算法分析基础

（一）Algorithm evaluation 算法评价

（A）图灵机:

使用 steps 作为时间

使用 tapes 作为空间

缺点:

1. 没有随机访问内存
2. 比普通计算机要更多步骤

（B）Random-Access-Machine 随机存取机

运行时间：基本操作的数量。

内存：所使用的内存单元数

（二）算法的正确性

我们需要规定一个Specification

名字,参数, inputs， output

（A）完全正确性：对于任何符合要求的输入

1. 可终止
2. 输出正确

（B）部分正确性：对于任何符合要求的输入

如果终止了则输出正确

(也就是 完全正确 或 不可终止)

（C）证明完全正确性

1. 使用循环不变式 证明 部分正确性
   - 初始时
   - Maintenance if it is true before, then true after loop.
   - 终止时 这个不变式可以证明算法的正确

性

2. 使用循环变体 证明 终止

良序集上 找到一些量在严格减小!

(三) 算法的效率

(A) 复杂度

时空，其他...

最坏 最好 平均

(B) Big O notation

(C) Big Ω notation

(D) Big Θ notation

(E) Small o and ω notation '<'

(F) 使用定义极限(应对复数)

$$n! \sim \sqrt{2\pi n} \cdot \left(\frac{n}{e}\right)^n$$

- When considering brute force algorithm to solve one problem, it is usually asymptotically equal to exponential functions.
- When an algorithm has a polynomial running time, we say it is **efficient,** and the corresponding problem is so-called **easy** or **tractable.**
  - The algorithm has typically exposes some **crucial structure** of the problem.

基本数据结构

（零）

A data structure is a way to store and organize data in order to facilitate access and modifications.

Abstract Data Type：抽象数据结构

An ADT is a logical description, and a data structure is a concrete implementation.

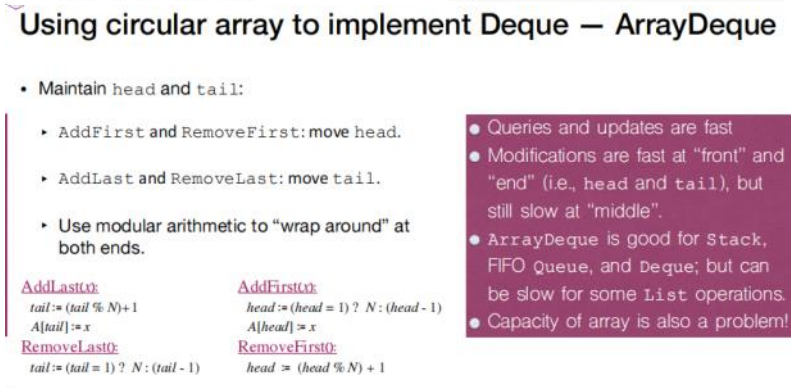
（A）Queue ADT：add/remove + 排队原则

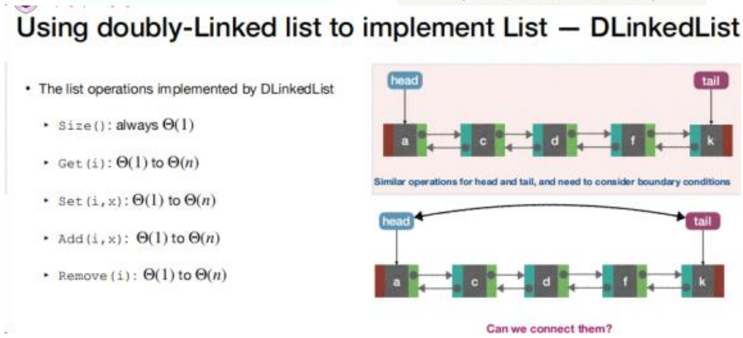
FIFO (first in first out)

LIFO （least in first out)

