高级算法 - 笔记

[TOC]

递归和分治

插入排序 & 渐进分析

Insertion Sort

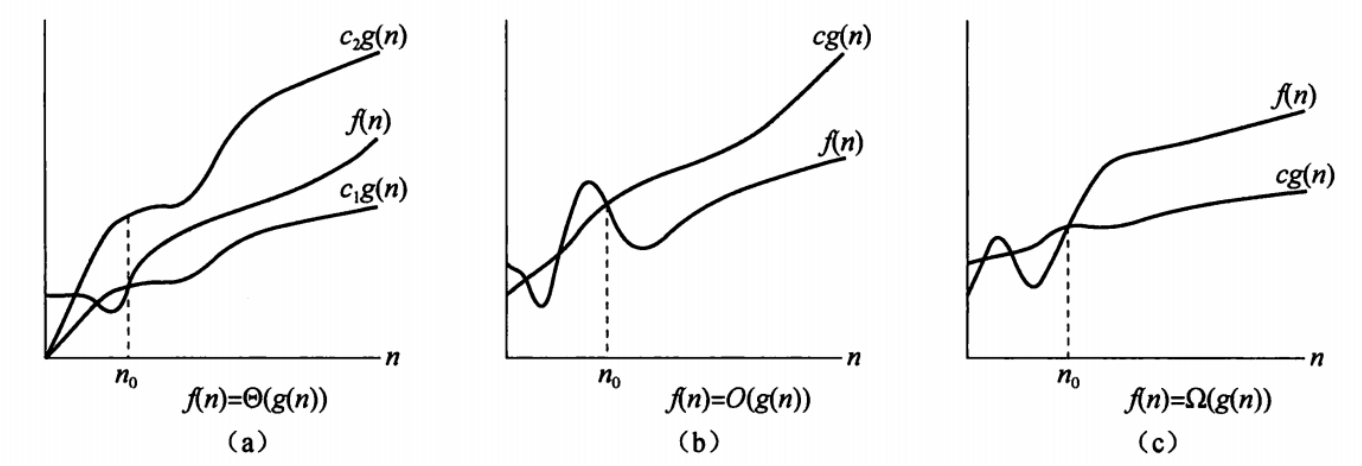
```
InsertionSort(A, n)
    for j ← 2 to n
    do
        key ← A[j]
        i ← j−1
        while i > 0 and A[i] > key
        do
            A[i+1] ← A[i]
            i ← i−1
        A[i+1] ← key
```

We can use **loop invariants** to prove the correctness of the algorithm:

Initialization It is true at the first loop **Maintenance** It is true before an iteration of loop, then true before next iteration **Termination** The invariant guarantee the correctness at last iteration

---

渐进记号



- **Θ 记号**

  若存在正常量 c_1 和 c_2，使得对于足够大的 n，函数 $f(n)$ 能够夹在 $c_1g(n)$ 和 $c_2g(n)$ 之间，则称 $f(n)$ 属于集合 $\Theta(g(n))$，记为 $f(n)=\Theta(g(n))$。称 $g(n)$ 是 $f(n)$ 的一个 **渐近紧确界。**

- **O 记号**

  对于足够大的 n，函数 $f(n)$ 的值总小于或等于 $cg(n)$，记为 $f(n)=O(g(n))$，O 记法表示 **渐进上界**(asymptotic upper bound)。

- **Ω 记号**

  对于足够大的 n,函数 $f(n)$ 的值总大于或等于 $cg(n)$，记为 $f(n)=\Omega(g(n))$，Ω 记法表示 **渐进下界。**

- **定理**

1. 对任意函数 $f(n), g(n)$，当且仅当 $f(n)=O(g(n)),且, f(n)=\Omega(g(n))$，有 $f(n)=\Theta(g(n))$。
2. 如果 $T_{1}(n)=\Theta(f(n)), and , T_{2}(n)=\Theta(g(n))$，那么：
    - $T_{1}(n)+T_{2}(n)=\max (\Theta(f(n)), \Theta(g(n)))$
    - $T_{1}(n) * T_{2}(n)=\Theta(f(n)) * \Theta(g(n))$

递归和分治

归并排序的时间： $$ T(n)=\left\{ \begin{array}{ll} \Theta(1) & \text { if } } n=1 \ 2 T(n / 2)+\Theta(n) & \text { if } } n \neq 1 \end{array} \right. $$

解递归方程的方法

- 代入法（substitution method）
- 递归树（recursion tree）
- 主定理（master method）

1. 代入法

分两步：

1. 猜测解的形式；
2. 用数学归纳法求出解中的常数，并证明对于 $n>n_0$，解是恒成立的。

例子：求递归式 $T(n)=2T(\lfloor n/2 \rfloor)+n$ 的上界。

We guess that the solution is $T(n)=O(nlgn)$, and what we need to prove is that $T(n) \le cn \lg n$ for an appropriate choice of the constant $c>0$.

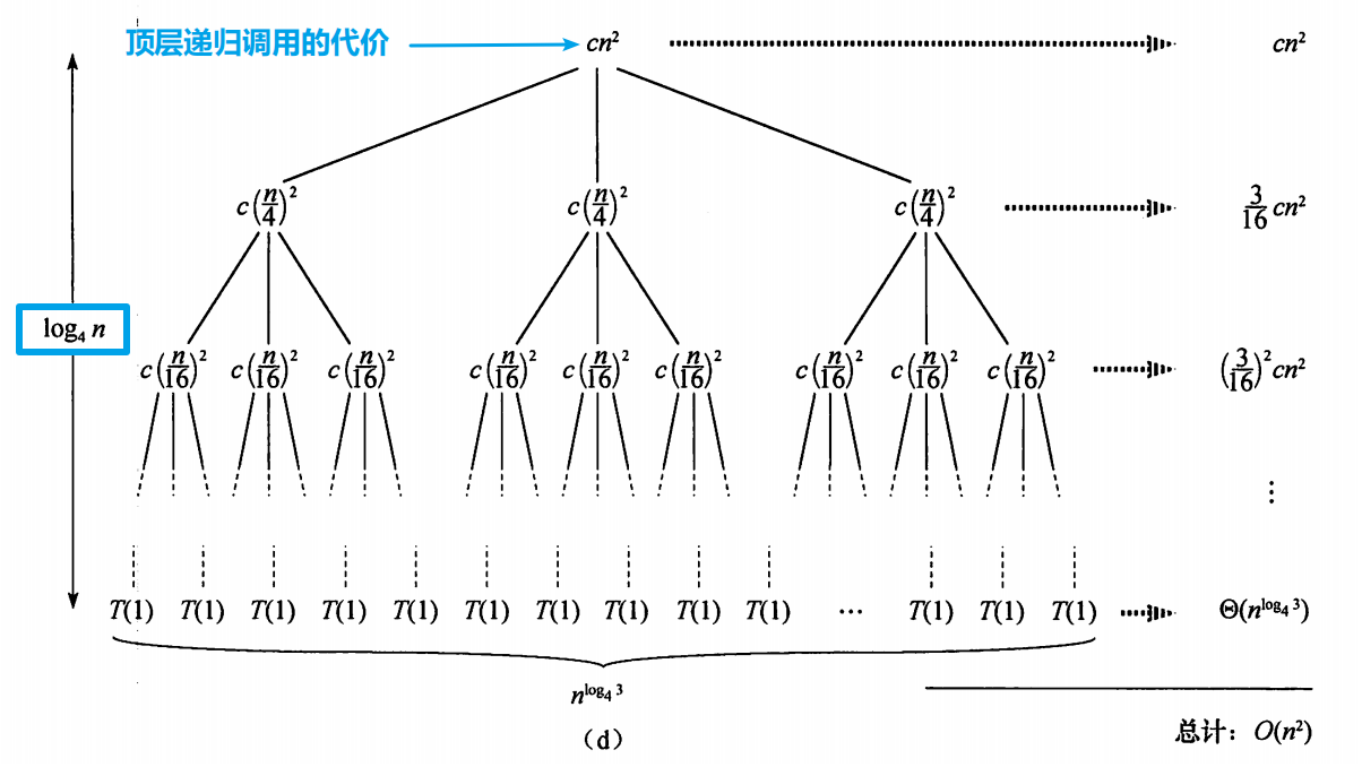
We start by assuming that this bound holds for all positive $m<n$，in particular for $m=\lfloor n/2 \rfloor$, yielding $T(\lfloor n/2 \rfloor) \le c \lfloor n/2 \rfloor \lg (\lfloor n/2 \rfloor)$. Substituting into the recurrence yields $$ \begin{align} T(n) & \le 2(c\lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)) + n \ & \le cn \lg(n/2) + n \ & = cn \lg n -cn + n \ & \le cn \lg n \end{align} $$ where the last step holds as long as $c \ge 1$.

Then we need to show that our solution holds for the **boundary conditions**. We can easily know that $T(1)=1, T(2)=2T(1)+2=4$，so when $c \ge 2 , and , n \ge 2$, our assumption always holds.

If requiring a tighter upper bound, we need to make another assumption and perform the processing above.

2. 递归树

构造递归树可以为代入法生成很好的猜测，下图是从递归式 $T(n)=3T(\lfloor n/4 \rfloor)+cn^2$ 创建递归树的过程：



- 根结点中的 cn^2 表示递归调用在顶层的代价；
- 根的三棵子树表示规模为 $n/4$ 的子问题在递归调用顶层的代价；
- 树高为 $\log_4 n$。

接下来确定所有层次代价之和： $$ \begin{align} T(n) &= cn^2 + \frac{3}{16}cn^2 + (\frac{3}{16})^2cn^2 + \cdots + (\frac{3}{16})^{\log_4 n-1}cn^2+\Theta(n^{\log_4 3}) \[0.2em] &= \sum_{i=0}^{\log_4 n-1}(\frac{3}{16})^i cn^2 + \Theta(n^{\log_4 3}) \[0.2em] &< \sum_{i=0}^{\infin}(\frac{3}{16})^i cn^2+\Theta(n^{\log_4 3}) \[0.2em] &= \frac{16}{13}cn^2 + \Theta(n^{\log_4 3}) \[0.2em] &= O(n^2) \end{align} $$

> 等比数列求和公式：$\sum_{i=0}^{n}q^i = a_1(1-q^n)/(1-q)$。
>
> 等比级数求和公式：$\sum_{i=0}^{\infin}q^i = 1/(1-q)$

3. 主方法

主方法用于求解如下递归式： $$ T(n)=aT(n/b) + f(n) $$ 适用于三种情况：

1. 若对某个常数 $\epsilon > 0$，有 $f(n)=O(n^{\log_b a - \epsilon})$，则 $T(n)=\Theta(n^{log_b a}).$
2. 若 $f(n)=\Theta(n^{\log_b a})$，则 $T(n)=\Theta(n^{\log_b a }\lg n).$
3. 若对某个常数 $\epsilon > 0$，有 $f(n)=\Omega(n^{\log_b a + \epsilon})$，且对某个常数 $c<1$ 和所有足够大的 n 有 $af(n/b) \leq cf(n)$，则 $T(n)=\Theta(f(n)).$

用人话来说就是，对于三种情况的每一种，我们将函数 $f(n)$ 与函数 $n^{log_b a}$ 进行比较。直觉上，**两个函数的较大者决定了递归式的解：**

- 若函数 $n^{\log_b a}$ 较大（情况 1），则解为 $T(n)=\Theta(n^{\log_b a})$
- 若函数 $f(n)$ 较大（情况 3），则解为 $T(n)=\Theta(f(n))$
- 若两个函数大小相当（情况 2），则乘上一个对数因子，解为 $T(n)=\Theta(n^{\log_b a }\lg n)$.

> 但是需要注意，不是 $f(n)$ 小于 $n^{\log_b a}$ 就够了，而是要 **多项式意义上的小于**，也就是说 $f(n)$ 必须渐进小于 $n^{\log_b a}$，要相差一个因子 n^ϵ，其中 $\epsilon > 0$。
>
> 这三种情况没有完全覆盖所有可能，存在一定的**间隙**：$f(n)$ 可能小于/大于 $n^{\log_b a}$ 但并非多项式意义上的小于/大于。这样就不能使用主方法来求解。

【例】：$T(n)=T(2n/3)+1$.

【解】：For this recurrence, we have $a=1, b=3/2, f(n)=1$, since $f(n)=O(n^{\log _{3/2}1})$, case 2 of the master theorem applies, and thus the solution to the recurrence is $T(n)=\Theta(\lg n)$.

【例】：$T(n)=2T(n/2) + n \lg n$.

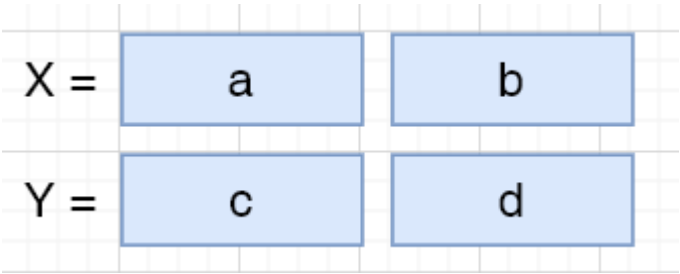
【解】：For this recurrence, we have $a=2,b=2, f(n)=n \lg n$, even though $f(n)=n \lg n$ is asymptotically larger than $n^{\log_b a}=n$, it is not **polynomially** larger. There is no such ω making $f(n)=nlgn>n^{\log_b a + \omega}=n^{1+ \omega}$ holds, because $\lg n$ is asymptotically larger than n^ω for any positive constant ω. Consequently, the recurrence falls into the gap between case 2 and case 3.

分治的例子

1. 大整数乘法

传统方法：$O(n^2)$。

分治：



$X=a2^{n/2}+b,\ Y=c2^{n/2}+d,\ XY=ac2^n+(ad+bc)2^{n/2}+bd$，递归方程为：$$ T(n)= \left{ \begin{array}{ll} \Theta(1) & n=1 \ 4 T(n / 2)+\Theta(n) & n > 1 \end{array} \right. $$ 时间还是 O(n^2)。

减少乘法次数：将 $(ad+bc)$ 变成 $(a-c)(b-d)+ac+bd$，这样的话：$$ XY=ac2^n+[(a-c)(b-d)+ac+bd]2^{n/2}+bd $$ 变成了三个乘法，递归方程为：$$ T(n)= \left{ \begin{array}{ll} \Theta(1) & n=1 \ 3 T(n / 2)+\Theta(n) & n > 1 \end{array} \right. $$ 时间减少到了 $O(n^{\lg 3})$。

