AVL Trees

- 子树移动规则：trouble finder原来被占的子树位置填充多出来无法安放的子树
- 下以p代表parent，g代表grandparent(trouble finder)
- balance factor
- LL（右旋），RR（左旋），LR（先左旋再右旋），RL
    - LR适用于“<”型，先对p旋转变直，再旋转g保持平衡
    - 形状的判定：从trouble finder到trouble maker所画的路径

树高问题

- 空子树树高为-1
- $$n_h = n_{h-1} + n_{h-2} + 1$$

  根据递推式可推出树高为h时AVL树的最少结点数

算法复杂度

- 建树：NlogN
- 搜索/插入： logN
- 删除：logN

Splay tree(伸展树)

- 从空树开始，任何 M 次连续的操作一共最多消耗 O(M log N) 的时间（单次可为O(N)），像AVL一样实现完全平衡是比较繁琐的，splay tree从摊还代价的角度提供了一个要求更低的做法
- 摊还时间界：O(logN)
- 不要求保留高度或平衡信息，可节省空间简化代码
- 操作：在树中，每次有节点被访问到，就将其旋转到根节点（查找中进行）
    - 如果只是单旋，在插入1..N后，恢复为skewed tree，依次访问可达到$\Omega(N^2)$
    - 分类讨论

查找

- X的父结点是树根：旋转X和P
- X的父结点不是树根：
    - zig-zig: 对G、P依次进行相同方向的旋转
    - zig-zag: 正常的AVL双旋

删除

- 先查找要删除的结点，经过查找操作，要被删除的结点被移动到了根节点
- 接着查找起左子树的最大元素用来替代，在查找过程中，该用来替代的结点被挪到了左子树的根节点，且没有右孩子
- 让被删除结点的右孩子成为用来替代的结点的右孩子，即完成操作

Amortized Analysis

- 分析的是一个序列的连续操作
- 相比于平均分析，不包含概率
- worst-case **bound** >= amortized **bound** >= average-case **bound**
- 把耗时长的大操作付出的代价摊还给每个耗时短的小操作，让大家共同承担

Aggregate analysis（聚合法）

- 一个**任意n**个操作的序列**最坏情况下**花费的总时间为T(N), amortized cost=T(N)/N

Accounting method（核算法）

- 摊还代价时刻大于实际代价
- 摊还代价为**预设支付给操作的钱**，实际代价为真正耗费的钱，余额作为credit，当预设支付的钱不够，则从credit中扣除
- 摊还的结果是：所有预设支付的钱/总操作数，预设支付的钱取最大那个

Potential method（势能法）

- g **good** potiential function should always assume its minimum at the start of the sequence（一般设为0，并证明后面的势能都大于0）
- 每一步的摊还代价=实际代价+势能变化
- 证明splay tree: 用rank：$\sum logSize(i)$ 即一棵树的大小（包括自己）再取对数
- 代价大的操作，势能应下降多一些，这样就能相互抵消
- 分析**单个操作**如证明dequeue和enqueue的摊还代价，则$\hat{c_i} = c_i + \phi(D_i) - \phi(D_{i-1})$，对单步进行分析即可
- 分析一个序列的操作，则用$\sum c_i + \phi(D_n) - \phi(D_0)$，要用上面得到的单步操作的上界来进行分析（取最大的乘以n）
    - 一般要保证n次操作后的势能大于初始的势能，这样得到的总代价一定会大于实际代价，才能得到**上界**

Red-Black Tree

- Every node is either red or black.

- The **root** is **black**.
- Every **leaf** (NIL) is **black**.
- If a node is **red**, then both its children are black.
- For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

Property

- 空子树高度为0,可以理解为NIL也是一个结点
- 树高：该结点到叶结点的路径长
- black height ：从 x 到叶节点的路径中黑色节点的个数（不包括自己，包括NIL）
- 区分internal node和external node(nil)
- $sizeof(x) >= 2^{bh(x)}-1$
- $bh(Tree) >= h(Tree)/2$
- $h(Tree) <= 2ln(N+1)$

插入

- 先按二叉搜索树的定义插入，保证二叉搜索树特性
- 通过改色、旋转来保证红黑树特性（注意旋转是保证二叉搜索树特性而改变树的结构的方法）
- 6种情况，T=O(h)=O(lgN)
- case
- if为根，直接涂黑，结束
    - 叔叔和parent都是红色：改色，转移到grandparent
- uncle是黑色（或nil)，x是右孩子：左旋变成左孩子（注意在旋转的过程中，parent和child的指针有所变化）
    - uncle是黑色（或nil），x是左孩子：改色，右旋
- 最后记得把根涂黑
- 都红则换色**转移**（不做旋转），不在一边则旋转，gp为红则换色旋转

删除

- 先按二叉搜索树的特性进行删除，保留原结点的颜色不变

- 再调整红黑树的特性。二叉搜索树删除1/2-degree node都会使用替代的方法，最后都要删除替代的结点，也即最后一定会删除一个**叶结点**。
  - 判断case时，x是用来替代并最后被删除的**叶结点**的**位置**，最后要把x删掉
  - 替代的时候，被替代位置的颜色保持不变
  - 当被删除的叶节点是红色，不影响红黑树性质，直接删除即可
  - 当被删除的叶节点是黑色，该路径黑高减少1，需要通过换色和旋转来保证被**根到替换位置**的路径**多一个黑结点**
- case
  - case1: parent黑，sibling红
  - case2: sibling和sibling的孩子都为黑
  - case3: sibling黑，远离自己的sibling孩子为黑 把sibling变红，孩子变黑，旋转
  - case3: sibling黑，远离自己的sibling孩子为红： sibling和parent换，孩子变黑

B+ Tree

Definition

- 根是**叶子**或**拥有*2~M个孩子**

- 所有除**根节点**以外的**叶子结点**有$\lceil M/2 \rceil$到M个孩子

- 所有叶子都在同一个深度

- **非根的叶节点**存储$\lceil M/2 \rceil$到M个key，非叶子结点存储M-1个key（M个指针）

- 复杂度：插入一个数据最坏情况是O(M)(插入数组内需要移动，每个节点的数据都要被改变，即都需要被插入)，则整个插入过程=树高*插入

- $T_{insert}(M,N) = O(M/logM)logN)$

- $Depth(M,N) = O(\lceil log_{\lceil M/2 \rceil} N \rceil)$ *最坏情况是每个结点都只有M/2向上取整个孩子*

