

ADS Summary

JerryG

(ADS 课程的一些数据结构、算法的重点总结和个人理解，不一定对)

数据结构部分：

一、 AVL 树, 伸展树 (含均摊分析) AVL tree, splay tree

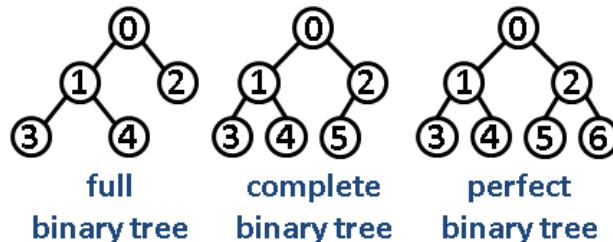
- 背景：搜索树是一种经常用于搜索/索引的数据结构，可以根据左右节点判断数值大小关系，将查询复杂度降到 \log 级别。其中以二叉搜索树 (BST) 为典型。对于普通的二叉树，其生成的结构与输入顺序有关，对于不好的输入，比如单调输入，会导致二叉树非常瘦高，不利于搜索。所以无论是 AVL 树、伸展树还是后续的红黑树，目的都是基于建立某种二叉树的构建规则，使得其尽量“平衡”。只不过构建策略有所区别，付出的代价、得到的收益也有所不同。

二叉树有如下修饰词，千万注意区分：

Perfect：“完美二叉树”：最理想的二叉树，即每个节点的左右子树的树高都相同（注意只有根节点的树高定义为 0，空树的树高定义为 -1，注意这个定义也可能是 1、0 要看具体说明）。但是可惜完美二叉树条件非常严格，需要节点数必须是 $2^{h+1} - 1$ ，并不是所有树都一定有这个节点数。所以我们的数据结构的目的是为了尽可能接近平衡。完美二叉树有一个完美平衡的性质：每个叶节点到根节点的路径长度都相同。

Complete：“完全二叉树”：指的是除去最后一层，上面是平衡二叉树，而最后一层从做往右添节点，可以没添满的树。这种树也非常平衡，但一般用于堆结构中，而非做搜索树用。

Full/Strictly：“完满二叉树”：指的是除了叶节点外，每个节点都必须有两个子节点的二叉树。即不可能存在有且仅有一个孩子的节点。如哈夫曼编码树。



Balanced：“平衡二叉树”：即 AVL 树，每个节点的左右子树高的差不超过 1 的树。

2. AVL 树

(1) 定义

【Definition】 An empty binary tree is height balanced. If T is a nonempty binary tree with T_L and T_R as its left and right subtrees, then T is **height balanced** iff

- (1) T_L and T_R are height balanced, and
- (2) $|h_L - h_R| \leq 1$ where h_L and h_R are the heights of T_L and T_R , respectively.

【Definition】 The balance factor $BF(\text{node}) = h_L - h_R$. In an AVL tree, $BF(\text{node}) = -1, 0, \text{ or } 1$.

AVL 树是一种以人名命名的二叉树，又称平衡二叉树，定义非常简单，就是任何一个节点左右子树的高的差不超过 1 的树（定义 BF 为左子树高减右子树高，相当于完美二叉树基础上做了妥协，BF 值从只能为 0 调整为 -1, 0, 1 均可）。

对于任何一种输入，都可以基于一套规则构建出 AVL 树。这套规则也是难点，以旋转操作为核心。

(2) 高度分析

先来分析 AVL 树的高度 h 。注意对于搜索树而言，搜索的最坏复杂度树 $O(h)$ ，所以一般把 h 根节点数 n 的关系作为衡量搜索树好坏的标准。

$$\Rightarrow n_h \approx \frac{1}{\sqrt{5}} \left(\frac{1 + \sqrt{5}}{2} \right)^{h+2} - 1 \quad \Rightarrow \quad h = O(\ln n)$$

可以证明（不难）节点数相对于树高 h 树斐波那契数列级别，因此高度树 $\log n$ 的。这对于任何 AVL 树都是成立的，换句话说只要构建出的树 AVL 树，就一定能保证搜索复杂度树 $\log n$ 级别，不存在像普通二叉树那样的最坏特例。这也是 AVL 树的重大优势。

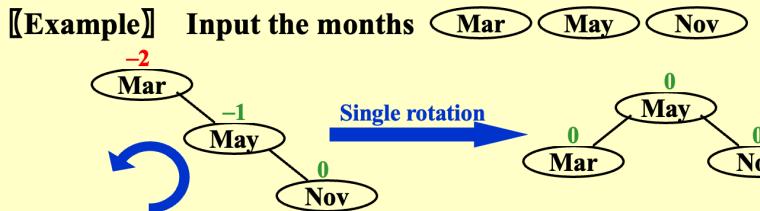
(3) 构建过程，旋转 rotation

AVL 树的构建过程大致是：按普通二叉树的构建方法来做，一旦发现某个操作（插入/删除）会导致不平衡节点（BF 值不满足要求）出现，那么对这个节点及相关节点进行合适的旋转操作，在不影响二叉搜索性质的同时恢复平衡。

插入：在插入一个元素（trouble maker）后（按普通二叉树的方法），需要向上更新路径上节点的 BF 值，如果发现某个节点不平衡（trouble finder），需要进行旋转操作（且只需一次）。旋转的方式由 maker 和 finder 的相对位置关系决定 (key)。

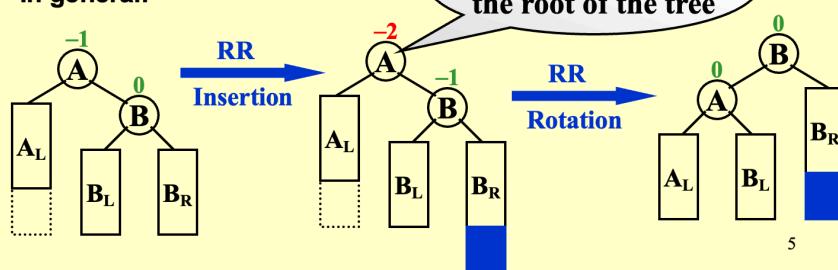
<1> maker 是 finder 的孩子：这必不可能，因为这样说明 finder 原先无左或右孩子，不可能因为多了这个孩子导致不平衡（很容易想明白）。maker 一定有且父节点一定 finder 的后代，此时我们关心这三者在树中的相对位置关系。

<2> maker 和 finder 在 maker 的父节点的异侧，即 maker 的父节点在 finder 右子树中且 maker 是它的右孩子（或者换成左同理）。此时我们进行单旋。



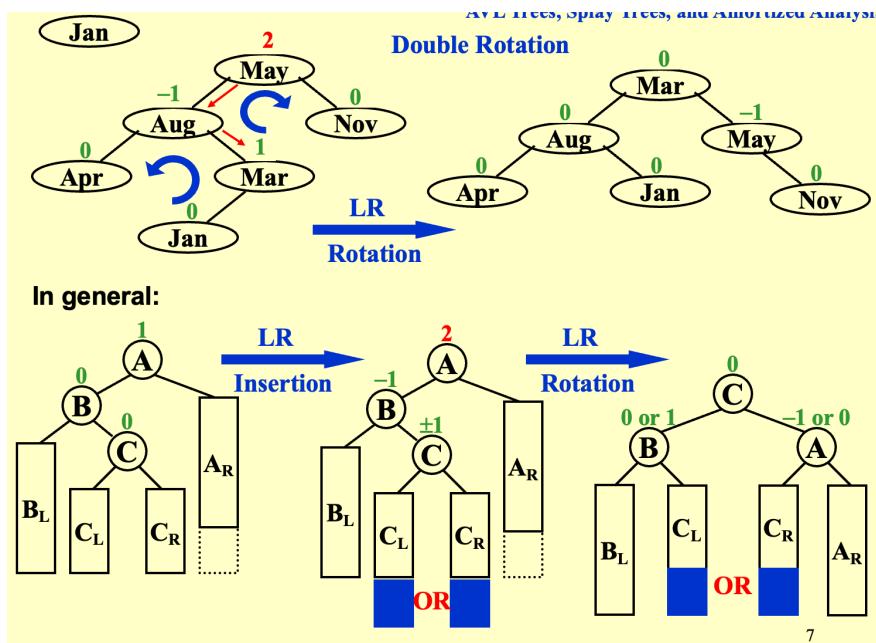
⦿ The trouble maker Nov is in the right subtree's right subtree of the trouble finder Mar. Hence it is called an RR rotation.

In general:



以右-右为例, A 是 finder, maker 在 BR 中, 注意 maker 的父节点可能是 B (此时 BR 只有 maker 一个节点), 也可能在 BR 中。按如图的方式单旋一次即可恢复平衡。

<3>maker 和 finder 在 maker 的父节点的同侧, 即 maker 的父节点在 finder 左子树中且 maker 是它的右孩子 (或左右交换同理)。此时我们进行双旋。



以左-右为例, 注意此时 maker 的父节点可能是 B (maker 是 C), 可能是 C, 也可能在 CL 或 CR 中。

技巧: 注意在实际操作中其实不必记这些旋转的具体规则, 只需要知道 finder、maker、maker father 这三个点及其位置关系, 然后确定单旋或双旋, 知道把哪个节点放在顶上即可, 其余的子树分配可以看情况做, 容易。

再次指出, AVL 的插入旋转操作最多一次 (如果把双旋转看作一个整体), 修正完一个 finder 后, 上面不会再受影响。

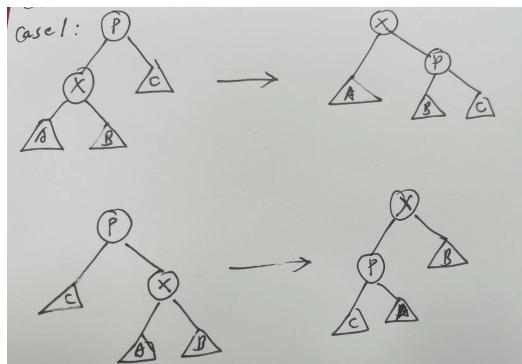
删除 : AVL 树的删除极为复杂, 分为多种子情况。且可能导致级联操作, 最坏情况下影响整条路径, 旋转复杂度为 $\log n$ 级别。实际中经常采取标记删除而非真正删除的策略。AVL 树的删除一般不会考。另外, 后面可以看到红黑树的删除效率更高。

3. 伸展树

(1)思想 : 伸展树 (splay tree) 的思想与 AVL 很不同 : 伸展树在访问 (find) 的过程中调整树的结构使之更平衡, 从而使得连续的访问平均消耗较低。基本思路是将访问的节点通过旋转操作转移到根节点上。

(2)访问/查找: 不能单纯只把访问节点 X 放到根节点上, 应该同时进行一些平衡化的操作。具体的旋转方法是 : 对于访问节点 X, 时刻追踪它, 当它不是跟节点时, 如图分为三种情况 :

第一种, X 的父节点是根节点, 此时只需简单旋转即可。(AVL 中没有这种操作)



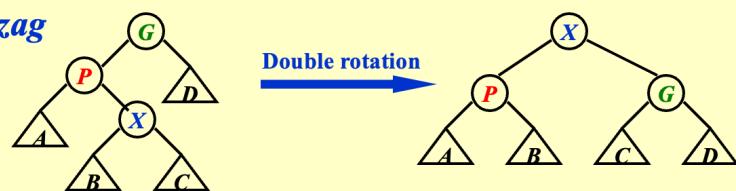
第二种和第三种如图, 分别是 zig-zag 和 zig-zig, 分别对应 AVL 操作中的双旋、单旋转操作。

Try again -- For any nonroot node X , denote its parent by P and grandparent by G :

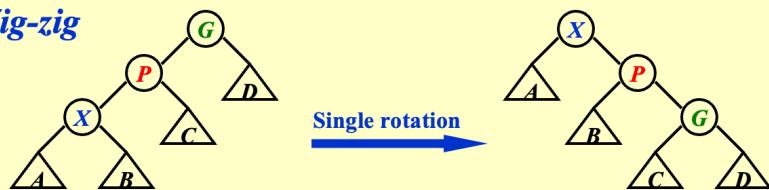
Case 1: P is the root \rightarrow Rotate X and P

Case 2: P is not the root

Zig-zag



Zig-zig



另外注意这是一个循环的过程, 每次循环完层后都要审视 X 现在是不是根节点, 如果不是, 还需要继续向上操作。所以一次访问可能有数量级为 h 的旋转。

(3)插入 : 伸展树的插入分为两部分, 先按 BST 的规则插入到正确的位置, 然后访问被插入的节点, 即把它升到根节点上来。这说明插入到节点一定在根节点上。

(4) 删除：伸展树的删除与普通 BST 并不相同。其分为四步：

Deletions:

- ☞ Step 1: Find X ; X will be at the root.
- ☞ Step 2: Remove X ; There will be two subtrees T_L and T_R .
- ☞ Step 3: FindMax (T_L) ; The largest element will be the root of T_L , and has no right child.
- ☞ Step 4: Make T_R the right child of the root of T_L .

先找到待删除的元素，注意查询这个过程已经完成了一次“伸展”，待删除节点现在在根节点，然后删掉，合并左右两个子树，合并过程如图，注意先在一棵树中 findmax，即一路向右查找，最后最大的元素在这棵树的根节点处，然后把另一棵树作为这个根节点的子树即可。可以看出，一次删除过程中，出现了两次“伸展”。

(5) 复杂度分析

均摊复杂度是 $\log N$ ，即对任意 M 次操作（查/插/删）的时间复杂度之和是 $M \log N$ 数量级。

4. 均摊分析(Amortized Analysis)



Target: Any M consecutive operations take at most $O(M \log N)$ time.