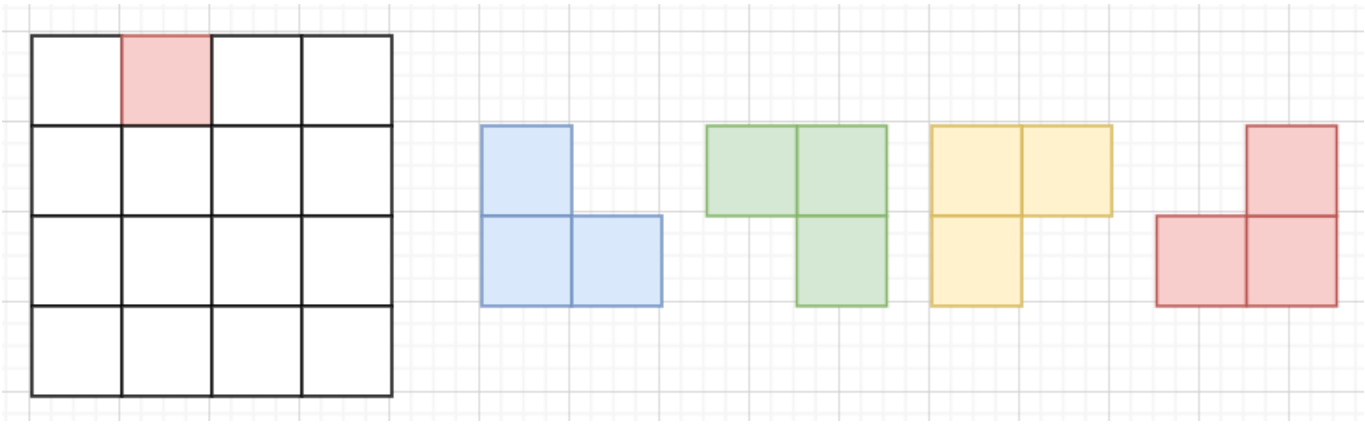
FFT 方法：快速傅里叶变换能将时间降到 $O(n \lg n)$。

2. Strassen 矩阵乘法

- 传统方法：$\Theta(n^3)$
- 分治：$\Theta(n^{\lg 7})=\Theta(n^{2.81})$

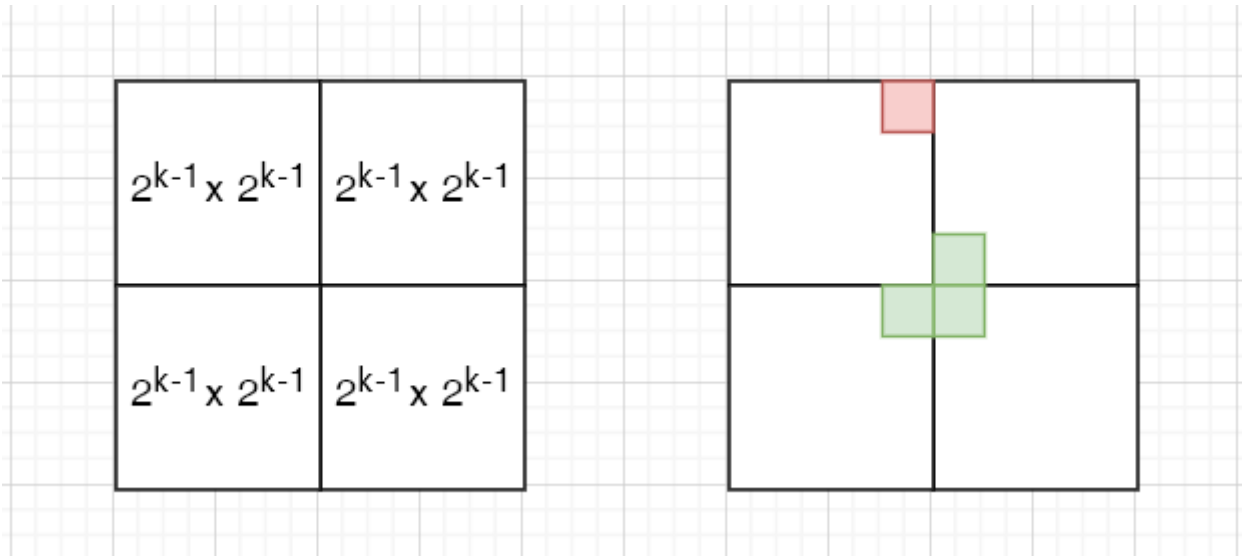
3. 棋盘覆盖

【问题】：如下图所示，要使用右边这四种 L形 填满左边的棋盘，棋盘中有一格(specific)不能被覆盖，剩余每一格只能被覆盖一次。



【解法】：

1. Partition original $2^k \times 2^k$ chessboard into four $2^{k-1} \times 2^{k-1}$ sub-chessboard, then **the specific** must be in one of the four sub-chessboard, and **the other three** have no specific.
2. Lay a L-shaped cards on the joint of **the three** sub-chessborad, then we get four smaller chessboard cover problem.
3. Do recursively until we get 1×1 chessboard.



递归方程为： $$ T(k)= \left{ \begin{array}{ll} \Theta(1) & k=0 \ 4T(k-1)+\Theta(1) & k>0 \end{array} \right. $$ 解为 $T(k)=\Theta(4^k)$。

4. 二叉搜索

Given n elements arranged in ascending order, find a particular element K.

Compare the middle element with the particular look up element X:

- if **X** is equal to **the middle element**, then the searching is successful and this algorithm is terminated;
- else if **X** is less than the **middle element**, continue the searching in the **first half** of the sequence;
- otherwise, continue the searching in the **second half** of the sequence.

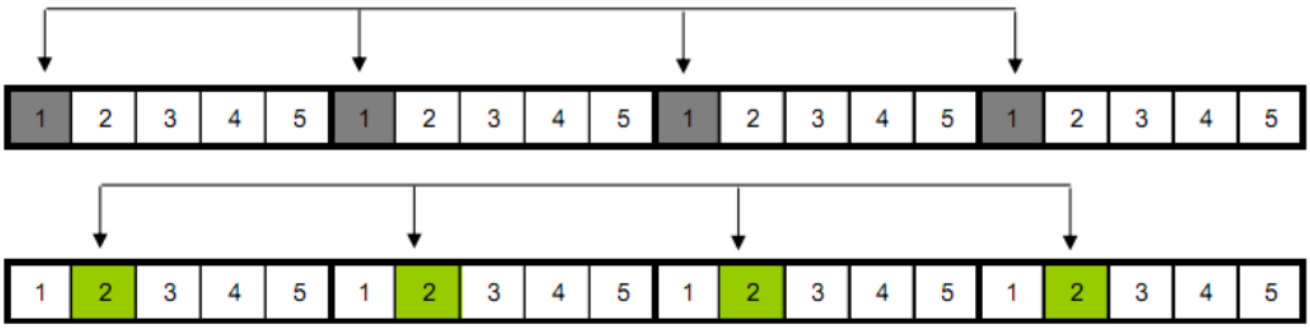
排序算法

希尔排序

Insertion sort is effective when:

Main idea: Shell sort allow elements to **move large steps**.

步长为 5 时，子序列的生成：

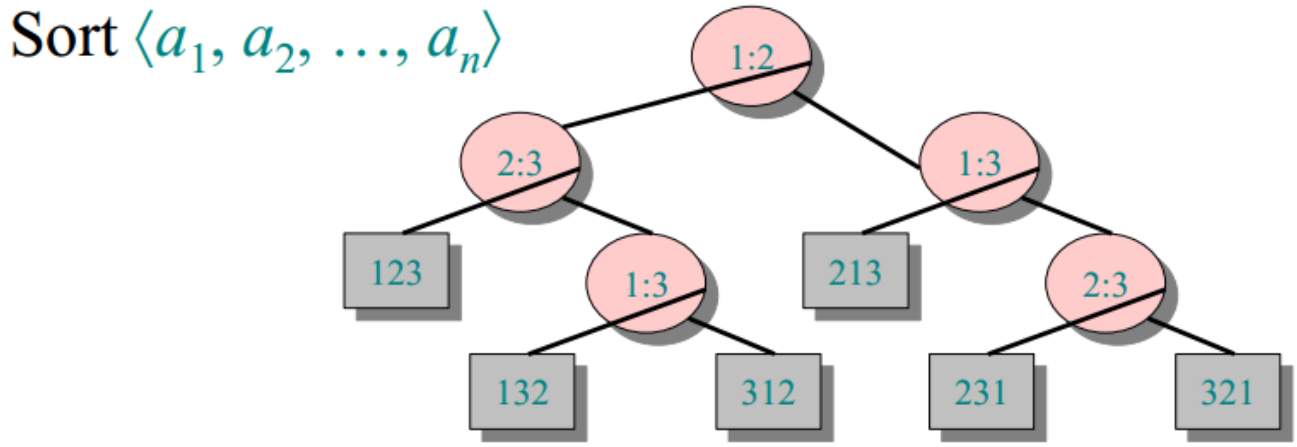


希尔排序实现：

```
void sort(Item a[], int sequence[], int start, int stop) {
    int step, i;
    for(step = 0; sequence[step] >= 1; step++) {
        int increase = sequence[step]; // 读出步长
        for(i = start + increase; i <= stop; i++) {
            // 循环处理每一个子序列
            int j = i;
            Item val = a[i];
            while(j >= i && val < a[j-increase]) {
                a[j] = a[j-increase];
                j -= increase;
            }
            a[j] = val;
        }
    }
}
```

决策树

可以利用决策树证明基于比较的排序算法的下界。



- 每个结点的左子树是 $a_i<a_j$ 时的结果；
- 右子树是 $a_i \ge a_j$ 时的结果。

决策树可以对所有比较排序建模：

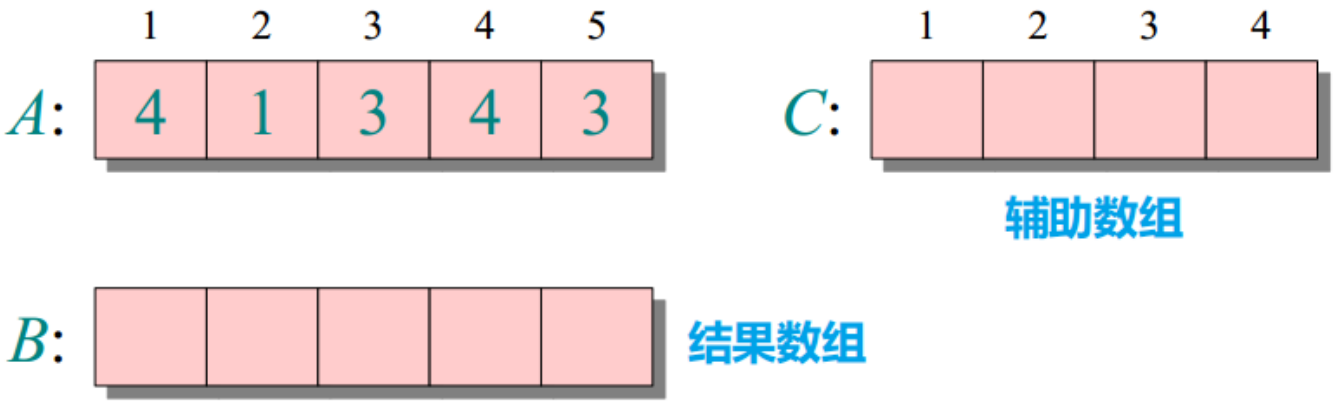
- 算法的执行时间 = 根到某一个叶结点的路径长度
- 最坏时间是树高

【定理】： Any decision tree that can sort n elements must have height $\Omega(n \lg n)$.

【证明】： The tree must contain $\ge n!$ leaves, since there are $n!$ possible permutations. A height-h binary tree has $\le 2^h$ leaves. Thus $n!\le 2^h$. Taking logarithms, we have $\lg (n!) \le h$. Stirling's approximation tells us $n!>(n/e)^n$, thus: $$ \begin{align} h & \ge \lg \Big(\frac{n}{e}\Big)^n \ &= n \lg n - n \lg e \[0.3em] &= \Omega(n \lg n) \end{align} $$

线性时间排序算法

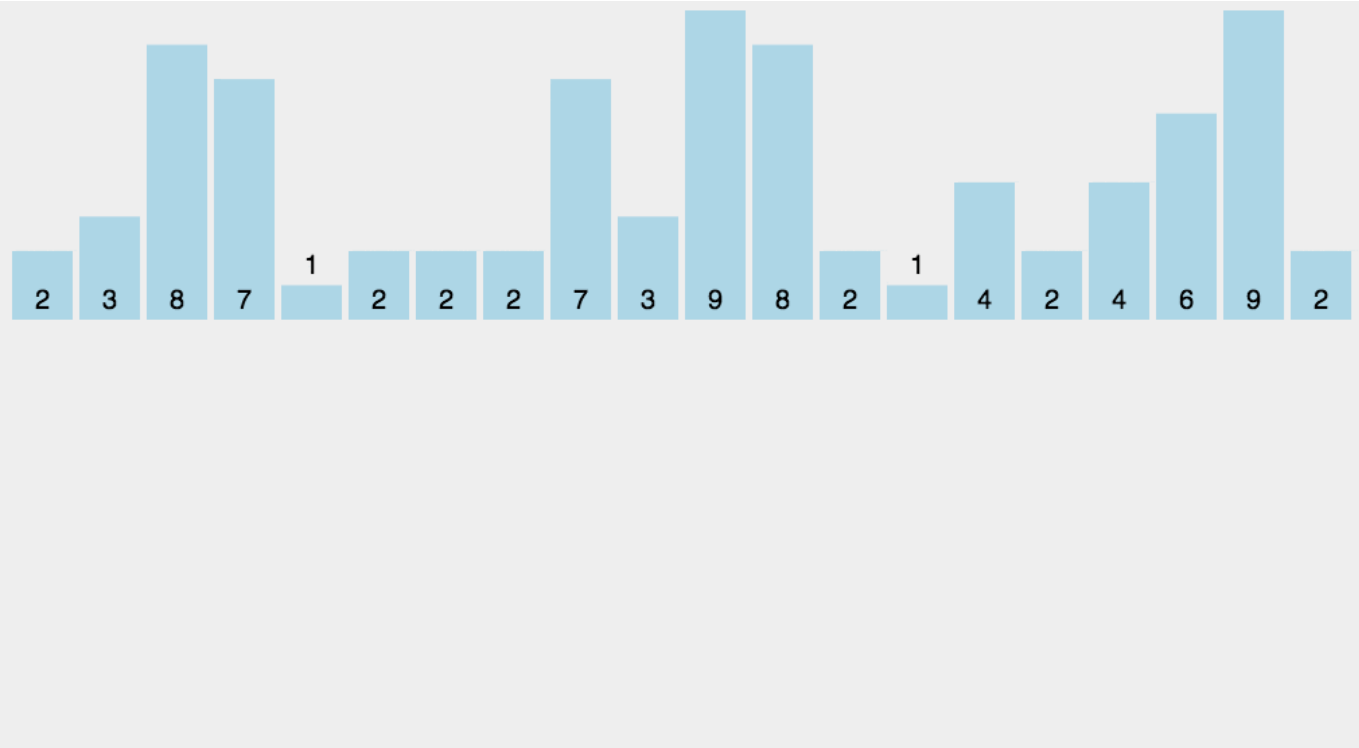
counting sort



> - $A[i] \in { 1,2,...,k }$ ， 例如此处 $k==4$。
> - 输入一共 n 个数，例如此处 $n==5$。

```
for i←1 to k
    do C[i]←0
for j←1 to n
    do C[A[j]]←C[A[j]]+1 // 记录 A 中每个数的个数
for i←2 to k
    do C[i]←C[i]+C[i−1] // 记录不大于该关键字的数字个数
for j←n down to 1 // 逆向填充，处理重复
    do B[C[A[j]]] ← A[j]
    C[A[j]] ← C[A[j]]−1
```

排序过程示例：



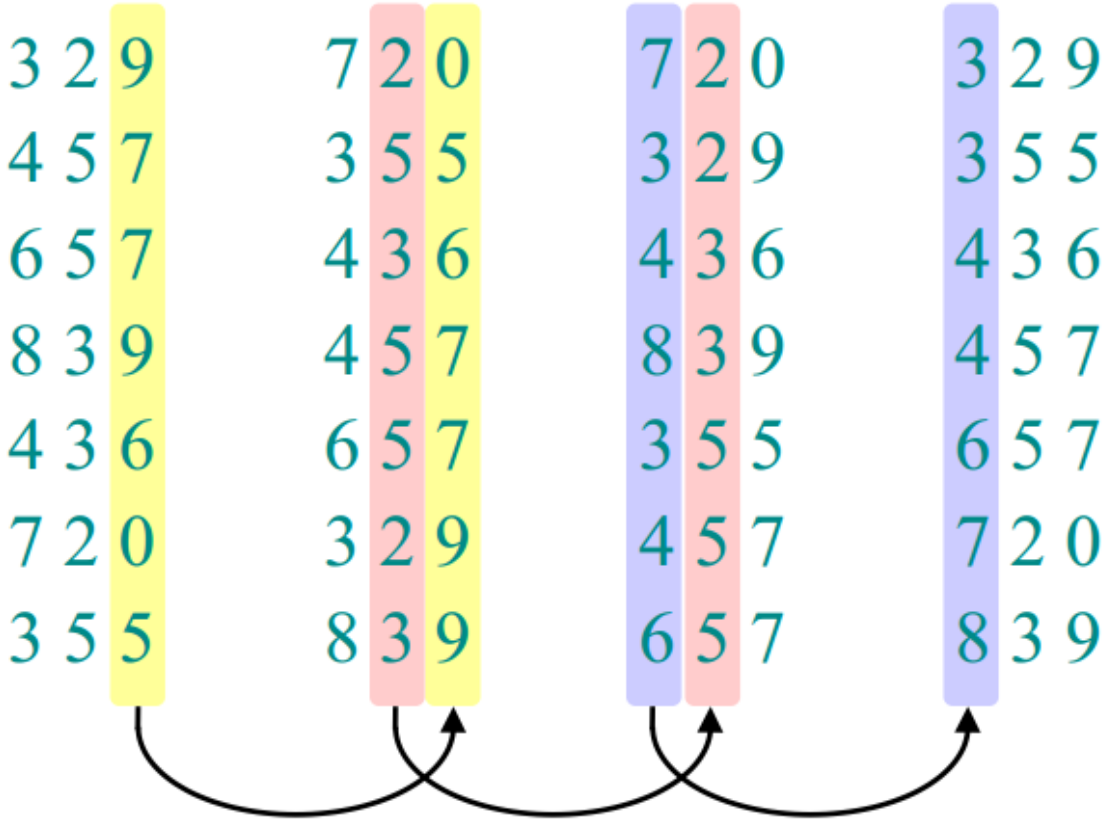
> 图片来源: https://www.runoob.com/w3cnote/counting-sort.html

