

Ch5 分治策略

1. 基本概念

1.1 概要

分治策略涉及一类算法设计技术，把输入分成若干部分，递归地求解每个部分的问题，然后把这些子问题的解组合成一个全局的解

分析分治算法的运行时间一般与求解一个递推关系有关

自然的暴力算法可能已经是多项式时间的，而分支策略是用来把运行时间减到一个更低的多项式时间，例如从 $O(n^2)$ 降低到 $O(n \log n)$

1.2 递推时间复杂度分析

首先阅读归并排序问题，然后再回来看本部分

我们把归并排序中的递推公式拓展至

$$\text{对于某个常数 } c, \text{ 当 } n > 2 \text{ 时}$$
$$T(n) \leq qT\left(\frac{n}{2}\right) + cn \text{ 并且 } T(2) \leq c$$

现在我们来分别分析 $q > 2$ 和 $q = 1$ 时候的时间复杂度

q>2

与观察模式相同，我们发现，对于递归到第 j 层，我们有 q^j 个不同的子问题，每个规模为 $\frac{n}{2^j}$ 因此在第 j 层执行的总工作为 $q^j(c\frac{n}{2^j}) = (\frac{q}{2})^j cn$

由于递归层数仍然有 $\log_2 n$ 层，则

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

化简可以得到

$$T(n) \leq (\frac{c}{r-1})n^{\log_2 q} \quad (r = \frac{q}{2})$$

最终

$$T(n) = O(n^{\log_2 q})$$

定义

当 $n > 1$, $q > 2$ 时, 任何满足 $T(n) \leq qT(\frac{n}{2}) + cn$ 的函数的时间复杂度为 $O(n^{\log_2 q})$

可以发现, 实际上, 对于较大的 q 至, 递归调用产生了更多的工作

q=1

由于递归层数为 $\log_2 n$ 层, 且每层子问题时间为 $\frac{cn}{2^j}$, 则

$$T(n) \leq \sum_{j=0}^{\log_2 n - 1} cn = 2cn$$

最终

$$T(n) = O(n)$$

定义

当 $n > 1$, $q = 1$ 时, 任何满足 $T(n) \leq T(\frac{n}{2}) + cn$ 的函数的时间复杂度为 $O(n)$

$$T(n) \leq 2T(\frac{n}{2}) + O(n^2)$$

我们把问题拓展到

对于某个常数 c , 当 $n > 2$ 时

$$T(n) \leq qT(\frac{n}{2}) + cn^2 \quad \text{并且} \quad T(2) \leq c$$

由于递归层数为 $\log_2 n$ 层, 且每层子问题时间为 $\frac{cn^2}{2^j}$, 则

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

1.3 总结

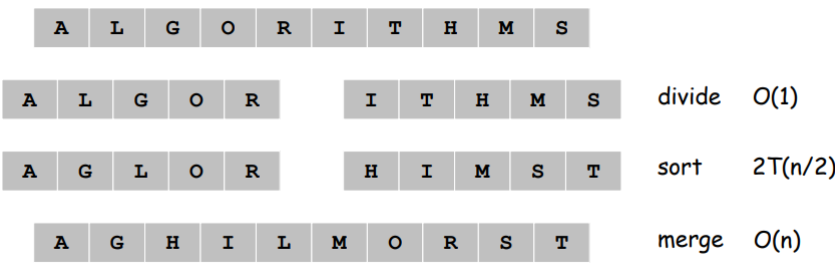
递推公式	时间复杂度
$T(n) = 2T(\frac{n}{2}) + cn$	$O(n \log n)$
$T(n) = qT(\frac{n}{2}) + cn \ (q > 2)$	$O(n^{\log_2 q})$
$T(n) = T(\frac{n}{2}) + cn$	$O(n)$
$T(n) = 2T(\frac{n}{2}) + cn^2$	$O(n^2)$

2. 归并排序

2.1 问题描述

给定 n 个元素，用归并排序按升序重新排列。

2.2 算法



- 1 将数组分成两半
- 2 递归地对每一半排序
- 3 将有序的两半合并使排序完整

2.3 时间复杂度分析

为了分析归并排序的运行时间，可以将它抽象成下面的模板

1. 把输入分成想等大小的两半
2. 使用递归在着两部分分别求解两个子问题
3. 把这两个结果组合成一个全局解，对于初始划分和重新组合仅使用线性时间

在归并排序中，我们需要一个递归的基础情况，一般是使它在某个常数规模的输入上下降到最低点，我们假设一旦输入减少到规模 2，就停止递归并且通过简单的相互比较就把这 2 个元素排序

考虑任何符合上述模式的算法，令 $T(n)$ 表示它在规模为 n 的输入实例上最坏的运行时间，假设 n 为偶数，算法用 $O(n)$ 的时间把每个输入分成每个为 $\frac{n}{2}$ 规模的输入最坏情况的运行时间

