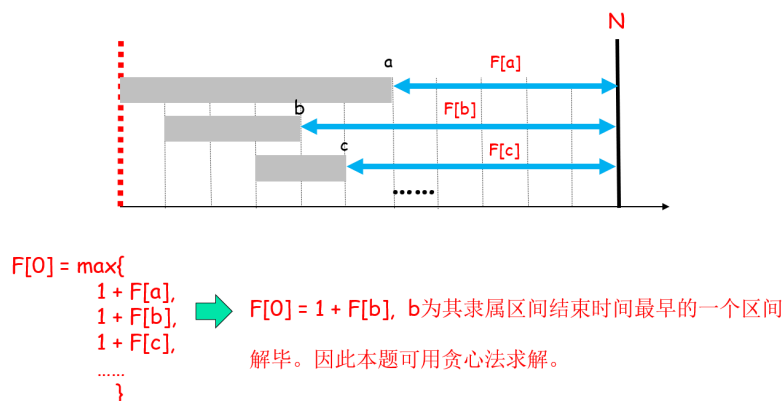


# 期末复习（精简版本复习1）

- 区间调度问题



- 稳定匹配算法( $G - S$ 算法)

- 匹配的定义

- 一个匹配 $S$ 是来自 $M * W$ 的有序对的集合，并且有如下性质：每个 $M$ 的成员和每个 $W$ 的成员至多出现在 $S$ 的一个有序对中

- 完美匹配

- 一个完美匹配 $S'$ 是具有如下性质的匹配： $M$ 的每个成员和 $W$ 的每个成员恰好出现在 $S'$ 的一个队里。

- 稳定匹配

- 不稳定因素

- 给定一个完美匹配 $S$ ,在 $S$ 中存在两个对 $(m, w)$ 和 $(m', w')$ ,如果 $m$ 更偏爱 $w'$ 而不爱 $w$ ,而且 $w'$ 更偏爱 $m$ 而不爱 $m'$ ....称 $(m, w')$ 是一个相对于 $S$ 的不稳定因素： $(m, w')$ 不属于 $S$ ,但是 $m$ 和 $w'$ 双方都偏爱另一方而不爱他们在 $S$ 中的伴侣。

- 我们说一个匹配 $S$ 是稳定的，如果

- 匹配 $S$ 是完美的
- 不存在相对于 $S$ 的不稳定因素

- $G - S$ 算法在至多 $n^2$ 次While循环的迭代后终止

- 定义 $P(t)$ : 迭代 $t$ 结束时， $m$ 已经向 $w$ 发出过邀请的那些 $(m, w)$ 的集合。
- 可知 $P(t)$ 大小严格递增。且 $(m, w)$ 只存在 $n^2$ 种可能。

- 算法实现

```

Initialize each person to be free.
while (some man is free and hasn't proposed to every woman) {
    Choose such a man m
    w = 1st woman on m's list to whom m has not yet proposed
    if (w is free)
        assign m and w to be engaged
    else if (w prefers m to m')
        assign m and w to be engaged, and m' to be free
    else
        w rejects m
}

```

- 女人如何判断接收/拒绝邀请？

- 女人对自己的优先表做预处理，反向变换；这样以后判别的时候就是常数阶的代价；

```

for i = 1 to n
    inverse[pref[i]] = i

```

- G-S算法的执行步骤与自由的男人的选择有关，如果选择不同，那么G-S算法所有的执行会得到同样的匹配吗？对！所有的执行得到同样的匹配！
- 如果存在一个稳定匹配包含了 $(m, w)$ 对，我们就说女人 $w$ 是男人 $m$ 的有效伴侣。如果 $w$ 是 $m$ 的有效伴侣，且没有别的在 $m$ 的排名中比 $w$ 更高的女人是他的有效伴侣，那么 $w$ 就是 $m$ 的最佳有效伴侣，记为 $best(m)$ 。即不存在某个匹配包含某个男孩 $m$ 没有获得自己的最佳伴侣 $w$ 。
- 男孩都获得了自己的最佳有效伴侣，女孩都获得了自己的最差有效伴侣