时间复杂度：$\Theta(n+k)$，如果 $k=O(n)$，那么计数排序需要时间 $\Theta(n)$。

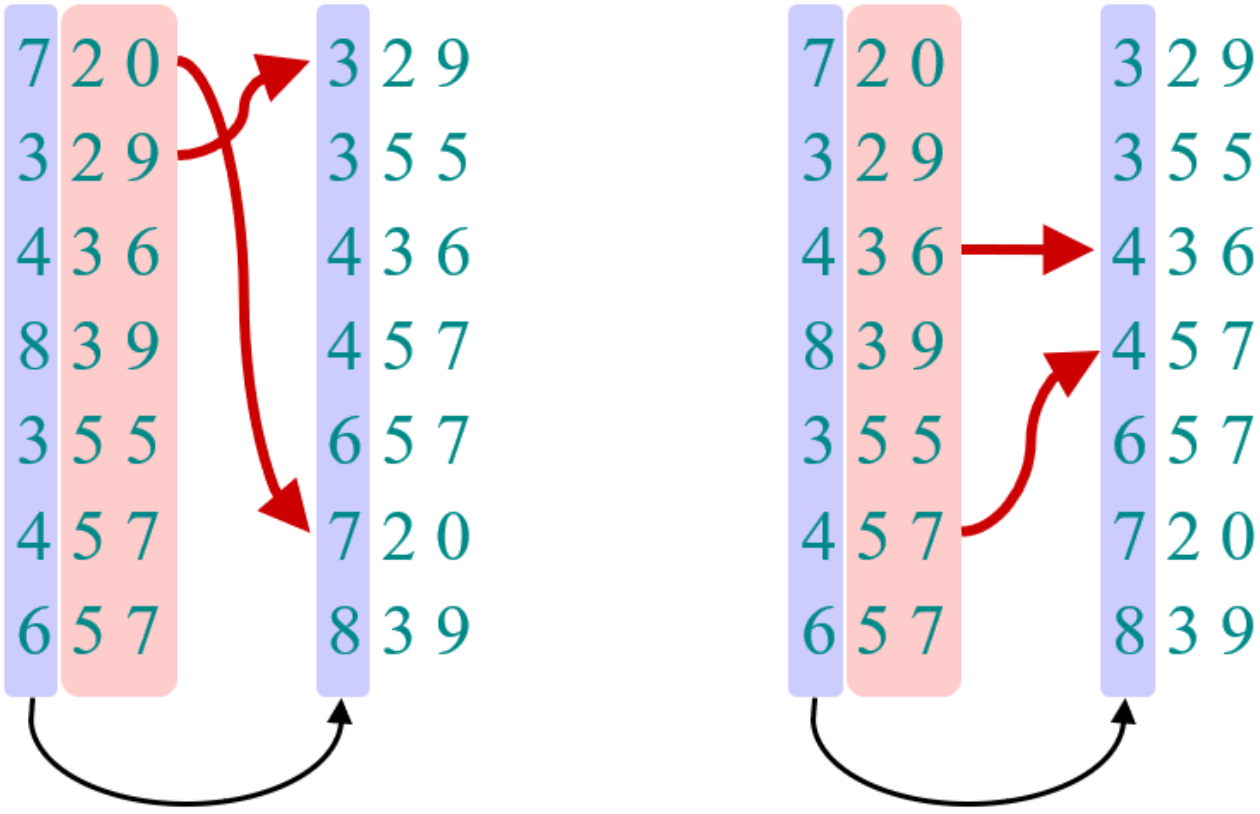
稳定性：**稳定。**

Radix Sort

Main idea： Sort on **least-significant digit first** with auxiliary **stable** sort.



基数排序的**正确性**归纳：



- Assume that the numbers are sorted by their lower-order $t-1$ digits.
- Sort on digit t:
    - Two numbers that differ in digit t are correctly sorted.

- Two numbers equal in digit t are put in the same order as the input, so we get correct order.

性能分析

Each word(b bits) can be viewed as having b/r base-2^r digits. For example, 32-bit word can be viewed as $4 \ \text{passes of counting sort on } \text{base}-2^8$ digits or $2 \ \text{passes of counting sort on } \text{base}-2^{16}$ digits.

But how many passes should we make?

计数排序处理 $0 \to (k-1)$ 范围的 n 个数需要时间为 $\Theta(n+k)$。所以如果将一个 b 位的数分成多个 r 位的部分时，每一部分表示的范围 $0 \to 2^r$，所以每一部分需要时间 $\Theta(n+2^r)$，b/r 个部分加在一起，得到： $$ T(n,b)=\Theta \Big(\frac{b}{r}(n+2^r) \Big) $$

- $2^r<n$ 即 $r<\lg n$ 时， 对于任何满足 $r\le b$ 的 r，$n+2^r=\Theta(n)$，则 $T(n,b)=\frac{b}{r}\Theta(n)$，当 $r=b$ 时，取到最小值 $\Theta(n)$；
- $r=\lg n$ 时，$T(n,b)=\Theta(bn /\lg n)$；
- $r>lg n$ 时，时间会以指数速度增长。

所以，当 $r=\lg n$ 时，取到最优解 $\Theta(bn/\lg n)$。通常情况下，$b=O{\lg n}$，所以让 $r \approx \lg n$，基数排序的时间可以达到 $\Theta(n)$。

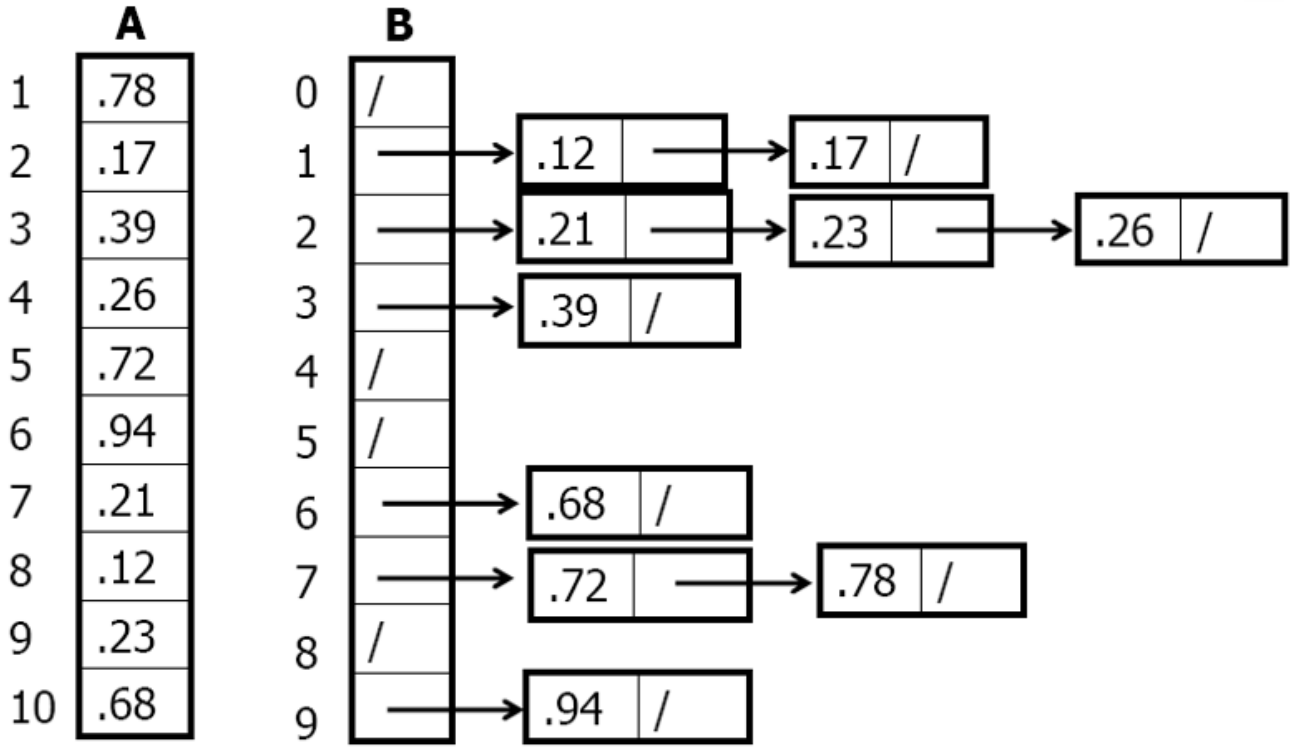
和比较排序的比较：

Which sorting algorithm we prefer depends on the characteristics of the **implementations**, of the **underlying machine**, and of the **input data**. Moreover, the version of radix sort that uses **counting sort** as the intermediate stable sort **does not sort in place**, which many of the $\Theta(n \lg n)$-time comparison sorts do. Thus , when primary memory storage is at a premium（主存容量比较宝贵时）, we might perfer an in-place algorithm such as quicksort.

Bucket Sort

算法思想：桶排序假设输入数据均匀分布在 [0, 1) 区间上，将这个区间划分为 n 个大小相同的子区间（bucket）；先将每个桶中的数排序，然后遍历每个桶就能得到结果。

在下面的桶排序代码中，假设输入是一个包含 n 个元素的数组 A，每个元素都在 0-1 之间，算法新建一个临时数组 B 存放链表（bucket）：




```
n = A>length
let B[1..n] be a new empty array
for i = 1 to n
    insert A[i] into B[nA[i].floor]
for i = 0 to n-1
    sort B with insertion sort
concatenate B[1],...,B[n] together in order
```

性能分析

假设 n_i 表示桶 $B[i]$ 中元素个数，因为使用插入排序处理每个桶中的子序列，所以桶排序时间为： $$ T(n)=\Theta(n)+\sum_{i=1}^n O(n_i^2) $$ 可以证明：$E[n_i^2]=2-1/n$，所以有 $T(n)=\Theta(n)$。

哈希表

解决冲突的两种方法：链接法和开放寻址法。

链接法（chaining）

在链接法中，把散列到同一个槽中的所有元素都放在一个链表中。

==**load factor**==

Given a hash table T with m slots that stores n elements, we define the **load factor** α for T as n/m, that is the **average number of elements** stored in a chain.

==**simple uniform hashing**==

We assume that any given element is equally likely to hash into any of the m slots, independently of where any other element has hashed to. We call this the assumption of **simple uniform hashing**.(简单均匀散列)

==**search cost**==

Theorem: In a hash table in which collisions are resolved by chaining, an **unsuccessful** search takes average-case time $\Theta(1+\alpha)$, under the assumption of simple uniform hashing.

Proof: Under the assumption of simple uniform hashing, any key k **not already stored** in the table is equally likely to hash to any of the m slots. The expected time to search **unsuccessfully** for a key k is the expected time to search to **the end of list** $T[h(k)]$, which has **expected length** $E[n_{h(k)}]=\alpha$. Thus, the expected number of elements examined in an unsuccessful search is α, and the total time required(including the time for computing h(k)) is $\Theta(1+\alpha)$.

Theorem: In a hash table in which collisions are resolved by chaining, a **successful** search takes average-case time $\Theta(1+\alpha)$, under the assumption of simple uniform hashing.

Proof: We assume that the element being searched for is equally likely to to any of the n elements stored in the table. The number of elements examined during a **successful** search for an element x is one more than the number of elements that appear before x in x's list. Because new elements are placed at the front of the list, elements before x in the list were all inserted after x was inserted. To find the expected number of elements examined, we take the average, over the n elements x in the table, of **1 plus the expected number of elements added to x's list after x was added to the list.** Let x_i denote the i-th element inserted into the table, for $i=1,2,...,n$, and let k_i denote the x_i's key. For keys k_i and k_j, we define the indicator random variable $X_{i j}=\mathrm{I}\left{h\left(k_{i}\right)=h\left(k_{j}\right)\right}$. Under the assumption of simple uniform hashing, we have $\operatorname{Pr}\left{h\left(k_{i}\right)=h\left(k_{j}\right)\right}=1 / m$, so $E[X_{ij}]=1/m$. Thus, the expected number of elements examined in a successful search is: $$ \begin{align} E\Big[\frac{1}{n} \sum_{i=1}^n \Big(1+\sum_{j=i+1}^n X_{ij} \Big) \Big]& = \frac{1}{n}\sum_{i=1}^n\Big(1+\sum_{j=i+1}^n \frac{1}{m} \Big) \[0.5em] &= 1+ \frac{1}{mn}\sum_{i=1}^n(n-i) \[0.5em] &= 1+\frac{n-1}{2m} \[0.5em] &= 1+\frac{\alpha}{2}-\frac{\alpha}{2n} \end{align} $$ Thus, the total time required for a successful search (including the time for computing the hash function) is $\Theta(1+1+\alpha/2-\alpha/2n)=\Theta(1+\alpha)$.

> 为了确定要检查元素的数目的期望，对 x 所在的链表，统计在 x 之后插入到表中的元素数（这些元素会放置在 x 之前），将其加 1 就得到了检查到 x 所需的查找次数。将所有表中 n 个元素的所需查找次数取均值，就得到了所求的最终期望，

What does this analysis mean? If the number of hash-table slots is at least proportional to the number of elements in the table, we have $n=O(m)$ and, consequently, $\alpha = n/m=O(m)/m=O(1)$. Thus, **searching takes constant time on average.**

散列函数

Division method

散列函数为：$h(k)=k \mod m$.。

m 取一个不太接近 2 的整数幂的素数。

For example, when $n=2000$, and the desired number of searched keys in an unsuccessful search is about 3. Then according to the following two conditions:

- approximately equals 2000/3
- a prime not close to any 2^r

We have $m=701$.

Multiplication method

假设 所有关键字都是整数，$m=2^r$， 计算机字长为 w，那么散列函数为： $$ h(k)=(A \cdot k \mod 2^w)rsh(w-r) $$ 其中：

- rsh 表示向右移位操作；
- $2^{w-1} < A < 2^w$，且 A 不能太接近两个端点。

开放寻址法（open addressing）

在开放寻址法中，所有元素都存放在散列表中（不像链接法将元素存放在链表中）。所以它的一个优点是不用存储指针，节省了空间。

稳了使用开放寻址法插入一个元素，需要连续地检查散列表，称为探查（**probe**），直到找到一个空槽为止。确定 **探查序列** 有几种常用方法：线性探查、二次探查、双重探查。

线性探查（Linear probing）

给定一个**辅助散列函数** $h'(k)$，线性探查使用的散列函数为： $$ h(k, i)=\left(h^{\prime}(k)+i\right) \bmod m, \quad i=0,1, \cdots, m-1 $$ Linear probing is easy to implement, but is suffers from a problem known as **primary clustering**. Long runs of occupied slots build up, increasing the average search time (随着连续被占用的槽不断增加，平均查找时间也随之不断增加). Clusters arise because an empty slot preceded by i full slots gets filled next with probability $(i+1)/m$ (一个空槽前面如果有 i 个连续被占用的槽，那么这个孔槽在下一步将被占用的概率为 $(i+1)/m$)，这就出现了聚簇问题. Long runs of occupied slots tend to get longer, and the average search time increases.

二次探查（Quadratic probing）

给定**辅助散列函数** $h'(k)$，和两个**辅助常数** c_1, c_2，二次探查使用的散列函数为： $$ h(k, i) = (h'(k)+c_1 i + c_2 i^2) \bmod m, \quad i=0,1,...,m-1 $$ If two keys have the same initial probe position, then their probe sequences are the same, since $h(k_1,0)=h(k_2,0)$ implies $h(k_1, i)=h(k_2,i)$. This property leads to a milder from of clustering, called **secondary clustering**.