于是，运行时间 $T(n)$ 满足下面的递推关系

$$\text{对于某个常数 } c, \text{ 当 } n > 2 \text{ 时}$$

$$T(n) \leq 2T\left(\frac{n}{2}\right) + cn \text{ 并且 } T(2) \leq c$$

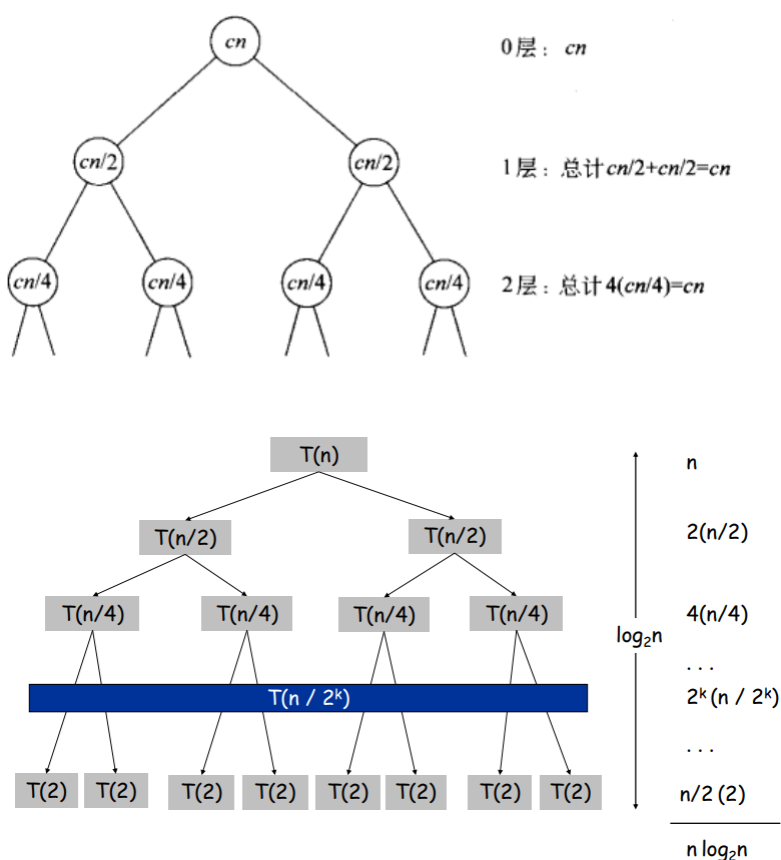
注意到我们可以忽略常数 c ，而把式子进一步写成

$$T(n) \leq 2T\left(\frac{n}{2}\right) + O(n)$$

为了求解该递推式的实际时间复杂度，有两种基本的求解方法

展开归并排序的递推式

求解递推式最直接的方法是按照最初几级的运行时间来展开这个递推式，找出展开的模式，然后在递归的所有级上对运行时间求和



- 递归第 0 层，我们有 1 个规模为 n 的问题，它用的时间最多为 cn
- 递归第 1 层，我们有 2 个规模为 $\frac{n}{2^1}$ 的问题，它用的时间最多为 cn
- 递归第 2 层，我们有 4 个规模为 $\frac{n}{2^2}$ 的问题，它用的时间最多为 cn

- ...
- 递归第 $\log_2(n)$ 层, 我们有 n 个规模为 $\frac{n}{n}$ 的问题, 它用的时间最多为 cn

$$\text{第0层: } T(n) \leq O(n) + 2T\left(\frac{n}{2}\right)$$

$$\text{第1层: } T(n) \leq O(n) + 2O\left(\frac{n}{2}\right) + 4T\left(\frac{n}{4}\right)$$

$$\text{第2层: } T(n) \leq O(n) + 2O\left(\frac{n}{2}\right) + 4O\left(\frac{n}{4}\right) + 8T\left(\frac{n}{8}\right)$$

...

$$\text{第}\log_2 n\text{层: } T(n) \leq O(n) + 2O\left(\frac{n}{2}\right) + 4O\left(\frac{n}{4}\right) + \dots nO(1)$$

为了使得它的规模从 n 减到 2, 输入必须被减半的次数是 $\log_2 n$, 我们得到 $O(n \log n)$ 的总运行时间

定义

当 $n > 1$ 时, 任何满足 $T(n) \leq 2T\left(\frac{n}{2}\right) + cn$ 的函数的时间复杂度为 $O(n \log n)$

把一个解带入归并排序的递推式 (归纳法)

开始有一个对解的猜想, 把它带入递推关系, 检查它是否正确

假设我们相信对所有 $n \geq 2$ 都有 $T(n) \leq cn \log_2 n$

1. 首先它显然对于 $n = 2 : T(2) \leq 2c$ 成立
2. 根据归纳法, 现在假设对所有小于 n 的 m 值都满足 $T(m) \leq cm \log_2 m$
3. 则对于 $T(n)$ 来说

$$\begin{aligned} T(n) &\leq 2T\left(\frac{n}{2}\right) + cn \leq 2c\left(\frac{n}{2}\right)\log_2 \frac{n}{2} + cn \\ &= cn \log_2 \frac{n}{2} + cn = cn(\log_2 n - 1) + cn = cn \log_2 n \end{aligned}$$

故由归纳法得证

部分替换方法

假设我们相信 $T(n) = O(n \log n)$, 但是我们不能确定里面具体的常数使得 $T(n) \leq kn \log_b n$, 为了确定 k 和 b , 我们尝试一下

$$T(n) \leq 2T\left(\frac{n}{2}\right) + cn \leq 2k\left(\frac{n}{2}\right)\log_b \frac{n}{2} + cn$$