- $T_{find}(M,N) = O(logN)$ *在每个结点的M个key里，要查找适合的branch，可以进行二分查找，depth乘以搜索复杂度logM，最后得到的查找复杂度与M无关*

删除

- 如果删除后key数量大于等于$\lceil M/2 \rceil$，直接删
- 如果小于：
  - 看左边结点可不可以借一个key，可以则借
  - 如果不可以借，直接和左边的结点进行合并

Inverted File Index

- Inverted file: Term Dictionary+ Posting List
- 搜索时从出现频率最小的词语开始找起可以更快
- retrieve: 查询

Index Generator

- 每读一个文件，就要对每个词进行分析和插入
- Word Stemming
- Stop words

Access Terms

Search trees/hashing

Memory management

- 读入文件，本来是有一个dictionary的空间来存储inverted index，若读入一个词时内存超出，则将当前的block写入磁盘（先不插入到最后的inverted index中）
- 将词语插入到inverted Index里面
- 在磁盘中，将所有block进行merge。在写磁盘前先排好序，归并效率更高

Distributed indexing

- Term-partitioned index
- Document-partitioned index

Dynamic indexing

- 删除不常索引的文件

- main index, auxiliary index(小型辅助的index，存储新进来的document，负责插入)

- 当auxilary index达到一定程度，就把这个部分并入main index

- 搜索时，要搜索main index和auxilary index

Compression

- 删除stop words, 把所有单词摊开成一条长串
- 用数组存储单词所在位置的下标
- 但下标可能会变得很大，可以采用**差值**的方法存储

Thresholding

- 两种阈值
  - Document: 只检索前面 x 个按权重（单个单词出现的频率）排序的文档
  - Query: 把句子中的词语按频率进行排序，只检索频率比较小的

User happiness(区分两种)

- **Data** retrieval performance
  - 响应时间
  - 索引空间
- **Information** retrieval performace
  - **相关性**
  - 需要:
    - A benchmark document collection （基准）
    - A benchmark suite of queries
    - A binary assessment of either Relevant or Irrelevant for each query-doc pair
  - recall和precision: 都是基于对方是1时进行的

Leftist Heap

- 传统二叉堆merge需要$\theta(n)$，左式堆的出现就是为了加快merge的速度，通过保证右路径的长度为一个较小的值来实现，左式堆是趋向不平衡的树
- 左式堆使用和二叉堆相同的顺序，不同的结构（操作在右路径上进行）
- Npl:
  - the length of the **shortest** path from X to a node without **two children**. Define Npl(NULL) = –1.
  - NPL=min(childeren_npl)+1
- 右路径：指最右边的路径，为最短路径
- **右路径上的结点**：注意包含了根节点！！
- 所有操作都在右路径上操作
- θ(N)降至**O(logN)**
- Merge的时候，算法复杂度是两个堆的右路径长之和，是O(logN1)+O(logN2),不是O(log(N1+N2))

定理

- 右路径上如果有r个结点，那么左式堆至少有$2^r - 1$个结点

- 一条**右路径**最多含有$\lfloor log(N+1) \rfloor$个结点

- 从空树开始按递增顺序插入$2^k + 1$个结点，会得到完美二叉树

- Merge:
    - 判断是否为空
    - 判断是否为single node
    - 比较根节点大小，Merge较小根节点的右孩子和另一左式堆
    - 递归进行
    - 如果Npl不符合，**交换左右孩子**
    - **更新Npl**
    - 返回H1
- DeleteMin:
    - 删除根节点
    - 合并两个子树

operation	time
merge/insert	O(logN)
deleteMin	O(logN)

Skew Heaps

operation	time
all	$T_{amortize}$=O(logN)

- 和左式堆类似，都是对右路径进行操作，是左式堆的改进版本（条件更宽松），目标是**任意M个连续**的操作都最多使用**O（MlogN）**时间，最坏情况可以达到O(N)
- 由于左式堆总是对右路径操作，要判断npl来进行左右子树的交换，比较麻烦，skew heap选择要操作的结点并不断交换左右孩子，没有孩子时，直接插入左侧，**不再需要npl**
- 注意是对每个经过的结点都要交换
- 如果有左孩子，交换
- 如果只有右孩子，直接插入
- 实际上，在不停的交换中，只对右路径进行了操作

定理

inserting keys 1 to 2^k-1 for any k>=0 in order into an initially empty skew heap is always a full binary tree.

摊还分析

- Note that the number of **descendants** of a node includes **the node itself.**

- heavy: 右子树的大小>=整棵树大小/2
- 选择heavy node的数目作为势能函数
- 总时间为T1+T2右路径的结点数
- T=l1+h1+l2+h2(l表示Light，h表示heavy)
- Merge时
  - 只有**右路径**上的node会改变heavy和light的状态(只有右路径上的点会进行swap)
  - 右路径的heavy node一定会变为light，因为原本为heavy，swap之后变为light
  - 部分light node会变为heavy node（因为light可能是左右相等？如叶节点）
  - 最坏的情况是所有light node都变成heavy node
  - $T_{amortized} = T_{worst} + \phi_{i+1} - \phi_i$ T_{worst}是两棵树的右路径长之和

Binomial Queue

- 改善插入时间复杂度：原来的heap单个插入平均O(logN), 从空开始建堆需要O(N)，则插入平均为O(1)，因此插入时间需要优化

对于一般树的特性：（不用二叉树表示）

- Bk共有2^k个节点。

- 最多有$\lceil logN \rceil$个root

- Bk的高度为k，共有k+1层

- Bk在深度i处恰好有C(k,i)个节点，其中i=0,1,2,...,k。

- 根的度数为k，k个孩子分别为B0，B1...Bk-1

- 任意大小的优先队列可以由二项队列唯一表示（二进制）

Operation

- Combine Tree: O(1)

- 遍历findmin: O(logN) （树的数目**最多**为$\lceil logN \rceil $，假设从1号位开始递增全部都有树)

- Merge:O(logN) **二进制表示**

- Insert:

  - 单次：c*(i+1) (Bi为最小的空树，插入是1，Link i 次)
  - 从空开始插入N次：O(N). Tavg=O(1), Tworst=O(logN),Tamortized=2=O(1)
    - 定义势函数为当前树的数目

    - 通过观察可以知道，如果一次操作（包括插入、link）消耗了c个单元时间，那么heap里面的树会变化2-c棵，因此$\hat{c}_i = c_i + (\phi_i - \phi_{i-1}) = c + 2 - c = 2$

- DeleteMin: O(logN)

    - 找到最小值 O(logN)
    - 删除该树
    - 把该树的根删除，再把k-1个孩子attach到一个新堆上 O(logN)
    - Merge

