算法设计与分析

分治策略 (I)

目录

分治策略的起源：西方

分而治之（拉丁语：divide et impera），在政治学和社会学中是指通过将较大的权力集中体分解成单独实力较弱的部分来获取和维持权力的策略。

[见原文第 3 页图片：腓力二世像]

" 分而治之" 这一格言被归于马其顿的腓力二世。罗马统治者尤利乌斯 · 凯撒和法国皇帝拿破仑都曾使用过这一策略。

分治策略的起源：东方

故用兵之法，十则围之，五则攻之，倍则战之，敌则能分之，……

—《孙子兵法》

[见原文第 4 页图片：秦王扫六合时，虎视何雄哉]

分治策略作为算法设计范式

[见原文第 5 页图片]

分治策略通过以下步骤解决问题：

分治策略作为算法设计范式

[见原文第 5 页图片]

分治策略通过以下步骤解决问题：

1. **分解**：将原问题分解为若干个规模较小的、可以独立求解的子问题

分治策略作为算法设计范式

[见原文第 5 页图片]

分治策略通过以下步骤解决问题：

1. **分解**：将原问题分解为若干个规模较小的、可以独立求解的子问题
2. **求解**：递归或迭代地求解这些子问题
   - 当子问题足够小时，直接求解

分治策略作为算法设计范式

[见原文第 5 页图片]

分治策略通过以下步骤解决问题：

1. **分解**：将原问题分解为若干个规模较小的、可以独立求解的子问题
2. **求解**：递归或迭代地求解这些子问题
   - 当子问题足够小时，直接求解
3. **合并**：将子问题的解组合成原问题的解
   - 由算法的核心递归结构协调

为什么使用分治策略

并非总是如此，但通常比暴力算法表现更好

为什么使用分治策略

并非总是如此，但通常比暴力算法表现更好
最常见的用法（以排序为例）

- 在线性时间内将规模为 n 的问题分解为两个规模为 $n/2$ 的子问题
- 递归求解两个子问题
- 在线性时间内将两个解合并为整体解

为什么使用分治策略

并非总是如此，但通常比暴力算法表现更好
最常见的用法（以排序为例）

- 在线性时间内将规模为 n 的问题分解为两个规模为 $n/2$ 的子问题
- 递归求解两个子问题
- 在线性时间内将两个解合并为整体解

暴力算法：$\Theta(n^2)$ vs. 分治算法：$\Theta(n \log n)$

为什么使用分治策略

并非总是如此，但通常比暴力算法表现更好
最常见的用法（以排序为例）

- 在线性时间内将规模为 n 的问题分解为两个规模为 $n/2$ 的子问题
- 递归求解两个子问题
- 在线性时间内将两个解合并为整体解

暴力算法：$\Theta(n^2)$ vs. 分治算法：$\Theta(n \log n)$

在并行计算环境中特别适用（效率会更高）

需要思考的问题

简单地将问题分成若干部分并不能使其变得更容易。

- 如果你必须按顺序解决所有部分，由于分解和合并的开销，总工作量甚至可能增加。

[见原文第 7 页图片]

为什么分治策略实际上能提高效率?

通用分治算法

Algorithm 1 Divide-and-Conquer(P)

1: **if** $|P| \leq s^*$ **then**
2: Solve(P); // 直接求解
3: **else**
4: 将 P 分解为 $P_1, P_2, \ldots, P_k$; // 分解
5: **for** $i \leftarrow 1$ **to** k **do**
6: $y_i \leftarrow$ Divide-and-Conquer(P_i) // 求解子问题
7: **end for**
8: **return** Merge($y_1, y_2, \ldots, y_k$) // 合并答案
9: **end if**

分治策略的复杂度

递推关系：

$$\begin{cases} T(n) = T(|P_1|) + T(|P_2|) + \cdots + T(|P_k|) + f(n) \\ T(s^*) = C \end{cases}$$

- $P_1, P_2, \ldots, P_k$ 是分解后的子问题
- $f(n)$ 是分解子问题和合并子问题答案为原问题答案的复杂度
- C 是规模为 s^* 的最小子问题的复杂度

接下来，我们介绍两种典型的递推关系。

情况 1：子问题规模减少常数

$$T(n) = \sum_{i=1}^{k} a_i T(n-i) + f(n)$$

求解方法

1. 迭代法（直接迭代或化简后迭代）
2. 递归树

例. 汉诺塔：$T(n) = 2\,T(n-1) + 1$

情况 2：子问题规模线性减少

$$T(n) = aT\left(\frac{n}{b}\right) + f(n), \quad h(n) = n^{\log_b a}$$

求解方法：递归树、主定理

$$T(n) = \begin{cases} \Theta(h(n)) & \text{若} f(n) = o(h(n)) \\ \Theta(h(n)\log n) & \text{若} f(n) = \Theta(h(n)) \\ \Theta(f(n)) & \text{若} f(n) = \omega(h(n)) \\ & \wedge \exists\ r < 1 \text{ s.t. } af(n/b) < rf(n) \end{cases}$$

例 1. 二分查找：$W(n) = W(n/2) + 1$
例 2. 归并排序：$W(n) = 2W(n/2) + (n-1)$

接下来，我们通过几个入门示例来阐释分治策略的主要思想。

汉诺塔

Algorithm 2 Hanoi(A, C, n) // 将 n 个圆盘从 A 移到 C

输入 $A(n), B(0), C(0)$
输出 $A(0), B(0), C(n)$

```
1: if n = 1 then
2:     move(A, C);                          // 将一个圆盘从 A 移到 C
3: else
4:     Hanoi(A, B, n − 1);
5:     move(A, C);
6:     Hanoi(B, C, n − 1)
7: end if
```

汉诺塔的复杂度

1. 将原问题归约为两个规模为 $n-1$ 的子问题
2. 继续归约直到子问题规模为 1
3. 从输入规模 1 到 $n-1$，合并答案直到规模回到 n

设 $T(n)$ 为移动 n 个圆盘的复杂度：所需的最少移动次数

$$\begin{cases} T(n) = 2\,T(n-1) + 1 \\ T(1) = 1 \end{cases} \Rightarrow T(n) = 2^n - 1$$

对于此问题不存在最坏情况、最好情况、平均情况的区分，因为输入仅依赖于输入规模。

二分查找

二分查找的复杂度

1. 通过将 x 与中位数比较，将原问题归约为规模减半的子问题：
   - 若 $x \le A[m]$，则 $A[l,r] := A[l,m]$，否则 $A[l,r] := A[m+1,r]$
2. 重复搜索 T 直到其规模变为 1，即 $l = r$
   - 此时，直接比较 x 和 $A[l]$，相等则返回 l，否则返回"0"

二分查找的最坏情况复杂度

$$h(n) = 1, f(n) = \Theta(h(n)) \Rightarrow \text{主定理（情况 2）}$$

$$\begin{cases} W(n) = W(\lceil n/2 \rceil) + 1 \\ W(1) = 1 \end{cases} \Rightarrow W(n) = \Theta(\log n)$$

分治策略范式回顾

我们通过示例展示了分治策略范式的特点：

- 将原问题分解为规模较小的独立子问题
  - 子问题与原问题属于同一类型
  - 当子问题足够小时，可以直接求解
- 算法可以递归或迭代地求解
- 复杂度分析：求解递推关系

目录

归并排序

Algorithm 3 MergeSort(A, n)

输入 未排序的 $A[n]$
输出 升序排列的 $A[n]$

```
1: l ← 1, r ← n;
2: if l < r then
3:     m ← ⌊(l + r)/2⌋                          // 对半分割
4:     MergeSort(A, l, m)                       // 子问题 1
5:     MergeSort(A, m + 1, r)                   // 子问题 2
6:     Merge(A[l, m], A[m + 1, r])              // 合并已排序的子数组
7: end if
```

如何递归地实现 Merge?

