数据结构与算法

Data Structure and Algorithm

极夜酱

目录

Chapter 1 计算复杂性理论

1.1 时间/空间复杂度

1.1.1 算法（Algorithm）

算法是解决问题的一种方法，由一系列的步骤组成。算法有 5 个特点：

1. 有穷性（finiteness）：算法必须在有限个步骤后终止。
2. 确定性（definiteness）：算法的每个步骤必须有确切的定义。
3. 输入项（input）：一个算法有 0 个或多个输入。
4. 输出项（output）：一个算法有 1 个或多个输出，没有输出的算法是毫无意义的。
5. 可行性（effectiveness）：算法中执行的任何步骤都可以被分解为基本的可执行操作。

1.1.2 时间复杂度（Time Complexity）

算法有高效的，也有拙劣的。衡量算法好坏的标准有时间复杂度和空间复杂度。

想象一个场景：老板让小灰和大黄完成一个需求，两人都完成并交付了各自的代码，代码的功能是一样的。但是，大黄的代码运行一次要花 100ms，占用 5MB 内存；小灰的代码运行一次要花 100s，占用 500MB 内存。

"小灰，收拾东西走人，明天不用来上班了！"

小灰虽然也按照老板的要求实现了功能，但他的代码存在很严重的问题：运行时间长、占用空间大。

算法的时间复杂度是指算法中基本运算的执行次数，其中基本运算指的是加减法、交换、比较等操作。

算法的效率应该取决于算法本身，与机器无关。分析算法运行效率时应该考虑的是运行时间与输入规模之间的关系。通过计算算法中基本运算的执行次数，可以得到一个关于输入规模 n 的函数。

时间复杂度一般采用大 O 表示法，表示算法的运行时间与输入规模之间的增长关系。常见的时间复杂度包括 $O(1)$、$O(logn)$、$O(n)$、$O(nlogn)$、$O(n^2)$、$O(2^n)$、$O(n!)$ 等。

时间复杂度需要满足以下原则：

1. 如果运行时间是常量级，则用 $O(1)$ 表示。
2. 只保留时间函数中的最高阶项。
3. 如果最高阶项存在，则省去最高阶项前的系数。

大 O 符号

大 O 符号用于表示时间复杂度的上界（最坏情况），即算法的阶不会超过大 O 符号中的函数 $f(n)$。

$$0 \le f(n) \le cg(n) \tag{1.1}$$

$$\begin{aligned} f(n) &= n^2 + n \\ &= O(n^2) \\ &= O(n^3) \end{aligned}$$

大 Ω 符号

大 Ω 符号用于表示时间复杂度的下界（最好情况），即算法的阶不会低于大 Ω 符号中的函数 $f(n)$。

$$0 \le cg(n) \le f(n) \tag{1.2}$$

$$\begin{aligned} f(n) &= n^2 + n \\ &= \Omega(n^2) \\ &= \Omega(100n) \end{aligned}$$

小 o 符号

小 o 符号用于表示时间复杂度的上界，即算法的阶一定低于小 o 符号中的函数 $f(n)$。

$$0 \le f(n) < cg(n) \tag{1.3}$$

$$\begin{aligned} f(n) &= n^2 + n \\ &= o(n^3) \end{aligned}$$

小 ω 符号

小 ω 符号用于表示时间复杂度的下界，即算法的阶一定高于小 ω 符号中的函数 $f(n)$。

$$0 \le cg(n) < f(n) \tag{1.4}$$

$$\begin{aligned} f(n) &= n^2 + n \\ &= \omega(n^2) \end{aligned}$$

Θ 符号

若 $f(n) = O(g(n))$ 且 $f(n) = \Omega(g(n))$，则称 $f(n)$ 的阶与 $g(n)$ 的阶相等：

$$f(n) = \Theta(g(n)) \tag{1.5}$$

$$\begin{aligned} f(n) &= n^2 + n \\ g(n) &= 100n^2 \\ &= \Theta(n^2) \end{aligned}$$

百钱买百鸡

公鸡 5 文钱 1 只，母鸡 3 文钱 1 只，小鸡 1 文钱 3 只，如果用 100 文钱买 100 只鸡，那么公鸡、母鸡和小鸡各应该买多少只？

```c
void buy(int n, int money) {
    for (int x = 0; x <= n / 5; x++) {
        for (int y = 0; y <= n / 3; y++) {
            int z = n - x - y;
            if (z > 0 && z % 3 == 0 && 5*x + 3*y + z/3 == money) {
                printf("x = %d, y = %d, z = %d\n", x, y, z);
            }
        }
    }
}
```

1.1.3 空间复杂度（Space Complexity）

算法占用的内存空间自然是越小越好，空间复杂度是对一个算法在运行过程中临时占用存储空间大小的量度，它同样使用了大 O 表示法。

正所谓鱼和熊掌不可兼得，很多时候不得不在时间复杂度和空间复杂度之间进行取舍。绝大多数时候，时间复杂度更为重要，宁可多分配一些内存空间，也要提升程序的执行速度。

1.1.4 均摊时间复杂度（Amortized Time Complexity）

均摊时间复杂度用于分析一组操作中，虽然某个操作的时间复杂度很高，但是经过若干次操作后，这组操作的平均时间复杂度较低的情况。

动态数组

```java
public void add(T element) {
    if (size == capacity) {
        capacity *= 2;
        T[] newArr = (T[]) new Object[capacity];
        System.arraycopy(arr, 0, newArr, 0, size);
        arr = newArr;
    }
    arr[size++] = element;
}
```

这段代码实现了往数组中添加数据的功能。在数组没有满的情况下，直接将数据添加到数组末尾，时间复杂度为 $O(1)$。

但是当数组已满时，需要对数组创建一个更大的数组，将原数组中的数据复制到新数组中，然后再将新数据添加到数组的末尾，这个过程的时间复杂度为 $O(n)$。

假设数组的容量为 n，每执行 n 次 add() 操作，才会进行一次扩容。那么，可以将这一次的 $O(n)$ 的时间均摊到前面的 n 次操作中，这样 add() 操作的均摊时间复杂度就是 $O(1)$。

1.2 递推方程

1.2.1 递推（Recurrence）

递归算法无法直接根据语句的执行次数计算出时间复杂度，但是可以通过递推方程迭代展开进行求解。

求和

```
int sum(int n) {
    if (n <= 0) {
        return 0;
    }
    return sum(n - 1) + n;
}
```

$$T(n) = \begin{cases} 1 & n = 0 \\ T(n-1) + 1 & n \geq 1 \end{cases} \tag{1.6}$$

$$\begin{aligned} T(n) &= T(n-1) + 1 \\ &= [T(n-2) + 1] + 1 \\ &= [[T(n-3) + 1] + 1] + 1 \\ &= \dots \\ &= T(n-k) + k \end{aligned}$$

$$n - k = 0 \Rightarrow n = k$$

$$\begin{aligned} T(n) &= T(n-n) + n \\ &= T(0) + n \\ &= 1 + n \\ &= O(n) \end{aligned}$$