-- *Amortized time bound*

worst-case bound \geq amortized bound \geq average-case bound

Probability is *not involved*

☞ Aggregate analysis

☞ Accounting method

☞ Potential method

(1) 介绍

伸展树这种数据结构有一个特点：任何一个操作(operation)，都会改变自身的结构。对于一个单次的操作，复杂度可能比较高，但是连续的操作的时间代价均摊在每个操作上是较好的。对于均摊时间复杂度分析，常用的有三种方法。这一部分较难，常考理论题。

(2) Aggregate analysis 累计分析

计算 m 次操作的总耗费时间，然后算平均值即可。使用范围不广，因为针对具体情况可能很难求耗费时间。

(3) Accounting method 会计分析/账目分析？

在假设了一个要证明的均摊复杂度后，对实际的每步操作，如果实际消耗比均摊消耗低，那么把差值作为“余额”存起来，余额可以为其他操作中实际消耗比均

摊消耗高出的那部分“付款”。最后只需要证明每一步的均摊消耗之和大于等于实际消耗之和。

(4) Potential method 势能分析

 **Potential method**

Idea: Take a closer look at the *credit* --

$$\hat{c}_i - c_i = Credit_i = \Phi(D_i) - \Phi(D_{i-1})$$

$$\sum_{i=1}^n \hat{c}_i = \sum_{i=1}^n (c_i + \Phi(D_i) - \Phi(D_{i-1}))$$

$$= \left(\sum_{i=1}^n c_i \right) + \underline{\Phi(D_n) - \Phi(D_0)} \geq 0$$

Potential function

In general, a good potential function should always assume its minimum at the start of the sequence.

对账目分析法的一种形式上的改进。定义数据结构自身状态的“势能”函数，每次操作可能改变自身的势能，势能的改变跟余额的改变（每一步均摊和实际的差值）有密切关系，或者说相等。**势能最开始取最小值，每一步中，势能增加量等于均摊消耗比实际消耗多出的部分**。定义完势能函数后，均摊复杂度一定是大于势能函数每一步的最大可能的增长值的，所以要定义增长尽量缓慢的势能函数，同时又能表现实际操作和均摊操作的关系（这里不太容易说清楚）。势能函数的最大值往往决定了均摊分析的结果。

势能函数的定义是均摊分析中最为有技巧性也是最难的部分，比如在伸展树问

题中： $\Phi(T) = \sum_{i \in T} \log S(i)$ where $S(i)$ is the number of descendants of i (i included). 是非常奇怪的定义，证明也较为复杂。

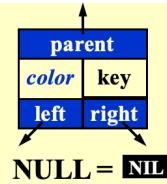
二、 红黑树 red-black tree

- 思想与概念：红黑树与 AVL 树都是为了逼近完美二叉树，但思路完全不同。AVL 树让左右子树的高只能差 1，而红黑树则是尝试直接修改树高的定义，使得树满足完美平衡的性质，即叶节点到根节点的“新树高”都相同。这里的新树高就是指的后面会提到的“黑高”。

Red-Black Trees

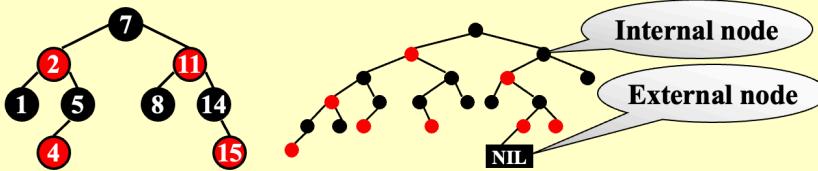


Target: Balanced binary search tree



【Definition】 A red-black tree is a binary search tree that satisfies the following red-black properties:

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



【Definition】 The **black-height** of any node x , denoted by $bh(x)$, is the number of **black nodes** on any simple path from x (x not included) down to a leaf. $bh(Tree) = bh(root)$.

【Lemma】 A red-black tree with N internal nodes has height at most $2\ln(N+1)$.

红黑树要求黑高相等，其实主要思想是：标红尽可能少的节点，让树忽略这些节点后的树高满足完美平衡。其中让红节点尽可能少的主要规定是红色节点的孩子必都是黑色节点，这等价于没有相邻的红色节点。注意红黑树定义叶节点全为 NIL，这纯粹是为了性质(5)的定义方便。性质(5)也是红黑树的最关键性质，这个 black node number 又称为点点黑高。根结点点黑高也是红黑树的黑高。

可以证明树的黑高最大是 $2\log(N+1)$ 。方法是递推证明 $\text{sizeof}(x) \geq 2^{bh(x)} - 1$ 。

2. 插入(复杂)

这里引用 csdn 上一篇文章的讲解

红黑树规定，插入的节点必须是红色的。而且，二叉查找树中新插入的节点都是放在叶子节点上。

关于插入操作的平衡调整，有这样两种特殊情况：

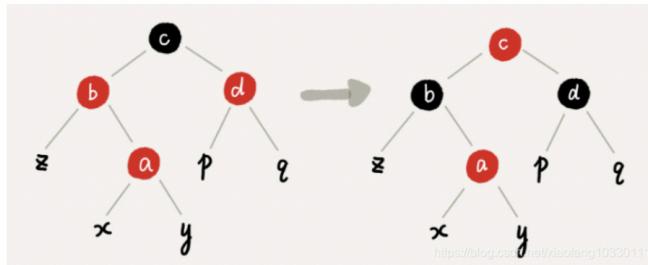
- 如果插入节点的父节点是黑色的，那我们什么都不用做，它仍然满足红黑树的定义。
- 如果插入的节点是根节点，那我们直接改变它的颜色，把它变成黑色就可以了。

除此之外，其他情况都会违背红黑树的定义，需要进行调整，调整的过程包含两种基础的操作：左右旋转和改变颜色。

红黑树的平衡调整过程是一个迭代的过程。把正在处理的节点叫作关注节点。关注节点会随着不停地迭代处理，而不断发生变化。最开始的关注节点就是新插入的节点。新节点插入之后，如果红黑树的平衡被打破，那一般会有下面三种情况：

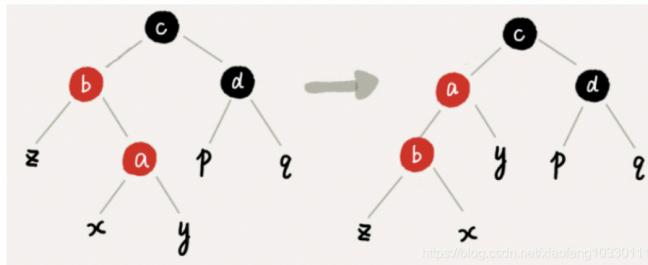
备注：我们只需要根据每种情况的特点，不停地调整，就可以让红黑树继续符合定义，也就是继续保持平衡。为了简化描述，把父节点的兄弟节点叫作叔叔节点，父节点的父节点叫作祖父节点。

情况一：如果关注节点是 a，它的叔叔节点 d 是红色



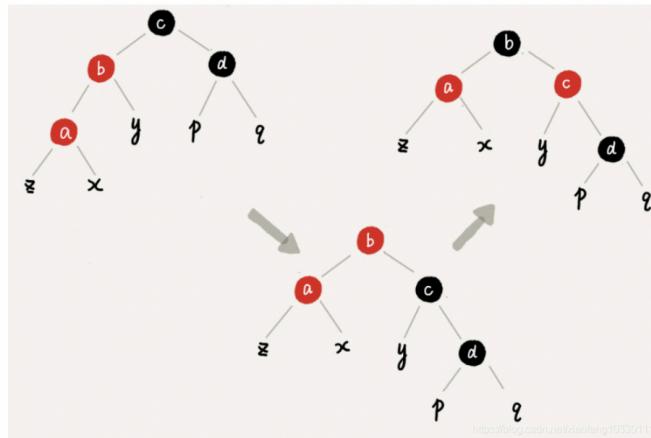
具体操作为：将关注节点 a 的父节点 b、叔叔节点 d 的颜色都设置成黑色；将关注节点 a 的祖父节点 c 的颜色设置成红色；关注节点变成 a 的父节点 c；跳到情况二或者情况三。

情况二：如果关注节点是 a，它的叔叔节点 d 是黑色，关注节点 a 是其父节点 b 的右子节点



具体操作为：关注节点变成节点 a 的父节点 b；围绕新的关注节点b 左旋；跳到情况三。

情况三：如果关注节点是 a，它的叔叔节点 d 是黑色，关注节点 a 是其父节点 b 的左子节点



具体操作为：围绕关注节点 a 的祖父节点 c 右旋；将关注节点 a 的父节点 b、兄弟节点 c 的颜色互换，调整结束。

关注节点是红黑树操作中的重要概念，旋转和染色是两种基本操作。注意虽然说是一个迭代过程，但其实最多迭代两次。

3. 删除

删除更为复杂，涉及到极多的子情况，一般不会考，就不解释了，具体可以在如下网址去看。

https://blog.csdn.net/xiaofeng10330111/article/details/106080394?ops_request_mis_c=%257B%2522request%255Fid%2522%253A%2522165579654616782388059115%2522%252C%2522scm%2522%253A%252220140713.130102334.pc%255Fall.%2522%257D&r

http://equest_id=165579654616782388059115&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-2~all~first_rank_ecpm_v1~hot_rank-1-106080394-null-null.142^v19^control,157^v15^new_3&utm_term=%E7%BA%A2%E9%BB%91%E6%A0%91&utm_spm=1018.2226.3001.4187

只需要知道，红黑树的删除最多经过三次迭代，而不是 AVL 树的 $\log N$ 级别。

Number of rotations	
AVL	Red-Black Tree
Insertion ≤ 2	≤ 2
Deletion $O(\log N)$	≤ 3

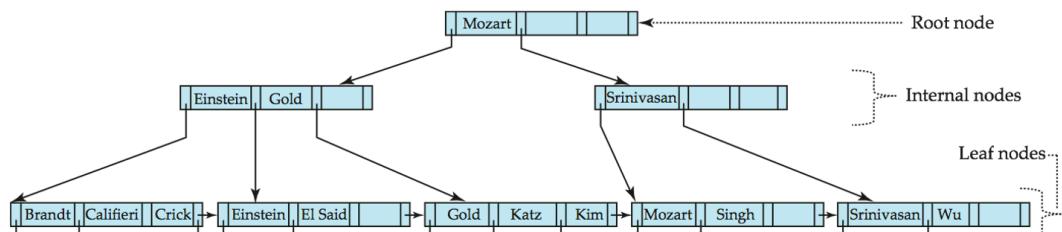
删除的效率是红黑树相比于 AVL 树的优势。虽然红黑树没有 AVL 那么平衡，但高度不会差很多（常数级别），所以查询效率也相近。STL 多采用改进的红黑树结构。

三、B+树 B plus tree

- 概念：B+树是一种平衡性较好的多叉搜索树，查询数据全存储在叶节点中，所有叶节点深度相同。是一种适合磁盘操作的索引存储结构。
具体定义为（没错用的数据库的课件）：

- All paths from root to leaf are of the same length – **balanced tree**
- Each node that is not a root or a leaf has between $\lceil n/2 \rceil$ and n children.
- A leaf node has between $\lceil (n-1)/2 \rceil$ and $n-1$ values
- Special cases:
 - If **the root** is not a leaf, it has at least 2 children.
 - If **the root** is a leaf (that is, there are no other nodes in the tree), it can have between 0 and $(n-1)$ values.

$n=4$ 的 B+树例子：



Typical node

P_1	K_1	P_2	\dots	P_{n-1}	K_{n-1}	P_n
-------	-------	-------	---------	-----------	-----------	-------

- K_i are the search-key values
- P_i are pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes)
- 通常，一个节点对应一个block

The search-keys in a node are ordered

$$K_1 < K_2 < K_3 < \dots < K_{n-1}$$

注意每个节点中，值一定比指针少一个， K_i 的值是 P_{i+1} 指向的子树中的最小值，即子树的叶层最左边的值。另外，叶节点之间有指针直接相连，加速某些操作。

2. 插入与删除，分裂与合并

插入时，若节点已满，则需要分裂（split），分成大小相差不超过 1 的两部分，然后给上层节点也进行改变，可能引发级联分裂。如果最终分裂到了根节点，树高会加 1。

删除类似，若导致节点元素太少，则与旁边相邻的节点进行合并，同样删除上层节点对应元素及指针，可能引发级联合并。如果根节点的子节点可以合并成一个节点，那么根节点会被删除，树高减 1。

此外，B+树在已知全部叶节点的情况下，有自顶向上、线性建树的方法。

具体细节不赘述。

3. B+树的优点与应用场景

B+树的树高是 $O(\lceil \log_{\lceil M/2 \rceil} N \rceil)$

这也是查询的复杂度。但注意虽然 B+树平衡，但它不是二叉树，分叉数有不确定性。另外，B+树的插入和删除都可以有级联情况，这一点不如 AVL 和红黑树。因此对于内存中的小规模数据，往往没必要使用 B+树。

B+树的最强应用场景在于**磁盘索引**，磁盘的场景与内存中的最大区别是，磁盘的随机访问代价非常大，必须减少随机访问、增多连续读写。而 B+树的节点可以很大，适合整块载入内存处理。另外，B+树支持自顶向上快速建树，适合磁盘已有大量数据并在上面建索引的情形。最后，相比于 B 树，B+树虽然可能查询效率稍低一点点，但实现更简单、查询效率更稳定，并且磁盘的随机读写代价更低。

四、 倒排文件索引 inverted file index

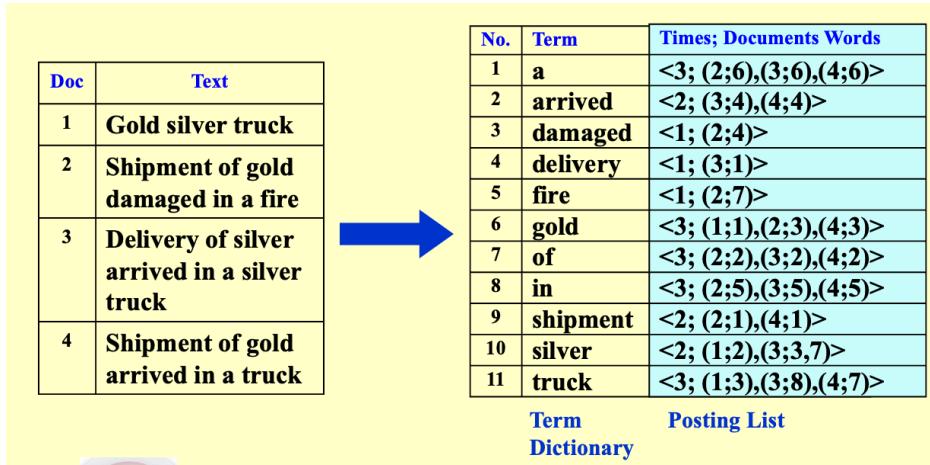
1. 问题背景：之前所提到的树索引，一直都说目标在于“查询、搜索”，但一直没有说白一点：那就是我们查询的目标往往不是键值本身（key，即树索引中用于比较的值），毕竟 key 我们是知道的何来查询一说，而是利用这个 key 去定位与其相关联的更大的结构，比如文件，而这个更大的结构只是提供了一个 key 供我们去建树。所以，我们在节点里同时存储 key 和另一个指针，这个指针指向要查询的真正对象（当然，也可以直接存在节点里）。这是索引查询的基本思想。

