算法设计与分析

分治法（I）

目录

1. 分治法简介
2. 归并排序
3. 快速排序
4. 芯片测试
5. 选择问题
   - 选择最大和最小
   - 选择第二大
   - 一般选择问题
6. 最近点对

分治法的起源：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拆分成各个较小的部分来获取和维持权力，这些部分单独的力量比实施该策略的一方更弱。

- 格言"divide et impera" 被认为出自马其顿的腓力二世。罗马统治者尤利乌斯·凯撒和法国皇帝拿破仑都曾使用此策略。

分治法的起源：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》

[秦王扫六合时，虎视何雄哉]

分治法作为算法设计范式

分治策略通过以下方式解决问题：

分治法作为算法设计范式

分治策略通过以下方式解决问题：

1. **分解**：将原问题分解为若干个规模较小、可以独立求解的子问题

分治法作为算法设计范式

分治策略通过以下方式解决问题：

1. **分解**：将原问题分解为若干个规模较小、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

分治法作为算法设计范式

分治策略通过以下方式解决问题：

1. **分解**：将原问题分解为若干个规模较小、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

为什么使用分治法

并非总是如此，但通常比暴力算法性能更好

为什么使用分治法

并非总是如此，但通常比暴力算法性能更好

最常见的用法（以排序为例）

- 在线性时间内将规模为 n 的问题分成两个规模为 $n/2$ 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

为什么使用分治法

并非总是如此，但通常比暴力算法性能更好

最常见的用法（以排序为例）

- 在线性时间内将规模为 n 的问题分成两个规模为 $n/2$ 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

暴力法：$\Theta(n^2)$ vs. 分治法：$\Theta(n \log n)$

为什么使用分治法

并非总是如此，但通常比暴力算法性能更好

最常见的用法（以排序为例）

- 在线性时间内将规模为 n 的问题分成两个规模为 $n/2$ 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

暴力法：$\Theta(n^2)$ vs. 分治法：$\Theta(n \log n)$

- 特别适用于并行计算环境（将更加高效）

思考

仅仅将问题切成碎片并不能使其更容易解决。

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

为什么分治法实际上能提高效率?

通用分治算法

1: **if** $|P| \leq s^*$ **then**
2: Solve(P) {直接求解}
3: **else**
4: 将 P 分成 $P_1, P_2, \ldots, P_k$ {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**

分治法的复杂度

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
- $f(n)$ 是分解子问题和将子问题答案合并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（直接迭代或化简后迭代）
2. 递归树

例：汉诺塔：$T(n) = 2T(n-1) + 1$

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

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

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说明分治法的主要思想。

汉诺塔

Require: $A(n), B(0), C(0)$
Ensure: $A(0), B(0), C(n)$
1: **if** $n = 1$ **then**
2: move(A, C) {将一个盘子从 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 个盘子的复杂度：所需的最小移动次数

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

- 此问题没有最坏情况、最好情况、平均情况之分，因为输入仅取决于输入规模

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 $x \leq A[m]$，则 $A[l,r] := A[l,m]$；否则 $A[l,r] := A[m+1,r]$
2. 重复搜索 T 直到规模变为 1，即 $l = r$
   - 此时，直接比较 x 和 $A[l]$，相等返回 l，否则返回"0"

二分查找的最坏情况复杂度

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

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

分治范式回顾

我们总结分治范式的特点：

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

目录

1. 分治法简介
2. 归并排序
3. 快速排序
4. 芯片测试
5. 选择问题
   - 选择最大和最小
   - 选择第二大
   - 一般选择问题
6. 最近点对

归并排序

Require: 未排序的 $A[n]$
Ensure: 升序排列的 $A[n]$
1: $l \leftarrow 1,\ r \leftarrow n$
2: **if** $l < r$ **then**
3: $\quad m \leftarrow \lfloor (l+r)/2 \rfloor$ {对半划分}
4: $\quad$ MergeSort(A, l, m) {子问题 1}
5: $\quad$ MergeSort$(A, m+1, r)$ {子问题 2}
6: $\quad$ Merge$(A[l, m], A[m+1, r])$ {合并已排序子数组}
7: **end if**

如何递归实现 Merge?