如果我们选择底数 $b = 2$, 那么将会有

$$T(n) \leq 2k\left(\frac{n}{2}\right)\log_b \frac{n}{2} + cn = 2k\left(\frac{n}{2}\right)(\log_2 n - 1) + cn = kn\log_2 n - kn + cn$$

最后我们来选取 k 以满足 $T(n) \leq kn\log_2 n$

那么我们可以很直观的看出，当 $c = k$ 的时候，该情况满足

所以我们得到了，若将两个部分解组合起来的时间为 kn ，则归并排序的真实时间复杂度上限为 $kn\log_2 n = O(n \log n)$

2.4 应用

排序算法的应用非常广泛

有些非常明显的应用

- 排序一系列名字
- 安排一个 MP3 收藏馆
- 展示 Google 的 PageRank 算法结果
- 按时间倒序列出 RSS 新闻项目

有一些问题在排序后很容易解决

- 找到中位数
- 找到最近对
- 数据库的二分查找
- 发现离群点
- 在邮件列表中查找副本

还有一些不明显的應用

- 数据压缩
- 计算图形学
- 计算生物学
- 供应链管理
- 亚马逊的图书推荐
- 计算机的负载均衡

3. 逆序对计数

3.1 问题描述

音乐网站试图匹配你与他人的歌曲喜好

- 你对 n 首歌进行打分
- 音乐网站通过数据库查找有相似品味的人

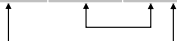
3.2 算法

假设把你与一个陌生人对同一组 n 个电影的排名进行比较，一种比较自然的方法是：

- 按照你的排名从 1 到 n 标记这些电影
- 按照陌生人的排名排序这些标记
- 看到有多少对次序出错

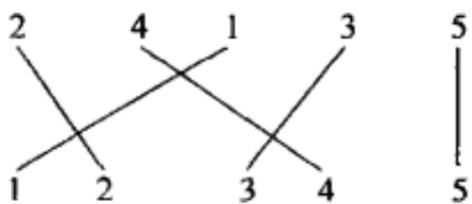
把这个概念量化的最自然的方法是计数逆序的个数

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



- 你的排名 $1, 2, \dots, n$
- 陌生人的排名 a_1, a_2, \dots, a_n
- 按照你的排序的两个曲子的编号为 i 和 j ，如果满足 $i < j, a_i > a_j$ 则它们是一组逆序对
- 如上图 3-2 是一组逆序对，4-2 是一组逆序对

考虑下面一个例子



- 它存在 (2,1) (4,1) 和 (4,3) 三个逆序对，在图中很直观的是，每一个交叉点表示一个逆序对

暴力的算法是，两个两个检查是否是逆序对，算法的时间复杂度为 $O(n^2)$

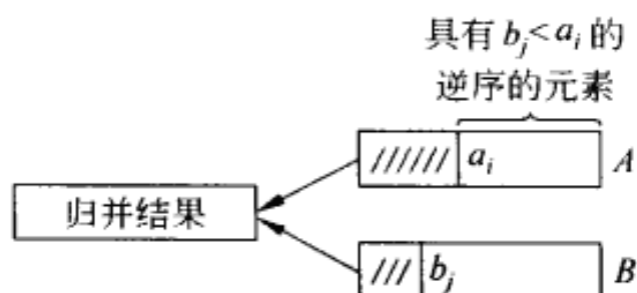
现在我们考虑使用分治法计算逆序对，假设我们有 a_1, a_2, \dots, a_n 个数据需要计算逆序对

- 分成两个部分 a_1, a_2, \dots, a_m 和 a_{m+1}, \dots, a_n
- 计算着两半中的每部分逆序对数
- 计算逆序 (a_i, a_j) 分别属于不同的两半，最好在 $O(n)$ 的时间内完成

注意到这些前一半 / 后一半的逆序有着比较好的形式，精确的说，它们是一个对 (a_i, a_j) 其中 a_i 在前一半， a_j 在后一半，且 $a_i > a_j$

假设我们已经递归地排序好了这个表的前一半和后一半并且计数了每部分的逆序，现在我们有二个排序好的表 A 和 B ，分别包含了前一半和后一半，我们想把它们合并产生一个排好序的表 C ，同时也计数 (a, b) 的个数，其中 $a \in A, b \in B, a > b$

由于 A 和 B 是排好序的，实际上非常容易记录下我们遇到的逆序的个数，每次元素 a_i 被加入到排序好的 C 中，不会遇到新的逆序，因为 a_i 小于表 B 中剩下每个元素，并且它出现在它们大家的前面，另一方面，如果 b_j 被添加到表 C ，那么它比表 A 中剩下的所有元素都小，并且它出现在它们大家的后面，因此我们把逆序对的计数加上在 A 中剩下的元素个数，就可以了



```
1 Merge-and-Count(A,B):
2     维护一个 current 指针指向每个表，初始话指向首元素
3     维护一个变量 Count 用于逆序的对数，初始化为 0
4     while(两个表都不空){
5         令 ai 和 bi 是由 current 指针指向的元素
6         把这两个中较小的元素加到输出表中
7         if(bi 是较小的元素){
8             把 Count 加上 A 中剩余的元素数
9         }
10        把较小的元素输出表，current指针向前移动
11    }
12    一旦一个表为空，把另一个表剩余的所有元素加入到输出中
13    返回 Count 和
```