双重散列（double hashing）

给定两个 **辅助散列函数** $h_1(k), h_2(k)$，双重散列使用的散列函数为： $$ h(k,i)=(h_1(k)+ih_2(k))\bmod m, \quad i=0,1,...,m-1 $$ 双重散列通常能产生很好的结果，但要求 $h_2(k)$ 与 m **互质**，为此可以令 m 为 2 的幂，同时让 $h_2(k)$ 只产生素数。

开放定址法探测次数分析

分析基于均匀散列假设（uniform hashing）：每一个关键字等可能地将 $m!$ 种排列中的一种作为它的探测序列。

Theorem: Given an open-addressed hash table with **load factor** $\alpha=n/m<1$, the expected number of probes in an **unsuccessful** search is at most $1/(1-\alpha)$.

Proof: In an **unsuccessful** search, every probe but the last **accesses an occupied slot** that **does not contain the desired key**, and **the last slot probed is empty**. Let us define the random variable X to be **the number of probes** made in an unsuccessful search, and event A_i, for $i=1,2,...$, to be the event that **an i-th probe occurs and it is to an occupied slot**. Then the event ${X \ge i}$ is the intersection of events $A_1 \cap A_2 \cap \cdots \cap A_{i-1}$ (探测 i 次才停，相当于从第一次到第 i-1 次探测都是已占用的). We will bound $Pr{X \ge i}$ by bounding $Pr{A_1 \cap A_2 \cap \cdots \cap A_{i-1}}$.(bound: 确定界限) $$ \begin{aligned} \operatorname{Pr}\left{A_{1} \cap A_{2} \cap \cdots \cap A_{i-1}\right}=& \operatorname{Pr}\left{A_{1}\right} \cdot \operatorname{Pr}\left{A_{2} \mid A_{1}\right} \cdot \operatorname{Pr}\left{A_{3} \mid A_{1} \cap A_{2}\right} \cdots \[0.5em] & \operatorname{Pr}\left{A_{i-1} \mid A_{1} \cap A_{2} \cap \cdots \cap A_{i-2}\right} \end{aligned} $$ Since there are n elements and m slots, $Pr{ A_1 }=n/m$. For $j>1$, the probability that there is a j-th probe and it is to an occupied slot, given that the first $j-1$ probes were to occupied slots, is $(n-(j-1))/(m-(j-1))$ （因为还有 $n-(j-1)$ 个未探查关键字和 $m-(j-1)$ 个未探查的槽）. Observing That $n<m$ implies that $(n-j)/(m-j)\le n/m$ for all j such that $0 \le j<m$, we have for all i such that $1\le i\le m$, $$ \begin{aligned} \operatorname{Pr}{X \geq i} &=\frac{n}{m} \cdot \frac{n-1}{m-1} \cdot \frac{n-2}{m-2} \cdots \frac{n-i+2}{m-i+2} \[0.5em] & \leq\left(\frac{n}{m}\right)^{i-1} \[0.5em] &=\alpha^{i-1} \end{aligned} $$ Thus, $$ {E}[X]=\sum_{i=1}^{\infty} \operatorname{Pr}{X \geq i} \leq \sum_{i=1}^{\infty} \alpha^{i-1} =\frac{1}{1-\alpha} $$

推论： 假设采用的是均匀散列，平均情况下，向一个装载因子为 α 的开放寻址散列表中插入一个元素至多需要做 $1/(1-\alpha)$ 次探测。

Theorem: Given an open-address hash table with load factor $\alpha < 1$, the expected number of probes in a successful search is at most $$ \frac{1}{\alpha} \ln \frac{1}{1-\alpha}. $$ **Proof**: A **search** for a key k reproduces the same probe sequence as with the element with key k was **inserted**. By the corollary above, if k was the $(i+1)$st key inserted into the hash table, the expected number of probes made in a search for k is at most $1 /(1-i / m)=m /(m-i)$. Averaging over all n keys in the hash table gives us the expected number of probes in a successful search: $$ \frac{1}{n} \sum_{i=0}^{n-1} \frac{m}{m-i}=\frac{1}{\alpha} \ln \frac{1}{1-\alpha}. $$

二叉查找树

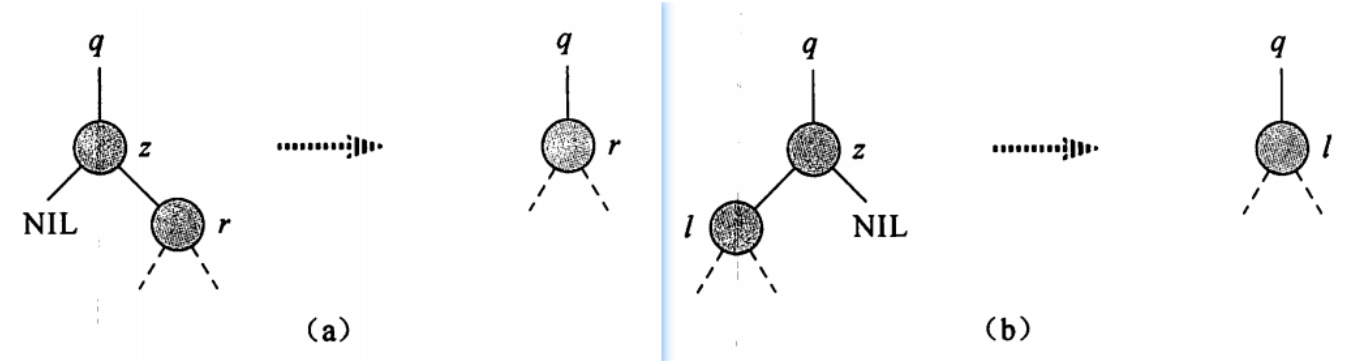
基本操作

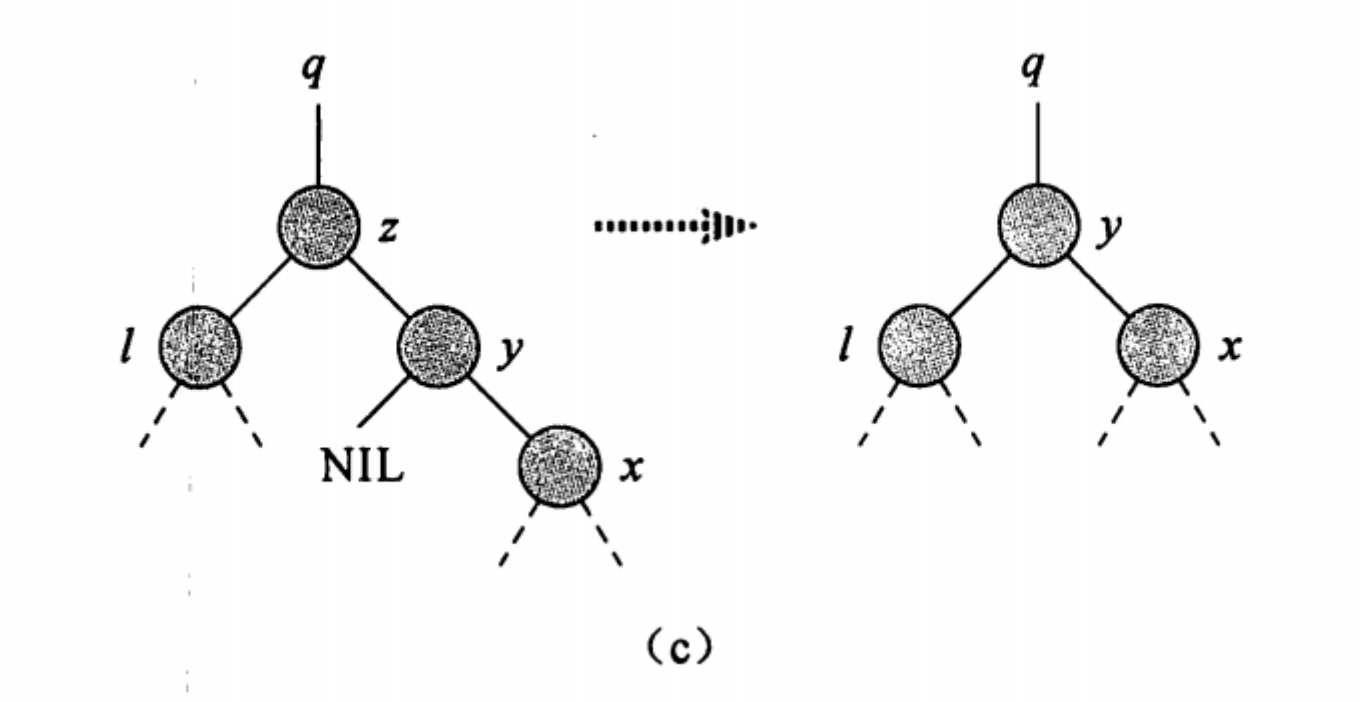
查找

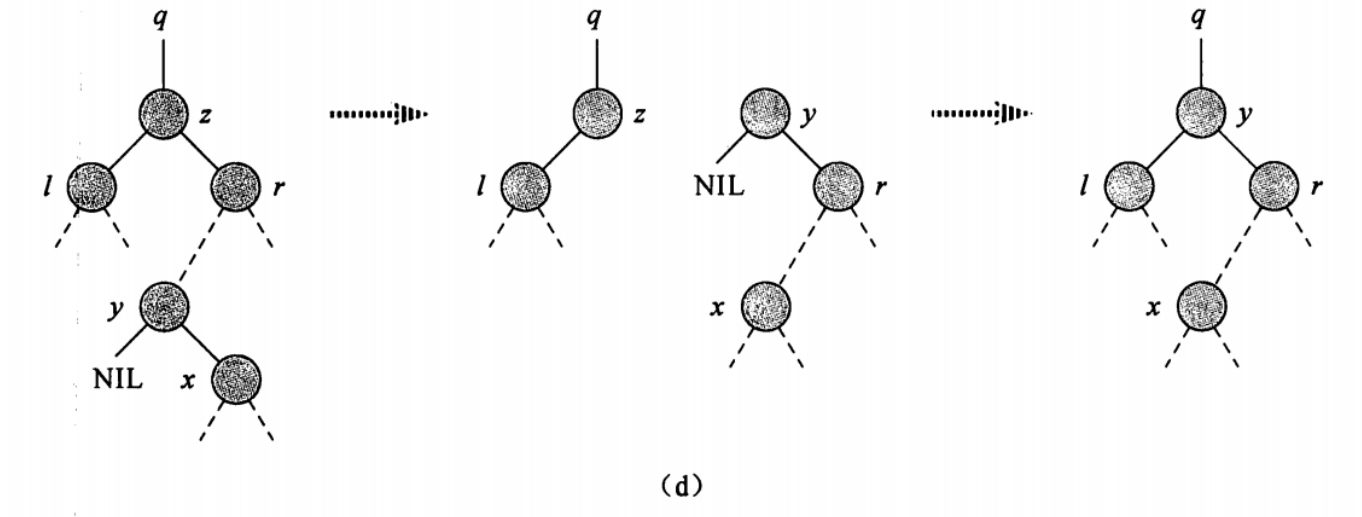
```
TreeSearch(x, k)
    while(x != NULL and k != x.key)
        if(k<x.key) x = x.left
        else x = x.right
    return x
```

插入：和查找过程一样，只是在最后 $x==NULL$ 时，将要插入的元素赋给 x。

删除：从一棵二叉搜索树中删除结点 z 有下面几种情况：







性能分析

一棵有 n 个不同关键字的随机构建二叉搜索树的期望高度为 $O(\lg n)$。

红黑树

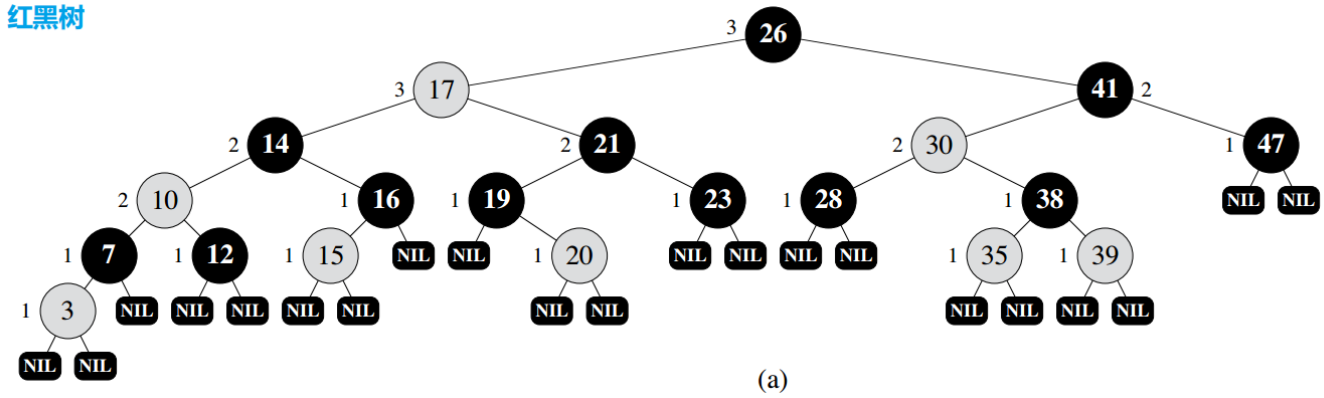
性质

红黑树是一棵 **二叉搜索树**，它在每个结点上增加一个存储位来表示结点的 **颜色**，可以是 RED 或 BLACK。通过对任何一条 **从根到叶子的简单路径** 上各个结点的颜色进行约束，红黑树确保 **没有一条路径会比其他路径长出两倍**，因而是近似于平衡的。

NIL 结点为 **叶结点**，带关键字的结点都是 **内部结点。**

五个属性：

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**.



黑高：从某个结点出发（不含该结点）到达任意一个叶结点的简单路径上的黑色结点个数称为该结点的 **黑高(black-height)**，红黑树的树高就是根结点的黑高。

引理：一棵有 n 个内部结点的红黑树的高度至多为 $2 \lg (n+1)$。

证明：We start by showing that **the subtree rooted at any node x contains at least $2^{bh(x)}-1$ internal nodes**. We prove this claim by induction on the height of x.

If the height of x is 0, then x must be a **leaf** (NIL), and the subtree rooted at x indeed contains at least $2^{bh(x)}-1=0$ internal nodes. (全为黑结点的满二叉树). For the inductive step, consider a node x that has positive height and is an internal node **with two children**. Each child has a **black-height** of either $bh(x)$ or $bh(x)-1$, depending on whether its color is red or black, respectively. Since the height of a child of x is less than the height of x itself, we can **apply the inductive hypothesis** to conclude that each child has at least $2^{bh(x)-1}-1$ interval nodes. Thus, the subtree rooted at x contains at least $(2^{bh(x)-1}-1)+(2^{bh(x)-1}-1)+1=2^{bh(x)}-1$ internal nodes, which proves the claim.