## • 算法分析基础

- 一个算法被称为是多项式时间的如果满足如下的性质：当算法输入的规模增长时，算法的运行时间是多项式有界的。也就是：存在常数 $c > 0, d > 0$ , 使得对于每一个问题输入的规模 $N$ , 算法的运行都能够在 $cN^d$ 步骤内完成。
- $O$ : 上界 最坏情况的上限 算法运行时间至多是 $f(n)$ 的常数倍  $\Omega$ : 下界 最好情况的下限 算法运行时间至少是 $f(n)$ 的常数倍  $\theta$ : 同阶

# $O, \Omega, \Theta$

## • 传递性

- If  $f = O(g)$  and  $g = O(h)$  then  $f = O(h)$ .
- If  $f = \Omega(g)$  and  $g = \Omega(h)$  then  $f = \Omega(h)$ .
- If  $f = \Theta(g)$  and  $g = \Theta(h)$  then  $f = \Theta(h)$ .
- 本质是：不等号的传递性！

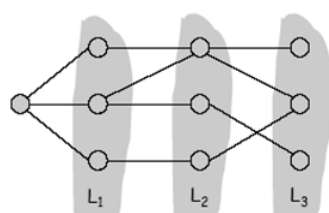
## • 可加性

- If  $f = O(h)$  and  $g = O(h)$  then  $f + g = O(h)$ .

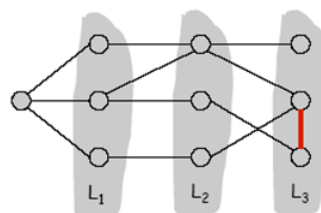
- If  $f = \Omega(h)$  and  $g = \Omega(h)$  then  $f + g = \Omega(h)$ .
- If  $f = Q(h)$  and  $g = Q(h)$  then  $f + g = Q(h)$ .
- 本质是：不等号的可加性！
- $O(n \log n)$ 
  - 快速排序(Quicksort), FFT(快速傅立叶变换), 归并排序, 堆排序
- $O(n^k)$ 
  - 对某个固定常数  $k$ , 我们想知道给定的  $n$  个结点的输入图是否有一个大小为  $k$  的独立集。
- 指数时间
  - 假设给定一个图, 需要找一个最大规模的独立集
- $O(n!)$ 
  - 对稳定匹配问题的穷举搜索:  $n!$
  - 二分匹配(二部图每边存在  $n$  个结点)问题, 如果穷举搜索, 代价是  $n!$

## • 图

- 下面任意两条都可以推出第三条。
  - $G$  是连通的。
  - $G$  不包含一个圈。
  - $G$  有  $n - 1$  条边。
- 如果图是邻接表给出,  $BFS$  算法的实现将以  $O(m + n)$  时间运行。
- 存在  $O(m + n)$  的有效算法判别图  $G$  是否强连通。
  - 先正向  $BFS$  看能不能遍历完所有的点, 再将变反向, 再次便利看能不能到达所有的点
  - $O(m + n)$   $m$  是点,  $n$  是边, 两遍都是  $O(m + n)$
- 二分性测试
  - 图  $G$  是二部图当且仅当图中没有奇圈。
  - 设  $G$  是一个连通图,  $L_0, \dots, L_k$  是从顶点  $s$  由  $BFS$  算法生成的层。那么下面两件事一定恰好成立其一：
    - $G$  中没有边与同一层的两个结点相交。这种情况下  $G$  是二部图, 其中偶数层的结点可以着红色, 奇数层结点可以着蓝色。
    - $G$  中有一条边与同一层的两个结点相交。此种情形下, 存在一个奇圈, 不可能是二部图。



Case (i)



Case (ii)

- 有向无圈图与拓扑排序
  - $DAG$  (Directed Acyclic Graphs): 有向图没有圈

- 如果  $G$  是一个拓扑排序, 那么  $G$  是一个  $DAG$

To compute a topological ordering of  $G$ :

Find a node  $v$  with no incoming edges and order it first

Delete  $v$  from  $G$

Recursively compute a topological ordering of  $G - \{v\}$

and append this order after  $v$

- 定理 上面算法在  $O(m + n)$  时间内找到一个拓扑排序
  - 考虑边逐次递减的代价  $O(m)$ ; 追踪被删除的结点代价  $O(n)$ . 每个节点: 被访问一次  $\rightarrow O(n)$  每条边: 被检查一次  $\rightarrow O(m)$