整个代码为

```
1 // 排序并返回逆序对的个数
2 Inversions-count(L)
3 if (这个表只有一个元素) {
4     return 0;
5 }else{
6     把这个表分成两半;
7     A = 前 n/2 个元素;
8     B = 剩下 n/2 个元素;
9     (rA, A) = Sort-and-Count(A);
10    (rB, B) = Sort-and-Count(A);
11    (r, L) = Merge-and-Count(A,B)
12 }
13
14 return r = rA + rB;
```

3.3 时间复杂度分析

由于我们的 Merge-and-Count 过程用 $O(n)$ 时间，整个 Inversions-count 过程的运行时间满足

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

时间复杂度为 $O(n \log n)$

3.4 应用

Web 上许多网站使用一种称为协同过滤的技术，它们试图使用这种技术把你（对于书、电影、餐馆）的嗜好与其它人进行你匹配，一旦 Web 网站已经识别某些人与你具有“类似”的口味，基于你与他们怎样评价各种事物的比较，它就可能推荐其它人所喜欢的这些新事物

另一个应用出现在 Web 上的元搜索工具中，它在许多不同的搜索引擎上执行同样的查询，然后试图通过在搜索引擎返回各种排名中寻找类似性与差别来综合这些结果

- 投票理论
- 协同过滤
- 测量数组的“排序”
- 谷歌排序函数的敏感性分析
- 用于Web上元搜索的排序聚合
- 非参数统计

4. 找到最邻近的点对

4.1 问题描述

给定平面上的 n 个点，找出它们之间欧几里德距离最小的一对

4.2 算法

给定平面上的 n 个点，找到最邻近的一对点

我们用 $P = \{p_1, p_2, \dots, p_n\}$ 表示点集，其中 p_i 坐标为 (x_i, y_i) ，对两个点 $p_i, p_j \in P$ 我们使用 $d(p_i, p_j)$ 表示它们之间的标准欧几里德距离，我们的目标是找出使得 $d(p_i, p_j)$ 最小的一对点 (p_i, p_j) ，这里我们假设 P 中没有两个点有同样的 x 坐标或者 y 坐标

我们计划使用归并排序中的分治，我们找 P 的左半边点中最邻近点和在 P 右半边点中最邻近对，然后使用这些信息在线性时间内找到全局解

1. 我们有一系列点的集合 P' ，每个集合会维护两个表 P'_x （其中 P' 中所有的点都已经按照 x 坐标增长的顺序排好序）， P'_y （其中 P' 中所有的点都已经按照 y 坐标增长的顺序排好序）
2. 定义 Q 是在表 P_x 的前 $\frac{n}{2}$ 个位置， R 是在 P_x 的后 $\frac{n}{2}$ 个位置的点的集合
3. 通过递归访问 Q 和 R 部分，我们获得了 Q 和 R 表中的最邻近点对 q_0^* 和 q_1^* ，类似的，我们确定在 R 中的最邻近点对 r_0^* 和 r_1^*
4. 令 δ 是 $d(q_0^*, q_1^*)$ 和 $d(r_0^*, r_1^*)$ 中的最小值，实际问题是：是否存在这点的点 $q \in Q$ 和 $r \in R$ 使得 $d(q, r) < \delta$ ，如果不存在，则我们在递归调用中已经找到了最邻近点对，如果存在，那么 $d(q, r)$ 是最邻近点对
5. 设线 L 描述的是划分 Q 和 P 的垂直线，则搜索的 q 和 r 的限制范围一定在距离 δ 之内的一条窄带，令这一部分叫 S
6. 令 S_y 表示 S 中按照 y 坐标递增序排序的点组成的表
7. 如果存在 $q \in Q$ 与 $r \in R$ 使得 $d(q, r) < \delta$ 当且仅当存在 $s, s' \in S$ 使得 $d(s, s') < \delta$ ，且如果有 $d(s, s') < \delta$ ，那么 s 和 s' 在排好序的 S_y 中相距 11 个位置值内
8. 对于每个 $s \in S_y$ 我们计算它到跟着 S_y 15 个点中每个点的距离，如果它们的距离小于 δ ，那么最邻近点对在 S 中

算法伪代码如下：

```
1  Closest-Pair(P):
2      构造 Px 和 Py;
3      (p0*, p1*) = Closest-Pair-Rec(Px, Py)
4
5  Closest-Pair-Rec(Px, Py)
6      if (|P| ≤ 3){
```

```

7      由度量所有两点间的距离找出最邻近的点；
8      }
9      (q0*, q1*) = Closest-Pair-Rec(Qx, Qy);
10     (r0*, r1*) = Closest-Pair-Rec(Rx, Ry)
11     δ = min(d(q0*, q1*), d(r0*, r1*));
12     x* = 集合 Q 中点最大的 x 的坐标; // 划分线
13     L = {(x,y): x = x*};
14
15     删除所有距离 L 大于 δ 的点;
16     S = P 中与 L 相距在 δ 内的点集;
17     按照 y 坐标排序每个点, 构造 Sy;
18     for (每个 s ∈ Sy){
19         计算从 s 到 Sy 的 11 个点每个点的距离;
20         令 s, s'' 达到的距离;
21         δ = min(δ, d(s, s'')) < δ);
22     }
23
24     return δ