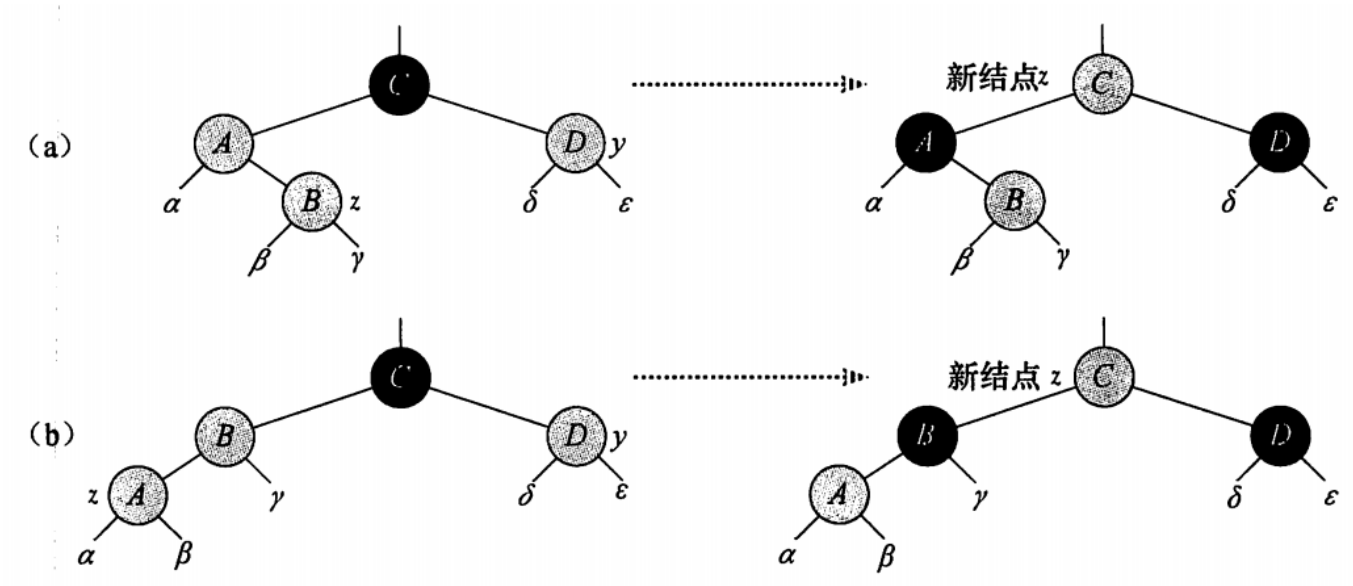
To complete the proof of the lemma, let h be the height of the tree. According to property 4, **at least half the nodes on any simple path from the root to a leaf, not including the root, must be black.** Consequently, the black-height of the root must be at least $h/2$; thus, $n \ge 2^{h/2}-1$. Moving the 1 to the left-hand side and taking logarithms on both sides yields $\lg(n+1) \ge h/2$, or $h \le 2 \lg(n+1)$.

结点插入

> 性质 4：红色结点的子结点必须是黑色结点。
>
> 性质 5：从某个结点开始，到它每个后代叶结点的路径上，黑色结点数量都相同。

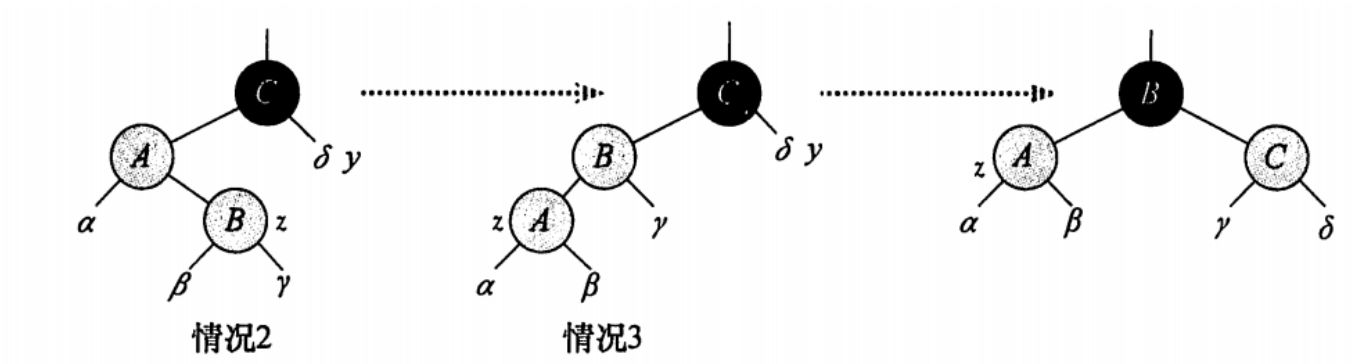
新插入的结点 z 的初始颜色都是红色，如果破坏了红黑树性质，那么对整棵树进行相应的结构和颜色调整。分成下面几种情况：

情况 1：



新插入的结点 z 和父结点都是红色的，z 的叔结点也是好红色的。所以性质 4 被违反，需要 **将父结点 $z.p$ 和叔结点 y 都变成黑色，将爷爷结点 $z.p.p$ 着为红色** 以保持性质 5。 然后将爷爷结点作为 z 继续这个循环。

情况 2 和 3：



z 和 $z.p$ 都是红色的，z 的叔结点是黑色的。性质 4 被违反。情况 2 做一次 **左旋转** 能变成情况 3，接着做一次 **右旋转** 得到满足各个性质的结果。

实现：

```
tbInsert(x)
    treeInsert(x);
    x.color = RED;
    while(x!=root && x.parent.color==RED)
        if(x.parent==x.parent.left) // x 是左子树
```

```
            y = x.uncle
            if(y.color==RED) // 情况 1
                x.parent.color=BLACK;
                y.color=BLACK;
                x.grandParent.color=RED;
                x=x.grandParent;
            else // 情况 2 和 3
                if(x==x.parent.right)
                    x=x.parent;
                    leftRotate(x);
                x.parent.color=BLACK;
                x.grandParent.color=RED;
                rightRotate(x.grandParent);
        else
            same as above, but with "right" & "left" exchanged
```

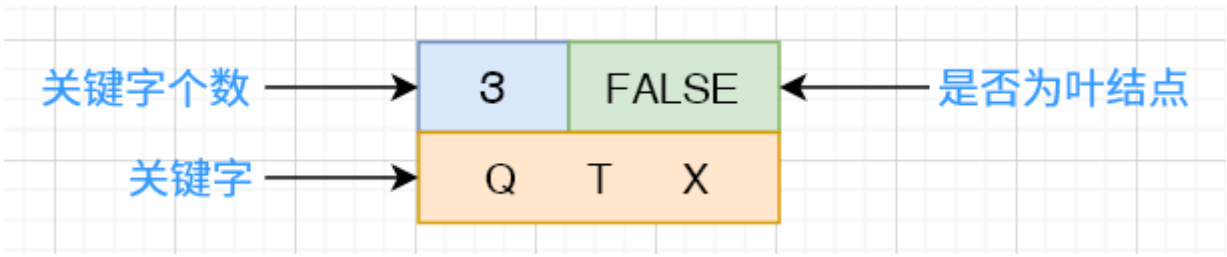
插入操作的复杂度为 $O(\lg n)$。

B 树

性质

一棵 B 树是具有以下性质的有根树：

1. 每个结点 x 有下面属性：



   - $x.n$ ——当前存储在结点 x 中的关键字个数。
   - $x.n$ 个关键字本身，以非降序存放。
   - $x.leaf$ ——一个布尔值，表示该结点是否为叶结点。

2. 每个内部结点 x 还包含 $x.n+1$ 个指向其孩子的指针 $x.c_1, x.c_2,...,x.c_{n+1}$。

3. 关键字 $x.key_i$ 对存储在各子树中的关键字范围加以分割。

4. 每个叶结点具有相同的深度，即树的高度。

5. 每个结点所包含的关键字个数有 **上界** 和 **下界**，用一个被称为 B 树的 **最小度数** 的固定整数 $t \ge 2$ 来表示这些界：

   - 除了根结点以外的每个结点必须至少有 **$t-1$ 个关键字**。因此，除了根结点以外的每个内部结点 **至少有 t 个孩子**，入股树非空，根结点至少有一个关键字。
   - 每个结点 **最多包含 $2t-1$ 个关键字**，即最多有 $2t$ 个孩子。

   $t=2$ 时的 B 树最简单，每个内部结点有 2、3、或 4 个孩子，称为 **2-3-4 树**。

定理：如果 $n\ge 1$，那么对任意一棵包含 n 个关键字、高度为 h、最小度数为 $t \ge 2$ 的 B 树 T，有 $$ h \le \log_t \frac{n+1}{2} $$ **证明**：可以利用关键字总数最小的情况进行证明。根结点至少有一个关键字，其他每个结点至少包含 $t-1$ 个关键字，因此，高度为 h 的 B 树在深度 1 （第二层）至少包含 2 个结点，在深度 2 至少包含 $2t$ 个结点，在深度 h 至少有 $2t^{h-1}$ 个结点。所以，关键字个数 n 满足不等式： $$ n \ge 1+(t-1)\sum_{i=1}^h 2t^{i-1}=2t^h-1 $$ 经过简单的变换就能证明该定理。

基本操作

查找

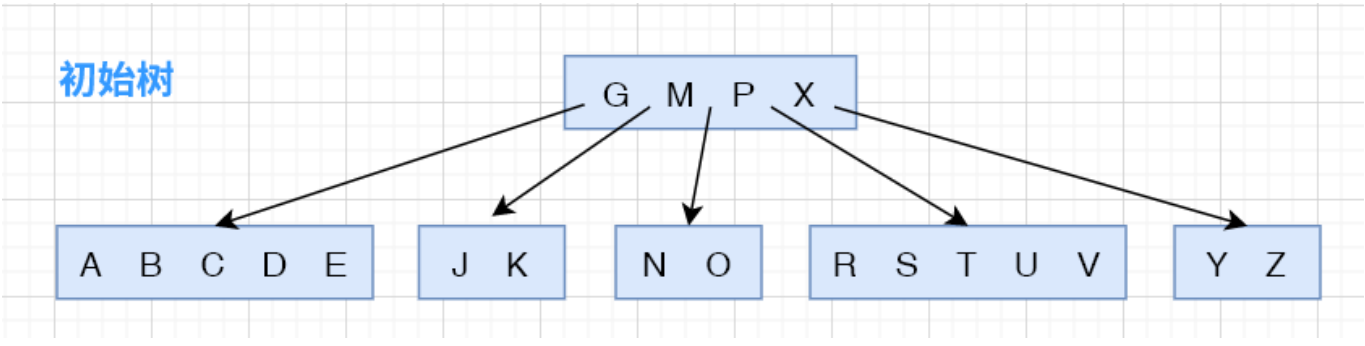
```
B-TREE-SEARCH(x, k)
    i=1
    while i <= x.n && k > x.key
        i = i+1
    if i <= x.n && k == x.key
        return (x,i)
    else if x.leaf
        return NIL
    else DISK-READ(x,ci)
        reutrn B-TREE-SEARCH(x.ci, k)
```

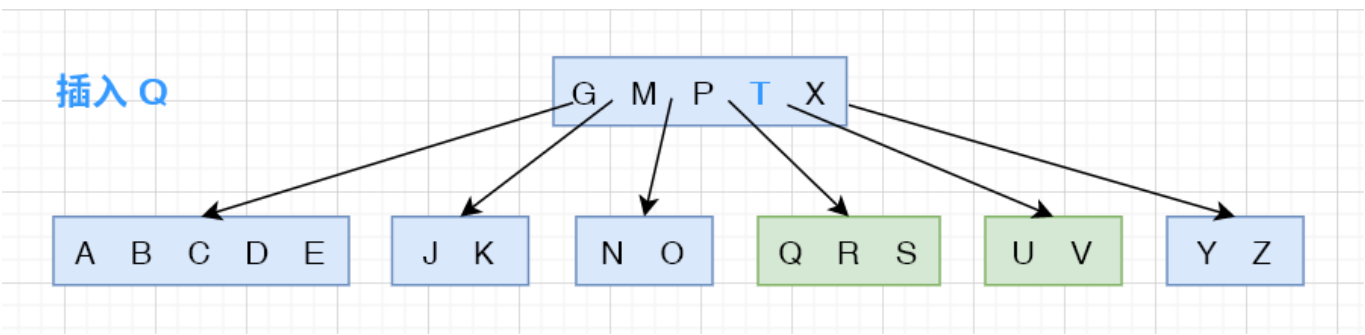
时间：磁盘操作 $O(\log_t n)$，总时间 $O(t\log _t n)$。

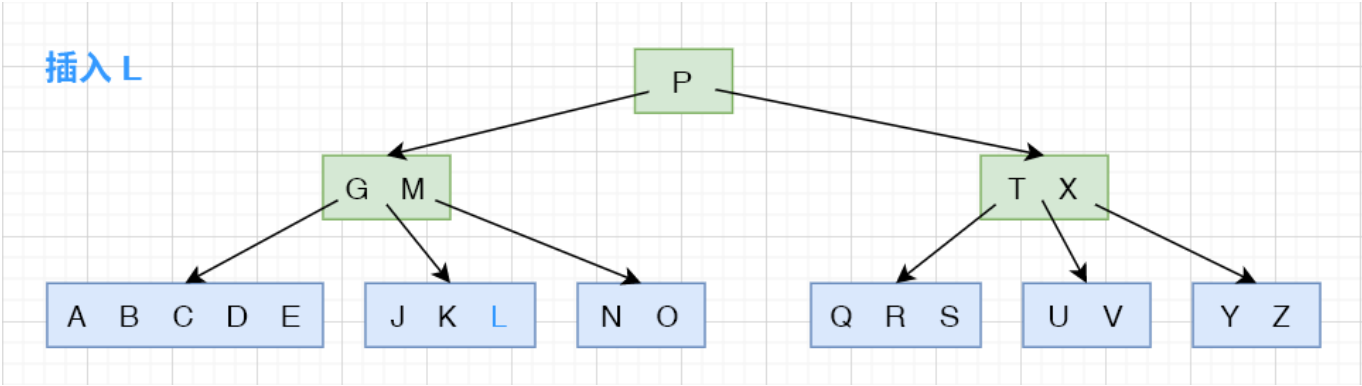
插入关键字

由于不能将关键字插入一个满的叶结点，故引入一个操作，将一个满的结点 y（有 $2t-1$ 个关键字）按其中间关键字分裂为两个各含 $t-1$ 个关键字的结点，中间关键字被提升到 y 的父结点。如果 y 的父结点也是满的，就必须在插入新结点之前将其分裂。

> 不是等到找出插入过程中实际要分裂的满结点时才做分裂，而是 **在沿着树往下查找新的关键字所属位置时，就分裂沿途遇到的每个满结点**。这样每当要分裂一个满结点时，都能确保它的父结点不是满的。







分裂操作的时间：磁盘操作 $O(1)$，CPU 时间 $\Theta(t)$。

数据结构的扩张

顺序统计

顺序统计树只是简单地在每个结点上存储了附加信息的一棵 **红黑树**，在结点 x 中，除了通常属性之外，还包括一个 $x.size$，表示以 x 结点为根的子树的结点数（**包括 x 本身**）。

一个元素的 **秩** 定义为在中序遍历时输出的位置。

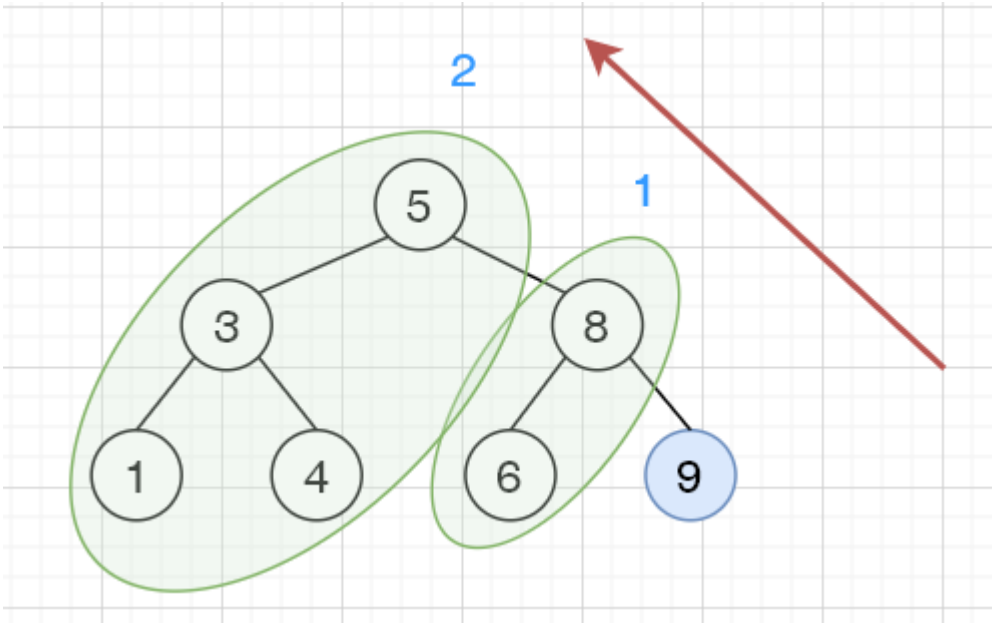
查找给定秩的元素

```
OS-SELECT(x,i)
    r=x.left.size+1
    if i==r
        return x
    else if i < r
        return OS-SELECT(x.left, i)
    else return OS-SELECT(x.right, i-r)
```