## 签到题目

### 2 玻璃瓶样品强度测试

你正在对不同的玻璃瓶样品做强度测试以确定它们从多高的高度掉下来而仍旧不碎。对某类特定的瓶子设计这样一个实验。你有一个具有  $n$  个横挡的阶梯, 并且你想找出最高的横挡, 能使一个瓶子的样品从那里下落而不被摔碎。我们把它称为“最高的安全横挡”。

尝试二分搜索可能是一种自然的想法: 使一个瓶子从中间的横挡下落, 看看它是否摔碎, 然后依赖于这个结果递归地从  $n/4$  横挡或者  $3n/4$  的横挡进行尝试。但是这个办法有缺点, 在找到答案之前, 你可能摔碎了一大堆瓶子。

如果你的主要目标是保护瓶子, 另一方面, 你可以尝试下面的策略。从第一个横挡开始让瓶子下落, 然后第二个横挡, 继续下去, 每次向上爬一个高度直到瓶子摔碎为止。以这种方式, 你只需要一个瓶子——在它摔碎的时刻, 你得到正确的答案——但是你可能不得不把它下落  $n$  次 (而不是在二分搜索求解时的  $\log n$  次)。

于是, 这里是某种权衡: 似乎如果你愿意打碎更多的瓶子, 你就可以执行更少的下落。为了更好地理解这种权衡在量级上是怎样起作用的, 让我们考虑给定固定  $k \geq 1$  个瓶子的“预算”。怎样来运行这个实验。换句话说, 你必须确定正确的答案——最高的安全横挡——并且在这个过程中至多可以用  $k$  个瓶子。

- (a) 假设只给你  $k = 2$  个瓶子。描述一种找到最高安全横挡的策略, 对某个比线性函数增长更慢的函数  $f(n)$ , 要求一个瓶子至多落  $f(n)$  次 (换句话说, 应该满足  $\lim_{n \rightarrow \infty} f(n)/n = 0$ )。

- (b) 假设你的瓶子数的预算是个给定的  $k > 2$ 。描述一种至多使用  $k$  个瓶子找到最高安全横挡的策略。如果  $f_k(n)$  表示依照你的策略需要下落瓶子的次数, 那么函数  $f_1, f_2, f_3, \dots$  应该有这样的性质, 每个函数渐近增长比前面的函数要慢: 对每个  $k$ ,  $\lim_{n \rightarrow \infty} f_k(n)/f_{k-1}(n) = 0$ 。

#### 2. 玻璃瓶样品强度测试

##### (a) $k = 2$ 个瓶子的策略

1. 分块策略: 将阶梯分为  $\sqrt{n}$  个块, 每个块大小为  $\sqrt{n}$ 。

2. 第一瓶测试: 依次测试每个块的最后一个横挡 (如  $\sqrt{n}, 2\sqrt{n}, \dots$ ), 直到瓶子破碎。

3. 第二瓶测试: 在破碎块内, 从低到高逐个测试横挡。

4. 总次数: 最多  $2\sqrt{n}$ , 满足  $\lim_{n \rightarrow \infty} \frac{2\sqrt{n}}{n} = 0$ 。

如果  $n$  不是一个完全平方数, 那么我们从  $\lfloor \sqrt{n} \rfloor$  的倍数高度扔第一个瓶子, 然后对第二个瓶子应用上述规则。这样, 我们最多扔第一个瓶子  $2\sqrt{n}$  次, 第二个瓶子最多扔  $\sqrt{n}$  次, 仍然得到  $O(\sqrt{n})$  的界限。

##### (b) $k > 2$ 个瓶子的策略

1. 递归分块: 将阶梯分为  $n^{1/k}$  个块, 每个块大小为  $n^{1-1/k}$ 。

2. 逐层细化: 用第一个瓶子确定破碎块后, 剩余  $k-1$  个瓶子递归处理子块。

3. 总次数: 最多  $k \cdot n^{1/k}$ , 满足  $\lim_{n \rightarrow \infty} \frac{k \cdot n^{1/k}}{n} = 0$ 。

(例如,  $k = 3$  时, 次数为  $O(n^{1/3})$ , 比  $O(n^{1/2})$  更慢)

#### 答案

1. 函数排序结果:

