# OI Wiki (Beta)

OI Wiki 项目组

2023 年 9 月 14 日

# 目录

# 第 1 章

# 算法基础

## 1.1 算法基础简介

本章主要介绍一些基础算法，为之后的进阶内容做铺垫。

一方面，这些内容可以让初学者对 OI 的一些思想有初步的认识；另一方面，本章介绍的大部分算法还会在以后的进阶内容中得到运用。

## 1.2 复杂度

> **Authors:** linehk, persdre

时间复杂度和空间复杂度是衡量一个算法效率的重要标准。

### 基本操作数

同一个算法在不同的计算机上运行的速度会有一定的差别，并且实际运行速度难以在理论上进行计算，实际去测量又比较麻烦，所以我们通常考虑的不是算法运行的实际用时，而是算法运行所需要进行的基本操作的数量。

在普通的计算机上，加减乘除、访问变量（基本数据类型的变量，下同）、给变量赋值等都可以看作基本操作。

对基本操作的计数或是估测可以作为评判算法用时的指标。

### 时间复杂度

#### 定义

衡量一个算法的快慢，一定要考虑数据规模的大小。所谓数据规模，一般指输入的数字个数、输入中给出的图的点数与边数等等。一般来说，数据规模越大，算法的用时就越长。而在算法竞赛中，我们衡量一个算法的效率时，最重要的不是看它在某个数据规模下的用时，而是看它的用时随数据规模而增长的趋势，即**时间复杂度**。

#### 引入

考虑用时随数据规模变化的趋势的主要原因有以下几点：

1. 现代计算机每秒可以处理数亿乃至更多次基本运算，因此我们处理的数据规模通常很大。如果算法 A 在规模为 $n$ 的数据上用时为 $100n$ 而算法 B 在规模为 $n$ 的数据上用时为 $n^2$，在数据规模小于 100 时算法 B 用时更短，

但在一秒钟内算法 A 可以处理数百万规模的数据，而算法 B 只能处理数万规模的数据。在允许算法执行时间更久时，时间复杂度对可处理数据规模的影响就会更加明显，远大于同一数据规模下用时的影响。

2. 我们采用基本操作数来表示算法的用时，而不同的基本操作实际用时是不同的，例如加减法的用时远小于除法的用时。计算时间复杂度而忽略不同基本操作之间的区别以及一次基本操作与十次基本操作之间的区别，可以消除基本操作间用时不同的影响。

当然，算法的运行用时并非完全由输入规模决定，而是也与输入的内容相关。所以，时间复杂度又分为几种，例如：

1. 最坏时间复杂度，即每个输入规模下用时最长的输入对应的时间复杂度。在算法竞赛中，由于输入可以在给定的数据范围内任意给定，我们为保证算法能够通过某个数据范围内的任何数据，一般考虑最坏时间复杂度。
2. 平均（期望）时间复杂度，即每个输入规模下所有可能输入对应用时的平均值的复杂度（随机输入下期望用时的复杂度）。

所谓「用时随数据规模而增长的趋势」是一个模糊的概念，我们需要借助下文所介绍的**渐进符号**来形式化地表示时间复杂度。

# 渐进符号的定义

渐进符号是函数的阶的规范描述。简单来说，渐进符号忽略了一个函数中增长较慢的部分以及各项的系数（在时间复杂度相关分析中，系数一般被称作「常数」），而保留了可以用来表明该函数增长趋势的重要部分。

一个简单的记忆方法是，含等于（非严格）用大写，不含等于（严格）用小写，相等是 $\Theta$，小于是 $O$，大于是 $\Omega$。大 $O$ 和小 $o$ 原本是希腊字母 Omicron，由于字形相同，也可以理解为拉丁字母的大 $O$ 和小 $o$。

在英文中，词根「-micro-」和「-mega-」常用于表示 10 的负六次方（百万分之一）和六次方（百万），也表示「小」和「大」。小和大也是希腊字母 Omicron 和 Omega 常表示的含义。

## 大 $\Theta$ 符号

对于函数 $f(n)$ 和 $g(n)$，$f(n)=\Theta(g(n))$，当且仅当 $\exists c_1,c_2,n_0>0$，使得 $\forall n\geq n_0, 0\leq c_1\cdot g(n)\leq f(n)\leq c_2\cdot g(n)$。

也就是说，如果函数 $f(n)=\Theta(g(n))$，那么我们能找到两个正数 $c_1,c_2$ 使得 $f(n)$ 被 $c_1\cdot g(n)$ 和 $c_2\cdot g(n)$ 夹在中间。

例如，$3n^2+5n-3=\Theta(n^2)$, 这里的 $c_1,c_2,n_0$ 可以分别是 $2,4,100$。$n\sqrt{n}+n\log^5 n+m\log m+nm=\Theta(n\sqrt{n}+m\log m+nm)$，这里的 $c_1,c_2,n_0$ 可以分别是 $1,2,100$。

## 大 O 符号

$\Theta$ 符号同时给了我们一个函数的上下界，如果只知道一个函数的渐进上界而不知道其渐进下界，可以使用 $O$ 符号。$f(n)=O(g(n))$，当且仅当 $\exists c,n_0$，使得 $\forall n\geq n_0, 0\leq f(n)\leq c\cdot g(n)$。

研究时间复杂度时通常会使用 $O$ 符号，因为我们关注的通常是程序用时的上界，而不关心其用时的下界。

需要注意的是，这里的「上界」和「下界」是对于函数的变化趋势而言的，而不是对算法而言的。算法用时的上界对应的是「最坏时间复杂度」而非大 $O$ 记号。所以，使用 $\Theta$ 记号表示最坏时间复杂度是完全可行的，甚至可以说 $\Theta$ 比 $O$ 更加精确，而使用 $O$ 记号的主要原因，一是我们有时只能证明时间复杂度的上界而无法证明其下界（这种情况一般出现在较为复杂的算法以及复杂度分析），二是 $O$ 在电脑上输入更方便一些。

## 大 $\Omega$ 符号

同样的，我们使用 $\Omega$ 符号来描述一个函数的渐进下界。$f(n)=\Omega(g(n))$，当且仅当 $\exists c,n_0$，使得 $\forall n\geq n_0, 0\leq c\cdot g(n)\leq f(n)$。

## 小 o 符号

如果说 $O$ 符号相当于小于等于号，那么 $o$ 符号就相当于小于号。

小 $o$ 符号大量应用于数学分析中，函数在某点处的泰勒展开式拥有皮亚诺余项，使用小 $o$ 符号表示严格小于，从而进行等价无穷小的渐进分析。

$f(n) = o(g(n))$，当且仅当对于任意给定的正数 $c$，$\exists n_0$，使得 $\forall n \geq n_0, 0 \leq f(n) < c \cdot g(n)$。

## 小 $\omega$ 符号

如果说 $\Omega$ 符号相当于大于等于号，那么 $\omega$ 符号就相当于大于号。

$f(n) = \omega(g(n))$，当且仅当对于任意给定的正数 $c$，$\exists n_0$，使得 $\forall n \geq n_0, 0 \leq c \cdot g(n) < f(n)$。

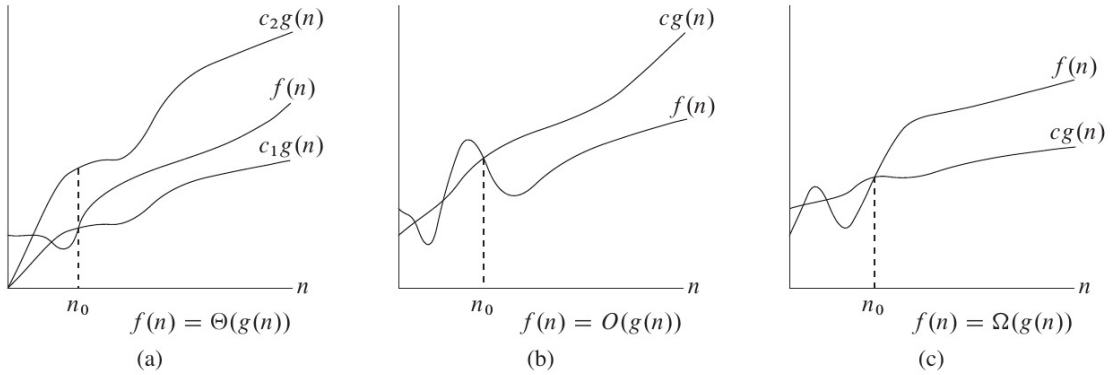


图 1.1

## 常见性质

- $f(n) = \Theta(g(n)) \iff f(n) = O(g(n)) \wedge f(n) = \Omega(g(n))$
- $f_1(n) + f_2(n) = O(\max(f_1(n), f_2(n)))$
- $f_1(n) \times f_2(n) = O(f_1(n) \times f_2(n))$
- $\forall a \neq 1, \log_a n = O(\log_2 n)$。由换底公式可以得知，任何对数函数无论底数为何，都具有相同的增长率，因此渐进时间复杂度中对数的底数一般省略不写。

# 简单的时间复杂度计算的例子

## for 循环

=== "C++"

```cpp
int n, m;
std::cin >> n >> m;
for (int i = 0; i < n; ++i) {
  for (int j = 0; j < n; ++j) {
    for (int k = 0; k < m; ++k) {
      std::cout << "hello world\n";
    }
  }
}
```

```
```

=== "Python"

```python
n = int(input())
m = int(input())
for i in range(0, n):
    for j in range(0, n):
        for k in range(0, m):
            print("hello world")
```

如果以输入的数值 $n$ 和 $m$ 的大小作为数据规模，则上面这段代码的时间复杂度为 $\Theta(n^2m)$。

### DFS

在对一张 $n$ 个点 $m$ 条边的图进行 DFS 时，由于每个节点和每条边都只会被访问常数次，复杂度为 $\Theta(n+m)$。

## 哪些量是常量？

当我们要进行若干次操作时，如何判断这若干次操作是否影响时间复杂度呢？例如：

=== "C++"

```cpp
const int N = 100000;
for (int i = 0; i < N; ++i) {
  std::cout << "hello world\n";
}
```

=== "Python"

```python
N = 100000
for i in range(0, N):
    print("hello world")
```

如果 $N$ 的大小不被看作输入规模，那么这段代码的时间复杂度就是 $O(1)$。

进行时间复杂度计算时，哪些变量被视作输入规模是很重要的，而所有和输入规模无关的量都被视作常量，计算复杂度时可当作 1 来处理。

需要注意的是，在进行时间复杂度相关的理论性讨论时，「算法能够解决任何规模的问题」是一个基本假设（当然，在实际中，由于时间和存储空间有限，无法解决规模过大的问题）。因此，能在常量时间内解决数据规模有限的问题（例如，对于数据范围内的每个可能输入预先计算出答案）并不能使一个算法的时间复杂度变为 $O(1)$。

## 主定理 (Master Theorem)

我们可以使用 Master Theorem 来快速求得关于递归算法的复杂度。Master Theorem 递推关系式如下

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \qquad \forall n > b$$

那么

$$T(n) = \begin{cases} \Theta(n^{\log_b a}) & f(n) = O(n^{\log_b a - \epsilon}), \epsilon > 0 \\ \Theta(f(n)) & f(n) = \Omega(n^{\log_b a + \epsilon}), \epsilon \geq 0 \\ \Theta(n^{\log_b a} \log^{k+1} n) & f(n) = \Theta(n^{\log_b a} \log^k n), k \geq 0 \end{cases}$$

需要注意的是，这里的第二种情况还需要满足 regularity condition, 即 $af(n/b) \leq cf(n)$，for some constant $c < 1$ and sufficiently large $n$。

证明思路是是将规模为 $n$ 的问题，分解为 $a$ 个规模为 $(\frac{n}{b})$ 的问题，然后依次合并，直到合并到最高层。每一次合并子问题，都需要花费 $f(n)$ 的时间。

**证明**

依据上文提到的证明思路，具体证明过程如下

对于第 0 层（最高层），合并子问题需要花费 $f(n)$ 的时间

对于第 1 层（第一次划分出来的子问题），共有 $a$ 个子问题，每个子问题合并需要花费 $f\left(\frac{n}{b}\right)$ 的时间，所以合并总共要花费 $af\left(\frac{n}{b}\right)$ 的时间。

层层递推，我们可以写出类推树如下：

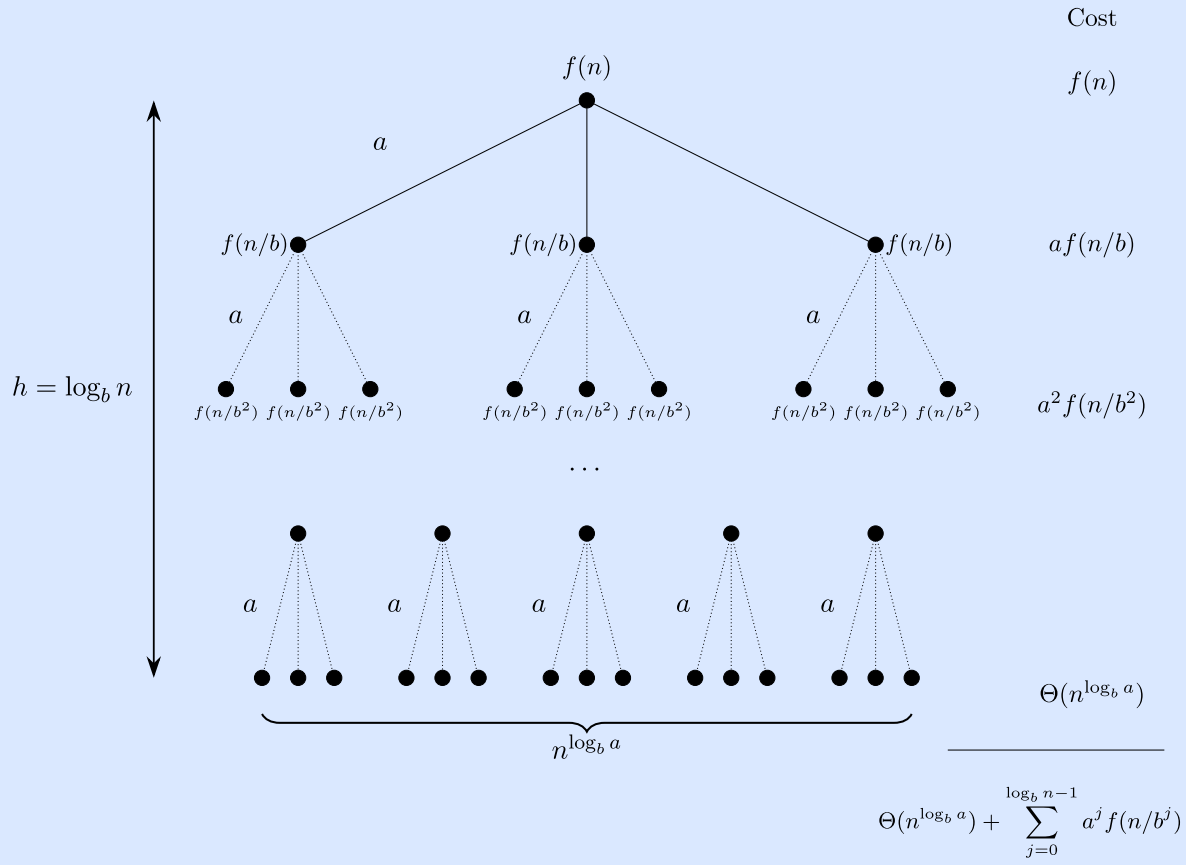


图 1.2

这棵树的高度为 $\log_b n$，共有 $n^{\log_b a}$ 个叶子，从而 $T(n) = \Theta(n^{\log_b a}) + g(n)$，其中 $g(n) = \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j)$。

针对于第一种情况：$f(n) = O(n^{\log_b a - \epsilon})$，因此 $g(n) = O(n^{\log_b a})$。

对于第二种情况而言：首先 $g(n) = \Omega(f(n))$，又因为 $af(\frac{n}{b}) \leq cf(n)$，只要 $c$ 的取值是一个足够小的正数，且 $n$ 的取值足够大，因此可以推导出：$g(n) = O(f(n))$。两侧夹逼可以得出，$g(n) = \Theta(f(n))$。

而对于第三种情况：$f(n) = \Theta(n^{\log_b a})$，因此 $g(n) = O(n^{\log_b a} \log n)$。$T(n)$ 的结果可在 $g(n)$ 得出后显然得到。

下面举几个例子来说明主定理如何使用。

例如 $T(n) = T\left(\frac{n}{2}\right) + 1$，那么 $a = 1, b = 2, \log_2 1 = 0$，那么 $\epsilon$ 可以取值为 0，从而满足第一种情况，所以 $T(n) = \Theta(\log n)$。

又例如 $T(n) = T\left(\frac{n}{2}\right) + n$，那么 $a = 1, b = 2, \log_2 1 = 0$，那么 $\epsilon$ 可以取值为 0.5，从而满足第二种情况，所以 $T(n) = \Theta(n)$。

再例如 $T(n)=T\left(\frac{n}{2}\right)+\log n$，那么 $a=1,b=2,\log_2 1=0$，那么 $k$ 可以取值为 1，从而满足第三种情况，所以 $T(n)=\Theta(\log^2 n)$。

## 均摊复杂度

算法往往是会对内存中的数据进行修改的，而同一个算法的多次执行，就会通过对数据的修改而互相影响。

例如快速排序中的「按大小分类」操作，单次执行的最坏时间复杂度，看似是 $O(n)$ 的，但是由于快排的分治过程，先前的「分类」操作每次都减小了数组长度，所以实际的总复杂度 $O(n\log n)$，分摊在每一次「分类」操作上，是 $O(\log n)$。

多次操作的总复杂度除以操作次数，就是这种操作的**均摊复杂度**。

## 势能分析

势能分析，是一种求均摊复杂度上界的方法。

求均摊复杂度，关键是表达出先前操作对当前操作的影响。势能分析用一个函数来表达此种影响。

定义「状态」$S$：即某一时刻的所有数据。在快排的例子中，一个「状态」就是当前过程需要排序的下标区间

定义「初始状态」$S_0$：即未进行任何操作时的状态。在快排的例子中，「初始状态」就是整个数组

假设存在从状态到数的函数 $F$，且对于任何状态 $S$，$F(S)\geq F(S_0)$，则有以下推论：

设 $S_1,S_2,\cdots,S_m$ 为从 $S_0$ 开始连续做 $m$ 次操作所得的状态序列，$c_i$ 为第 $i$ 次操作的时间开销。

记 $p_i=c_i+F(S_i)-F(S_{i-1})$，则 $m$ 次操作的总时间花销为

$$\sum_{i=1}^{m}p_i+F(S_0)-F(S_m)$$

（正负相消，证明显然）

又因为 $F(S)\geq F(S_0)$，所以有

$$\sum_{i=1}^{m}p_i\geq\sum_{i=1}^{m}c_i$$

因此，若 $p_i=O(T(n))$，则 $O(T(n))$ 是均摊复杂度的一个上界。

势能分析在实际应用中有很多技巧，在此不详细展开。

## 空间复杂度

类似地，算法所使用的空间随输入规模变化的趋势可以用**空间复杂度**来衡量。

## 计算复杂性

本文主要从算法分析的角度对复杂度进行了介绍，如果有兴趣的话可以在计算复杂性进行更深入的了解。

# 1.3 枚举

**Authors:** Early0v0, frank-xjh, Great-designer, ksyx, qiqistyle, Tiphereth-A , Saisyc, shuzhouliu, Xeonacid, xyf007

本页面将简要介绍枚举算法。

# 简介

枚举（英语：Enumerate）是基于已有知识来猜测答案的一种问题求解策略。

枚举的思想是不断地猜测，从可能的集合中一一尝试，然后再判断题目的条件是否成立。

# 要点

## 给出解空间

建立简洁的数学模型。

枚举的时候要想清楚：可能的情况是什么？要枚举哪些要素？

## 减少枚举的空间

枚举的范围是什么？是所有的内容都需要枚举吗？

在用枚举法解决问题的时候，一定要想清楚这两件事，否则会带来不必要的时间开销。

## 选择合适的枚举顺序

根据题目判断。比如例题中要求的是最大的符合条件的素数，那自然是从大到小枚举比较合适。

# 例题

以下是一个使用枚举解题与优化枚举范围的例子。

**Note**

一个数组中的数互不相同，求其中和为 0 的数对的个数。

**解题思路**

枚举两个数的代码很容易就可以写出来。

=== "C++"

```cpp
for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j)
        if (a[i] + a[j] == 0) ++ans;
```

=== "Python"

```python
for i in range(n):
    for j in range(n):
        if a[i] + a[j] == 0:
            ans += 1
```

来看看枚举的范围如何优化。由于题中没要求数对是有序的，答案就是有序的情况的两倍（考虑如果 `(a, b)` 是答案，那么 `(b, a)` 也是答案）。对于这种情况，只需统计人为要求有顺序之后的答案，最后再乘上 2 就好了。

不妨要求第一个数要出现在靠前的位置。代码如下：

=== "C++"

```cpp
for (int i = 0; i < n; ++i)
    for (int j = 0; j < i; ++j)
        if (a[i] + a[j] == 0) ++ans;
```

=== "Python"

```python
for i in range(n):
    for j in range(i):
        if a[i] + a[j] == 0:
            ans += 1
```

不难发现这里已经减少了 $j$ 的枚举范围，减少了这段代码的时间开销。

我们可以在此之上进一步优化。

两个数是否都一定要枚举出来呢？枚举其中一个数之后，题目的条件已经确定了其他的要素（另一个数）的条件，如果能找到一种方法直接判断题目要求的那个数是否存在，就可以省掉枚举后一个数的时间了。较为进阶地，在数据范围允许的情况下，我们可以使用桶[1-1] 记录遍历过的数。

=== "C++"

```cpp
bool met[MAXN * 2];
memset(met, 0, sizeof(met));
for (int i = 0; i < n; ++i) {
    if (met[MAXN - a[i]]) ++ans;
    met[MAXN + a[i]] = true;
}
```

=== "Python"

```python
met = [False] * MAXN * 2
for i in range(n):
    if met[MAXN - a[i]]:
        ans += 1
    met[a[i] + MAXN] = True
```

## 复杂度分析

- 时间复杂度分析：对 $a$ 数组遍历了一遍就能完成题目要求，当 $n$ 足够大的时候时间复杂度为 $O(n)$。
- 空间复杂度分析：$O(n + \max\{|x| : x \in a\})$。

# 习题

- 2811: 熄灯问题 - OpenJudge[2]

## 脚注

[1] 桶排序 以及主元素问题以及 Stack Overflow 上对桶数据结构的讲解（英文）[1-1] [1-2]

[2] 2811: 熄灯问题 - OpenJudge

# 1.4 模拟

本页面将简要介绍模拟算法。

## 简介

模拟就是用计算机来模拟题目中要求的操作。

模拟题目通常具有码量大、操作多、思路繁复的特点。由于它码量大，经常会出现难以查错的情况，如果在考试中写错是相当浪费时间的。

## 技巧

写模拟题时，遵循以下的建议有可能会提升做题速度：

- 在动手写代码之前，在草纸上尽可能地写好要实现的流程。
- 在代码中，尽量把每个部分模块化，写成函数、结构体或类。
- 对于一些可能重复用到的概念，可以统一转化，方便处理：如，某题给你”YY-MM-DD 时：分” 把它抽取到一个函数，处理成秒，会减少概念混淆。
- 调试时分块调试。模块化的好处就是可以方便的单独调某一部分。
- 写代码的时候一定要思路清晰，不要想到什么写什么，要按照落在纸上的步骤写。

实际上，上述步骤在解决其它类型的题目时也是很有帮助的。

## 例题详解

### Climbing Worm

一只长度不计的蠕虫位于 $n$ 英寸深的井的底部。它每次向上爬 $u$ 英寸，但是必须休息一次才能再次向上爬。在休息的时候，它滑落了 $d$ 英寸。之后它将重复向上爬和休息的过程。蠕虫爬出井口需要至少爬多少次？如果蠕虫爬完后刚好到达井的顶部，我们也设作蠕虫已经爬出井口。

### 解题思路

直接使用程序模拟蠕虫爬井的过程就可以了。用一个循环重复蠕虫的爬井过程，当攀爬的长度超过或者等于井的深度时跳出。

### 参考代码

=== ”C++”

```cpp
#include <cstdio>
```

```
int main(void) {
  int n = 0, u = 0, d = 0;
  std::scanf("%d%d%d", &u, &d, &n);
  int time = 0, dist = 0;
  while (true) {  // 用死循环来枚举
    dist += u;
    time++;
    if (dist >= n) break;  // 满足条件则退出死循环
    dist -= d;
  }
  printf("%d\n", time);  // 输出得到的结果
  return 0;
}
```

=== "Python"

```python
u, d, n = map(int, input().split())
time = dist = 0
while True:  # 用死循环来枚举
    dist += u
    time += 1
    if dist >= n:  # 满足条件则退出死循环
        break
    dist -= d
print(time)  # 输出得到的结果
```

## 习题

- 「NOIP2014」 生活大爆炸版石头剪刀布 - Universal Online Judge[1]
- 「OpenJudge 3750」 魔兽世界[2]
- 「SDOI2010」 猪国杀 - LibreOJ[3]

### 参考资料与注释

[1] 「NOIP2014」生活大爆炸版石头剪刀布 - Universal Online Judge

[2] 「OpenJudge 3750」魔兽世界

[3] 「SDOI2010」猪国杀 - LibreOJ

# 1.5 递归 & 分治

> **Authors:** fudonglai, AngelKitty, labuladong

本页面将介绍递归与分治算法的区别与结合运用。

# 递归

## 定义

递归（英语：Recursion），在数学和计算机科学中是指在函数的定义中使用函数自身的方法，在计算机科学中还额外指一种通过重复将问题分解为同类的子问题而解决问题的方法。

## 引入

> 要理解递归，就得先理解什么是递归。

递归的基本思想是某个函数直接或者间接地调用自身，这样原问题的求解就转换为了许多性质相同但是规模更小的子问题。求解时只需要关注如何把原问题划分成符合条件的子问题，而不需要过分关注这个子问题是如何被解决的。

以下是一些有助于理解递归的例子：

1. 什么是递归？
2. 如何给一堆数字排序？答：分成两半，先排左半边再排右半边，最后合并就行了，至于怎么排左边和右边，请重新阅读这句话。
3. 你今年几岁？答：去年的岁数加一岁，1999 年我出生。


图 1.3 一个用于理解递归的例子

4.