```

4.3 时间复杂度分析

构造 P_x 和 P_y 需要 $O(n \log n)$ 的时间, 在合并部分需要的时间复杂度为 $O(n)$, 故满足

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n \log n)$$

时间复杂度为 $O(n \log n)$

5. 整数乘法

5.1 问题描述

有可能用更快的算法来做乘法吗?

5.2 算法

我们把二进制数 x 写成 $x_1 \cdot 2^{\frac{n}{2}} + x_0$, x_1 对应高阶 $\frac{n}{2}$ 个位, x_0 对应低阶 $\frac{n}{2}$ 个位, 类似的, 我们写 $y = y_1 \cdot 2^{\frac{n}{2}} + y_0$

$$\begin{aligned}
 xy &= (x_1 \cdot 2^{\frac{n}{2}} + x_0) \times (y_1 \cdot 2^{\frac{n}{2}} + y_0) \\
 &= x_1 y_1 \cdot 2^n + (x_1 y_0 + x_0 y_1) \cdot 2^{\frac{n}{2}} + y_0
 \end{aligned}$$

等于把求解两个 n 位数规约成求解四个 $\frac{n}{2}$ 位的实例，故有

$$T(n) \leq 4T\left(\frac{n}{2}\right) + cn$$

即 $T(n) = O(n^2)$

实时上，我们可以采用三次递归调用来解决这个问题，进一步优化该算法

前面我们有 $xy = x_1y_1 \cdot 2^n + (x_1y_0 + x_0y_1) \cdot 2^{\frac{n}{2}} + x_0y_0$ ，考虑一次乘法的结果 $(x_1 + x_0)(y_1 + y_0) = x_1y_1 + x_0y_0 + x_0y_1 + x_1y_0$ 它以一次递归乘法的代价，实现了上述四个乘积的相加，如果现在我们调用递归确定 x_1y_1 与 x_0y_0 由于中间项为 $(x_1y_0 + x_0y_1) \cdot 2^{\frac{n}{2}}$ ，那么由于确定外面两个项，通过从 $(x_1 + x_0)(y_1 + y_0)$ 减去 x_1y_1 与 x_0y_0 得到中间的项

算法为

```
1 Recursive-Multiply:
2   写  $x = x_1 \cdot 2^{\{n/2\}} + x_0$ ;
3   写  $y = y_1 \cdot 2^{\{n/2\}} + y_0$ ;
4   计算  $x_1+x_0$  与  $y_1+y_0$ ;
5    $p = \text{Recursive-Multiply}(x_1+x_0, y_1+y_0)$ ;
6    $x_1y_1 = \text{Recursive-Multiply}(x_1, y_1)$ ;
7    $x_0y_0 = \text{Recursive-Multiply}(x_0, y_0)$ ;
8   return  $x_1y_1 \cdot 2^n + (p - x_1y_1 - x_0y_0) \cdot 2^{\{n/2\}} + x_0y_0$ ;
```

5.3 时间复杂度分析

根据最优解

$$T(n) \leq 3T\left(\frac{n}{2}\right) + cn$$

则 $T(n) = O(n^{\log_2 3}) = O(n^{1.59})$

6. 卷积与快速傅里叶变换

6.1 卷积的定义

给定两个长为 n 的向量 $a = (a_0, a_1, \dots, a_{n-1})$ 与 $b = (b_0, b_1, \dots, b_{n-1})$ 那么我们记 a 和 b 的卷积为 $a * b$ ，它是一个 $2n - 1$ 长的向量，其中坐标 k 上的值等于

$$\sum_{\substack{(i,j) i+j=k \\ i,j < n}} a_i b_j$$

考虑一张 $n \times n$ 的表，它的 (i, j) 项是 $a_i b_j$

$$\begin{array}{ccccc} a_0 b_0 & a_0 b_1 & \cdots & a_0 b_{n-2} & a_0 b_{n-1} \\ a_1 b_0 & a_1 b_1 & \cdots & a_1 b_{n-2} & a_1 b_{n-1} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{n-1} b_0 & a_{n-1} b_1 & \cdots & a_{n-1} b_{n-2} & a_{n-1} b_{n-1} \end{array}$$

通过沿着对角线求和来计算卷积向量中的坐标

$$(a_0 b_0, a_0 b_1 + a_1 b_0 + a_0 b_2 + a_1 b_1 + a_2 b_0, \dots, a_{n-1} b_{n-2} + a_{n-1} + b_{n-2}, a_{n-1} b_{n-1})$$