$g_1 \prec g_4 \prec g_3 \prec g_5 \prec g_2 \prec g_7 \prec g_6$

2. (a) 策略: 分块大小为  $\sqrt{n}$ , 最多  $2\sqrt{n}$  次测试。

(b) 策略: 递归分块  $n^{1/k}$ , 次数为  $O(k \cdot n^{1/k})$ 。

## 2 玻璃瓶样品强度测试

- (a) 为了简化问题, 假设  $n$  是一个完全平方数。我们从  $\sqrt{n}$  的倍数高度 (即从  $\sqrt{n}, 2\sqrt{n}, 3\sqrt{n}, \dots$ ) 开始往下扔第一个罐子, 直到它破碎为止。

如果我们从最高层往下扔并且它幸存了, 那么我们的工作也就完成了。否则, 假设它在高度  $j\sqrt{n}$  处破碎。那么我们知道最高的安全层位于  $(j-1)\sqrt{n}$  和  $j\sqrt{n}$  之间, 因此我们从  $1 + (j-1)\sqrt{n}$  层开始往上扔第二个罐子, 每次增加一层。这样, 我们每个罐子最多扔  $\sqrt{n}$  次, 总共最多扔  $2\sqrt{n}$  次。

如果  $n$  不是一个完全平方数, 那么我们从  $\lfloor \sqrt{n} \rfloor$  的倍数高度扔第一个罐子, 然后对第二个罐子应用上述规则。这样, 我们最多扔第一个罐子  $2\sqrt{n}$  次, 第二个罐子最多扔  $\sqrt{n}$  次, 仍然得到  $O(\sqrt{n})$  的界限。

- (b) 我们通过归纳法声称  $f_k(n) \leq 2kn^{1/k}$ 。我们首先从  $\lfloor n^{(k-1)/k} \rfloor$  的倍数高度扔第一个罐子。这样, 我们最多扔第一个罐子  $2n/n^{(k-1)/k} = 2n^{1/k}$  次, 从而将可能的层数缩小到最多  $n^{(k-1)/k}$  的长度区间。

然后我们递归地对  $k-1$  个罐子应用这个策略。通过归纳, 它最多使用  $2(k-1)(n^{(k-1)/k})^{1/(k-1)} = 2(k-1)n^{1/k}$  次。加上使用第一个罐子所做的  $\leq 2n^{1/k}$  次, 我们得到了  $2kn^{1/k}$  的界限, 完成了归纳步骤。

## • 贪心算法

### • 区间调度问题

- 一组需求  $1, 2, \dots, n$ ; 第  $i$  个需求与一个始于  $s(i)$  且止于  $f(i)$  的时间区间相对应。如果没有两个需求在时间上重叠, 我们就说需求的子集是相容的。目标是给出一个最大的相容子集。
- 我们应该接受最早结束的需求, 即  $f(i)$  尽可能小的需求  $i$  为第一个需求。这样的好处是资源尽可能早被释放, 以便于安排下面的需求。
- 算法实现: 把任务(需求)按照结束时间递增排序。依次选取与前面已选定任务相容的新任务。算法时间复杂度?  $N \log N$

初始令  $R$  是所有需求的集合, 设  $A$  为空

While  $R$  非空

    选择一个最小结束时间的需求  $i \in R$

    把  $i$  加到  $A$  中

    从  $R$  中删除与需求  $i$  不相容的所有需求

Endwhile

返回集合  $A$  作为被接受的需求集合

### • 最小延迟调度

#### • 问题描述

需求  $j$  需要长度为  $t_j$  时间段, 截至时间为  $d_j$ 。

需求  $j$  如果在  $s_j$  开始, 那么结束时间是  $f_j = s_j + t_j$ 。

延迟的定义:  $l_j = \max \{ 0, f_j - d_j \}$ 。

目标: 安排所有的需求, 使得计划具有最小的延迟调度:

让  $L = \max l_j$  最小。

- 贪心算法(最早截至时间优先)按照结束时间  $d_i$  增长的次序排序(最早截止时间优先)

```

Sort n jobs by deadline so that  $d_1 \leq d_2 \leq \dots \leq d_n$ 

t ← 0
for j = 1 to n
    Assign job j to interval [t, t + tj]
    sj ← t, fj ← t + tj
    t ← t + tj
output intervals [sj, fj]