有时，一个 key 对应多个查询对象（文件），面临的问题是给一个 key 去找出所有与这个 key 相关的文件（比如包含关系），这就是倒排文件索引的背景。可以说，倒排

索引是树索引的下游延伸。

倒排索引最常见的应用是搜索引擎：给定一个词 (key)，需要返回所有包含这个词的文件 (网页)。正常的思路是扫描每一个文件的每一个词依次判断是否包含 key，这种操作虽然不需要维护，但是消耗过大显然不可行。而倒排索引之所以成为倒排，就是因为它维护了一个数据结构，可以用 key 查询哪些文件包含它，从而将查询速度将为常数级别，当然也要有额外的维护开销。倒排索引是一种常见的以空间换时间的策略，维护元信息以加速后续操作，这是重要思想。

2. 倒排索引的数据结构



如图，倒排索引维护了一个表，每一行是 term (词，也就是 key)，然后对应了一个 posting list，存的是包含这个 term 的文件信息列表，包括有几个文件包含，哪些文件包含了它，在每个文件中 term 出现的次数和位置等。利用这个数据结构，可以根据 term 直接找出包含它的文件。

维护：有得必有失，为了维护这个数据结构，每次载入文件时，每读一个词，都需要及时更新倒排索引，方法非常简单：

Index Generator

```
while ( read a document D ) {
    while ( read a term T in D ) {
        if ( Find( Dictionary, T ) == false )
            Insert( Dictionary, T );
        Get T's posting list;
        Insert a node to T's posting list;
    }
}
Write the inverted index to disk;
```

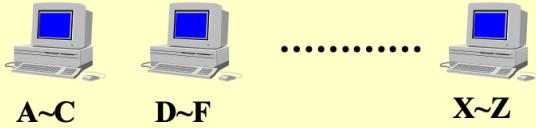
3. 搜索引擎的细节

- (1) term access 的方法，开头说了，倒排索引是普通树索引的下游延伸，所以这个问题中给定一个 term，也要通过 term 的一个树去查询这个 term 对应的在倒排索引中的下标，而不要线性扫描。当然，也可以直接把 posting list 存在 term 的索引树里。另外，hash 也可以用来索引 term，但是不适合范围查询。
- (2) word stemming 词干提取：搜索引擎中对于等价的词语应该一视同仁，化成最简

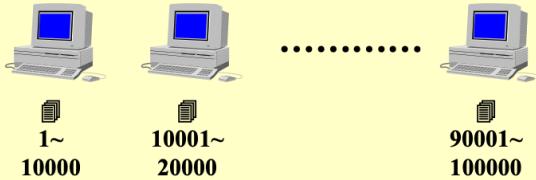
形放在 term 表中。

- (3) stop words : 对于一些简短的意义较少的词，如语气词，可以忽略，不加入索引文件中，或做上一个标记，在查询的时候不返回结果。
- (4) distributed indexing 分布式索引 : 注意索引文件通常比较大，是记录在磁盘上的，如果需要多个磁盘，涉及到内容分配的策略。

☞ Solution 1: Term-partitioned index



☞ Solution 2: Document-partitioned index



这两种分配方式中，term-partition 每个存储单元只存储一部分单词，好处是对任何一个 term 可以直接给出结果，坏处是需要先定位 term 在哪个磁盘；而 document-partition 每个存储单元只存储一部分文件中的结果，好处是如果只搜索一部分文件中的内容效率较高，坏处是需要访问所有存储单元才能返回所有结果。

(5) dynamic indexing 动态索引

将新插入的结果放在辅助索引 (auxiliary index) 中，容量小访问快，优先在其中查找，如果找不到再去主索引文件去找。当辅助索引太大时与主索引合并。

(6) thresholding 阈值控制

这里有两个概念：

第一个是文件返回阈值：只返回关联度最高的 x 个文件/网页。

第二个是查询选择阈值：对于一次搜索多个 term 的情况，只搜索前 $p\%$ 的 term，剩下的不管。注意先对搜索的 term 排序，按词频升序排，优先查询频率低的前 $p\%$ 个词。注意这里词频指的是总词频，即单个单词在所有文件中的出现次数，也可以按包含它的文件树来排。这是因为频率低的词被认为更有可能代表要查的内容。

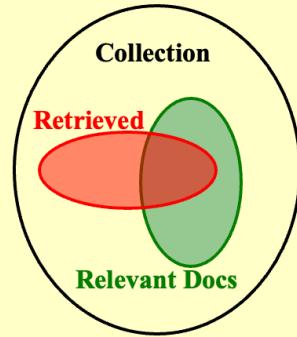
(7) 性能评价

除了搜索速度外，我们关心搜索引擎的相关度分析 (relevance measurement)，即搜寻的结果到底多相关。

Relevance measurement requires 3 elements:

1. A benchmark document collection
2. A benchmark suite of queries
3. A binary assessment of either Relevant or Irrelevant for each query-doc pair

	Relevant	Irrelevant
Retrieved	R_R	I_R
Not Retrieved	R_N	I_N



$$\text{Precision } P = R_R / (R_R + I_R)$$

$$\text{Recall } R = R_R / (R_R + R_N)$$

如图, 相关度主要包含两个指标 :precision 精准度、recall 召回率。其中 Precision 指的是返回的内容中相关文件的比重。Recall 指的是所有相关文件中被返回了的文件比重。一般来说, 返回阈值设置的越高, precision 越低、recall 越高, 两者难以兼得, 要根据具体需要有所权衡。

五、左倾堆, 斜堆 leftist heap, skew heap

1. 堆的概念回顾 :

堆 (heap), 是用来实现优先队列的最常见的数据结构。优先队列是一种抽象的数据结构, 每次可以 **log** 级别地弹出一堆数据的**最值 (最主要目的)**。普通的堆由数组实现, 抽象是可以看成一颗完全二叉树, 有过滤 (percolate) 这个基本操作 ($\log N$), 可用于实现 pop、push 等基本操作, 另外也有线性建堆的经典算法, 都在 fds 中学过。与搜索树不同, 堆不支持随机访问及随机删除, 只能删除堆顶元素。并且堆的二叉树结构中不能保证子节点的大小关系, 只能保证底层数据一定比高层数据大/小。但是普通的堆有一个缺点: 合并 (merge) 效率低下, 需要 N 的复杂度 (假设两个堆大小都是 N 级别, 方法是重新建堆)。而左倾堆和斜堆的目的就是把 merge 的复杂度降到 \log 级别, 注意插入和删除本质可以化为 merge 的特例, 所以实际上只需要解决 \log 级别 merge 的问题。

2. 左倾堆 leftist heap

【Definition】 The **null path length**, $Npl(X)$, of any node X is the length of the shortest path from X to a node without two children. Define $Npl(NULL) = -1$.

Note:

$$Npl(X) = \min \{ Npl(C) + 1 \text{ for all } C \text{ as children of } X \}$$

【Definition】 The **leftist heap property** is that for every node X in the heap, the null path length of the **left child** is **at least as large as** that of the **right child**.

左倾堆由一个重要概念是节点的 npl, 定义如图, 没有两个孩子的叶节点的 npl 是 0。左倾堆保证任一个节点的左孩子 npl 一定大于等于右孩子的 npl。并且有一个关键性质 : 若一个左倾堆的 right path 上有 r 个节点, 那么堆至少一共有 $2^r - 1$ 个节点。这说明右路径长度级别是 $\log N$, 所以提示我们, 左倾堆的操作要在右路径上进行。

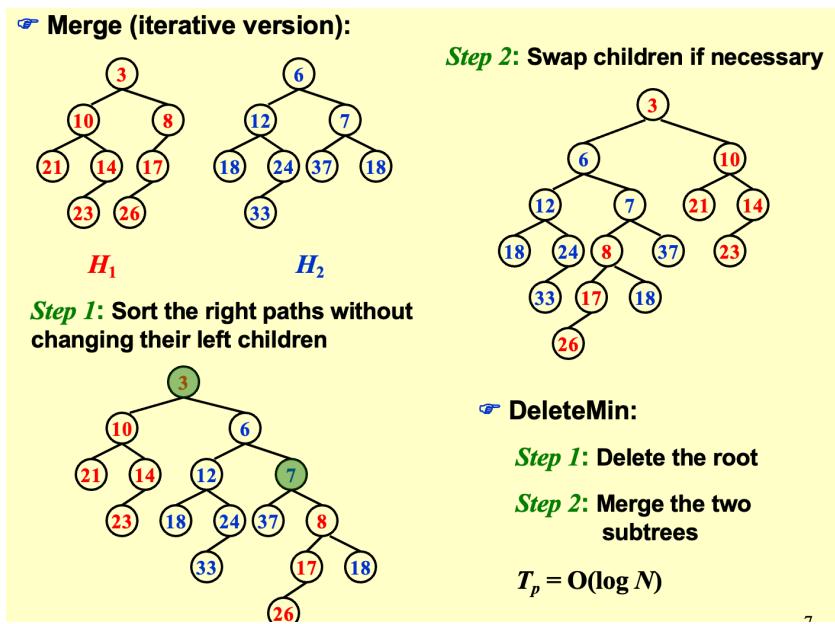
```

PriorityQueue Merge ( PriorityQueue H1, PriorityQueue H2 )
{
    if ( H1 == NULL )  return H2;
    if ( H2 == NULL )  return H1;
    if ( H1->Element < H2->Element )  return Merge1( H1, H2 );
    else return Merge1( H2, H1 );
}

static PriorityQueue
Merge1 ( PriorityQueue H1, PriorityQueue H2 )
{
    if ( H1->Left == NULL ) /* single node */
        H1->Left = H2;      /* H1->Right is already NULL
                                and H1->Npl is already 0 */
    else {
        H1->Right = Merge( H1->Right, H2 ); /* Step 1 & 2 */
        if ( H1->Left->Npl < H1->Right->Npl )
            SwapChildren( H1 );                /* Step 3 */
        H1->Npl = H1->Right->Npl + 1;
    } /* end else */
    return H1;
}
 $T_p = O(\log N)$ 

```

具体 merge 方法如上图, 即每次将堆顶较大的堆根堆顶较小的堆的右子树递归 merge, 如果结果导致合并的堆不符合 npl 要求, 则交换 (swap) 左右子树。注意更新节点的 npl 值。



递归算法不利于人力推导, 迭代法更容易理解, 这种方法我们先只考虑在右路径上不断合并, 最后从右路径的最低的两个孩子的节点开始, 逐步向上检查 npl, 如果不符合再 swap。

3. 斜堆 skew heap

斜堆是自调整的数据结构，没有 npl 的概念。在 merge 的递归或迭代过程中，不管任何情况，**每次都交换左右子树**。在迭代的推导中，右路径合并完后，对每个右路径上的节点都进行 swap。

省去了维护 npl 的复杂度，均摊复杂度仍是 $\log N$ 。

Amortized Analysis for Skew Heaps

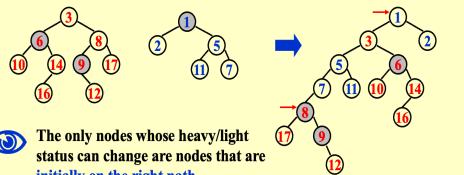
Insert & Delete are just Merge

$$T_{\text{amortized}} = O(\log N) ?$$

D_i = the root of the resulting tree

$\Phi(D_i)$ = number of **heavy** nodes

【Definition】 A node p is **heavy** if the number of descendants of p 's right subtree is at least half of the number of descendants of p , and **light** otherwise. Note that the number of descendants of a node includes the node itself.



The only nodes whose heavy/light status can change are nodes that are initially on the right path.

$$H_i : l_i + h_i \quad (i = 1, 2) \quad \rightarrow \quad T_{\text{worst}} = l_1 + h_1 + l_2 + h_2$$

Along the right path

$$\text{Before merge: } \Phi_i = h_1 + h_2 + h \quad T_{\text{amortized}} = T_{\text{worst}} + \Phi_{i+1} - \Phi_i$$

$$\text{After merge: } \Phi_{i+1} \leq l_1 + l_2 + h \quad \leq 2(l_1 + l_2)$$

$$l = O(\log N) \quad \rightarrow \quad T_{\text{amortized}} = O(\log N)$$

具体证明方法中，势能函数定义也比较复杂。

六、二项队列 binomial queue

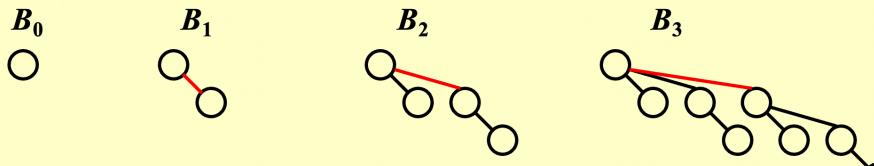
1. 概念：虽然左倾堆和斜堆解决了 \log 级别 merge 的问题，但是它们丢掉了普通堆本存在的一个优良性质：可以自顶向上线性建堆。左倾堆和斜堆没有这样的算法，只能逐个插入，复杂度为 $N \log N$ ，不佳。二项队列是一种极为精巧的数据结构，能同时保证 \log 级别 merge 和线性建堆的可行性，思路来自于二进制数的运算。二项队列用森林（forest，即若干个树）来实现。

Structure:

A binomial queue is not a heap-ordered tree, but rather a **collection** of heap-ordered trees, known as a **forest**. Each heap-ordered tree is a **binomial tree**.

A binomial tree of height 0 is a one-node tree.

A binomial tree, B_k , of height k is formed by attaching a binomial tree, B_{k-1} , to the root of another binomial tree, B_{k-1} .



Observation: B_k consists of a root with k children, which are B_0, B_1, \dots, B_{k-1} . B_k has exactly 2^k nodes. The number of nodes at depth d is $\binom{k}{d}$.