插入排序

```
void insert_sort(int arr[], int n) {
    if (n <= 1) {
        return;
    }
    insert_sort(arr, n-1);
    int last = arr[n-1];
    int j = n - 2;
    while (j >= 0 && arr[j] > last) {
        arr[j+1] = arr[j];
        j--;
    }
    arr[j+1] = last;
}
```

$$T(n) = \begin{cases} 1 & n <= 1 \\ T(n-1) + n & n \geq 1 \end{cases} \tag{1.7}$$

$$\begin{aligned} T(n) &= T(n-1) + n \\ &= [T(n-2) + n] + n \\ &= [[T(n-3) + n] + n] + n \\ &= \dots \\ &= T(n-k) + nk \end{aligned}$$

$$n - k = 0 \Rightarrow n = k$$

$$\begin{aligned} T(n) &= T(n-n) + n^2 \\ &= T(0) + n^2 \\ &= 1 + n^2 \\ &= O(n^2) \end{aligned}$$

汉诺塔

有三根柱子 A、B、C，A 柱子上从下到上套有 n 个圆盘，要求将 A 柱子上的圆盘移动到 C 柱子上。每次只能移动一个圆盘，且大圆盘始终不能叠在小圆盘上面。

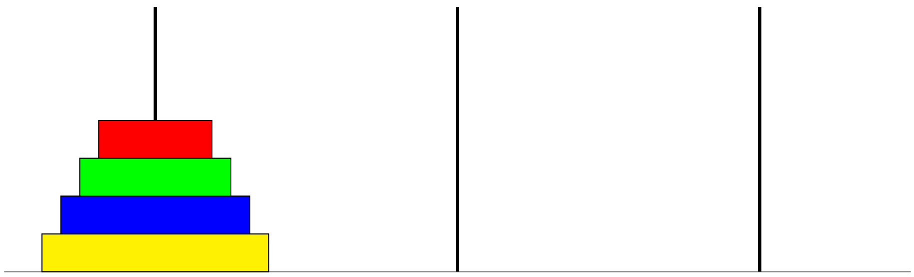

图 1.1: 汉诺塔

递归算法求解汉诺塔问题：

1. 将前 n - 1 个圆盘从 A 柱借助于 C 柱搬到 B 柱。
2. 将最后一个圆盘直接从 A 柱搬到 C 柱。
3. 将 n - 1 个圆盘从 B 柱借助于 A 柱搬到 C 柱。

```
def hanoi(n, A, B, C):
    if n == 1
        move(1, A, C)
    else
        hanoi(n-1, A, C, B)
        move(n, A, C)
        hanoi(n-1, B, A, C)
```

$$T(n) = \begin{cases} 1 & n = 1 \\ 2T(n-1)+1 & n > 1 \end{cases} \tag{1.8}$$

$$
\begin{aligned}
T(n) &= 2T(n-1)+1 \\
&= 2[2T(n-2)+1]+1 \\
&= 2[2[2T(n-3)+1]+1]+1 \\
&= \dots \\
&= 2^k * T(n-k) + 2^k - 1
\end{aligned}
$$

$$
n-k=1 \Rightarrow n=k+1
$$

$$
\begin{aligned}
T(n) &= 2^{n-1} * T(1) + 2^{n-1} - 1 \\
&= 2^{n-1} + 2^{n-1} - 1 \\
&= 2^n - 1 \\
&= O(2^n)
\end{aligned}
$$

假设每次移动花费 1 秒，解决一个 64 层的汉诺塔问题大约需要 5800 亿年。

1.3 P=NP?

1.3.1 旅行商问题（Traveling Salesman Problem）

小灰最近在工作中遇到了一个棘手的问题。公司正在开发一个物流项目，其中一个需求是为快递员自动规划送快递的路线。

有一个快递员，要分别给三家顾客送快递，他自己到达每个顾客家的路程各不相同，每个顾客之间的路程也各不相同。那么想要把快递依次送达这三家，并最终回到起点，哪一条路线所走的总距离是最短的呢?

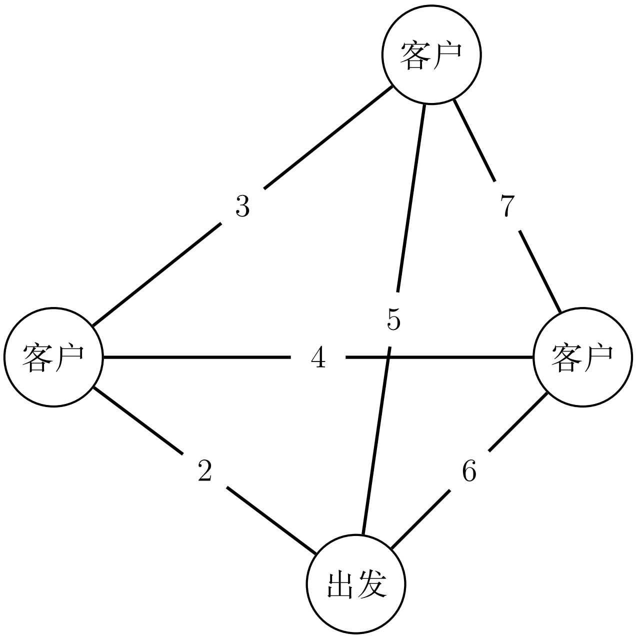

图 1.2: 快递客户路线

为了寻求最优路线，小灰研究了好久，可惜还是没有找到一个高效率的解决方案。不只是小灰，当前的计算机科学家们也没有找到一个行之有效的优化方案，这是典型的旅行商问题。

有一个商品推销员，要去若干个城市推销商品。该推销员从一个城市出发，需要经过所有城市后，回到出发地。每个城市之间都有道路连通，且距离各不相同，推销员应该如何选择路线，使得总行程最短呢?

这个问题看起来很简单，却很难找到一个真正高效的求解算法。其中最容易想到

的，是使用穷举法把所有可能的路线穷举出来，计算出每一条路线的总行程。通过排列组合，从所有路线中找出总行程最短的路线。显然，这个方法的时间复杂度是 $O(n!)$，随着城市数量的增长，花费的运算时间简直不可想象！

后来，人们想出了许多相对优化的解决方案，比如动态规划法和分枝定界法等。但是，这些算法的时间复杂度仍然是指数级的，并没有让性能问题得到根本的解决。

像这样的问题有很多，旅行商问题仅仅是其中的一例。对于这类问题统称为 NP 问题。

1.3.2 P=NP?

算法的设计与分析在计算机科学领域有着重要的应用背景。1966 ~ 2005 年期间，Turing 奖获奖 50 人，其中 10 人以算法设计，7 人以计算理论、自动机和复杂性研究领域的杰出贡献获奖。计算复杂性理论中的 P=NP? 问题是世界七大数学难题之一。

一些常见的算法的时间复杂度，例如二分查找法 $O(logn)$、归并排序 $O(nlogn)$、Floyd 最短路径 $O(n^3)$ 等，都可以用 $O(n^k)$ 表示。这些算法都是多项式时间算法，即能在多项式时间内解决问题。这类问题被称为 P 问题（Polynomial）。

人们常说，能用钱解决的问题都不是问题。在计算机科学家眼中，能用多项式时间解决的问题都不是问题。

然而，世间还存在许多变态的问题，是无法（至少是暂时无法）在多项式时间内解决的，比如一些算法的时间复杂度是 $O(2^n)$，甚至 $O(n!)$。随着问题规模 n 的增长，计算量的增长速度是非常恐怖的。这类问题被称为 NP 问题（Non-deterministic Polynomial），意思是“不确定是否能在多项式时间内解决”。