当然也可以把卷积推广到不同长度的 $a = (a_0, a_1, \dots, a_{m-1})$ 与 $b = (b_0, b_1, \dots, b_{n-1})$ ，那么 $a * b$ 是一个有 $m + n - 1$ 个坐标的向量，其中坐标 k 上的值等于

$$\sum_{\substack{(i,j) i+j=k \\ i < m, j < n}} a_i b_j$$

6.2 卷积的应用

多项式乘法

任何多项式 $A(x) = a_0 + a_1 x + a_2 x^2 + \dots + a_{m-1} x^{m-1}$ ，可以仅仅使用它的系数向量 $a = (a_0, a_1, \dots, a_m)$ 来表示同理 $B(x)$ 的系数向量为 $b = (b_0, b_1, \dots, b_{n-1})$

考虑它们的乘积多项式 $C(x) = A(x)B(x)$ ，在这个多项式中， x^k 项的系数等于

$$c_k = \sum_{i,j \ i+j=k} a_i b_j$$

换句话说， $C(x)$ 的系数向量 c 是 $A(x)$ 与 $B(x)$ 的系数向量的卷积

组合直方图问题

假设我们研究人口的分布，下面有两个直方图，一个显示人口中所有男人的年收入，一个显示所有女人的年收入

产生一个新的直方图，对每个 k 显示 (M, W) 对的个数，其中男人 M 和女人 W 组合收入为 k

这恰好是一个卷积，对任何满足 $i + j = k$ 的 (i, j) 对，选一个具有收入 i 的男人和一个具有收入 j 的女人的方法数

因此，组合直方图 $c = (c_0, c_1, \dots, c_{m+n-1})$ 不过是 M 和 W 的卷积

6.3 问题描述

有没有什么快速的方法来计算多项式？

多项式的系数表达与计算

一般多项式以系数形式表达，如果两个多项式

$$\begin{aligned} A(x) &= a_0 + a_1x + a_2x^2 + \dots + a_{n-1}x^{n-1} \\ B(x) &= b_0 + b_1x + b_2x^2 + \dots + b_{n-1}x^{n-1} \end{aligned}$$

加法

可以在 $O(n)$ 时间内完成

$$A(x) + B(x) = (a_0 + b_0) + (a_1 + b_1)x + \dots + (a_{n-1} + b_{n-1})x^{n-1}$$

估值

可以在 $O(n)$ 时间内完成

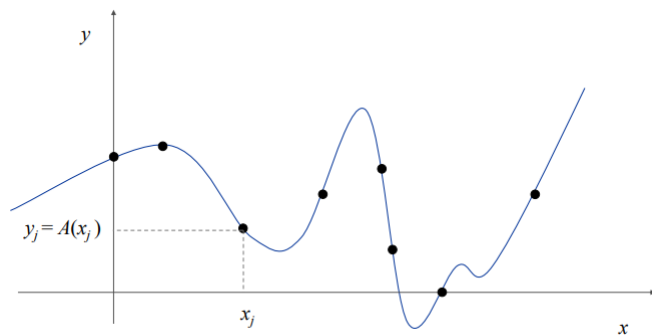
$$A(x) = a_0 + (x(a_1 + x(a_2 + \dots + x(a_{n-2} + x(a_{n-1}))))$$

乘法

暴力求解需要 $O(n^2)$ 完成

多项式的点集表达与计算

一个 $n - 1$ 次多项式 $A(x)$ 是由它在 x 的 n 个不同值处的计算唯一指定的



多项式如果以点集表达，则

$$A(x) : (x_0, y_0), \dots, (x_{n-1}, y_{n-1})$$

$$B(x) : (x_0, z_0), \dots, (x_{n-1}, z_{n-1})$$

加法

可以在 $O(n)$ 时间内完成

$$A(x) + B(x) = (x_0, y_0 + z_0), \dots, (x_{n-1}, y_{n-1} + z_{n-1})$$

乘法

可以在 $O(n)$ 时间内完成

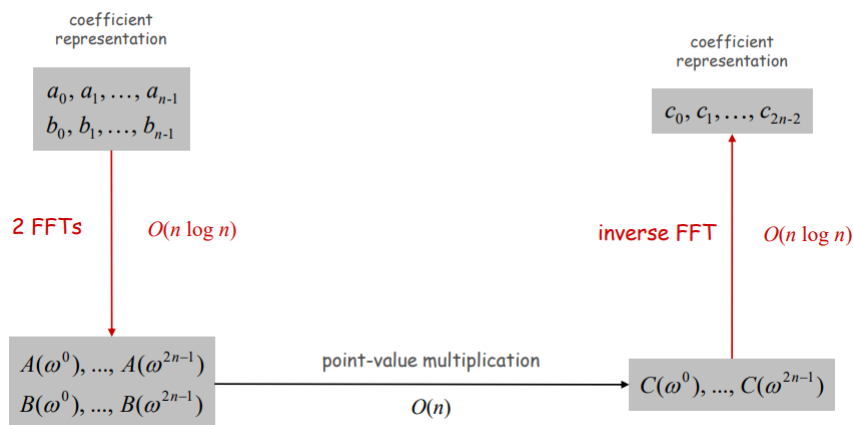
$$A(x) \times B(x) = (x_0, y_0 \times z_0), \dots, (x_{n-1}, y_{n-1} \times z_{n-1})$$