```

- 时间复杂度
- 最优算法:  $O(n \log n)$
- 超高速缓存
  - When  $d_i$  需要被放入超高速缓存收回在最远的将来被需要的那个项
- 一个图的最短路径
  - *Dijkstra* 算法
    - 复杂度:  $O(mn)$
    - 采用优先队列实现代价?  $O(m)$  次考虑边,  $n$  次 *ExtractMin*,  $m$  次 *ChangeKey*; 复杂度:  $O(m \log n)$
- 最小生成树问题
  - *Kruskal's algorithm*. 初始  $T = \emptyset$ . 边按照费用递增次序排列. 通过不断插入边来建立一棵生成树: 把边  $e$  插入  $T$  只要不构成圈. *Kruskal* 更适合稀疏图

```

Kruskal(G, c) {
    Sort edges weights so that  $c_1 \leq c_2 \leq \dots \leq c_m$ .
    T ← ∅

    foreach (u ∈ V) make a set containing singleton u

    for i = 1 to m
        (u, v) = ei
        if (u and v are in different sets) {
            T ← T ∪ {ei}
            merge the sets containing u and v
        }
    return T
}

```

- 采用 *union - find* 数据结构. 成一个最小生成树  $T$ . 保持每一个连通分支的集合. 算法代价:  $O(m \log n)$ .
- *Prim's algorithm*. 初始  $S = s$ , 然后贪心选择的增长树  $T$ . 每步选择一端在  $T$  中, 费用最小的边与  $T$  连接. *Prime* 算法更适合稠密图

```

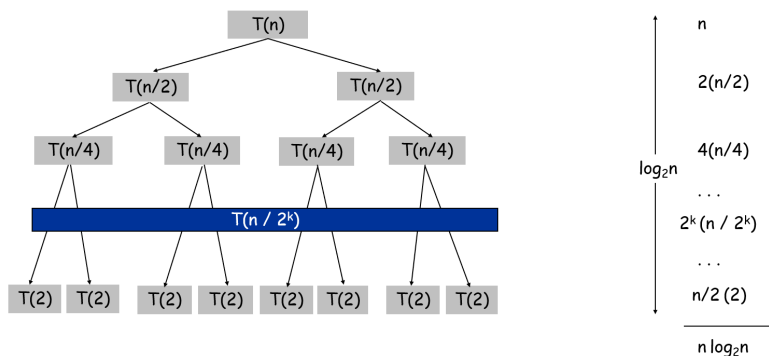
Prim(G, c) {
    foreach (v ∈ V) a[v] ← ∞
    Initialize an empty priority queue Q
    foreach (v ∈ V) insert v onto Q
    Initialize set of explored nodes S ← ∅

    while (Q is not empty) {
        u ← delete min element from Q
        S ← S ∪ {u}
        foreach (edge e = (u, v) incident to u)
            if ((v ∉ S) and (ce < a[v]))
                decrease priority a[v] to ce
    }
}

```

- *Huffman* 码与数据压缩

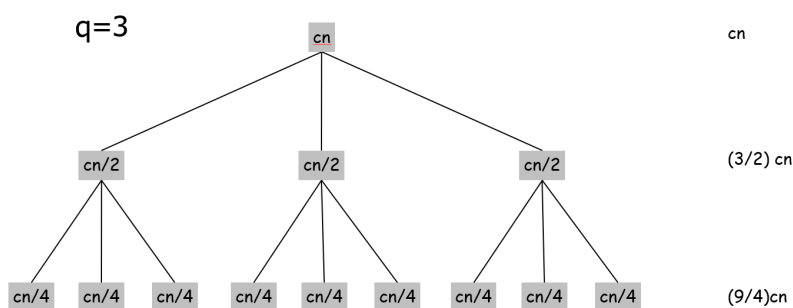
- 一般的代价
  - 识别最低频率的字母在 $O(k)$ 时间内，迭代求和总的代价为 $O(k^2)$
- 采用优先队列(用堆实现)
  - 每次插入和最小元素的取出调整时间 $O(\log k)$ ,总运行时间 $O(k \log k)$
- 堆的调整时间为 $O(\log n)$
- 分治算法
  - 归并排序
    - 如何求解递推关系式的时间复杂度 $q = 2$ 时，时间复杂度为 $O(n \log_2 n)$