递归归并算法

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** $A[1] \leq B[1]$ **then**
8: **return** $A[1] \circ \text{Merge}(A[2, k], B[1, l])$
9: **else**
10: **return** $B[1] \circ \text{Merge}(A[1, k], B[2, l])$
11: **end if**

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

归并排序的复杂度

1. 将原问题划分为 2 个规模为 $n/2$ 的子问题
2. 继续划分直到子问题规模为 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

由于数组数据结构的特性，子问题的划分已经完成，所有真正需要做的工作是归并。

归并排序回顾

由于数组数据结构的特性，子问题的划分已经完成，所有真正需要做的工作是归并。

这种观点表明归并排序可以从单元素数组到原始数组以自底向上的方式迭代进行。

目录

1. 分治法简介
2. 归并排序
3. 快速排序
4. 芯片测试
5. 选择问题
   - 选择最大和最小
   - 选择第二大
   - 一般选择问题
6. 最近点对

基本思想

1. 选择第一个元素 x 作为基准，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 位于正确位置
2. 递归排序 A_L 和 A_R，直到子数组规模为 1

快速排序的伪代码

Require: $A[l \dots r]$
Ensure: 升序排列的 A
1: **if** $l = r$ **then**
2: **return** {达到最小情况}
3: **end if**
4: **if** $l < r$ **then**
5: $k \leftarrow \text{Partition}(A, l, r)$
6: $A[l] \leftrightarrow A[k]$
7: $\text{QuickSort}(A, l, k-1)$
8: $\text{QuickSort}(A, k+1, r)$
9: **end if**

与归并排序不同，在快速排序中没有合并步骤，所有真正需要做的工作是划分。

划分的伪代码

1: $x \leftarrow A[l]$ {设第一个元素为基准}
2: $i \leftarrow l, j \leftarrow r+1$ {初始化左/右指针}
3: **while** true **do**
4: **repeat**
5: $j \leftarrow j-1$
6: **until** $A[j] \leq x$ {小于 x}
7: **repeat**
8: $i \leftarrow i+1$
9: **until** $A[i] > x$ {大于 x}
10: **if** $i < j$ **then**
11: $A[i] \leftrightarrow A[j]$
12: **else**
13: **return** j {交叉发生，找到位置}
14: **end if**
15: **end while**

划分演示

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:

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

常数比例划分的复杂度

常数比例划分：|子问题|/|原问题|是固定常数，比如 $1:9$。

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

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

平均情况复杂度

假设第一个元素最终以相同概率 $1/n$ 出现在位置 $1, 2, \ldots, n$。
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

$$T(n) = \frac{2}{n}\sum_{k=1}^{n-1} T(k) + n - 1, \quad T(1) = 0$$

通过相减化简递推关系 $\Rightarrow$

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

目录

1. 分治法简介
2. 归并排序
3. 快速排序
4. 芯片测试
5. 选择问题
   - 选择最大和最小
   - 选择第二大
   - 一般选择问题
6. 最近点对

芯片测试

芯片工厂只允许基本测试方法。

基本测试方法：将两个芯片 A 和 B 放在测试台上，开始互测

- 测试报告是" 好" 或" 坏"

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 都是好的或都是坏的
B 是好的	A 是坏的	至少一个是坏的
B 是坏的	A 是好的	至少一个是坏的
B 是坏的	A 是坏的	至少一个是坏的

芯片测试问题

输入：n 个芯片，$\#($ 好芯片 $) - \#($ 坏芯片 $) \geq 1$
问题：设计一种测试方法，从 n 个芯片中选出一个好芯片
要求：互测次数最少

芯片测试问题

输入：n 个芯片，$\#($ 好芯片 $) - \#($ 坏芯片 $) \geq 1$
问题：设计一种测试方法，从 n 个芯片中选出一个好芯片
要求：互测次数最少