递归合并算法

Algorithm 4 Merge$(A[1,k], B[1,l])$

```
1: if k = 0 then
2:     return B[1, l];
3: end if
4: if l = 0 then
5:     return A[1, k];
6: end if
7: if A[1] ≤ B[1] then
8:     return A[1]∘ Merge(A[2, k], B[1, l]);
9: else
10:    return B[1]∘ Merge(A[1, k], B[2, l]);
11: end if
```

Merge 过程每次递归调用执行常数量的工作，总运行时间为 $O(k+l)$。

归并排序的复杂度

1. 将原问题分解为 2 个规模为 $n/2$ 的子问题
2. 继续分解直到子问题规模为 1
3. 从输入规模 1 到 $n/2$，合并两个相邻的已排序子数组
   - 每次合并后子数组规模翻倍，直到达到原始规模

假设 $n = 2^k$，归并排序的最坏情况复杂度为：

$$h(n) = n, f(n) = \Theta(h(n)) \Rightarrow \text{主定理（情况 2）}$$

$$\begin{cases} W(n) = 2\,W(n/2) + n - 1 \\ W(1) = 0 \end{cases} \Rightarrow W(n) = \Theta(n \log n)$$

归并排序回顾

由于数组的数据结构，子问题的分解已经完成，所有真正需要做的工作是合并。

归并排序回顾

由于数组的数据结构，子问题的分解已经完成，所有真正需要做的工作是合并。
这一观点表明归并排序可以以自底向上的方式从单元素数组迭代到原始数组。

归并排序回顾

由于数组的数据结构，子问题的分解已经完成，所有真正需要做的工作是合并。
这一观点表明归并排序可以以自底向上的方式从单元素数组迭代到原始数组。

[见原文第 22 页图片：归并排序示意图]

目录

基本思想

1. 选择第一个元素 x 作为枢轴，将 A 划分为两个子数组：
   - 低子数组 A_L：小于 x 的元素
   - 高子数组 A_R：大于 x 的元素
   - x 处于正确的位置
2. 递归排序 A_L 和 A_R，直到子数组规模为 1

快速排序伪代码

Algorithm 5 QuickSort(A, l, r)

输入 $A[l \dots r]$
输出 升序排列的 A

```
1: if l = r then
2:     return ;                                  // 到达最小情况
3: end if
4: if l < r then
5:     k ← Partition(A, l, r);
6:     A[l] ↔ A[k];
7:     QuickSort(A, l, k − 1);
8:     QuickSort(A, k + 1, r);
9: end if
```

与归并排序相比，快速排序没有合并步骤，所有真正需要做的工作是分解。

划分的伪代码

Algorithm 6 Partition(A, l, r)

```
1:  x ← A[l]                                   // 将第一个元素设为枢轴
2:  i ← l, j ← r + 1                           // 初始化左/右指针
3:  while true do
4:      repeat
5:          j ← j − 1
6:      until A[j] ≤ x                         // 小于 x
7:      repeat
8:          i ← i + 1
9:      until A[i] > x                         // 大于 x
10:     if i < j then
11:         A[i] ↔ A[j];
12:     else
13:         return j;                          // 交叉发生，找到位置
14:     end if
15: end while
```

划分演示

[见原文第 27 页图片：划分过程演示]

复杂度分析

最坏情况:

$$\begin{cases} W(n) = W(n-1) + n - 1 \\ W(1) = 0 \end{cases} \Rightarrow W(n) = n(n-1)/2$$

最好情况:

$$\begin{cases} T(n) = 2\,T(n/2) + n - 1 \\ T(1) = 0 \end{cases} \Rightarrow T(n) = \Theta(n \log n)$$

常数划分的复杂度

常数划分. |子问题|/|原问题| 是一个固定常数，例如 $1:9$。

$$\begin{cases} T(n) = T(n/10) + T(9n/10) + n \\ T(1) = 0 \end{cases}$$

通过递归树求解：$\Rightarrow T(n) = \Theta(n \log n)$

[见原文第 29 页图片：递归树示意图]

平均情况复杂度

假设第一个元素最终出现在位置 $1, 2, \ldots, n$ 的概率相等，即 $1/n$。分析产生的子问题规模：

- 出现在位置 1： $T(0), T(n-1)$
- 出现在位置 2： $T(1), T(n-2)$
- $\ldots$
- 出现在位置 $n-1$： $T(n-2), T(1)$
- 出现在位置 n： $T(n-1), T(0)$

所有子问题的代价：$2(T(1) + T(2) + \cdots + T(n-1))$
划分的代价：$n-1$ 次比较

平均情况复杂度

$$T(n) = \frac{1}{n}\sum_{k=1}^{n-1}(T(k) + T(n-k)) + n - 1$$

$$\begin{cases} T(n) = \frac{2}{n}\sum_{k=1}^{n-1} T(k) + n - 1 \\ T(1) = 0 \end{cases}$$

通过相减法化简递推关系 $\Rightarrow$

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

如果忘记了，请参见第 3 讲第 47 页。

目录

芯片测试

芯片工厂只允许基本测试方法。

基本测试方法. 将两块芯片 A 和 B 放在测试台上，开始互测

- 测试报告为" 好" 或" 坏"

[见原文第 33 页图片：芯片互测示意图]

假设. 好芯片的报告总是正确的，但坏芯片的报告是不确定的（可能出错）

测试报告分析

A 的报告	B 的报告	结论
B 是好的	A 是好的	A, B 同好或同坏
B 是好的	A 是坏的	至少有一个是坏的
B 是坏的	A 是好的	至少有一个是坏的
B 是坏的	A 是坏的	至少有一个是坏的

芯片测试问题

输入. n 块芯片，$\#(\text{好}) - \#(\text{坏}) \geq 1$
问题. 设计一种测试方法，从 n 块芯片中选出一块好芯片
要求. 互测次数最少

芯片测试问题

输入. n 块芯片，$\#(\text{好}) - \#(\text{坏}) \geq 1$
问题. 设计一种测试方法，从 n 块芯片中选出一块好芯片
要求. 互测次数最少

出发点. 给定一块芯片 A，如何检验 A 是好是坏
方法. 用其他 $n-1$ 块芯片测试 A。

- 思路：利用 n 的奇偶性

情况 1：n 为奇数

例. $n = 7$，#(好芯片) ≥ 4。

- A 是好的 $\Rightarrow$ 6 份报告中至少 3 份报" 好"
- A 是坏的 $\Rightarrow$ 6 份报告中至少 4 份报" 坏"

推广到 n 为奇数，#(好芯片) $\geq (n+1)/2$。

- A 是好的：$\Rightarrow$ 至少 $(n-1)/2$ 份报告报" 好"
- A 是坏的：$\Rightarrow$ 至少 $(n+1)/2$ 份报告报" 坏"