数据结构

- 需要：可以快速找到所有子树，且所有子树要被排好序

- firstchild-nextsibling （一般树变为二叉树表示，对Bk而言，B0..Bk-1都是child，而在其中，Bi又有自己的子树，对root而言，B0...Bk-1互为sibling，根只有child没有sibling）

    - 可以先写出原来的树再化为二叉树形式

- 从上往下子树大小按decreasing order排序

- 按二叉树的看法，如果以firstchild为根，它总是**完全二叉树**, 因为firstchild的左子树和右子树都是串好的Bk-2...B0

Backtracking

- pruning 剪枝

- 对Xi, 遍历下一步Xi+1，可以则继续; 若遍历所有i均为找到解决方案，删除Xi, 回溯到Xi+1

- 套模板即可，如果要求有最大最小等，可以先对序列进行排序或者从前往后/从后往前进行遍历

```
bool backtracking(int x){
    if(x>n){
        do something on the final result or check if result is ok;
        return;
    }
    for(every x in S){
            assume x is ..
            ok=check(x)
            if(ok){
            flag=backtracking(x+1);
            if(flag)
                return true;
```

```
            undo x;
        }
    }
    return false;

}
```

Divide and Conquer

- 主要解决递归式里的f(N)，如果是线性的，可以达到O(NlogN)

sample1: 最近点对问题

- 和寻找最大子序列和一样，先找两边，再找跨中间的点对
- 问题: 找中间点对最坏情况可以到达O（N^2)
- 优化寻找策略： 设在两边分别得到的点对最小距离为a、b，取σ=min(a,b), 作多个$2\sigma * \sigma$矩形
- 问题变成：在矩形里，最多可以容纳几个点，使其两两之间距离>=σ?
    - 注意到点对必须是**跨中间线**的，否则如果有两个点同时在一个小正方形里，会有矛盾
    - 在一个小正方形里，最多可以容纳4个点，使其两两之间距离>=σ（鸽巢证明）
    - 由此，在一个矩形中，最多可以容纳8个点（其中有两个在中间线重合）
    - 对于1个点，该点只需要搜寻其他的7个点，也即，在一个矩形中，查找最小距离的时间复杂度是**常数**

Substitution Method

- 猜一个值，并用归纳法证明
- 注意最后得到的形式一定严格用最开始的c（形式要相同）

Recursion-tree Method

- 对于不完整的，看作完整求出上界即可

Master Method

- 主要是讨论拆分部分还是合并部分对算法复杂度的决定性大，耗时更大的那一部分决定了算法复杂度

补充定理

- $a^{log_b n} = n^{log_b a}$
- $T_1(N) + T_2(N) = max(O(f(N), g(n)))$
- $T_1(N) * T_2(N) = O(f(N) * g(N))$
- 可以用极限来计算
- 如果another form计算不了，就用原始的形式

tips

- 在推渐进界时，可以用cn-d这样更小的界去推

- 必须显示证明和归纳假设严格一致的形式

- 可以换元

- lgN在证明时常换为lg(N-2)，把底看为2，并放缩n-2<n-2+1

Dynamic Programming

- 用空间换时间

```
初始化
切分子问题（步长  起点
初始化该子问题答案
遍历切割点，取最小
```

Ordering Matrix Multiplications

- 分成两段
- 计算$M_i * \ldots M_j$的最优时间m_{ij}为递归式：
  - i=j 0
  - j>i $min(m_{il} + m_{l+1,j} + r_{i-1}r_l r_j)$ //两边分开计算，最后两个矩阵相乘
- 三重循环：步长，起点，中间值L，自底向上，利用递归式子更新矩阵
- $O(N^3)$

Optimal Binary Search Tree *The best for **static searching** (without insertion and deletion)*

- 本质是建构子树并迭代构建OBST
- 左子树和右子树在自己的树里整体深度比在整棵树里少1，因此对整棵树而言，cij要加上左子树和右子树的weight

- 先按字典序**排序**，再切分，自底向上
- 对切分的子问题进行分析，计算optimal cost，选择对应的根并进行更新
- 自顶向上重新构建OBST
- $O(N^3)$

All-Pairs Shortest Path

- k是路径长（不算权重），由小的距离推出远的距离的带权重的值

Process

- characterize an optimal solution (找到合适的子问题)
- recursively define the optimal values （递归等式）
- compute the values in some order（自底向上）
- reconstruct the solving strategy

一般把子问题切割的条件放在外循环, 如子问题的规模大小

两种结构

- top-down with memorization

- bottom-up method

  ```
  initialize;
  for every subproblems{
      initialziez the final answer;
      traverse all the posssible options;
  }
  ```

Elements of DP

- 重叠子问题 Overlapping sub-problems
- 最优子结构(optimal sunstructure)
  - 问题的最优解由对应子问题的最优解组合而成，而这些子问题的最优解可以**独立**求出
  - 最长路径问题不符合"独立求出"的要求，在求解某一个子问题时由于占用资源，导致在另一个子问题中该资源不能被使用，也就产生了关联性

不可以使用DP的情况

- history-dependency
- sub-problems are not overlapping. （如斐波那契数会重复计算小的斐波那契数，但像二分查找等，可以直接用分治法）

Greedy

- 能用greedy解决的，通常都可以用DP解决，但是DP会更笨重（cumbersome)
- constraints, feasible solutions, optimal solution, greedy criterion
- 每一步optimal的结果都被包含在最终的结果里面且不会改变，才能使用greedy

证明使用greedy的合理性：

- 证明greedy-choice的正确性
- 证明最优子结构的正确性

Activity selection algorithm

- 可以使用DP，对于每个事件ak，前后加起来即可，O(N^2)

- greedy O(NlogN)

  - 选择最早结束的事件
  - 最晚开始的事件也可行
  - 将问题缩减到了二维

```
//S_k: 在事件a_k后发生的事件
//s: start f: finish k:subproblem n:size
f_recursive(s,f,k,n){
    m=k+1;
    while(m<=n && s[m]<f[k])
        m=m+1; //找到S_k里最早结束的事件
    if(m<=n)
        return {a_m} union f_recursive(s,f,m,n);
    else
        return empty_set;

}

f_recursive(s,f,0,n); //the first subproblem starts from 0
```

Huffman Codes

- 用不同长短的编码来处理不同频率的字母，可以缩减存储空间

- prefix code：

  - 没有任何一个编码是另外任意编码的前缀（否则就会有二义性）
  - prefix code的树满足的条件： 所有的字符都被放在树的**叶节点**上，且树为**full**