（B）Dequeue ADT

（C）List ADT





队列的应用：buffer

栈的应用：匹配括号、函数调用

尾递归：

如果一个函数的每次激活都最多进行一次递归调用，并且在该调用之后立即返回，则该函数被称为尾部递归。尾递归可以化成循环。

分治

二分策略

归并排序

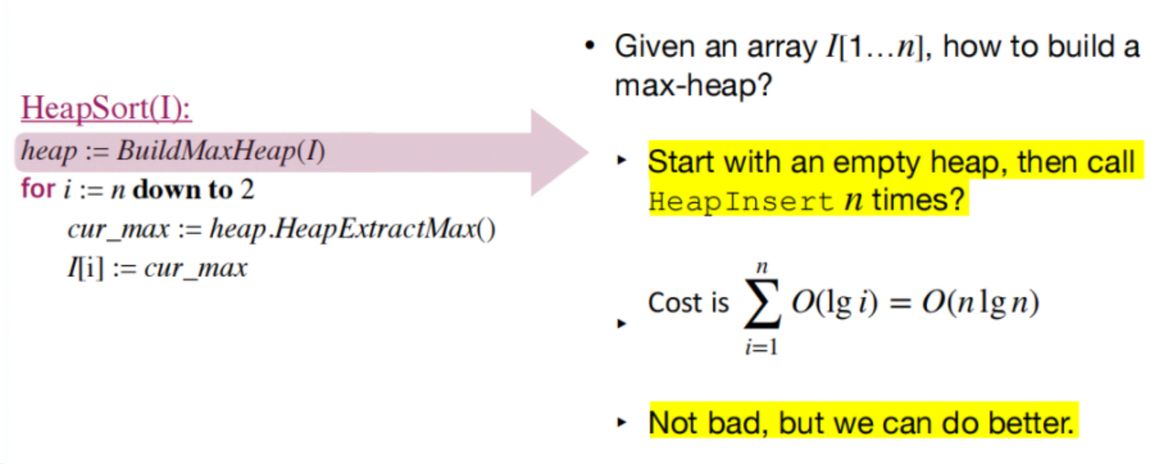
堆

2023年12月27日　　20:25

堆的实现 应该比较简单

然后关于堆的复杂度也比较清晰

主要是HeapSort看一下

1. 

2. 全部填进去，然后heapify，button to up using merge method （O(n)）

排序

2024年1月6日 19:10

Insertion

- $n(n-1)/2$ **swaps,** and $n \cdot (n-1)/2$ **comparisons** -> worst
- $n(n-1)/4$ **swaps,** and $n \cdot (n-1)/4$ **comparisons** -> on average

Selection

- $n-1$ **swaps,** and $n \cdot (n-1)/2$ **comparisons**

Bubble

- $n \cdot (n-1)/2$ **swaps,** and $n \cdot (n-1)/2$ **comparisons**

关于插入排序:

size小、基本顺序 -> 表现好

移动多 -> 表现差

希尔排序:

```
void shell_sort(int arr[], int len) {
    int gap, i, j;
    int temp;
    for (gap = len >> 1; gap > 0; gap >>= 1) {
        for (i = gap; i < len; i++) {
            temp = arr[i];
            for (j = i - gap; j >= 0 && arr[j] > temp; j -= gap)
                arr[j + gap] = arr[j];
            arr[j + gap] = temp;
        }
    }
}
```

均摊分析

2024年1月6日 21:38

- Consider a sequence operations: c_i = actual cost of the i^{th} operation; $\hat{c}_i$ = amortized cost of the i^{th} operation. Then, the amortized cost to be valid: $\sum_{i=1}^{k} c_i \le \sum_{i=1}^{k} \hat{c}_i$ for any $k \in \mathbb{N}^+$.

Now let us consider the amortized cost in a higher level than the specific value in one operation (accounting)!

- Design a **potential function** Φ that maps data structure status to real values
  - $\Phi(D_0)$: initial potential of the data structure, usually set to 0.
  - $\Phi(D_i)$: potential of the data structure after i^{th} operation.
- Define $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$
- For amortized cost to be valid, need $\Phi(D_k) \ge \Phi(D_0)$ for all k.

Back to CircularArray based Queue

- Now suppose we need to shrink array for space consideration
  - **Solution(1):** Reduce array size to half when array only half loaded after Remove. (Allocate new array of half size, copy items to new array, and delete old array.)
  - **Solution(2):** Reduce array size to half when array only $1/4$ loaded after Remove. (Allocate new array of half size, copy items to new array, and delete old array.)
- Quiz: which one is better with respect to amortized cost?

上面一个 震荡时O(n)

下面一个 O(1)