二项树 B_k 的结构是一个根节点，下面有 k 个孩子分别是 $B_1 \sim B_{k-1}$ 。一个二项队列由若干个二项树组成。组成的方式由内部节点数（的二进制表达）决定。具体来说，一个节点数为 N 的二项队列， N 二进制表达中为 1 的位置 i ，对应于实现这个

二项队列的森林中的一个二项树 B_i , 而二项队列的 merge 就相当于两组二项树做加法。从而把二项队列的 merge 转化为二项树的 merge 和进位问题。

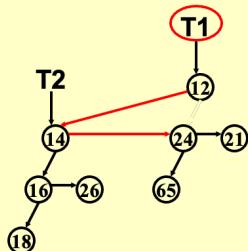
2. 实现的数据结构

真正的二项树并不是用多叉树实现的，而是采用 firstchild-nextsibling 的方法

```

BinTree
CombineTrees( BinTree T1, BinTree T2 )
{ /* merge equal-sized T1 and T2 */
    if ( T1->Element > T2->Element )
        /* attach the larger one to the smaller one */
        return CombineTrees( T2, T1 );
    /* insert T2 to the front of the children list of T1 */
    T2->NextSibling = T1->LeftChild;
    T1->LeftChild = T2;
    return T1;
}
 $T_p = O(1)$ 

```



9

无论是存储还是计算二项树的合并结果，都比较方便。（编程细节较多需要注意！）

3. 合并算法

```

BinQueue Merge( BinQueue H1, BinQueue H2 )
{ BinTree T1, T2, Carry = NULL;
    int i, j;
    if ( H1->CurrentSize + H2->CurrentSize > Capacity ) ErrorMessage();
    H1->CurrentSize += H2->CurrentSize;
    for ( i=0, j=1; j<= H1->CurrentSize; i++, j*=2 ) {
        T1 = H1->TheTrees[i]; T2 = H2->TheTrees[i]; /*current trees */
        switch( 4*!Carry + 2*!T2 + !!T1 ) { /* assign each digit to a tree */
            case 0: /* 000 */
            case 1: /* 001 */ break;
            case 2: /* 010 */ H1->TheTrees[i] = T2; H2->TheTrees[i] = NULL; break;
            case 4: /* 100 */ H1->TheTrees[i] = Carry; Carry = NULL; break;
            case 3: /* 011 */ Carry = CombineTrees( T1, T2 );
                H1->TheTrees[i] = H2->TheTrees[i] = NULL; break;
            case 5: /* 101 */ Carry = CombineTrees( T1, Carry );
                H1->TheTrees[i] = NULL; break;
            case 6: /* 110 */ Carry = CombineTrees( T2, Carry );
                H2->TheTrees[i] = NULL; break;
            case 7: /* 111 */ H1->TheTrees[i] = Carry;
                Carry = CombineTrees( T1, T2 );
                H2->TheTrees[i] = NULL; break;
        } /* end switch */
    } /* end for-loop */
    return H1;
}

```

模拟二进制树的加法。这张图要理解。

二项队列的均摊复杂度是 $\log N$ (平均复杂度不一定)。势函数取为二项树的个数。

4. DeleteMin

先找到最小的元素 (最小堆)。然后将该元素所在树 B_k 堆顶删除变成 k 个堆，这 k 个堆形成一个新的二项队列，然后进行合并。

算法部分：

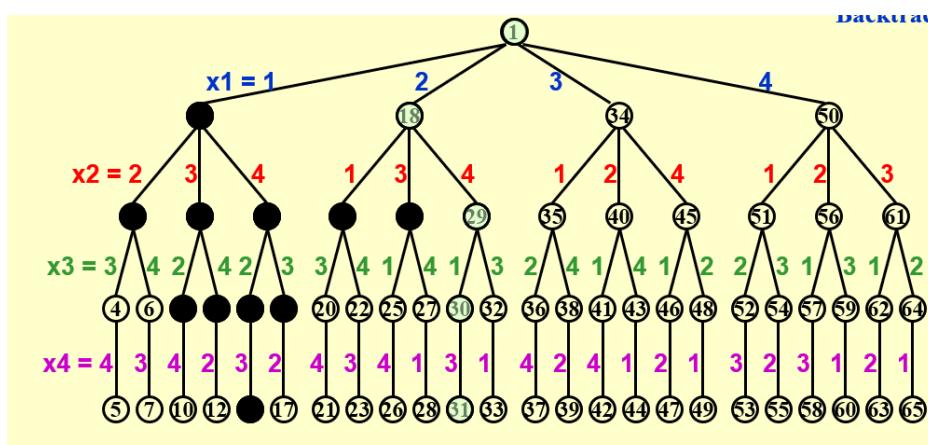
七、回溯法 backtracking

1. 解决的问题：有一类问题是搜索类问题，即在搜索空间内返回符合要求的元素（解），并且往往输入给的信息较少，甚至没有输入，只是给了一套规则和目标。比如排序问题也可以理解为搜索问题，空间是 $n!$ 个排列方式，但一般不归入我们说的这种搜索问题中，因为往往可以利用输入的信息有更好的解题思路。最典型的搜索问题是棋盘类问题。

2. 所谓回溯法，其实就是对暴力枚举发的一种改进。暴力枚举法枚举每一种可能性，而回溯法利用树型遍历结构，及时剪枝（prune），避免不必要的枚举。

```
bool Backtracking ( int i )
{ Found = false;
  if ( i > N )
    return true; /* solved with (x1, ..., xN) */
  for ( each xi ∈ Si ) {
    /* check if satisfies the restriction R */
    OK = Check((x1, ..., xi), R); /* pruning */
    if ( OK ) {
      Count xi in;
      Found = Backtracking( i+1 );
      if ( !Found )
        Undo( i ); /* recover to (x1, ..., xi-1) */
    }
    if ( Found ) break;
  }
  return Found;
}
```

这是回溯法的模板，这里假定搜索空间是 n 维的，目标就是找出可行解 $(x_1 \sim x_n)$ 。check 函数是关键函数，负责检查部分解 $(x_1 \sim x_i)$ 有没有可能继续生成可行解的可能性，如果没有，则剪枝，回溯到更小的部分解。如何利用 check 分析剪枝，是关键所在，如果 check 太细致，容易导致内部复杂度过高，反之如果 check 太粗略，则不容易剪枝，导致不必要的搜索，设计出简明且有效的剪枝函数 check 是回溯法的关键。



3. 典例

典例 1：八皇后问题：在 8*8 的国际象棋棋盘上放八个皇后使其横竖、交叉不到一起。

(即按照国际象棋的玩法不会互相攻击)

搜索空间 ($x_1 \sim x_8$)， $1 \leq x_i \leq 8$ 为整数。

剪枝条件为：

$$\textcircled{2} \quad x_i \neq x_j \text{ if } i \neq j \quad \textcircled{3} \quad (x_i - x_j) / (i - j) \neq \pm 1$$

此时为完全剪枝：剪枝 \Leftrightarrow 该节点不可能生成继续可行解。注意一般来讲剪枝是不可继续的充分必要条件，即只是完成一个初步的可能性筛查，排除绝对不可能的以减少搜索。这里八皇后问题条件比较简单，可以直接等价判断。

典例 2： α - β 剪枝

常见于棋类算法中，或者说博弈类问题中，但其思想是剪枝的条件不止跟当前节点有关，而且跟上层或同层中的、已遍历过的节点有关。或者说，可以利用已经遍历的节点带来的信息，来帮助排除不必要搜索、进行剪枝。

比如八皇后问题中，对于已经排好的五个节点，我们检查其交叉情况并剪枝，但并不关心其它五节点排列的情况（也不会有影响），只是根据当前情况做决定。而在有些问题如棋类问题中，涉及到博弈时，搜索树内部关联度高，可以利用已遍历的节点信息。

另外注意在博弈问题的 α - β 剪枝中，目的并不只是求搜索空间的最优解（搜索树叶节点），更重要的找到最优路径（难说清楚）。

评价函数 f ：对于每一个搜索空间的部分解，可以赋予一个函数，将这个部分解的局面映射到一个实数，表示这个部分解的可行程度（或优秀程度）。比如对于井字棋，我们定义 $f = W_c - W_h$ ，其中 W_c 表示自己的潜在的“成三”方法数， W_h 是对手的。对于更复杂的博弈问题，如围棋，评价函数更难制定，并且是关键所在。如果 f 设置的不好，可能结果很差。

min-max 层与剪枝策略：

搜索树从顶到上，每层交替为 max、min 层，第一层（根节点）是 max 层。

搜索时给每个节点都赋评价函数值，且产生顺序与节点遍历顺序相同。

max 层是自己的决策层，每个节点的值取的是其所有子节点的最大值。

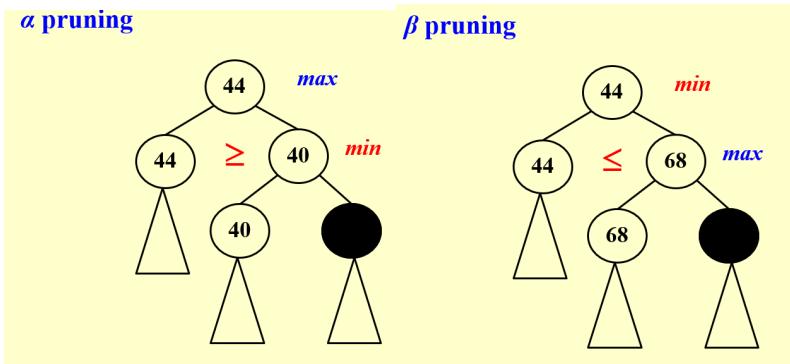
min 层是对手的应对层，每个节点取所有子节点的最小值。

即 max 层选下面 min 层的最大者，min 层选下面 max 层的最小者。

在选择过程中是把自身的值实时更新的，如果发现没必要继续分析，则剪枝：

α 剪枝：min 层在选最小值时，发现自己已经不可能是同一 min 层中的最大者，不可能被 max 层选中，剪掉其所有未被遍历过的子节点。

β 剪枝：max 层在选最小值时，发现自己已经不可能是同一 max 层中的最小者，不可能被 min 层选中，剪掉其所有未被遍历过的子节点。



按这种剪枝策略，最终搜索复杂度不超过 $O(\sqrt{\text{size of search tree}})$ 。

最常见的问法是最先被剪枝的节点是哪个：

注意遍历的顺序是先从左往右、再从下往上的，二叉树中，被剪掉的节点往往是右节点。

八、 分治 divide and conquer

1. 思想：分治是非常重要的算法思想：即将大问题先转化（divide）为若干个子问题，子问题解决完（conquer）后想办法合并（merge/combine），给出大问题解决方案。
注意子问题往往可以继续分解为更小的子问题，这就涉及了递归（recursion）
2. 注意点
 - (1) 分解成的不同子问题之间在各自的解决上必须互无关联，即形式上可以作为原问题的规模更小的 instance。子问题不能需要自身不包含的信息才能解决，否则无“分治”可言。所以要定义好子问题，保证其独立性。
 - (2) 要想好如何根据小问题的解合并成大问题的解，这步同样不能复杂度太高。
 - (3) 对于足够小的子问题，可能直接解决比继续划分效率更高，如快速排序在子数列足够小时用普通排序方法速度更快。
 - (4) 分治法应用极其广泛，但并不是所有问题都能应用分治思想，尤其是全局内部关联度非常高的问题，比如棋类博弈问题。

3. 理论分析形式

设分治法每次都将大问题分为 a 个相同大小的子问题，每个子问题大小为 N/b ，注意这里 b 未必等于 a 。（why, think）。 $f(N)$ 是合并这 a 个子问题的结果所需要的时间复杂度（往往是比较难算的部分），通常来讲，可以化为 $N^p * (\log N)^q$ 的形式。

$$T(N) = a T(N/b) + f(N)$$

求 $T(N)$ 是分治问题的常见考法，比较简单。分为三种方法 substitution method, recursion tree method, master method。

第一种方法靠假设、递归论证，第二种靠生成递归树理论计算（较难，且细节多）

第三种针对特定情形的 f 套公式，可应对绝大多数情况：

【Theorem】 The solution to the equation

$$T(N) = a T(N/b) + \Theta(N^k \log^p N),$$

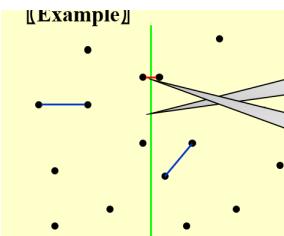
where $a \geq 1$, $b > 1$, and $p \geq 0$ is

$$T(N) = \begin{cases} O(N^{\log_b a}) & \text{if } a > b^k \\ O(N^k \log^{p+1} N) & \text{if } a = b^k \\ O(N^k \log^p N) & \text{if } a < b^k \end{cases}$$

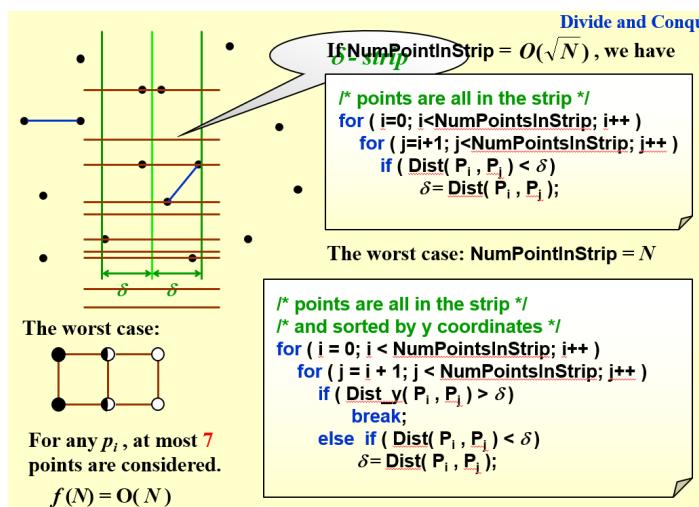
可以感性地理解为（记忆技巧）：递归递归产生的复杂度是 $N^{\log_b a}$, 这个指数 $\log_b a$ 跟 k 做比较，更大的一方显然可以主导复杂度。如果两者相等，给 f 的 \log 的指数加一作为补偿（必须注意这个补偿！）。