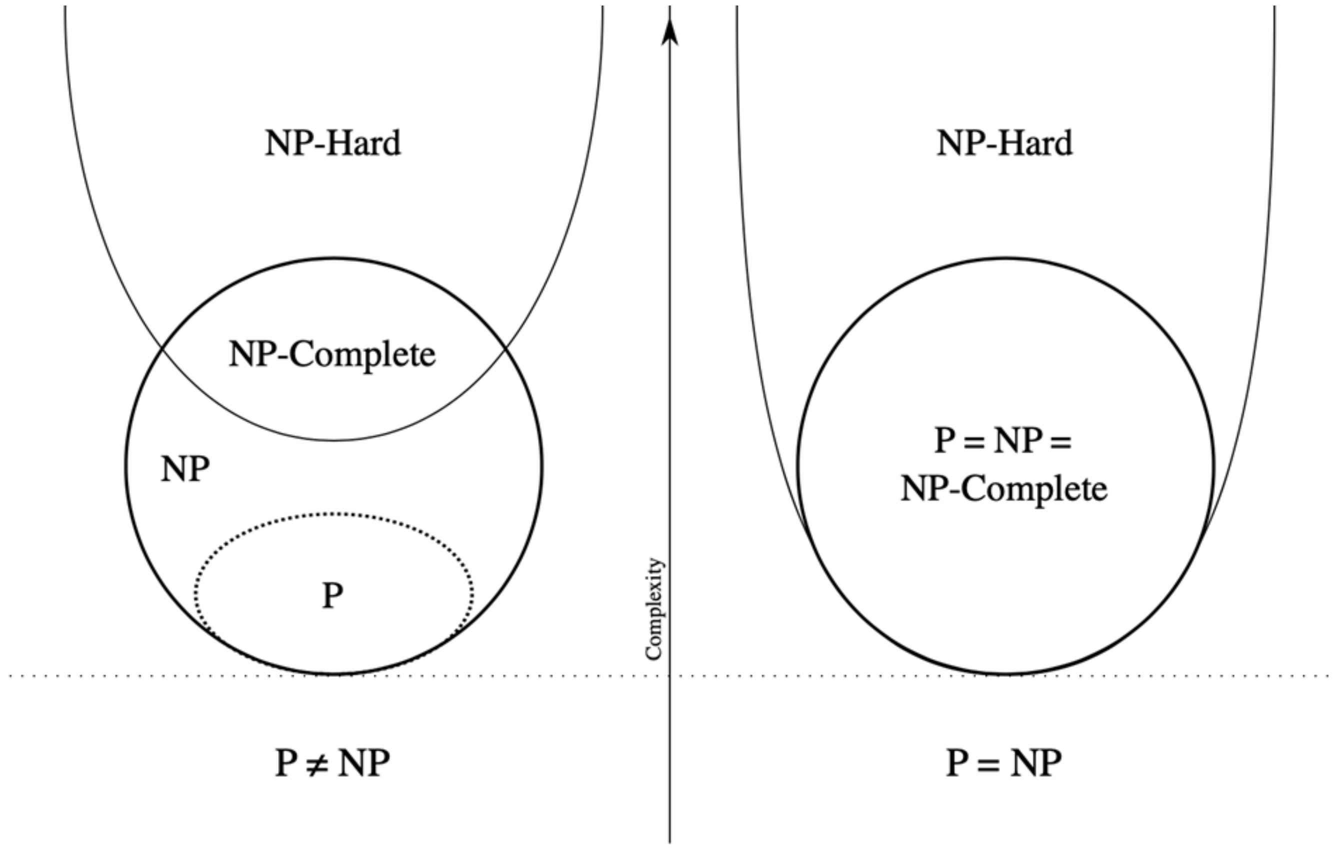
    - 自底向上构建
- 霍夫曼编码在满足前缀码的条件上，找出码长x频率之和最小的树
- 过程：
    - O(ClogC)

```
//C是character的数量
创建最小堆，把所有结点插入
for(i=1;i<C;i++){
        创造一个new的node
        从最小堆中delete出两个元素a和b
        将a和b分别作为node的左右孩子
        更新node的weight是左右孩子的和
        重新把node插入到最小堆里面
}
```

- lemma:
    - 拥有最小频率的两个character x和y，总有一个optimal solution使得x和y的编码拥有相同的长度（即这样的方案的cost总是最优的，是一个Lower bound）
    - 用结点z来代替两个拥有最小频率的节点x和y，生成的solution依旧可以是optimal的(可以prove by contracdition)

NP

概述



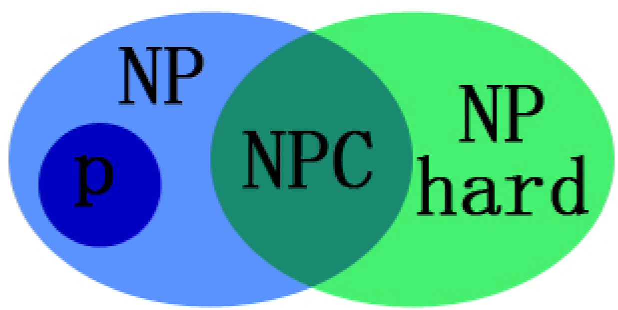


图1 P NP NPC NPhard关系的图形表示

图灵机

- 确定型： 一步一步来，是确定的算法
- 非确定型： 下一步总是选择最优的，是聪明的机器

非确定型图灵机的运行过程像一棵**计算树**，如果在一定时间内可以选择出**最优的一步**，那么问题就可以被非确定型图灵机解决

比如NP问题，NP问题的解是可以在**多项式时间**内被**确定型图灵机判定**的，所以非确定型图灵机可以选择出一个聪明的方案来**解决**NP问题

Decision problem

- NP-hard中有判定型问题和非判定型问题，而其中的判定型问题中有decidable也有undeciadable（如halting problem)的。但是否所有undeciadable问题都属于NP-hard，暂时还没有找到资料

P：

- 多项式时间内可解的**具体判定问题**的集合
- $P = \{L \subseteq \{0,1\}^* : 存在一个算法A，可以在多项式时间内判定L\}$
- 语言类定义： $P = \{L : L能被一个多项式时间算法所接受\}$

NP： nondeterministic polynomial time

- 例子： 哈密顿回路

- 可以在**多项式时间**内**被非确定型图灵机解决**

- 可以被一个多项式时间算法**验证**的语言类

- 我们能在多项式的时间内**验证**一个解的正确性，可是我们却**不知道**该问题是否存在一个多项式时间的算法，每次都能解决他

NPC

- 例子：

  - 哈密顿回路问题
  - TSP
  - Satisfiablity problem(circuit-SAT， 即给定逻辑表达式，是否可能为真)
  - cique problem（分团问题）
  - 顶点覆盖问题

- L是NPC则：

  - $L \in NP$
  - 对于每一个$L' \in NP_2$，有$L' \leq_p L$（即任意NP问题可以在多项式时间内转化为NPC问题）

NP-hard

- 满足NPC的第二条，不一定满足第一条
- 是一个**至少**和**任意NPC**问题一样难的问题
- 例：停机问题

NP=P?