出发点：给定芯片 A，如何检查 A 是好是坏?
方法：用其他 $n-1$ 个芯片测试 A

- 思路：利用 n 的奇偶性

情况 1：n 是奇数

例：$n = 7$，$\#($ 好芯片 $) \geq 4$

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

推广到 n 是奇数，$\#($ 好芯片 $) \geq (n+1)/2$

关键观察：测试结果是互斥的，因此测试条件实际上是充要条件。

判定标准：在 $n-1$ 个报告中

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

情况 2：n 是偶数

例：$n = 8$，$\#($ 好芯片 $) \geq 5$

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

推广到 n 是偶数，$\#($ 好芯片 $) \geq n/2 + 1$

判定标准：在 $n - 1$ 个报告中

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

暴力算法

测试方法：随机选一个芯片，应用上述测试。如果是好的，测试结束。否则，丢弃它，从剩余芯片中随机选另一个，直到得到一个好芯片。

- 正确性：$\#($ 好芯片 $)$ 总是超过一半

时间复杂度

- 第 1 轮：随机选到坏的，最多 $n-1$ 次测试
- 第 2 轮：随机选到坏的，最多 $n-2$ 次测试
- $\cdots$
- 第 i 轮：随机选到坏的，最多 $n-i$ 次测试

最坏情况下的总体复杂度是 $\Theta(n^2)$

分治法

假设 n 是偶数，将 n 个芯片分成两组并开始互测；剩余芯片形成子问题，开始下一轮测试

分治法

假设 n 是偶数，将 n 个芯片分成两组并开始互测；剩余芯片形成子问题，开始下一轮测试

测试-淘汰规则

- " 好，好" $\rightarrow$ 随机选一个进入下一轮
- 其他情况 $\rightarrow$ 全部丢弃

分治法

假设 n 是偶数，将 n 个芯片分成两组并开始互测；剩余芯片形成子问题，开始下一轮测试

测试-淘汰规则

- " 好，好" $\rightarrow$ 随机选一个进入下一轮
- 其他情况 $\rightarrow$ 全部丢弃

递归终止条件：$n \leq 3$

- 3 个芯片：一次测试足够
  1. " 好，好"：随机选一个输出
  2. " 好，坏"：输出剩余那个
  3. " 坏，坏"：输出剩余那个
- 1 或 2 个芯片：都是好的，不需要更多测试

分治算法的正确性

断言：当 n 是偶数时，经过一轮测试后，在剩余芯片中，$\#($ 好芯片 $) - \#($ 坏芯片 $) \geq 1$

证明：考虑以下三种情况：

1. 两个都是好的（i 组）$\rightarrow$ 保留随机一个
2. 一好一坏（j 组）$\rightarrow$ 全部丢弃
3. 两个都是坏的（k 组）$\rightarrow$ 保留随机一个或全部丢弃

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

$$\left.\begin{array}{r} 2i + 2j + 2k = n \quad \text{（测试前芯片数）} \\ 2i + j > 2k + j \quad \text{（好芯片数 } > \text{ 坏芯片数）} \end{array}\right\} \Rightarrow i > k$$

n 是奇数时的调整

当 n 是奇数时，会有一个芯片没有配对。

调整：当 n 是奇数时，对未配对芯片增加一轮直接测试

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

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 是奇数时的额外调整）：$\Theta(n)$

递推关系

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

分治芯片测试算法小结

- 调整 $\rightarrow$ 保证子问题与原问题类型相同
- 分支因子 $a = 1$ 且分解-合并代价 $f(n) = \Theta(n) \rightarrow$ 确保相对暴力算法有显著效率提升

目录

1. 分治法简介
2. 归并排序
3. 快速排序
4. 芯片测试
5. 选择问题
   - 选择最大和最小
   - 选择第二大
   - 一般选择问题
6. 最近点对

一般选择问题

选择：给定来自全序集 S 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：$k = \lfloor (n+1)/2 \rfloor$