有些科学家认为，所有的 NP 问题终究都可以在多项式时间内解决，只是我们暂时还没有找到方法；也有些科学家认为，某些 NP 问题永远无法在多项式时间内

解决。这个业界争论用 P=NP? 这个公式来表达。

这还不算完，在所有的 NP 问题当中，还存在着一些大 BOSS，被称为 NPC 问题。

1.3.3 NPC

这里所说的 NPC 问题可不是游戏当中的 NPC。要想理解 NPC 问题，需要先了解归约的概念。

归约（reduction）可以简单理解成问题之间的转化。例如问题是一个一元一次方程的求解问题 $Q : 3x + 6 = 12$，这个问题可以转化成一个一元二次方程 $Q' : 0x^2 + 3x + 6 = 12$。

问题 Q 并不比问题 Q' 难解决，只要有办法解决 Q'，就一定能够解决 Q。对于这种情况，可以说问题 Q 归约于问题 Q'。

同时，这种归约可以逐级传递，比如问题 A 归约于问题 B，问题 B 归约于问题 C，问题 C 归约于问题 D，那么可以说问题 A 归约于问题 D。

在 NP 问题之间，也可以存在归约关系。把众多的 NP 问题层层归约，必定会得到一个或多个终极问题，这些归约的终点就是所谓的 NPC 问题 (NP-Complete)。旅行商问题被科学家证明属于 NPC 问题。

俗话说擒贼先擒王，只要有朝一日，我们能够找到 NPC 问题的多项式时间算法，就能够解决掉所有的 NP 问题！但遗憾的是，至今还没有人能够找到可行的方法，很多人认为这个问题是无解的。

回到最初的快递路线规划问题，既然是工程问题，与其钻牛角尖寻求最优解，不如用小得多的代价寻求次优解。最简单的办法是使用贪心算法，先选择距离起点最近的地点 A，再选择距离 A 最近的地点 B，以此类推，每一步都保证局部最优。这样规划出的路线未必是全局最优，但平均情况下也不会比最优方案差多少。

Chapter 2 数组

2.1 查找算法

2.1.1 顺序查找（Sequence Search）

顺序查找也称线性查找，是一种按照序列原有顺序进行遍历比较的查找算法。

对于任意一个序列以及一个需要查找的元素（关键字），将关键字与序列中元素依次比较，直到找出与给定关键字相同的元素，或者将序列中的元素与其都比较完为止。若某个元素的值与关键字相等，则查找成功；如果直到最后一个元素，元素的值和关键字比较都不等时，则查找不成功。

最好的情况就是在第一个位置就找到，算法时间复杂度为 $O(1)$。最坏情况是关键字不存在，需要进行 n 次比较，时间复杂度为 $O(n)$。

顺序查找

```
template <typename T>
int sequence_search(T *arr, int n, T key) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == key) {
            return i;
        }
    }
    return -1;
}
```

2.1.2 二分查找（Binary Search）

二分查找法也称折半查找，是一种效率较高的查找方法。折半查找要求线性表必须采用顺序存储结构，而且表中元素按关键字有序排列。

算法思想是假设表中元素是按升序排列，将表中间位置的关键字与查找关键字比较，如果两者相等，则查找成功；否则利用中间位置记录将表分成前、后两个子表，如果中间位置的关键字大于查找关键字，则进一步查找前一子表，否则进一步查找后一子表。重复以上过程，直到找到满足条件的记录，使查找成功，或直到子表不存在为止，此时查找不成功。

二分查找法的时间复杂度为 $O(logn)$。

二分查找

```cpp
template <typename T>
int binary_search(T *arr, int n, T key) {
    int start = 0;
    int end = n - 1;

    while (start <= end) {
        int mid = (start + end) / 2;
        if (arr[mid] == key) {
            return mid;
        } else if (arr[mid] < key) {
            start = mid + 1;
        } else {
            end = mid - 1;
        }
    }
    return -1;
}
```

2.2 数组

2.2.1 数组（Array）

数组是数据结构中最简单的结构，很多编程语言都内置数组。数组是有限个相同类型的变量所组成的集合，数组中的每一个变量被称为元素。

创建数组时会在内存中划分出一块连续的内存，将数据按顺序进行存储，数组中的每一个元素都有着自己的下标（index），下标从 0 开始。

对于数组来说，读取元素是最简单的操作。由于数组在内存中顺序存储，所以只要给出数组的下标，就可以读取到对应位置的元素。像这种根据下标读取元素的方式叫作随机读取。数组读取元素的时间复杂度是 $O(1)$。

数组的劣势体现在插入和删除元素方面。由于数组元素连续紧密地存储在内存中，插入、删除元素都会导致大量元素被迫移动，影响效率。总的来说，数组所适合的是读操作多、写操作少的场景。

2.2.2 插入元素

在数组中插入元素存在 3 种情况：

0	1	2	3	4	5	6	7
data	data	data	data	data	data	data	data

图 2.1: 数组

尾部插入

直接把插入的元素放在数组尾部的空闲位置即可。

中间插入

首先把插入位置及后面的元素向后移动，腾出位置，再把要插入的元素放入该位置上。

扩容

数组的长度在创建时就已经确定了，要实现数组的扩容，只能创建一个新数组，长度是旧数组的 2 倍，再把旧数组中的元素全部复制过去。

数组插入元素最好情况是尾部插入，无需移动任何元素，时间复杂度为 $O(1)$。最坏情况是在第一个位置插入，这样就需要移动后面所有 $n-1$ 个元素，时间复杂度为 $O(n)$。

插入元素

```
public void add(T elem) {
    if (size == capacity) {
        resize();
    }
    data[size++] = elem;
}

public void add(int index, T elem) throws IndexOutOfBoundsException {
    if (index < 0 || index > size) {
        throw new IndexOutOfBoundsException("Index out of bounds");
    }

    if (size == capacity) {
        resize();
    }

    for (int i = size - 1; i >= index; i--) {
        data[i + 1] = data[i];
    }
```

```java

    data[index] = elem;
    size++;
}
```

2.2.3 删除元素

数组的删除操作与插入操作过程相反，如果被删除的元素位于数组中间，其后的元素都需要向前挪动一位。

删除元素

```java
public T remove(int index) throws IndexOutOfBoundsException {
    if (index < 0 || index >= size) {
        throw new IndexOutOfBoundsException("Index out of bounds");
    }

    T elem = data[index];
    for (int i = index; i < size - 1; i++) {
        data[i] = data[i + 1];
    }
    size--;
    return elem;
}
```

数组的删除操作，由于涉及元素的移动，时间复杂度为 $O(n)$。

Chapter 3 链表

3.1 链表种类

3.1.1 单向链表（Singly Linked List）

为避免元素的移动，采用线性表的另一种存储方式：链式存储结构。链表是一种在物理上非连续、非顺序的数据结构，由若干结点（node）所组成。

单向链表的每一个结点又包含两部分，一部分是存放数据的数据域 data，另一部分是指向下一个结点的指针域 next。结点可以在运行时动态生成。

```python
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

链表的第一个结点被称为头结点，最后一个节点被称为尾结点，尾结点的 next 指针指向 NULL。

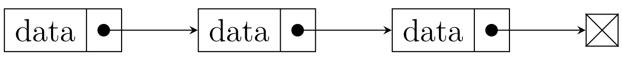

图 3.1: 单向链表

与数组按照下标来随机寻找元素不同，对于链表的其中一个结点 A，只能根据结点 A 的 next 指针来找到该结点的下一个结点 B，再根据结点 B 的 next 指针找到下一个节点 C……

数组在内存中的存储方式是顺序存储，链表在内存中的存储方式则是随机存储。链表采用了见缝插针的方式，每一个结点分布在内存的不同位置，依靠 next 指针关联起来。这样可以灵活有效地利用零散的碎片空间。

3.1.2 双向链表（Doubly Linked List）

那么，通过链表的一个结点，如何能快速找到它的前一个结点呢？要想让每个结点都能回溯到它的前置结点，可以使用双向链表。

双向链表比单向链表稍微复杂一点，它的每一个结点除了拥有 data 和 next 指针，还拥有指向前置结点的 prev 指针。