- 是否所有能在多项式时间内验证得出正确解的问题，都具有多项式时间算法？
- 为了验证这个问题，将NPC问题引入
- 能够在**多项式时间内**解决NPC问题中的**一个**，就意味着所有NP问题都可以在**多项式时间**内解决，也即NP=P （NP问题-->多项式时间内转化为NPC问题-->多项式时间内解决-->问题NPC的答案等于NP的答案）

概念

多项式时间

- 多项式时间算法往往可以找到复杂度更低的算法
- 模型a用多项式时间解决，模型b往往也能
- 算术运算中多项式封闭

因此，多项式时间是一个很重要的分水岭

抽象问题

实例-抽象问题关系-解集合（类比ER图）

```
eg:

i=（Graph，vertexu，vertexv ，k(path length))

relation=Path(problem)

Path(i)=1/0  (exist or not exist)
```

具体问题

- 被编码后的问题是具体问题（因为可以被计算机读懂）

- 具体问题的算法复杂度极度依赖于输入，我们的目标是解决抽象问题的多项式时间判定，那么对于每一个具体问题，我们都可以知道其算法复杂度

编码

- 处理问题时我们将**抽象对象**转换成计算机能懂的**编码**

- 为解决抽象问题，我们往往将其分割为多个具体问题，解决了所有的具体问题也就解决了抽象问题

- 也即，将抽象问题映射到具体问题上，e(i)的解一定也是Q(i)的解

- `e(i)` 具体问题i的编码
- `Q` 抽象判定问题
- `e(Q)` 具体判定问题

多项式相关

- 多项式时间可计算的**函数**： 存在一个算法，使得任意输入x，都可以在多项式时间内算出f(x)
- 多项式相关的**编码**： 两个编码可以通过一个多项式时间的算法根据函数f求出
- 引理： 两个多项式相关的编码解决抽象判定问题得到的解e1、e2，如果其中一个属于P，则另一个也属于P
- 选取编码的种类对抽象问题实例的复杂性没有影响，但对实例有影响
- 一般用二进制

形式语言体系

- 连结(concatenation)，闭包
- 接受、拒绝
- 判定：结果只有0和1
- 语言L： 有多种，是字母表的闭包的子集

可归约性

- $L_1 \leq_p L_2$: 如果L1可以通过一个**多项式时间**内可以计算的**函数(reduction function)**转化为L2，则称L1可以在**多项式时间**内归约为L2
- 产生f的多项式时间内的**算法**称为reduction algorithm
- 如果L2属于P，那么L1也属于P

例子

哈密顿回路与TSP（由NPC问题证明NPC）

- 已知哈密顿回路是NPC
- 哈密顿回路：给定图，是否有circle可以走遍所有顶点？
- TSP： 给定**完全图**G，常数k，是否存在一个**简单回路**使得其经过**所有顶点**且总权重**小于等于k**
- 证明TSP是NPC：

- 显然TSP是NP
- 证明哈密顿回路问题可以在多项式时间内归约为TSP：
  - 先画一个哈密顿回路的图
  - 添加有权重的边使其成为完全图（TSP中的图），这个过程使用$O(N^2)$（添加的最多边数）将每一个哈密顿回路问题都映射成为一个TSP问题
  - 有哈密顿回路=TSP问题中有权重为V的答案

Cique和vertex cover问题（由NPC问题证明NPC问题）

- Cique Problem: 给定一个无向图G和整数K，是否存在一个**完全子图**至少含有K个顶点？
- Vertex coverproblem：给定无向图G和整数K，是都存在一个**顶点的集合**，使得顶点数**至多为K**且G中的每条边都有一个顶点在该集合中？（一个大小为k的团支配了整个图）
- 证明：
  - 首先证明顶点覆盖问题为NP。验证算法：检查给定的certificate V', 检查大小是否为k，检查每条边是否都包含其中的顶点。边数为$O(N^2)$,最大顶点数为N，所以验证的算法是$O(N^3)$（多项式时间），所以该问题是NP问题
  - 证明分团问题不难于顶点覆盖问题。
    - G的补图：包含G的所有顶点和相补的边
    - 映射：证明G有clique of size K当且仅当补图有|V|-K规模的顶点覆盖。设总顶点集合为V
      - =>如果G有clique,设clique为V'
        - 使(u,v)是补图中的**任意**一条边，则u和v至少有一个是不属于V'的（因为如果都属于V'，那么一定属于clique，也就一定属于G，不会存在于补图中），因此**u和v中至少有一个属于V-V'**，也即证明了补图有规模为|V|-K的顶点覆盖
      - <= 如果补图有顶点覆盖，设顶点覆盖的集合为V'
        - 对任意u，v，如果u和v都不属于V'，那么(u,v)一定在G里（否则就会顶点覆盖）
        - 由上，因为V-V'不属于V'，则一定在原图G里，且顶点集合中每条边(u,v)都在G里， 所以V-V'一定是个clique

Aproximation

常用来解决NPH问题
证明通常用反证法

Bin Packing

- 给定N个item，求使用的bin的最少数量 NP-hard
- 给定N个item，可以用K个bin全部放入吗 NPC

Next-fit：读一个放一个，不行就再开一个 2M-1
First-Fit:从第一个开始扫，找到合适的箱子 （O(NlogN)
Best-Fit:找合适的箱子使得塞进去后容量最少（1.7）

在线算法：没有办法修改之前得到的结果
There are inputs that force any on-line bin-packing algorithm to use at least 5/3 the optimal number of bins.