递归在数学中非常常见。例如，集合论对自然数的正式定义是：1 是一个自然数，每个自然数都有一个后继，这一个后继也是自然数。

递归代码最重要的两个特征：结束条件和自我调用。自我调用是在解决子问题，而结束条件定义了最简子问题的答案。

```
int func(传入数值) {
  if (终止条件) return 最小子问题解;
  return func(缩小规模);
}
```

## 为什么要写递归

1. 结构清晰，可读性强。例如，分别用不同的方法实现 归并排序：

   === "C++"

   ```cpp
   // 不使用递归的归并排序算法
   template <typename T>
   void merge_sort(vector<T> a) {
   ```

```
  int n = a.size();
  for (int seg = 1; seg < n; seg = seg + seg)
    for (int start = 0; start < n - seg; start += seg + seg)
      merge(a, start, start + seg - 1, std::min(start + seg + seg - 1, n - 1));
}

// 使用递归的归并排序算法
template <typename T>
void merge_sort(vector<T> a, int front, int end) {
  if (front >= end) return;
  int mid = front + (end - front) / 2;
  merge_sort(a, front, mid);
  merge_sort(a, mid + 1, end);
  merge(a, front, mid, end);
}
```

=== "Python"

```python
# 不使用递归的归并排序算法
def merge_sort(a):
  n = len(a)
  seg, start = 1, 0
  while seg < n:
      while start < n - seg:
          merge(a, start, start + seg - 1, min(start + seg + seg - 1, n - 1))
          start = start + seg + seg
      seg = seg + seg

# 使用递归的归并排序算法
def merge_sort(a, front, end):
  if front >= end:
      return
  mid = front + (end - front) / 2
  merge_sort(a, front, mid)
  merge_sort(a, mid + 1, end)
  merge(a, front, mid, end)
```

显然，递归版本比非递归版本更易理解。递归版本的做法一目了然：把左半边排序，把右半边排序，最后合并两边。而非递归版本看起来不知所云，充斥着各种难以理解的边界计算细节，特别容易出 bug，且难以调试。

2. 练习分析问题的结构。当发现问题可以被分解成相同结构的小问题时，递归写多了就能敏锐发现这个特点，进而高效解决问题。

## 递归的缺点

在程序执行中，递归是利用堆栈来实现的。每当进入一个函数调用，栈就会增加一层栈帧，每次函数返回，栈就会减少一层栈帧。而栈不是无限大的，当递归层数过多时，就会造成**栈溢出**的后果。

显然有时候递归处理是高效的，比如归并排序；**有时候是低效的**，比如数孙悟空身上的毛，因为堆栈会消耗额外空间，而简单的递推不会消耗空间。比如这个例子，给一个链表头，计算它的长度：

```
// 典型的递推遍历框架
int size(Node *head) {
  int size = 0;
```

```
  for (Node *p = head; p != nullptr; p = p->next) size++;
  return size;
}

// 我就是要写递归，递归天下第一
int size_recursion(Node *head) {
  if (head == nullptr) return 0;
  return size_recursion(head->next) + 1;
}
```

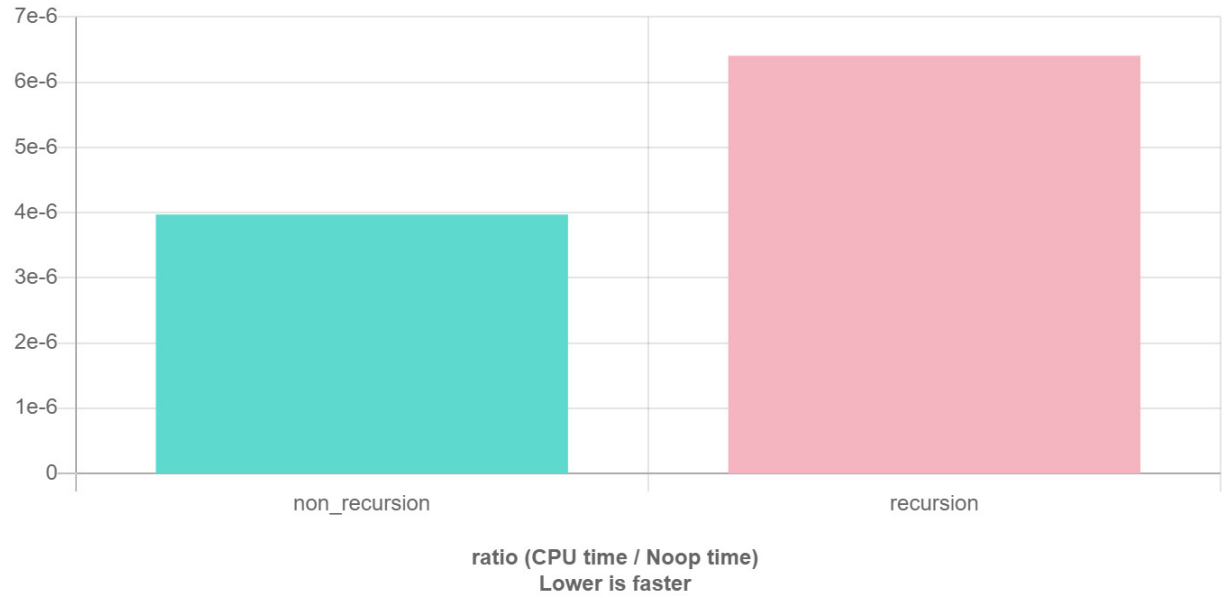


图 1.4 [二者的对比，compiler 设为 Clang 10.0，优化设为 O1](https://quick-bench.com/q/rZ7jWPmSdltparOO5ndLgmS9BVc)

## 递归的优化

主页面：搜索优化和记忆化搜索

比较初级的递归实现可能递归次数太多，容易超时。这时需要对递归进行优化。[1]

# 分治

## 定义

分治（英语：Divide and Conquer），字面上的解释是「分而治之」，就是把一个复杂的问题分成两个或更多的相同或相似的子问题，直到最后子问题可以简单的直接求解，原问题的解即子问题的解的合并。

## 过程

分治算法的核心思想就是「分而治之」。

大概的流程可以分为三步：分解 -> 解决 -> 合并。

1. 分解原问题为结构相同的子问题。
2. 分解到某个容易求解的边界之后，进行递归求解。
3. 将子问题的解合并成原问题的解。

分治法能解决的问题一般有如下特征：

- 该问题的规模缩小到一定的程度就可以容易地解决。
- 该问题可以分解为若干个规模较小的相同问题，即该问题具有最优子结构性质，利用该问题分解出的子问题的解可以合并为该问题的解。
- 该问题所分解出的各个子问题是相互独立的，即子问题之间不包含公共的子问题。

**注意**

如果各子问题是不独立的，则分治法要重复地解公共的子问题，也就做了许多不必要的工作。此时虽然也可用分治法，但一般用动态规划较好。

以归并排序为例。假设实现归并排序的函数名为 `merge_sort`。明确该函数的职责，即**对传入的一个数组排序**。这个问题显然可以分解。给一个数组排序等于给该数组的左右两半分别排序，然后合并成一个数组。

```
void merge_sort(一个数组) {
  if (可以很容易处理) return;
  merge_sort(左半个数组);
  merge_sort(右半个数组);
  merge(左半个数组, 右半个数组);
}
```

传给它半个数组，那么处理完后这半个数组就已经被排好了。注意到，`merge_sort` 与二叉树的后序遍历模板极其相似。因为分治算法的套路是**分解 -> 解决（触底）-> 合并（回溯）**，先左右分解，再处理合并，回溯就是在退栈，即相当于后序遍历。

`merge` 函数的实现方式与两个有序链表的合并一致。

# 要点

## 写递归的要点

**明白一个函数的作用并相信它能完成这个任务，千万不要跳进这个函数里面企图探究更多细节，**否则就会陷入无穷的细节无法自拔，人脑能压几个栈啊。

以遍历二叉树为例。

```
void traverse(TreeNode* root) {
  if (root == nullptr) return;
  traverse(root->left);
  traverse(root->right);
}
```

这几行代码就足以遍历任何一棵二叉树了。对于递归函数 `traverse(root)`，只要相信给它一个根节点 `root`，它就能遍历这棵树。所以只需要把这个节点的左右节点再传给这个函数就行了。

同样扩展到遍历一棵 N 叉树。与二叉树的写法一模一样。不过，对于 N 叉树，显然没有中序遍历。

```
void traverse(TreeNode* root) {
  if (root == nullptr) return;
  for (auto child : root->children) traverse(child);
}
```

# 区别

## 递归与枚举的区别

递归和枚举的区别在于：枚举是横向地把问题划分，然后依次求解子问题；而递归是把问题逐级分解，是纵向的拆分。

## 递归与分治的区别

递归是一种编程技巧，一种解决问题的思维方式；分治算法很大程度上是基于递归的，解决更具体问题的算法思想。

# 例题详解

### 437. 路径总和 III

给定一个二叉树，它的每个结点都存放着一个整数值。

找出路径和等于给定数值的路径总数。

路径不需要从根节点开始，也不需要在叶子节点结束，但是路径方向必须是向下的（只能从父节点到子节点）。

二叉树不超过 1000 个节点，且节点数值范围是 [-1000000,1000000] 的整数。

示例：

```
root = [10,5,-3,3,2,null,11,3,-2,null,1], sum = 8

      10
     /  \
    5   -3
   / \    \
  3   2   11
 / \   \
3  -2   1

返回 3。和等于 8 的路径有:

1.  5 -> 3
2.  5 -> 2 -> 1
3. -3 -> 11
```

```
/**
 * 二叉树结点的定义
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
```

### 参考代码

```
int pathSum(TreeNode *root, int sum) {
  if (root == nullptr) return 0;
  return count(root, sum) + pathSum(root->left, sum) +
         pathSum(root->right, sum);
}

int count(TreeNode *node, int sum) {
  if (node == nullptr) return 0;
  return (node->val == sum) + count(node->left, sum - node->val) +
         count(node->right, sum - node->val);
}
```

**题目解析**

题目看起来很复杂，不过代码却极其简洁。

首先明确，递归求解树的问题必然是要遍历整棵树的，所以二叉树的遍历框架（分别对左右子树递归调用函数本身）必然要出现在主函数 pathSum 中。那么对于每个节点，它们应该干什么呢？它们应该看看，自己和它们的子树包含多少条符合条件的路径。好了，这道题就结束了。

按照前面说的技巧，根据刚才的分析来定义清楚每个递归函数应该做的事：

PathSum 函数：给定一个节点和一个目标值，返回以这个节点为根的树中，和为目标值的路径总数。

count 函数：给定一个节点和一个目标值，返回以这个节点为根的树中，能凑出几个以该节点为路径开头，和为目标值的路径总数。

## 参考代码（附注释）

```
int pathSum(TreeNode *root, int sum) {
  if (root == nullptr) return 0;
  int pathImLeading = count(root, sum);  // 自己为开头的路径数
  int leftPathSum = pathSum(root->left, sum);  // 左边路径总数（相信它能算出来）
  int rightPathSum =
      pathSum(root->right, sum);  // 右边路径总数（相信它能算出来）
  return leftPathSum + rightPathSum + pathImLeading;
}

int count(TreeNode *node, int sum) {
  if (node == nullptr) return 0;
  // 能不能作为一条单独的路径呢？
  int isMe = (node->val == sum) ? 1 : 0;
  // 左边的，你那边能凑几个 sum - node.val ？
  int leftNode = count(node->left, sum - node->val);
  // 右边的，你那边能凑几个 sum - node.val ？
  int rightNode = count(node->right, sum - node->val);
  return isMe + leftNode + rightNode;  // 我这能凑这么多个
}
```

还是那句话，**明白每个函数能做的事，并相信它们能够完成**。

总结下，PathSum 函数提供了二叉树遍历框架，在遍历中对每个节点调用 count 函数（这里用的是先序遍历，不过中序遍历和后序遍历也可以）。count 函数也是一个二叉树遍历，用于寻找以该节点开头的目标值路径。

## 习题

- LeetCode 上的递归专题练习[2]
- LeetCode 上的分治算法专项练习[3]

## 参考资料与注释

[1] labuladong 的算法小抄 - 递归详解

[2] LeetCode 上的递归专题练习

[3] LeetCode 上的分治算法专项练习

# 1.6 贪心

本页面将简要介绍贪心算法。

## 引入

贪心算法（英语：greedy algorithm），是用计算机来模拟一个「贪心」的人做出决策的过程。这个人十分贪婪，每一步行动总是按某种指标选取最优的操作。而且他目光短浅，总是只看眼前，并不考虑以后可能造成的影响。

可想而知，并不是所有的时候贪心法都能获得最优解，所以一般使用贪心法的时候，都要确保自己能证明其正确性。

## 解释

### 适用范围

贪心算法在有最优子结构的问题中尤为有效。最优子结构的意思是问题能够分解成子问题来解决，子问题的最优解能递推到最终问题的最优解。[1]

### 证明

贪心算法有两种证明方法：反证法和归纳法。一般情况下，一道题只会用到其中的一种方法来证明。

1. 反证法：如果交换方案中任意两个元素/相邻的两个元素后，答案不会变得更好，那么可以推定目前的解已经是最优解了。
2. 归纳法：先算得出边界情况（例如 $n = 1$）的最优解 $F_1$，然后再证明：对于每个 $n$，$F_{n+1}$ 都可以由 $F_n$ 推导出结果。

## 要点

### 常见题型

在提高组难度以下的题目中，最常见的贪心有两种。

- 「我们将 XXX 按照某某顺序排序，然后按某种顺序（例如从小到大）选择。」。
- 「我们每次都取 XXX 中最大/小的东西，并更新 XXX。」（有时「XXX 中最大/小的东西」可以优化，比如用优先队列维护）

二者的区别在于一种是离线的，先处理后选择；一种是在线的，边处理边选择。

### 排序解法

用排序法常见的情况是输入一个包含几个（一般一到两个）权值的数组，通过排序然后遍历模拟计算的方法求出最优值。

### 后悔解法

思路是无论当前的选项是否最优都接受，然后进行比较，如果选择之后不是最优了，则反悔，舍弃掉这个选项；否则，正式接受。如此往复。

## 区别

### 与动态规划的区别

贪心算法与动态规划的不同在于它对每个子问题的解决方案都做出选择，不能回退。动态规划则会保存以前的运算结果，并根据以前的结果对当前进行选择，有回退功能。

## 例题详解

### 邻项交换法的例题

**NOIP 2012 国王游戏**

恰逢 H 国国庆，国王邀请 n 位大臣来玩一个有奖游戏。首先，他让每个大臣在左、右手上面分别写下一个整数，国王自己也在左、右手上各写一个整数。然后，让这 n 位大臣排成一排，国王站在队伍的最前面。排好队后，所有的大臣都会获得国王奖赏的若干金币，每位大臣获得的金币数分别是：排在该大臣前面的所有人的左手上的数的乘积除以他自己右手上的数，然后向下取整得到的结果。

国王不希望某一个大臣获得特别多的奖赏，所以他想请你帮他重新安排一下队伍的顺序，使得获得奖赏最多的大臣，所获奖赏尽可能的少。注意，国王的位置始终在队伍的最前面。

**解题思路**

设排序后第 $i$ 个大臣左右手上的数分别为 $a_i, b_i$。考虑通过邻项交换法推导贪心策略。

用 $s$ 表示第 $i$ 个大臣前面所有人的 $a_i$ 的乘积，那么第 $i$ 个大臣得到的奖赏就是 $\frac{s}{b_i}$，第 $i+1$ 个大臣得到的奖赏就是 $\frac{s \cdot a_i}{b_{i+1}}$。

如果我们交换第 $i$ 个大臣与第 $i+1$ 个大臣，那么此时的第 $i$ 个大臣得到的奖赏就是 $\frac{s}{b_{i+1}}$，第 $i+1$ 个大臣得到的奖赏就是 $\frac{s \cdot a_{i+1}}{b_i}$。

如果交换前更优当且仅当

$$\max\left(\frac{s}{b_i}, \frac{s \cdot a_i}{b_{i+1}}\right) < \max\left(\frac{s}{b_{i+1}}, \frac{s \cdot a_{i+1}}{b_i}\right)$$

提取出相同的 $s$ 并约分得到

$$\max\left(\frac{1}{b_i}, \frac{a_i}{b_{i+1}}\right) < \max\left(\frac{1}{b_{i+1}}, \frac{a_{i+1}}{b_i}\right)$$

然后分式化成整式得到

$$\max(b_{i+1}, a_i \cdot b_i) < \max(b_i, a_{i+1} \cdot b_{i+1})$$

实现的时候我们将输入的两个数用一个结构体来保存并重载运算符：

```
struct uv {
  int a, b;

  bool operator<(const uv &x) const {
    return max(x.b, a * b) < max(b, x.a * x.b);
  }
};
```

## 后悔法的例题

**「USACO09OPEN」 工作调度 Work Scheduling**

约翰的工作日从 0 时刻开始，有 $10^9$ 个单位时间。在任一单位时间，他都可以选择编号 1 到 $N$ 的 $N(1 \le N \le 10^5)$ 项工作中的任意一项工作来完成。工作 $i$ 的截止时间是 $D_i(1 \le D_i \le 10^9)$，完成后获利是 $P_i(1 \le P_i \le 10^9)$。在给定的工作利润和截止时间下，求约翰能够获得的利润最大为多少。

**解题思路**

1. 先假设每一项工作都做，将各项工作按截止时间排序后入队；
2. 在判断第 `i` 项工作做与不做时，若其截至时间符合条件，则将其与队中报酬最小的元素比较，若第 `i` 项工作报酬较高（后悔），则 `ans += a[i].p - q.top()`。

   用优先队列（小根堆）来维护队首元素最小。
3. 当 `a[i].d<=q.size()` 可以这么理解从 0 开始到 `a[i].d` 这个时间段只能做 `a[i].d` 个任务，而若 `q.size()>=a[i].d` 说明完成 `q.size()` 个任务时间大于等于 `a[i].d` 的时间，所以当第 `i` 个任务获利比较大的时候应该把最小的任务从优先级队列中换出。

**参考代码**

=== "C++"

```cpp
#include <algorithm>
#include <cmath>
#include <cstdio>
#include <cstring>
#include <iostream>
#include <queue>
using namespace std;

struct f {
  long long d;
  long long p;
} a[100005];

bool cmp(f A, f B) { return A.d < B.d; }

priority_queue<long long, vector<long long>, greater<long long> >
    q;  // 小根堆维护最小值

int main() {
  long long n, i;
```

```
  cin >> n;
  for (i = 1; i <= n; i++) {
    scanf("%lld%lld", &a[i].d, &a[i].p);
  }
  sort(a + 1, a + n + 1, cmp);
  long long ans = 0;
  for (i = 1; i <= n; i++) {
    if (a[i].d <= (int)q.size()) {  // 超过截止时间
      if (q.top() < a[i].p) {       // 后悔
        ans += a[i].p - q.top();
        q.pop();
        q.push(a[i].p);
      }
    } else {  // 直接加入队列
      ans += a[i].p;
      q.push(a[i].p);
    }
  }
  cout << ans << endl;
  return 0;
}

```

=== ”Python”

```python
from collections import defaultdict
from heapq import heappush, heappop

a = defaultdict(list)
for _ in range(int(input())):
    d, p = map(int, input().split())
    a[d].append(p)  # 存放对应时间的收益

ans = 0  # 记录总收益
q = []  # 小根堆维护最小值
l = sorted(a.keys(), reverse=True)
for i, j in zip(l, l[1:] + [0]):
    for k in a.pop(i):
        heappush(q, ~k)
    for _ in range(i - j):
        if q:  # 从堆中取出收益最多的工作
            ans += ~heappop(q)
        else:  # 堆为空时退出循环
            break
print(ans)
```

**复杂度分析**

- 空间复杂度：当输入 $n$ 个任务时使用 $n$ 个 $a$ 数组元素，优先队列中最差情况下会储存 $n$ 个元素，则空间复杂度为 $O(n)$。
- 时间复杂度：`std::sort` 的时间复杂度为 $O(n\log n)$，维护优先队列的时间复杂度为 $O(n\log n)$，综上所述，时间复杂度为 $O(n\log n)$。

# 习题

- P1209[USACO1.3] 修理牛棚 Barn Repair - 洛谷[2]
- P2123 皇后游戏 - 洛谷[3]
- LeetCode 上标签为贪心算法的题目[4]

# 参考资料与注释

[1] 贪心算法 - 维基百科，自由的百科全书

[2] P1209[USACO1.3] 修理牛棚 Barn Repair - 洛谷

[3] P2123 皇后游戏 - 洛谷

[4] LeetCode 上标签为贪心算法的题目

# 1.7 排序

## 1.7.1 排序简介

本页面将简要介绍排序算法。

### 定义

**排序算法**（英语：Sorting algorithm）是一种将一组特定的数据按某种顺序进行排列的算法。排序算法多种多样，性质也大多不同。

### 性质

#### 稳定性

稳定性是指相等的元素经过排序之后相对顺序是否发生了改变。

拥有稳定性这一特性的算法会让原本有相等键值的纪录维持相对次序，即如果一个排序算法是稳定的，当有两个相等键值的纪录 $R$ 和 $S$，且在原本的列表中 $R$ 出现在 $S$ 之前，在排序过的列表中 $R$ 也将会是在 $S$ 之前。