```python
class Node:
    def __init__(self, data):
        self.data = data
        self.prev = None
        self.next = None
```

图 3.2: 双向链表

单向链表只能从头到尾遍历，只能找到后继，无法找到前驱，因此遍历的时候不会死循环。而双向链表需要多分配一个指针的存储空间，每个结点有两个指针，分别指向直接前驱和直接后继。

3.1.3 循环链表（Circular Linked List）

除了单向链表和双向链表以外，还有循环链表。对于单向循环链表，尾结点的 next 指针指向头结点。对于双向循环链表，尾结点的 next 指针指向头结点，并且头结点的 prev 指针指向尾结点。

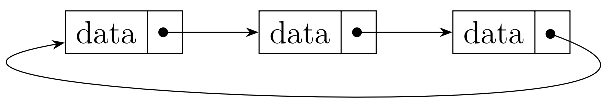

图 3.3: 单向循环链表

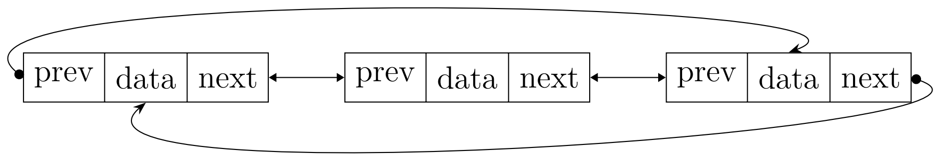

图 3.4: 双向循环链表

3.2 链表

3.2.1 查找结点

在查找元素时，链表不像数组那样可以通过下标快速进行定位，只能从头结点开始向后一个一个结点逐一查找。

链表中的数据只能按顺序进行访问，最坏的时间复杂度是 $O(n)$。

查找结点

```java
public T get(int index) throws IndexOutOfBoundsException {
    if (index < 0 || index >= size) {
        throw new IndexOutOfBoundsException("Index out of bounds");
    }

    Node<T> current = head;
    for (int i = 0; i < index; i++) {
        current = current.next;
    }
    return current.data;
}
```

3.2.2 更新结点

如果不考虑查找结点的过程，链表的更新过程会像数组那样简单，直接把旧数据替换成新数据即可。

更新结点

```java
public void set(int index, T data) throws IndexOutOfBoundsException {
    if (index < 0 || index >= size) {
        throw new IndexOutOfBoundsException("Index out of bounds");
```

```
    }

    Node<T> current = head;
    for (int i = 0; i < index; i++) {
        current = current.next;
    }
    current.data = data;
}
```

3.2.3 插入结点

链表插入结点，分为 3 种情况：

尾部插入

把最后一个结点的 next 指针指向新插入的结点。

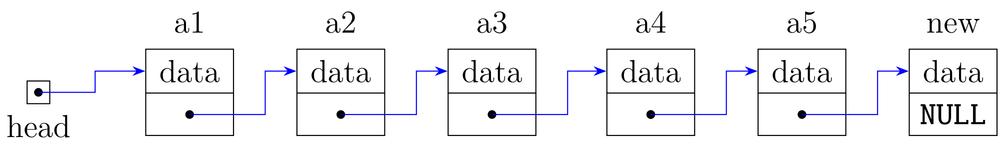

图 3.5: 尾部插入

头部插入

先把新结点的 next 指针指向原先的头结点，再把新结点设置为链表的头结点。

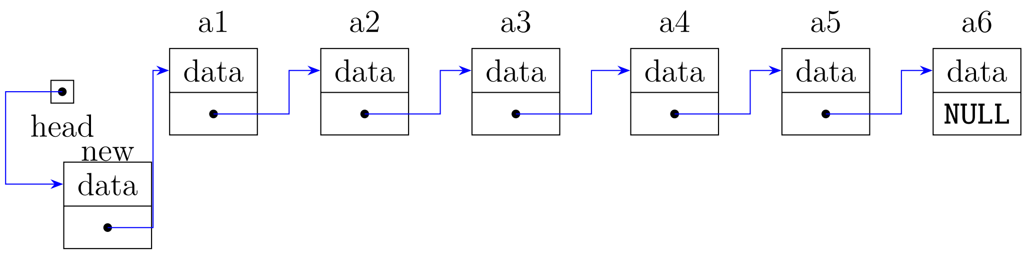

图 3.6: 头部插入

中间插入

先把新结点的 next 指针指向插入位置的结点，再将插入位置的前置结点的 next 指针指向新结点。

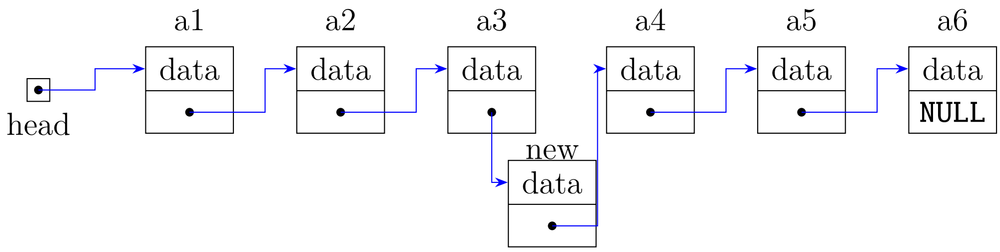

图 3.7: 中间插入

只要内存空间允许，能够插入链表的元素是无穷无尽的，不需要像数组考虑扩容的问题。如果不考虑插入之前的查找元素的过程，只考虑纯粹的插入操作，时间复杂度是 $O(1)$。

插入结点

```
public void add(T data) {
    Node<T> newNode = new Node<T>(data, null);
    if (isEmpty()) {
        head = newNode;
        tail = newNode;
    } else {
        tail.next = newNode;
        tail = newNode;
    }
    size++;
}

public void add(int index, T data) throws IndexOutOfBoundsException {
    if (index < 0 || index > size) {
        throw new IndexOutOfBoundsException("Index out of bounds");
    }

