOST(4,9)+4, BCOST(4,10)+2, BCOST(4,11)+5}
= 16

S到t的最小成本路径的成本 = 16

Warshall's算法

- ***Example***

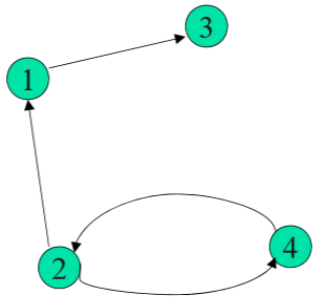


adjacent matrix

$$A=\begin{array}{c} \\ a \\ b \\ c \\ d \end{array}\begin{array}{c}\begin{array}{cccc} a & b & c & d \end{array}\\ \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix}\end{array}$$

Transitive Closure

$$T=\begin{array}{c} \\ a \\ b \\ c \\ d \end{array}\begin{array}{c}\begin{array}{cccc} a & b & c & d \end{array}\\ \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}\end{array}$$



adjacent matrix

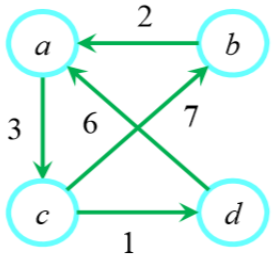
$$A=\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\begin{array}{c}\begin{array}{cccc} 1 & 2 & 3 & 4 \end{array}\\ \begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}\end{array}$$

Transitive Closure

$$T=\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\begin{array}{c}\begin{array}{cccc} 1 & 2 & 3 & 4 \end{array}\\ \begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}\end{array}$$

Floyd's算法

- ***Example***

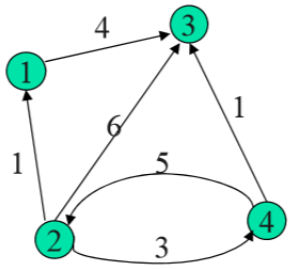


weight matrix

$$W=\begin{array}{c} \\ a \\ b \\ c \\ d \end{array}\begin{array}{c}\begin{array}{cccc} a & b & c & d \end{array}\\ \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \infty & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \infty & 0 \end{bmatrix}\end{array}$$

Distance Matrix

$$D=\begin{array}{c} \\ a \\ b \\ c \\ d \end{array}\begin{array}{c}\begin{array}{cccc} a & b & c & d \end{array}\\ \begin{bmatrix} 0 & 10 & 3 & 4 \\ 2 & 0 & 5 & 6 \\ 7 & 7 & 0 & 1 \\ 6 & 16 & 9 & 0 \end{bmatrix}\end{array}$$



weight matrix

$$W=\begin{array}{c} \\ a \\ b \\ c \\ d \end{array}\begin{array}{c}\begin{array}{cccc} a & b & c & d \end{array}\\ \begin{bmatrix} 0 & \infty & 4 & \infty \\ 1 & 0 & 6 & 3 \\ \infty & \infty & 0 & \infty \\ \infty & 5 & 1 & 0 \end{bmatrix}\end{array}$$

Distance Matrix

$$D=\begin{array}{c} \\ a \\ b \\ c \\ d \end{array}\begin{array}{c}\begin{array}{cccc} a & b & c & d \end{array}\\ \begin{bmatrix} 0 & \infty & 4 & \infty \\ 1 & 0 & \mathbf{4} & 3 \\ \infty & \infty & 0 & \infty \\ \mathbf{6} & 5 & 1 & 0 \end{bmatrix}\end{array}$$

第九章 贪心算法

贪心算法概念

是一种在每一步选择中都采取在当前状态下最好或最优（即最有利）的选择，从而希望导致结果是最好或最优的算法。也就是说，不从整体最优上加以考虑，他所做出的仅是在某种意义上的局部最优解。

贪心选择性质

贪心选择是指所求问题的整体最优解可以通过一系列局部最优的选择，即贪心选择来达到。这是贪心算法可行的第一个基本要素，也是贪心算法与动态规划算法的主要区别。贪心选择是采用从顶向下、以迭代的方法做出相继选择，每做一次贪心选择就将所求问题简化为一个规模更小的子问题。