基数排序、计数排序、插入排序、冒泡排序、归并排序是稳定排序。

选择排序、堆排序、快速排序、希尔排序不是稳定排序。

#### 时间复杂度

主页面：复杂度

时间复杂度用来衡量一个算法的运行时间和输入规模的关系，通常用 $O$ 表示。

简单计算复杂度的方法一般是统计「简单操作」的执行次数，有时候也可以直接数循环的层数来近似估计。

时间复杂度分为最优时间复杂度、平均时间复杂度和最坏时间复杂度。OI 竞赛中要考虑的一般是最坏时间复杂度，因为它代表的是算法运行水平的下界，在评测中不会出现更差的结果了。

基于比较的排序算法的时间复杂度下限是 $O(n \log n)$ 的。

当然也有不是 $O(n \log n)$ 的。例如，计数排序 的时间复杂度是 $O(n + w)$，其中 $w$ 代表输入数据的值域大小。

以下是几种排序算法的比较。


图 1.5 几种排序算法的比较

### 空间复杂度

与时间复杂度类似，空间复杂度用来描述算法空间消耗的规模。一般来说，空间复杂度越小，算法越好。

## 外部链接

- 排序算法 - 维基百科 ， 自由的百科全书[1]

## 参考资料与注释

[1] 排序算法 - 维基百科，自由的百科全书

# 1.7.2 选择排序

本页面将简要介绍选择排序。

## 定义

选择排序（英语：Selection sort）是一种简单直观的排序算法。它的工作原理是每次找出第 $i$ 小的元素（也就是 $A_{i..n}$ 中最小的元素），然后将这个元素与数组第 $i$ 个位置上的元素交换。

## 性质

### 稳定性

由于 swap（交换两个元素）操作的存在，选择排序是一种不稳定的排序算法。

### 时间复杂度

选择排序的最优时间复杂度、平均时间复杂度和最坏时间复杂度均为 $O(n^2)$。

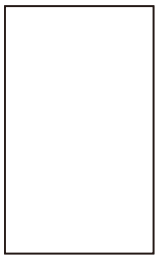

图 1.6 selection sort animate example

# 代码实现

## 伪代码

1 **Input.** An array $A$ consisting of $n$ elements.
2 **Output.** $A$ will be sorted in nondecreasing order.
3 **Method.**
4 **for** $i \leftarrow 1$ **to** $n-1$
5 $\quad ith \leftarrow i$
6 $\quad$ **for** $j \leftarrow i+1$ **to** $n$
7 $\quad\quad$ **if** $A[j] < A[ith]$
8 $\quad\quad\quad ith \leftarrow j$
9 $\quad$ swap $A[i]$ and $A[ith]$

=== "C++"

```cpp
void selection_sort(int* a, int n) {
  for (int i = 1; i < n; ++i) {
    int ith = i;
    for (int j = i + 1; j <= n; ++j) {
      if (a[j] < a[ith]) {
        ith = j;
      }
    }
    std::swap(a[i], a[ith]);
```

```
  }
}
```
```

=== "Python"

```python
def selection_sort(a, n):
    for i in range(1, n):
        ith = i
        for j in range(i + 1, n + 1):
            if a[j] < a[ith]:
                ith = j
        a[i], a[ith] = a[ith], a[i]
```

## 1.7.3 冒泡排序

本页面将简要介绍冒泡排序。

### ci-test

This is a test.

### 定义

冒泡排序（英语：Bubble sort）是一种简单的排序算法。由于在算法的执行过程中，较小的元素像是气泡般慢慢「浮」到数列的顶端，故叫做冒泡排序。

### 过程

它的工作原理是每次检查相邻两个元素，如果前面的元素与后面的元素满足给定的排序条件，就将相邻两个元素交换。当没有相邻的元素需要交换时，排序就完成了。

经过 $i$ 次扫描后，数列的末尾 $i$ 项必然是最大的 $i$ 项，因此冒泡排序最多需要扫描 $n-1$ 遍数组就能完成排序。

### 性质

#### 稳定性

冒泡排序是一种稳定的排序算法。

#### 时间复杂度

在序列完全有序时，冒泡排序只需遍历一遍数组，不用执行任何交换操作，时间复杂度为 $O(n)$。

在最坏情况下，冒泡排序要执行 $\frac{(n-1)n}{2}$ 次交换操作，时间复杂度为 $O(n^2)$。

冒泡排序的平均时间复杂度为 $O(n^2)$。
```

## 代码实现

### 伪代码

```
1   Input. An array A consisting of n elements.
2   Output. A will be sorted in nondecreasing order stably.
3   Method.
4   flag ← True
5   while flag
6       flag ← False
7       for i ← 1 to n − 1
8           if A[i] > A[i + 1]
9               flag ← True
10              Swap A[i] and A[i + 1]
```

=== "C++"

```cpp
// 假设数组的大小是 n + 1，冒泡排序从数组下标 1 开始
void bubble_sort(int *a, int n) {
  bool flag = true;
  while (flag) {
    flag = false;
    for (int i = 1; i < n; ++i) {
      if (a[i] > a[i + 1]) {
        flag = true;
        int t = a[i];
        a[i] = a[i + 1];
        a[i + 1] = t;
      }
    }
  }
}
```

=== "Python"

```python
def bubble_sort(a, n):
    flag = True
    while flag:
        flag = False
        for i in range(1, n):
            if a[i] > a[i + 1]:
                flag = True
                a[i], a[i + 1] = a[i + 1], a[i]
```

# 1.7.4 插入排序

本页面将简要介绍插入排序。

## 定义

插入排序（英语：Insertion sort）是一种简单直观的排序算法。它的工作原理为将待排列元素划分为「已排序」和「未排序」两部分，每次从「未排序的」元素中选择一个插入到「已排序的」元素中的正确位置。

一个与插入排序相同的操作是打扑克牌时，从牌桌上抓一张牌，按牌面大小插到手牌后，再抓下一张牌。

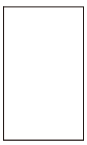

图 1.7 insertion sort animate example

## 性质

### 稳定性

插入排序是一种稳定的排序算法。

### 时间复杂度

插入排序的最优时间复杂度为 $O(n)$，在数列几乎有序时效率很高。

插入排序的最坏时间复杂度和平均时间复杂度都为 $O(n^2)$。

## 代码实现

### 伪代码

```
1   Input. An array A consisting of n elements.
2   Output. A will be sorted in nondecreasing order stably.
3   Method.
4   for i ← 2 to n
5       key ← A[i]
6       j ← i − 1
7       while j > 0 and A[j] > key
8           A[j + 1] ← A[j]
9           j ← j − 1
10      A[j + 1] ← key
```

=== "C++"

```cpp
void insertion_sort(int arr[], int len) {
    for (int i = 1; i < len; ++i) {
        int key = arr[i];
        int j = i - 1;
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}
```

=== "Python"

```python
def insertion_sort(arr, n):
    for i in range(1, n):
        key = arr[i]
        j = i - 1
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j = j - 1
        arr[j + 1] = key
```

### 折半插入排序

插入排序还可以通过二分算法优化性能，在排序元素数量较多时优化的效果比较明显。

#### 时间复杂度

折半插入排序与直接插入排序的基本思想是一致的，折半插入排序仅对插入排序时间复杂度中的常数进行了优化，所以优化后的时间复杂度仍然不变。

#### 代码实现

=== "C++"

```cpp
void insertion_sort(int arr[], int len) {
    if (len < 2) return;
    for (int i = 1; i != len; ++i) {
        int key = arr[i];
        auto index = upper_bound(arr, arr + i, key) - arr;
        // 使用 memmove 移动元素，比使用 for 循环速度更快，时间复杂度仍为 O(n)
        memmove(arr + index + 1, arr + index, (i - index) * sizeof(int));
        arr[i] = key;
    }
}
```

## 1.7.5 计数排序

Warning

本页面要介绍的不是 **基数排序**。

本页面将简要介绍计数排序。

# 定义

计数排序（英语：Counting sort）是一种线性时间的排序算法。

# 过程

计数排序的工作原理是使用一个额外的数组 $C$，其中第 $i$ 个元素是待排序数组 $A$ 中值等于 $i$ 的元素的个数，然后根据数组 $C$ 来将 $A$ 中的元素排到正确的位置。[1]

它的工作过程分为三个步骤：

1. 计算每个数出现了几次；
2. 求出每个数出现次数的 前缀和；
3. 利用出现次数的前缀和，从右至左计算每个数的排名。

## 计算前缀和的原因

阅读本章内容只需要了解前缀和概念即可

直接将 $C$ 中正数对应的元素依次放入 $A$ 中不能解决元素重复的情形。

我们通过为额外数组 $C$ 中的每一项计算前缀和，结合每一项的数值，就可以为重复元素确定一个唯一排名：

额外数组 $C$ 中每一项的数值即是该 key 值下重复元素的个数，而该项的前缀和即是排在最后一个的重复元素的排名。

如果按照 $A$ 的逆序进行排列，那么显然排序后的数组将保持 $A$ 的原序（相同 key 值情况下），也即得到一种稳定的排序算法。

# 性质

## 稳定性

计数排序是一种稳定的排序算法。

## 时间复杂度

计数排序的时间复杂度为 $O(n+w)$，其中 $w$ 代表待排序数据的值域大小。

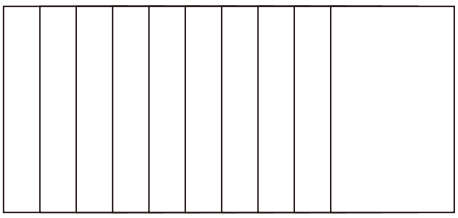

图 1.8 counting sort animate example

# 代码实现

## 伪代码

```
1   Input. An array A consisting of n positive integers no greater than w.
2   Output. Array A after sorting in nondecreasing order stably.
3   Method.
4   for i ← 0 to w
5       cnt[i] ← 0
6   for i ← 1 to n
7       cnt[A[i]] ← cnt[A[i]] + 1
8   for i ← 1 to w
9       cnt[i] ← cnt[i] + cnt[i − 1]
10  for i ← n downto 1
11      B[cnt[A[i]]] ← A[i]
12      cnt[A[i]] ← cnt[A[i]] − 1
13  return B
```

=== "C++"

```cpp
const int N = 100010;
const int W = 100010;

int n, w, a[N], cnt[W], b[N];
```

```
void counting_sort() {
    memset(cnt, 0, sizeof(cnt));
    for (int i = 1; i <= n; ++i) ++cnt[a[i]];
    for (int i = 1; i <= w; ++i) cnt[i] += cnt[i - 1];
    for (int i = n; i >= 1; --i) b[cnt[a[i]]--] = a[i];
}
```

=== "Python"

```python
N = W = 100010
n = w = 0
a = b = [0] * N
cnt = [0] * W

def counting_sort():
    for i in range(1, n + 1):
        cnt[a[i]] += 1
    for i in range(1, w + 1):
        cnt[i] += cnt[i - 1]
    for i in range(n, 0, -1):
        b[cnt[a[i]] - 1] = a[i]
        cnt[a[i]] -= 1
```

## 参考资料与注释

[1] 计数排序 - 维基百科， 自由的百科全书

# 1.7.6 基数排序

**提醒**

本页面要介绍的不是 **计数排序**。

本页面将简要介绍基数排序。

## 定义

基数排序（英语：Radix sort）是一种非比较型的排序算法，最早用于解决卡片排序的问题。基数排序将待排序的元素拆分为 $k$ 个关键字，逐一对各个关键字排序后完成对所有元素的排序。

如果是从第 1 关键字到第 $k$ 关键字顺序进行比较，则该基数排序称为 MSD（Most Significant Digit first）基数排序；

如果是从第 $k$ 关键字到第 1 关键字顺序进行比较，则该基数排序称为 LSD（Least Significant Digit first）基数排序。

## k - 关键字元素的比较

下面用 $a_i$ 表示元素 $a$ 的第 $i$ 关键字。

假如元素有 $k$ 个关键字，对于两个元素 $a$ 和 $b$，默认的比较方法是：

- 比较两个元素的第 1 关键字 $a_1$ 和 $b_1$，如果 $a_1 < b_1$ 则 $a < b$，如果 $a_1 > b_1$ 则 $a > b$，如果 $a_1 = b_1$ 则进行下一步；
- 比较两个元素的第 2 关键字 $a_2$ 和 $b_2$，如果 $a_2 < b_2$ 则 $a < b$，如果 $a_2 > b_2$ 则 $a > b$，如果 $a_2 = b_2$ 则进行下一步；
- ……
- 比较两个元素的第 $k$ 关键字 $a_k$ 和 $b_k$，如果 $a_k < b_k$ 则 $a < b$，如果 $a_k > b_k$ 则 $a > b$，如果 $a_k = b_k$ 则 $a = b$。

例子：

- 如果对自然数进行比较，将自然数按个位对齐后往高位补齐 0，则一个数字从左往右数第 $i$ 位数就可以作为第 $i$ 关键字；
- 如果对字符串基于字典序进行比较，一个字符串从左往右数第 $i$ 个字符就可以作为第 $i$ 关键字；
- C++ 自带的 `std::pair` 与 `std::tuple` 的默认比较方法与上述的相同。

## MSD 基数排序

基于 k - 关键字元素的比较方法，可以想到：先比较所有元素的第 1 关键字，就可以确定出各元素大致的大小关系；然后对**具有相同第 1 关键字的元素**，再比较它们的第 2 关键字……以此类推。

由于是从第 1 关键字到第 $k$ 关键字顺序进行比较，由上述思想导出的排序算法称为 MSD（Most Significant Digit first）基数排序。

### 算法流程

将待排序的元素拆分为 $k$ 个关键字，先对第 1 关键字进行稳定排序，然后对于每组**具有相同关键字的元素**再对第 2 关键字进行稳定排序（递归执行）……最后对于每组**具有相同关键字的元素**再对第 $k$ 关键字进行稳定排序。

MSD 基数排序需要借助一种**稳定算法**完成内层对关键字的排序，通常使用计数排序来完成。

正确性参考上文 k - 关键字元素的比较。

### 参考代码

#### 对自然数排序

下面是使用迭代式 MSD 基数排序对 `unsigned int` 范围内元素进行排序的 C++ 参考代码，可调整 $W$ 和 $\log_2 W$ 的值（建议将 $\log_2 W$ 设为 $2^k$ 以便位运算优化）。

```cpp
#include <algorithm>
#include <stack>
#include <tuple>
#include <vector>

using std::copy;  // from <algorithm>
using std::make_tuple;
using std::stack;
using std::tie;
using std::tuple;
using std::vector;

typedef unsigned int u32;
typedef unsigned int* u32ptr;

void MSD_radix_sort(u32ptr first, u32ptr last) {
  const size_t maxW = 0x100000000llu;
  const u32 maxlogW = 32;  // = log_2 W
```

```
  const u32 W = 256;  // 计数排序的值域
  const u32 logW = 8;
  const u32 mask = W - 1;  // 用位运算替代取模，详见下面的 key 函数

  u32ptr tmp =
      (u32ptr)calloc(last - first, sizeof(u32));  // 计数排序用的输出空间

  typedef tuple<u32ptr, u32ptr, u32> node;
  stack<node, vector<node>> s;
  s.push(make_tuple(first, last, maxlogW - logW));

  while (!s.empty()) {
    u32ptr begin, end;
    size_t shift, length;

    tie(begin, end, shift) = s.top();
    length = end - begin;
    s.pop();

    if (begin + 1 >= end) continue;  // elements <= 1

    // 计数排序
    u32 cnt[W] = {};
    auto key = [](const u32 x, const u32 shift) { return (x >> shift) & mask; };

    for (u32ptr it = begin; it != end; ++it) ++cnt[key(*it, shift)];
    for (u32 value = 1; value < W; ++value) cnt[value] += cnt[value - 1];

    // 求完前缀和后，计算相同关键字的元素范围
    if (shift >= logW) {
      s.push(make_tuple(begin, begin + cnt[0], shift - logW));
      for (u32 value = 1; value < W; ++value)
        s.push(make_tuple(begin + cnt[value - 1], begin + cnt[value],
                          shift - logW));
    }

    u32ptr it = end;
    do {
      --it;
      --cnt[key(*it, shift)];
      tmp[cnt[key(*it, shift)]] = *it;
    } while (it != begin);

    copy(tmp, tmp + length, begin);
  }
}
```

**对字符串排序**

下面是使用迭代式 MSD 基数排序对 空终止字节字符串[2] 基于字典序进行排序的 C++ 参考代码：

```
#include <algorithm>
#include <stack>
#include <tuple>
```

```cpp
#include <vector>

using std::copy;  // from <algorithm>
using std::make_tuple;
using std::stack;
using std::tie;
using std::tuple;
using std::vector;

typedef char* NTBS;  // 空终止字节字符串
typedef NTBS* NTBSptr;

void MSD_radix_sort(NTBSptr first, NTBSptr last) {
  const size_t W = 128;
  const size_t logW = 7;
  const size_t mask = W - 1;

  NTBSptr tmp = (NTBSptr)calloc(last - first, sizeof(NTBS));

  typedef tuple<NTBSptr, NTBSptr, size_t> node;
  stack<node, vector<node>> s;
  s.push(make_tuple(first, last, 0));

  while (!s.empty()) {
    NTBSptr begin, end;
    size_t index, length;

    tie(begin, end, index) = s.top();
    length = end - begin;
    s.pop();

    if (begin + 1 >= end) continue;  // elements <= 1

    // 计数排序
    size_t cnt[W] = {};
    auto key = [](const NTBS str, const size_t index) { return str[index]; };

    for (NTBSptr it = begin; it != end; ++it) ++cnt[key(*it, index)];
    for (char ch = 1; value < W; ++value) cnt[ch] += cnt[ch - 1];

    // 求完前缀和后，计算相同关键字的元素范围
    // 对于 NTBS，如果此刻末尾的字符是 \0 则说明这两个字符串相等，不必继续迭代
    for (char ch = 1; ch < W; ++ch)
      s.push(make_tuple(begin + cnt[ch - 1], begin + cnt[ch], index + 1));

    NTBSptr it = end;
    do {
      --it;
      --cnt[key(*it, index)];
      tmp[cnt[key(*it, index)]] = *it;
    } while (it != begin);

    copy(tmp, tmp + length, begin);
  }
```

```
    free(tmp);
}
```

由于两个字符串的比较很容易冲上 $O(n)$ 的线性复杂度，因此在字符串排序这件事情上，MSD 基数排序比大多数基于比较的排序算法在时间复杂度和实际用时上都更加优秀。

### 与桶排序的关系

前置知识：桶排序

桶排序需要其它的排序算法来完成对每个桶内部元素的排序。但实际上，完全可以对每个桶继续执行桶排序，直至某一步桶的元素数量 $\leq 1$。

因此 MSD 基数排序的另一种理解方式是：使用桶排序实现的桶排序。

也因此，可以提出 MSD 基数排序在时间常数上的一种优化方法：假如到某一步桶的元素数量 $\leq B$（$B$ 是自己选的常数），则直接执行插入排序然后返回，降低递归次数。

## LSD 基数排序

MSD 基数排序从第 1 关键字到第 $k$ 关键字顺序进行比较，为此需要借助递归或迭代来实现，时间常数还是较大，而且在比较自然数上还是略显不便。

而将递归的操作反过来：从第 $k$ 关键字到第 1 关键字顺序进行比较，就可以得到 LSD（Least Significant Digit first）基数排序，不使用递归就可以完成的排序算法。

### 算法流程

将待排序的元素拆分为 $k$ 个关键字，然后先对**所有元素**的第 $k$ 关键字进行稳定排序，再对**所有元素**的第 $k-1$ 关键字进行稳定排序，再对**所有元素**的第 $k-2$ 关键字进行稳定排序……最后对**所有元素**的第 1 关键字进行稳定排序，这样就完成了对整个待排序序列的稳定排序。

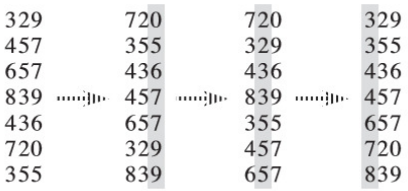


图 1.9 一个 LSD 基数排序全流程的例子

LSD 基数排序也需要借助一种**稳定算法**完成内层对关键字的排序。同样的，通常使用计数排序来完成。

LSD 基数排序的正确性可以参考 《 算法导论 （ 第三版 ）》 第 8.3-3 题的解法[3] 或参考下面的解释：

### 正确性

回顾一下 k - 关键字元素的比较方法，

- 假如想通过 $a_1$ 和 $b_1$ 就比较出两个元素 $a$ 和 $b$ 的大小，则需要提前知道通过比较 $a_2$ 和 $b_2$ 得到的结论，以便于应对 $a_1 = b_1$ 的情况；
- 而想通过 $a_2$ 和 $b_2$ 就比较出两个元素 $a$ 和 $b$ 的大小，则需要提前知道通过比较 $a_3$ 和 $b_3$ 得到的结论，以便于应对 $a_2 = b_2$ 的情况；
- ……
- 而想通过 $a_{k-1}$ 和 $b_{k-1}$ 就比较出两个元素 $a$ 和 $b$ 的大小，则需要提前知道通过比较 $a_k$ 和 $b_k$ 得到的结论，以便于应对 $a_{k-1} = b_{k-1}$ 的情况；
- $a_k$ 和 $b_k$ 可以直接比较。

现在，将顺序反过来：

- $a_k$ 和 $b_k$ 可以直接比较；
- 而知道通过比较 $a_k$ 和 $b_k$ 得到的结论后，就可以得到比较 $a_{k-1}$ 和 $b_{k-1}$ 的结论；
- ……
- 而知道通过比较 $a_2$ 和 $b_2$ 得到的结论后，就可以得到比较 $a_1$ 和 $b_1$ 的结论；
- 而知道通过比较 $a_1$ 和 $b_1$ 得到的结论后，就最终得到了比较 $a$ 和 $b$ 的结论。

在这个过程中，对每个关键字边比较边重排元素的顺序，就得到了 LSD 基数排序。

## 伪代码

1 **Input.** An array $A$ consisting of $n$ elements, where each element has $k$ keys.
2 **Output.** Array $A$ will be sorted in nondecreasing order stably.
3 **Method.**
4 **for** $i \leftarrow k$ **down to** 1
5 　　sort $A$ into nondecreasing order by the $i$-th key stably

## 参考代码

下面是使用 LSD 基数排序实现的对 k - 关键字元素的排序。

```
const int N = 100010;
const int W = 100010;
const int K = 100;

int n, w[K], k, cnt[W];

struct Element {
  int key[K];

  bool operator<(const Element& y) const {
    // 两个元素的比较流程
    for (int i = 1; i <= k; ++i) {
      if (key[i] == y.key[i]) continue;
      return key[i] < y.key[i];
    }
    return false;
  }
} a[N], b[N];

void counting_sort(int p) {
  memset(cnt, 0, sizeof(cnt));
  for (int i = 1; i <= n; ++i) ++cnt[a[i].key[p]];
  for (int i = 1; i <= w[p]; ++i) cnt[i] += cnt[i - 1];
  // 为保证排序的稳定性，此处循环 i 应从 n 到 1
  // 即当两元素关键字的值相同时，原先排在后面的元素在排序后仍应排在后面
  for (int i = n; i >= 1; --i) b[cnt[a[i].key[p]]--] = a[i];
  memcpy(a, b, sizeof(a));
}

void radix_sort() {
  for (int i = k; i >= 1; --i) {
    // 借助计数排序完成对关键字的排序
```

```
    counting_sort(i);
  }
}
```

实际上并非必须从后往前枚举才是稳定排序，只需对 `cnt` 数组进行等价于 `std::exclusive_scan` 的操作即可。

**例题 洛谷 P1177【 模板 】 快速排序**

给出 $n$ 个正整数，从小到大输出。

```cpp
#include <algorithm>
#include <iostream>
#include <utility>

void radix_sort(int n, int a[]) {
  int *b = new int[n];  // 临时空间
  int *cnt = new int[1 << 8];
  int mask = (1 << 8) - 1;
  int *x = a, *y = b;
  for (int i = 0; i < 32; i += 8) {
    for (int j = 0; j != (1 << 8); ++j) cnt[j] = 0;
    for (int j = 0; j != n; ++j) ++cnt[x[j] >> i & mask];
    for (int sum = 0, j = 0; j != (1 << 8); ++j) {
      // 等价于 std::exclusive_scan(cnt, cnt + (1 << 8), cnt, 0);
      sum += cnt[j], cnt[j] = sum - cnt[j];
    }
    for (int j = 0; j != n; ++j) y[cnt[x[j] >> i & mask]++] = x[j];
    std::swap(x, y);
  }
  delete[] cnt;
  delete[] b;
}

int main() {
  std::ios::sync_with_stdio(false);
  std::cin.tie(0);
  int n;
  std::cin >> n;
  int *a = new int[n];
  for (int i = 0; i < n; ++i) std::cin >> a[i];
  radix_sort(n, a);
  for (int i = 0; i < n; ++i) std::cout << a[i] << ' ';
  delete[] a;
  return 0;
}
```

## 性质

### 稳定性

如果对内层关键字的排序是稳定的，则 MSD 基数排序和 LSD 基数排序都是稳定的排序算法。

### 时间复杂度

通常而言，基数排序比基于比较的排序算法（比如快速排序）要快。但由于需要额外的内存空间，因此当内存空间稀缺时，原地置换算法（比如快速排序）或许是个更好的选择。[1]

一般来说，如果每个关键字的值域都不大，就可以使用 计数排序 作为内层排序，此时的复杂度为 $O(kn+\sum_{i=1}^{k} w_i)$，其中 $w_i$ 为第 $i$ 关键字的值域大小。如果关键字值域很大，就可以直接使用基于比较的 $O(nk\log n)$ 排序而无需使用基数排序了。

### 空间复杂度

MSD 基数排序和 LSD 基数排序的空间复杂度都为 $O(k+n)$。

## 参考资料与注释

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein.*Introduction to Algorithms*(3rd ed.). MIT Press and McGraw-Hill, 2009. ISBN 978-0-262-03384-8. ”8.3 Radix sort”, pp. 199.