确定一个元素的秩

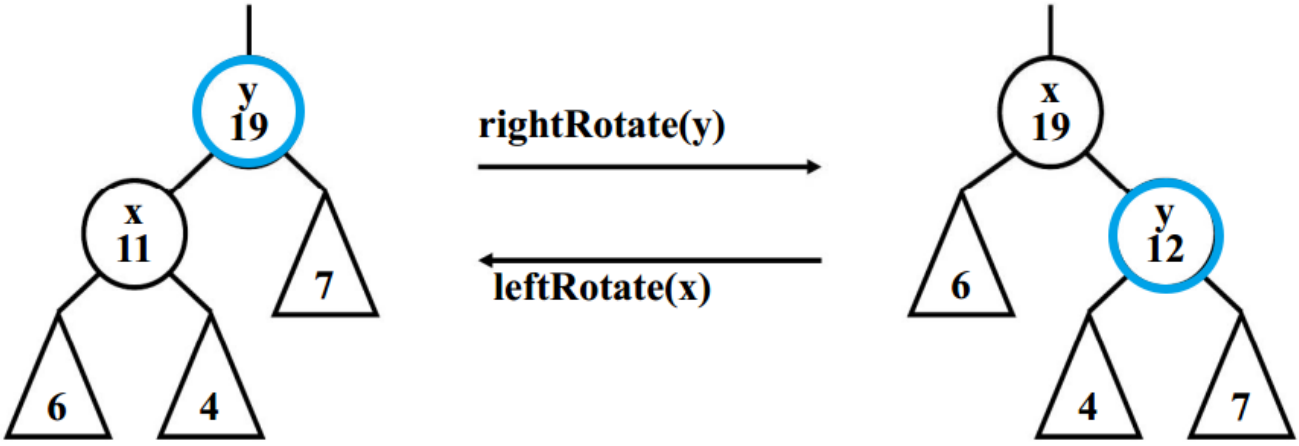
```
OS-RANK(T,x)
    r = x.left.size + 1
    y = x
    while y != T.root
        if y == y.p.right // 结点 y 是右孩子
            r = r + y.p.left.size + 1
        y = y.p
    return r
```

我们可以认为结点的秩等于中序遍历时排在该结点之前的结点总数加上 1，代码中的 `r=r+y.p.left.size+1` 就表示循环向上层统计左孩子的结点总数，如下图所示：



查找给定秩的元素、确定元素的秩这两个操作的时间都和树高成正比，即 $O(\log n)$。

树结构的维护——旋转



扩张数据结构的方法

扩展一种数据结构一般分成 4 步：

1. 选择一种**基础数据结构**；
2. 确定基础数据结构中要维护的**附加信息**；
3. 检验基础数据结构上的基本修改操作能否**维护附加信息**；
4. 设计一些**新操作**。

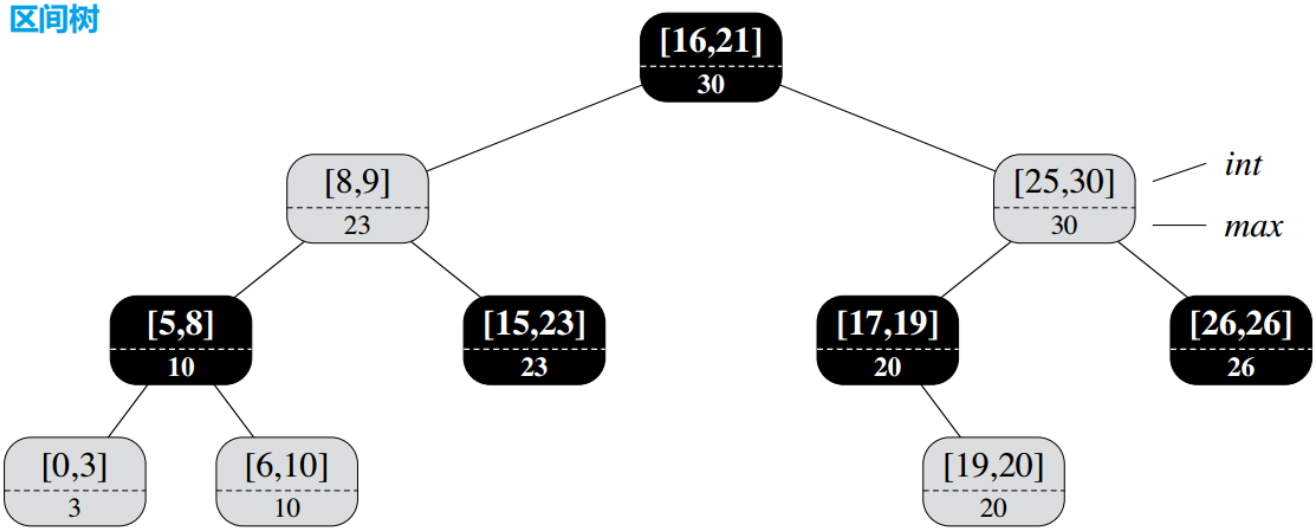
区间树

我们可以把一个区间 $[t_1,t_2]$ 表示成一个对象 i，包括三个属性—— $low, high,max$，其中 $i.low=t_1$ 表示低端点，$i.high=t_2$ 表示高端点，$i.max$ 表示子结点中区间端点最大值。如果两个区间有交叉，则称这两个对象**重叠**。任何两个区间 i,i' 都满足区间三分律（**interval trichotomy**）：

- i and i' overlap,
- i is to the left of i',

- i is to the right of i'.

区间树



在区间树中，**每个结点表示一个区间**，显示在虚线上方；一个以该结点为根的子树中包含的 **区间端点的最大值**，显示在虚线下方。

4 步法

步骤一：基础数据结构。选择这样一棵红黑树，每个结点 x 包含一个区间属性 $x.int$，且 x 的关键字为区间的**低端点** $x.int.low$。那么该树的中序遍历列出的就是 **按低端点排列** 的各区间。

步骤二：附加信息。每个结点还包含一个值 $x.max$，表示以 x 为根的子树中所有区间的端点的最大值。

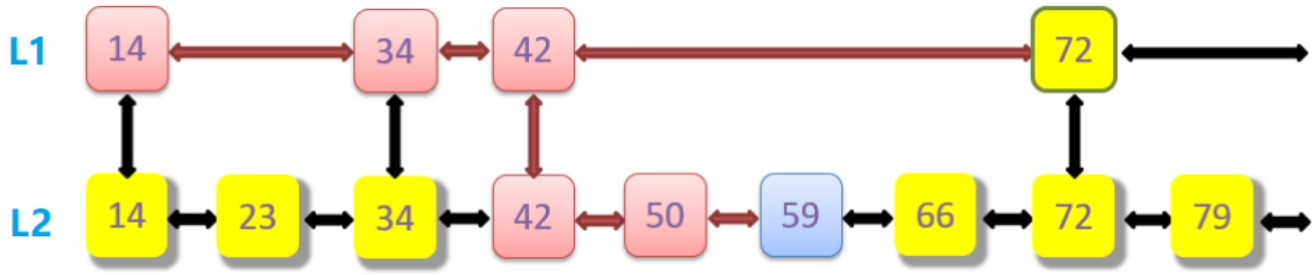
步骤三：对信息的维护。必须验证 n 个结点的区间树上的插入和删除操作能否在 $O(\lg n)$ 时间内完成。$x.max=\max(x.int.high,\ x.left.max,\ x.right.max)$。

步骤四：设计新的操作。`INTERVAL-SEARCH(T,i)`，用来 **找出树 T 中与区间 i 重叠的那个结点。**

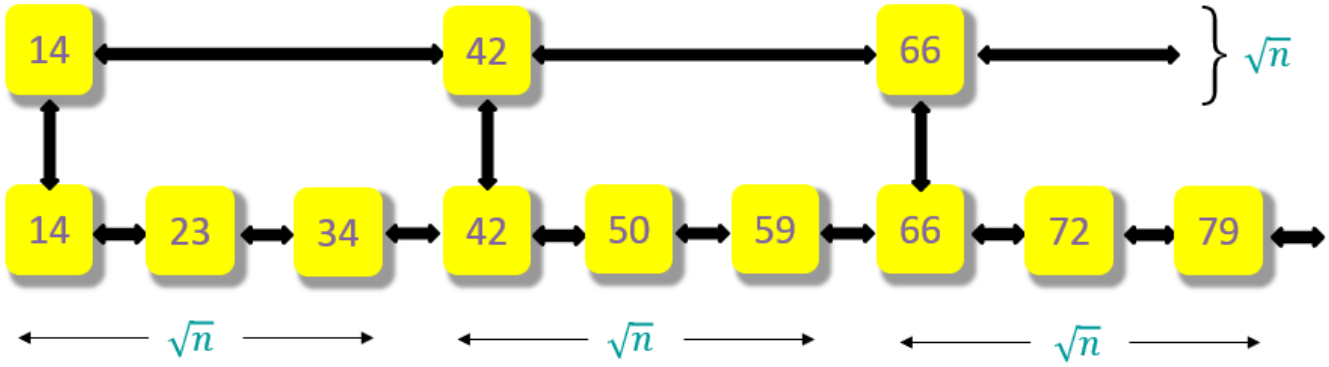
```
INTERVAL-SEARCH(T,i)
    x=T.root
    while x != T.nil and i does not overlap x.int
        if x.left != T.nil and x.left.max >= i.low
            x = x.left
        else
            x = x.right
    return x
```

这个过程从树根开始，逐步向下搜索，当找到一个重叠区间或者叶结点时过程结束。由于基本循环的每次迭代耗费 $O(1)$ 时间，又因为 n 个结点的红黑树高度为 $O(\lg n)$，所以这个过程耗费 $O(\lg n)$ 时间。

跳表



搜索代价是 $|L_1|+\frac{|L_2|}{|L_1|}$，设原表 L_2 的长度为 n，则当 $L_1=\sqrt{n}$ 时代价最小，这时 $$ |L_1|+\frac{|L_2|}{|L_1|}=\sqrt{n}+\frac{n}{\sqrt{n}}=2\sqrt{n} $$



如果有三个表，则代价为 $3 \cdot \sqrt[3]{n}$，有 k 个表时，代价为 $k \cdot \sqrt[k]{n}$。

当有 $\lg n$ 个表时，代价为 $\lg n \cdot \sqrt[\lg n]n=2\lg n$。这是理想的跳表。

> 对数公式：
>
> - 换底：$\log_a x=\frac{\log_b x}{\log_b a}$
> - 次方：$\log_{a^n}{x^m}=\frac{m}{n}\log_a x$
> - 还原：$a^{\log_a x}=x$
> - 互换：$M^{\log_a N}=N^{\log_a M}$

With-high-probability theorem: With high probability, every search in an n-element skip list costs $O(\lg n)$。 (Event E occurs with high probability if, for any $\alpha \ge 1$, there is an appropriate choice of constants for which E occurs with probability at least $1-O(1/n^\alpha)$.)

Boole's inequality/union bound: For any random events $E_1, E_2,...,E_k$, $$ Pr{ E_1 \cup E_2 \cup \cdots \cup E_k } \le Pr{E_1} + Pr{ E_2 }+\cdots+Pr{ E_k } $$ **Lemma**: With high probability, n-element skip list has $O(\lg n)$ levels.

Proof of lemma: Let **error probability** for having at most $c \lg n$ levels be P, then

$$ \begin{align} P &= Pr{ \text{more than } c\lg n \text{levels} } \[0.5em] &\le n \cdot Pr{ \text{element x promoted at least } c\lg n \text{times} } \[0.5em] &= n \cdot (1/2)^{c \lg n} \[0.5em] &= n \cdot (1/n^c) \[0.5em] &= 1/n^{c-1} \end{align} $$

> 每个元素被提升到上一层的概率为 1/2.

平摊分析

三种常用方法：

- the **aggregate** method（聚合分析）
- the **accounting** method（记账法）
- the **potential** method（势能法）

The aggregate method. though simple, **lacks the precision** of the other two methods. In particular, the accounting and potential methods allow a specific **amortized cost** to be allocated to each operation.

聚合分析

以栈操作为例，假设只有三种操作：`POP, PUSH, MULTIPOP(S,k)`，最后一个表示连续弹出 `k` 个对象，如果栈中少于 `k` 个元素，就将其全部弹出。`POP, PUSH` 代价为 1，`MULTIPOP(S,k)` 代价为 $\min(s.length,k)$。

接下来分析一个由 n 个 `PUSH, POP, MULTIPOP` 组成的操作序列在一个空栈上的执行情况。

首先单独分析每个操作：序列中一个 `MULTIPOP` 操作的最坏情况代价为 $O(n)$，因此，一个 n 个操作的序列的最坏情况代价为 $O(n^2)$，因为序列可能包含 $O(n)$ 个 `MULTIPOP` 操作。但这样得到的最坏情况并不是一个确界。

然后使用聚合分析，考虑整个序列的操作，可以得到更好的上界。对于一个非空的栈，可以执行的 `POP` 次数（包括 `MULTIPOP` 中的 `POP`）最多与 `PUSH` 操作的次数相当，即最多 n 次。因此，任意一个操作序列最多花费时间为 $O(n)$，那么单独一个操作的平均时间为 $O(n)/n=O(1)$。在聚合分析中，我们将每个操作的平摊代价设定为平均代价，此处三种操作的平均代价都是 $O(1)$。

记账法

对不同操作赋予不同费用，费用可能多于或者少于其真实代价。我们将这个费用称为 **平摊代价**。当一个操作的平摊代价超出其实际代价时，我们将差额存入数据结构中的特定对象，存入的差额称为**信用**。对于 **后续操作** 中平摊代价小于实际代价的情况，信用可以用来支付差额（ 信用是基于每个**元素**的，而不是每个操作的，发生在同一个元素上的不同操作，可以共用信用）。因此，我们可以将一个操作的平摊代价分解为其实际代价和信用。

例如在上面的栈操作中，为 `PUSH` 赋予平摊代价 2，其余操作 0。我们 **用 1 美元支付压栈操作的实际代价，剩余的 1 美元存为信用**（共缴费 2 美元）。在任何时间点，栈中的每个元素都存储了与之对应的 1 美元信用，作为它将来被弹出时支付的实际代价（包括 `POP` 操作和 `MULTIPOP` 操作）。因此，对于任意 n 个上述三种操作组成的序列，总的平摊代价 $O(n)$ 总为实际代价的上界。

另一个例子：**数组的动态扩展**。动态数组在要插入新元素而空间不够时，会将空间扩大一倍，并将原有元素赋值到新数组中（这时除了插入新元素的 1 美元代价，还有复制原有 m 个元素的 m 美元代价），扩展空间的时机是插入第 2、3、5 ..... 2^k+1 个元素时，其对应的代价分别为 2、3、5 ..... 2^k+1 美元，其余元素的插入代价都是 1 美元。除了第一个元素收取的平摊代价为 2 美元外，其他元素的平摊代价都为 3 美元，这样在每次扩展空间时，存下来的代价(bank)都刚好够用来复制旧元素（每次扩展空间之后都会将存下来的代价用光，紧接着的下一个元素插入之后，就又存下来 2 美元的代价，然后紧接着的下一步如果不需要扩展，就又会存下来 2 美元，这样总共存下来 4 美元，以此类推，bank 中每次增加 2 美元，直到下一次扩展将这些存款用光。）

i	1	2	3	4	5	6	7	8	9	10
$size_i$	1	2	4	4	8	8	8	8	16	16
c_i	1	2	3	1	5	1	1	1	9	1
$\hat{c}_i$	2*	3	3	3	3	3	3	3	3	3
$bank_i$	1	2	2	4	2	4	6	8	2	4

- i 表示插入第几个元素
- $size_i$ 表示插入第 i 个元素时的数组空间大小
- c_i 表示每一步的实际代价
- $\hat{c_i}$ 表示平摊代价
- $bank_i$ 表示每一步完成后的存款