    Node<T> newNode = new Node<T>(data, null);
```

```
    if (index == 0) {
        newNode.next = head;
        head = newNode;
    } else {
        Node<T> prev = head;
        for (int i = 0; i < index - 1; i++) {
            prev = prev.next;
        }
        newNode.next = prev.next;
        prev.next = newNode;
    }
    size++;
}
```

3.2.4 删除结点

链表的删除操作也分 3 种情况：

尾部删除

把倒数第二个结点的 next 指针指向空。

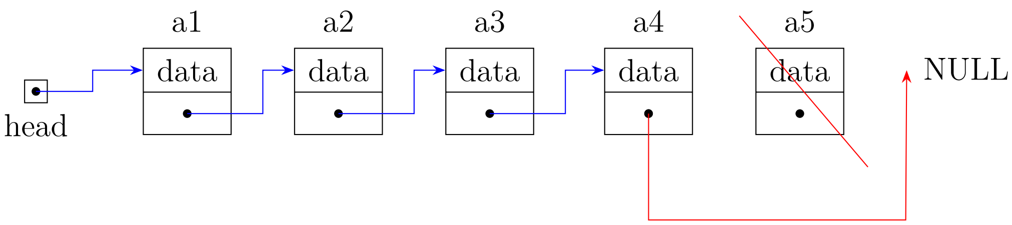

图 3.8: 尾部删除

头部删除

把链表的头结点设置为原先头结点的 next 指针。

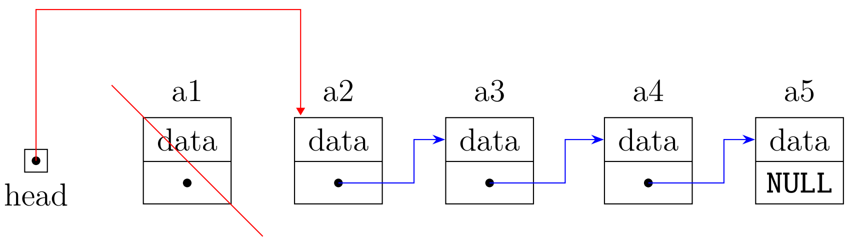

图 3.9: 头部删除

中间删除

把要删除的结点的前置结点的 next 指针，指向要删除结点的下一个结点。

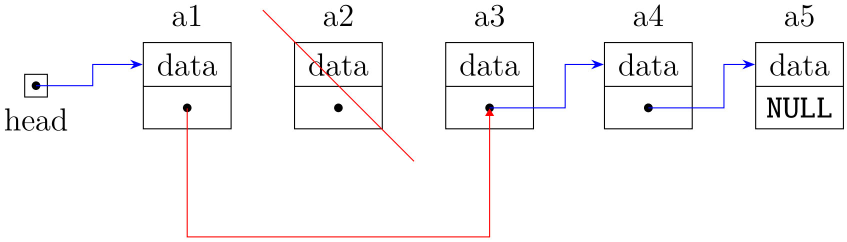

图 3.10: 中间删除

许多高级语言，如 Java，拥有自动化的垃圾回收机制，所以不用刻意去释放被删除的结点，只要没有外部引用指向它们，被删除的结点会被自动回收。

如果不考虑删除操作之前的查找的过程，只考虑纯粹的删除操作，时间复杂度是 $O(1)$。

删除结点

```
public T remove(int index) throws IndexOutOfBoundsException {
    if (index < 0 || index >= size) {
        throw new IndexOutOfBoundsException("Index out of bounds");
    }

    T data;
    if (index == 0) {
        data = head.data;
        head = head.next;
```

```java
    } else {
        Node<T> prev = head;
        for (int i = 0; i < index - 1; i++) {
            prev = prev.next;
        }
        data = prev.next.data;
        prev.next = prev.next.next;
    }
    size--;
    return data;
}
```

3.2.5 反转链表

反转一个链表需要调整链表中指针的方向。

递归反转法的实现思想是从链表的尾结点开始，依次向前遍历，遍历过程依次改变各结点的指向，即另其指向前一个结点。

而迭代反转法的实现思想非常直接，就是从当前链表的首结点开始，一直遍历至链表尾部，期间会逐个改变所遍历到的结点的指针域，另其指向前一个结点。

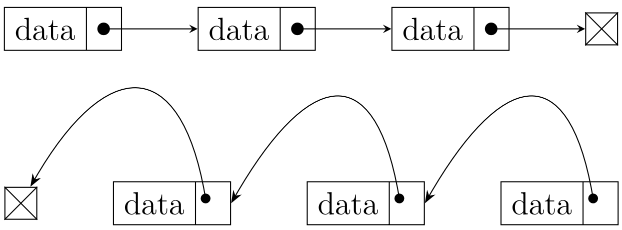

图 3.11: 反转链表

反转链表（递归）

```java
private Node<T> reverseRecursive(Node<T> current) {
    if (current == null || current.next == null) {
        return current;
```

```java
    }
    Node<T> newHead = reverse(current.next);
    current.next.next = current;
    current.next = null;
    return newHead;
}

public void reverseRecursive() {
    head = reverseRecursive(head);
}
```

反转链表（迭代）

```java
public void reverse() {
    Node<T> prev = null;
    Node<T> current = head;
    Node<T> next = null;
    while (current != null) {
        next = current.next;
        current.next = prev;
        prev = current;
        current = next;
    }
    head = prev;
}
```

Chapter 4 栈

4.1 栈

4.1.1 栈（Stack）

栈，又名堆栈，是一种运算受限的线性数据结构，栈只能在表尾进行插入和删除操作。

栈中的元素只能先进后出（FILO, First In Last Out）。最早进入栈的元素所存放的位置叫作栈底（bottom），最后进入栈的元素存放的位置叫作栈顶（top）。

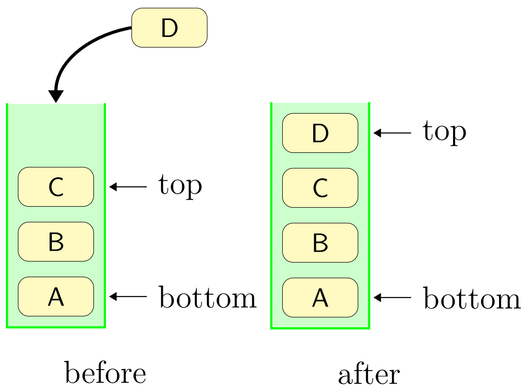

图 4.1: 栈

栈这种数据结构既可以用数组来实现，也可以用链表来实现。

4.1.2 顺序栈

使用数组方式实现的栈称为静态栈。可以根据下标来表示栈顶在数组中的位置，对于空栈，栈顶为-1。

进行入栈操作时，栈顶指针 +1；出栈时，栈顶指针-1。

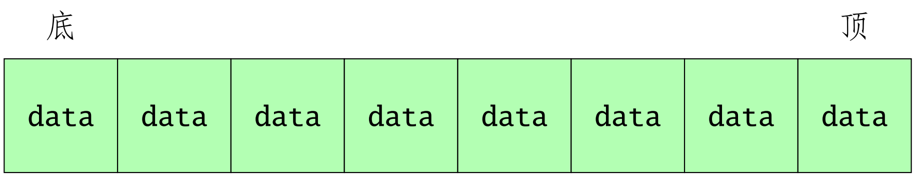

图 4.2: 顺序栈

对满栈进行入栈和对空栈进行出栈操作操作都会产生数组的越界并引起程序崩溃，称为上溢和下溢。因此使用顺序栈前需要提前声明一个数组的大小，如果数组大小不够则可能发生数组越界，如果数组太大则会浪费一定的空间。

使用数组实现的栈的执行效率会比用链表来实现的高，入栈和出栈不需要移动大量元素，只需要移动栈顶指针即可。

4.1.3 链式栈

使用链表方式实现的栈称为动态栈。通过在表头插入一个元素来实现入栈，通过删除表尾元素来实现出栈。

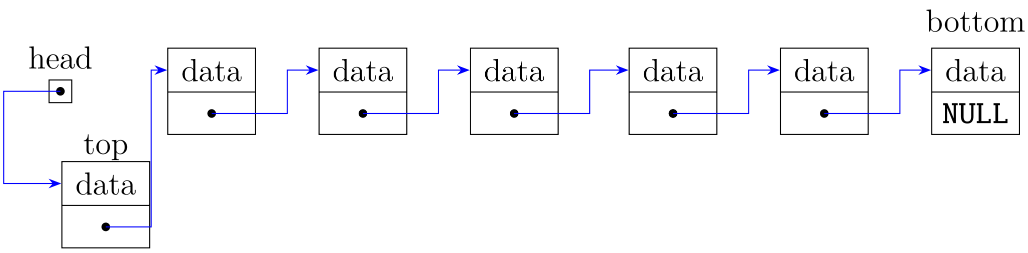

图 4.3: 链式栈

动态栈有链表的部分特性，元素与元素之间在物理存储上可以不连续，但是功能有些受限制，动态栈只能在栈顶处进行插入和删除操作，不能在栈尾或栈中间进行插入和删除操作。

动态栈的元素内存是动态分配的，避免了静态栈可能会浪费空间的问题，但是对申请和释放空间的调用开销会比较大。

4.1.4 入栈（Push）

入栈操作就是把新元素放入栈中，只允许从栈顶一侧放入元素，新元素的位置将会成为新的栈顶。最初，栈为空，栈顶的初始值为-1。每当向栈中添加元素时，栈顶指针 +1。

入栈只影响最后一个元素，不涉及元素的整体移动，所以无论是以数组还是链表实现，时间复杂度都是 $O(1)$。

入栈