[2] 空终止字节字符串

[3] 《算法导论（第三版）》第 8.3-3 题的解法

# 1.7.7 快速排序

本页面将简要介绍快速排序。

## 定义

快速排序（英语：Quicksort），又称分区交换排序（英语：partition-exchange sort），简称「快排」，是一种被广泛运用的排序算法。

## 基本原理与实现

### 过程

快速排序的工作原理是通过 分治 的方式来将一个数组排序。

快速排序分为三个过程：

1. 将数列划分为两部分（要求保证相对大小关系）；
2. 递归到两个子序列中分别进行快速排序；
3. 不用合并，因为此时数列已经完全有序。

和归并排序不同，第一步并不是直接分成前后两个序列，而是在分的过程中要保证相对大小关系。具体来说，第一步要是要把数列分成两个部分，然后保证前一个子数列中的数都小于后一个子数列中的数。为了保证平均时间复杂度，一般是随机选择一个数 $m$ 来当做两个子数列的分界。

之后，维护一前一后两个指针 $p$ 和 $q$，依次考虑当前的数是否放在了应该放的位置（前还是后）。如果当前的数没放对，比如说如果后面的指针 $q$ 遇到了一个比 $m$ 小的数，那么可以交换 $p$ 和 $q$ 位置上的数，再把 $p$ 向后移一位。当前的数的位置全放对后，再移动指针继续处理，直到两个指针相遇。

其实，快速排序没有指定应如何具体实现第一步，不论是选择 $m$ 的过程还是划分的过程，都有不止一种实现方法。

第三步中的序列已经分别有序且第一个序列中的数都小于第二个数，所以直接拼接起来就好了。

=== ”C++[2-1]”

```cpp
struct Range {
  int start, end;
```

```
  Range(int s = 0, int e = 0) { start = s, end = e; }
};

template <typename T>
void quick_sort(T arr[], const int len) {
  if (len <= 0) return;
  Range r[len];
  int p = 0;
  r[p++] = Range(0, len - 1);
  while (p) {
    Range range = r[--p];
    if (range.start >= range.end) continue;
    T mid = arr[range.end];
    int left = range.start, right = range.end - 1;
    while (left < right) {
      while (arr[left] < mid && left < right) left++;
      while (arr[right] >= mid && left < right) right--;
      std::swap(arr[left], arr[right]);
    }
    if (arr[left] >= arr[range.end])
      std::swap(arr[left], arr[range.end]);
    else
      left++;
    r[p++] = Range(range.start, left - 1);
    r[p++] = Range(left + 1, range.end);
  }
}
```

=== "Python[2-2]"

```python
def quick_sort(alist, first, last):
    if first >= last:
        return
    mid_value = alist[first]
    low = first
    high = last
    while low < high:
        while low < high and alist[high] >= mid_value:
            high -= 1
        alist[low] = alist[high]
        while low < high and alist[low] < mid_value:
            low += 1
        alist[high] = alist[low]
    alist[low] = mid_value
    quick_sort(alist, first, low - 1)
    quick_sort(alist, low + 1, last)
```

# 性质

## 稳定性

快速排序是一种不稳定的排序算法。

## 时间复杂度

快速排序的最优时间复杂度和平均时间复杂度为 $O(n\log n)$，最坏时间复杂度为 $O(n^2)$。

对于最优情况，每一次选择的分界值都是序列的中位数，此时算法时间复杂度满足的递推式为 $T(n)=2T(\frac{n}{2})+\Theta(n)$，由主定理，$T(n)=\Theta(n\log n)$。

对于最坏情况，每一次选择的分界值都是序列的最值，此时算法时间复杂度满足的递推式为 $T(n)=T(n-1)+\Theta(n)$，累加可得 $T(n)=\Theta(n^2)$。

对于平均情况，每一次选择的分界值可以看作是等概率随机的。

**证明**

下面我们来证明这种情况下算法的时间复杂度是 $O(n\log n)$。

**引理 1：** 当对 $n$ 个元素的数组进行快速排序时，假设在划分元素时总共的比较次数为 $X$，则快速排序的时间复杂度是 $O(n+X)$。

由于在每次划分元素的过程中，都会选择一个元素作为分界，所以划分元素的过程至多发生 $n$ 次。又由于划分元素的过程中比较的次数和其他基础操作的次数在一个数量级，所以总时间复杂度是 $O(n+X)$ 的。

设 $a_i$ 为原数组中第 $i$ 小的数，定义 $A_{i,j}$ 为 $\{a_i,a_{i+1},\dots,a_j\}$，$X_{i,j}$ 是一个取值为 0 或者 1 的离散随机变量表示在排序过程中 $a_i$ 是否和 $a_j$ 发生比较。

显然每次选取的分界值是不同的，而元素只会和分界值比较，所以总比较次数

$$X=\sum_{i=1}^{n-1}\sum_{j=i+1}^{n}X_{i,j}$$

由期望的线性性，

$$\begin{aligned}E[X]&=E\left[\sum_{i=1}^{n-1}\sum_{j=i+1}^{n}X_{i,j}\right]\\&=\sum_{i=1}^{n-1}\sum_{j=i+1}^{n}E[X_{i,j}]\\&=\sum_{i=1}^{n-1}\sum_{j=i+1}^{n}P(a_i\text{ 和 }a_j\text{ 比较})\end{aligned}$$

**引理 2：** $a_i$ 和 $a_j$ 比较的充要条件是 $a_i$ 或 $a_j$ 是集合 $A_{i,j}$ 中第一个被选中的分界值。

先证必要性，即若 $a_i$ 和 $a_j$ 都不是集合 $A_{i,j}$ 中第一个被选中的分界值，则 $a_i$ 不和 $a_j$ 比较。

若 $a_i$ 和 $a_j$ 都不是集合 $A_{i,j}$ 中第一个被选中的分界值，则一定存在一个 $x$ 满足 $i<x<j$，使得 $a_x$ 是 $A_{i,j}$ 中第一个被选中的分界值。在以 $a_x$ 为分界值的划分中，$a_i$ 和 $a_j$ 被划分到数组的两个不同的子序列中，所以之后 $a_i$ 和 $a_j$ 一定不会比较。又因为元素只和分界值比较，所以 $a_i$ 和 $a_j$ 在此次划分前和划分中没有比较。所以 $a_i$ 不和 $a_j$ 比较。

再证充分性，即若 $a_i$ 或 $a_j$ 是集合 $A_{i,j}$ 中第一个被选中的分界值，则 $a_i$ 和 $a_j$ 比较。

不失一般地，假设 $a_i$ 是集合 $A_{i,j}$ 中第一个被选中的分界值。由于 $A_{i,j}$ 中没有其他数选为分界值，所以 $A_{i,j}$ 中的元素都在数组的同一子序列中。在以 $a_i$ 为分界值的划分中，$a_i$ 和当前子序列中所有元素都进行了比较，所以 $a_i$ 和 $a_j$ 进行了比较。

考虑计算 $P(a_i\text{ 和 }a_j\text{ 比较})$。在 $A_{i,j}$ 中某个元素被选为分界值之前，$A_{i,j}$ 中的元素都在数组的同一子序列中。所以 $A_{i,j}$ 中每个元素都会被等可能地第一个被选为分界值。由于 $A_{i,j}$ 中有 $j-i+1$ 个元素，由引理 2，

$$P(a_i\text{和}a_j\text{比较})=P(a_i\text{或}a_j\text{是集合}A_{i,j}\text{中第一个被选中的分界值})=\frac{2}{j-i+1}$$

所以

$$\begin{aligned} E[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} P(a_i \text{ 和 } a_j \text{ 比较}) \\ &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\ &= \sum_{i=1}^{n-1} \sum_{k=2}^{n-i+1} \frac{2}{k} \\ &= \sum_{i=1}^{n-1} O(\log n) \\ &= O(n \log n) \end{aligned}$$

由此，快速排序的期望时间复杂度为 $O(n \log n)$。

在实践中，几乎不可能达到最坏情况，而快速排序的内存访问遵循局部性原理，所以多数情况下快速排序的表现大幅优于堆排序等其他复杂度为 $O(n \log n)$ 的排序算法。[1]

# 优化

## 朴素优化思想

如果仅按照上文所述的基本思想来实现快速排序（或者是直接照抄模板）的话，那大概率是通不过 P1177【模板】快速排序[5] 这道模板的。因为有毒瘤数据能够把朴素的快速排序卡成 $O(n^2)$。

所以，我们需要对朴素快速排序思想加以优化。较为常见的优化思路有以下三种[3]。

- 通过三**数取中（即选取第一个、最后一个以及中间的元素中的中位数）**的方法来选择两个子序列的分界元素（即比较基准）。这样可以避免极端数据（如升序序列或降序序列）带来的退化；
- 当序列较短时，使用**插入排序**的效率更高；
- 每趟排序后，**将与分界元素相等的元素聚集在分界元素周围**，这样可以避免极端数据（如序列中大部分元素都相等）带来的退化。

下面列举了几种较为成熟的快速排序优化方式。

## 三路快速排序

### 定义

三路快速排序（英语：3-way Radix Quicksort）是快速排序和 基数排序 的混合。它的算法思想基于 荷兰国旗问题[6] 的解法。

### 过程

与原始的快速排序不同，三路快速排序在随机选取分界点 $m$ 后，将待排数列划分为三个部分：小于 $m$、等于 $m$ 以及大于 $m$。这样做即实现了将与分界元素相等的元素聚集在分界元素周围这一效果。

### 性质

三路快速排序在处理含有多个重复值的数组时，效率远高于原始快速排序。其最佳时间复杂度为 $O(n)$。

### 实现

三路快速排序实现起来非常简单，下面给出了一种三路快排的 C++ 实现。

=== "C++"

```cpp
// 模板的 T 参数表示元素的类型，此类型需要定义小于 (<) 运算
```

```
template <typename T>
// arr 为需要被排序的数组，len 为数组长度
void quick_sort(T arr[], const int len) {
  if (len <= 1) return;
  // 随机选择基准 (pivot)
  const T pivot = arr[rand() % len];
  // i: 当前操作的元素下标
  // arr[0, j): 存储小于 pivot 的元素
  // arr[k, len): 存储大于 pivot 的元素
  int i = 0, j = 0, k = len;
  // 完成一趟三路快排，将序列分为：
  // 小于 pivot 的元素 | 等于 pivot 的元素 | 大于 pivot 的元素
  while (i < k) {
    if (arr[i] < pivot)
      swap(arr[i++], arr[j++]);
    else if (pivot < arr[i])
      swap(arr[i], arr[--k]);
    else
      i++;
  }
  // 递归完成对于两个子序列的快速排序
  quick_sort(arr, j);
  quick_sort(arr + k, len - k);
}
```

=== "Python[2-3]"

```python
def quick_sort(arr, l, r):
    if l >= r:
        return
    random_index = random.randint(l, r)
    pivot = arr[random_index]
    arr[l], arr[random_index] = arr[random_index], arr[l]
    i = l + 1
    j = l
    k = r + 1
    while i < k:
        if arr[i] < pivot:
            arr[i], arr[j + 1] = arr[j + 1], arr[i]
            j += 1
            i += 1
        elif arr[i] > pivot:
            arr[i], arr[k - 1] = arr[k - 1], arr[i]
            k -= 1
        else:
            i += 1
    arr[l], arr[j] = arr[j], arr[l]
    quick_sort(arr, l, j - 1)
    quick_sort(arr, k, r)
```

### 内省排序

#### 定义

内省排序（英语：Introsort 或 Introspective sort）[4] 是快速排序和 堆排序 的结合，由 David Musser 于 1997 年发明。内省排序其实是对快速排序的一种优化，保证了最差时间复杂度为 $O(n \log n)$。

#### 性质

内省排序将快速排序的最大递归深度限制为 $\lfloor \log_2 n \rfloor$，超过限制时就转换为堆排序。这样既保留了快速排序内存访问的局部性，又可以防止快速排序在某些情况下性能退化为 $O(n^2)$。

#### 实现

从 2000 年 6 月起，SGI C++ STL 的 `stl_algo.h` 中 `sort()` 函数的实现采用了内省排序算法。

## 线性找第 k 大的数

在下面的代码示例中，第 $k$ 大的数被定义为序列排成升序时，第 $k$ 个位置上的数（编号从 0 开始）。

找第 $k$ 大的数（K-th order statistic），最简单的方法是先排序，然后直接找到第 $k$ 大的位置的元素。这样做的时间复杂度是 $O(n \log n)$，对于这个问题来说很不划算。

我们可以借助快速排序的思想解决这个问题。考虑快速排序的划分过程，在快速排序的「划分」结束后，数列 $A_p \cdots A_r$ 被分成了 $A_p \cdots A_q$ 和 $A_{q+1} \cdots A_r$，此时可以按照左边元素的个数（$q-p+1$）和 $k$ 的大小关系来判断是只在左边还是只在右边递归地求解。

和快速排序一样，该方法的时间复杂度依赖于每次划分时选择的分界值。如果采用随机选取分界值的方式，可以证明在期望意义下，程序的时间复杂度为 $O(n)$。

### 实现（C++）

```cpp
// 模板的 T 参数表示元素的类型，此类型需要定义小于 (<) 运算
template <typename T>
// arr 为查找范围数组，rk 为需要查找的排名 (从 0 开始)，len 为数组长度
T find_kth_element(T arr[], int rk, const int len) {
  if (len <= 1) return arr[0];
  // 随机选择基准 (pivot)
  const T pivot = arr[rand() % len];
  // i: 当前操作的元素
  // j: 第一个等于 pivot 的元素
  // k: 第一个大于 pivot 的元素
  int i = 0, j = 0, k = len;
  // 完成一趟三路快排，将序列分为：
  // 小于 pivot 的元素 | 等于 pivot 的元素 | 大于 pivot 的元素
  while (i < k) {
    if (arr[i] < pivot)
      swap(arr[i++], arr[j++]);
    else if (pivot < arr[i])
      swap(arr[i], arr[--k]);
    else
      i++;
  }
  // 根据要找的排名与两条分界线的位置，去不同的区间递归查找第 k 大的数
  // 如果小于 pivot 的元素个数比 k 多，则第 k 大的元素一定是一个小于 pivot 的元素
  if (rk < j) return find_kth_element(arr, rk, j);
  // 否则，如果小于 pivot 和等于 pivot 的元素加起来也没有 k 多，
  // 则第 k 大的元素一定是一个大于 pivot 的元素
```

```
  else if (rk >= k)
    return find_kth_element(arr + k, rk - k, len - k);
  // 否则，pivot 就是第 k 大的元素
  return pivot;
}
```

### 改进：中位数中的中位数

中位数中的中位数（英文：Median of medians），提供了一种确定性的选择划分过程中分界值的方法，从而能够让找第 $k$ 大的数算法在最坏情况下也能实现线性时间复杂度。

该算法的流程如下：

1. 将整个序列划分为 $\left\lfloor \frac{n}{5} \right\rfloor$ 组，每组元素数不超过 5 个；
2. 寻找每组元素的中位数（因为元素个数较少，可以直接使用 插入排序 等算法）。
3. 找出这 $\left\lfloor \frac{n}{5} \right\rfloor$ 组元素中位数中的中位数。将该元素作为前述算法中每次划分时的分界值即可。

#### 时间复杂度证明

下面将证明，该算法在最坏情况下的时间复杂度为 $O(n)$。设 $T(n)$ 为问题规模为 $n$ 时，解决问题需要的计算量。

先分析前两步——划分与寻找中位数。由于划分后每组内的元素数量非常少，可以认为寻找一组元素的中位数的时间复杂度为 $O(1)$。因此找出所有 $\left\lfloor \frac{n}{5} \right\rfloor$ 组元素中位数的时间复杂度为 $O(n)$。

接下来分析第三步——递归过程。这一步进行了两次递归调用：第一次是寻找各组中位数中的中位数，需要的开销显然为 $T(\frac{n}{5})$，第二次是进入分界值的左侧部分或右侧部分。根据我们选取的划分元素，有 $\frac{1}{2} \times \left\lfloor \frac{n}{5} \right\rfloor = \left\lfloor \frac{n}{10} \right\rfloor$ 组元素的中位数小于分界值，这几组元素中，比中位数还小的元素也一定比分界值要小，从而整个序列中小于分界值的元素至少有 $3 \times \left\lfloor \frac{n}{10} \right\rfloor = \left\lfloor \frac{3n}{10} \right\rfloor$ 个。同理，整个序列中大于分界值的元素也至少有 $\left\lfloor \frac{3n}{10} \right\rfloor$ 个。因此，分界值的左边或右边至多有 $\frac{7n}{10}$ 个元素，这次递归的时间开销的上界为 $T(\frac{7n}{10}$。

综上，我们可以列出这样的不等式：

$$T(n) \leq T(\frac{n}{5}) + T(\frac{7n}{10}) + O(n)$$

假设 $T(n) = O(n)$ 在问题规模足够小时成立。根据定义，此时有 $T(n) \leq cn$，其中 $c$ 为一正常数。将不等式右边的所有 $T(n)$ 进行代换：

$$\begin{aligned} T(n) &\leq T(\frac{n}{5}) + T(\frac{7n}{10}) + O(n) \\ &\leq \frac{cn}{5} + \frac{7cn}{10} + O(n) \\ &\leq \frac{9cn}{10} + O(n) \\ &= O(n) \end{aligned}$$

到这里我们就证明了，该算法在最坏情况下也具有 $O(n)$ 的时间复杂度。

## 参考资料与注释


[1] C++ 性能榨汁机之局部性原理 - I'm Root lee !

[2] 算法实现 / 排序 / 快速排序 - 维基教科书， 自由的教学读本 [2-1] [2-2] [2-3]

[3] 三种快速排序以及快速排序的优化

[4] introsort


[5] P1177【模板】快速排序

[6] 荷兰国旗问题

# 1.7.8 归并排序

## 定义

归并排序（merge sort[1]）是高效的基于比较的稳定排序算法。

## 性质

归并排序基于分治思想将数组分段排序后合并，时间复杂度在最优、最坏与平均情况下均为 $\Theta(n \log n)$，空间复杂度为 $\Theta(n)$。

归并排序可以只使用 $\Theta(1)$ 的辅助空间，但为便捷通常使用与原数组等长的辅助数组。

## 过程

### 合并

归并排序最核心的部分是合并（merge）过程：将两个有序的数组 `a[i]` 和 `b[j]` 合并为一个有序数组 `c[k]`。

从左往右枚举 `a[i]` 和 `b[j]`，找出最小的值并放入数组 `c[k]`；重复上述过程直到 `a[i]` 和 `b[j]` 有一个为空时，将另一个数组剩下的元素放入 `c[k]`。

为保证排序的稳定性，前段首元素小于或等于后段首元素时（`a[i] <= b[j]`）而非小于时（`a[i] < b[j]`）就要作为最小值放入 `c[k]`。

#### 实现

=== "C/C++"

```
=== " 数组实现"

    ```cpp
    void merge(const int *a, size_t aLen, const int *b, size_t bLen, int *c) {
      size_t i = 0, j = 0, k = 0;
      while (i < aLen && j < bLen) {
        if (b[j] < a[i]) {  // <!> 先判断 b[j] < a[i], 保证稳定性
          c[k] = b[j];
          ++j;
        } else {
          c[k] = a[i];
          ++i;
        }
        ++k;
      }
      // 此时一个数组已空, 另一个数组非空, 将非空的数组并入 c 中
      for (; i < aLen; ++i, ++k) c[k] = a[i];
      for (; j < bLen; ++j, ++k) c[k] = b[j];
    }
    ```

=== " 指针实现"
```

```
    ```cpp
    void merge(const int *aBegin, const int *aEnd, const int *bBegin, const int
*bEnd, int *c) {
      while (aBegin != aEnd && bBegin != bEnd) {
        if (*bBegin < *aBegin) {
          *c = *bBegin;
          ++bBegin;
        } else {
          *c = *aBegin;
          ++aBegin;
        }
        ++c;
      }
      for (; aBegin != aEnd; ++aBegin, ++c) *c = *aBegin;
      for (; bBegin != bEnd; ++bBegin, ++c) *c = *bBegin;
    }
    ```

也可使用 `<algorithm>` 库的 `merge` 函数，用法与上述指针式写法的相同。
```

=== "Python"

```
```python
def merge(a, b):
    i, j = 0, 0
    c = []
    while(i < len(a) and j < len(b)):
        # <!> 先判断 b[j] < a[i]，保证稳定性
        if(b[j] < a[i]):
            c.append(b[j])
            j += 1
        else:
            c.append(a[i])
            i += 1
    # 此时一个数组已空，另一个数组非空，将非空的数组并入 c 中
    c.extend(a[i:])
    c.extend(b[j:])
    return c
```
```

## 分治法实现归并排序

1. 当数组长度为 1 时，该数组就已经是有序的，不用再分解。
2. 当数组长度大于 1 时，该数组很可能不是有序的。此时将该数组分为两段，再分别检查两个数组是否有序（用第 1 条）。如果有序，则将它们合并为一个有序数组；否则对不有序的数组重复第 2 条，再合并。