关键观察. 测试结果是互斥的，因此测试条件实际上是充分必要的。
判定准则：在 $n-1$ 份报告中

- 至少一半报" 好" $\Rightarrow$ A 是好的
- 超过一半报" 坏" $\Rightarrow$ A 是坏的

情况 2：n 为偶数

例. $n = 8$，#(好芯片) ≥ 5。

- A 是好的 $\Rightarrow$ 7 份报告中至少 4 份报" 好"
- A 是坏的 $\Rightarrow$ 7 份报告中至少 5 份报" 坏"

推广到 n 为偶数，#(好芯片) $\geq n/2 + 1$。

- A 是好的 $\Rightarrow$ 至少 $n/2$ 份报告报" 好"
- A 是坏的 $\Rightarrow$ 至少 $n/2 + 1$ 份报告报" 坏"

关键观察. 测试结果是互斥的，因此测试条件也是充分必要的。
判定准则：在 $n - 1$ 份报告中

- 至少一半报" 好" $\Rightarrow$ A 是好的
- 超过一半报" 坏" $\Rightarrow$ A 是坏的

暴力算法

测试方法. 随机选取一块芯片，应用前述测试。如果是好的，测试结束。否则，丢弃它并从剩余芯片中随机选取另一块，直到得到一块好芯片。

- 正确性：$\#$(好芯片) 始终超过一半。

时间复杂度

- 第 1 轮：随机选中坏芯片，最多 $n-1$ 次测试
- 第 2 轮：随机选中坏芯片，最多 $n-2$ 次测试
- ...
- 第 i 轮：随机选中坏芯片，最多 $n-i$ 次测试

暴力算法

测试方法. 随机选取一块芯片，应用前述测试。如果是好的，测试结束。否则，丢弃它并从剩余芯片中随机选取另一块，直到得到一块好芯片。

- 正确性：$\#$(好芯片) 始终超过一半。

时间复杂度

- 第 1 轮：随机选中坏芯片，最多 $n-1$ 次测试
- 第 2 轮：随机选中坏芯片，最多 $n-2$ 次测试
- ...
- 第 i 轮：随机选中坏芯片，最多 $n-i$ 次测试

最坏情况下的总体复杂度是 $\Theta(n^2)$

优化

江锴杰 的精彩发现：在 $i > 1$ 轮，我们可以先随机丢弃一块芯片再测试；最多需要 $n - 1 - 2i$ 次测试

优化

江锴杰 的精彩发现：在 $i > 1$ 轮，我们可以先随机丢弃一块芯片再测试；最多需要 $n - 1 - 2i$ 次测试

赵渊宁 的另一种可能优化：在 $i > 1$ 轮，如果我们记录测试结果，那么可以丢弃所有对坏芯片报" 好" 的芯片

优化

江锴杰 的精彩发现：在 $i > 1$ 轮，我们可以先随机丢弃一块芯片再测试；最多需要 $n - 1 - 2i$ 次测试

赵渊宁 的另一种可能优化：在 $i > 1$ 轮，如果我们记录测试结果，那么可以丢弃所有对坏芯片报" 好" 的芯片

然而，上述技巧并不能改变最坏情况复杂度。

分治策略

假设 n 为偶数，将 n 块芯片分成两组并开始互测；剩余的芯片形成子问题并进入下一轮测试

分治策略

假设 n 为偶数，将 n 块芯片分成两组并开始互测；剩余的芯片形成子问题并进入下一轮测试

测试-淘汰规则

- " 好，好" $\Rightarrow$ 随机选一块进入下一轮
- 其他情况 $\Rightarrow$ 全部丢弃

分治策略

假设 n 为偶数，将 n 块芯片分成两组并开始互测；剩余的芯片形成子问题并进入下一轮测试

测试-淘汰规则

- " 好，好" $\Rightarrow$ 随机选一块进入下一轮
- 其他情况 $\Rightarrow$ 全部丢弃

递归结束条件：$n \leq 3$

- 3 块芯片：一次测试足够（想想为什么？保持与原问题相同的性质）
  1. " 好，好"：随机选一块输出
  2. " 好，坏"：输出剩下的那块
  3. " 坏，坏"：输出剩下的那块
- 1 或 2 块芯片：都是好的，不需要更多测试

分治算法的正确性

命题. 当 n 为偶数时，一轮测试后，在剩余芯片中，$\#(\text{好芯片}) - \#(\text{坏芯片}) \geq 1$

证明. 考虑以下三种情况：

1. 两块都是好的（i 组）$\Rightarrow$ 随机保留一块
2. 一好一坏（j 组）$\Rightarrow$ 全部丢弃
3. 两块都是坏的（k 组）$\Rightarrow$ 随机保留一块或全部丢弃

一轮测试后，$\#(\text{好芯片}) = i$，$\#(\text{坏芯片}) \leq k$

$$\begin{cases} 2i + 2j + 2k = n & \#(\text{测试前的芯片}) \\ 2i + j > 2k + j & \#(\text{好芯片}) > \#(\text{坏芯片}) \end{cases} \Rightarrow i > k$$

n 为奇数时的调整

当 n 为奇数时，会有一块芯片没有配对。

[见原文第 42 页图片：奇数情况处理示意图]

调整. 当 n 为奇数时，对未配对的芯片增加一轮直接测试

- 如果是好的，算法结束
- 否则，丢弃它并进入下一轮（因为 $n-1$ 块芯片满足原始性质）

伪代码

复杂度分析

对于输入规模 n，每轮测试后，芯片数量至少减半

- #(测试)（包括 n 为奇数时的额外调整）：$\Theta(n)$

递推关系

$$\begin{cases} W(n) = W(n/2) + \Theta(n) \\ W(3) = 1, W(2) = W(1) = 0 \end{cases} \Rightarrow W(n) = \Theta(n)$$

分治芯片测试算法总结

- 调整 $\Rightarrow$ 保证子问题与原问题属于同一类型
- 分支因子 $a = 1$ & 分解-合并代价 $f(n) = \Theta(n) \Rightarrow$ 确保相对于暴力算法有显著的效率提升

目录

目录

一般选择问题

选择. 给定全序集合 S 中的 n 个元素，找到第 k 小的元素。

- 最小值：$k = 1 \Rightarrow$ 最小元素
- 最大值：$k = n \Rightarrow$ 最大元素
- 中位数：$k = \lfloor (n+1)/2 \rfloor$
  - n 为奇数，中位数唯一，$k = (n+1)/2$
  - n 为偶数，中位数有两个选择：$n/2$ 和 $n/2+1$，通常选择 $k = n/2$

已知结果

- 选择最小值或最大值：$O(n)$ 次比较
- 一般选择的朴素算法：排序需 $O(n \log n)$ 次比较，用二叉堆需 $O(n \log k)$ 次比较

应用. 顺序统计量；选择" 前 k 个"；瓶颈路径

问. 一般选择能否用 $O(n)$ 次比较完成?