- off-line algorithm
  先排序，再用first-fit或 bestfit 得到近似比是11M/9+6/9

背包问题(Knapsack problem)

给定背包容量M，N个物品，每个物品有对应的重量和profit，求最大profit

- 简单版本：每个item可以选择一定比例装入，选择density最大的装即可（最多只会有一个物品被切）
- 01背包（**NPH**）：要么塞要么不塞
  - 如果全塞最大profit的，有可能他们很重，导致错过最优解
  - 如果全塞最轻的，有可能他们profit很少，导致错过最优解
- greedy近似比是2
- DP：
  - Wip:从1到i选出来的价值为p的物品所拥有的最小重量
  - 复杂度:$O(n^2 * P_{max})$(P是最大的价值)
  - 如果P很大，可以同除一个倍数，但这样得到的算法不一定是正确的。近似得到的结果满足约束：$(1+e)P_{alg} \leq P$

K-center problem

- **NPH**
- 给定一堆宿舍，K个食堂，求K个食堂的分布使得食堂半径最小（半径是食堂到一个宿舍的最远距离，选择所有食堂中的最大半径）
- greedy:
  - 减少候选集：把食堂建在宿舍上。如果我们已经知道最优解r(C*)<=r（一个常数），那么约束r至少为2r(C*)，因为两个宿舍的距离不会超过2r(C*)，**近似比就是2**
  - 算法：选择一个宿舍为中心，删除离该中心距离小于等于2r的居民点（因为直径内，一定会覆盖到），直到所有宿舍都被选中或都被删除
  - 证明算法的合理性：如果有一个大小小于等于K的C*且r(C*)<=r，该算法会选出小于等于K个中心
  - 怎么去逼近r(C*)呢？用上面的算法，我们可以知道，如果用自己定义的r，该算法可以返回等于k个结果，那么最优半径一定会小于r/2；如果不可以，那么证明r太小了。于是可以不断逼近logr次，得到一个比较精确的结果

- Be far away
  - 上面的算法已经很好了，但是要在外层做二分
  - 采用“选最远”办法，第一次随便选一个宿舍为食堂，然后选择离该宿舍最远的一个宿舍把它当做食堂
  - **近似比为2**

除非P=NP，对于这个问题我们得不到近似比小于2的解

- 用Dominating-Set做证明：
- 支配集问题：在一些点中选择顶点集合，使得剩余点的邻居至少有一个在选中的顶点集合中（即被支配）
- 支配集问题(NPC)可以归约为K-center，支配集问题的解决方案为K=K-center问题最优半径1，如果K-center可以有一个多项式时间的（2-e）近似算法，即半径小于2，由于支配集问题的半径都是1，所以最后得到多项式近似算法可以得到支配集问题最优解，即NPC问题在多项式时间内被解决，p=NP

Local Search

- 寻找近似结果的一种方法

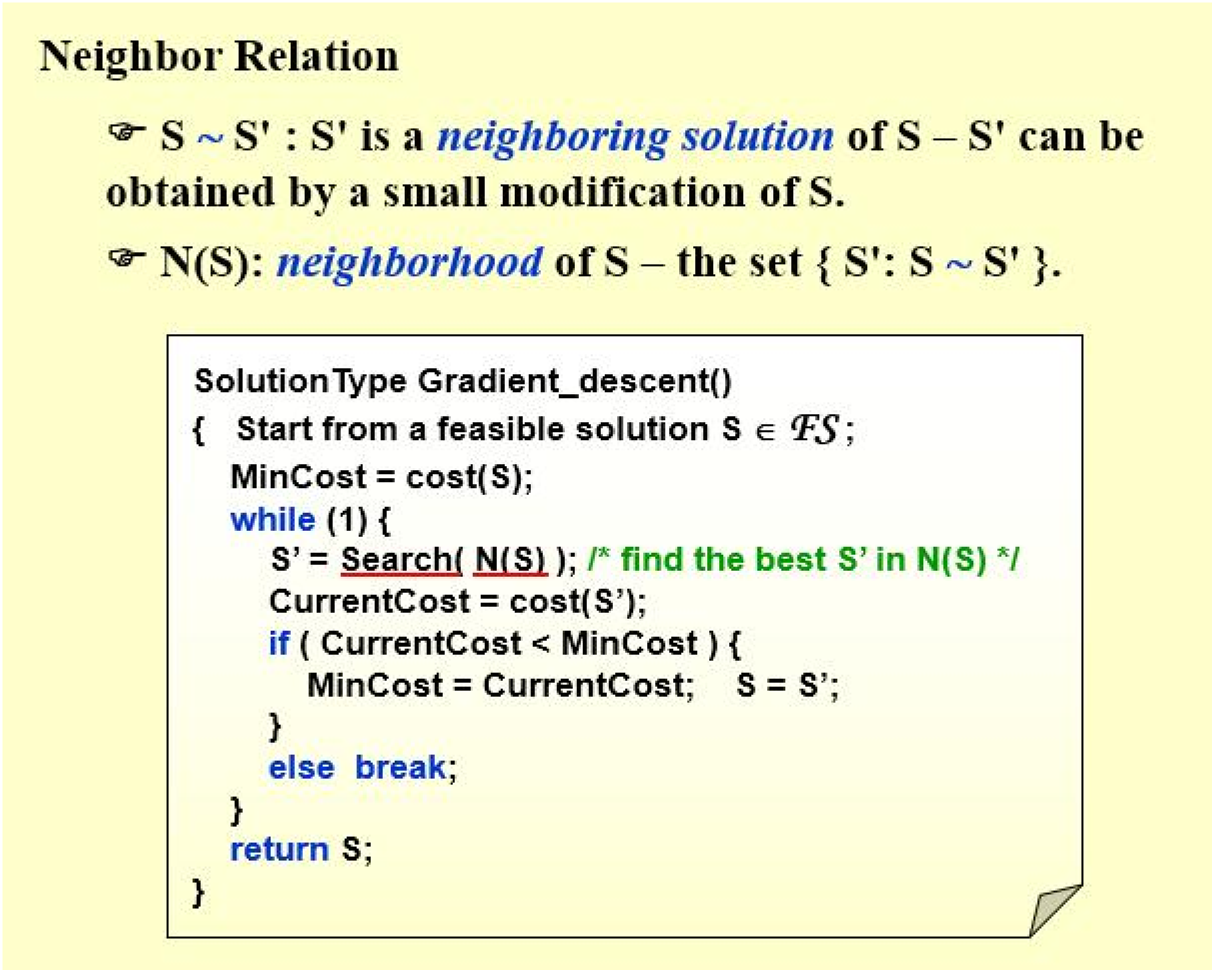
  - 猜测一个起点
  - 找到邻居集合，选择最优解，将当前点更新为邻居
  - 不断寻找最优解，直到没有最优解出现