$$T(n) \leq \begin{cases} 2T(n/2) + cn, & n > 2 \\ c, & n = 2 \end{cases}$$

- 其他递归关系： $q > 2, q = 1, q = 2$ 递推关系的性质不同，任何满足5.3并具有 $q > 2$ 的函数 $T(\cdot)$ 是有界的。

$$T(n) \leq \begin{cases} qT(n/2) + cn, & n > 2 \\ c, & n = 2 \end{cases}$$



- $q > 2$ 时，时间复杂度为 $O(n^{\log_2 q})$

$$T(n) \leq \sum_{j=0}^{\log_2 n - 1} \left(\frac{q}{2}\right)^j cn = cn \sum_{j=0}^{\log_2 n - 1} \left(\frac{q}{2}\right)^j = cn \left(\frac{r^{\log_2 n} - 1}{r - 1}\right)$$

$$\leq cn \left(\frac{r^{\log_2 n}}{r - 1}\right) = \frac{c}{r - 1} n \cdot r^{\log_2 n} = \frac{c}{r - 1} n \cdot n^{\log_2 r} = \frac{c}{r - 1} n^{\log_2 q} = O(n^{\log_2 q})$$

定理5.4 任何满足5.3并具有 $q > 2$ 的函数 $T(\cdot)$ 是  $O(n^{\log_2 q})$  有界的。

- 任何满足5.3式并具有 $q = 1$ 的函数 $T(\cdot)$  ( $T(n) \leq T(n/2) + O(n)$ ) 是  $O(n)$  有界的。
- $T(n)$  的复杂度应该是  $O(n^2)$

$$T(n) \leq \begin{cases} 2T(n/2) + cn^2, & n > 2 \\ c, & n = 2 \end{cases}$$


### 计数逆序

- 两个指标  $i < j$  构成一个逆序, 如果  $a_i > a_j$ .

Songs

	A	B	C	D	E
Me	1	2	3	4	5
You	1	3	4	2	5

Inversions  
3-2, 4-2



- 划分: 把序列一分为二; 分别递归求出左右各自部分的逆序数;
- 组合: 计数左右部份之间的逆序数, 返回总的逆序数. 左边还有几个数右边要加的逆序数就是几

1 5 4 8 10 2 6 9 12 11 3 7      Divide:  $O(1)$ .

1 5 4 8 10 2      6 9 12 11 3 7      Conquer:  $2T(n/2)$   
5 blue-blue inversions      8 green-green inversions

9 blue-green inversions      Total =  $5 + 8 + 9 = 22$ .  
5-3, 4-3, 8-6, 8-3, 8-7, 10-6, 10-9, 10-3, 10-7

3 7 10 14 18 19      2 11 16 17 23 25  
6 3 2 2 0 0      Count:  $O(n)$

13 blue-green inversions:  $6 + 3 + 2 + 2 + 0 + 0$

2 3 7 10 11 14 16 17 18 19 23 25      Merge:  $O(n)$

$$T(n) \leq T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n) \Rightarrow T(n) = O(n \log n)$$

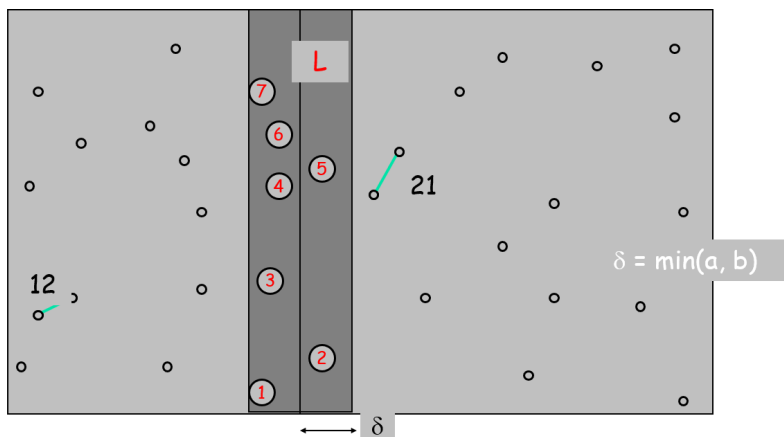
### 算法



- *Merge – and – Count*( $A, B$ )
- *Sort – and – Count*( $L$ )
- *Sort – and – Count*算法正确对输入表排序并且计数逆序个数；它对具有 $n$ 个元素的表运行在 $O(n \log n)$ 时间。