答. 可以！选择比排序更容易。

关于中位数

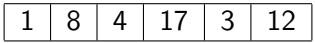
数字列表的中位数是其第 50 百分位数：一半的数字比它大，一半比它小。
例. $[45, 1, 10, 30, 25]$ 的中位数是 25。
中位数的意义. 用单个典型值来概括一组数字。

- 均值或平均值也常用于此目的。
- 但是，中位数在某种意义上更具代表性：
  - 总是数据值之一，不像均值
  - 对异常值不敏感

反例. 数百个 1 的中位数是 1，均值也是 1。然而，如果其中一个数字被篡改为 10000，均值会飙升到 100 以上，而中位数很大概率不受影响。

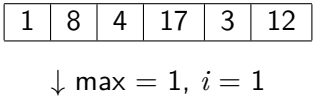
选择最大值

算法. 顺序比较



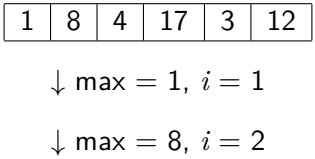
选择最大值

算法. 顺序比较



选择最大值

算法. 顺序比较



选择最大值

算法. 顺序比较

1	8	4	17	3	12

$\downarrow$ max $= 1$, $i = 1$

$\downarrow$ max $= 8$, $i = 2$

$\downarrow$ max $= 8$, $i = 2$ （不变）

选择最大值

算法. 顺序比较

1	8	4	17	3	12

$\downarrow$ max $= 1$, $i = 1$

$\downarrow$ max $= 8$, $i = 2$

$\downarrow$ max $= 8$, $i = 2$ （不变）

$\downarrow$ max $= 17$, $i = 4$

选择最大值

算法. 顺序比较

1	8	4	17	3	12

$\downarrow$ max $= 1$, $i = 1$

$\downarrow$ max $= 8$, $i = 2$

$\downarrow$ max $= 8$, $i = 2$ （不变）

$\downarrow$ max $= 17$, $i = 4$

$\downarrow$ max $= 17$, $i = 4$ （不变）

输出. max $= 17$, $i = 4$
最坏情况复杂度. $W(n) = n - 1$

选择最大值伪代码

Algorithm 7 SelectMax(A, n)

输入 $A[n]$
输出 max, j

```
1: max ← A[1];
2: j ← 1;
3: for i ← 2 to n do
4:     if max < A[i] then
5:         max ← A[i];
6:         j ← i;
7:     end if
8: end for
9: return max, j
```

选择最大值和最小值

朴素算法

1. 顺序比较，先选出最大值并移除
2. 然后在剩余列表中选出最小值，使用相同的算法但每次比较后保留较小的元素

选择最大值和最小值

朴素算法

1. 顺序比较，先选出最大值并移除
2. 然后在剩余列表中选出最小值，使用相同的算法但每次比较后保留较小的元素

最坏情况时间复杂度

$$W(n) = n - 1 + n - 2 = 2n - 3$$

分组算法

思路. 将列表分为高列表和低列表。

[见原文第 52 页图片：分组算法示意图]

分组算法

思路. 将列表分为高列表和低列表。

[见原文第 52 页图片：分组算法示意图]

选择最大值和最小值的分组方法伪代码

Algorithm 8 SelectMaxMin(A, n)

输入 未排序的 $A[n]$

输出 max, min

1: 将 n 个元素分成 $\lfloor n/2 \rfloor$ 组;
2: 比较每组中的两个元素，得到 $\lfloor n/2 \rfloor$ 个较小值和 $\lfloor n/2 \rfloor$ 个较大值;
3: 在 $\lfloor n/2 \rfloor$ 个较大元素和额外元素中选择 max;
4: 在 $\lfloor n/2 \rfloor$ 个较小元素和额外元素中选择 min;

选择最大值和最小值的分组方法伪代码

Algorithm 9 SelectMaxMin(A, n)

输入 未排序的 $A[n]$

输出 max, min

1: 将 n 个元素分成 $\lfloor n/2 \rfloor$ 组;
2: 比较每组中的两个元素，得到 $\lfloor n/2 \rfloor$ 个较小值和 $\lfloor n/2 \rfloor$ 个较大值;
3: 在 $\lfloor n/2 \rfloor$ 个较大元素和额外元素中选择 max;
4: 在 $\lfloor n/2 \rfloor$ 个较小元素和额外元素中选择 min;

总结，$W(n) \approx 3\lfloor n/2 \rfloor$

- 组内比较：$\lfloor n/2 \rfloor$
- 当 n 为偶数时：选 max：$n/2-1$，选 min：$n/2-1$
- 当 n 为奇数时：选 max：$(n-1)/2+1-1$，选 min：$(n-1)/2+1-1$

分治策略

分组算法优于朴素算法，因为组内比较花费 $\lfloor n/2 \rfloor$ 次比较，但节省了约 $(n/2) \times 2$ 次比较。

分治策略

分组算法优于朴素算法，因为组内比较花费 $\lfloor n/2 \rfloor$ 次比较，但节省了约 $(n/2) \times 2$ 次比较。

我们能否通过分治策略设计 SelectMaxMin?

分治策略

分组算法优于朴素算法，因为组内比较花费 $\lfloor n/2 \rfloor$ 次比较，但节省了约 $(n/2) \times 2$ 次比较。

我们能否通过分治策略设计 SelectMaxMin?

1. 将 A 分成左半部分 A_1 和右半部分 A_2
2. 递归选择 A_1 中的 max_1 和 min_1
3. 递归选择 A_2 中的 max_2 和 min_2
4. $\mathsf{max} \leftarrow \mathsf{max}\{\mathsf{max}_1, \mathsf{max}_2\}$
5. $\mathsf{min} \leftarrow \mathsf{min}\{\mathsf{min}_1, \mathsf{min}_2\}$

分治策略

分组算法优于朴素算法，因为组内比较花费 $\lfloor n/2 \rfloor$ 次比较，但节省了约 $(n/2) \times 2$ 次比较。

我们能否通过分治策略设计 SelectMaxMin?

1. 将 A 分成左半部分 A_1 和右半部分 A_2
2. 递归选择 A_1 中的 max_1 和 min_1
3. 递归选择 A_2 中的 max_2 和 min_2
4. $\mathsf{max} \leftarrow \mathsf{max}\{\mathsf{max}_1, \mathsf{max}_2\}$
5. $\mathsf{min} \leftarrow \mathsf{min}\{\mathsf{min}_1, \mathsf{min}_2\}$

[2020 游泓慧] 这个递归算法可以像归并排序一样做成迭代的。

最坏情况复杂度

假设 $n = 2^k$，$W(n)$ 的递推关系如下：

$$\begin{cases} W(n) = 2\,W(n/2) + 2 \\ W(2) = 1 \end{cases}$$

通过换元-迭代法求精确值：

$$\begin{aligned} W(2^k) &= 2\,W(2^{k-1}) + 2 \\ &= 2(2\,W(2^{k-2}) + 2) + 2 \\ &= 2^2\,W(2^{k-2}) + 2^2 + 2 \\ &= 2^i\,W(2^{k-i}) + 2^i + \cdots + 2 \end{aligned}$$