估值

暴力求解需要 $O(n^2)$ 完成

快速傅里叶变换实现效果

下面我们将介绍快速傅里叶变化，它将用 $O(n \log n)$ 的时间复杂度在系数形式与点集形式和之间转换，将所有基本运算优化到 $O(n \log n)$ 内



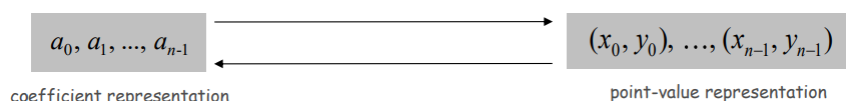
6.4 算法

目的

多项式的系数表达或点集在不同的运算上需要的时间复杂度不一样

表达	乘法	估值
系数表达	$O(n^2)$	$O(n)$
点集表达	$O(n)$	$O(n^2)$

我们希望结合两种表达的特性，所以我们想找到一种有效的两种形式的转换



系数形式转化成点集形式-暴力求解 $O(n^2)$

给出一个多项式 $a_0 + a_1x + \dots + a_{n-1}x^{n-1}$ ，把它以 n 个独立的点 $(x_0, y_0), \dots, (x_{n-1}, y_{n-1})$ 表示出来

矩阵乘法

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

点集形式转化成系数形式-暴力求解 $O(n^3)$

给定 n 个独立的点 $(x_0, y_0), \dots, (x_{n-1}, y_{n-1})$, 把它们表示成多项式的形式
 $a_0 + a_1x + \dots + a_{n-1}x^{n-1}$

高斯消元法

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

离散快速傅里叶变换 FFT (系数→点集)

Given a polynomial $a_0 + a_1x + \dots + a_{n-1}x^{n-1}$, evaluate it at n distinct points x_0, \dots, x_{n-1} .

Divide

假设我们有多项式 $A(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + a_4x^4 + a_5x^5 + a_6x^6 + a_7x^7$

它可以被分开为

- $A_{\text{even}}(x) = a_0 + a_2x + a_4x^2 + a_6x^3$
- $A_{\text{odd}}(x) = a_1 + a_3x + a_5x^2 + a_7x^3$

则我们有

- $A(x) = A_{\text{even}}(x^2) + xA_{\text{odd}}(x^2)$
- $A(-x) = A_{\text{even}}(x^2) - xA_{\text{odd}}(x^2)$

可以观察到, 对于复数点 ± 1 和 $\pm i$

- $A(1) = A_{\text{even}}(1) + 1A_{\text{odd}}(1)$
- $A(-1) = A_{\text{even}}(1) - 1A_{\text{odd}}(1)$
- $A(i) = A_{\text{even}}(-1) + iA_{\text{odd}}(-1)$
- $A(-i) = A_{\text{even}}(-1) - iA_{\text{odd}}(-1)$

离散傅里叶变换

记住，系数转化成点集的核心目的是需要快速挑出 n 个点

这样一组理想的数是单位 **1** 的复数根，对于正整数 n ，多项式方程 $x^n = 1$ 有 n 个不同的复数根，并且很容易找到

对于 $j = 0, 1, 2, \dots, n-1$ ，每个复数 $w_{j,n} = e^{\frac{2\pi ji}{n}}$ 满足它，因为

$$\left(e^{\frac{2\pi ji}{n}}\right)^n = e^{2\pi ji} = (e^{2\pi i})^j = 1^j = 1$$

这些数各不相同，因此它们都是根，可以把这些数看作单位 **1** 的 n 次方根

于是问题被进一步抽象成

Evaluate a degree $n-1$ polynomial $A(x) = a_0 + \dots + a_{n-1}x^{n-1}$ at its n th roots of unity: w_0, w_1, \dots, w_{n-1} .

这些数挑出来，进行矩阵乘法，同样也可以得出一系列点集

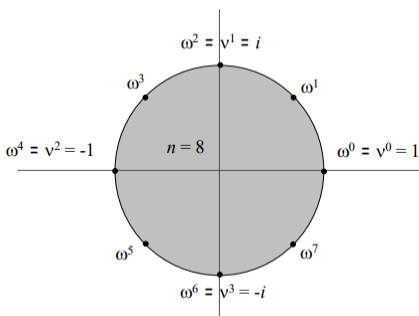
$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & \dots & 1 \\ 1 & \omega^1 & \omega^2 & \omega^3 & \dots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \omega^6 & \dots & \omega^{2(n-1)} \\ 1 & \omega^3 & \omega^6 & \omega^9 & \dots & \omega^{3(n-1)}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

\uparrow DFT \uparrow Fourier matrix F_n

我们以 $n = 8$ 为示例

我们设 $v_{j,n} = w_{j,n}^2 = e^{\frac{4\pi ji}{n}}$ 从 $v_0, v_1, \dots, v^{\frac{n}{2}-1}$

在上述条件下 $v_{j,8} = w^2_{j,8} = e^{\pi ji}$ ，共有 4 个点



还记得我们有多项式 $A(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + a_4x^4 + a_5x^5 + a_6x^6 + a_7x^7$