4. 典例：最近点问题

给出平面上的 N 个点，找出距离最相近的两个点。两两比较距离显然需要平方的复杂度。分治算法中，首先做一些准备工作，把点按 x 坐标排序，然后设一条竖直线 l ($x = k$, k 取 x 组的中间值) 分成两部分，分别递归求最小距离值，此外，还要考虑跨过 l 的点之间的距离，如果两两比较，合并的复杂度仍是平方，所以我们利用



在子问题中得到的两个最小值中的最小者（不跨过 l 的两个点的最小点距）的一半，作为 strip，在 l 的两侧窄条上下划分出若干个网格区域，每个点只需要考虑周围网格内的其他点，不需要考虑更远的点，这样可以保证最多考虑七个点。



公式为 $T(N) = 2T(N/2) + O(N)$, 总复杂度为 $N \log N$

九、 动态规划 dynamic programming

1. 思想：动态规划（简称 dp），是非常重要和常见的算法思想，基本想法也是把大问题转化为较小的子问题，但与分治思想不同的是，分治是将大问题转化为若干个不相关联的小问题，关键在于如何合并。而 dp 则是尝试从小问题开始，逐步生成最终的大问题，关键在于如何推进生成。注意也要讲 dp 区别于递归，递归是先从顶层考虑，一路向下，遇到基本情况后再回溯，而 dp 则是先“搭地基”，把基本情况全部解出，逐步向上最终得到结果，相比于递归，dp 往往能减少重复计算，这是因为从算法的形式来讲，dp 的形式往往是迭代。

2. 基本方法：

- (1) 确定子问题，定义状态值
- (2) 列出状态初始值（基准值），写出状态转移方程（关键）
- (3) 按正确顺序计算全部状态值
- (4) 根据状态转移过程重塑结果

其中定义合理的子问题和状态转移方程是最关键的。状态一般用一个多维数组来表示，而所谓状态转移方程就是小问题推向大问题的过程，常涉及到参数筛选。初始值和转移方程的形式决定了状态数组的计算顺序。计算过程中，状态转移方程的筛选结果可以记录下来，方便重塑结果。（重塑：有时求的不只是最终的目标值，还有目标值的构建过程，或者构建结果，参考典例）。

3. 典例

典例 1：斐波那契数列，最经典的说明 dp 相对于递归优越性的例子，用递归是指数级别，用 dp 是线性级别。太简单，略。

典例 2：矩阵乘法顺序

给你由 n 个矩阵组成的乘法式，决定一种计算顺序使得总计算量最小（即加括号）。注意 axb 的矩阵和 bxc 的矩阵相乘的复杂度是 abc 。

如果采用暴力枚举，搜索空间大小是 catalan number，接近指数级，无法接受。

Suppose we are to multiply n matrices $M_1 * \dots * M_n$
where M_i is an $r_{i-1} \times r_i$ matrix. Let m_{ij} be the cost of the optimal way
to compute $M_i * \dots * M_j$. Then we have the recurrence
equations:

$$m_{ij} = \begin{cases} 0 & \text{if } i = j \\ \min_{i \leq l < j} \{ m_{il} + m_{l+1j} + r_{i-1}r_l r_j \} & \text{if } j > i \end{cases}$$

Dp 定义了这样的子问题以及初始条件，状态转移方程。按照这种算法按一定顺序求完 m 这个二维数组， $m[1][n]$ 就是最终答案，复杂度是 N^3 。

典例 3：优化二叉查找树 OBST（最难）

传统的二叉查找树只需要满足符合二叉大小条件的同时尽量保持平衡即可，但是有的情形下，不同数据（节点）有着不同的被访问到的概率/频率，所以希望能让这些节点更上层，从而减少平均查询时间（定义如图，根结点的深度 d 是 0）。我们希望能得到最少的平均访问时间，并且得到构建优化二叉树的一种原则/流程。

Given N words $w_1 < w_2 < \dots < w_N$, and the probability of searching for each w_i is p_i . Arrange these words in a binary search tree in a way that minimize the expected total access time. $T(N) = \sum_{i=1}^N p_i \cdot (1 + d_i)$

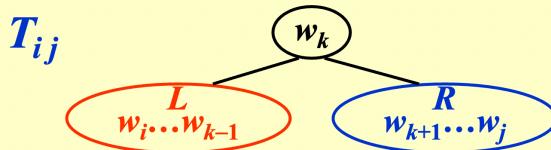
算法：子问题取为之考虑 $i \sim j$ 这些元素生成的 OBST，并且加了一些辅助参数比如 w 辅助理解，可以直接根据输入算出。

$T_{ij} ::= \text{OBST for } w_i, \dots, w_j \ (i < j)$

$c_{ij} ::= \text{cost of } T_{ij} \ (c_{ii} = 0)$

$r_{ij} ::= \text{root of } T_{ij}$

$w_{ij} ::= \text{weight of } T_{ij} = \sum_{k=i}^j p_k \ (w_{ii} = p_i)$



$$\begin{aligned} c_{ij} &= p_k + \text{cost}(L) + \text{cost}(R) + \text{weight}(L) + \text{weight}(R) \\ &= p_k + c_{i,k-1} + c_{k+1,j} + w_{i,k-1} + w_{k+1,j} = w_{ij} + \underline{c_{i,k-1}} + \underline{c_{k+1,j}} \end{aligned}$$

T_{ij} is optimal $\Rightarrow r_{ij} = k$ is such that $c_{ij} = \min_{i < l \leq j} \{w_{ij} + c_{i,l-1} + c_{l+1,j}\}$

状态转移方程就是最下面那个式子。

注意这个方法求出 c_{1n} ，即最小消耗后，需要根据状态转移过程中每一步 k 的选择来帮助重塑二叉树。方法是逐步列表（略）。

时间复杂度为 N^3 。

典例 4：全最短路径问题（Floyd 最短路径算法）

给定一个图（可有权，无负环），关于最短距离问题。我们知道 Dijkstra 算法可以用 bfs 思想算出一个点到其他所有点的最短距离，对于稀疏图复杂度是 V^2 。我们现在想求每两对节点之间的最短距离，如果用 dijkstra，需要一次把每个点作为源（source），总复杂度是 V^3 ，但不太适合稠密图，所以考虑用 dp 降低复杂度。

Method 2 Define

$D^k[i][j] = \min\{\text{length of path } i \rightarrow \{l \leq k\} \rightarrow j\}$

and $D^{-1}[i][j] = \text{Cost}[i][j]$. Then the length of the shortest path from i to j is $D^{N-1}[i][j]$.

Algorithm

Start from D^{-1} and successively generate D^0, D^1, \dots, D^{N-1} . If D^{k-1} is done, then either

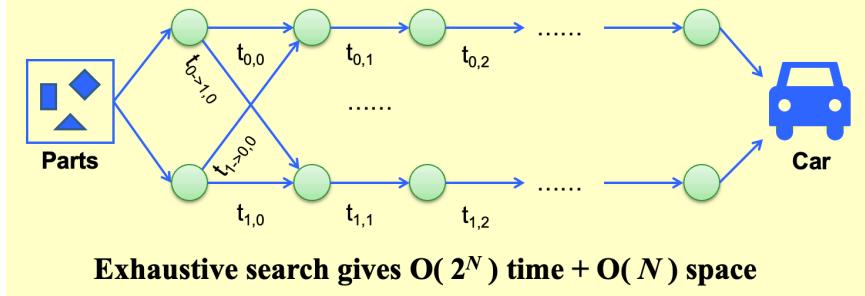
① $k \notin \text{the shortest path } i \rightarrow \{l \leq k\} \rightarrow j \Rightarrow D^k = D^{k-1}$; or

② $k \in \text{the shortest path } i \rightarrow \{l \leq k\} \rightarrow j$
 $= \{\text{the S.P. from } i \text{ to } k\} \cup \{\text{the S.P. from } k \text{ to } j\}$
 $\Rightarrow D^k[i][j] = D^{k-1}[i][k] + D^{k-1}[k][j]$

$\therefore D^k[i][j] = \min\{D^{k-1}[i][j], D^{k-1}[i][k] + D^{k-1}[k][j]\}, k \geq 0$

这是一个三维的 dp， k 的作用是限制路径可以经过的点。状态转移方程氛围经过第 k 个点和不经过第 k 个点两种情况。这样的算法最终复杂度是仍是 V^3 ，但在稠密图上效果比较好（这里复杂度不是太重要，主要是 dp 的方法）。

典例 5：产品装配问题



生产产品有两条流水线，每条流水线上有若干个状态，推进到下一状态需要时间。另外，半产品在某一状态时可以选择转移到另一条流水线的下一个状态，并且耗费不同的时间。需要找出最短的装配时间（以及装配方式）。即决定产品在每个状态到底是继续在自己的流水线上组装还是跳到另一条流水线上。

这个问题中状态和转移方程都比较简单， $f[\text{line}][\text{stage}]$ 表示从最开始，到流水线 line 上的 stage 状态所需的最短路，由于状态的不可回退的，可以利用 dp

$$f[\text{line}][\text{stage}] = \min \{ f[\text{line}][\text{stage}-1] + t < (\text{line}, \text{stage}-1) \rightarrow (\text{line}, \text{stage}), f[\sim \text{line}][\text{stage}-1] + t < (\sim \text{line}, \text{stage}-1) \rightarrow (\text{line}, \text{stage}) \}$$

dp 的复杂度是线性的，且可以根据每一步 line 的选择还原决策过程。

典例 6：0-1 背包问题 (0-1 knapsack problem)

给定一个容量为 M 的背包以及 N 个物品，每个物品有自己的大小 w_i 和收益 p_i ，每个物品要么放入背包要么不放且最多放一次，想求一种将物品装入背包的方式使总收益最大。（就是求长为 N 的一个 0-1 串）

☞ A Dynamic Programming Solution

$W_{i,p}$ = the minimum weight of a collection from $\{1, \dots, i\}$ with total profit being exactly p

① take i : $W_{i,p} = w_i + W_{i-1,p-p_i}$

② skip i : $W_{i,p} = W_{i-1,p}$

③ impossible to get p : $W_{i,p} = \infty$

$$W_{i,p} = \begin{cases} \infty & i = 0 \\ W_{i-1,p} & p_i > p \\ \min\{W_{i-1,p}, w_i + W_{i-1,p-p_i}\} & \text{otherwise} \end{cases}$$

$i = 1, \dots, n; p = 1, \dots, \text{np}_{\max} \rightarrow O(n^2 p_{\max})$

这里定义的子问题比较特殊：用前 i 个物品正好获得收益 p 所需要最小总空间。最后只需要算出使得 $W(i, q) \leq M$ 的最大的 q 即可。注意复杂度有系数 p_{max} ，即物品最高价格，这个值可能很大（值相对于图灵机比特存储位数而言是指数级别的），所以认为 0-1 背包问题是 np 难的。

十、 贪心算法 greedy

- 思想：可以说贪心算法面对的也是搜索类问题，尤其是最优解问题。贪心的基本思想是：**每一步追求最优，最终得到全局的最优**。并且与回溯法不同的是，贪心法不去撤销已经做过的决策，一路走到底。所以纯粹的贪心法往往复杂度比较低，可以减少大量搜索量，但是能否找到最优解需要看技巧和问题性质。注意，可行的贪心保证的是能得到某一个最优解。

- 典例

典例 1：不重合线段选择问题

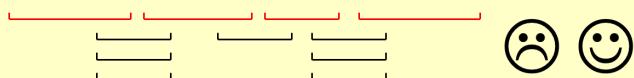
给定 n 条线段（即 n 个数对 $s_i < f_i, i=1,2..n$ ），求互相不覆盖的线段的最多个数。用 dp 的方法做是平方复杂度，贪心算法可以减少复杂度，但是必须制定合适的贪心策略，否则可能得不到最优解。

由于贪心是不可撤销的，所以关键就在于，在已经选好一些线段（或空）的情况下，如何再贪心地选下一条线段？显然这条线段不能跟已选线段重合，这是基础条件，还需要加上贪心条件：经过尝试，发现选起点最前的、长度最短的、冲突最少的都不行，都能构造出反例，最终选择的方案上：**选终点最靠前的（或选起点最靠后的）**。

☞ **Greedy Rule 2: Select the interval which is the *shortest* (but not overlapping the already chosen intervals)**

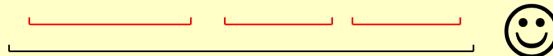


☞ **Greedy Rule 3: Select the interval with the *fewest conflicts* with other remaining intervals (but not overlapping the already chosen intervals)**



☞ **Greedy Rule 4: Select the interval which *ends first* (but not overlapping the already chosen intervals)**

Resource become free as soon as possible



可以在数学上证明这种策略一定能选到最优解。

必须注意的一点是，可行的贪心算法保证得到最优解中的一个，或者说不会有解比它更优，但不能保证问题的任何一个最优解都一定能通过这个贪心算法得到。在证明贪心算法可行性的时侯，思路通常是：假定一个存在的最优解，分析其有没有可能由这个贪心算法得到，如果不可能，则考虑能否将这个最优解转化为可由贪心得到的最优解。

典例 2: Huffman 编码

在编码问题中，如果给每个字符以等长的比特长度，有空间上的浪费。但是定长的好处是很容易分割字符，如果全是不定长编码，可能出现解释程序无法判断字符开始结束位置，出现歧义现象。解决这种问题的关键在于避免字符之间的编码存在前缀关系（prefix）。另外，由于不同字符出现频率不同，所以应尽量给频率高的字符以较短的编码，节省空间。

避免前缀关系最简单也是最实用（甚至具有等价性）的方法是二叉树编码：所有叶节点对应字符，从根节点开始向下寻找叶节点时，向左分叉编码加一个 0，向右则加一个 1，直到叶节点，这样不可能存在前缀关系，并且解释时，只需从根节点开始，根据每一位的 0/1 值向左/右在编码树里推进，直到遇到叶节点，此时进行翻译，然后重新从根节点开始解释下一个比特串。

所以 huffman 编码解决的是用贪心算法生成二叉编码树使得平均编码长度（利用字符频率和长度，长度也是树中叶节点深度计算出的期望值）最短的问题。（补充一点，注意为了让平均长度短（叶节点平均深度低），构建出的树必为 full-tree，即不存在只有一个子节点的节点，因为没有必要，可以把这种节点跟子节点合并）。