当 $i = k - 1$ 时右边达到初始值，求和为：

$$2^{k-1} + (2^{k-1} + \cdots + 2^2 + 2) = 3 \cdot 2^{k-1} - 2 = 3n/2 - 2$$

总结

选择最大值. 顺序比较，最多需要 $n-1$ 次比较

选择最大值和最小值（最坏情况）

- 朴素算法：$2n-3$
- 分组算法：$3\lfloor n/2 \rfloor$
- 分治策略：$3n/2-2$

可以证明分组算法和分治算法对于 SelectMaxMin 是最优的，达到了下界。

目录

选择次大值

输入. $A[n]$
输出. 次大值 max′

选择次大值

输入. $A[n]$
输出. 次大值 max′

朴素算法：顺序比较

1. 通过顺序比较从 $A[n]$ 中选出 max
2. 从 $A[n] \setminus \{\text{max}\}$ 中选出 max′，这正是次大值

时间复杂度： $W(n) = (n-1) + (n-2) = 2n - 3$

优化方法

观察. 成为次大值的充分必要条件：只被最大值击败
为了确定次大元素，我们必须先知道最大元素。

优化方法

观察. 成为次大值的充分必要条件：只被最大值击败
为了确定次大元素，我们必须先知道最大元素。

思路. 以空间换时间

- 在选择最大值的过程中，将被最大元素击败的元素记录在集合 L 中
- 在 L 中的元素中选择最大元素

选择次大值的锦标赛算法

1. 将元素分成大小为 2 的组
2. 在每组中，两个元素比较，较大的进入下一层，并（仅）在其列表中记录被击败的元素
3. 重复上述步骤直到只剩一个元素，即 max
4. 从 max 的列表中选择最大元素，即 $\max'$

名称来源于单淘汰赛制：选手进行两两对决，胜者晋级下一轮。这一层次结构持续到决赛决出最终胜者。锦标赛确定了最佳选手，但在决赛中被击败的选手可能不是第二好的——他可能不如胜者之前击败的其他选手。

SelectSecond 伪代码

Algorithm 10 SelectSecond(A, n)

输入 $A[n]$

输出 次大元素 max′

1: $k \leftarrow n$ // 元素数量
2: 将 k 个元素分成 $\lfloor k/2 \rfloor$ 组;
3: 在每组中，两个元素比较选出较大者;
4: 将败者记入胜者的列表;
5: **if** k 为奇数 **then**
6: $k \leftarrow 1 + \lfloor k/2 \rfloor$;
7: **else**
8: $k \leftarrow k/2$;
9: **end if**
10: **if** $k > 1$ **then**
11: **goto** 2;
12: **end if**
13: max′ ← 最终胜者;

SelectSecond 演示

[见原文第 62 页图片：锦标赛算法演示]

复杂度分析 (1/3)

命题 1. 假设有 n 个元素，第 i 轮比赛后最多剩余 $\lceil n/2^i \rceil$ 个元素。
证明. 对 i 进行数学归纳：
归纳基础 $i = 1$：分成 $\lfloor n/2 \rfloor$ 组，淘汰 $\lfloor n/2 \rfloor$ 个元素，晋级下一层的元素数量为

$$n - \lfloor n/2 \rfloor = \lceil n/2 \rceil$$

归纳步骤：$P(i) \Rightarrow P(i+1)$。假设第 i 轮比赛后元素数量最多为 $\lceil n/2^i \rceil$，则第 $i+1$ 轮比赛后，元素数量为

$$\text{连续取整性质} \Rightarrow \lceil \lceil n/2^i \rceil / 2 \rceil = \lceil n/2^{i+1} \rceil$$

复杂度分析 (2/3)

命题 2. max 与 $\lceil \log n \rceil$ 个元素进行过比较

证明. 假设 max 在第 k 轮比赛后被选出。根据命题 1，$\lceil n/2^k \rceil = 1$。

- 若对某个 $d \in \mathbb{Z}$，$n = 2^d$，则：

$$\log n = \lceil \log n \rceil, \quad k = d = \lceil \log n \rceil$$

- 否则对某个 $d \in \mathbb{Z}$，$2^d < n < 2^{d+1}$，则：

$$d < \log n < d + 1, \quad k = d + 1 = \lceil \log n \rceil$$

复杂度分析 (3/3)

阶段 1：元素数量为 n

- 总比较次数 $= n-1 \Leftarrow n-1$ 个元素被淘汰（一次比较淘汰一个元素）

阶段 2：元素数量为 $\lceil \log n \rceil$，根据命题 2 这正是胜者列表的大小

- 比较次数 $= \lceil \log n \rceil - 1 \Leftarrow$ 顺序比较或锦标赛算法（$\lceil \log n \rceil - 1$ 个元素被淘汰）

总体时间复杂度：

$$W(n) = n - 1 + \lceil \log n \rceil - 1 = n + \lceil \log n \rceil - 2$$

选择次大值总结

选择次大值

- 朴素算法（调用 SelectMax 两次）：$2n-3$
- 锦标赛算法：$n+\lceil \log n \rceil -2$
  - 主要技巧：以空间换效率

目录

一般选择问题

问题. 选择第 k 小的元素
输入. 列表 $A[n]$，整数 $k \in [n]$
输出. 第 k 小的元素

一般选择有广泛的应用

- **例 1**. $A = \{3, 4, 8, 2, 5, 9, 10\}$，$k = 4$，解 $= 5$
- **例 2**. 统计数据集 S，$|S| = n$，选择中位数，$k = \lceil n/2 \rceil$

朴素算法

算法 1

- 调用 SelectMin 算法 k 次
- 时间复杂度：$O(kn)$

朴素算法

算法 1

- 调用 SelectMin 算法 k 次
- 时间复杂度：$O(kn)$

算法 2

- 排序后选择第 k 小的数
- 时间复杂度：$O(n \log n)$

朴素算法

算法 1

- 调用 SelectMin 算法 k 次
- 时间复杂度：$O(kn)$

算法 2

- 排序后选择第 k 小的数
- 时间复杂度：$O(n \log n)$

理想情况下我们期望线性复杂度

- 这是有希望的，因为排序做的远比我们真正需要的多——我们不关心其余元素的相对顺序。

快速选择

[Hoa71] Algorithm 65: Find

[见原文第 70 页图片：Hoare 论文]

快速选择

[Hoa71] Algorithm 65: Find

[见原文第 70 页图片：Hoare 论文]

快速选择使用与快速排序相同的整体方法——选择一个元素 m^* 作为枢轴来划分 S，使 m^* 就位，较小元素在左子数组 S_1，较大元素在右子数组 S_2

1. 若 $k \leq |S_1|$，则在 S_1 中选择第 k 小
2. 若 $k = |S_1| + 1$，则 m^* 就是第 k 小
3. 若 $k > |S_1| + 1$，则在 S_2 中选择第 $k - |S_1| - 1$ 小

快速选择

[Hoa71] Algorithm 65: Find

[见原文第 70 页图片：Hoare 论文]

快速选择使用与快速排序相同的整体方法——选择一个元素 m^* 作为枢轴来划分 S，使 m^* 就位，较小元素在左子数组 S_1，较大元素在右子数组 S_2

1. 若 $k \leq |S_1|$，则在 S_1 中选择第 k 小
2. 若 $k = |S_1| + 1$，则 m^* 就是第 k 小
3. 若 $k > |S_1| + 1$，则在 S_2 中选择第 $k - |S_1| - 1$ 小

与快速排序递归进入两边不同，快速选择只递归进入一边——包含所找元素的那一边。

- 这将平均情况复杂度从 $O(n \log n)$ 降低到 $O(n)$
- 最好情况复杂度是 $O(n)$，而最坏情况复杂度是 $O(n^2)$

如何改进最坏情况复杂度?

与快速排序一样，快速选择的复杂度由枢轴的质量决定

- 总是选择第一个元素作为枢轴在实践中是高效的，平均情况性能好，但最坏情况性能差

如何改进最坏情况复杂度?

与快速排序一样，快速选择的复杂度由枢轴的质量决定

- 总是选择第一个元素作为枢轴在实践中是高效的，平均情况性能好，但最坏情况性能差

如何选择好的枢轴 m^* 以获得好的平均情况和最坏情况性能?