它可以被拆分为

- $A_{\text{even}}(x) = a_0 + a_2x + a_4x^2 + a_6x^3$
- $A_{\text{odd}}(x) = a_1 + a_3x + a_5x^2 + a_7x^3$
- $A(x) = A_{\text{even}}(x^2) + xA_{\text{odd}}(x^2)$
- $A(-x) = A_{\text{even}}(x^2) - xA_{\text{odd}}(x^2)$

Conquer

现在假设我们对每个 A_{even} 和 A_{odd} 在单位 1 的 n 次方根上求值，这恰好是 $v_0, v_1, \dots, v_{\frac{n}{2}-1}$ 这一组点，于是，我们对 A_{even} 和 A_{odd} 可以在 $T(\frac{n}{2})$ 的时间内实现求这些值，即

Combine

合起来是很容易的，即

$$\begin{aligned} A(w_{j,n}) &= A_{\text{even}}(w_{j,n}^2) + w_{j,n}A_{\text{odd}}(w_{j,n}^2) \\ A(w_{j+\frac{n}{2},n}) &= A_{\text{even}}(w_{j,n}^2) - w_{j,n}A_{\text{odd}}(w_{j,n}^2) \end{aligned}$$

注意到， $w_{j+\frac{n}{2},n}$ 其实相当于点 $w_{j,n}$ 在复数平面上转 180° ，也能在 $O(1)$ 的时间内计算出来

合并起所有的数需要 $O(n)$ 的时间

故算法为

```
1  FFT(n,a0,a1,...,an-1):
2
3      if(n == 1){
4          return a0;
5      }
6
7      // T(n/2) Divide and Conquer
8      (e0,e1,...,e_{n/2-1}) = FFT(n/2, a0,a2,a4,...,a_{n-2});
9      (d0,d1,...,d_{n/2-1}) = FFT(n/2, a1,a3,a5,...,a_{n-1});
10
11     // Combine
12     for (j = 0 to n/2-1){
13         w^j = e^{2πik/n};
14         y_j = e_j + w^j·d_j;
15         y_{j+n/2} = e_j - w^j·d_j;
16     }
17
18     return (y0,y1,...,y_{n-1})
```

离散快速傅里叶变换 IFFT (点集→系数)

Given n distinct points x_0, \dots, x_{n-1} and values y_0, \dots, y_{n-1} , find unique polynomial $a_0 + a_1x + \dots + a_{n-1}x^{n-1}$, that has given values at given points.

假设有一序列 $(x_0, y_0), (x_1, y_1), \dots, (x_{n-1}, y_{n-1})$

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

↑
↑
 Inverse DFT Fourier matrix inverse $(F_n)^{-1}$

傅里叶矩阵 F_n 的转置 $(F_n)^{-1}$ 为

$$G_n = \frac{1}{n} \begin{bmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega^{-1} & \omega^{-2} & \omega^{-3} & \cdots & \omega^{-(n-1)} \\ 1 & \omega^{-2} & \omega^{-4} & \omega^{-6} & \cdots & \omega^{-2(n-1)} \\ 1 & \omega^{-3} & \omega^{-6} & \omega^{-9} & \cdots & \omega^{-3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{-(n-1)} & \omega^{-2(n-1)} & \omega^{-3(n-1)} & \cdots & \omega^{-(n-1)(n-1)} \end{bmatrix}$$

为了计算 IFFT，使用相同的算法，但是注意 $w_j = e^{\frac{-2\pi i j}{n}}$ ，并且需要除以 n （证明过程略）

```

1  FFT(n,y0,y1,...,yn-1):
2
3      if(n == 1){
4          return y0;
5      }
6
7      // T(n/2) Divide and Conquer
8      (e0,e1,...,e_{n/2-1}) = FFT(n/2, y0,y2,y4,...,y_{n-2});
9      (d0,d1,...,d_{n/2-1}) = FFT(n/2, y1,y3,y5,...,y_{n-1});
10
11     // Combine
12     for (j = 0 to n/2-1){
13         w^j = e^{-2\pi i k/n};
14         a_j = (e_j + w^j \cdot d_j)/n;
15         a_{j+n/2} = (e_j - w^j \cdot d_j)/n;
16     }

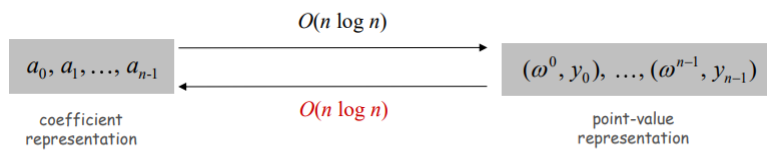
```

6.5 时间复杂度分析

FFT 和 IFFT 均满足

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n \log n)$$

时间复杂度为 $O(n \log n)$



6.6 应用

FFT 是世纪真正伟大的计算发展之一，它极大地改变了科学和工程的面貌，毫不夸张地说，如果没有 FFT，我们所知道的生活将会非常不同

- 光学，声学，量子物理，电信，雷达、控制系统、信号处理、语音识别、数据压缩、图像处理、地震学、质谱分析.....
- 数字媒体（DVD、JPEG、MP3）
- 内科诊断学（MRI、CT、PET扫描）
- 泊松方程的数值解
- 肖尔的量子分解算法