- 最邻近点对

- 给定平面上的 $n$ 个点，找最邻近的一对点，是否能找到一个渐进的比平方阶更快的算法
- 分治算法
  - 划分的时候：用一条垂直线把点集分成相等的两部分
  - 在左右半部分寻找各自最邻近的点对
  - 在“交界”的边中寻找最近的点对
  - 最后返回上面3个解中的最优解
- 根据平面几何的约束性质，只需要考虑分界线附近的点集情况。



- 存在 $O(n \log n)$ 的算法，通过顺便按照 $y$ 归并排序来实现
- 整数乘法

- 采用分治策略：每个整数分成高位，低位

$$x = 2^{n/2} \cdot x_1 + x_0$$

$$y = 2^{n/2} \cdot y_1 + y_0$$

$$xy = 2^n \cdot x_1 y_1 + 2^{n/2} \cdot (x_1 y_0 + x_0 y_1) + x_0 y_0$$

$$= \underbrace{2^n \cdot x_1 y_1}_A + 2^{n/2} \cdot \underbrace{((x_1 + x_0)(y_1 + y_0) - x_1 y_1 - x_0 y_0)}_B + \underbrace{x_0 y_0}_C$$

所以可以把4个乘法变成3个！

- [Karatsuba-Ofman, 1962]Recursive-Multiply算法在两个 $n$ 位因数上的运行时间是 $O(n^{\log_2 3}) = O(n^{1.59})$
- 矩阵乘积
  - 矩阵快速乘积运算. (Strassen, 1969)
    - 先把 $A, B$ 分成 $\frac{1}{2}n \times \frac{1}{2}n$ 的小块；

- 计算：通过10次加减法运算，生成14个 $\frac{1}{2}n \times \frac{1}{2}n$ 矩阵。
- 分治：递归生成7个 $\frac{1}{2}n \times \frac{1}{2}n$ 的矩阵乘积。
- 组合：7个矩阵乘积通过8次矩阵的加减法生成需要的四个矩阵乘积

$$T(n) = \underset{\substack{1 \quad 4 \quad 2 \quad 43 \\ \text{recursive calls}}}{7T(n/2)} + \underset{\substack{1 \quad 4 \quad 2 \quad 4 \quad 3 \\ \text{add, subtract}}}{\Theta(n^2)} \Rightarrow T(n) = \Theta(n^{\log_2 7}) = O(n^{2.81})$$

# 卷积与快速傅立叶变换

- 多项式表达，点值表达，希望能够找到一种新的表达方式，能够在这两种表达之间进行高效的转换

Representation	乘法	求值
系数表达	$O(n^2)$	$O(n)$
点值表达	$O(n)$	$O(n^2)$

- 系数表达到点值的表达，给定多项式 $a_0 + a_1x + \dots + a_{n-1}x^{n-1}$ ,求它在n个不同点 $x_0, \dots, x_{n-1}$ 处的值( $O(n^2)$  formatrix-vector multiply)

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & x_0 & x_0^2 & \dots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \dots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \dots & x_2^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \dots & x_{n-1}^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{bmatrix}$$

- 分治策略(小规模多项式的点值情况已知的情况下，直接测算出2倍规模问题的点值情况)
  - $A(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + a_4x^4 + a_5x^5 + a_6x^6 + a_7x^7$ .
  - $A_{\text{even}}(x) = a_0 + a_2x + a_4x^2 + a_6x^3$ .
  - $A_{\text{odd}}(x) = a_1 + a_3x + a_5x^2 + a_7x^3$ .
  - $A(x) = A_{\text{even}}(x^2) + xA_{\text{odd}}(x^2)$ .
  - $A(-x) = A_{\text{even}}(x^2) - xA_{\text{odd}}(x^2)$ .
- 系数表达到点值表达：给定多项式 $a_0 + a_1x + \dots + a_{n-1}x^{n-1}$ ,求它在n个不同点 $x_0, \dots, x_{n-1}$ 处的值
- 关键之处：选择 $x_k = \omega^k$ ，其中 $\omega$ 是n次单位根。

$$\begin{matrix}
 \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{bmatrix} & = & \begin{bmatrix} 1 & 1 & 1 & 1 & \dots & 1 \\ 1 & \omega^1 & \omega^2 & \omega^3 & \dots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \omega^6 & \dots & \omega^{2(n-1)} \\ 1 & \omega^3 & \omega^6 & \omega^9 & \dots & \omega^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ M & M & M & M & O & M \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \omega^{3(n-1)} & \dots & \omega^{(n-1)(n-1)} \end{bmatrix} & \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{bmatrix}
 \end{matrix}$$