用数学归纳法可以证明该流程可以将一个数组转变为有序数组。

为保证排序的复杂度，通常将数组分为尽量等长的两段（$mid = \left\lfloor \frac{l+r}{2} \right\rfloor$）。

### 实现

注意下面的代码所表示的区间分别是 $[l, r)$，$[l, mid)$，$[mid, r)$。

=== "C/C++"

```cpp
void merge_sort(int *a, int l, int r) {
  if (r - l <= 1) return;
  // 分解
  int mid = l + ((r - l) >> 1);
  merge_sort(a, l, mid), merge_sort(a, mid, r);
  // 合并
  int tmp[1024] = {};  // 请结合实际情况设置 tmp 数组的长度（与 a 相同），或使用
                       // vector；先将合并的结果放在 tmp 里，再返回到数组 a
  merge(a + l, a + mid, a + mid, a + r, tmp + l);  // pointer-style merge
  for (int i = l; i < r; ++i) a[i] = tmp[i];
}
```

=== "Python"

```python
def merge_sort(a, ll, rr):
    if rr - ll <= 1:
        return
    # 分解
    mid = (rr + ll) // 2
    merge_sort(a, ll, mid)
    merge_sort(a, mid, rr)
    # 合并
    a[ll:rr] = merge(a[ll:mid], a[mid:rr])
```

### 倍增法实现归并排序

已知当数组长度为 1 时，该数组就已经是有序的。

将数组全部切成长度为 1 的段。

从左往右依次合并两个长度为 1 的有序段，得到一系列长度 $\leq 2$ 的有序段；

从左往右依次合并两个长度 $\leq 2$ 的有序段，得到一系列长度 $\leq 4$ 的有序段；

从左往右依次合并两个长度 $\leq 4$ 的有序段，得到一系列长度 $\leq 8$ 的有序段；

……

重复上述过程直至数组只剩一个有序段，该段就是排好序的原数组。

**为什么是 $\leq n$ 而不是 $= n$**

数组的长度很可能不是 $2^x$，此时在最后就可能出现长度不完整的段，可能出现最后一个段是独立的情况。

#### 实现

=== "C/C++"

```cpp
void merge_sort(int *a, size_t n) {
  int tmp[1024] = {};  // 请结合实际情况设置 tmp 数组的长度（与 a 相同），或使用
                       // vector；先将合并的结果放在 tmp 里，再返回到数组 a
  for (size_t seg = 1; seg < n; seg <<= 1) {
    for (size_t left1 = 0; left1 < n - seg;
         left1 += seg + seg) {  // n - seg: 如果最后只有一个段就不用合并
      size_t right1 = left1 + seg;
```

```
      size_t left2 = right1;
      size_t right2 = std::min(left2 + seg, n);  // <!> 注意最后一个段的边界
      merge(a + left1, a + right1, a + left2, a + right2,
            tmp + left1);  // pointer-style merge
      for (size_t i = left1; i < right2; ++i) a[i] = tmp[i];
    }
  }
}
```

=== "Python"

```python
def merge_sort(a):
    seg = 1
    while seg < len(a):
        for l1 in range(0, len(a) - seg, seg + seg):
            r1 = l1 + seg
            l2 = r1
            r2 = l2 + seg
            a[l1:r2] = merge(a[l1:r1], a[l2:r2])
    seg <<= 1
```

## 逆序对

逆序对是 $i < j$ 且 $a_i > a_j$ 的有序数对 $(i, j)$。

排序后的数组无逆序对，归并排序的合并操作中，每次后段首元素被作为当前最小值取出时，前段剩余元素个数之和即是合并操作减少的逆序对数量；故归并排序计算逆序对数量的额外时间复杂度为 $\Theta(n \log n)$，对于 C/C++ 代码将 `merge` 过程的 `if(b[j] < a[i])` 部分加上 `cnt += aLen - i` 或 `cnt += aEnd - aBegin` 即可，对于 Python 代码将 `merge` 过程的 `if(b[j] < a[i]):` 部分加上 `cnt += len(a) - i` 即可。

此外，逆序对计数即是将元素依次加入数组时统计当前大于其的元素数量，将数组离散化后即是区间求和问题，使用树状数组或线段树解决的时间复杂度为 $O(n \log n)$ 且空间复杂度为 $\Theta(n)$。

## 外部链接

- Merge Sort - GeeksforGeeks[2]
- 归并排序 - 维基百科，自由的百科全书[3]
- 逆序对 - 维基百科，自由的百科全书[4]

## 参考资料与注释

[1] merge sort

[2] Merge Sort - GeeksforGeeks

[3] 归并排序 - 维基百科，自由的百科全书

[4] 逆序对 - 维基百科，自由的百科全书

# 1.7.9 堆排序

本页面将简要介绍堆排序。

## 定义

堆排序（英语：Heapsort）是指利用二叉堆这种数据结构所设计的一种排序算法。堆排序的适用数据结构为数组。

## 过程

堆排序的本质是建立在堆上的选择排序。

### 排序

首先建立大顶堆，然后将堆顶的元素取出，作为最大值，与数组尾部的元素交换，并维持残余堆的性质；

之后将堆顶的元素取出，作为次大值，与数组倒数第二位元素交换，并维持残余堆的性质；

以此类推，在第 $n-1$ 次操作后，整个数组就完成了排序。

### 在数组上建立二叉堆

从根节点开始，依次将每一层的节点排列在数组里。

于是有数组中下标为 `i` 的节点，对应的父结点、左子结点和右子结点如下：

```
iParent(i) = (i - 1) / 2;
iLeftChild(i) = 2 * i + 1;
iRightChild(i) = 2 * i + 2;
```

## 性质

### 稳定性

同选择排序一样，由于其中交换位置的操作，所以是不稳定的排序算法。

### 时间复杂度

堆排序的最优时间复杂度、平均时间复杂度、最坏时间复杂度均为 $O(n\log n)$。

### 空间复杂度

由于可以在输入数组上建立堆，所以这是一个原地算法。

## 实现

=== "C++"

```
```cpp
void sift_down(int arr[], int start, int end) {
  // 计算父结点和子结点的下标
  int parent = start;
  int child = parent * 2 + 1;
  while (child <= end) {  // 子结点下标在范围内才做比较
    // 先比较两个子结点大小，选择最大的
    if (child + 1 <= end && arr[child] < arr[child + 1]) child++;
    // 如果父结点比子结点大，代表调整完毕，直接跳出函数
    if (arr[parent] >= arr[child])
```
```

```
      return;
    else {  // 否则交换父子内容，子结点再和孙结点比较
      swap(arr[parent], arr[child]);
      parent = child;
      child = parent * 2 + 1;
    }
  }
}

void heap_sort(int arr[], int len) {
  // 从最后一个节点的父节点开始 sift down 以完成堆化 (heapify)
  for (int i = (len - 1 - 1) / 2; i >= 0; i--) sift_down(arr, i, len - 1);
  // 先将第一个元素和已经排好的元素前一位做交换，再重新调整（刚调整的元素之前的元素
), 直到排序完毕
  for (int i = len - 1; i > 0; i--) {
    swap(arr[0], arr[i]);
    sift_down(arr, 0, i - 1);
  }
}
```

=== "Python"

```python
def sift_down(arr, start, end):
    # 计算父结点和子结点的下标
    parent = int(start)
    child = int(parent * 2 + 1)
    while child <= end: # 子结点下标在范围内才做比较
        # 先比较两个子结点大小，选择最大的
        if child + 1 <= end and arr[child] < arr[child + 1]:
            child += 1
        # 如果父结点比子结点大，代表调整完毕，直接跳出函数
        if arr[parent] >= arr[child]:
            return
        else: # 否则交换父子内容，子结点再和孙结点比较
            arr[parent], arr[child] = arr[child], arr[parent]
            parent = child
            child = int(parent * 2 + 1)

def heap_sort(arr, len):
  # 从最后一个节点的父节点开始 sift down 以完成堆化 (heapify)
    i = (len - 1 - 1) / 2
    while(i >= 0):
        sift_down(arr, i, len - 1)
        i -= 1
  # 先将第一个元素和已经排好的元素前一位做交换，再重新调整（刚调整的元素之前的元素
), 直到排序完毕
    i = len - 1
    while(i > 0):
        arr[0], arr[i] = arr[i], arr[0]
        sift_down(arr, 0, i - 1)
        i -= 1
```

## 外部链接

- 堆排序 - 维基百科，自由的百科全书[1]

## 参考资料与注释

[1] 堆排序 - 维基百科，自由的百科全书

# 1.7.10 桶排序

本页面将简要介绍桶排序。

## 定义

桶排序（英文：Bucket sort）是排序算法的一种，适用于待排序数据值域较大但分布比较均匀的情况。

## 过程

桶排序按下列步骤进行：

1. 设置一个定量的数组当作空桶；
2. 遍历序列，并将元素一个个放到对应的桶中；
3. 对每个不是空的桶进行排序；
4. 从不是空的桶里把元素再放回原来的序列中。

## 性质

### 稳定性

如果使用稳定的内层排序，并且将元素插入桶中时不改变元素间的相对顺序，那么桶排序就是一种稳定的排序算法。

由于每块元素不多，一般使用插入排序。此时桶排序是一种稳定的排序算法。

### 时间复杂度

桶排序的平均时间复杂度为 $O(n+n^2/k+k)$（将值域平均分成 $n$ 块 + 排序 + 重新合并元素），当 $k \approx n$ 时为 $O(n)$。[1]

桶排序的最坏时间复杂度为 $O(n^2)$。

## 实现

=== "C++"

```cpp
const int N = 100010;

int n, w, a[N];
vector<int> bucket[N];

void insertion_sort(vector<int>& A) {
  for (int i = 1; i < A.size(); ++i) {
    int key = A[i];
```

```
    int j = i - 1;
    while (j >= 0 && A[j] > key) {
      A[j + 1] = A[j];
      --j;
    }
    A[j + 1] = key;
  }
}

void bucket_sort() {
  int bucket_size = w / n + 1;
  for (int i = 0; i < n; ++i) {
    bucket[i].clear();
  }
  for (int i = 1; i <= n; ++i) {
    bucket[a[i] / bucket_size].push_back(a[i]);
  }
  int p = 0;
  for (int i = 0; i < n; ++i) {
    insertion_sort(bucket[i]);
    for (int j = 0; j < bucket[i].size(); ++j) {
      a[++p] = bucket[i][j];
    }
  }
}
```

=== ”Python”

```python
N = 100010
w = n = 0
a = [0] * N
bucket = [[] for i in range(N)]

def insertion_sort(A):
    for i in range(1, len(A)):
        key = A[i]
        j = i - 1
        while j >= 0 and A[j] > key:
            A[j + 1] = A[j]
            j -= 1
        A[j + 1] = key

def bucket_sort():
    bucket_size = int(w / n + 1)
    for i in range(0, n):
        bucket[i].clear()
    for i in range(1, n + 1):
        bucket[int(a[i] / bucket_size)].append(a[i])
    p = 0
    for i in range(0, n):
        insertion_sort(bucket[i])
        for j in range(0, len(bucket[i])):
            a[p] = bucket[i][j]
```

```
            p += 1
```

## 参考资料与注释

[1] （英文）Bucket sort - Wikipedia

# 1.7.11 希尔排序

本页面将简要介绍希尔排序。

## 定义

希尔排序（英语：Shell sort），也称为缩小增量排序法，是 插入排序 的一种改进版本。希尔排序以它的发明者希尔（英语：Donald Shell）命名。

## 过程

排序对不相邻的记录进行比较和移动：

1. 将待排序序列分为若干子序列（每个子序列的元素在原始数组中间距相同）；
2. 对这些子序列进行插入排序；
3. 减小每个子序列中元素之间的间距，重复上述过程直至间距减少为 1。

## 性质

### 稳定性

希尔排序是一种不稳定的排序算法。

### 时间复杂度

希尔排序的最优时间复杂度为 $O(n)$。

希尔排序的平均时间复杂度和最坏时间复杂度与间距序列的选取。设间距序列为 $H$，下面给出 $H$ 的两种经典选取方式，这两种选取方式均使得排序算法的复杂度降为 $o(n^2)$ 级别。

**命题** 1：若间距序列为 $H = \{2^k - 1 \mid k = 1, 2, \dots, \lfloor \log_2 n \rfloor\}$（从大到小），则希尔排序算法的时间复杂度为 $O(n^{3/2})$。

**命题** 2：若间距序列为 $H = \{k = 2^p \cdot 3^q \mid p, q \in \mathbb{N}, k \leq n\}$（从大到小），则希尔排序算法的时间复杂度为 $O(n \log^2 n)$。

为证明这两个命题，我们先给出一个重要的定理并证明它，这个定理反应了希尔排序的最主要特征。

**定理** 1：只要程序执行了一次 InsertionSort$(h)$，不管之后怎样调用 InsertionSort 函数，$A$ 数组怎样变换，下列性质均会被一直保持：

$$
\begin{gathered}
A_1, A_{1+h}, A_{1+2h}, \dots \\
A_2, A_{2+h}, A_{2+2h}, \dots \\
\vdots \\
A_h, A_{h+h}, A_{h+2h}, \dots
\end{gathered}
$$

证明：

我们先证明一个引理：

**引理** 1：对于整数 $n,m$、正整数 $l$ 与两个数组 $X(x_1,x_2,\dots,x_{n+l}),Y(y_1,y_2,\dots,y_{m+l})$，满足如下要求：

$$y_1 \le x_{n+1}, y_2 \le x_{n+2}, \dots, y_l \le x_{n+l}$$

则我们将两个数组分别升序排序后，上述要求依然成立。

**证明**：

设数组 $X$ 排序完为数组 $X'(x'_1,\dots,x'_{n+l})$，数组 $Y$ 排序完为数组 $Y'(y'_1,\dots,y'_{m+l})$。

对于任何 $1 \le i \le l$，$x'_{n+i}$ 小等于数组 $X'$ 中的 $l-i$ 个元素，也小等于数组 $X$ 中的 $l-i$ 个元素（这是因为 $X$ 与 $X'$ 的元素可重集合是相同的）。

那么在可重集合 $\{x_{n+1},\dots,x_{n+l}\} \subset X$ 中，大等于 $x'_{n+i}$ 的元素个数不超过 $l-i$ 个。

进而小于 $x'_{n+i}$ 的元素个数至少有 $i$ 个，取出其中的 $i$ 个，设它们为 $x_{n+k_1},x_{n+k_2},\dots,x_{n+k_i}$。于是有：

$$y_{k_1} \le x_{n+k_1} \le x'_{n+i}, y_{k_2} \le x_{n+k_2} \le x'_{n+i}, \dots, y_{k_i} \le x_{n+k_i} \le x'_{n+i}$$

所以 $x'_{n+i}$ 至少大等于 $Y$ 也即 $Y'$ 中的 $i$ 个元素，那么自然有 $y'_i \le x'_{n+i}\,(1 \le i \le l)$。

**证毕**

再回到原命题的证明：

我们实际上只需要证明调用完 InsertionSort($h$) 的紧接着下一次调用 InsertionSort($k$) 后，$h$ 个子列仍有序即可，之后容易用归纳法得出。下面只考虑下一个调用：

执行完 InsertionSort($h$) 后，如下组已经完成排序：

$$\begin{gathered} A_1, A_{1+h}, A_{1+2h}, \dots \\ A_2, A_{2+h}, A_{2+2h}, \dots \\ \vdots \\ A_h, A_{h+h}, A_{h+2h}, \dots \end{gathered}$$

而之后执行 InsertionSort($k$)，则会将如下组排序：

$$\begin{gathered} A_1, A_{1+k}, A_{1+2k}, \dots \\ A_2, A_{2+k}, A_{2+2k}, \dots \\ \vdots \\ A_k, A_{k+k}, A_{k+2k}, \dots \end{gathered}$$

对于每个 $i$ $(1 \le i \le \min(h,k))$，考虑如下两个组：

$$\begin{gathered} A_i, A_{i+k}, A_{i+2k}, \dots \\ \dots, A_{i+h}, A_{i+h+k}, A_{i+h+2k}, \dots \end{gathered}$$

第二个组前面也加上“...”的原因是可能 $i+h \ge k$ 从而前面也有元素。

则第二个组就是引理 1 中的 $X$ 数组，第一个组就是 $Y$ 数组，$l$ 就是第二个组从 $i+h$ 之后顶到末尾的长度，$n$ 是第二个组中前面那个“...”的长度，$m$ 是第一个组去掉前 $l$ 个后剩下的个数。

又因为有：

$$A_i \le A_{i+h}, A_{i+k} \le A_{i+h+k}, \dots$$

所以由引理 1 可得执行 InsertionSort($k$) 将两个组分别排序后，这个关系依然满足，即依然有 $A_i \le A_{i+h}\,(1 \le i \le \min(h,k))$。

若有 $i > \min(h,k)$，容易发现取正整数 $w$ $(1 \le w \le \min(h,k))$ 再加上若干个 $k$ 即可得到 $i$，则之前的情况已经蕴含了此情况的证明。

综合以上论述便有：执行完 InsertionSort($k$) 依然有 $A_i \le A_{i+h}\,(1 \le i \le n-h)$。

得证。

**证毕**

这个定理揭示了希尔排序在特定集合 $H$ 下可以优化复杂度的关键，因为在整个过程中，它可以一致保持前面的成果不被摧毁（即 $h$ 个子列分别有序），从而使后面的调用中，指针 $i$ 的移动次数大大减少。

接下来我们单拎出来一个数论引理进行证明。这个定理在 OI 界因 小凯的疑惑[2] 一题而大为出名。而在希尔排序复杂度的证明中，它也使得定理 1 得到了很大的扩展。

> **引理 2**：若 $a,b$ 均为正整数且互素，则不在集合 $\{ax+by \mid x,y\in\mathbb{N}\}$ 中的最大正整数为 $ab-a-b$。
>
> **证明**：
>
> 分两步证明：
>
> - 先证明方程 $ax+by=ab-a-b$ 没有 $x,y$ 均为非负整数的解：
>
>  若无非负整数的限制，容易得到两组解 $(b-1,-1),(-1,a-1)$。
>
>  通过其通解形式 $x=x_0+tb,y=y_0-ta$，容易得到上面两组解是“相邻”的（因为 $b-1-b=-1$）。
>
>  当 $t$ 递增时，$x$ 递增，$y$ 递减，所以如果方程有非负整数解，必然会夹在这两组解中间，但这两组解“相邻”，中间没有别的解。
>
>  故不可能有非负整数解。
> - 再证明对任意整数 $c>ab-a-b$，方程 $ax+by=c$ 有非负整数解：
>
>  我们找一组解 $(x_0,y_0)$ 满足 $0\le x_0<b$（由通解的表达式，这可以做到）。
>
>  则有：
>
> $$by_0=c-ax_0\ge c-a(b-1)>ab-a-b-ab+a=-b$$
>
>  所以 $b(y_0+1)>0$，又因为 $b>0$，所以 $y_0+1>0$，所以 $y_0\ge 0$。
>
>  所以 $(x_0,y_0)$ 为一组非负整数解。
>
> 综上得证。
>
> **证毕**

而下面这个定理则揭示了引理 2 是如何扩展定理 1 的。

**定理 2**：如果 $\gcd(h_{t+1},h_t)=1$，则程序先执行完 InsertionSort$(h_{t+1})$ 与 InsertionSort$(h_t)$ 后，执行 InsertionSort$(h_{t-1})$ 的时间复杂度为 $O\left(\frac{nh_{t+1}h_t}{h_{t-1}}\right)$，且对于每个 $j$，其 $i$ 的移动次数是 $O\left(\frac{h_{t+1}h_t}{h_{t-1}}\right)$ 级别的。

**证明**：

对于 $j\le h_{t+1}h_t$ 的部分，$i$ 的移动次数显然是是 $O\left(\frac{h_{t+1}h_t}{h_{t-1}}\right)$ 级别的。

故以下假设 $j>h_{t+1}h_t$。

对于任意的正整数 $k$ 满足 $1\le k\le j-h_{t+1}h_t$，注意到：$h_{t+1}h_t-h_{t+1}-h_t<h_{t+1}h_t\le j-k\le j-1$

又因为 $\gcd(h_{t+1},h_t)=1$，故由引理 2，得存在非负整数 $a,b$，使得：$ah_{t+1}+bh_t=j-k$。

即得：

$$k=j-ah_{t+1}-bh_t$$

由定理 1，得：

$$A_{j-bh_t}\le A_{j-(b-1)h_t}\le\ldots\le A_{j-h_t}\le A_j$$

与

$$A_{j-bh_t-ah_{t+1}}\le A_{j-bh_t-(a-1)h_{t+1}}\le\ldots\le A_{j-bh_t-h_{t+1}}\le A_{j-bh_t}$$

综合以上既有：$A_k=A_{j-ah_{t+1}-bh_t}\le A_j$。

所以对于任何 $1\le k\le j-h_{t+1}h_t$，有 $A_k\le A_j$。

在 Shell-Sort 伪代码中 $i$ 指针每次减 $h_{t-1}$，减 $O\left(\frac{h_{t+1}h_t}{h_{t-1}}\right)$ 次，即可使得 $i\le j-h_{t+1}h_t$，进而有 $A_i\le A_j$，不满足 while 循环的条件退出。

证明完对于每个 $j$ 的移动复杂度后，即可得到总的时间复杂度：

$$\sum_{j=h_{t-1}+1}^{n} O\left(\frac{h_{t+1}h_t}{h_{t-1}}\right) = O\left(\frac{nh_{t+1}h_t}{h_{t-1}}\right)$$

得证。

**证毕**

认真观察定理 2 的证明过程，可以发现：定理 1 可以进行“线性组合”，即 $A$ 以 $h$ 为间隔有序，以 $k$ 为间隔亦有序，则以 $h$ 和 $k$ 的非负系数线性组合仍是有序的。而这种“线性性”即是由引理 2 保证的。

有了这两个定理，我们可以命题 1 与 2。

先证明命题 1：

**证明**：

将 $H$ 写为序列的形式：

$$H(h_1 = 1, h_2 = 3, h_3 = 7, \dots, h_{\lfloor \log_2 n \rfloor} = 2^{\lfloor \log_2 n \rfloor} - 1)$$

Shell-Sort 执行顺序为：$\text{InsertionSort}(h_{\lfloor \log_2 n \rfloor}), \text{InsertionSort}(h_{\lfloor \log_2 n \rfloor - 1}), \dots, \text{InsertionSort}(h_2), \text{InsertionSort}(h_1)$.