已知结果

- 选最小或最大：$O(n)$ 次比较
- 一般选择的朴素算法：通过排序 $O(n \log n)$ 次比较，用二叉堆 $O(n \log k)$ 次比较

问：能否用 $O(n)$ 次比较完成一般选择?
答：可以！选择比排序更容易。

选择最大值

算法：顺序比较
输出：$\max = 17$，$i = 4$
最坏情况复杂度：$W(n) = n - 1$

选择最大和最小

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最小值，使用相同算法但每次比较后保留较小元素

选择最大和最小

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最小值，使用相同算法但每次比较后保留较小元素

最坏情况时间复杂度

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

分组算法

思路：将列表分成高列表和低列表

分组法选择最大最小的伪代码

Require: 未排序的 $A[n]$
Ensure: $\max, \min$
1: 将 n 个元素分成 $\lfloor n/2 \rfloor$ 组
2: 在每组内比较两个元素，得到 $\lfloor n/2 \rfloor$ 个较小的和 $\lfloor n/2 \rfloor$ 个较大的
3: 在 $\lfloor n/2 \rfloor$ 个较大元素和额外元素中选最大
4: 在 $\lfloor n/2 \rfloor$ 个较小元素和额外元素中选最小

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

- 组内比较：$\lfloor n/2 \rfloor$
- n 是偶数：选最大 $n/2-1$，选最小 $n/2-1$
- n 是奇数：选最大 $(n-1)/2$，选最小 $(n-1)/2$

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。

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。

能否用分治法设计 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。

能否用分治法设计 SelectMaxMin?

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

最坏情况复杂度

假设 $n = 2^k$，$W(n)$ 的递推关系如下：

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

用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，求和得：

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

小结

选择最大值：顺序比较，最多需要 $n-1$ 次比较
选择最大和最小（最坏情况）：

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

选择第二大

输入：$A[n]$
输出：第二大的 $\max'$

选择第二大

输入：$A[n]$
输出：第二大的 $\max'$
朴素算法：顺序比较

1. 通过顺序比较从 $A[n]$ 中选出 $\max$
2. 从 $A[n] \setminus \max$ 中选出 $\max'$，即为第二大

时间复杂度：$W(n) = (n-1) + (n-2) = 2n - 3$

优化方法

观察：成为第二大的充要条件：只被最大的打败
要确定第二大元素，必须先知道最大元素。

优化方法

观察：成为第二大的充要条件：只被最大的打败
要确定第二大元素，必须先知道最大元素。
思路：以空间换时间

- 在选择最大元素的过程中，将被最大元素打败的元素记录在集合 L 中
- 在 L 中的元素中选择最大的

锦标赛算法选择第二大

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（只）将被打败的元素记录在其列表中
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 $\max'$

- 名称来自单淘汰锦标赛：选手两两比赛，胜者晋级。层次结构持续直到决赛决定最终胜者。锦标赛决定了最佳选手，但决赛中被打败的选手可能不是第二好的——他可能不如胜者之前打败的其他选手。

SelectSecond 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 $\lceil n/2^i \rceil$ 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，晋级到下一层的元素数为

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

- 归纳步骤：$P(i) \Rightarrow P(i+1)$。假设第 i 轮后元素数最多为 $\lceil n/2^i \rceil$，则第 $i+1$ 轮后元素数为

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

复杂度分析（2/3）

命题 2：max 与 $\lceil \log n \rceil$ 个元素比较

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

- 若 $n = 2^d$（$d \in \mathbb{Z}$），则：

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

- 否则 $2^d < n < 2^{d+1}$（$d \in \mathbb{Z}$），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：元素数为 $\lceil \log n \rceil$，根据命题 2 正好是胜者列表的大小

- 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：

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

选择第二大小结

选择第二大

- 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

一般选择问题

问题：选择第 k 小
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，$|S| = n$，选中位数，$k = \lceil n/2 \rceil$