- 被trap的两种可能：

  - 步长太大，导致来回横跳
  - 一开始就把最优解给排除了

- 关键在于：
  起点是什么
  怎么通过很小的改动获得邻居集合

* 最优解怎么判断



Vertex-cover

- 选择所有结点为起点，删除一个结点，看剩下的结点能不能覆盖所有边
- 每删一个点，和它在同一条边的neighbour不能被删
- Metropolis Algorithm
  - 陷入入局部最优解的原因：没有后悔的机会
  - 这个算法允许后悔，在选择的时候，随机选取一个邻居；如果当前的cost小，就update；但当cost大的时候，也有一定几率会被选中（引入概率）
- Simulated Annealing: 模拟退火

Hopfield Neural Networks

- 图的边有权重w，w<0则两端顶点状态想要趋向相同，否则趋向相反
- |w|反映了趋向的程度
- good edge: $ws_us_v < 0$
- u is satisfied: u的所有ws_us_v之和小于等于0
- configuration is stable: 所有顶点都满足
- ***State-flipping Algorithm**
  - 当S不是稳定状态，找到一个不符合要求的顶点，翻转它的值

- 一个顶点有可能会被翻转多于1次
- 每个local的最佳值都是stable配置（也就是不用找到全局最优，找到局部最优解就OK）
- 最多进行W次迭代（边权和）
- 时间复杂度不一定是polynomial，W不确定

> - 等式中的wgood和wbad是翻转后的
> - S是整个图
> - 最坏情况是一开始所有都是坏边
> - 所有good edge的weight之和大于等于0（如果一开始都是坏边），因为每次都至少增加1，经过W次后，可以将所有边都变成好边

Claim: The state-flipping algorithm terminates at a stable configuration after *at most* $W = \Sigma_e |w_e|$ iterations.

Proof: Consider the measure of progress

$$\Phi(S) = \Sigma_{e \text{ is good}} |w_e|$$

When u flips state (S becomes S'):

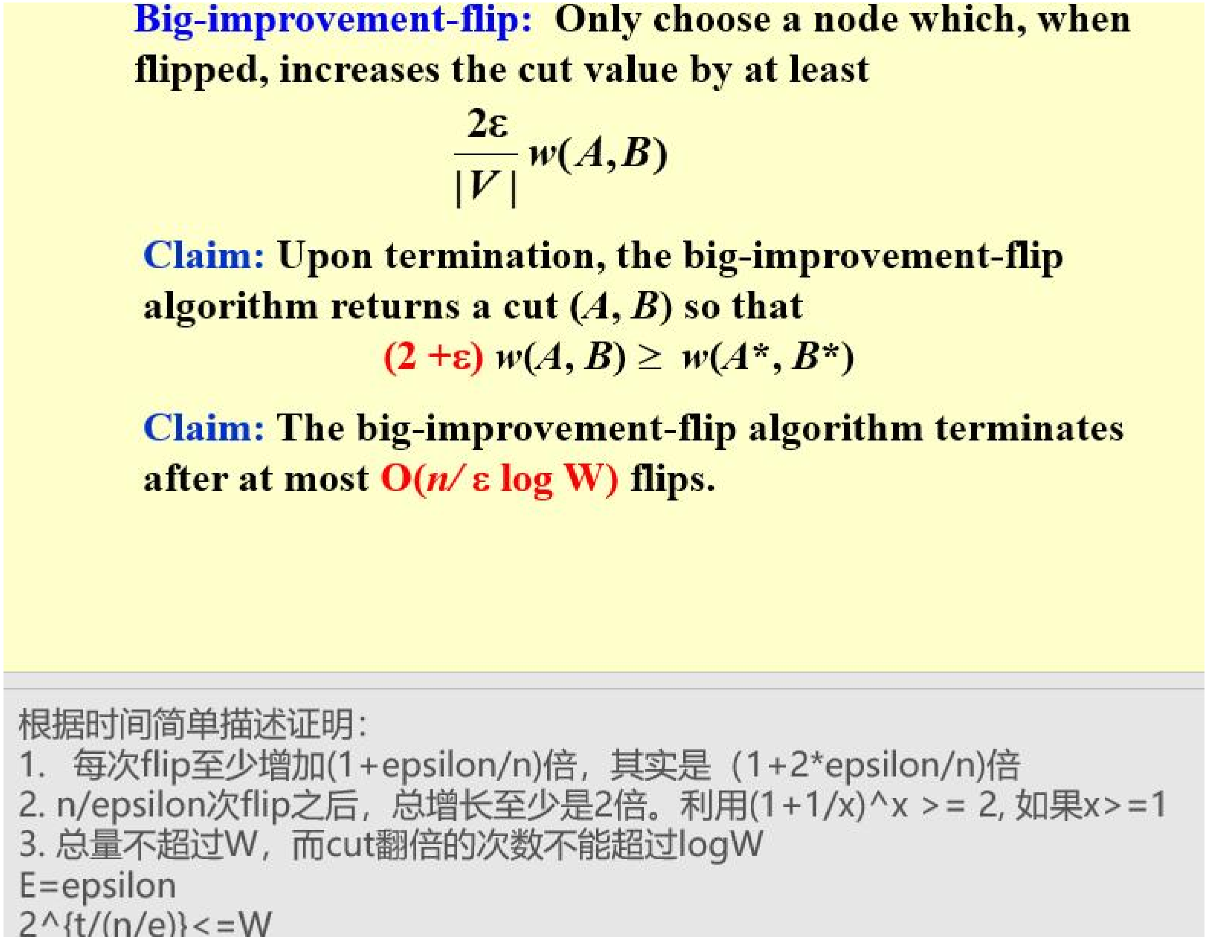
- all good edges incident to u become bad
- all bad edges incident to u become good
- all other edges remain the same

$$\Phi(S') = \Phi(S) - \sum_{\substack{e:\, e=(u,v)\in E \\ e \text{ is bad}}} |w_e| + \sum_{\substack{e:\, e=(u,v)\in E \\ e \text{ is good}}} |w_e| \ \geq \ \Phi(S) + 1$$

Clearly $0 \leq \Phi(S) \leq W$ ∎

- **Maximum Cut Problem**
  - 给定有权无向图，将顶点分成两个集合，求具体的划分使得两个划分之间的edge权重加起来最大
  - 是hopfield neural network的一个特殊情况，边权都是正数；现在的目标是要使好边的边权加起来最大
  - 解释看PPT
  - $w(A, B) \geq 1/2w(A*, B*)$ (local最优, global最优)
  - 现有解逼近1
  - 除非P=NP 不存在17/16的近似比