remove = 7 = 4/n * (create n/2 + copy n/4 +delete n),
insert = 3

他人整理

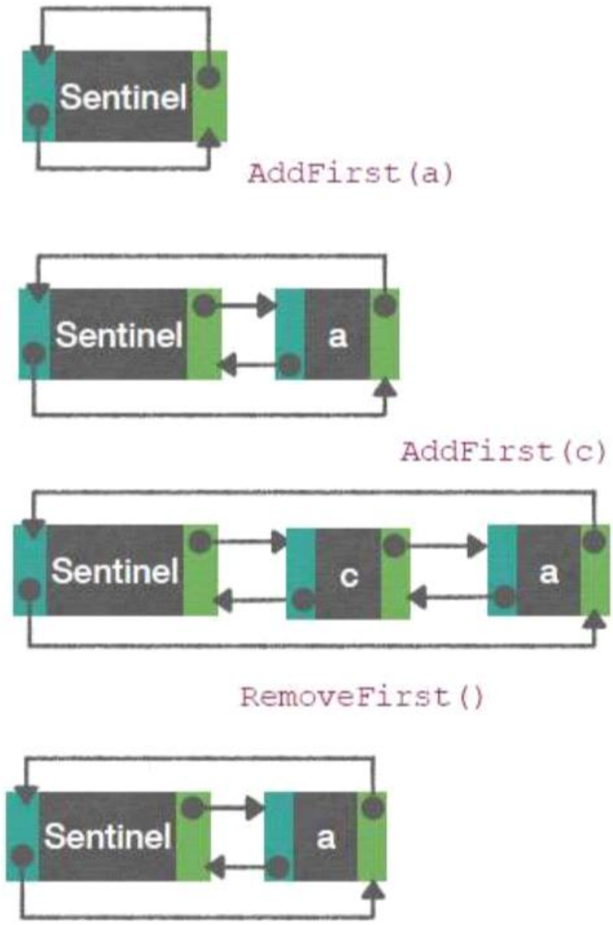
Cheat Sheet

分析基础

1. 插入排序
    1. 类比整理扑克牌
    2. 不变量 (Invariant) 是 $A[1..\,.j-1]$ 有序
    3. 稍微修改就成为冒泡算法
2. 时间复杂度
    1. $f(n) \in O(g(n))$ 表示存在某些常量 c 使得在 n_0 之后 $f(n) \leqslant cg(n)$
    2. 说人话, 就是一个可以达到的上界
    3. $f(n) \in \Omega(g(n))$ 表示存在某些常量 c 使得在 n_0 之后 $f(n) \geqslant cg(n)$
    4. 说人话, 就是一个可以达到的下界
    5. 既是 O 也是 Ω 的话, 就成为 Θ 了, 即等阶
    6. $f(n) \in o(g(n))$ 表示对任何常量 c, 在 n_0 之后 $f(n) < cg(n)$
    7. 说人话, 就是一个达不到的上界, 和 Ω 恰好相反
    8. $f(n) \in \omega(g(n))$ 表示对任何常量 c, 在 n_0 之后 $f(n) > cg(n)$
    9. 说人话, 就是一个达不到的下界, 和 O 恰好相反
3. 常用工具
    1. 洛必达法则
    2. 斯特林近似: $n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$, $\sqrt{c_0 n}\left(\frac{n}{e}\right)^n \leqslant n! \leqslant \sqrt{c_1 n}\left(\frac{n}{e}\right)^n$

结构基础

1. 抽象数据类型 (Abstract Data Type)
2. 队列
    1. FIFO 队列
        1. `Enqueue(x)`
        2. `Dequeue()`
    2. LIFO 队列, 栈 (Stack)
        1. `Add(x)` 或 `Push(x)`
        2. `Remove()` 或 `Pop()`
    3. 双端队列 (Deque)
    4. 列表 (List)
        1. `Size()`
        2. `Get(i)`
        3. `Set(i, x)`
        4. `Add(i, x)`
        5. `Remove(i)`
        6. 我们可以用循环数组来实现列表, 进而实现双端队列
        7. 如果用双向链表实现的话, 记得加一个哨兵 (Sentinel)



3. 应用
    1. 栈可以用来平衡符号 (Balancing Symbols)
    2. 栈可以用来计算后序表达式, 不需要括号和优先级, 就已经有能力表示和计算任何的二元表达式
        1. 设定两个栈, 操作数栈和操作符栈, 便可以计算中序表达式
    3. 两个 FIFO 可以实现一个 LIFO, 反之亦然
    4. 栈可以实现函数调用, 同理, 任何的递归也可以改用循环和栈来实现
    5. 尾递归可以直接改成循环, 不需要栈, 只需要在 `while(True)` 循环尾部将函数参数改为对应参数即可
    6. 随机队列: 添加时加入到数组尾部, 取出时随机取出, 然后将数组尾部元素移动到该位置
    7. 使用 $O(n)$ 的时间和 $O(1)$ 的空间可以逆转一个链表
    8. 使用异或可以只用一个指针就实现两个指针的效果

分治法

1. 分治: 将问题递归地划分为子问题, 分别进行处理, 最后将处理结果统合起来
2. MergeSort
    1. 进行二分, 然后在 $O(n)$ 的时间内将两个有序的数组融合 (Combine).
    2. 时间复杂度: $T(n) = 2T(n/2) + O(n)$