朴素算法

算法 1

- 调用算法 SelectMin k 次
- 时间复杂度：$O(kn)$

朴素算法

算法 1

- 调用算法 SelectMin k 次
- 时间复杂度：$O(kn)$

算法 2

- 排序后选第 k 小的数
- 时间复杂度：$O(n \log n)$

朴素算法

算法 1

- 调用算法 SelectMin k 次
- 时间复杂度：$O(kn)$

算法 2

- 排序后选第 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，因为排序做的远超我们真正需要的——我们不关心其余元素的相对顺序

QuickSelect

[Hoa71] 算法 65：Find

QuickSelect 使用与 QuickSort 相同的整体方法——选择一个元素 m^* 作为基准来划分 S，使得 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 中选第 $k - |S_1| - 1$ 小

QuickSelect

[Hoa71] 算法 65：Find

QuickSelect 使用与 QuickSort 相同的整体方法——选择一个元素 m^* 作为基准来划分 S，使得 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 中选第 $k - |S_1| - 1$ 小

与 QuickSort 递归进入两边不同，QuickSelect 只递归进入一边——包含目标元素的那边

- 这将平均情况复杂度从 $O(n \log n)$ 降到 $O(n)$
- 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?

像 QuickSort 一样，QuickSelect 的复杂度由基准的质量决定

- 总是选第一个元素作为基准在实践中高效且有好的平均情况性能，但最坏情况性能差

如何改进最坏情况复杂度?

像 QuickSort 一样，QuickSelect 的复杂度由基准的质量决定

- 总是选第一个元素作为基准在实践中高效且有好的平均情况性能，但最坏情况性能差

如何选择好的基准 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?

像 QuickSort 一样，QuickSelect 的复杂度由基准的质量决定

- 总是选第一个元素作为基准在实践中高效且有好的平均情况性能，但最坏情况性能差

如何选择好的基准 m^* 以获得好的平均情况和最坏情况性能?

- 随机选择不能改进最坏情况复杂度（恰好选到坏的随机性）

如何改进最坏情况复杂度?

像 QuickSort 一样，QuickSelect 的复杂度由基准的质量决定

- 总是选第一个元素作为基准在实践中高效且有好的平均情况性能，但最坏情况性能差

如何选择好的基准 m^* 以获得好的平均情况和最坏情况性能?

- 随机选择不能改进最坏情况复杂度（恰好选到坏的随机性）

理想情况：选择精确的中位数，但这意味着我们必须先解决同样规模的问题
实际情况：使用准中位数代替——中位数的中位数

选择中位数的中位数

1. 将每列按降序排序
2. 选择中位数的中位数——m^*
3. 重新组织列：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
   - A 区域：需要与 m^* 比较
   - B 区域：大于 m^*
   - C 区域：小于 m^*
   - D 区域：需要与 m^* 比较

QuickSelect 的伪代码

1: 将 A 中元素分成大小为 5 的组，共 $m = \lceil n/5 \rceil$ 组
2: 对每组排序，将中位数放入 M
3: $m^* \leftarrow \text{QuickSelect}(M, \lceil |M|/2 \rceil)$ {将 S 分成 A, B, C, D}
4: 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
5: $S_1 \leftarrow S_1 \cup C$, $S_2 \leftarrow S_2 \cup B$
6: **if** $k = |S_1| + 1$ **then**
7: 　输出 m^*
8: **else if** $k \leq |S_1|$ **then**
9: 　$\text{QuickSelect}(S_1, k)$
10: **else**
11: 　$\text{QuickSelect}(S_2, k - |S_1| - 1)$
12: **end if**

最坏情况复杂度

假设 $n = 5(2r+1)$，$|A| = |D| = 2r$

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

划分后子问题的规模最多为：

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

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 M 的中位数 m^*
- 第 4 行：$\Theta(n)$　//用 m^* 划分 S（只需比较 A 和 D）
- 第 8-9 行：$W(7n/10)$　//递归调用子问题

递推关系：

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

通过递归树求解

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

树的深度是 $\Theta(\log n) \Rightarrow$ 叶节点数是 $\Theta(n)$；解决最小问题的代价是常数 $\Rightarrow$ 所有最小问题的代价是 $\Theta(n)$

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

讨论

为什么必须将元素分成大小为 **5** 的组? 这是你的幸运数字吗?
能否选组大小为 **3** 或 **7**?