➤ Huffman's Algorithm (1952)

```
void Huffman ( PriorityQueue heap[ ], int C )
{ consider the C characters as C single node binary trees,
and initialize them into a min heap;
for ( i = 1; i < C; i++ ) {
    create a new node;
    /* be greedy here */
    delete root from min heap and attach it to left_child of node;
    delete root from min heap and attach it to right_child of node;
    weight of node = sum of weights of its children;
    /* weight of a tree = sum of the frequencies of its leaves */
    insert node into min heap;
}
}
```

$$T = O(C \log C)$$

Huffman 编码算法比较简单：先把 C 个节点写入一个最小堆中，每个节点的值就是对应字符的频率，然后每次新建一个节点，左孩子、右孩子设为从堆里依次推出的两个节点，并将新节点的值设为子节点的值之和，再写入堆中。这样堆每次大小减少 1，重复 C 次即可。其贪心体现在：每次 pop 堆中两个元素，相当于较早地取频率最低两个元素，这意味着它们将放在最终编码树的较低层位置，对应的是让低频字符深度大，从而高频字符深度小。

证明思路大致是：对于一个使平均编码长度最小的编码树 T，考虑频率最低的两个字符 x, y，要么 x, y 是在 T 的最底层的两个兄弟叶节点上，要么 T 可以通过不改变平均编码长度的转化，编程符合这个条件的 T'。

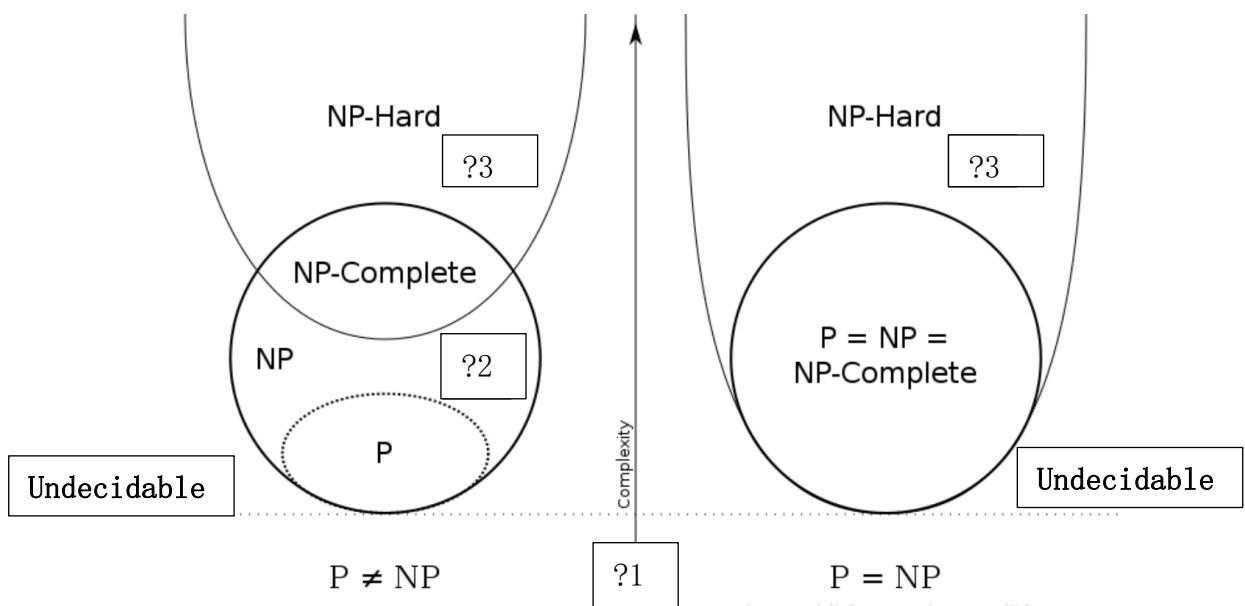
十一、NP问题 NP problem

1. 背景：NP分析，主要是为了确定问题的性质的，或者说这个问题的“难易度”。
2. 图灵机：

图灵机是一种抽象的计算模型，具有内部状态控制、存储、计算的基本功能。这个抽象模型中存储是无限大的，目的是用于模拟人类的一切计算过程。完整的图灵机数学定义是比较抽象的，目前只需要知道图灵机在每一步可以执行计算、寻址、读写、改变自身状态这类基本操作即可。

之所以要给出图灵机的概念，是因为后续问题复杂度分析时，输入大小 N 和运行时间 T 都是由图灵机定义的（但是暂时可以不管）。

3. 问题分类



不确定的东西（?的部分）：

? 1: <P=NP?> 不确定是否 $P = NP$ （目前无法证明一个问题属于 NP 但是不属于 P）。注意图中画的假定 $P \neq NP$ 的情况，此时 P 和 NPC 必不想交。如果 $P=NP$ ，那么 P、NP、NPC 将是 $P=NP=NP$ 的关系。

? 2: <NP – P – NPC = 空 ?> 即使确定了 $P \neq NP$ ，也不确定是否有这样一个 NP 问题，它不属于 P 也不属于 NPC。（这一条我不是很确定，也许必定是存在的）

? 3: <NPC = NP hard ?> 即使确定了 P 和 NP 的关系，也不确定是否 $NPC = NP$ hard（目前似乎无法证明一个非 undecidable 的问题不能用多项式时间验证解的正确性，即证明其不属于 NP）

4. NP的概念

简单地来说，NP问题是指出对问题的任何一个可能的解，**都能在多项式时间内验证其是否正确的问题**。

当然这里涉及到什么是“问题可能的解”和“多项式时间”的定义，但这里不仔细解释抽象定义了。注意问题本身也可能是判断类的问题，此时需要转化成解空间特例的问题。

容易证明 P 一定是属于 NP 的，即能在多项式内找出解的问题一定能在多项式时间内判断解是否正确。

5. 约化 reduce

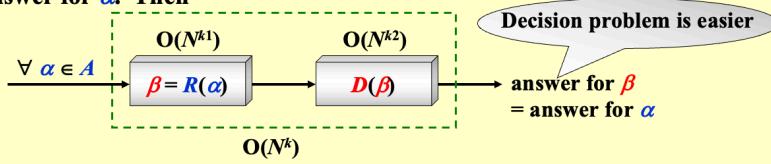
NP-Complete Problems – the hardest

An **NP-complete problem** has the property that any problem in NP can be **polynomially reduced** to it.



If we can solve **any** NP-complete problem in **polynomial** time, then we will be able to solve, in **polynomial** time, **all** the problems in NP!

Given any instance $\alpha \in$ Problem A, if we can find a program $R(\alpha) \rightarrow \beta$ \in Problem B with $T_R(N) = O(N^{k_1})$, and another program $D(\beta)$ to get an answer in time $O(N^{k_2})$. And more, if the answer for β is the same as the answer for α . Then



约化，即问题转化，是问题分析的常用手段，把不熟悉的问题转化为更熟悉的、更有普适性的问题。在 NP 问题分析中，我们一般要求约化过程必须可行且是多项式复杂度的。这种约化在数学上表现为两个问题解空间的映射，由 A 的解空间映射到 B 的解空间，注意这里说的是 B 可以约化成 A，A 是更难的问题，解决了 A 一定可以解决 B。

在 NP 问题中，存在这样一类问题（NPC），所有 NP 问题都可以约化成它，即所有 NP 问题都可以通过 NPC 问题经过某中（多项式时间内的）转化转化而来。NPC 之间彼此等价，解决了任何一个 NPC 问题就相当于解决了所有 NP 问题。

6. 经典 NPC 问题

(1) SAT 问题（第一个被证明为 NPC 的问题）

给定一个 n 个布尔变量组成的布尔表达式，判断其有没有可能为真

(2) 哈密顿回路问题

给定一个图，判断是否有一个 simple cycle 恰好经过每个顶点一次。

(3) 旅行商问题(TSP)

给定一个带权图，问是否存在一个 simple cycle 经历过每个节点且经历边的权重和不超过 K？

(4) 定点覆盖问题(vertex cover problem)

给定一个图，判断是否存在一个大小不超过 K 的点集，使得图中任一条边至少有一个顶点在这个点集中。

(5) 团问题(clique problem)

给定一个图，判断图是否有一个大小至少为 K 的完全子图（团）。

(6) 最长路径问题（略）

(7) 背包问题（略）

十二、 近似算法 approximation

- 思想：对于某些比较复杂的问题比如 NPC 问题，我们很难或者不可能真正在多项式时间内求得最优解，所以我们做出退让，在较短时间内尽可能逼近最优解。

Approximation Ratio

【Definition】 An algorithm has an *approximation ratio* of $\rho(n)$ if, for any input of size n , the cost C of the solution produced by the algorithm is within a factor of $\rho(n)$ of the cost C^* of an optimal solution:

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \leq \rho(n)$$

If an algorithm achieves an approximation ratio of $\rho(n)$, we call it a $\rho(n)$ -approximation algorithm.

【Definition】 An *approximation scheme* for an optimization problem is an approximation algorithm that takes as input not only an instance of the problem, but also a value $\varepsilon > 0$ such that for any fixed ε , the scheme is a $(1 + \varepsilon)$ -approximation algorithm.

We say that an approximation scheme is a *fully polynomial-time approximation scheme (FPTAS)* if for any fixed $\varepsilon > 0$, the scheme runs in time polynomial in the size n of its input instance.

$$O(n^{2/\varepsilon}) \quad O((1/\varepsilon)^2 n^3)$$

fully polynomial-time
approximation scheme
(FPTAS)

对于近似算法有近似率的概念，如图。注意一点，对于近似算法，可以决定何时停止近似，同一类型的近似算法，如果只是停止的时机控制不同，那么认为它们属于同一种“approximation schema”，先命名为近似方案吧。对于一种近似方案，为了达到更高的近似率可能需要延后停止时机，从而增大时间复杂度。为了能达到 $1 + \varepsilon$ 的近似率需要达到的复杂度是跟 ε 的一个 n 的函数式，如果这个式子对于确定的 ε ，是多项式的，那么该近似方案是 PTAS，若 n 的多项式指数不会随 ε 趋于 0 而趋于无限大（粗浅定义，不一定是这样），那么认为是 FPTAS。

通常来讲近似算法要有一种策略，往往跟贪心有关。

2. 典例

典例 1：装箱问题：

问题描述：给定 n 个物品，大小均在 $0 \sim 1$ 之间，把它们装进若干个容量均为 1 的箱子，问容纳它们的箱子的最小数目 m 。这是一个 NPC 的问题，考虑用近似算法近似求 m 。

法 1 next fit：第一个物品放在第一个箱子，然后每次对一个新物品，只考虑能否将其放入上一个物品放进的箱子，能就放，不能就用一个新箱子装它。可以证明，这种方法不会用超过 $2m - 1$ 箱子，且存在一种物品大小输入，使得刚好用 $2m - 1$ 个箱子，所以这种算法的近似率是 2。证明非常简单，反正法，证明如果用大于等于 $2m$ 个箱子，则物品总占据空间超过 m ，从而至少用 m 个箱子，矛盾。

从这里可以看出近似率证明的思路：反证，假设存在超出近似率的近似解，证明在这种特例下最优解不可能达到预先设好的值。

法 2 first fit : 也是非常自然的算法，每次对于一个物品，并不是只能放到上一个物品放的箱子，而是遍历所有已有的箱子看哪个有足够的空间可以把这个物品放进去，就放，若都没有空间就新开箱子。这种算法可以证明近似率是 1.7。

概念：online algorithm : 指的是依次处理输入元素的算法策略，而没有先整体性地对输入进行处理。可以证明，对于背包问题，在线算法近似率最佳是 $5/3$ ，不可能更优。

法 3 : 结合 offline algorithm : 离线算法，先将所有输入载入内存并对其整体分析处理。在这个问题中，麻烦的是大物品，所以我们按大小递减的顺序给物品排序，然后再从大物品开始，执行 first fit 策略。可以证明这种情形下用的箱子不会超过 $11m/9 + 6/9$ ，注意不能说近似率是 $11/9$ 。这种先排序再贪心的算法已经比较接近正确结果了。

典例 2: 背包问题 (knapsack)

0-1 背包问题已经在 dp 一节提出。现在考虑一个变形：分数背包问题。而相比于 0-1 背包，分数背包的区别在于每个物品并不是要么放要么不放，而是可以选择放的百分比 x_i ，这样所占空间是 $x_i \cdot w_i$ ，收益是 $x_i \cdot p_i$ 。更加灵活。这种情况下每次只需要选择收益率 p_i/w_i 最高物品尽量放即可，直至最后一个物品放满。这样是可以得到最优解的。

但对于 0-1 的情况，这种贪心（每次选剩余的收益率最高的且能放得下的物品）只能得到近似解。并且可以证明此时近似率是 2。

The approximation ratio is 2.