2. MergeSort
    1. 进行二分, 然后在 $O(n)$ 的时间内将两个有序的数组融合 (Combine).
    2. 时间复杂度: $T(n)=2T(n/2)+O(n)$
    3. 每一层复杂度都是 n, 一共 $\log n$ 层, 最后的时间复杂度为 $T(n)=n\log n$
    4. 或者使用一个 FIFO 队列, 每次取出两个有序数组, 融合成一个有序数组之后重新放入队列中
3. 整数乘法
    1. $xy=x_ly_l\cdot 2^n+[(x_l+x_r)(y_l+y_r)-x_ly_l-x_ry_r]\cdot 2^{\frac{n}{2}}+x_ry_r$
    2. $T(n)=3T(n/2)+O(n)$
    3. $T(n)=O(n^{\lg 3})<O(n^2)$
4. 矩阵乘法
    1. $T(n)=7T(n/2)+\Theta(n^2)$
5. 替代法
    1. 已知 $T(n)=7T(n/2)+cn^2$
    2. 猜测 $T(n)\leqslant d_1n^{\lg 7}-d_2n^2$
    3. 用归纳法, 带入, 放缩即可
6. 递归树法
    1. 将递归树画出来, 分析每一层的消耗
    2. 指数递减: 只看第一层, 则 $T(n)=O(f(n))$
    3. 各层相等: 每一层加起来, 则 $T(n)=O(f(n)\lg n)$
    4. 指数递增: 只看最后一层, 则 $T(n)=O(n^{\log_c r})$
7. 主定理
    1. 即递归树法的增强版本
    2. 通过分析 $f(n)$ 与 $aT(n/b)$ 的关系, 即可知道究竟是三种情况中的哪种情况
    3. 指数递减: $f(n)$ 比较大, $f(n)=\Omega(n^{\log_b a+\epsilon})$
    4. 各层相等: $f(n)$ 恰好相等, $f(n)=\Theta(n^{\log_b a})$
    5. 指数递增: $f(n)$ 比较小, $f(n)=O(n^{\log_b a-\epsilon})$
8. 划分规模不等
    1. 先使用递归树法获得一个猜测值
    2. 再使用替代法证明结果是正确的
9. 使用 MergeSort 可以用来计算逆序对个数
    1. 逆序对 = 两个子序列的逆序对 + 融合过程中产生的新的逆序对
10. 一些奇怪的复杂度分析
    1. $T(n)=T(n-2)+T(n/2)+n$
        1. 画出递推树, 发现可以向左下角倾斜地计算, 每一层 $(\frac{n}{4^i})^2$ 最多出现 n^i 次
        2. $T(n)\leqslant\sum_{i=1}^{\log n}(\frac{n}{4^i})^2\cdot n^i=n^{O(\log n)}$
    2. $T(n)=T(\alpha n)+T((1-\alpha)n)+cn$
        1. $T(n)=O(n\log n)$
11. 选举人问题
    1. 划分成两个子问题, 然后取出每一个组的主选举人和拥有相同派别的人数

堆

1. 二叉堆

1. 二叉堆
    1. 最大堆
        1. 顶部有着最大值
        2. 每个子堆也是最大堆
        3. 是完全二叉树
2. 用数组表示二叉堆
    1. 父节点: `idx / 2`
    2. 左子节点: `2 * idx`
    3. 右子节点: `2 * idx + 1`
3. 最大堆
    1. `HeapInsert()`
        1. 加到数组尾部, 然后与父节点比较, 大于则交换, 逐层向上
        2. $T(n) = O(\log n)$
    2. `HeapGetMax()`
        1. 返回数组的首值
        2. $T(n) = O(1)$
    3. `HeapExtractMax()`
        1. 去除数组头部, 将数组尾部放到头部
        2. 头部与两个子节点比较, 小于的话, 与更大的子节点交换, 逐层向下
        3. $T(n) = O(\log n)$
4. 优先队列
    1. 可以用最大堆实现优先队列
5. 堆排序
    1. 建好一个最大堆, 然后不断从中取出最大值
    2. 从数组中建堆
        1. 不断加入值
            1. $T(n) = n \log n$
        2. 或者直接用原数组建堆
            1. 逐层使用 `MaxHeapify(i)`
            2. 即 `for i in range(n / 2, 1) do MaxHeapify(i)`
            3. $T(n) = \sum_{h=0}^{\log n}(\frac{n}{2^h} \cdot O(h)) = O(n \cdot \sum_{h=0}^{\log n} \frac{h}{2^h}) = O(n)$
    3. 不断取出最大值, 然后放入堆尾部 (先把尾部的放置于堆顶)
        1. $T(n) = n \log n$
6. 在最大堆中取出第 k 大的元素
    1. 要求时间复杂度为 $O(k \log k)$, 而不是 $O(k \log n)$
    2. 使用另一个堆 H 来协助, 先往 H 里加入 M 的最大值
    3. 不断在 H 取出最大值, 然后往 H 放入该最大值的两个子节点
    4. 执行 $k - 1$ 之后, H 里就是第 k 大的元素了