案例研究：$t = 3$

$n = 3(2r + 1)$，$r = (n/3 - 1)/2 = n/6 - 1/2$
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：

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

$1/3 + 4/6 = 1 \Rightarrow$ 不满足主定理条件

选择问题小结

选择最大或最小

- 朴素顺序比较：$W(n) = n - 1$

选择最大和最小

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

选择第二大：锦标赛算法

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

一般选择问题：分治算法

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

目录

1. 分治法简介
2. 归并排序
3. 快速排序
4. 芯片测试
5. 选择问题
   - 选择最大和最小
   - 选择第二大
   - 一般选择问题
6. 最近点对

寻找最近点对

输入：给定平面上 $n > 1$ 个点 P，找出欧氏距离最小的一对点。

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、欧氏最小生成树、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：输入未按顺序给出）

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：输入未按顺序给出）

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 对的距离计算 $\rightarrow$ 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：输入未按顺序给出）

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 对的距离计算 $\rightarrow$ 时间复杂度 $\Theta(n^2)$

非退化假设：没有两点有相同的 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：输入未按顺序给出）

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 对的距离计算 $\rightarrow$ 时间复杂度 $\Theta(n^2)$
非退化假设：没有两点有相同的 x 坐标或 y 坐标

$\Theta(n^2)$ 复杂度似乎不可避免。能做得更好吗?

分治法

思路：将 P 划分为大小大致相同的 P_L 和 P_R

- **分解**：画垂直线 l 使每边各有 $n/2$ 个点：P_L 和 P_R
- **求解**：递归找每边的最近点对
- **合并**：找一个点在每边的最近点对
- 返回 3 个解中最近的

MinDistance 的伪代码

Require: 点集 P，坐标集 X 和 Y
Ensure: 最近点对和距离
1: **if** $|P| < 3$ **then**
2: 　直接计算
3: **end if**
4: 排序 X 和 Y
5: 画中线 l 将 P 划分为 P_L 和 P_R
6: $\delta_L \leftarrow \text{MinDistance}(P_L, X_L, Y_L)$
7: $\delta_R \leftarrow \text{MinDistance}(P_R, X_R, Y_R)$
8: $\delta = \min(\delta_L, \delta_R)$ $\{\delta_L, \delta_R$ 是子问题的解$\}$
9: 检查距离 l 一定距离内的节点
10: **if** 距离小于 δ **then**
11: 　更新 δ 为此值
12: **end if**

识别关键

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。

如何找一个点在每边的最近点对?

观察 1：只需考虑距线 l 在 δ 范围内的点
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

如何实现这个思路?

对于给定的点，如何高效找到对应的 6 个点?

跨中线点的处理

如何实现这个思路?

对于给定的点，如何高效找到对应的 6 个点?

按 y 坐标对 2δ 带中的点排序

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

跨中线点的处理

如何实现这个思路?

对于给定的点，如何高效找到对应的 6 个点?

按 y 坐标对 2δ 带中的点排序

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

然后顺序测试：

1. 选择当前点在 δ 垂直距离内的邻居：最多上面 7 个，下面 7 个（$7 = 8 - 1$）
2. 如果邻居在同一侧，跳过；否则，计算到该邻居的距离

对于一个点，选择后计算可在常数时间完成，n 个点共 $\Theta(n)$

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	$\Theta(n)$

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

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。

能否进一步改进？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。

能否进一步改进？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。

能否进一步改进？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. 划分时拆分排序的 X 和 Y，得到 P_L 的排序 X_L、Y_L 和 P_R 的排序 X_R、Y_R

当原问题规模为 n 时，拆分复杂度为 $\Theta(n)$

每次从头排序 $\rightarrow$ 排序一次然后拆分

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

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

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（计算欧氏距离后比较）中，任何最近点对算法（即使在一维）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.