分两部分去分析复杂度：

- 对于前面的若干个满足 $h_t \geq \sqrt{n}$ 的 $h_t$，显然有 $\text{InsertionSort}(h_t)$ 的时间复杂度为 $O\left(\frac{n^2}{h_t}\right)$。

  考虑对最接近 $\sqrt{n}$ 的项 $h_k$，有：

  $$O\left(\frac{n^2}{h_t}\right) = O(n^{3/2})$$

  而对于 $i > k$ 的 $h_i$，因为有 $2h_i < h_{i+1}$，所以可得：

  $$O\left(\frac{n^2}{h_i}\right) = O(n^{3/2}/2^{i-k})\,(i > k)$$

  所以大等于 $\sqrt{n}$ 部分的总时间复杂度为：

  $$\sum_{i=k}^{\lfloor \log_2 n \rfloor} O(n^{3/2}/2^{i-k}) = O(n^{3/2})$$

- 对于后面剩下的满足 $h_t < \sqrt{n}$ 的项，前两项的复杂度还是 $O(n^{3/2})$，而对于后面的项 $h_t$，有定理 2 可得时间复杂度为：

  $$O\left(\frac{nh_{t+2}h_{t+1}}{h_t}\right) = O\left(\frac{nh_{t+2} \cdot h_{t+2}/2}{h_{t+2}/4}\right) = O(nh_{t+2})$$

  再次利用 $2h_i < h_{i+1}$ 性质可得此部分总时间复杂度为（下式中 $k$ 沿用了上一种情况中的含义）：

  $$2O(n^{3/2}) + \sum_{i=1}^{k-3} O(nh_{i+1}) = O(n^{3/2}) + \sum_{i=1}^{k-3} O(nh_{k-1}/2^{k-i-3}) = O(n^{3/2}) + O(nh_{k-1}) = O(n^{3/2})$$

  综上可得总时间复杂度即为 $O(n^{3/2})$。

**证毕**

再证明命题 2：

**证明**：

注意到一个事实：如果已经执行过了 $\text{InsertionSort}(2)$ 与 $\text{InsertionSort}(3)$，那么因为 $2 \cdot 3 - 2 - 3 = 1$，所以由定理 2，每个元素只有与它相邻的前一个元素可能大于它，之前的元素全部都小于它。于是 $i$ 指针只需要最多两次就可以退出 while 循环。也就是说，此时再执行 $\text{InsertionSort}(1)$，复杂度降为 $O(n)$。

更进一步：如果已经执行过了 $\text{InsertionSort}(4)$ 与 $\text{InsertionSort}(6)$，我们考虑所有的下标为奇数的元素组成的子列与下标为偶数的元素组成的子列。则这相当于把这两个子列分别执行 $\text{InsertionSort}(2)$ 与 $\text{InsertionSort}(3)$。那么也

是一样，这时候再执行 InsertionSort(2)，相当于对两个子列分别执行 InsertionSort(1)，也只需要两个序列和的级别，即 $O(n)$ 的复杂度就可以将数组变为 2 间隔有序。

不断归纳，就可以得到：如果已经执行过了 InsertionSort($2h$) 与 InsertionSort($3h$)，则执行 InsertionSort($h$) 的复杂度也只有 $O(n)$。

接下来分为两部分分析复杂度：

- 对于 $h_t > n/3$ 的部分，则执行每个 InsertionSort($h_t$) 的复杂度为 $O(n^2/h_t)$。

  而 $n^2/h_t < 3n$，所以单词插入排序复杂度为 $O(n)$。

  而这一部分元素个数是 $O(\log^2 n)$ 级别的，所以这一部分时间复杂度为 $O(n \log^2 n)$。
- 对于 $h_t \leq n/3$ 的部分，因为 $3h_t \leq n$，所以这之前已经执行了 InsertionSort($2h_t$) 与 InsertionSort($3h_t$)，于是执行 InsertionSort($h_t$) 的时间复杂度是 $O(n)$。

  还是一样的，这一部分元素个数也是 $O(\log^2 n)$ 级别的，所以这一部分时间复杂度为 $O(n \log^2 n)$。

综上可得总时间复杂度即为 $O(n \log^2 n)$。

**证毕**

### 空间复杂度

希尔排序的空间复杂度为 $O(1)$。

## 实现

=== "C++[1]"

```cpp
template <typename T>
void shell_sort(T array[], int length) {
  int h = 1;
  while (h < length / 3) {
    h = 3 * h + 1;
  }
  while (h >= 1) {
    for (int i = h; i < length; i++) {
      for (int j = i; j >= h && array[j] < array[j - h]; j -= h) {
        std::swap(array[j], array[j - h]);
      }
    }
    h = h / 3;
  }
}
```

=== "Python"

```python
def shell_sort(array, length):
    h = 1
    while h < length / 3:
        h = int(3 * h + 1)
    while h >= 1:
        for i in range(h, length):
            j = i
            while j >= h and array[j] < array[j - h]:
                array[j], array[j - h] = array[j - h], array[j]
```

```
                j -= h
        h = int(h / 3)
```

## 参考资料与注释

[1] 希尔排序 - 维基百科， 自由的百科全书

[2] 小凯的疑惑

# 1.7.12 锦标赛排序

本页面将简要介绍锦标赛排序。

## 定义

锦标赛排序（英文：Tournament sort），又被称为树形选择排序，是 选择排序 的优化版本，堆排序 的一种变体（均采用完全二叉树）。它在选择排序的基础上使用优先队列查找下一个该选择的元素。

## 引入

锦标赛排序的名字来源于单败淘汰制的竞赛形式。在这种赛制中有许多选手参与比赛，他们两两比较，胜者进入下一轮比赛。这种淘汰方式能够决定最好的选手，但是在最后一轮比赛中被淘汰的选手不一定是第二好的——他可能不如先前被淘汰的选手。

## 过程

以**最小锦标赛排序树**为例：

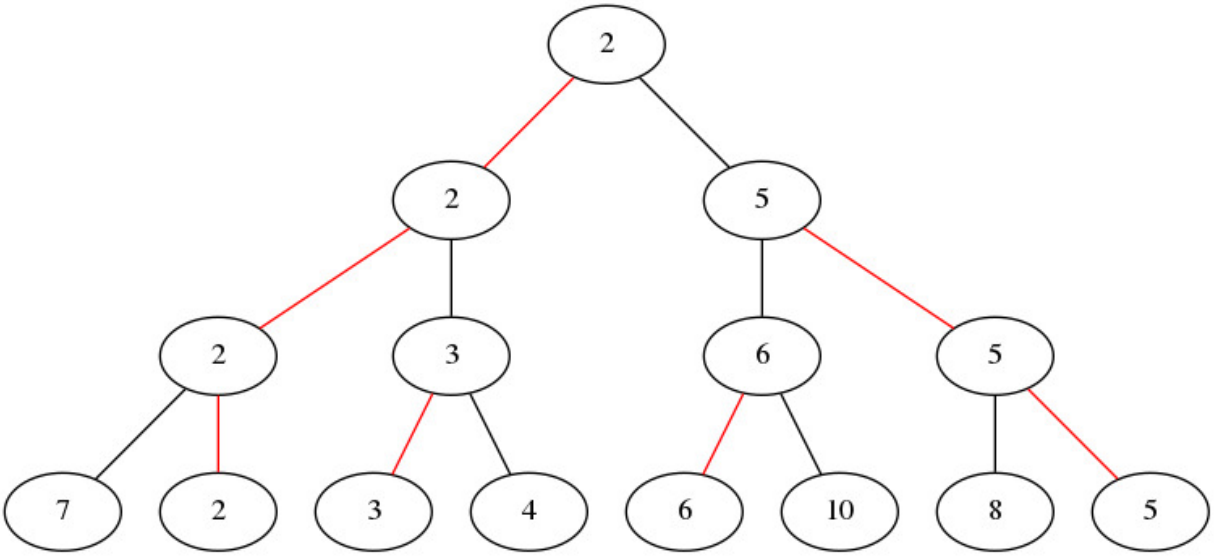


图 1.10 tournament-sort1

待排序元素是叶子节点显示的元素。红色边显示的是每一轮比较中较小的元素的胜出路径。显然，完成一次 " 锦标赛 " 可以选出一组元素中最小的那一个。

每一轮对 $n$ 个元素进行比较后可以得到 $\frac{n}{2}$ 个「优胜者」，每一对中较小的元素进入下一轮比较。如果无法凑齐一对元素，那么这个元素直接进入下一轮的比较。

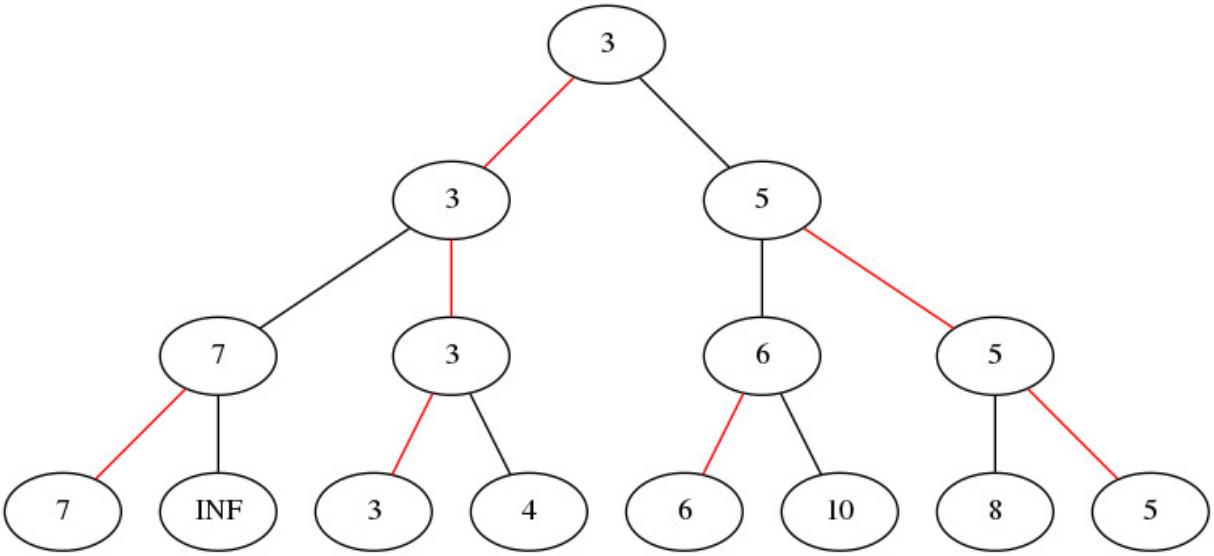


图 1.11 tournament-sort2

完成一次「锦标赛」后需要将被选出的元素去除。直接将其设置为 $\infty$（这个操作类似 堆排序），然后再次举行「锦标赛」选出次小的元素。

之后一直重复这个操作，直至所有元素有序。

## 性质

### 稳定性

锦标赛排序是一种不稳定的排序算法。

### 时间复杂度

锦标赛排序的最优时间复杂度、平均时间复杂度和最坏时间复杂度均为 $O(n \log n)$。它用 $O(n)$ 的时间初始化「锦标赛」，然后用 $O(\log n)$ 的时间从 $n$ 个元素中选取一个元素。

### 空间复杂度

锦标赛排序的空间复杂度为 $O(n)$。

## 实现

=== "C++"

```cpp
int n, a[maxn], tmp[maxn << 1];

int winner(int pos1, int pos2) {
  int u = pos1 >= n ? pos1 : tmp[pos1];
  int v = pos2 >= n ? pos2 : tmp[pos2];
  if (tmp[u] <= tmp[v]) return u;
  return v;
}

void creat_tree(int &value) {
  for (int i = 0; i < n; i++) tmp[n + i] = a[i];
  for (int i = 2 * n - 1; i > 1; i -= 2) {
```

```
    int k = i / 2;
    int j = i - 1;
    tmp[k] = winner(i, j);
  }
  value = tmp[tmp[1]];
  tmp[tmp[1]] = INF;
}

void recreat(int &value) {
  int i = tmp[1];
  while (i > 1) {
    int j, k = i / 2;
    if (i % 2 == 0 && i < 2 * n - 1)
      j = i + 1;
    else
      j = i - 1;
    tmp[k] = winner(i, j);
    i = k;
  }
  value = tmp[tmp[1]];
  tmp[tmp[1]] = INF;
}

void tournament_sort() {
  int value;
  creat_tree(value);
  for (int i = 0; i < n; i++) {
    a[i] = value;
    recreat(value);
  }
}
```

=== ”Python”

```python
n = 0
a = [0] * maxn
tmp = [0] * maxn * 2

def winner(pos1, pos2):
    u = pos1 if pos1 >= n else tmp[pos1]
    v = pos2 if pos2 >= n else tmp[pos2]
    if tmp[u] <= tmp[v]:
        return u
    return v

def creat_tree(value):
    for i in range(0, n):
        tmp[n + 1] = a[i]
    for i in range(2 * n -1, 1, -2):
        k = int(i / 2)
        j = i - 1
        tmp[k] = winner(i, j)
    value = tmp[tmp[i]]
```

```
    tmp[tmp[i]] = INF

def recreat(value):
    i = tmp[1]
    while i > 1:
        j = k = int(i / 2)
        if i % 2 == 0 and i < 2 * n - 1:
            j = i + 1
        else:
            j = i - 1
        tmp[k] = winner(i, j)
        i = k
    value = tmp[tmp[1]]
    tmp[tmp[1]] = INF

def tournament_sort():
    value = 0
    creat_tree(value)
    for i in range(0, n):
        a[i] = value
        recreat(value)
```

## 外部链接

- Tournament sort - Wikipedia[1]

## 参考资料与注释

[1] Tournament sort - Wikipedia

# 1.7.13 tim 排序

Author:Backl1ght

tim 排序是归并排序和插入排序的结合，是一个**稳定**的排序算法，由 Tim Peters 于 2002 年用 Python 实现。现在，tim 排序是 Python 的标准排序算法，且被 Java SE7 用于对非原始类型的数组排序。

tim 排序在最好情况下的时间复杂度为 $O(n)$，最差情况下的时间复杂度为 $O(n \log n)$，期望时间复杂度为 $O(n \log n)$。tim 排序在最坏情况下的空间复杂度为 $O(n)$。

## 算法

众所周知，归并排序是先将数组划分为两部分，然后递归地对两个子数组进行归并排序，最后合并两个子数组。这样一来，归并排序合并操作的最小单位就是单个元素。但是，数组中可能原本就存在连续且有序的子数组，归并排序无法利用这个特性。

tim 排序为了利用数组中本身就存在的连续且有序的子数组，以 RUN 作为合并操作的最小单位。其中，RUN 是一个满足以下性质的子数组：

- 一个 RUN 要么是非降序的，要么是严格升序的。
- 一个 RUN 存在一个长度的下限。

tim 排序的过程就是一个类似归并排序的过程，将数组划分为多个 RUN，然后以某种规则不断地合并两个 RUN，直到数组有序。具体过程如下：

令 $nRemaining$ 初始化为数组的大小，$minRun$ 初始化为 $getMinRunLength(nRemaining)$。

1 **do**
2 确定$run$的起点
3 **if**$run$比$minRun$短
4 延长$run$直至 $\min(minRun, nRemaining)$
5 **push**$run$放到$pending-runstack$上
6 **if**$pending-runstack$最顶上的 2 个$run$长度相近
7 合并$pending-runstack$最顶上的 2 个$run$
8 start index $\leftarrow$ start index $+$ $run$的长度
9 $nRemaining \leftarrow nRemaining -$ $run$的长度
10 **while** $nRemaining \neq 0$

其中，$getMinRunLength$ 函数是根据当前数组长度确定 $minRun$ 具体值的函数，natural run 的意思是原本就非降序或者严格升序的 run，扩展长度不够的 run 就是用插入排序往 run 中添加元素。

## 复杂度证明

最好情况下，数组本身就有序，即数组本身就是一个 RUN，这个时候 tim 排序就遍历了一遍数组，找到了唯一的 RUN，就结束了。所以，最好的情况下，tim 排序的时间复杂度为 $O(n)$。

## 写在后面

本文只是简单介绍了 tim 排序的原理，实际上 tim 排序在实现的时候还有一些其他的优化，这里不再一一列举。tim 排序在 java 中的实现写得非常好，要想知道真正的 tim 排序推荐去看 java 中 tim 排序的实现。

## 参考资料

1. Timsort[1]
2. On the Worst-Case Complexity of TimSort[2]
3. original explanation by Tim Peters[3]
4. java 实现[4]
5. c 语言实现[5]

## 参考资料与注释

[1] Timsort

[2] On the Worst-Case Complexity of TimSort

[3] original explanation by Tim Peters

[4] java 实现

[5] c 语言实现

# 1.7.14 排序相关 STL

本页面将简要介绍 C 和 C++ 标准库中实现的排序算法。

除已说明的函数外，本页所列函数默认定义于头文件 `<algorithm>` 中。

## qsort

参见：`qsort`[2]，`std::qsort`[3]

该函数为 C 标准库实现的 快速排序，定义在 `<stdlib.h>` 中。在 C++ 标准库里，该函数定义在 `<cstdlib>` 中。

### qsort 与 bsearch 的比较函数

qsort 函数有四个参数：数组名、元素个数、元素大小、比较规则。其中，比较规则通过指定比较函数来实现，指定不同的比较函数可以实现不同的排序规则。

比较函数的参数限定为两个 const void 类型的指针。返回值规定为正数、负数和 0。

比较函数的一种示例写法为：

```
int compare(const void *p1, const void *p2)  // int 类型数组的比较函数
{
  int *a = (int *)p1;
  int *b = (int *)p2;
  if (*a > *b)
    return 1;  // 返回正数表示 a 大于 b
  else if (*a < *b)
    return -1;  // 返回负数表示 a 小于 b
  else
    return 0;  // 返回 0 表示 a 与 b 等价
}
```

注意：返回值用两个元素相减代替正负数是一种典型的错误写法，因为这样可能会导致溢出错误。

以下是排序结构体的一个示例：

```
struct eg  // 示例结构体
{
  int e;
  int g;
};

int compare(const void *p1,
            const void *p2)  // struct eg 类型数组的比较函数：按成员 e 排序
{
  struct eg *a = (struct eg *)p1;
  struct eg *b = (struct eg *)p2;
  if (a->e > b->e)
    return 1;  // 返回正数表示 a 大于 b
  else if (a->e < b->e)
    return -1;  // 返回负数表示 a 小于 b
  else
    return 0;  // 返回 0 表示 a 与 b 等价
}
```

这里也可以看出，等价不代表相等，只代表在此比较规则下两元素等价。

## std::sort

参见：`std::sort`[4]

用法：

```
// a[0] .. a[n - 1] 为需要排序的数列
// 对 a 原地排序，将其按从小到大的顺序排列
std::sort(a, a + n);

// cmp 为自定义的比较函数
std::sort(a, a + n, cmp);
```

注意：sort 的比较函数的返回值是 true 和 false，用 true 和 false 表示两个元素的大小（先后）关系，这与 qsort 的三值比较函数的语义完全不同。具体内容详见上方给出的 sort 的文档。

如果要将 sort 简单改写为 qsort，维持排序顺序整体上不变（不考虑等价的元素），需要将返回 true 改为 - 1，返回 false 改为 1。

`std::sort` 函数是更常用的 C++ 库比较函数。该函数的最后一个参数为二元比较函数，未指定 `cmp` 函数时，默认按从小到大的顺序排序。

旧版 C++ 标准中仅要求它的**平均**时间复杂度达到 $O(n \log n)$。C++11 标准以及后续标准要求它的**最坏**时间复杂度达到 $O(n \log n)$。

C++ 标准并未严格要求此函数的实现算法，具体实现取决于编译器。libstdc++[5] 和 libc++[6] 中的实现都使用了 内省排序。

## std::nth_element

参见：`std::nth_element`[7]

用法：

```
std::nth_element(first, nth, last);
std::nth_element(first, nth, last, cmp);
```

它重排 `[first, last)` 中的元素，使得 `nth` 所指向的元素被更改为 `[first, last)` 排好序后该位置会出现的元素。这个新的 `nth` 元素前的所有元素小于或等于新的 `nth` 元素后的所有元素。

实现算法是未完成的内省排序。

对于以上两种用法，C++ 标准要求它的平均时间复杂度为 $O(n)$，其中 n 为 `std::distance(first, last)`。

它常用于构建 K-D Tree。

## std::stable_sort

参见：`std::stable_sort`[8]

用法：

```
std::stable_sort(first, last);
std::stable_sort(first, last, cmp);
```

稳定排序，保证相等元素排序后的相对位置与原序列相同。

时间复杂度为 $O(n \log(n)^2)$，当额外内存可用时，复杂度为 $O(n \log n)$。

## std::partial_sort

参见：`std::partial_sort`[9]

用法：

```
// mid = first + k
```

```
std::partial_sort(first, mid, last);
std::partial_sort(first, mid, last, cmp);
```

将序列中前 k 元素按 cmp 给定的顺序进行原地排序，后面的元素不保证顺序。未指定 cmp 函数时，默认按从小到大的顺序排序。

复杂度：约 $(last - first)\log(mid - first)$ 次应用 cmp。

原理：

std::partial_sort 的思想是：对原始容器内区间为 [first, mid) 的元素执行 make_heap() 操作，构造一个大根堆，然后将 [mid, last) 中的每个元素和 first 进行比较，保证 first 内的元素为堆内的最大值。如果小于该最大值，则互换元素位置，并对 [first, mid) 内的元素进行调整，使其保持最大堆序。比较完之后，再对 [first, mid) 内的元素做一次堆排序 sort_heap() 操作，使其按增序排列。注意，堆序和增序是不同的。

## 自定义比较

参见：运算符重载[10]

内置类型（如 int）和用户定义的结构体允许定制调用 STL 排序函数时使用的比较函数。可以在调用该函数时，在最后一个参数中传入一个实现二元比较的函数。

对于用户定义的结构体，对其使用 STL 排序函数前必须定义至少一种关系运算符，或是在使用函数时提供二元比较函数。通常推荐定义 operator<。[1]

示例：

```
int a[1009], n = 10;
// ...
std::sort(a + 1, a + 1 + n);                  // 从小到大排序
std::sort(a + 1, a + 1 + n, greater<int>());  // 从大到小排序
```

```
struct data {
  int a, b;

  bool operator<(const data rhs) const {
    return (a == rhs.a) ? (b < rhs.b) : (a < rhs.a);
  }
} da[1009];

bool cmp(const data u1, const data u2) {
  return (u1.a == u2.a) ? (u1.b > u2.b) : (u1.a > u2.a);
}

// ...
std::sort(da + 1, da + 1 + 10);  // 使用结构体中定义的 < 运算符，从小到大排序
std::sort(da + 1, da + 1 + 10, cmp);  // 使用 cmp 函数进行比较，从大到小排序
```

### 严格弱序

参见：Strict weak orderings[11]

进行排序的运算符必须满足严格弱序，否则会出现不可预料的情况（如运行时错误、无法正确排序）。

严格弱序的要求：

1. $x \not< x$（非自反性）
2. 若 $x < y$，则 $y \not< x$（非对称性）
3. 若 $x < y, y < z$，则 $x < z$（传递性）
4. 若 $x \not< y, y \not< x, y \not< z, z \not< y$，则 $x \not< z, z \not< x$（不可比性的传递性）

常见的错误做法：

- 使用 `<=` 来定义排序中的小于运算符。
- 在调用排序运算符时，读取外部数值可能会改变的数组（常见于最短路算法）。
- 将多个数的最大最小值进行比较的结果作为排序运算符（如皇后游戏/加工生产调度中的经典错误）。

## 外部链接

- 浅谈邻项交换排序的应用以及需要注意的问题[12]

## 参考资料与注释

[1] 因为大部分标准算法默认使用 `operator<` 进行比较。

[2] `qsort`

[3] `std::qsort`

[4] `std::sort`

[5] libstdc++

[6] libc++

[7] `std::nth_element`

[8] `std::stable_sort`

[9] `std::partial_sort`

[10] 运算符重载

[11] Strict weak orderings

[12] 浅谈邻项交换排序的应用以及需要注意的问题

# 1.7.15 排序应用

本页面将简要介绍排序的用法。

## 理解数据的特点

使用排序处理数据有利于理解数据的特点，方便我们之后的分析与视觉化。像一些生活中的例子比如词典，菜单，如果不是按照一定顺序排列的话，人们想要找到自己需要的东西的时间就会大大增加。

计算机需要处理大规模的数据，排序后，人们可以根据数据的特点和需求来设计计算机的后续处理流程。

### 降低时间复杂度

使用排序预处理可以降低求解问题所需要的时间复杂度，通常是一个以空间换取时间的平衡。如果一个排序好的列表需要被多次分析的话，只需要耗费一次排序所需要的资源是很划算的，因为之后的每次分析都可以减少很多时间。