↑ Discrete Fourier transform      ↑ Fourier matrix  $F_n$

### 快速傅立叶变换

- 目标：求  $A(x) = a_0 + \dots + a_{n-1}x^{n-1}$  在  $n$  次单位根  $w_0, w_1, \dots, w_{n-1}$  处的值。
- 分治策略：
- 把多项式分成偶数部分和奇数部分
- $A_{\text{even}}(x) = a_0 + a_2x + a_4x^2 + \dots + a_{n/2-2}x^{(n-1)/2}$ .
- $A_{\text{odd}}(x) = a_1 + a_3x + a_5x^2 + \dots + a_{n/2-1}x^{(n-1)/2}$ .
- $A(x) = A_{\text{even}}(x^2) + xA_{\text{odd}}(x^2)$ .
- 分别计算：计算多项式  $A_{\text{even}}(x)$ ,  $A_{\text{odd}}(x)$  在  $\frac{1}{2}n^{\text{th}}$  单位根  $v^0, v^1, \dots, v^{n/2-1}$  处的值。
- 组合：
- $A(w^k) = A_{\text{even}}(v^k) + w^k A_{\text{odd}}(v^k), 0 \leq k < n/2$
- $A(w^{k+n/2}) = A_{\text{even}}(v^k) - w^k A_{\text{odd}}(v^k), 0 \leq k < n/2$
- $A(x) = A_{\text{even}}(x^2) + xA_{\text{odd}}(x^2)$ .
- FFT**: FFT 算法能够在  $O(n \log n)$  步骤内计算  $n-1$  次多项式在每个  $n$  次单位根处的值 ( $n$  是 2 的整数次幂)。

```

fft(n, a_0, a_1, ..., a_{n-1}) {
    if (n == 1) return a_0

    (e_0, e_1, ..., e_{n/2-1}) ← FFT(n/2, a_0, a_2, a_4, ..., a_{n-2})
    (d_0, d_1, ..., d_{n/2-1}) ← FFT(n/2, a_1, a_3, a_5, ..., a_{n-1})

    for k = 0 to n/2 - 1 {
        ω^k ← e^{2πik/n}
        y_k ← e_k + ω^k d_k
        y_{k+n/2} ← e_k - ω^k d_k
    }

    return (y_0, y_1, ..., y_{n-1})
}

```

- $T(2n) = 2T(n) + O(n)$   $T(n) = O(n \log n)$ .

- 傅立叶变换逆变换

$$G_n = \frac{1}{n} \begin{bmatrix} 1 & 1 & 1 & 1 & L & 1 \\ 1 & \omega^{-1} & \omega^{-2} & \omega^{-3} & L & \omega^{-(n-1)} \\ 1 & \omega^{-2} & \omega^{-4} & \omega^{-6} & L & \omega^{-2(n-1)} \\ 1 & \omega^{-3} & \omega^{-6} & \omega^{-9} & L & \omega^{-3(n-1)} \\ M & M & M & M & O & M \\ 1 & \omega^{-(n-1)} & \omega^{-2(n-1)} & \omega^{-3(n-1)} & L & \omega^{-(n-1)(n-1)} \end{bmatrix}$$

- 算法

```

ifft(n, a0, a1, ..., an-1) {
    if (n == 1) return a0

    (e0, e1, ..., en/2-1) ← IFFT(n/2, a0, a2, a4, ..., an-2)
    (d0, d1, ..., dn/2-1) ← IFFT(n/2, a1, a3, a5, ..., an-1)

    for k = 0 to n/2 - 1 {
        ωk ← e-2πik/n
        yk ← (ek + ωk dk) / n
        yk+n/2 ← (ek - ωk dk) / n
    }

    return (y0, y1, ..., yn-1)
}

```

- 使用快速傅立叶变换，可以在 $O(n \log n)$  时间内计算初始向量 $a$ 和 $b$ 的卷积。