因为 $3n = \sum_{i=1}^{n}\hat{c_i} \ge \sum_{i=1}^n c_i$，所以总的代价为 $O(n)$。

势能法

势能法并不将预付代价表示为数据结构中特定对象的信用，而是表示为势能，将势能与 **整个数据结构** 而不是 **特定对象** 相关联。

势函数 Φ 将每个数据结构 D_i 映射到一个实数 $\Phi(D_i)$，此值即为关联到数据结构 D_i 的势能。第 i 个操作的平摊代价 $\hat{c_i}$ 用势函数定义为： $$ \hat{c_i}=c_i+\Phi(D_i)-\Phi(D_{i-1}) $$ 那么 n 个操作总的平摊代价为： $$ \sum_{i=1}^n \hat{c}*i*=*sum*{i=1}^n c_i + \Phi(D_n)-\Phi(D_0) $$ 如果能定义一个势函数 Φ 使得 $\Phi(D_n)\ge \Phi(D_0)$，则总的平摊代价 $\sum_{i=1}^n \hat{c}*i$ 给出了总的实际代价 $*sum*{i=1}^n c_i$ 的一个上界。通常将 $\Phi(D_0)$ 定义为 0，然后说明对所有 i，有 $\Phi(D_i)\ge 0$。

延续前面的栈操作例子，将一个栈的 **势函数定义为其中的对象数量**，那么 $\Phi(D_0)=0,\ \Phi(D_i)\ge 0$，所以用 Φ 定义的 n 个操作的总平摊代价即为实际代价的一个上界。下面计算不同操作的平摊代价。

- 如果第 i 个操作是压栈操作，此时栈中包含 s 个对象，则势差为：$\Phi(D_i)-\Phi(D_{i-1})=s+1-s=1$，其实际代价为 1，则压栈操作的平摊代价为 $\hat{c_i}=c_i+\Delta{\Phi}=1+1=2$。
- 如果第 i 个操作是`MULTIPOP(S,k)` 操作，设 $k'=\min(k,s)$，则实际代价为 k'，势差为 $\Phi(D_i)-\Phi(D_{i-1})=-k'$，因此平摊代价为 $\hat{c_i}=c_i+\Delta{\Phi}=k'-k'=0$。
- 类似地，普通出栈操作的平摊代价也是 0。

每个操作的平摊代价都是 $O(1)$，所以总的平摊代价为 $O(n)$，由于我们已经证明总的平摊代价是实际代价的上界，所以总的实际代价为 $O(n)$。

动态规划

前面学习的分治方法将问题划分为 **互不相交** 的子问题，递归地求解子问题，然后将它们的解组合起来，得到原问题的解。

动态规划应用于 **子问题重叠** 的情况，即不同子问题具有公共的**子子问题**（子问题需要划分为更小的子子问题来递归求解）。

当子问题有重叠时，分治法会做很多不必要的工作，反复求解公共子子问题；而动态规划会将每个子子问题的解保存在表格中，不会重复计算。

设计一个动态规划算法通常有四个步骤：

1. 刻画一个最优解的结构特征
2. 递归地定义最优解的值
3. 计算最优解的值（通常使用自底向上方法）
4. 利用计算出的信息构造一个最优解

矩阵链乘法

矩阵链乘法问题（matrix-chain multiplication problem）可以这样描述：给定 n 个矩阵的链 $<A_1,A_2,...,A_n>$，矩阵 A_i 的规模为 $p_{i-1}\times p_i$，求 **完全括号化方案**，使得计算矩阵链乘积时所需乘法次数最少。

第一步：最优解的结构特征

将矩阵链乘法 $A_iA_{i+1}...A_j$ 表示为 $A[i:j]$。假设在 k 处分割（加括号）能得到最优顺序：$\left(A_{1} A_{2} \ldots A_{n}\right)=\left(\left(A_{1} \ldots A_{k}\right)\left(A_{k+1} \ldots A_{n}\right)\right)$，即 $A[1:n]=A[1:k]\ A[k+1, n]$。

第二步：递归方案

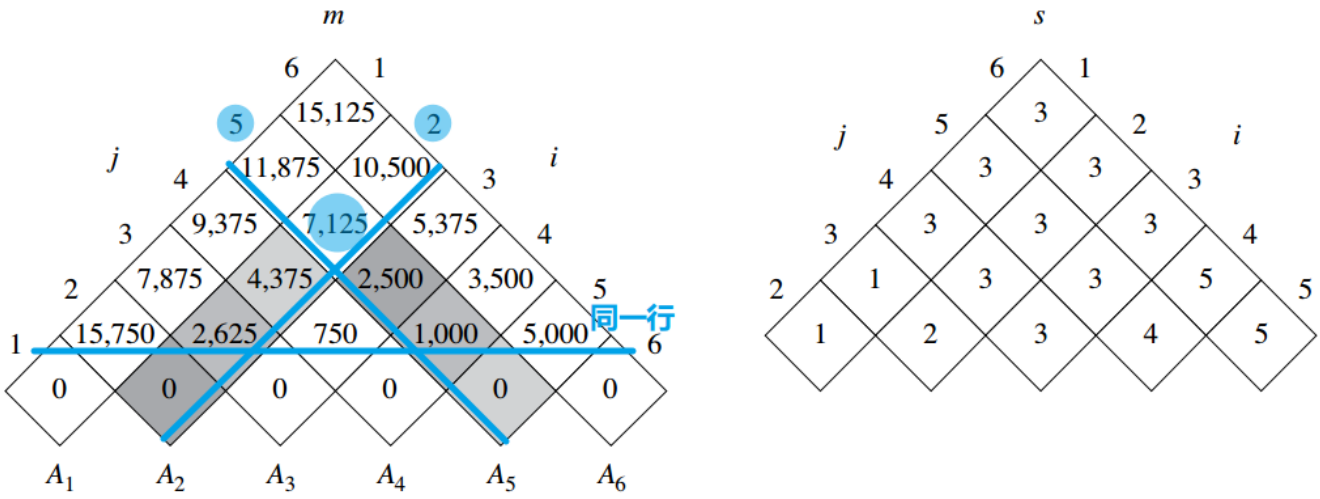
使用 $m[i,j]$ 表示计算 $A[i:j]$ 的最小代价，则有递归式： $$ m[i,j]= \left{ \begin{array}{ll} 0 & i=j \ \min_{i \le k < j}{ m[i,k]+m[k+1,j]+p_{i-1}p_kp_j } & k>0 \end{array} \right. $$ 例如 $$ m[2,4]=\min \left{ \begin{array}{ll} m[2,2]+m[3,4]+p_1p_2p_4 \[0.3em] m[2,3]+m[4,4]+p_1p_3p_4 \end{array} \right. $$ **第三步：计算最优解**

```
MATRIX-CHAIN-ORDER(p)
    n=p.length-1
    let m[1..n,1..,n] and s[1..n-1,2..n] be new tables
    for i=1 to n
        m[i,i]=0 //所有长度为 1 的链的最小代价为 0
    for l=2 to n
        for i=1 to n-l+1 // 计算各个长度为 l 的链的代价
            j=i+l-1
            m[i,j]=INFINITY
            for k=i to j-1
                q=m[i,k]+m[k+1,j]+p_(i-1)p_kp_j
                if q<m[i,j]
                    m[i,j]=q
                    s[i,j]=k // s 记录最优划分的位置
return m and s
```

> 时间复杂度 $O(n^3)$，空间复杂度 $O(n^2)$。

上面代码中，6-14 行的循环计算出了各种长度的链的代价：第一次循环时，对所有 $i=1,2,...,n-2$ 计算 $m[i,i+1]$（长度 $l=2$ 的链的最小代价）。第二次循环时，计算所有 $m[i,i+2]$（长度为 3 的链的最小代价），以此类推。在每次循环中，11-14 行计算 $m[i,j]$ 时仅依赖于 **已经计算出的表项** $m[i,k],\ m[k+1,j]$。

下面是一个例子：



当 $n=6$ 和矩阵规模如下表时，MATRIX-CHAIN-ORDER 计算出的 m 表和 s 表。

矩阵	A_1	A_2	A_3	A_4	A_5	A_6
规模	30×35	35×15	15×5	5×10	10×20	20×25

我们将两个表进行了旋转，使得主对角线方向变为水平方向。表 m 只使用主对角线和上三角部分，表 s 只使用上三角部分。6 个矩阵相乘所需的最少标量乘法运算次数为 $m[1,6]=15\,125$。表中有些表项被标记了深色阴影，相同的阴影表示过程在第 10 行中计算 $m[2,5]$时同时访问了这些表项：

$$
\begin{aligned}
m[2,5] &= \min\begin{cases} m[2,2]+m[3,5]+p_1p_2p_5 = 0+2\,500+35\cdot 15\cdot 20 = 13\,000 \\ m[2,3]+m[4,5]+p_1p_3p_5 = 2\,625+1\,000+35\cdot 5\cdot 20 = 7\,125 \\ m[2,4]+m[5,5]+p_1p_4p_5 = 4\,375+0+35\cdot 10\cdot 20 = 11\,375 \end{cases} \\
&= 7\,125
\end{aligned}
$$

因为 $m[i,j]$ 只在 $i \ge j$ 时有意义，所以表格中只有主对角线之上的部分，且图中的表格向左旋转了 45°。**在这种布局中**，$m[i,j]$ 恰好位于 A_i 向左下的那条线和 A_j 向右下的那条线的交点上，同一行的表项对应**同样长度**的矩阵链。上面的算法按照这个表的形式，自下而上、自左至右地计算所有行。当计算表项 $m[i,j]$ 时，会用到 **乘积** $p_{i-1}p_kp_j$，以及 $m[i,j]$ **左下和右下** 的所有表项。

另外，表 s 记录了每个子链对应的最佳分割位置：例如表项 $s[i,j]$ 记录了一个 k 值，表示 $A_iA_{i+1}...A_j$ 的最佳分割位置在 A_k 和 A_{k+1} 之间，然后 $s[i,k]$ 和 $s[k+1,j]$ 又分别记录着两个子链的最佳分割位置，这样递归地找下去就能得到最终括号划分。

```
PRINT-OPTIMAL-PARENS(s,i,j)
    if i==j
        print "A"_i // 递归出口
    else print "("
```

```
        PRINT-OPTIMAL-PARENS(s, i, s[i,j])
        PRINT-OPTIMAL-PARENS(s, s[i,j], j)
        print")"
```

最长公共子序列

一个给定序列的 **子序列**，就是将给定序列中若干个元素去掉之后得到的结果（元素不必连续）。

最长公共子序列问题（longest-common-subsequence problem）给定两个序列 $X=<x_1,...,x_m>, \ Y=<y_1,...,y_n>$，求 $X,\ Y$ 的最长公共子序列。

第一步：刻画最长公共子序列的特征

子问题的自然分类对应两个输入序列的 **前缀** 对，前缀的定义为：给定一个序列 $X=<x_1,...,x_m>$，定义 X 的第 i 前缀为 $X_i=<x_1,...,x_i>$。

定理（**LCS 的最优子结构**）：令 $X=<x_1,...,x_m>, \ Y=<y_1,...,y_n>$ 为两个序列，$Z=<z_1,...,z_k>$ 为 $X,\ Y$ 的任意 LCS，则

1. 如果 $x_m=y_n$，则 $z_k=x_m=y_n$，且 Z_{k-1} 是 X_{m-1} 和 Y_{n-1} 的一个 LCS。
2. 如果 $x_m \ne y_n$，且 $z_k \ne x_m$，那么 Z 是 X_{m-1} 和 Y 的一个 LCS。
3. 如果 $x_m \ne y_n$，且 $z_k \ne y_m$，那么 Z 是 X_{m} 和 Y_{n-1} 的一个 LCS。

第二步：递归解

从上面定理中我们知道：

- 如果 $x_m=y_n$，我们应该求解 $X_{m-1},\ Y_{n-1}$ 的 LCS，再将 $x_m=y_n$ 追加到其末尾就得到 $X,\ Y$ 的一个 LCS。
- 如果 $x_m \ne y_m$，我们需要求解两个子问题：
    - $X_{m-1}, \ Y$ 的 LCS
    - $X,\ Y_{n-1}$ 的 LCS

  两个 LCS 的较大者即为 X 和 Y 的一个 LCS。

用 $c[i,j]$ 表示 $X_i,\ Y_j$ 的 LCS 长度，递归式为： $$ c[i,j]= \left{ \begin{array}{ll} 0 & 若\ i=0 \ 或\ j=0 \ c[i-1,j-1]+1 & 若\ i,j>0 \ 且 \ x_i=y_i \ \max(c[i,j-1],c[i-1,j]) & 若\ i,j>0\ 且\ x_i \ne y_i \end{array} \right. $$ **第三步：计算**

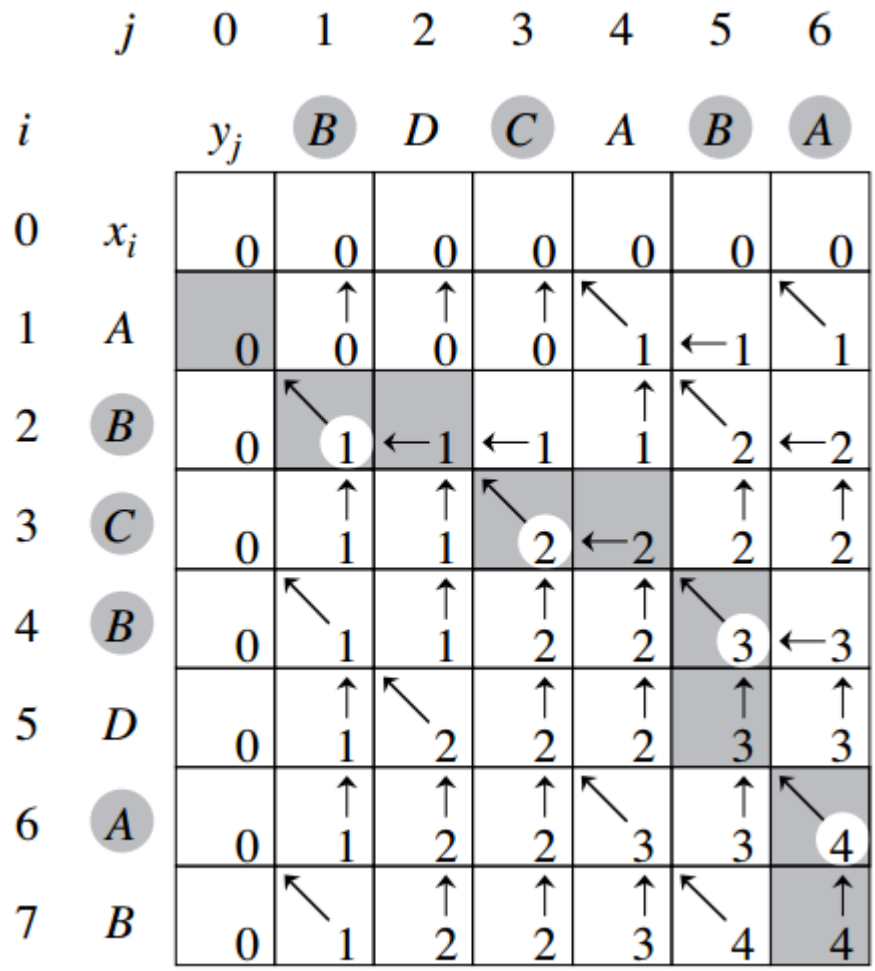
在下面的代码中，接收两个序列作为输入，将 $c[i,j]$ 的值保存在表 $c[0..m,0..n]$ 中，并依次从左至右计算每一行。还维护一个表 $b[1..m,1..n]$ 帮助构造最优解，$b[i,j]$ 表示计算 $c[i,j]$ 时所选择的子问题最优解。

```
LCS-LENGTH(X,Y)
    m=X.length, n=Y.length
    let b[1..m,1..n] and c[0..m,0..n] be new tables
    for i=1 to m c[i,0]=0
    for j=0 to n c[0,j]=0
    for i=1 to m
        for j=1 to n
```

```
            if x_i==y_i
                c[i,j]=c[i-1,j-1]+1
                b[i,j]="↖"
            // 下面是 max(c[i-1,j],c[i,j-1])
            else if c[i-1,j]>=c[i,j-1]
                c[i,j]=c[j-1,j]
                b[i,j]="↑"
            else
                c[i,j]=c[i,j-1]
                b[i,j]="←"
    return c and b
```

运行时间为 $O(mn)$。

下面是计算 $X=\left\langle A,B,C,B,D,A,B\right\rangle$ 和 $Y=\left\langle B,D,C,A,B,A\right\rangle$ 的 LCS 的过程：