如何改进最坏情况复杂度?

与快速排序一样，快速选择的复杂度由枢轴的质量决定

- 总是选择第一个元素作为枢轴在实践中是高效的，平均情况性能好，但最坏情况性能差

如何选择好的枢轴 m^* 以获得好的平均情况和最坏情况性能?

- 随机选择并不能改进最坏情况复杂度（恰好选到坏的随机数）

如何改进最坏情况复杂度?

与快速排序一样，快速选择的复杂度由枢轴的质量决定

- 总是选择第一个元素作为枢轴在实践中是高效的，平均情况性能好，但最坏情况性能差

如何选择好的枢轴 m^* 以获得好的平均情况和最坏情况性能?

- 随机选择并不能改进最坏情况复杂度（恰好选到坏的随机数）

理想情况：选择精确的中位数，但这意味着我们必须先解决一个同等规模的问题
实际情况：使用近似中位数代替——中位数的中位数

选择中位数的中位数

[见原文第 72 页图片：中位数的中位数选择示意图]

1. 对每列按降序排序

选择中位数的中位数

[见原文第 72 页图片：中位数的中位数选择示意图]

1. 对每列按降序排序
2. 选择中位数的中位数——m^*

划分

3. 重新组织各列：包含 m^* 的列在中间，中位数小于 m^* 的列在左边，中位数大于 m^* 的列在右边

[见原文第 73 页图片：ABCD 四个区域划分示意图]

演示：$n = 15$，$k = 6$

分解为子问题

S_1	8	14	15	S_2
	7	11	13	
	5	9	12	
	3	6	10	
	2	1	4	

子问题：$\{8, 7, 5, 3, 2, 6, 1, 4\}$
子问题规模 $= 8$，$k = 6$

QuickSelect 伪代码

Algorithm 11 QuickSelect$(A[n], k)$

将 A 中的元素分成大小为 5 的组，共 $m = \lceil n/5 \rceil$ 组;
对每组排序并将中位数放入 M;
$m^* \leftarrow$ QuickSelect$(M, \lceil |M|/2 \rceil)$ // 将 S 分成 A, B, C, D
对于 A 和 D 中的元素，将小于 m^* 的记入 S_1，大于 m^* 的记入 S_2;
$S_1 \leftarrow S_1 \cup C$，$S_2 \leftarrow S_2 \cup B$;
if $k = |S_1| + 1$ **then**
    输出 m^*;
else if $k \leq |S_1|$ **then**
    QuickSelect(S_1, k);
else
    QuickSelect$(S_2, k - |S_1| - 1)$;
end if

复杂度分析

每轮 QuickSelect 算法由两次 QuickSelect 递归调用组成：

1. 从中位数 M 中选择中位数作为分解的枢轴
2. 真正的子问题

算法的总体复杂度由枢轴的质量决定

最坏情况复杂度

[见原文第 78 页图片：不平衡划分示意图]

我们考虑一种极端情况：A 区和 D 区的元素去向同一边。

- $n = 5(2r+1)$，$|A| = |D| = 2r$
- 子问题规模最多为：$2r + 2r + 3r + 2 = 7r + 2$

子问题规模估计

假设 $n = 5(2r+1)$，$|A| = |D| = 2r$

$$r = \frac{n/5 - 1}{2} = \frac{n}{10} - \frac{1}{2}$$

分解后子问题规模最多为：

$$7r + 2 = 7\left(\frac{n}{10} - \frac{1}{2}\right) + 2 = \frac{7n}{10} - \frac{3}{2} < \frac{7n}{10}$$

最坏情况复杂度的递推关系

最坏情况复杂度 $W(n)$

- 第 2 行：$\Theta(n)$ // 在每 5 个元素中选中位数（常数时间），形成 M
- 第 3 行：$W(n/5)$ // 选择 M 的中位数 m^*
- 第 4 行：$\Theta(n)$ // 用 m^* 分解 S（只需比较 A 和 D）
- 第 8-9 行：$W(7n/10)$ // 对子问题的递归调用

递推关系为：

$$W(n) \leq W(n/5) + W(7n/10) + \Theta(n)$$

通过递归树求解

$$W(n) \leq W(n/5) + W(7n/10) + \Theta(n)$$

[见原文第 81 页图片：递归树示意图]

树的深度是 $\Theta(\log n) \Rightarrow$ 叶节点数量是 $\Theta(n)$；求解最小问题的代价是常数 $\Rightarrow$ 所有最小问题的代价是 $\Theta(n)$

$$W(n) \leq cn(1 + 0.9 + 0.9^2 + \ldots) + \Theta(n) = \Theta(n)$$

讨论

为什么我们必须将元素分成大小为 5 的组? 这是你的幸运数字吗?

我们能选择组大小为 3 或 7 吗?

案例研究：$t = 3$

[见原文第 83 页图片：t=3 时的划分示意图]

$n = 3(2r + 1)$，$r = (n/3 - 1)/2 = n/6 - 1/2$
子问题规模最多为：$4r + 1 = 4n/6 - 1$
最坏情况复杂度的递推关系为：

$$W(n) = W(n/3) + W(4n/6) + cn$$

通过递归树求解 $\Rightarrow$ $W(n) = \Theta(n \log n)$

关于组大小

组大小决定了 m^* 的质量和选择代价，进而影响总体复杂度。设 t 为组大小。

1. 选择 m^* 的代价与 $|M| = n/t$ 有关。t 越大，$|M|$ 越小。
2. 分解后子问题的规模与 t 有关。t 越大，$|S_i|$ 越大。

我们必须找到最佳平衡点。

[见原文第 84 页图片：平衡点示意图]

关键. 当 $|M| + |S_i| < n$ 时，递归树内部节点的总代价形成公比小于 1 的等比级数。只有在这种情况下 $W(n)$ 才是 $\Theta(n)$。

中位数的中位数

QuickSelect 的第 1-2 步构成了近似中位数（中位数的中位数）选择算法。

[BFP+73] Blum-Floyd-Pratt-Rivest-Tarjan 1973：存在一种基于比较的选择算法，其 $W(n) = O(n)$。

[见原文第 85 页图片：BFPRT 论文]

它最初以 PICK 的名称单独研究，常用于为精确选择算法（最常见的是 QuickSelect）提供好的枢轴。

更多关于中位数的中位数

理论

- BFPRT 的优化版本：$\leq 5.4305n$ 次比较
- 已知最好上界 [Dor-Zwick 1995]：$\leq 2.95n$
- 已知最好下界 [Dor-Zwick 1999]：$\geq (2+\epsilon)n$

实践

- 常数和开销（目前）太大，实用性不强

Musser [Mus97]（因其在泛型编程方面的工作而闻名 $\Rightarrow$ C++ 标准模板库），为了为 C++ STL 提供泛型算法，引入了 IntroSelect（" 内省选择" 的缩写）：原始随机版 QuickSelect 和中位数的中位数的混合体

- 乐观地从 QuickSelect 开始，只有在递归次数过多而进展不足时才切换到最坏情况线性时间选择算法

选择中位数的应用：最优管道设计

问题. 假设有 n 口油井，任务是建造一个管道系统连接 n 口油井。管道系统由一条水平主管道组成，每口油井通过一条垂直管道连接到主管道。
优化目标. 如何选择主管道的位置以最小化垂直管道的总长度?