排序问题

4. 执行 $n-1$ 之后, H 里就是第 n 大的元素了

排序问题

1. 排序算法的性质
   1. 插入排序
      1. $O(n^2)$ 时间, $O(1)$ 额外空间
      2. 稳定
   2. 归并排序
      1. $O(n\log n)$ 时间, $O(n)$ 额外空间
      2. 稳定
   3. 堆排序
      1. $O(n\log n)$ 时间, $O(1)$ 额外空间
      2. 不稳定
2. 快排
   1. 选出主元
   2. 使用 `Partition` 将数组分为两部分
   3. 还可以加入随机化算法
3. 问题的上界和下界
   1. 对于某个具体的问题, 有其对应的上界和下界
   2. 对于某个具体的算法才有上界, 即最坏情况下算法 A 所需要的时间
   3. 解决该问题的所有算法对应的上界, 取其最小值, 即是下界
4. 如何证明问题的下界

   1. 使用对手论证, 假设一个 Eve 在与你作对, 然后你问问题, 它回答, 你要问够足够多的信息才有可能正确回答这个问题
   2. 例如可以使用这种方式证明 $\Omega(n)$ 是排序问题的一个下界
5. 如果计算问题的下界
   1. 使用决策树法
   2. 我们使用比较的方法来进行排序, 每次比较都会分成两个分支, 所以是二叉树
   3. 排序问题的所有结果可能性, 共有 $n!$ 中, 即这棵树至少有 $n!$ 片叶子
   4. 这棵树的高度至少为 $\log n! = \Omega(n\log n)$
6. 桶排序
   1. 不基于比较, 类似于哈希表
   2. 加入到 d 个桶之后, 然后进行排序
7. 基数排序
   1. 桶排序的一种应用场景
   2. 可以用来排序字符串

选择问题

1. 同时找出最大值和最小值
   1. 我们先两两分组, 然后找出 "局部" 最大值和最小值
   2. 然后从这些 "局部" 最大值和最小值中分别找 "全局" 最大值和最小值
   3. 这也是我们能做到的最好了
      1. 给每个元素标号 "+" 表示可能为最大值, "-" 表示可能为最小值

    2. 然后从这些"局部"最大值和最小值中分别找"全局"最大值和最小值
    3. 这也是我们能做到的最好了
        1. 给每个元素标号, "+" 表示可能为最大值, "-" 表示可能为最小值
        2. 我们要做的就是把标号清除
2. 通用选择问题
    1. 魔改快排
    2. 我们可以证明, 预期花费的时间是 $O(n)$
    3. 并且, 在找到之后, 会将数组左边变为都比其小的元素, 右边变为都比其大的元素
    4. 我们可以使用 Median of medians 的方法, 将预期为 $O(n)$ 变为一定是 $O(n)$

树

1. 二叉树的分类
    1. 满二叉树: 任何一个节点要么没有子节点, 要么有两个子节点
    2. 完全二叉树: 类比堆
    3. 完美二叉树: 刚刚好形成一个三角形的那种情况
2. 树的遍历
    1. 先序遍历
    2. 中序遍历
    3. 后序遍历
    4. 层次遍历: 使用一个 FIFO 队列实现
3. 二叉搜索树
    1. 唯一要注意的就是删除, 要用到找后继

哈希

1. 链表式哈希表
    1. 搜索 `search` 用时 $O(k)$, k 是链表长度 (也即冲突程度).
    2. 插入 `insert` 用时 $O(1)$, 只需要插入链表头即可.
    3. 删除 `remove` 用时 $O(1)$ (存疑)
2. 简单均匀哈希假设
    1. 加载因子 $\alpha = n/m$
    2. 平均搜索耗时 $O(1+\alpha)$
3. 统一哈希
    1. $P(h(x)=h(y)) \leqslant \dfrac{1}{m}$ 对于所有 $x \neq y$
    2. 常见哈希族 $ak+b \mod p$
4. 开放寻址哈希
5. 完美哈希表
    1. 空间为 $m=n^2$ 即 $O(n^2)$ 的哈希表只需要随机选取哈希函数即可
    2. 空间为 $O(n)$ 的完美哈希表
        1. 先建一个空间为 $O(n)$ 的一级链表式哈希表
        2. 对于每个含有多个元素的链表, 建造 $O(n^2)$ 的二级完美哈希表
        3. 组合起来就是一个空间为 $O(n)$ 的哈希表了
        4. 但是只支持查询操作

均摊开销

        3. 组合起来就是一个空间为 $O(n)$ 的哈希表」
        4. 但是只支持查询操作

均摊开销

1. 余额法
2. 势函数法

并查集

1. 并查集接口
    1. `MakeSet(x)`
    2. `Union(x, y)`
    3. `Find(x)`
2. 并查集可以用来实现连通图的判定
3. 基于链表的并查集
    1. 使用最普通的方式的话, 一系列 `Union` 操作的时间复杂度为 $O(n^2)$
    2. 基于权重的话, 一系列 `Union` 操作的时间复杂度为 $O(n \log n)$
4. 基于树的并查集
    1. 基于路径压缩, 每次 `Find` 的时候将路径上的节点的父节点都改为根节点
    2. 分析时间复杂度的话, 改为 `PartialFind`