**示例：检查给定数列中是否有相等的元素**

考虑一个数列，你需要检查其中是否有元素相等。

一个朴素的做法是检查每一个数对，并判断这一对数是否相等。时间复杂度是 $O(n^2)$。

我们不妨先对这一列数排序，之后不难发现：如果有相等的两个数，它们一定在新数列中处于相邻的位置上。这时，只需要 $O(n)$ 地扫一遍新数列了。

总的时间复杂度是排序的复杂度 $O(n \log n)$。

### 作为查找的预处理

排序是 二分查找 所要做的预处理工作。在排序后使用二分查找，可以以 $O(\log n)$ 的时间在序列中查找指定的元素。

# 1.8 前缀和 & 差分

## 前缀和

### 定义

前缀和可以简单理解为「数列的前 $n$ 项的和」，是一种重要的预处理方式，能大大降低查询的时间复杂度。[1]

C++ 标准库中实现了前缀和函数 `std::partial_sum`[2]，定义于头文件 `<numeric>` 中。

### 例题

**Note**

有 $N$ 个的正整数放到数组 $A$ 里，现在要求一个新的数组 $B$，新数组的第 $i$ 个数 $B[i]$ 是原数组 $A$ 第 0 到第 $i$ 个数的和。

输入：

```
5
1 2 3 4 5
```

输出：

```
1 3 6 10 15
```

**解题思路**

递推：`B[0] = A[0]`，对于 $i \geq 1$ 则 `B[i] = B[i-1] + A[i]`。

**参考代码**

=== "C++"

```cpp
#include <iostream>
```

```
using namespace std;

int N, A[10000], B[10000];

int main() {
  cin >> N;
  for (int i = 0; i < N; i++) {
    cin >> A[i];
  }

  // 前缀和数组的第一项和原数组的第一项是相等的。
  B[0] = A[0];

  for (int i = 1; i < N; i++) {
    // 前缀和数组的第 i 项 = 原数组的 0 到 i-1 项的和 + 原数组的第 i 项。
    B[i] = B[i - 1] + A[i];
  }

  for (int i = 0; i < N; i++) {
    cout << B[i] << " ";
  }

  return 0;
}
```

=== "Python"

```python
from itertools import accumulate
input()
print(*accumulate(map(int, input().split())))
```

## 二维/多维前缀和

多维前缀和的普通求解方法几乎都是基于容斥原理。

**示例：一维前缀和扩展到二维前缀和**

比如我们有这样一个矩阵 $a$，可以视为二维数组：

```
1 2 4 3
5 1 2 4
6 3 5 9
```

我们定义一个矩阵 $sum$ 使得 $sum_{x,y} = \sum_{i=1}^{x}\sum_{j=1}^{y} a_{i,j}$，

那么这个矩阵长这样：

```
1  3  7  10
6  9  15 22
12 18 29 45
```

第一个问题就是递推求 $sum$ 的过程，$sum_{i,j} = sum_{i-1,j} + sum_{i,j-1} - sum_{i-1,j-1} + a_{i,j}$。

因为同时加了 $sum_{i-1,j}$ 和 $sum_{i,j-1}$，故重复了 $sum_{i-1,j-1}$，减去。

第二个问题就是如何应用，譬如求 $(x_1, y_1) - (x_2, y_2)$ 子矩阵的和。

那么，根据类似的思考过程，易得答案为 $sum_{x_2,y_2} - sum_{x_1-1,y_2} - sum_{x_2,y_1-1} + sum_{x_1-1,y_1-1}$。

## 例题

### 洛谷 P1387 最大正方形[10]

在一个 $n \times m$ 的只包含 0 和 1 的矩阵里找出一个不包含 0 的最大正方形，输出边长。

### 参考代码

=== "C++"

```cpp
#include <algorithm>
#include <iostream>
using namespace std;
int a[103][103];
int b[103][103];  // 前缀和数组，相当于上文的 sum[]

int main() {
  int n, m;
  cin >> n >> m;

  for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= m; j++) {
      cin >> a[i][j];
      b[i][j] =
          b[i][j - 1] + b[i - 1][j] - b[i - 1][j - 1] + a[i][j];  // 求前缀和
    }
  }

  int ans = 1;

  int l = 2;
  while (l <= min(n, m)) {  // 判断条件
    for (int i = l; i <= n; i++) {
      for (int j = l; j <= m; j++) {
        if (b[i][j] - b[i - l][j] - b[i][j - l] + b[i - l][j - l] == l * l) {
          ans = max(ans, l);  // 在这里统计答案
        }
      }
    }
    l++;
  }

  cout << ans << endl;
  return 0;
}
```

=== "Python"

```python
n, m = map(int, input().split())
```

```
a = [list(map(int, input().split())) for _ in range(n)]
b = [a[0]] + [[i[0]] + [0] * (m - 1) for i in a[1:]]
ans = 0
for i in range(1, n):
    for j in range(1, m):
        if a[i][j]:
            b[i][j] = min(b[i-1][j], b[i][j-1], b[i-1][j-1]) + 1
            ans = max(ans, b[i][j])
print(ans)
```

## 基于 DP 计算高维前缀和

基于容斥原理来计算高维前缀和的方法，其优点在于形式较为简单，无需特别记忆，但当维数升高时，其复杂度较高。这里介绍一种基于 DP 计算高维前缀和的方法。该方法即通常语境中所称的**高维前缀和**。

设高维空间 $U$ 共有 $D$ 维，需要对 $f[\cdot]$ 求高维前缀和 $\mathrm{sum}[\cdot]$。令 $\mathrm{sum}[i][\mathrm{state}]$ 表示同 state 后 $D-i$ 维相同的所有点对于 state 点高维前缀和的贡献。由定义可知 $\mathrm{sum}[0][\mathrm{state}]=f[\mathrm{state}]$，以及 $\mathrm{sum}[\mathrm{state}]=\mathrm{sum}[D][\mathrm{state}]$。

其递推关系为 $\mathrm{sum}[i][\mathrm{state}]=\mathrm{sum}[i-1][\mathrm{state}]+\mathrm{sum}[i][\mathrm{state}']$，其中 $\mathrm{state}'$ 为第 $i$ 维恰好比 state 少 1 的点。该方法的复杂度为 $O(D\times|U|)$，其中 $|U|$ 为高维空间 $U$ 的大小。

一种实现的伪代码如下：

**for** $state$
  $sum[state]\leftarrow f[state]$
**for** $i\leftarrow 0$ **to** $D$
  **for** $state'$ **in** lexicographical order
    $sum[state]\leftarrow sum[state]+sum[state']$

## 树上前缀和

设 $sum_i$ 表示结点 $i$ 到根节点的权值总和。

然后：

- 若是点权，$x,y$ 路径上的和为 $sum_x+sum_y-sum_{lca}-sum_{fa_{lca}}$。
- 若是边权，$x,y$ 路径上的和为 $sum_x+sum_y-2\cdot sum_{lca}$。
  LCA 的求法参见最近公共祖先。

# 差分

## 解释

差分是一种和前缀和相对的策略，可以当做是求和的逆运算。

这种策略的定义是令 $b_i=\begin{cases}a_i-a_{i-1} & i\in[2,n]\\ a_1 & i=1\end{cases}$

## 性质

- $a_i$ 的值是 $b_i$ 的前缀和，即 $a_n=\sum_{i=1}^{n}b_i$
- 计算 $a_i$ 的前缀和 $sum=\sum_{i=1}^{n}a_i=\sum_{i=1}^{n}\sum_{j=1}^{i}b_j=\sum_{i}^{n}(n-i+1)b_i$

它可以维护多次对序列的一个区间加上一个数，并在最后询问某一位的数或是多次询问某一位的数。注意修改操作一定要在查询操作之前。

**示例**

譬如使 $[l,r]$ 中的每个数加上一个 $k$，即

$$b_l \leftarrow b_l + k, b_{r+1} \leftarrow b_{r+1} - k$$

其中 $b_l + k = a_l + k - a_{l-1}$，$b_{r+1} - k = a_{r+1} - (a_r + k)$

最后做一遍前缀和就好了。

C++ 标准库中实现了差分函数 `std::adjacent_difference`[3]，定义于头文件 `<numeric>` 中。

## 树上差分

树上差分可以理解为对树上的某一段路径进行差分操作，这里的路径可以类比一维数组的区间进行理解。例如在对树上的一些路径进行频繁操作，并且询问某条边或者某个点在经过操作后的值的时候，就可以运用树上差分思想了。

树上差分通常会结合树基础和最近公共祖先来进行考察。树上差分又分为**点差分**与**边差分**，在实现上会稍有不同。

### 点差分

举例：对树上的一些路径 $\delta(s_1,t_1),\delta(s_2,t_2),\delta(s_3,t_3)\ldots$ 进行访问，问一条路径 $\delta(s,t)$ 上的点被访问的次数。

对于一次 $\delta(s,t)$ 的访问，需要找到 $s$ 与 $t$ 的公共祖先，然后对这条路径上的点进行访问（点的权值加一），若采用 DFS 算法对每个点进行访问，由于有太多的路径需要访问，时间上承受不了。这里进行差分操作：

$$\begin{aligned} d_s &\leftarrow d_s + 1 \\ d_{lca} &\leftarrow d_{lca} - 1 \\ d_t &\leftarrow d_t + 1 \\ d_{f(lca)} &\leftarrow d_{f(lca)} - 1 \end{aligned}$$

其中 $f(x)$ 表示 $x$ 的父亲节点，$d_i$ 为点权 $a_i$ 的差分数组。

可以认为公式中的前两条是对蓝色方框内的路径进行操作，后两条是对红色方框内的路径进行操作。不妨令 $lca$ 左侧的直系子节点为 $left$。那么有 $d_{lca} - 1 = a_{lca} - (a_{left} + 1)$，$d_{f(lca)} - 1 = a_{f(lca)} - (a_{lca} + 1)$。可以发现实际上点差分的操作和上文一维数组的差分操作是类似的。

### 边差分

若是对路径中的边进行访问，就需要采用边差分策略了，使用以下公式：

$$\begin{aligned} d_s &\leftarrow d_s + 1 \\ d_t &\leftarrow d_t + 1 \\ d_{lca} &\leftarrow d_{lca} - 2 \end{aligned}$$

由于在边上直接进行差分比较困难，所以将本来应当累加到红色边上的值向下移动到附近的点里，那么操作起来也就方便了。对于公式，有了点差分的理解基础后也不难推导，同样是对两段区间进行差分。

## 例题

**洛谷 3128 最大流**[20]

FJ 给他的牛棚的 $N(2 \le N \le 50,000)$ 个隔间之间安装了 $N-1$ 根管道，隔间编号从 1 到 $N$。所有隔间都被管道连通了。

FJ 有 $K(1 \le K \le 100,000)$ 条运输牛奶的路线，第 $i$ 条路线从隔间 $s_i$ 运输到隔间 $t_i$。一条运输路线会给它

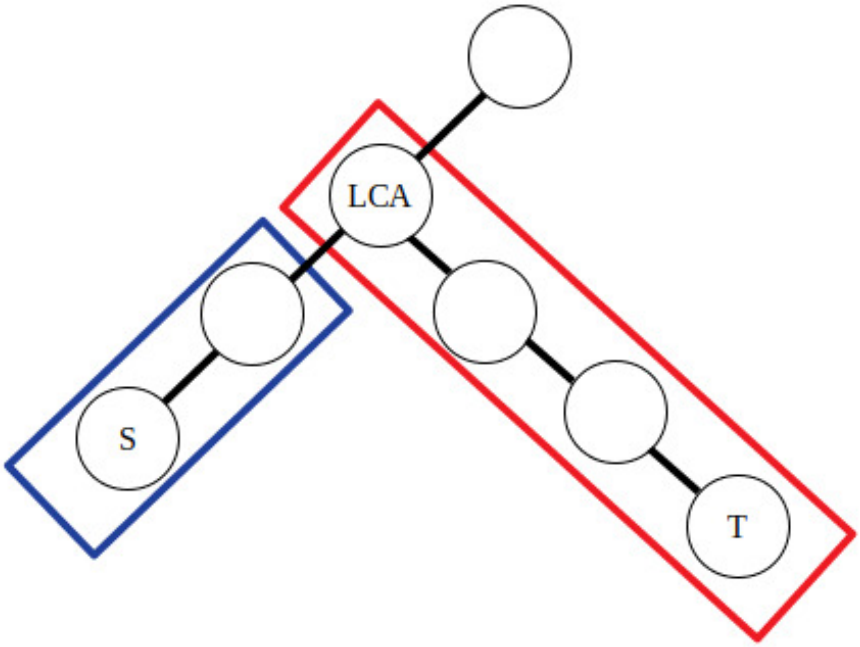


图 1.12

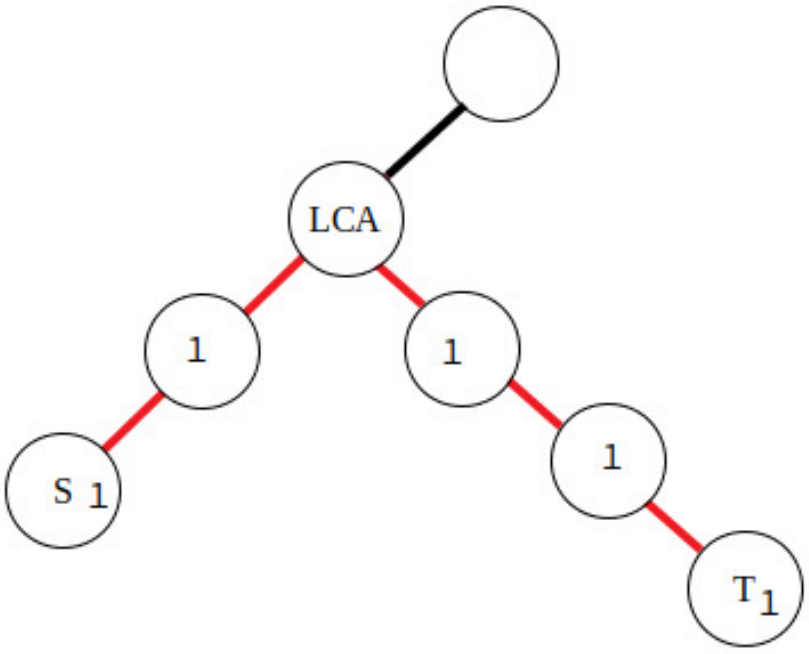


图 1.13

的两个端点处的隔间以及中间途径的所有隔间带来一个单位的运输压力，你需要计算压力最大的隔间的压力是多少。

### 解题思路

需要统计每个点经过了多少次，那么就用树上差分将每一次的路径上的点加一，可以很快得到每个点经过的次数。这里采用倍增法计算 LCA，最后对 DFS 遍历整棵树，在回溯时对差分数组求和就能求得答案了。

### 参考代码

```cpp
#include <bits/stdc++.h>

using namespace std;
#define maxn 50010

struct node {
  int to, next;
} edge[maxn << 1];

int fa[maxn][30], head[maxn << 1];
int power[maxn];
int depth[maxn], lg[maxn];
int n, k, ans = 0, tot = 0;

void add(int x, int y) {  // 加边
  edge[++tot].to = y;
  edge[tot].next = head[x];
  head[x] = tot;
}

void dfs(int now, int father) {  // dfs 求最大压力
  fa[now][0] = father;
  depth[now] = depth[father] + 1;
  for (int i = 1; i <= lg[depth[now]]; ++i)
    fa[now][i] = fa[fa[now][i - 1]][i - 1];
  for (int i = head[now]; i; i = edge[i].next)
    if (edge[i].to != father) dfs(edge[i].to, now);
}

int lca(int x, int y) {  // 求 LCA，最近公共祖先
  if (depth[x] < depth[y]) swap(x, y);
  while (depth[x] > depth[y]) x = fa[x][lg[depth[x] - depth[y]] - 1];
  if (x == y) return x;
  for (int k = lg[depth[x]] - 1; k >= 0; k--) {
    if (fa[x][k] != fa[y][k]) x = fa[x][k], y = fa[y][k];
  }
  return fa[x][0];
}

// 用 dfs 求最大压力，回溯时将子树的权值加上
void get_ans(int u, int father) {
  for (int i = head[u]; i; i = edge[i].next) {
    int to = edge[i].to;
```

```
    if (to == father) continue;
    get_ans(to, u);
    power[u] += power[to];
  }
  ans = max(ans, power[u]);
}

int main() {
  scanf("%d %d", &n, &k);
  int x, y;
  for (int i = 1; i <= n; i++) {
    lg[i] = lg[i - 1] + (1 << lg[i - 1] == i);
  }
  for (int i = 1; i <= n - 1; i++) {  // 建图
    scanf("%d %d", &x, &y);
    add(x, y);
    add(y, x);
  }
  dfs(1, 0);
  int s, t;
  for (int i = 1; i <= k; i++) {
    scanf("%d %d", &s, &t);
    int ancestor = lca(s, t);
    // 树上差分
    power[s]++;
    power[t]++;
    power[ancestor]--;
    power[fa[ancestor][0]]--;
  }
  get_ans(1, 0);
  printf("%d\n", ans);
  return 0;
}
```

# 习题

前缀和：

- 洛谷 B3612【深进 1. 例 1】求区间和[4]
- 洛谷 U69096 前缀和的逆[5]
- AT2412 最大 の 和[6]
- 「USACO16JAN」子共七 Subsequences Summing to Sevens[7]
- 「USACO05JAN」Moo Volume S[8]

二维/多维前缀和：

- HDU 6514 Monitor[9]
- 洛谷 P1387 最大正方形[10]
- 「HNOI2003」激光炸弹[11]

基于 DP 计算高维前缀和：

- CF 165E Compatible Numbers[12]
- CF 383E Vowels[13]

- ARC 100C Or Plus Max[14]

树上前缀和：

- LOJ 10134.Dis[15]
- LOJ 2491. 求和[16]

差分：

- 树状数组 3：区间修改，区间查询[17]
- P3397 地毯[18]
- 「Poetize6」IncDec Sequence[19]

树上差分：

- 洛谷 3128 最大流[20]
- JLOI2014 松鼠的新家[21]
- NOIP2015 运输计划[22]
- NOIP2016 天天爱跑步[23]

# 参考资料与注释

[1] 南海区青少年信息学奥林匹克内部训练教材

[2] `std::partial_sum`

[3] `std::adjacent_difference`

[4] 洛谷 B3612【深进 1. 例 1】求区间和

[5] 洛谷 U69096 前缀和的逆

[6] AT2412 最大の和

[7] 「USACO16JAN」子共七 Subsequences Summing to Sevens

[8] 「USACO05JAN」Moo Volume S

[9] HDU 6514 Monitor

[10] 洛谷 P1387 最大正方形

[11] 「HNOI2003」激光炸弹

[12] CF 165E Compatible Numbers

[13] CF 383E Vowels

[14] ARC 100C Or Plus Max

[15] LOJ 10134.Dis

[16] LOJ 2491. 求和

[17] 树状数组 3：区间修改，区间查询

[18] P3397 地毯

[19] 「Poetize6」IncDec Sequence

[20] 洛谷 3128 最大流

[21] JLOI2014 松鼠的新家

[22] NOIP2015 运输计划

[23] NOIP2016 天天爱跑步

# 1.9 二分

本页面将简要介绍二分查找，由二分法衍生的三分法以及二分答案。

## 二分法

### 定义

二分查找（英语：binary search），也称折半搜索（英语：half-interval search）、对数搜索（英语：logarithmic search），是用来在一个有序数组中查找某一元素的算法。

### 过程

以在一个升序数组中查找一个数为例。

它每次考察数组当前部分的中间元素，如果中间元素刚好是要找的，就结束搜索过程；如果中间元素小于所查找的值，那么左侧的只会更小，不会有所查找的元素，只需到右侧查找；如果中间元素大于所查找的值同理，只需到左侧查找。

### 性质

#### 时间复杂度

二分查找的最优时间复杂度为 $O(1)$。

二分查找的平均时间复杂度和最坏时间复杂度均为 $O(\log n)$。因为在二分搜索过程中，算法每次都把查询的区间减半，所以对于一个长度为 $n$ 的数组，至多会进行 $O(\log n)$ 次查找。

### 空间复杂度

迭代版本的二分查找的空间复杂度为 $O(1)$。

递归（无尾调用消除）版本的二分查找的空间复杂度为 $O(\log n)$。

## 实现

```c
int binary_search(int start, int end, int key) {
  int ret = -1;  // 未搜索到数据返回-1 下标
  int mid;
  while (start <= end) {
    mid = start + ((end - start) >> 1);  // 直接平均可能会溢出，所以用这个算法
    if (arr[mid] < key)
      start = mid + 1;
    else if (arr[mid] > key)
      end = mid - 1;
    else {  // 最后检测相等是因为多数搜索情况不是大于就是小于
      ret = mid;
      break;
    }
  }
  return ret;  // 单一出口
}
```

Note

参考编译优化 `#` 位运算代替乘法，对于 $n$ 是有符号数的情况，当你可以保证 $n \geq 0$ 时，`n >> 1` 比 `n / 2` 指令数更少。

## 最大值最小化

注意，这里的有序是广义的有序，如果一个数组中的左侧或者右侧都满足某一种条件，而另一侧都不满足这种条件，也可以看作是一种有序（如果把满足条件看做 1，不满足看做 0，至少对于这个条件的这一维度是有序的）。换言之，二分搜索法可以用来查找满足某种条件的最大（最小）的值。

要求满足某种条件的最大值的最小可能情况（最大值最小化），首先的想法是从小到大枚举这个作为答案的「最大值」，然后去判断是否合法。若答案单调，就可以使用二分搜索法来更快地找到答案。因此，要想使用二分搜索法来解这种「最大值最小化」的题目，需要满足以下三个条件：

1. 答案在一个固定区间内；
2. 可能查找一个符合条件的值不是很容易，但是要求能比较容易地判断某个值是否是符合条件的；
3. 可行解对于区间满足一定的单调性。换言之，如果 $x$ 是符合条件的，那么有 $x+1$ 或者 $x-1$ 也符合条件。（这样下来就满足了上面提到的单调性）

当然，最小值最大化是同理的。

## STL 的二分查找

C++ 标准库中实现了查找首个不小于给定值的元素的函数 `std::lower_bound`[1] 和查找首个大于给定值的元素的函数 `std::upper_bound`[2]，二者均定义于头文件 `<algorithm>` 中。

二者均采用二分实现，所以调用前必须保证元素有序。

## bsearch

bsearch 函数为 C 标准库实现的二分查找，定义在 `<stdlib.h>` 中。在 C++ 标准库里，该函数定义在 `<cstdlib>` 中。qsort 和 bsearch 是 C 语言中唯二的两个算法类函数。

bsearch 函数相比 qsort（排序相关 STL）的四个参数，在最左边增加了参数「待查元素的地址」。之所以按照地址的形式传入，是为了方便直接套用与 qsort 相同的比较函数，从而实现排序后的立即查找。因此这个参数不能直接传入具体值，而是要先将待查值用一个变量存储，再传入该变量地址。

于是 bsearch 函数总共有五个参数：待查元素的地址、数组名、元素个数、元素大小、比较规则。比较规则仍然通过指定比较函数实现，详见 排序相关 STL。

bsearch 函数的返回值是查找到的元素的地址，该地址为 void 类型。

注意：bsearch 与上文的 lower_bound 和 upper_bound 有两点不同：

- 当符合条件的元素有重复多个的时候，会返回执行二分查找时第一个符合条件的元素，从而这个元素可能位于重复多个元素的中间部分。
- 当查找不到相应的元素时，会返回 NULL。

用 lower_bound 可以实现与 bsearch 完全相同的功能，所以可以使用 bsearch 通过的题目，直接改写成 lower_bound 同样可以实现。但是鉴于上述不同之处的第二点，例如，在序列 1、2、4、5、6 中查找 3，bsearch 实现 lower_bound 的功能会变得困难。