```
stack_t *stack_push(stack_t *stack, T elem) {
    array_append(stack->data, elem);
    return stack;
}
```

4.1.5 出栈（Pop）

出栈操作就是把新元素从栈中弹出，只有栈顶元素才允许出栈，出栈元素的前一个元素将会成为新的栈顶。从栈中移出元素，栈顶指针-1。数组中元素的删除并非真正意义上把元素从内存中清除，出栈只需对栈顶-1 即可，后期向栈中添加元素时，新元素会将旧元素覆盖。

出栈只影响最后一个元素，不涉及元素的整体移动，所以无论是以数组还是链表实现，时间复杂度都是 $O(1)$。

出栈

```
T stack_pop(stack_t *stack) {
    return array_remove(stack->data, stack_size(stack) - 1);
}
```

4.2 括号匹配

4.2.1 括号匹配

给定一个只包括"("、")"、"["、"]"、"{" 和"}" 的字符串，判断字符串是否有效。有效字符串需满足左括号必须以正确的顺序用相同类型的右括号闭合。

判断括号的有效性可以使用栈来解决。通过遍历字符串，当遇到左括号时，会期望在后续的遍历中，有一个相同类型的右括号将其闭合。由于后遇到的左括号要先闭合，因此将这个左括号放入栈顶。当遇到右括号时，需要将一个相同类型的左括号闭合。此时可以取出栈顶的左括号并判断它们是否是相同类型的括号。如果不是，或者栈中没有左括号，那么字符串无效。在遍历结束后，如果为空栈，说明字符串中的所有左括号闭合。

注意有效字符串的长度一定为偶数，因此如果字符串的长度为奇数，可以直接返回判断出字符串无效，省去后续的遍历判断过程。

括号匹配

```
def valid_paratheses(s):
    if len(s) % 2 == 1:
        return False

    pairs = {")": "(", "]": "[", "}": "{"}
    stack = list()
    for paran in s:
        if paran in pairs:
            if not stack or stack[-1] != pairs[paran]:
                return False
            stack.pop()
        else:
            stack.append(paran)

    return not stack
```

4.3 表达式求值

4.3.1 表达式求值

逆波兰表达式是一种后缀表达式，所谓后缀就是指运算符写在运算数的后面。平常使用的算式则是一种中缀表达式，如 $(1+2)*(3+4)$，该算式的逆波兰表达式写法为 $1\ 2 + 3\ 4 + *$。

逆波兰表达式的优点在于去掉了中缀表达式中的括号后表达式无歧义，因此适合用栈操作运算。遇到数字则入栈，遇到算符则取出栈顶两个数字进行计算，并将结果压入栈中。

表达式求值

```
def calculate(expression):
    stack = []
    tokens = expression.split()

    for token in tokens:
        if token in "+-*/":
            right = stack.pop()
            left = stack.pop()
            if token == "+":
                stack.append(left + right)
            elif token == "-":
                stack.append(left - right)
            elif token == "*":
                stack.append(left * right)
            elif token == "/":
                stack.append(left / right)
        else:
            stack.append(int(token))

    return stack.pop()

```

```
expressions = [
    "1 2 +",              # 1 + 2 = 3
    "2 3 4 + *",          # 2 * (3 + 4) = 14
    "1 2 + 3 4 + *",      # (1 + 2) * (3 + 4) = 21
    "3 4 2 + * 5 *",      # 3 * (4 + 2) * 5 = 90
    "50 20 - 2 /",        # (50 - 20) / 2 = 15
]

for expression in expressions:
    print(expression, "=", calculate(expression))
```

Chapter 5 队列

5.1 队列

5.1.1 队列（Queue）

队列是一种运算受限的线性数据结构，不同于栈的先进后出（FILO），队列中的元素只能先进先出（FIFO, First In First Out）。

队列的出口端叫作队头（front），队列的入口端叫作队尾（rear）。队列只允许在队尾进行入队（enqueue），在队头进行出队（dequeue）。

与栈类似，队列既可以用数组来实现，也可以用链表来实现。其中用数组实现时，为了入队操作的方便，把队尾位置规定为最后入队元素的下一个位置。

5.1.2 入列（enqueue）

入队就是把新元素放入队列中，只允许在队尾的位置放入元素，新元素的下一个位置将会成为新的队尾。入队操作的时间复杂度是 $O(1)$。

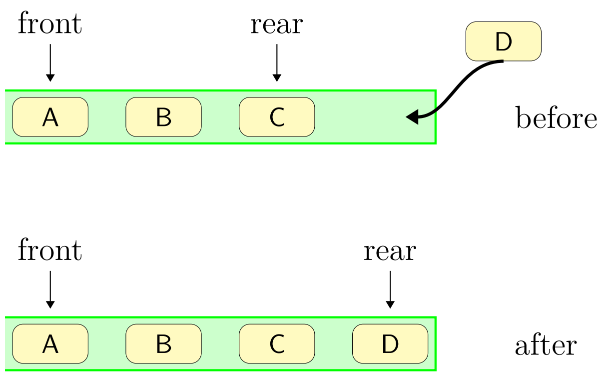

图 5.1: 入队

5.1.3 出队（dequeue）

出队就是把元素移出队列，只允许在队头一侧移出元素，出队元素的后一个元素将成为新的队头。出队操作的时间复杂度是 $O(1)$。

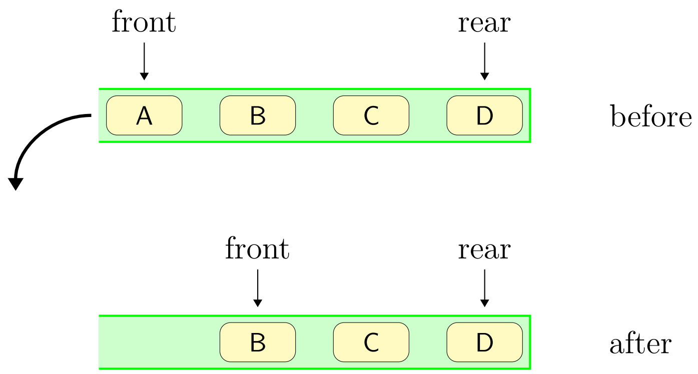

图 5.2: 出队

5.1.4 循环队列（Circular Queue）

如果不断出队，队头左边的空间就失去了作用，那队列的容量就会变得越来越小。

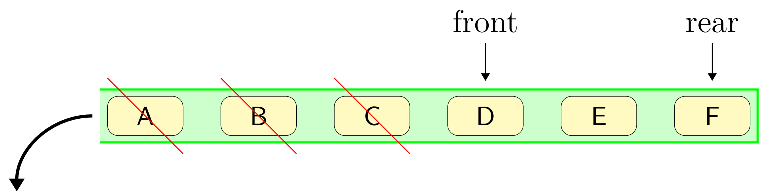

图 5.3: 队列存在的问题

用数组实现的队列可以采用循环队列的方式来维持队列容量的恒定。为充分利用空间，克服假溢出的现象，在数组不做扩容的情况下，将队列想象为一个首尾相接的圆环，可以利用已出队元素留下的空间，让队尾指针重新指回数组的首位。这样一来整个队列的元素就循环起来了。

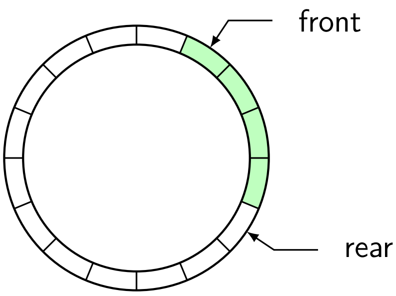

图 5.4: 循环队列

在物理存储上，队尾的位置也可以在队头之前。当再有元素入队时，将其放入数组的首位，队尾指针继续后移即可。队头和队尾互相追赶，这个追赶的过程就是入队的出队的过程。

如果队尾追上队头说明队列满了，如果队头追上队尾说明队列为空。循环队列并非真正地把数组弯曲，利用求余操作就能使队头和队尾指针不会跑出数组的范围，逻辑上实现了弯曲的效果。

假设数组的最大容量为 MAX：

- 入队时队尾指针后移：(rear + 1) % MAX
- 出队时队头指针后移：(front + 1) % MAX
- 判断队满：(rear + 1) % MAX == front
- 判断队空：front == rear

需要注意的是，队尾指针指向的位置永远空出一位，所以队列最大容量比数组长度小 1。

入队