图

1. 图的表示方法
    1. 邻接矩阵
        1. 无向图邻接矩阵是对称矩阵
        2. 对角线上的边要存在自环时才有值
        3. 易于判断两个顶点是否相邻, 难于穷举所有边
    2. 邻接链表
        1. 空间开销小
        2. 易于穷举所有边, 难于判断两个顶点是否相邻
2. 图遍历
    1. 宽度优先搜索 (BFS)
    2. 深度优先搜索 (DFS)
        1. 树边: DFS 森林里的边
        2. 回边: 连接节点和其祖先节点的边
        3. 向前边: 连接节点和其孙代及往后节点的边
        4. 交叉边: 其他类型的边
        5. 括号定理: 发现时间和完成时间就像匹配的左右括号
        6. 白路径定理: 当前节点和后代节点当且仅当有一条全为白色顶点的路径
        7. 无向图的边要么是树边要么是回边
3. 宽度优先搜索可以用来判断无向图的二部图
    1. 没有奇数环的无向连通图的二部图
    2. 给每个节点附上 BFS 生成的 `dist` 值, 即可通过其判断
4. 深度优先搜索的应用

        1. 没有奇数环的无向连通图的二部图
        2. 给每个节点附上 BFS 生成的 `dist` 值, 即可通过其判断
4. 深度优先搜索的应用
    1. 有向无环图 (DAG)
    2. 拓扑排序
        1. 每个 DAG 都有其拓扑排序
        2. 每个 DAG 至少有一个源点和一个聚点
        3. 算法: 进行 DFS, 计算所有节点的完成时间, 当节点完成时加入链表头部
    3. 强连通分量 (SCC)
        1. 有向图的强连通分量图 (Component Graph) 是 DAG
        2. 算法:
            1. 计算 G^R
            2. 在 G^R 中运行 DFS, 计算完成时间 f
            3. 依照 f 的逆序来进行 `DFSAll`
            4. 每一棵 DFS 树就是一个 SCC
        3. 时间复杂度 $O(|V|+|E|)$
5. 最小生成树
    1. 切属性 (Cut Property)
    2. Kruskal 算法
        1. 将边按照权重排序 (时间复杂度为 $O(|E|\log|E|)$)
        2. 为每个顶点设立一个并查集
        3. 遍历每一条边, 通过并查集判断他们是否成环
            1. 不成环就将边加入 A
            2. 并且将两个顶点 `Union`
        4. 时间复杂度为 $O(|E|\log|V|)$
    3. Prim 算法
        1. 从任意一个顶点开始, 设定 `dist`, 然后为所有顶点建立一个优先队列
        2. 进入循环, 不断从优先队列取出顶点, 并设置 `u.in = True`
        3. 对与 `u` 相连的每个顶点 (边) 进行遍历, 并在优先队列中更新
        4. PS: 此处与迪杰斯特拉算法的不同在于, 是更新 CC 与其他顶点的距离, 而不是 `s` 与其他顶点的距离
        5. 时间复杂度为 $O(|E|\log|V|)$
    4. Boruvka 算法
        1. 并行化的 Prim 算法, 同时对所有连通分量进行合并
    5. 无向图边权重不同, 则最小生成树不同
6. 单源点最短路径 (SSSP)
    1. 四种情况:
        1. 单位权重: BFS
        2. 任意正数权重: Dijkstra
        3. 任意权重无环: DAGSSSP
        4. 任意权重: Bellman-Ford
    2. Dijkstra 算法