Proof: $P_{max} \leq P_{opt} \leq P_{frac}$

$$P_{max} \leq P_{greedy} \quad \rightarrow \quad P_{opt} / P_{greedy} \leq 1 + P_{max} / P_{greedy} \leq 2$$

$$P_{opt} \leq P_{greedy} + P_{max}$$

12

其中 p_{max} 是物品收益的最高值， p_{opt} 是 0-1 问题最大收益， p_{frac} 是假定是分数情形下的最大收益， p_{greedy} 是 0-1 问题用收益率贪婪算法得到的最大收益。

典例 3: K-center problem 中心选择问题

给定平面上 n 个不同的点，要求构造出平面上的 k 个点作为 center，使得 n 个点到最近的 center 的距离的最大值最小。这是很典型的 np 难的问题，必须用近似算法。这个问题有个等价的说法：找出最小的 r ，使得存在平面上的 k 个以 r 为半径的圆，能够覆盖这 n 个点。

提供一种近似率为 2 点近似算法，使得这种算法可以得到 $r \sim 2r$ 之内的半径结果。一个简单的引理论是： n 个节点中，每个节点的 $2r$ 半径范围内必有另一个点。具体方法是，对于一个给定的 R ，我们每次删去一个点和这个点的 $2R$ 的范围内所有其他点，不断重复，如果这个过程不能在 k 次内结束，即选了 k 个以选点为圆心以 $2R$ 为半径的圆，仍不能覆盖所有点，那么说明 $R < r$ 。换句话我们有办法在多项式时间内确定一个半径值是否小于目标值 r 。接下来选定一个大 R_{max} ，从 0 到 R_{max} 用二分法确定 r ，经过有限的迭代后一定可以找到 $r \leq R \leq 2r$ 的 R ，这个值就是近似算法的结果。

可以证明如果 p 不等于 np ，k-center 近似率最好就是 2。

十三、局部搜索 local search

- 思想：局部搜索的想法是用不准确的解作为起始，根据“周围”的条件，以特定的“方向”去逼近最优解。在逼近的过程中一般用到贪婪的思想，但是与贪婪算法不同的点主要在于，贪婪算法是从无到有，一步一步贪婪地生成最终结果，而局部搜索先生成一个起点作为初始的、完整的解，然后贪婪地向最优解的方向逼近。所以局部搜索除了搜索策略外，有时初始解的选择也极为重要。另外一点，纯贪婪算法理论上是没有撤销操作的，但局部搜索为了避免被困在某个局部，可能会回撤操作。

Neighbor Relation

- $S \sim S'$: S' is a **neighboring solution** of $S - S'$ can be obtained by a small modification of S .
- $N(S)$: **neighborhood** of S – the set { S' : $S \sim S'$ }.

```
SolutionType Gradient_descent()
{ Start from a feasible solution S ∈ FS;
  MinCost = cost(S);
  while (1) {
    S' = Search( N(S) ); /* find the best S' in N(S) */
    CurrentCost = cost(S');
    if ( CurrentCost < MinCost ) {
      MinCost = CurrentCost; S = S';
    }
    else break;
  }
  return S;
}
```

其中 neighborhood 是局部搜索中的重要概念，一般是当前解经过一些简单变换后可能生成的新解的集合，局部搜索的操作就是在这些集合中选取正确的变换方向。

需要注意的是局部搜索经常不能得到最优解，可能在最优解“两侧”之间“游荡”，经常用于解决 np 难的问题得到近似解。

注意这个伪代码中，如果发现局部搜索的下一步不能得到更优的结果，则直接退出，但其实这可能会导致 trap 问题，即 trap 在某个局部最优而非全局最优的位置。

所以在此时可以按一定概率 $e^{-\Delta \text{cost}/(kT)}$ 回撤操作，尝试别的搜索路径。这个 T 决定了回撤的概率，类似于“温度”，越高越不稳定，容易回撤，正因如此，一种刚开始让 T 较高、越到后 T 越低的算法称为模拟退火算法，经常用来解决 np 难问题。

2. 典例

典例 1: 定点覆盖问题

NP 一节已经给出定义，这里稍做修改，把判断是否存在大小为 k 的覆盖点集变为找最小的覆盖点集。在这个问题中，初识解设为全部点的集合，每一次局部搜索的操作是删除一个顶点。

典例 2: Hopfield Neural Networks

给定一个带权图，权取任何实数。要求给每个点赋状态 -1 或 1。定义“好边”是端点状态相同且权为负，或端点状态不同且权为正的边，坏边相反。

对于图中的顶点 u , 称其是 stable (稳定) 的, 如果满足其相邻的好边权重绝对值和

$$\sum_{v: e=(u,v) \in E} w_e s_u s_v \leq 0$$

大于坏边的权重绝对值和 : 。对于一种赋值方式, 如果所有

点都是 stable 的, 那么称这个赋值方式也是 stable 的。是不是每个图都一定有稳定的赋值方案呢?

状态反转法 : 为了找到稳定赋值方法。随机选择一种赋值方式, 如果不是稳定的, 找到任一个不稳定的点, 将其翻转 (flip), 即改变状态。注意这样一次操作过后这个节点必然变成稳定点, 但也可能导致与之相连的其他点稳定性发生改变。并且对于边来讲, 每个跟它相临的边的好坏性也都发生翻转, 并且其他边不受影响。

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

Proof: Consider the measure of progress

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

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

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

Clearly $0 \leq \Phi(S) \leq W$

■

这个问题 tricky 点地方就在于, 从任何一个初始的赋值方法, 经过有限次翻转后一定可以得到它(这也是这个问题特别符合局部搜索思想的体现)。因为存在一个在翻转过程中严格递增的值 : 好边的权重绝对值之和 (利用不稳定点的定义和边的翻转性很容易证明), 而这个值又是有上界的, 所以必定终止于没有不稳定点的情况。但注意, 这并不是一个多项式时间的算法, 因为 w 可能很大。

典例 3: 最大切问题 (max cut problem)

给定一个图, 找到一种将其点分成两个集合 A 、 B 的方法, 使得两端分别在 A 、 B 中电边的权重和最大 (也可以是说说 A 、 B 内部的边权重和最小)。

可以借助 Hopfield Neural Networks 问题的思想 : 所有边权重均非负, 此时分成 AB 集合就相当于给每个点-1 或 1 的状态, 只不过选择我们希望好边 (两端点状态不同的边) 权重和最大。依然使用翻转法, 因为已经证明翻转过程中好边绝对值不断增大。最终状态终止于所有点都是稳定点。

但注意这个算法终止情况是不存在不稳定点, 这并不等价于好边的权重和最大, 而只是一个近似, 可以证明, 近似率为 2 :

Claim: Let (A, B) be a local optimal partition and let (A^*, B^*) be a global optimal partition. Then $w(A, B) \geq \frac{1}{2} w(A^*, B^*)$.

Proof: Since (A, B) is a local optimal partition, for any $u \in A$

$$\sum_{v \in A} w_{uv} \leq \sum_{v \in B} w_{uv}$$

Summing up for all $u \in A$

$$2 \sum_{\{u,v\} \subseteq A} w_{uv} = \sum_{u \in A} \sum_{v \in A} w_{uv} \leq \sum_{u \in A} \sum_{v \in B} w_{uv} = w(A, B)$$

$$2 \sum_{\{u,v\} \subseteq B} w_{uv} \leq w(A, B)$$

$$w(A^*, B^*) \leq \sum_{\{u,v\} \subseteq A} w_{uv} + \sum_{\{u,v\} \subseteq B} w_{uv} + w(A, B) \leq 2w(A, B)$$

■

十四、 随机算法 random algorithm

- 随机的概念：随机在算法一般有两种含义，一种是指输入在指定的集合内是随机生成的，另一种是指算法的行为具有随机性（相同输入，相同步骤时，下一步操作可能不同）。随机算法一般指的是第二种。
- 随机算法的目的：采用随机算法，往往是为了规避 bad input。随机的思想并不能真正减少平均复杂度，但是可以把 bad input 转化为 bad random，从而避免连续的 bad result (bad input 是可能连续输入的，bad random 不太可能)。
- 随机输入的重随机化：为了随机输入是坏输入且连续出现，我们可以对输入进行重随机排列 (random permutation)。具体方法是给予每个输入项一个随机优先级 (random priority)，比如 1~N^3 之间的随机数。然后根据优先级进行排序即可。这样可以保证每个输入项在各个位置的可能性是相同的，缺点是排序需要耗时。
- 典例

典例 1：雇佣问题

背景：依次面试 N 个人，目的是找到最好的那个人（按某个质量数组排列）。每个人面试完必须马上告知是否录用，如果已录用，必须支付代价 C_h ，每次面试也有代价 C_i ，但是 C_i 远小于 C_h 。所以要制定策略。让代价较小的同时找到最好的那个人的概率最大 (trade off)。

法 1：如果采用最简单的方法：跟找最大值相同的算法，即但凡当前面试到的人比之前所有人都优秀，就录用。这样最坏有 N 次 hire。平均来说，复杂度是 $O(C_h * \ln N + C_i * N)$ 中 $\ln N$ 是在概率计算过程中由 $1/1 + 1/2 + \dots + 1/N$ 近似出来的。

法 2 (hire only once)：加以控制参数 k，规定对前 k 个人，采用法 1 的策略，即总是录用最优秀的，从第 k+1 个人开始，一旦遇到一个比前 k 个中选出的最优秀的人还优秀的，就马上录用这个人，并停止面试，不管剩下的人。

$$\Pr[S] = \sum_{i=k+1}^N \Pr[S_i] = \sum_{i=k+1}^N \frac{k}{N(i-1)} = \frac{k}{N} \sum_{i=k}^{N-1} \frac{1}{i}$$

可以得到能选到最优秀的人概率是这个值，近似为 $(k/N) \ln (N/k)$ 。

为了让 \Pr 最大，取 $k = N/e$ ，概率值是 $1/e$ 。这种方法下，总花费平均而言是 ? ?

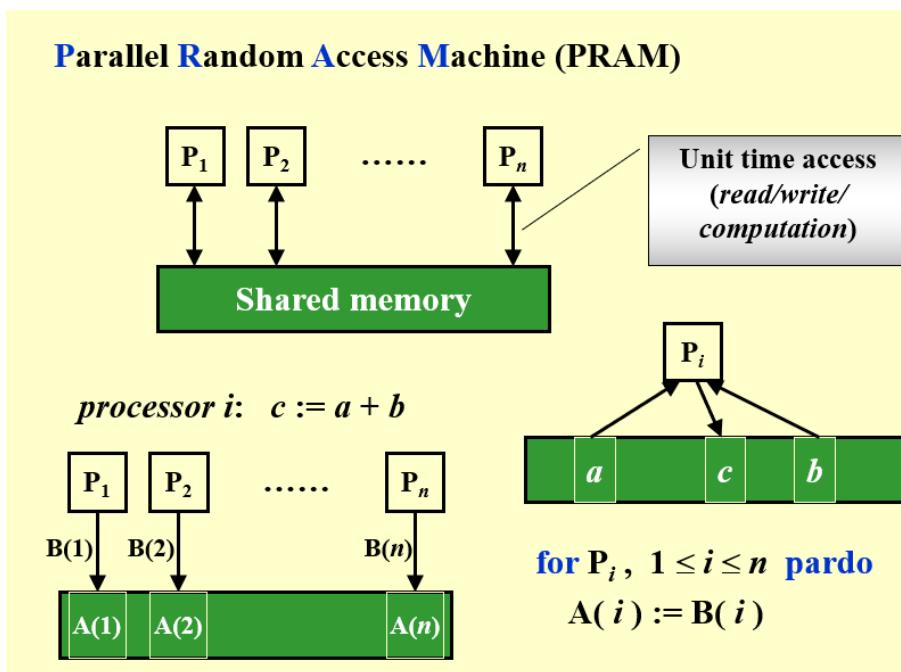
典例 2：快排 pivot 的随机选择

快速排序的核心思想是利用 pivot，将元素分别移到 pivot 两侧的位置，从而可以递归解决。理想的 pivot 应该靠近数组大小的中间位置，选择 pivot 的策略是快排的关键，传统的 pivot 是选择数组两端及中间这三个元素的中间值，虽然有一定道理，但仍不能避免最坏复杂度是 N^2 的情况。

central splitter 算法规规定 pivot 必须在按序排列后的左 1/4 和右 1/4 元素中间，即必须在 1/4 至 3/4 这个数组的中间区间中。如果不是，则重新随机选 pivot（如何判断是不是？）。这个限制中，子问题的大小不超过父问题的 3/4.

十五、 并行算法 parallel algorithm

- 需求：由于流水线、多核 CPU 等硬件技术的存在，使得处理器对外表现可以同时（在一个时钟周期内）执行多条指令。一个算法，或者说解决问题的一个程序，本质上是一堆需要执行的指令集合，串行情况下为顺序执行，后执行的指令可能依赖于先前执行的指令。并行算法，目的就是利用 CPU 可以同时执行多条指令这个特性，合理调度指令执行方式，使得工作量不增多的情况下减少运行时间。
- PRAM (Parallel Random Access Machine)
实现并行的理论机器模型。



内存只有一份，且（理论上）无限大，称为共享内存，所有处理器均可以访问。处理器作为黑箱，基本操作是从共享内存中读取数据、内部计算、向内存写数据（三种）。所谓并行，指的就是各个处理器可以同时（在一个时钟周期内）各自独立地读算写。交换数据（通信）的媒介是内存本身。

For $P_i: 1 \leq i \leq n$, pardo $A(i) := B(i);$

(pardo 并行，B 读算操作， $:=$ 写操作)

- Access conflict 访问冲突

并行一定会带来一个问题就是同一份内存可能同时被多个处理器访问，而某些处理

器可能在写数据，这可能导致其他处理器读到被改过的数据，或者同时写时，先写的被后写的内容覆盖，出现意外情况。

根据处理访问冲突方式的不同，可以将 PRAM 分成三种

(C: concurrent, E: exclusive, R: read, W: write)

EREW: 同一块内存不能同时被多个处理器访问

CRCW: 可以同时读，不允许同时写

CRCW: 可以同时读或写（不加限制）

对于 CRCW 策略，需要解释同时写的情况下，如何决定内存值。

Arbitrary rule：任意赋值，随机

Priority rule：保留编号最(小)的处理器写的结果

Common rule：？同值

4. Word Depth

与 PRAM 不同的是，WD 并不是每一个时钟都给所有 cpu 工作任务（如果无需做，也执行空指令），而是把使用哪些 cpu 也考虑进来，不需要操作的 cpu 不参与程序执行，可以节省能耗，或用于其他程序。

5. 性能评价指标

(1) Work Load W(n): 解决问题所需要的基本操作 (operation) 的数量

如果问题可以用串行算法解决，W(n)一般大于等于 T(n)，且优秀的并行算法能使得两者相同。（并行算法无法减少工作量本身，只能利用并行减少运行时间，为了实现并行可能有额外的操作增加工作量）

(2) Worst-case running time T(n): 最坏时间复杂度。包括计算时间和通信时间两部分。并行算法必须保证 T(n)比串行算法的要小，否则并行无意义。可以以时钟为单位，并行算法中一个时钟内可以由多个 cpu 执行指令，这也是 T(n)比串行情况小的主要原因。

???（有一些奇奇怪怪的参数，不是很懂）

6. 典例

典例 1：求和问题

【Example】 The summation problem.