贪心算法子结构特点

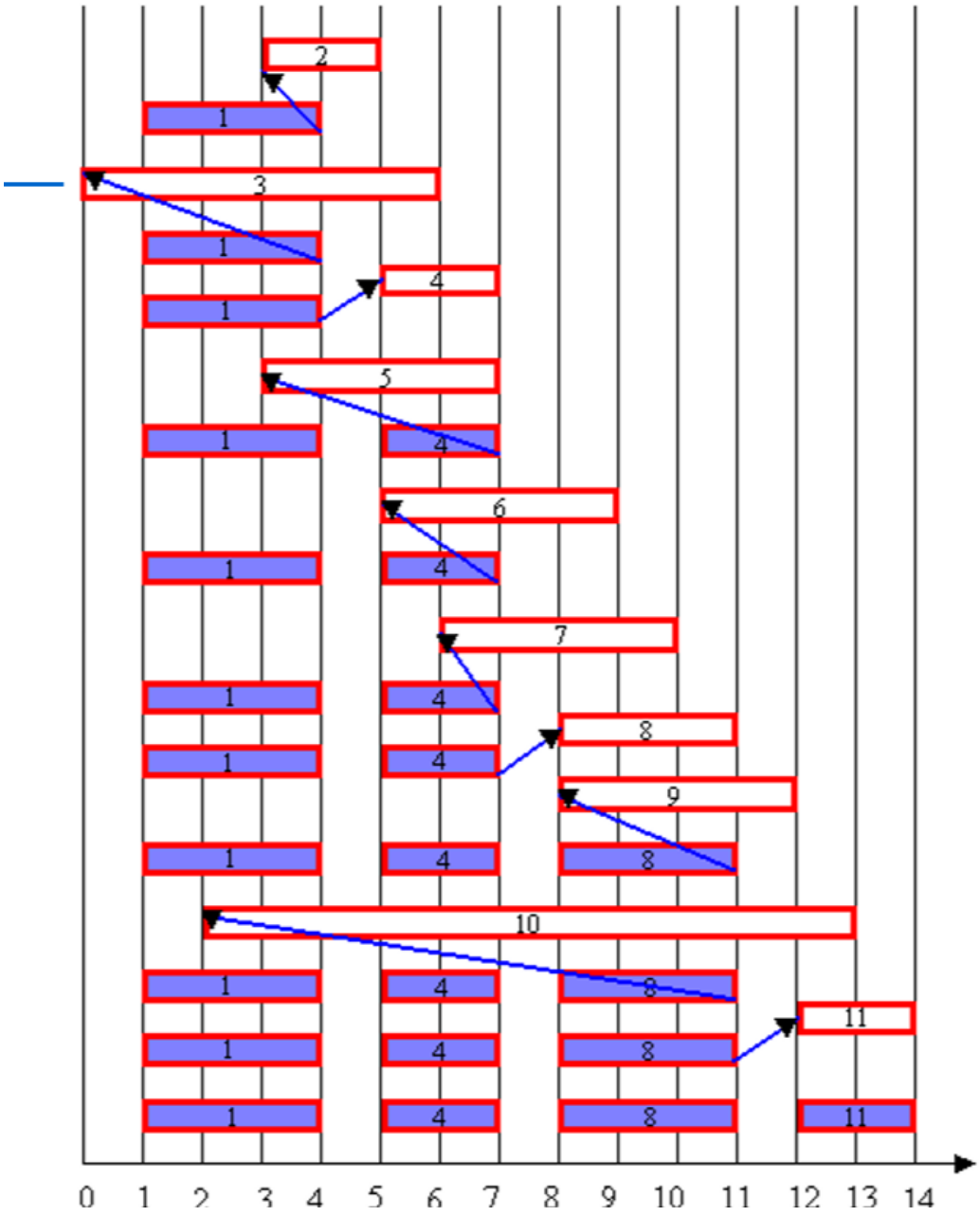
- 可行
- 局部最优
- 不可撤销

与动态规划的区别与联系

- 贪心算法和动态规划算法的共同点：要求问题具有最优子结构性质
- 动态规划算法：每步所做的选择往往依赖于子问题的解，只有在解出相 关子问题后才能作出选择
- 贪心算法：仅在当前状态下作出最好选择，即局部最优选择，然后再去作出这个选择后产生的相应的子问题，不依赖于子问题的解，
- 动态规划算法：通常以自底向上的方式解各子问题，
- 贪心算法：通常以自顶向下的方式进行，以迭代的方式作出相继的贪心选择，每作一次贪心选择就将所求问题简化为规模更小的子问题。