        4. 任意权重: Bellman-Ford
    2. Dijkstra 算法
        1. 从任意一个顶点开始, 设定 `dist`, 然后为所有顶点建立一个优先队列
        2. 进入循环, 不断从优先队列取出顶点
        3. 对与 `u` 相连的每个顶点 (边) 进行遍历, 并在优先队列中更新
        4. 时间复杂度为 $O(|E|\log|V|)$
    3. Bellman-Ford 算法
        1. 基于对最短路径上跑 Dijkstra 中的 update 肯定不会出事的思想
        2. 我们进行 n - 1 次循环
        3. 每次循环对每一条边进行处理, 更新 dist
        4. 时间复杂度为 $O(|E|\cdot|V|)$
        5. 并且如果我们加上第 n 次循环, 就可以判断是否有负环
    4. DAGSSSP 算法
        1. 对于有向无环图, 我们只要使用 DFS 进行一次拓扑排序, 就能知道所有最短路径均位于拓扑排序中
        2. 按照拓扑排序进行 update
        3. 时间复杂度为 $O(|E|+|V|)$, 甚至小于 Dijkstra 算法
        4. 应用上: 可以用来判断关键路径, 可以说是迭代式动态规划在图论上的一个算法
7. 全配对最短路径 (APSP)
    1. Johnson 算法
        1. 修改 Dijkstra 算法以适应负边
        2. 只需要 $\hat{w}(u,v)=h(u)+w(u,v)-h(v)\geqslant 0$
        3. 我们加入一个节点 z, 其与所有 G 中的顶点以 0 权重的边连接
        4. 则只需要 $\hat{w}(u,v)=\operatorname{dist}(z,u)+w(u,v)-\operatorname{dist}(z,v)$
        5. 算法:
            1. 创建包含 z 的新图
            2. 通过 Bellman-Fold 算法获取 $h(v)=\operatorname{dist}(z,v)$
            3. 基于 $\hat{w}(u,v)=h(u)+w(u,v)-h(v)$ 进行 Dijkstra 算法
        6. 时间复杂度为 $O(|V|^3\log|V|)$
    2. Floyd-Warshall 算法
        1. 基于表达式 $\operatorname{dist}(u,v,r)=\min(\operatorname{dist}(u,v,r-1),\operatorname{dist}(u,x_r,r-1)+\operatorname{dist}(x_r,v,r-1))$
        2. 先设置每条边 (u,v) 的 dist 即 `dist[u, v, 0] = w(u, v)`, 否则正无穷
        3. 迭代 n 次, 并对每个顶点对 u,v 进行更新, 依照上面提到的那个数学表达式
        4. 时间复杂度为 $O(|V|^3)$

贪心与动态规划

1. 贪心算法
    1. 活动选择问题
        1. 算法
            1. 按照结束时间进行排列
            2. 总是选取结束最早且兼容的活动加入
        2. 证明
            1. 先证明贪心策略, 最早结束的活动必然位于一个最优解法中, 可以使用反证法证明
            2. 然后证明最优子结构性质, 即可以分解成最早结束的活动和剩下活动对应的子问题

            1. 先证明贪心策略, 最早结束的活动必然位于一个最优解法中, 可以使用反证法证明
            2. 然后证明最优子结构性质, 即可以分解成最早结束的活动和剩下活动对应的子问题
            3. 然后使用数学归纳法证明
      2. 最优子结构: 问题的最优解法包含其子问题的最优解法
      3. 贪心策略: 不用知道子问题的最优解法的结果, 便能做出当前问题的选择
      4. 拥有最优子结构是贪心算法和动态规划算法的前提, 是否拥有贪心策略是二者的差别
      5. 可分背包问题
         1. 物品可分
         2. 计算所有物品的利润比, 进行排序, 从利润高到低不断选取
      6. 零一背包问题
         1. 物品不可分
         2. 不能使用贪心算法
      7. 霍夫曼编码
         1. 将所有字符的频次统计出来
         2. 将其变为一棵满二叉树
         3. 不断合并两个最低频次的字符
         4. 使用优先队列来实现
      8. 集合覆盖
         1. 总是选取当前能覆盖最多剩余节点的集合
         2. 但是这只能给出接近最优的解法
         3. 设 k 为最优使用集合数, 则能保证最多只用 $k\log n$ 个集合
            1. 设 n_t 是经过第 t 次迭代后的剩余节点数
            2. 由最优解法是 k 可知 n_t 能被 k 个集合覆盖
            3. 所以每一次迭代至少能覆盖掉 n_t/k 个剩余节点
            4. 因此 $n_t \leqslant n_{t-1} - n_{t-1}/k \leqslant n_0(1-1/k)^t < n_0(e^{-1/k})^t = ne^{-t/k}$
            5. 因此 $t = k\ln n$ 便有 $n_t < 1$, 即终止
2. 动态规划
   1. 木棒切割问题
      1. 最优子结构: $r_n = \max\limits_{1\leqslant i\leqslant n}(p_i + r_{n-i})$
      2. 但是不能使用贪心策略, 可以找出反例
   2. 动态规划可以有两种形式
      1. 递归型动态规划
         1. 自顶向下算法
         2. 使用一个数组或哈希表进行对子问题答案的记录
         3. 易于理解, 但是代码量和空间复杂度相对较大
      2. 迭代型动态规划
         1. 自底向上算法
         2. 使用一个数组对子问题答案进行记录
         3. 找出一个顺序, 使得能够保证当前问题的所有子问题已解决
            1. 一维一般就是从开始到结尾或从结尾到开始
            2. 二维一般就是从矩阵左上角方块向右下角扩展
            3. 图论一般就是拓扑排序