利用 bsearch 实现 lower_bound 的功能比较困难，是否一定就不能实现？答案是否定的，存在比较 tricky 的技巧。借助编译器处理比较函数的特性：总是将第一个参数指向待查元素，将第二个参数指向待查数组中的元素，也可以用 bsearch 实现 lower_bound 和 upper_bound，如下文示例。只是，这要求待查数组必须是全局数组，从而可以直接传入首地址。

```c
int A[100005];  // 示例全局数组

// 查找首个不小于待查元素的元素的地址
int lower(const void *p1, const void *p2) {
  int *a = (int *)p1;
  int *b = (int *)p2;
  if ((b == A || compare(a, b - 1) > 0) && compare(a, b) > 0)
    return 1;
  else if (b != A && compare(a, b - 1) <= 0)
    return -1;  // 用到地址的减法，因此必须指定元素类型
  else
    return 0;
}

// 查找首个大于待查元素的元素的地址
int upper(const void *p1, const void *p2) {
  int *a = (int *)p1;
  int *b = (int *)p2;
  if ((b == A || compare(a, b - 1) >= 0) && compare(a, b) >= 0)
    return 1;
  else if (b != A && compare(a, b - 1) < 0)
    return -1;  // 用到地址的减法，因此必须指定元素类型
  else
    return 0;
}
```

因为现在的 OI 选手很少写纯 C，并且此方法作用有限，所以不是重点。对于新手而言，建议老老实实地使用 C++ 中的 lower_bound 和 upper_bound 函数。

## 二分答案

解题的时候往往会考虑枚举答案然后检验枚举的值是否正确。若满足单调性，则满足使用二分法的条件。把这里的枚举换成二分，就变成了「二分答案」。

### Luogu P1873 砍树

伐木工人米尔科需要砍倒 $M$ 米长的木材。这是一个对米尔科来说很容易的工作，因为他有一个漂亮的新伐木机，可以像野火一样砍倒森林。不过，米尔科只被允许砍倒单行树木。

米尔科的伐木机工作过程如下：米尔科设置一个高度参数 $H$（米），伐木机升起一个巨大的锯片到高度 $H$，并锯掉所有的树比 $H$ 高的部分（当然，树木不高于 $H$ 米的部分保持不变）。米尔科就得到树木被锯下的部分。

例如，如果一行树的高度分别为 20，15，10，17，米尔科把锯片升到 15 米的高度，切割后树木剩下的高度将是 15，15，10，15，而米尔科将从第 1 棵树得到 5 米木材，从第 4 棵树得到 2 米木材，共 7 米木材。

米尔科非常关注生态保护，所以他不会砍掉过多的木材。这正是他尽可能高地设定伐木机锯片的原因。你的任务是帮助米尔科找到伐木机锯片的最大的整数高度 $H$，使得他能得到木材至少为 $M$ 米。即，如果再升高 1 米锯片，则他将得不到 $M$ 米木材。

### 解题思路

我们可以在 1 到 $10^9$ 中枚举答案，但是这种朴素写法肯定拿不到满分，因为从 1 枚举到 $10^9$ 太耗时间。我们可以在 $[1, 10^9]$ 的区间上进行二分作为答案，然后检查各个答案的可行性（一般使用贪心法）。**这就是二分答案。**

### 参考代码

```cpp
int a[1000005];
int n, m;

bool check(int k) {  // 检查可行性，k 为锯片高度
  long long sum = 0;
  for (int i = 1; i <= n; i++)       // 检查每一棵树
    if (a[i] > k)                    // 如果树高于锯片高度
      sum += (long long)(a[i] - k);  // 累加树木长度
  return sum >= m;                   // 如果满足最少长度代表可行
}

int find() {
  int l = 1, r = 1e9 + 1;   // 因为是左闭右开的，所以 10^9 要加 1
  while (l + 1 < r) {       // 如果两点不相邻
    int mid = (l + r) / 2;  // 取中间值
    if (check(mid))         // 如果可行
      l = mid;              // 升高锯片高度
    else
      r = mid;  // 否则降低锯片高度
  }
  return l;  // 返回左边值
}

int main() {
  cin >> n >> m;
  for (int i = 1; i <= n; i++) cin >> a[i];
  cout << find();
  return 0;
}
```

看完了上面的代码，你肯定会有两个疑问：

1. 为何搜索区间是左闭右开的？

   因为搜到最后，会这样（以合法的最大值为例）：


图 1.14

然后会


图 1.15

合法的最小值恰恰相反。

2. 为何返回左边值？

   同上。

# 三分法

## 引入

如果需要求出单峰函数的极值点，通常使用二分法衍生出的三分法求单峰函数的极值点。

**为什么不通过求导函数的零点来求极值点？**

客观上，求出导数后，通过二分法求出导数的零点（由于函数是单峰函数，其导数在同一范围内的零点是唯一的）得到单峰函数的极值点是可行的。

但首先，对于一些函数，求导的过程和结果比较复杂。

其次，某些题中需要求极值点的单峰函数并非一个单独的函数，而是多个函数进行特殊运算得到的函数（如求多个单调性不完全相同的一次函数的最小值的最大值）。此时函数的导函数可能是分段函数，且在函数某些点上可能不可导。

**注意**

只要函数是单峰函数，三分法既可以求出其最大值，也可以求出其最小值。为行文方便，除特殊说明外，下文中均以求单峰函数的最小值为例。

三分法与二分法的基本思想类似，但每次操作需在当前区间 $[l,r]$（下图中除去虚线范围内的部分）内任取两点 $lmid, rmid (lmid < rmid)$（下图中的两蓝点）。如下图，如果 $f(lmid) < f(rmid)$，则在 $[rmid, r]$（下图中的红色部分）中函数必然单调递增，最小值所在点（下图中的绿点）必然不在这一区间内，可舍去这一区间。反之亦然。

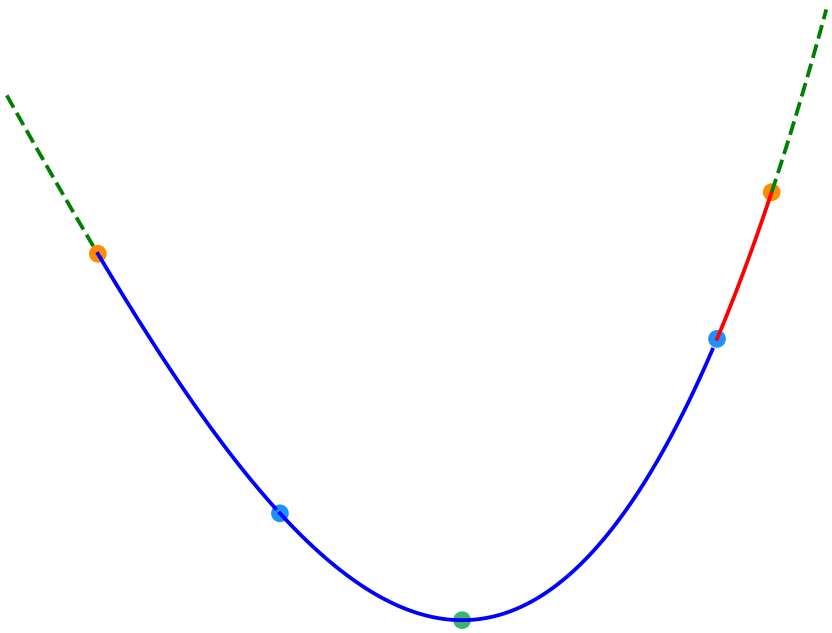

图 1.16

**注意**

在计算 $lmid$ 和 $rmid$ 时，需要防止数据溢出的现象出现。

三分法每次操作会舍去两侧区间中的其中一个。为减少三分法的操作次数，应使两侧区间尽可能大。因此，每一次操作时的 $lmid$ 和 $rmid$ 分别取 $mid - \varepsilon$ 和 $mid + \varepsilon$ 是一个不错的选择。

# 实现

## 伪代码

**Input.** A range $[l, r]$ meaning that the domain of $f(x)$.
**Output.** The maximum value of $f(x)$ and the value of $x$ at that time .
**Method.**
**while** $r - l > \varepsilon$
    $mid \leftarrow \frac{lmid+rmid}{2}$
    $lmid \leftarrow mid - \varepsilon$
    $rmid \leftarrow mid + \varepsilon$
    **if** $f(lmid) < f(rmid)$
        $r \leftarrow mid$
    **else**
        $l \leftarrow mid$

C++

```
while (r - l > eps) {
  mid = (lmid + rmid) / 2;
  lmid = mid - eps;
  rmid = mid + eps;
  if (f(lmid) < f(rmid))
    r = mid;
  else
    l = mid;
}
```

# 例题

**洛谷 P3382 - 【 模板 】 三分法**

给定一个 $N$ 次函数和范围 $[l, r]$，求出使函数在 $[l, x]$ 上单调递增且在 $[x, r]$ 上单调递减的唯一的 $x$ 的值。

**解题思路**

本题要求求 $N$ 次函数在 $[l, r]$ 取最大值时自变量的值，显然可以使用三分法。

**参考代码**

=== "C++"

```cpp
#include <cmath>
#include <cstdio>
using namespace std;

const double eps = 0.0000001;
int N;
double l, r, A[20], mid, lmid, rmid;

double f(double x) {
  double res = (double)0;
```

```
  for (int i = N; i >= 0; i--) res += A[i] * pow(x, i);
  return res;
}

int main() {
  scanf("%d%lf%lf", &N, &l, &r);
  for (int i = N; i >= 0; i--) scanf("%lf", &A[i]);
  while (r - l > eps) {
    mid = (l + r) / 2;
    lmid = mid - eps;
    rmid = mid + eps;
    if (f(lmid) > f(rmid))
      r = mid;
    else
      l = mid;
  }
  printf("%6lf", l);
  return 0;
}
```

=== "Python"

```python
n, l, r = map(float, input().split())
a = [i * float(j) for i, j in enumerate(input().split()[::-1])][1:]
while r - l > 1e-6:
    mid = (l + r) / 2
    if sum(mid ** i * j for i, j in enumerate(a)) < 0:
        r = mid
    else:
        l = mid
print(l)
```

## 习题

- Uva 1476 - Error Curves[3]
- Uva 10385 - Duathlon[4]
- UOJ 162 -【清华集训 2015】灯泡测试[5]
- 洛谷 P7579 -「RdOI R2」称重 (weigh)[6]

# 分数规划

参见：分数规划

分数规划通常描述为下列问题：每个物品有两个属性 $c_i$，$d_i$，要求通过某种方式选出若干个，使得 $\frac{\sum c_i}{\sum d_i}$ 最大或最小。

经典的例子有最优比率环、最优比率生成树等等。

分数规划可以用二分法来解决。

## 参考资料与注释

[1] std::lower_bound

[2] std::upper_bound

[3] Uva 1476 - Error Curves

[4] Uva 10385 - Duathlon

[5] UOJ 162 -【清华集训 2015】灯泡测试

[6] 洛谷 P7579 -「RdOI R2」称重（weigh）

# 1.10 倍增

**Authors:** Ir1d, ShadowsEpic, Fomalhauthmj, siger-young, MingqiHuang, Xeonacid, hsfzLZH1, orzAtalod, NachtgeistW

本页面将简要介绍倍增法。

## 定义

倍增法（英语：binary lifting），顾名思义就是翻倍。它能够使线性的处理转化为对数级的处理，大大地优化时间复杂度。

这个方法在很多算法中均有应用，其中最常用的是 RMQ 问题和求 LCA（ 最近公共祖先 ）。

## 应用

### RMQ 问题

参见：RMQ 专题

RMQ 是 Range Maximum/Minimum Query 的缩写，表示区间最大（最小）值。使用倍增思想解决 RMQ 问题的方法是 ST 表。

### 树上倍增求 LCA

参见：最近公共祖先

## 例题

### 题 1

**例题**

如何用尽可能少的砝码称量出 $[0, 31]$ 之间的所有重量？（只能在天平的一端放砝码）

**解题思路**

答案是使用 1 2 4 8 16 这五个砝码，可以称量出 $[0,31]$ 之间的所有重量。同样，如果要称量 $[0,127]$ 之间的所有重量，可以使用 1 2 4 8 16 32 64 这七个砝码。每次我们都选择 2 的整次幂作砝码的重量，就可以使用极少的砝码个数量出任意我们所需要的重量。

为什么说是极少呢？因为如果我们要量出 $[0,1023]$ 之间的所有重量，只需要 10 个砝码，需要量出 $[0,1048575]$ 之间的所有重量，只需要 20 个。如果我们的目标重量翻倍，砝码个数只需要增加 1。这叫「对数级」的增长速度，因为砝码的所需个数与目标重量的范围的对数成正比。

## 题 2

**例题**

给出一个长度为 $n$ 的环和一个常数 $k$，每次会从第 $i$ 个点跳到第 $(i+k) \bmod n+1$ 个点，总共跳了 $m$ 次。每个点都有一个权值，记为 $a_i$，求 $m$ 次跳跃的起点的权值之和对 $10^9+7$ 取模的结果。

数据范围：$1 \le n \le 10^6$，$1 \le m \le 10^{18}$，$1 \le k \le n$，$0 \le a_i \le 10^9$。

**解题思路**

这里显然不能暴力模拟跳 $m$ 次。因为 $m$ 最大可到 $10^{18}$ 级别，如果暴力模拟的话，时间承受不住。

所以就需要进行一些预处理，提前整合一些信息，以便于在查询的时候更快得出结果。如果记录下来每一个可能的跳跃次数的结果的话，不论是时间还是空间都难以承受。

那么应该如何预处理呢？看看第一道例题。有思路了吗？

回到本题。我们要预处理一些信息，然后用预处理的信息尽量快的整合出答案。同时预处理的信息也不能太多。所以可以预处理出以 2 的整次幂为单位的信息，这样的话在预处理的时候只需要处理少量信息，在整合的时候也不需要大费周章。

在这题上，就是我们预处理出从每个点开始跳 1、2、4、8 等等步之后的结果（所处点和点权和），然后如果要跳 13 步，只需要跳 1+4+8 步就好了。也就是说先在起始点跳 1 步，然后再在跳了之后的终点跳 4 步，再接着跳 8 步，同时统计一下预先处理好的点权和，就可以知道跳 13 步的点权和了。

对于每一个点开始的 $2^i$ 步，记录一个 `go[i][x]` 表示第 $x$ 个点跳 $2^i$ 步之后的终点，而 `sum[i][x]` 表示第 $x$ 个点跳 $2^i$ 步之后能获得的点权和。预处理的时候，开两重循环，对于跳 $2^i$ 步的信息，我们可以看作是先跳了 $2^{i-1}$ 步，再跳 $2^{i-1}$ 步，因为显然有 $2^{i-1}+2^{i-1}=2^i$。即我们有 `sum[i][x] = sum[i-1][x]+sum[i-1][go[i-1][x]]`，且 `go[i][x] = go[i-1][go[i-1][x]]`。

当然还有一些实现细节需要注意。为了保证统计的时候不重不漏，我们一般预处理出「左闭右开」的点权和。亦即，对于跳 1 步的情况，我们只记录该点的点权和；对于跳 2 步的情况，我们只记录该点及其下一个点的点权和。相当于总是不将终点的点权和计入 sum。这样在预处理的时候，只需要将两部分的点权和直接相加就可以了，不需要担心第一段的终点和第二段的起点会被重复计算。

这题的 $m \le 10^{18}$，虽然看似恐怖，但是实际上只需要预处理出 65 以内的 $i$，就可以轻松解决，比起暴力枚举快了很多。用行话讲，这个做法的 时间复杂度 是预处理 $\Theta(n \log m)$，查询每次 $\Theta(\log m)$。

**参考代码**

```cpp
#include <cstdio>
using namespace std;

const int mod = 1000000007;

int modadd(int a, int b) {
  if (a + b >= mod) return a + b - mod;  // 减法代替取模，加快运算
```

```
  return a + b;
}

int vi[1000005];

int go[75][1000005];  // 将数组稍微开大以避免越界，小的一维尽量定义在前面
int sum[75][1000005];

int main() {
  int n, k;
  scanf("%d%d", &n, &k);
  for (int i = 1; i <= n; ++i) {
    scanf("%d", vi + i);
  }

  for (int i = 1; i <= n; ++i) {
    go[0][i] = (i + k) % n + 1;
    sum[0][i] = vi[i];
  }

  int logn = 31 - __builtin_clz(n);  // 一个快捷的取对数的方法
  for (int i = 1; i <= logn; ++i) {
    for (int j = 1; j <= n; ++j) {
      go[i][j] = go[i - 1][go[i - 1][j]];
      sum[i][j] = modadd(sum[i - 1][j], sum[i - 1][go[i - 1][j]]);
    }
  }

  long long m;
  scanf("%lld", &m);

  int ans = 0;
  int curx = 1;
  for (int i = 0; m; ++i) {
    if (m & (1ll << i)) {  // 参见位运算的相关内容，意为 m 的第 i 位是否为 1
      ans = modadd(ans, sum[i][curx]);
      curx = go[i][curx];
      m ^= 1ll << i;  // 将第 i 位置零
    }
  }

  printf("%d\n", ans);
}
```

# 1.11 构造

> **Authors:** leoleoasd, yzxoi

本页面将简要介绍构造题这类题型。

## 引入

构造题是比赛中常见的一类题型。

从形式上来看，问题的答案往往具有某种规律性，使得在问题规模迅速增大的时候，仍然有机会比较容易地得到答案。

这要求解题时要思考问题规模增长对答案的影响，这种影响是否可以推广。例如，在设计动态规划方法的时候，要考虑从一个状态到后继状态的转移会造成什么影响。

# 特点

构造题一个很显著的特点就是高自由度，也就是说一道题的构造方式可能有很多种，但是会有一种较为简单的构造方式满足题意。看起来是放宽了要求，让题目变的简单了，但很多时候，正是这种高自由度导致题目没有明确思路而无从下手。

构造题另一个特点就是形式灵活，变化多样。并不存在一个通用解法或套路可以解决所有构造题，甚至很难找出解题思路的共性。

# 例题

下面将列举一些例题帮助读者体会构造题的一些思想内涵，给予思路上的启发。建议大家深入思考后再查看题解，也欢迎大家参与分享有趣的构造题。

## 例题 1

Codeforces Round #384 (Div. 2) C.Vladik and fractions

构造一组 $x,y,z$，使得对于给定的 $n$，满足 $\frac{1}{x}+\frac{1}{y}+\frac{1}{z}=\frac{2}{n}$

**解题思路**

从样例二可以看出本题的构造方法。

显然 $n,n+1,n(n+1)$ 为一组合法解。特殊地，当 $n=1$ 时，无解，这是因为 $n+1$ 与 $n(n+1)$ 此时相等。

至于构造思路是怎么产生的，大概就是观察样例加上一点点数感了吧。此题对于数学直觉较强的人来说并不难。

## 例题 2

Luogu P3599 Koishi Loves Construction

Task1：试判断能否构造并构造一个长度为 $n$ 的 $1\dots n$ 的排列，满足其 $n$ 个前缀和在模 $n$ 的意义下互不相同

Taks2：试判断能否构造并构造一个长度为 $n$ 的 $1\dots n$ 的排列，满足其 $n$ 个前缀积在模 $n$ 的意义下互不相同

**解题思路**

对于 task1：

当 $n$ 为奇数时，无法构造出合法解；

当 $n$ 为偶数时，可以构造一个形如 $n,1,n-2,3,\cdots$ 这样的数列。

首先，我们可以发现 $n$ 必定出现在数列的第一位，否则 $n$ 出现前后的两个前缀和必然会陷入模意义下相等的尴尬境地；

然后，我们考虑构造出整个序列的方式：

考虑通过构造前缀和序列的方式来获得原数列，可以发现前缀和序列两两之间的差在模意义下不能相等，因为前缀和序列的差分序列对应着原来的排列。

因此我们尝试以前缀和数列在模意义下为

$$0,1,-1,2,-2,\cdots$$

这样的形式来构造这个序列，不难发现它完美地满足所有限制条件。

对于 task2：

当 $n$ 为除 4 以外的合数时，无法构造出合法解

当 $n$ 为质数或 4 时，可以构造一个形如 $1,\frac{2}{1},\frac{3}{2},\cdots,\frac{n-1}{n-2},n$ 这样的数列

先考虑什么时候有解：

显然，当 $n$ 为合数时无解。因为对于一个合数来说，存在两个比它小的数 $p,q$ 使得 $p\times q\equiv 0\pmod n$，如 $(3\times 6)\%9=0$。那么，当 $p,q$ 均出现过后，数列的前缀积将一直为 0，故合数时无解。特殊地，我们可以发现 $4=2\times 2$，无满足条件的 $p,q$，因此存在合法解。

我们考虑如何构造这个数列：

和 task1 同样的思路，我们发现 1 必定出现在数列的第一位，否则 1 出现前后的两个前缀积必然相等；而 $n$ 必定出现在数列的最后一位，因为 $n$ 出现位置后的所有前缀积在模意义下都为 0。手玩几组样例以后发现，所有样例中均有一组合法解满足前缀积在模意义下为 $1,2,3,\cdots,n$，因此我们可以构造出上文所述的数列来满足这个条件。那么我们只需证明这 $n$ 个数互不相同即可。

我们发现这些数均为 $1\cdots n-2$ 的逆元 $+1$，因此各不相同，此题得解。

## 例题 3

**AtCoder Grand Contest 032 B**

给定一个整数 $N$，试构造一个节点数为 $N$ 无向图。令节点编号为 $1\ldots N$，要求其满足以下条件：

- 这是一个简单连通图。
- 存在一个整数 $S$ 使得对于任意节点，与其相邻节点的下标和为 $S$。

保证输入数据有解。

**解题思路**

手玩一下 $n=3,4,5$ 的情况，我们可以找到一个构造思路。

构造一个完全 $k$ 分图，保证这 $k$ 部分和相等。则每个点的 $S$ 均相等，为 $\frac{(k-1)\sum_{i=1}^{n}i}{k}$。

如果 $n$ 为偶数，那么我们可以前后两两配对，即 $\{1,n\},\{2,n-1\}\cdots$

如果 $n$ 为奇数，那么我们可以把 $n$ 单拿出来作为一组，剩余的 $n-1$ 个两两配对，即 $\{n\},\{1,n-1\},\{2,n-2\}\cdots$

这样构造出的图在 $n\geq 3$ 时连通性易证，在此不加赘述。

此题得解。

## 例题 4

**BZOJ 4971「Lydsy1708 月赛」记忆中的背包**

经过一天辛苦的工作，小 Q 进入了梦乡。他脑海中浮现出了刚进大学时学 01 背包的情景，那时还是大一萌新的小 Q 解决了一道简单的 01 背包问题。这个问题是这样的：

给定 $n$ 个物品，每个物品的体积分别为 $v_1,v_2,...,v_n$，请计算从中选择一些物品（也可以不选），使得总体积恰好为 $w$ 的方案数。因为答案可能非常大，你只需要输出答案对 $P$ 取模的结果。

因为长期熬夜刷题，他只看到样例输入中的 $w$ 和 $P$，以及样例输出是 $k$，看不清到底有几个物品，也看不清

每个物品的体积是多少。直到梦醒，小 Q 也没有看清 $n$ 和 $v$，请写一个程序，帮助小 Q 一起回忆曾经的样例输入。

**解题思路**

这道题是自由度最高的构造题之一了。这就导致了没有头绪，难以入手的情况。

首先，不难发现模数是假的。由于我们自由构造数据，我们一定可以让方案数不超过模数。

通过奇怪的方式，我们想到可以通过构造 $n$ 个代价为 1 的小物品和几个代价大于 $\frac{w}{2}$ 的大物品。

由于大物品只能取一件，所以每个代价为 $x$ 的大物品对方案数的贡献为 $\binom{n}{w-x}$。

令 $f_{i,j}$ 表示有 $i$ 个 1，方案数为 $j$ 的最小大物品数。

用 dp 预处理出 $f$，通过计算可知只需预处理 $i \leq 20$ 的所有值即可。

此题得解。