[见原文第 87 页图片：管道设计示意图]

最优解：Y 坐标的中位数

主管道是水平的 $\Rightarrow$ 最优解与 X 坐标的分布无关

最优解：Y 坐标的中位数

主管道是水平的 $\Rightarrow$ 最优解与 X 坐标的分布无关
如果中位数唯一，则选择它；否则，选择两个中位数中的任何一个都可以（两个中位数之间的任何水平线也可以）。

[见原文第 88 页图片：最优解示意图]

最优解：Y 坐标的中位数

主管道是水平的 $\Rightarrow$ 最优解与 X 坐标的分布无关
如果中位数唯一，则选择它；否则，选择两个中位数中的任何一个都可以（两个中位数之间的任何水平线也可以）。

[见原文第 88 页图片：最优解示意图]

分析

我们首先考虑向下移动的影响：

分析

我们首先考虑向下移动的影响：
若 n 为奇数：中位数唯一（中位数上方/下方的油井数 $n' = (n-1)/2$）

- 变化量：$+(n'+1)\Delta$，最多 $\pm k\Delta$，$-(n'-k)\Delta$，$1 \le k \le n'$
- 变化量之和 $= \Delta \pm k\Delta + k\Delta > 0$

分析

我们首先考虑向下移动的影响：

若 n 为奇数：中位数唯一（中位数上方/下方的油井数 $n' = (n-1)/2$）

- 变化量：$+(n'+1)\Delta$，最多 $\pm k\Delta$，$-(n'-k)\Delta$，$1 \le k \le n'$
- 变化量之和 $= \Delta \pm k\Delta + k\Delta > 0$

若 n 为偶数：不妨设（上方 $n' = n/2$，下方 $n' = n/2$）

- 变化量：$+n'\Delta$，最多 $\pm k\Delta$，$-(n'-k)\Delta$，$1 \le k \le n'$
- 变化量之和 $= \pm k\Delta + k\Delta \ge 0$

分析

我们首先考虑向下移动的影响：

若 n 为奇数：中位数唯一（中位数上方/下方的油井数 $n' = (n-1)/2$）

- 变化量：$+(n'+1)\Delta$，最多 $\pm k\Delta$，$-(n'-k)\Delta$，$1 \le k \le n'$
- 变化量之和 $= \Delta \pm k\Delta + k\Delta > 0$

若 n 为偶数：不妨设（上方 $n' = n/2$，下方 $n' = n/2$）

- 变化量：$+n'\Delta$，最多 $\pm k\Delta$，$-(n'-k)\Delta$，$1 \le k \le n'$
- 变化量之和 $= \pm k\Delta + k\Delta \ge 0$

总之，如果主管道从中位数向下移动，垂直管道的总长度会增加。

分析

我们首先考虑向下移动的影响：
若 n 为奇数：中位数唯一（中位数上方/下方的油井数 $n' = (n-1)/2$）

- 变化量：$+(n'+1)\Delta$，最多 $\pm k\Delta$，$-(n'-k)\Delta$，$1 \le k \le n'$
- 变化量之和 $= \Delta \pm k\Delta + k\Delta > 0$

若 n 为偶数：不妨设（上方 $n' = n/2$，下方 $n' = n/2$）

- 变化量：$+n'\Delta$，最多 $\pm k\Delta$，$-(n'-k)\Delta$，$1 \le k \le n'$
- 变化量之和 $= \pm k\Delta + k\Delta \ge 0$

总之，如果主管道从中位数向下移动，垂直管道的总长度会增加。
相同的分析也适用于向上移动的情况，效果相同。

选择问题总结

选择最大值或最小值

- 朴素顺序比较：$W(n) = n - 1$

选择最大值和最小值

- 分组算法：$W(n) = 3\lfloor n/2 \rfloor$
- 分治策略：$W(n) = 3n/2 - 2$

选择次大值：锦标赛算法

$$W(n) = n + \lceil \log n \rceil - 2$$

一般选择问题：分治算法

$$W(n) = \Theta(n) \quad (\approx 44n)$$

目录

寻找最近点对

输入. 给定平面上 $n > 1$ 个点 P，找到欧氏距离最小的一对点。

[见原文第 92 页图片：最近点对示意图]

寻找最近点对

输入. 给定平面上 $n > 1$ 个点 P，找到欧氏距离最小的一对点。

[见原文第 92 页图片：最近点对示意图]

基本几何原语

- 图形学、计算机视觉、地理信息系统、分子建模、空中交通管制
- 最近邻、欧氏 MST、Voronoi 图的特例
  - 快速最近点对启发了这些问题的快速算法

尝试

一维版本. 如果 n 个点在一条线上，$O(n \log n)$ 算法（注意：输入不是按顺序给出的）

1. 按 x 坐标排序 n 个点
2. 计算相邻点之间的距离
3. 选择最短的

尝试

一维版本. 如果 n 个点在一条线上，$O(n \log n)$ 算法（注意：输入不是按顺序给出的）

1. 按 x 坐标排序 n 个点
2. 计算相邻点之间的距离
3. 选择最短的

暴力算法. 检查所有 C_n^2 对点并计算距离 $\Rightarrow$ 时间复杂度 $\Theta(n^2)$

尝试

一维版本. 如果 n 个点在一条线上，$O(n \log n)$ 算法（注意：输入不是按顺序给出的）

1. 按 x 坐标排序 n 个点
2. 计算相邻点之间的距离
3. 选择最短的

暴力算法. 检查所有 C_n^2 对点并计算距离 $\Rightarrow$ 时间复杂度 $\Theta(n^2)$

非退化假设. 没有两个点具有相同的 x 坐标或 y 坐标。

尝试

一维版本. 如果 n 个点在一条线上，$O(n \log n)$ 算法（注意：输入不是按顺序给出的）

1. 按 x 坐标排序 n 个点
2. 计算相邻点之间的距离
3. 选择最短的

暴力算法. 检查所有 C_n^2 对点并计算距离 $\Rightarrow$ 时间复杂度 $\Theta(n^2)$

非退化假设. 没有两个点具有相同的 x 坐标或 y 坐标。

$\Theta(n^2)$ 复杂度似乎是不可避免的。我们能做得更好吗?

分治策略

思路. 将 P 划分为大小大致相同的 P_L 和 P_R

- **分解**：画一条垂直线 l 使每边有 $n/2$ 个点：P_L 和 P_R
- **求解**：递归地在每边找最近点对
- **合并**：找每边各有一个点的最近点对
- 返回 3 个解中最近的

划分演示：$n = 10$

[见原文第 95 页图片：划分演示]

MinDistance 伪代码

Algorithm 12 MinDistance(P, X, Y)

输入 点集 P，坐标集 X 和 Y
输出 最近点对及距离

1: **if** $|P| < 3$ **then**
2: 直接计算;
3: **end if**
4: 已排序的 X 和 Y;
5: 画中线 l 将 P 划分为 P_L 和 P_R;
6: $\delta_L \leftarrow$ MinDistance(P_L, X_L, Y_L);
7: $\delta_R \leftarrow$ MinDistance(P_R, X_R, Y_R);
8: $\delta = \min(\delta_L, \delta_R)$; // δ_L, δ_R 是子问题的解
9: 检查距离 l 一定范围内的点;
10: **if** 距离小于 δ **then**
11: 更新 δ 为此值;
12: **end if**

识别关键点

合并步骤似乎又需要 $\Theta(n^2)$。

第 7 步需要精细的设计和分析。

如何找每边各有一个点的最近点对?