```
queue_t *queue_enqueue(queue_t *quque, T elem) {
    queue->data[queue->rear] = elem;
    queue->rear = (queue->rear + 1) % queue->max;
    return queue;
}
```

出队

```
T queue_enqueue(queue_t *queue) {
    T elem = queue->data[queue->front];
    queue->front = (queue->front + 1) % queue->max;
    return elem;
}
```

5.2 双端队列

5.2.1 双端队列（Deque, Double Ended Queue）

双端队列是一种同时具有队列和栈的性质的数据结构，双端队列可以从其两端插入和删除元素。

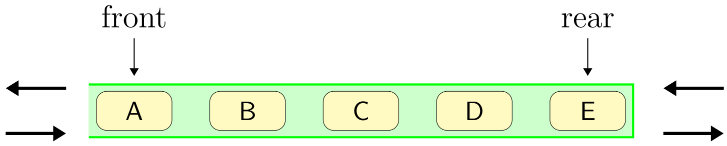

图 5.5: 双端队列

双端队列

```python
class Deque:
    def __init__(self):
        self.__data = []

    def is_empty(self):
        return self.__data == []

    def __len__(self):
        return len(self.__data)

    def push_front(self, elem):
        self.__data.insert(0, elem)

    def push_back(self, elem):
        self.__data.append(elem)

    def pop_front(self):
        return self.__data.pop(0)

    def pop_back(self):
        return self.__data.pop()

```

```
    def front(self):
        return self.__data[0]