        1. 一维一般就是从开始到结尾或从结尾到开始
        2. 二维一般就是从矩阵左上角方块向右下角扩展
        3. 图论一般就是拓扑排序
3. 矩阵乘法
    1. 可以使用结合律减小计算开销
    2. 不管是什么顺序, 最后步骤一定为 $(A_1A_2\cdots A_k)\cdot(A_{k+1}\cdots A_n)$
    3. 所以有 $m[i,j]=\min\limits_{i\leqslant k<j}(m[i,k]+m[k+1,j]+p_{i-1}p_kp_j)$
4. 编辑距离
    1. 判断两个字符串的相似程度
    2. 对字符串最末尾分析, 可以分成三种情况
        1. $\frac{-}{B[n]}$ 或 $\frac{A[m]}{B[n]}$ 或 $\frac{A[m]}{-}$
        2. 然后就可以拆分成子问题, 并有数学表达式
        3. $\text{dist}(i,j)=\min(\text{dist}(i,j-1)+1,\text{dist}(i-1,j)+1,\text{dist}(i-1,j-1)+1(A[i]=B[j]))$
5. 最大独立集
    1. 独立集是从图中选出一系列不相邻顶点的集合
    2. 如果有环就很难判断, 但是对于树很好分析
6. 最短路径问题有最优子结构性质, 最长路径问题没有最优子结构性质
7. 自底向上的迭代型动态规划, 在一定情况下可以减小空间开销
    1. 例如 Floyd-Warshall 算法, 可以将 `dist[u,v,r]` 变为 `dist[u,v]`, 效果一致, 空间开销却从 $O(n^3)$ 降为 $O(n^2)$
    2. 编辑距离问题这种二维问题, 也可以通过保存一个 `distLast[n]` 和 `distCur[n]` 来替换原来的 `dist[n, r]`, 空间开销从 $O(n^2)$ 降为 $O(n)$
8. 子集求和问题, 无法用动态规划减小时间开销
    1. 给定元素个数为 n 的整数集合, 是否有一个子集相加等于 T
    2. 用最简单的遍历法, 时间开销为 $O(2^n)$
    3. 用动态规划
        1. 定义 $ss(i,t)=true$ 当且仅当 $X[i\cdots n],t$ 有解
        2. $ss(i,t)=\begin{cases}true, & t=0\\ ss(i+1,t), & t<X[i]\\ false, & i>n\\ ss(i+1,t)\vee ss(i+1,t-X[i]) & \end{cases}$
        3. 时间开销为 $O(nT)$

选择考点猜测

2024年1月6日　　22:23

复杂性:

信息熵

(我最喜欢的小猪佩奇试毒药 $(T+1)^x \geq N$ T测试次数，x小猪个数，N毒药个数)

小球称重，

1. 有 N 个小球，已知有 1 个小球比其他小球偏重，使用天平，最少要多少次才能确保找到那个小球？（题干改成「偏轻」也可以）
2. 有 N 个小球，已知有 1 个小球和其他小球质量不同，使用天平，最少要多少次才能确保找到那个质量不同的小球（并知道它比其他小球轻还是重）？
3. 有 N 个小球，已知有 1 个小球和其他小球质量不同，使用天平，最少要多少次才能确保找到那个质量不同的小球（无需知道它的轻重）？

这 3 个问题，表述上有微小差别，答案也有所不同。第 1 个问题的答案是 $\lceil \log_3 N \rceil$（*⌈ ⌉代表向上取整*），第 2 个问题的答案是 $\lceil \log_3(2N+3) \rceil$，第 3 个问题的答案是 $\lceil \log_3(2N+1) \rceil$。

换句话说，使用天平 k 次，在 3 个问题中，我们分别最多能在 3^k 、 $\frac{3^k-3}{2}$ 、 $\frac{3^k-1}{2}$ 个小球中分辨出问题小球。

主定理

分治递归树求复杂度

数据结构:

均摊分析

并查集 $(nlg(n), \lg(n), lg*(n))$

hashtable $(O(1+\alpha)|O\left(\frac{1}{1-\alpha}\right), O(\alpha \lg(\frac{1}{1-\alpha}))$

二叉搜索树 插入删除，红黑树插入

各类搜索的复杂度比较

	Search(S,k)	Insert(S,x)	Remove(S,x)
BinarySearchTree	$O(h)$ in worst case	$O(h)$ in worst case	$O(h)$ in worst case
Treap	$O(\log n)$ in expectation	$O(\log n)$ in expectation	$O(\log n)$ in expectation
RB-Tree	$O(\log n)$ in worst case	$O(\log n)$ in worst case	$O(\log n)$ in worst case
SkipList	$O(\log n)$ in expectation	$O(\log n)$ in expectation	$O(\log n)$ in expectation

图论:

dfs bfs

拓扑排序

特殊问题:

Sort

Select 中间的中间

强连通图
最小生成树
最短路径