在上面表格中，单元格 (i,j) 记录了 $c[i,j]$ 的值和 $b[i,j]$ 的箭头。$c[i,j]$ 仅依赖于是否 $x_i=x_j$ 以及 $c[i-1,j],\ c[i,j-1], c[i-1, j-1]$ 的值 ， 而**这些值会在计算 $c[i,j]$ 之前被计算出来。**

右下角 的 $c[7,6]$ 记录的 4 就是 $X,\ Y$ 的一个 LCS 的长度，从右下角开始沿着 $b[i,j]$ 的箭头向上追溯，每个 “↖” 对应的表项就是 LCS 中的一个元素。

第四步：构造 LCS

只需简单地从 $b[m,n]$ 开始，按箭头方向回溯即可——当遇到 `b[i,j]=="↖"` 时，表示 $x_i=y_j$ 是 LCS 的一个元素。

```
PRINT-LCS(b,X,i,j)
    if i==0 or j==0 return
    if b[i,j]=="↖"
        PRINT-LCS(b,X,i-1,j-1)
        print x_i
    else if b[i,j]=="↑"
        PRINT-LCS(b,X,i-1,j)
    else
        PRINT-LCS(b,X,i,j-1)
```

运行时间为 $O(m+n)$。

凸多边形的三角剖分

凸多边形的三角剖分是指利用弦集（chord set）T 将凸多边形分割成互不相交的三角形。一个包含 n 个顶点的凸多边形刚好能用 $n-3$ 条弦划分为 $n-2$ 个三角形。

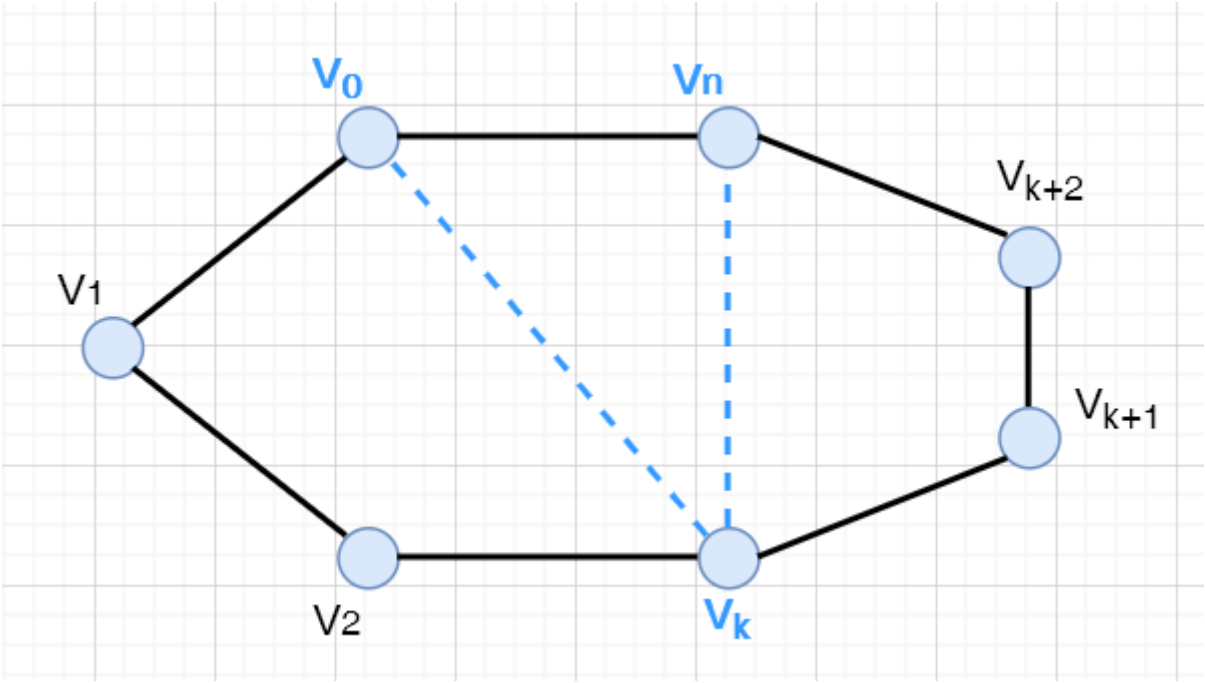
定义权重函数：$w(v_iv_jv_k)=|v_iv_j|+|v_jv_k|+|v_jv_k|$。（三个顶点组成的三角形各边的距离之和）。

最优三角剖分是指得到的剖分结果中，**所有三角形的边权之和最小。**

最优子结构性质

若 $n+1$ 边形的最优三角剖分 T 包含三角形 $V_0V_kV_n$，则 T 的权为三部分之和：

- $\triangle V_0V_kV_n$ 的权；
- 多边形 $V_0V_1...V_k$ 的最优三角剖分权值；
- 多边形 $V_kV_{k+1}...V_n$ 的最优三角剖分权值。



递归式

设 $t[i][j]$ 表示凸多边形 $V_{i-1}V_i...V_j$ 的最优三角剖分的权值，从最优子结构性质可得到递归式： $$ t[i][j]= \left{ \begin{array}{ll} 0, & i=j \ \min_{i\le k < j}{ t[i][k]+t[k+1][j]+w(V_{i-1}V_kV_j) } & i<j \end{array} \right. $$ 所以有 $$ t[1][\mathrm{n}]= \left{ \begin{array}{ll} 0 & i=j \ \min *{1 \leq i \leq k<j \leq n}\left{t[1][k]+t[k+1][n]+w\left(V*{0} V_{k} V_{n}\right)\right} & i<j \end{array} \right. $$

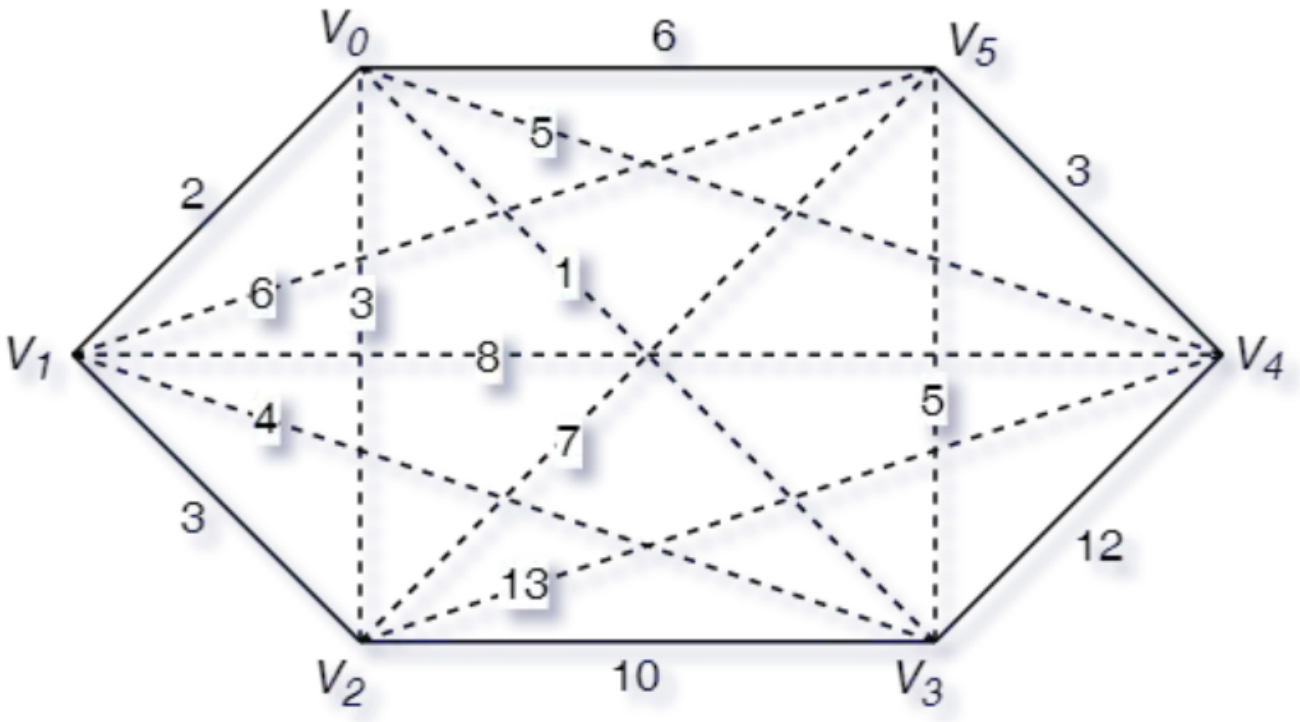
> 注意 k 的取值范围：$[i,j)$

程序执行过程：

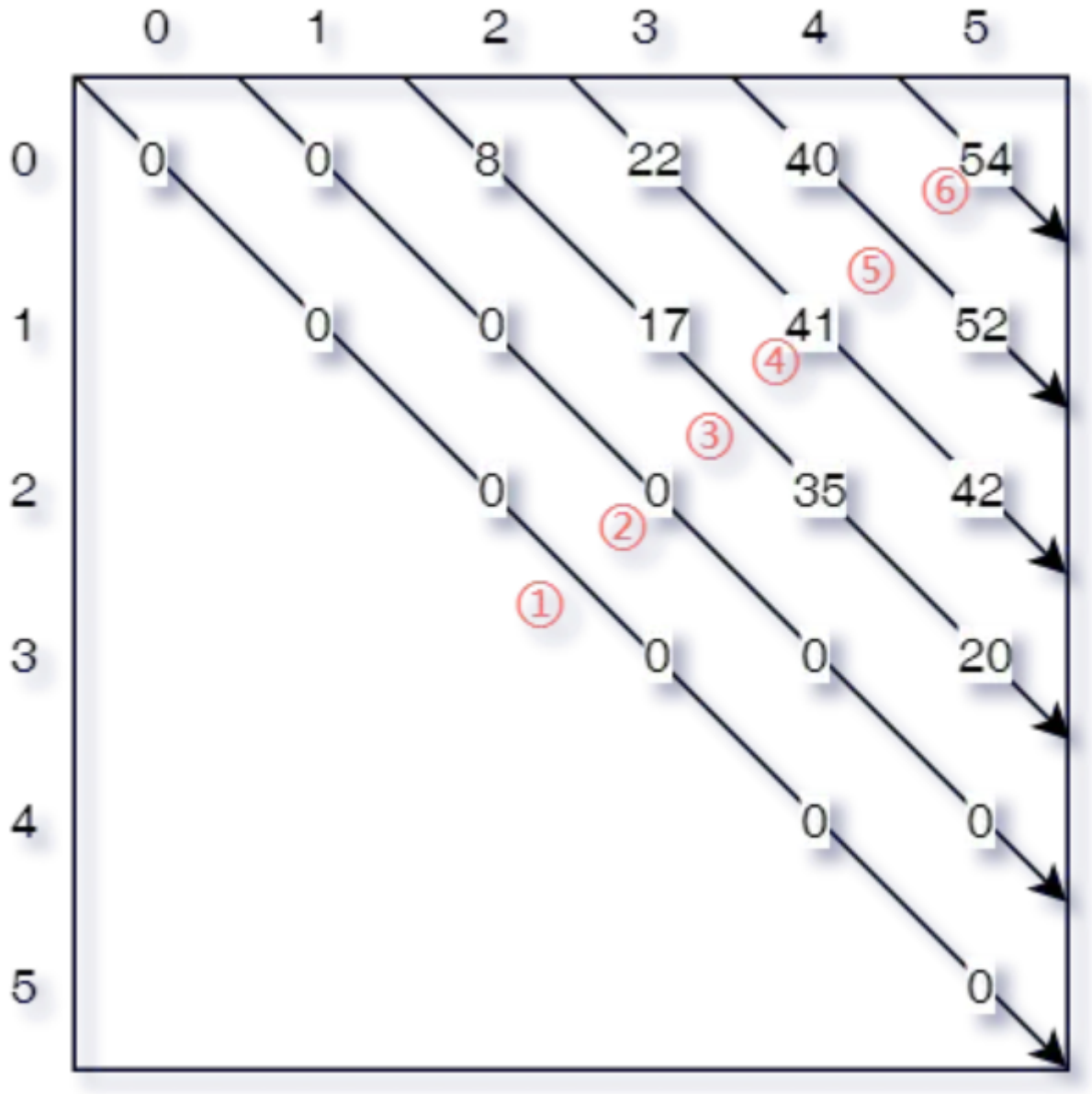
1. 初始化：以每个顶点作为 $t[i][j]$ 中的 i 值（顶点序列的起点），初始化表格的对角线为 0（当 $i=j$ 时，$t[i][j]=0$）
2. 以 1 为步长，分别计算 $t[1,2],\ t[2,3],...,t[n-1,n]$ 的值；
3. 以 2 为步长，分别计算 $t[1,3],t[2,4],...,t[n-2,n]$ 的值；
4. 不断增加步长，循环执行上述运算，直到步长为 n，此时得出最终结果。

下面是一个例子：

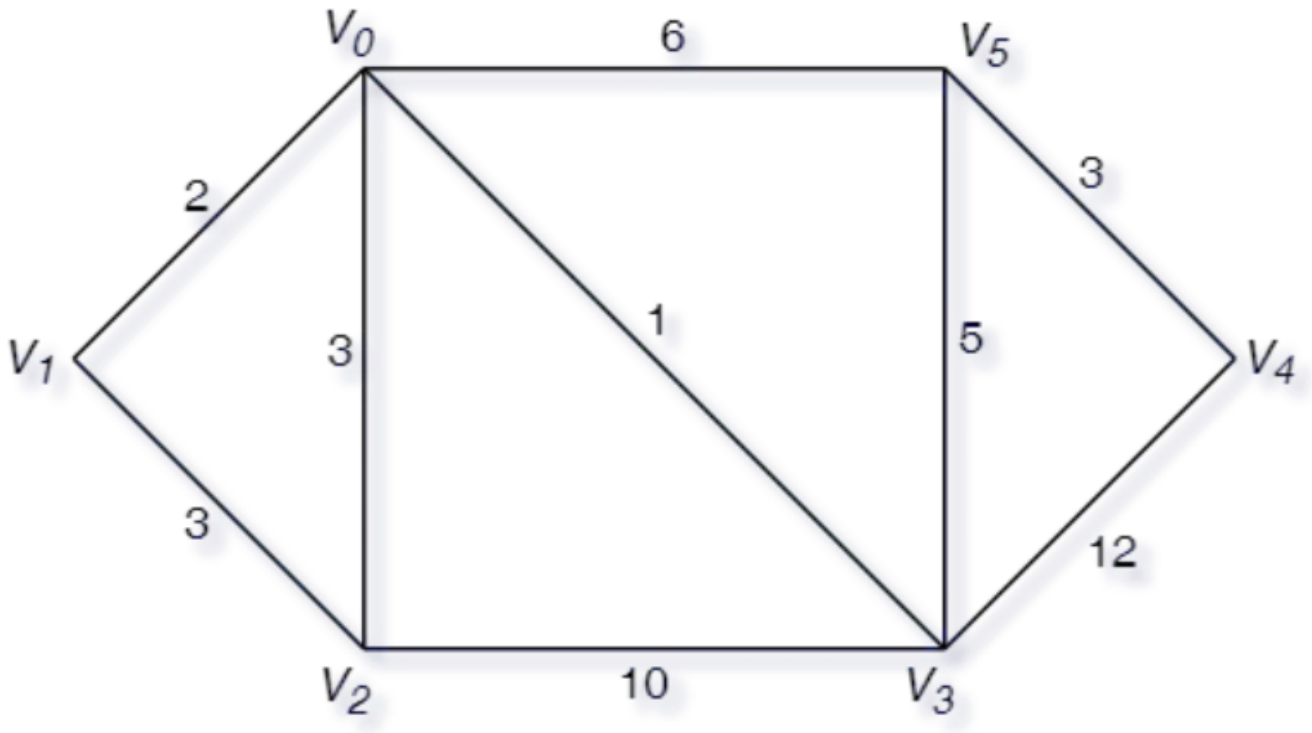
给出一个所示带有权重的凸六边形：



按照下图中的计算顺序计算：先计算步长为 0 的序列（主对角线），然后计算步长为 1 的序列（和主对角线紧邻的对角线），以此类推。



最后得到最优划分结果如下：



贪心算法

线性规划

快速傅里叶变换