- big-improvement-flip
    - 只有flip之后权重可以增加一定值的才flip，当新的局部最优解的增长的幅度小于设定值的时候就停止，为了让算法可以在多项式时间内结束



算法问题

- 证明常常找出最优解的一个下界

顶点覆盖问题

```
选定一个u，如果它的边对面有顶点(v1,v2..)，删除u和对应的边
标记v1,v2..为不可删除的点，重复操作，直到没有可以删的点
```

- 近似算法1（ratio=2）

```
while(边的集合不为空){}
       在图中随便选一条边(u,v)
       将u，v选入顶点覆盖集合
```

```
        边的集合中删除所有和u，v关联的边
}
```

- local search(Metropolis algorithm)
  - 陷入局部最优解的原因：没有后悔的机会。这个算法允许后悔，在选择的时候，随机选取一个邻居；如果当前的cost小，就update；但当cost大的时候，也有一定几率会被选中

TSP

- 如果TSP不符合三角不等式，则不存在具有固定近似比的多项式时间近似算法，除非P=NP
- 满足三角不等式的旅行商问题：
  - 生成一棵最小生成树并进行先序遍历，ratio=2
  - 证明：
    - c(最优解)>=c(最小生成树)
    - 如果进行完全遍历，c(W)=2c(最小生成树)
    - c（W）<=2c（最小生成树）

Randomized Algorithms

- 随机算法： 通过随机选择算法的每一步，通常假设每个选择概率相同，来避免最坏情况的发生，有两种
  - 高效，但只产生大概率的结果
  - 结果总是正确的，但不高效

Hiring Problem

- 雇佣人，其中雇佣和面试都需要cost，雇佣的cost远大于面试cost，现在要找到一个较好的面试、雇佣方案来尽量雇佣好的人并减少消耗
- C_i: interview cost， C_h: hire cost
- $T = O(NC_i + MC_h)$, N为面试的人数，M为雇佣人数

naive solution

- 线性Interview，一旦当前的人是最优的，就雇佣他
- 最坏情况：人按递增序列进行interview $O(NC_h)$

Radomized Algorithm

- 随机生成随机数序列赋给应聘者，并将其排序
  - 随机数相同的概率应尽量小，可以通过对N^3取模来实现
- 这样分析时，M可以通过数学期望算出来，1/i级数上界得到logN
- $T = O(C_h lnN + NC_i)$

- 好处：不再需要分析应聘者前来应聘的顺序
- 坏处：产生随机序列需要时间

Online Hiring Algorithm

- 设置一个"残忍系数"k，对前k个人得到最优值，**但不雇佣**，相当于提供了一个基准；在k+1到N个人的时候，一旦找到一个比最优值好的人就立刻雇佣并停止算法
- 如果最优的出现在前K个，返回最优值是最后一个人
- 问题在于：给定k，我们雇佣最优人的概率是多少？k要怎么选？
- 我们能雇佣最优人的概率P_s：计算第i个人是最优的概率，并将k+1到N的概率加起来
- 根据微积分可以得到P_s上下界，对下界求偏导可以确定最优k
- 1/e的概率，k=N/e

Quicksort

- 详见pdf
- 普通快排：
  - 最坏：$\theta(N^2)$
  - 平均：$\theta(NlogN)$
- 随机快排(随机选择pivot)：
  - 最坏 $O(NlogN)$

Parallel Algorithm

Parallel Random Access Machine (PRAM)

- 有一个shared memory
- 读写会产生冲突，要有规范
  - EREW （exclusive)
  - CREW（concurret read exclusive write)
  - CRCW(写冲突依旧在，有不同的role)
    - Abitrary rule: 随机选一个处理器让它写入
    - Priority Rule：只允许最小标号的处理器写
    - Common rule：只有当所有处理器都写同一个值的时候才允许写
- 尽管部分处理器在执行过程中不需要工作，但还是给了idle指令，导致最后其实每个处理器的workload是一样的
- 缺点
  - 不能从这个模型中看到处理器数目对算法的影响
  - 这个模型需要对处理器分配有详细的描述，还不够抽象

Work-Depth (WD)

- 在每一层都pardo，相比于PRAM节省了idlle的资源

Measuring the performance

- work load: W(n) (规模为n时的workload)
- worst -case running time: T(n)
- 使用的几种衡量方式都是渐进相同的
- 例子
  * prefix sums
  logn
  W(n)=n

Merge（有序的两个序列）

知道了每个rank之后，并行算法就可以用O(1)时间搞定merge，并且用O（n+m)的work load，所以重点在于：怎么计算rank？
T(n)=(logn)
W(n)=n

- Rank
  - 二分法：logn,nlogn
  - 串行：n+m
- ok为了加快rank，我们继续分治
  - 从A、B里每隔logn就选出一个元素，选出n/logn个
  - 为这些选择的元素计算rank，logn，workload=plogn=n
- 上面选出来之后，我们实际上把问题进行了划分，划分成了n/logn组，每份大小是logn
- 对于每个子问题，我们只要计算出它们的rank就好了。因为每份大小是logn，时间就是logn，因为一共最多试2n/logn组，workload就是n

Maximum Finding

- balanced tree的方法： O(logn) W(n)=O(n)
- 全部都比较一遍，如果有比它大的就记录为1，到最后选出为0的即可 O(1) W=O(n^2)
- doubly-logarithmic 切成根号n的大小，得到根号n个最大值(线性比较）再用前面全部比较一遍的办法
  $O(loglogN)$ $O(nloglogn)$
- 一份切成loglogn的大小： O（loglogn） O(n)
- random sampling: 很大概率会得到O(1) O(n) 失败概率是O(1/n^c)

External sorting

- 减少pass数
  - k way，增加了磁盘时间，需要2k个磁带
  - $1 + \lceil logN/M \rceil$次pass
- 减少磁带数
  - 使用斐波那契数，则可以用3条磁带进行2-way merge
  - 如果不是斐波那契数，可以增加一些虚拟的run来凑
  - k way使用k条磁带
    - 斐波那契数变为：初始化前k-1个是0，第k个是1，Fn等于前k个数加起来
    - 取最小那段的数量输出到空带，剩下的放在原位
- buffer hadling
  - 并行，k way给2k个输入缓冲区，2个输出缓冲区（要把完整的block切分，因为内存大小是固定的） 因为Block大小变小，磁盘IO增加
- 使用更长的run
  - heap sort，可以产生平均2M大小的run（Replacement selection）
- 减少merge time:
  - 霍夫曼编码，等于各个internal node的weight加起来