Input: A(1), A(2), ..., A(**n**)

Output: A(1) + A(2) + ... +A(**n**)

基本思路：并行两两求和，每次将新数组长度缩短一倍，最终求得综合。

Work-Depth (WD) Presentation

```
for Pi, 1 ≤ i ≤ n pardo
    B(0, i) := A(i)
for h = 1 to log n
    for Pi, 1 ≤ i ≤ n/2h pardo
        B(h, i) := B(h-1, 2i-1) + B(h-1, 2i)
for i = 1 pardo
    output B(log n, 1)
```

典例 2 : prefix sum 前缀和问题

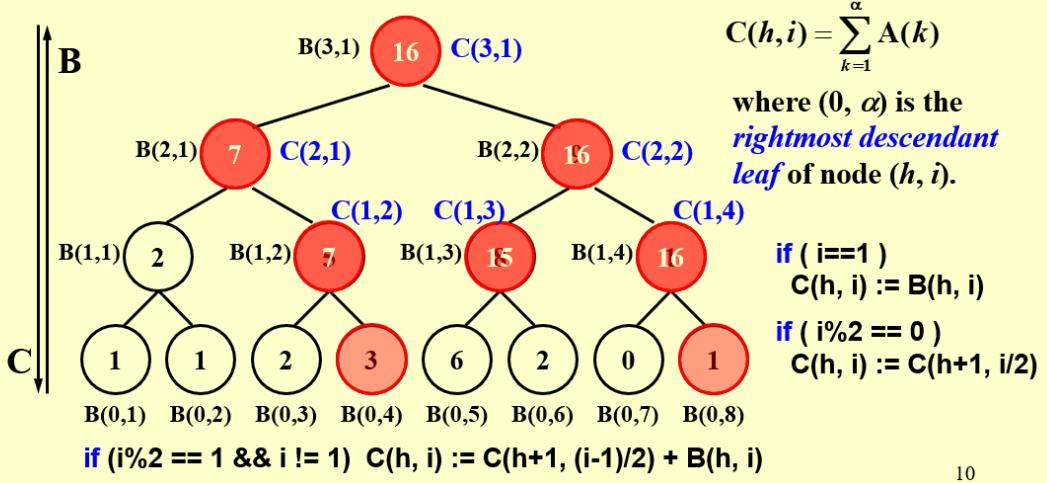
Parallel Algorithms

【Example】 Prefix-Sums.

Input: $A(1), A(2), \dots, A(n)$

Output: $\sum_{i=1}^1 A(i), \sum_{i=1}^2 A(i), \dots, \sum_{i=1}^n A(i)$

* Technique: Balanced Binary Trees



建立在求和算法（求出了所有 B 值）的基础上， $B(h, i)$ 是第 i 层（叶为 0）的从左往右第 i 个节点，其字数的叶节点的和。按如图方法计算 $C(h, i)$ 。对于 C，其值不是子树叶节点和，而是从最左边的叶节点到子树最右边的叶节点的数之和。注意若节点是父节点的右孩子，即 i 为偶数，那么 C 值与父节点相同，否则若是左孩子，可以分成两部分计算左侧叔节点的 C 值和自己的 B 值。每一层的 C 值与同层的 C 值无关，只需要用到上一层的 C 值和已经算好的 B 值，故也可以逐层并行计算，注意 C 是从上往下并行计算的，而且工作量逐层增加，这点与 B 值相反。

计算顺序：

B : h 从小到大， i 并行算出

C : h 从大到小， i 并行算出。利用 B 值的顺序 : h 从大到小， i 并行

典例 3 : 归并数列

归并两个不减数列（简化为等长）：A (1~n) 和 B (1~n) 到 C (1~2n)

首先进行问题转化：并行算法的常用操作：某种操作数组的问题，转化成求一个可以并行求出的辅助数组，利用辅助数组又可以并行重塑操作。

Merging \rightarrow Ranking

```
RANK(j, A) = i, if A(i) < B(j) < A(i + 1), for 1 ≤ i ≤ n
RANK(j, A) = 0, if B(j) < A(1)
RANK(j, A) = n, if B(j) > A(n)
```

The ranking problem, denoted **RANK(A,B)** is to compute:

1. RANK(i, B) for every $1 \leq i \leq n$, and
2. RANK(i, A) for every $1 \leq i \leq n$

Claim: Given a solution to the ranking problem, the merging problem can be solved in $O(1)$ time and $O(n+m)$ work.

```
for P_i, 1 ≤ i ≤ n pardo
    C(i + RANK(i, B)) := A(i)
for P_i, 1 ≤ i ≤ n pardo
    C(i + RANK(i, A)) := B(i)
```

这里转化为 Rank 数组：rank (j, A) 大致是 B (j) 若在 A 数组中，应排到的位置。根据 rank 可以直接在 C 上并行还原 C (tricky 的地方)，从而问题转化为并行求 rank。而对于数组 B 中的某个元素，其在 A 中的 rank 值跟 B 的其他元素显然是无关的，所以并行就很自然了。

Binary Search

```
for Pi, 1 ≤ i ≤ n pardo
RANK(i, B) := BS(A(i), B)
RANK(i, A) := BS(B(i), A)
```

$$T(n) = O(\log n)$$

$$W(n) = O(n \log n)$$

Serial Ranking

```
i = j = 0;
while ( i ≤ n || j ≤ m ) {
    if ( A(i+1) < B(j+1) )
        RANK(++i, B) = j;
    else RANK(++j, A) = i;
}
```

$$T(n) = W(n) = O(n + m)$$

第一种方法即并行地用二分查找求每个元素在另一个元素中的 rank，第二种算法则是线性法一次性得到，但不符合并行思想。

但是第一种方法并不让人满意，因为工作量 W (n) 比串行算法高了，我们希望维持 W 仍是线性的情况下减少 T。

Parallel ranking: 先再 A, B 中分别间隔地选出 p 和元素。对这两个子数组进行二分归并，注意 rank 仍是在另一个完整数组内的排序而不是筛选完的，这样工作量是 $p \log n$ 。在间隔归并完成后，可以根据 rank 的位置，将未归并的剩余元素分组对应，每个组里约有 $n/p - 1$ 个元素，两两对应归并。

取 p 最佳为 $n/\log n$ ，最终的总共工作量可以变成线性，T 仍是 $\log n$ 。

典例 4：寻找最大值

法 1：串行算法找最大值即为线性扫描，很简单。对于并行情况（不考虑 CPU 数量限制），可以并行计算两两比较的结果，在 $O(1)$ 的时间内求出最大值，但这样工作量是 n^2 ，不满意，所以考虑分组，在时间和工作量上达到均衡。

法 2：方根分组法

一种递归的、分组解决的思路，将待排序的长度为 n 的数组分成 \sqrt{n} 组，每组 \sqrt{n} 个元素，并行地对每一组求最大值后（继续分组递归做），对生成的 \sqrt{n} 个最大值，用法 1 求出最大值作为最终结果。

$$T(n) \leq T(\sqrt{n}) + c_1, \quad W(n) \leq \sqrt{n} W(\sqrt{n}) + c_2 n$$

$$\Rightarrow T(n) = O(\log \log n), \quad W(n) = O(n \log \log n)$$

递归分析复杂度和工作量会得到这个结果。工作量依然不是很让人满意。

法 3：双对数分组法

对法 2 和法 1 的结合改进，将数列分成 n/h 组，每组 h 个元素，其中 $h = \log \log n$ 对每一组的 h 个数据，并行使用法 1 的方法，最终对生成的 n/h 个最大值使用法 2 的方法。

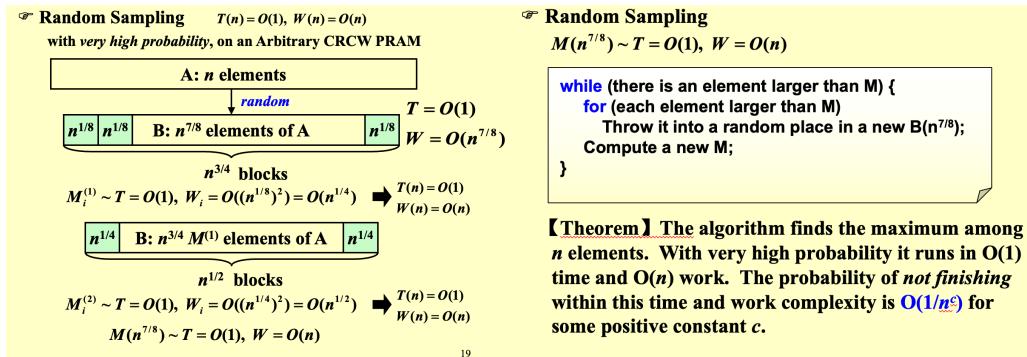
其中第一步 T 为 h ，第二部 T 为 $\log \log (n/h)$ ，最终可得 T 为 $\log \log n$ 级别

第一步 W 为 $h * n / h = n$ (法 1 不增加工作量), 第二步为 $n/h \log \log n / h$, 最终 w 为线性级别。其中 h 取 loglog 的主要目的就是使 W 在第二步刚好不会超线性！

法 4: random sampling 随机抽样法

一种奇怪的、带有随机性的方法, 可以保证有较大概率以 T 为 1, W 为 n 的情况下完成最大值选取。

基本思想是 : 只取输入数组 A 的 $n^{7/8}$ 子部分 B 进行处理 (这里 q 取 7/8)。对这部分 B 使用某种算法达到 T (1)、W (n), 然后进行右图操作。(这里没搞懂其实)



十六、外部排序 external sort

- 问题背景 : 有时需要排序的内容并不在内存, 而是在磁盘 (disk, 或者用 tape 表示) 上 : 给你一条大小为 M 的 tape, 以及大小为 N 的内存, 要求利用内存, 将 tape 中的 M 个元素排序好。(大小指的就是需要排序的元素块的数量, 内存是固定的, tape 有可能有多余的可供利用)。

“外部”指的是待排序的数据在内存外部, 即磁盘中。这种情况下, 对数组的随机访问可能涉及到磁盘读写, 所以外部排序的一个重要目标是合理安排内存载入哪些内容, 减少磁盘 IO。

时间主要消耗分为三种

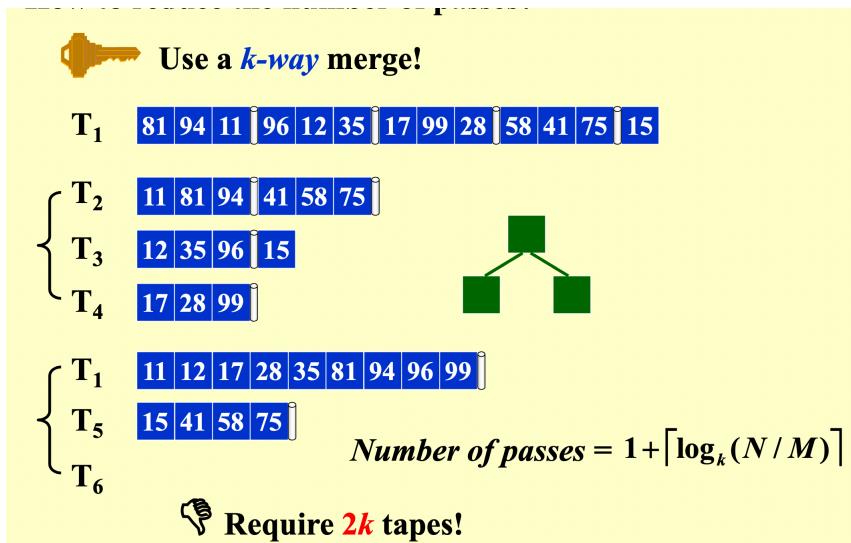
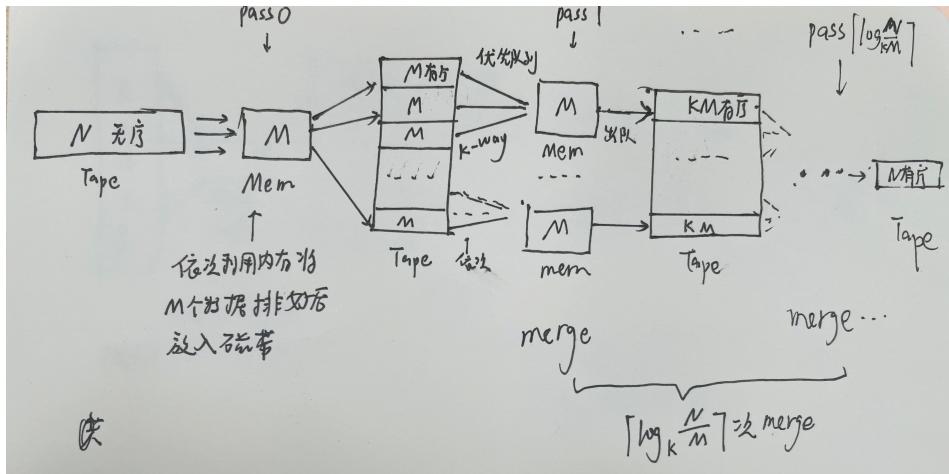
- 内存操作 : 主要用于内部排序, 单步速度极快
- Disk transfer : 将内存中一块数据加载到内存中 (以便于后续计算), 较慢。
- Disk seek : 磁头寻址, 定位到磁盘一系列数据块的起始位置, 极慢。

为了减少运行时间, 一定要减少磁盘 IO 次数, 尤其是 seek。磁盘 IO 主要在 : 从磁盘读数据到内存, 从内存写数据到磁盘。如果必要写, 尽量将内存中较多内容连续写入磁盘的连续块, 避免 seek。

基本参数设定 : N 是磁盘待排序的数据量, M 是内存中可以暂存的数据量, 这里认为 M 比 N 要小, 否则可以先全部加载进入内存, 内部排序后再返回磁盘中。

- k-way merge

外部排序的基本思想是利用内存段将 tape 内容分段都进行 (内部) 排序并暂存在临时 tape 中, 然后再利用内存进行归并 (merge)。其中归并往往用长度为 k 的优先队列。



Pass： 所谓 pass 可以理解为一次磁盘内容的重构，得到有序长度更长的若干个数组，另外更好的理解方式是：(平均) 每个数据从磁盘载入内存的次数。其中内部排序只进行了一次，就是最开始时每 M 个数据在内存中排序后放入新磁带，这个过程是不可避免的，比消耗一个 pass，后面的 pass 均是 merge 产生。

Tape 数量：注意这个方法中，每个 merge 阶段，每归并 k 个 M 长的数据时，需要这 k 个段在不同的磁带中，所以第一张图中的 Tape 并不是真正的磁带结构，而是如图 2，每轮用额外的 k 个 tape，每个 tape 存储若干个 M 段，每次都是对这些 tape 的对应位置段 M 段进行归并。

并行、buffer (不是非常明白，略过了)