找零钱问题

活动安排问题



贪心策略求解背包问题

度量标准的选择：

- 以目标函数作为度量

利益非增次序一件一件放入背包，放不进去时取一部分

- 以容量作为度量

物品重量升序装入

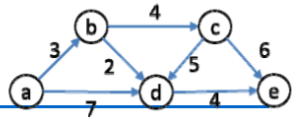
- 以单位利益值作为度量（V）

按单位利益值降序考虑，得最优解

效率$O(nlogn)$

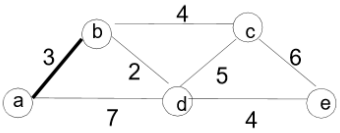
Dijekstra算法

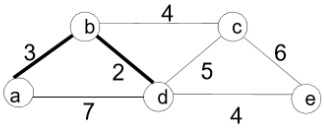
Single-source Shortest Paths

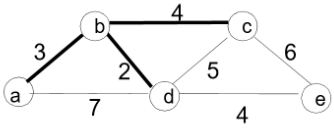


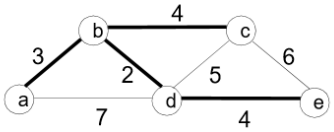
- ***Example : Dijkstra's***

Tree vertices	Remaining vertices
a(-,0)	b(a,3) c(-,∞) d(a,7) e(-,∞)
b(a,3)	c(b,3+4) d(b,3+2) e(-,∞)
d(b,5)	c(b,7) e(d,5+4)
c(b,7)	e(d,9)
e(d,9)	









The shortest paths (identified by following nonnumeric labels backward from a destination vertex in the left column to the source) and their lengths (given by numeric labels of the tree vertices) are as follows:
from a to b : a − b of length 3
from a to d : a − b − d of length 5
from a to c : a − b − c of length 7
from a to e : a − b − d − e of length 9

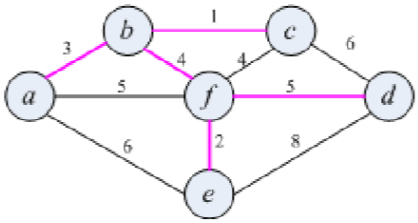
最小生成树

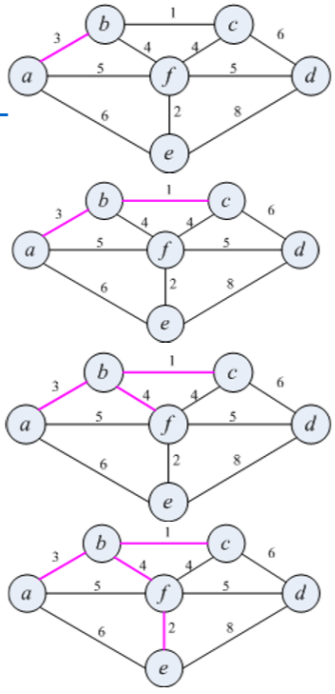
Prim算法

Minimum Spanning Tree

- *Example : Prim*

Tree vertices	Remaining vertices
a(-,0)	b(a,3) c(-,∞) d(-, ∞) e(a,6) f(a,5)
b(a,3)	c(b,1) d(-, ∞) e(a,6) f(b,4)
c(b,1)	d(c, 6) e(a,6) f(b,4)
f(b,4)	d(f, 5) e(f,2)
e(f,2)	d(f, 5)
d(f, 5)	





Kruskal算法

Minimum Spanning Tree

Example: Kruskal's

Tree edges	Sorted list of edges									
	bc	ef	ab	bf	cf	af	df	ae	cd	de
	1	2	3	4	4	5	5	6	6	8
bc										
1										
ef										
2										
ab										
3										
bf										
4										
df										
5