观察 1. 只需考虑距离线 l 在 δ 范围内的点。

[见原文第 98 页图片： 范围示意图]

观察 2. 在右侧的每个矩形中：点数 ≤ 1

- 每个点最多需要检查（计算后比较）对面 6 个点（因为每个单元格最多 1 个点）
- 检查一个点需要常数时间 $\Rightarrow$ 比较 $\Theta(n)$ 个点需要 $\Theta(n)$ 时间

跨中线点的处理

如何实现这个想法?

对于给定的点，如何高效地找到对应的 6 个点?

跨中线点的处理

如何实现这个想法?
对于给定的点，如何高效地找到对应的 6 个点?

按 y 坐标对 2δ 条带中的点排序。

- 这个排序列表可以在 $\Theta(n)$ 时间内从已排序的 Y 导出（想想怎么做?）

跨中线点的处理

如何实现这个想法?
对于给定的点，如何高效地找到对应的 6 个点?

按 y 坐标对 2δ 条带中的点排序。

- 这个排序列表可以在 $\Theta(n)$ 时间内从已排序的 Y 导出（想想怎么做?）

然后顺序测试:

1. 选择当前点垂直距离在 δ 范围内的邻居：最多 7 个在上方，最多 7 个在下方（$7 = 8 - 1$）
2. 如果邻居在同一侧，则跳过；否则，计算到该邻居的距离

跨中线点的处理

如何实现这个想法?
对于给定的点，如何高效地找到对应的 6 个点?

按 y 坐标对 2δ 条带中的点排序。

- 这个排序列表可以在 $\Theta(n)$ 时间内从已排序的 Y 导出（想想怎么做?）

然后顺序测试:

1. 选择当前点垂直距离在 δ 范围内的邻居：最多 7 个在上方，最多 7 个在下方（$7 = 8 - 1$）
2. 如果邻居在同一侧，则跳过；否则，计算到该邻居的距离

对于一个点，选择后计算可以在常数时间内完成，n 个点共 $\Theta(n)$

跨中线点的处理

如何实现这个想法?
对于给定的点，如何高效地找到对应的 6 个点?

按 y 坐标对 2δ 条带中的点排序。

- 这个排序列表可以在 $\Theta(n)$ 时间内从已排序的 Y 导出（想想怎么做?）

然后顺序测试:

1. 选择当前点垂直距离在 δ 范围内的邻居：最多 7 个在上方，最多 7 个在下方（$7 = 8 - 1$）
2. 如果邻居在同一侧，则跳过；否则，计算到该邻居的距离

对于一个点，选择后计算可以在常数时间内完成，n 个点共 $\Theta(n)$
为什么不分别对两个 δ 条带排序? 因为固定一侧的一个点，在对面定位"6 个邻居" 可能很复杂。

复杂度分析

步骤	操作	时间复杂度
1	最小问题	$O(1)$
2	排序 X 和 Y	$\Theta(n \log n)$
3	划分	$O(n)$
4-5	子问题	$2\,W(n/2)$
6	$\delta = \min\{\delta_L, \delta_R\}$	$O(1)$
7	跨中线处理	$\Theta(n)$

为什么排序 X 和 Y 是必要的?

- 排序 X: 将 P 划分为 P_L 和 P_R
- 排序 Y: 处理条带

$$\begin{cases} W(n) = 2\,W(n/2) + \Theta(n \log n) \\ W(n) = O(1), n \leq 3 \end{cases}$$

我们有: 递归树 $\Rightarrow$ $W(n) = \Theta(n \log^2 n)$

回顾

与暴力算法相比，分治算法的复杂度好得多。

回顾

与暴力算法相比，分治算法的复杂度好得多。

我们能进一步改进吗？特别是降低排序的复杂度。

回顾

与暴力算法相比，分治算法的复杂度好得多。

我们能进一步改进吗？特别是降低排序的复杂度。

原始方法. 划分后重新排序子问题的坐标

回顾

与暴力算法相比，分治算法的复杂度好得多。

我们能进一步改进吗？特别是降低排序的复杂度。

原始方法. 划分后重新排序子问题的坐标

改进方法

1. **预处理**. 递归前排序 X 和 Y
2. 划分时分割已排序的 X 和 Y，得到 P_L 的已排序 X_L, Y_L 和 P_R 的已排序 X_R, Y_R
   - 分割 X 很简单：按中位数分割
   - 分割 Y：根据 X 的分割结果

当原问题规模为 n 时，分割复杂度为 $\Theta(n)$

回顾

与暴力算法相比，分治算法的复杂度好得多。

我们能进一步改进吗？特别是降低排序的复杂度。

原始方法. 划分后重新排序子问题的坐标

改进方法

1. **预处理**. 递归前排序 X 和 Y
2. 划分时分割已排序的 X 和 Y，得到 P_L 的已排序 X_L, Y_L 和 P_R 的已排序 X_R, Y_R
   - 分割 X 很简单：按中位数分割
   - 分割 Y：根据 X 的分割结果

当原问题规模为 n 时，分割复杂度为 $\Theta(n)$

每次从头排序 $\Rightarrow$ 排序一次然后分割

排序和分割的细节

数据结构. 两个列表 $X[n]$, $Y[n]$，每个元素是标签 i，按 x, y 坐标升序排序一次

分割 X：简单，但需要额外技巧来方便分割 Y。设 n 为当前问题规模

- 生成大小为 n 的指示映射 H，$H[i] = 0$ 表示点 i 在左边，$H[i] = 1$ 表示点在右边
- 时间复杂度 $\Theta(n)$

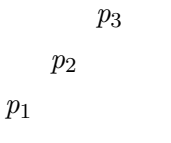
分割 Y：顺序扫描 $Y[n]$

- 如果 $H[Y[i]] = 0$，将 $Y[i]$ 归类到左边，否则归类到右边，得到已排序的 Y_L 和 Y_R
- 时间复杂度 $\Theta(n)$

递归中分割的演示

输入

P	1	2	3	4
x	0.5	2	-2	1
y	2	3	4	-1



预处理：排序

X	3	1	4	2
Y	4	1	2	3

分割

X_L	3	1
Y_L	1	3

分割

X_R	4	2
Y_R	4	2

改进的分治算法

$T(n)$ 是总体复杂度，$\Theta(n\log n)$ 是全局预处理的复杂度，$T'(n)$ 是主递归算法的复杂度，

$$\begin{cases} T(n) = T'(n) + \Theta(n\log n) \\ T'(n) = 2T'(n/2) + \Theta(n) \\ T'(n) = O(1) \quad n \le 3 \end{cases}$$

主定理（情况 2）$\Rightarrow T'(n) = \Theta(n\log n)$

综合以上，$T(n) = \Theta(n\log n)$

下界. 在二次决策树模型中（计算欧氏距离后比较），任何最近点对算法（即使在一维中）都需要 $\Theta(n\log n)$ 次二次测试。

参考文献


Manuel Blum, Robert W. Floyd, Vaughan R. Pratt, Ronald L. Rivest, and Robert Endre Tarjan.
Time bounds for selection.
J. Comput. Syst. Sci., 7(4):448–461, 1973.


C. A. R. Hoare.
Proof of a program: FIND.
Commun. ACM, 14(1):39–45, 1971.


David R. Musser.
Introspective sorting and selection algorithms.
Softw. Pract. Exp., 27(8):983–993, 1997.