    def back(self):
        return self.__data[-1]
```

Chapter 6 哈希表

6.1 哈希表

6.1.1 哈希表（Hash Table）

例如开发一个学生管理系统，需要有通过输入学号快速查出对应学生的姓名的功能。这里不必每次都去查询数据库，而可以在内存中建立一个缓存表，这样做可以提高查询效率。

学号	**姓名**
001121	Alice
002123	Bob
002931	Charlie
003278	Daniel

表 6.1: 学生名单

再例如需要统计一本英文书里某些单词出现的频率，就需要遍历整本书的内容，把这些单词出现的次数记录在内存中。

单词	**出现次数**
this	108
and	56
are	79
by	46

表 6.2: 词频统计

因为这些需要，一个重要的数据结构诞生了，这个数据结构就是哈希表。哈希表也称散列表，哈希表提供了键（key）和值（value）的映射关系，只要给出一个 key，就可以高效地查找到它所匹配的 value。

哈希表的时间复杂度几乎是常量 $O(1)$，即查找时间与问题规模无关。

哈希表的两项基本工作：

1. 计算位置：构造哈希函数确定关键字的存储位置。
2. 解决冲突：应用某种策略解决多个关键字位置相同的问题。

6.1.2 哈希函数（Hash Function）

哈希的基本思想是将键 key 通过一个确定的函数，计算出对应的函数值 value 作为数据对象的存储地址，这个函数就是哈希函数。

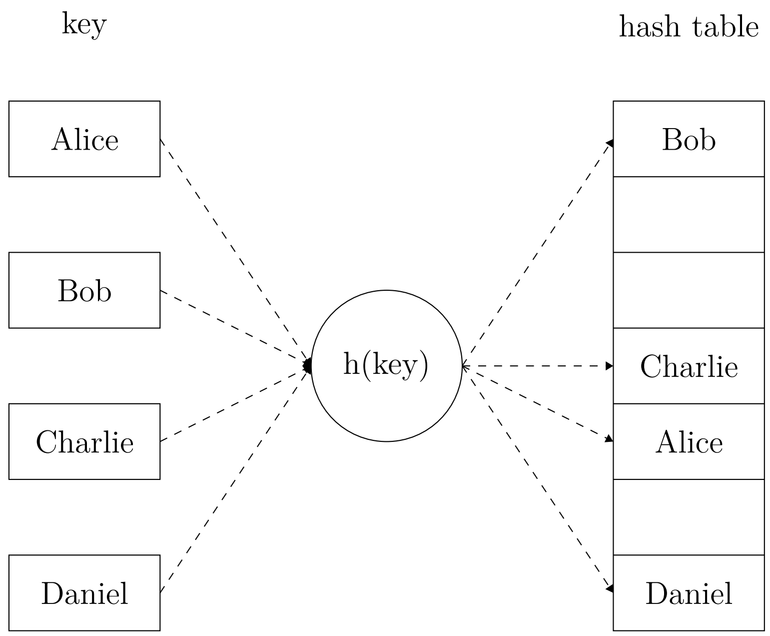

图 6.1: 哈希函数

哈希表本质上也是一个数组，可是数组只能根据下标来访问，而哈希表的 key 则是以字符串类型为主的。

在不同的语言中，哈希函数的实现方式是不一样的。假设需要存储整型变量，转化为数组的下标就不难实现了。最简单的转化方式就是按照数组长度进行取模运算。

一个好的哈希函数应该考虑两个因素：

1. 计算简单，以便提高转换速度。
2. 关键字对应的地址空间分布均匀，以尽量减少冲突。

6.1.3 数字关键字的哈希函数构造方法

对于数字类型的关键字，哈希函数有以下几种常用的构造方法：

直接定址法

取关键字的某个线性函数值为散列地址。

$$h(key) = a * key + b$$

例如根据出生年份计算人口数量 h(key) = key - 1990：

地址	出生年份	人数
0	1990	1285 万
1	1991	1281 万
2	1992	1280 万
...	...	...
10	2000	1250 万
...	...	...
21	2011	1180 万

表 6.3: 直接定址法

除留余数法

哈希函数为 h(key) = key % p，p 一般取素数。

例如 h(key) = key % 17：

地址	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
关键字	34	18	2	20			23	7	42		27	11		30		15	

表 6.4: 除留余数法

数字分析法

分析数字关键字在各位上的变化情况，取比较随机的位作为散列地址。

例如取 11 位手机号码的后 4 位作为地址 h(key) = int(key + 7)。

再例如取 18 位身份证号码中变化较为随机的位数：

1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**
3	3	0	1	0	6	1	9	9	0	1	0	0	8	0	4	1	9
省		市		区		年				月		日		辖			校验

表 6.5: 数字分析法

折叠法

把关键字分割成位数相同的几个部分，然后叠加。

例如将整数 56793542 每三位进行分割：

$$\begin{array}{rr} & 542 \\ & 793 \\ + & 056 \\ \hline = & 1319 \end{array}$$

$$h(56793542) = 319$$

平方取中法

计算关键字的平方，取中间几位。

例如整数 56793542：

$$\begin{array}{lr} & 56793542 \\ * & 56793542 \\ \hline = & 3225506412905764 \end{array}$$

$$h(56793542) = 641$$

6.1.4 字符串关键字的哈希函数构造方法

对于字符串类型的关键字，哈希函数有以下几种常用的构造方法：

ASCII 码加和法

$$h(key) = \left(\sum key[i]\right)\ mod\ N$$

但是对于某些字符串会导致严重冲突，例如：a3、b2、c1 或 eat、tea 等。

移位法

取前 3 个字符移位。

$$h(key) = \left(key[0] \times 27^2 + key[1] \times 27|key[2]\right)\ mod\ N$$

对于一些字符串仍然会冲突，例如 string、strong、street、structure 等。

一个有效的改进是涉及关键字中所有 n 个字符：

$$h(key) = \left(\sum_{i=0}^{n-1} key[n-i-1] \times 32^i\right)\ mod\ N$$

6.2 哈希冲突

6.2.1 装填因子（Load Factor）

假设哈希表空间大小为 m，填入表中元素个数是 n，则称 $\alpha = n/m$ 为哈希表的装填因子。

当哈希表元素太多，即装填因子 α 太大时，查找效率会下降。实用最大装填因子一般取 $0.5 \le \alpha \le 0.85$。当装填因子过大时，解决的方法是加倍扩大哈希表，这个过程叫作再散列（rehashing）。

再散列的过程需要遍历原哈希表，把所有的关键字重新散列到新数组中。为什么需要重新散列呢？因为长度扩大以后，散列的规则也随之改变。经过扩容，原本拥挤的哈希表重新变得稀疏，原有的关键字也重新得到了尽可能均匀的分配。

装填因子也是影响产生哈希冲突的因素之一。当不同的关键字可能会映射到同一个散列地址上，就导致了哈希冲突（collision），即 $h(key_i) = h(key_j)$，$key_i \neq key_j$，因此需要某种冲突解决策略。

例如有 11 个数据对象的集合 {18, 23, 11, 20, 2, 7, 27, 30, 42, 15, 34, 35}，哈希表的大小为 17，哈希函数选择 h(key) = key % size。

地址	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
关键字	34	18	2	20			23	7	42		27	11		30		15	

在插入最后一个关键字 35 之前，都没有产生任何冲突。但是 h(35) = 1，位置已有对象，就导致了冲突。

常用的处理冲突的思路有两种：

1. 开放地址法（open addressing）：一旦产生了冲突，就按某种规则去寻找另一空地址。开放地址法主要有线性探测法、平方探测法（二次探测法）和双散列法。

2. 分离链接法：将相应位置上有冲突的所有关键字存储在同一个单链表中。

6.2.2 线性探测法（Linear Probing）

当产生冲突时，以增量序列 1, 2, 3, ..., n - 1 循环试探下一个存储地址。

例如序列 {47, 7, 29, 11, 9, 84, 54, 20, 30}，哈希表表长为 13，哈希函数 h(key) = key % 11，用线性探测法处理冲突。

key	**47**	**7**	**29**	**11**	**9**	**84**	**54**	**20**	**30**
h(key)	3	7	7	0	9	7	10	9	8
冲突次数	0	0	1	0	0	3	1	3	6

地址	0	1	2	3	4	5	6	7	8	9	10	11	12	Δ
插入 47				47										0
插入 7				47				7						0
插入 29				47				7	29					1
插入 11	11			47				7	29					0
插入 9	11			47				7	29	9				0
插入 84	11			47				7	29	9	84			3
插入 54	11			47				7	29	9	84	54		1
插入 20	11			47				7	29	9	84	54	20	3
插入 30	11	30		47				7	29	9	84	54	20	6

表 6.6: 线性探测法

线性探测法的缺陷在于容易出现聚集现象。

6.2.3 平方探测法（Quadratic Probing）

平方探测法也称为二次探测法，以增量序列 $1^2, -1^2, 2^2, -2^2, \ldots, q^2, -q^2$ ($q \leq \lfloor N/2 \rfloor$) 循环试探下一个存储地址。

例如序列 {47, 7, 29, 11, 9, 84, 54, 20, 30}，哈希表表长为 11，哈希函数 h(key) = key % 11，用平方探测法处理冲突。

key	47	7	29	11	9	84	54	20	30
h(key)	3	7	7	0	9	7	10	9	8
冲突次数	0	0	1	0	0	2	0	3	3

地址	0	1	2	3	4	5	6	7	8	9	10	△
插入 47				47								0
插入 7				47				7				0
插入 29				47				7	29			1
插入 11	11			47				7	29			0
插入 9	11			47				7	29	9		0
插入 84	11			47			84	7	29	9		-1
插入 54	11			47			84	7	29	9	54	0
插入 20	11		20	47			84	7	29	9	54	4
插入 30	11	30	20	47			84	7	29	9	54	4

表 6.7: 平方探测法

但是只要还有空间，平方探测法就一定能找到空闲位置吗?

例如对于以下哈希表，插入关键字 11，哈希函数 h(key) = key % 5，用平方探测法处理冲突。

下标	0	1	2	3	4
key	5	6	7		

表 6.8: 平方探测法存在的问题

对关键字 11 进行平方探测的结果一直在下标 0 和 2 之间波动，永远无法达到其它空的位置。

但是有定理证明，如果哈希表长度是满足 $4k + 3$ ($k \in Z^+$) 形式的素数时，平方探测法就可以探查到整个哈希表空间。

6.2.4 双散列探测法（Double Hashing）

设定另一个哈希函数 $h_2(key)$，探测序列为 $h_2(key)$，$2h_2(key)$，$3h_2(key), \ldots$。

探测序列应该保证所有的散列存储单元都应该能够被探测到，选择以下形式有良好的效果：

$$h_2(key) = p - (key \ \% \ p) \ \ (p < N \wedge p, N \in \text{素数})$$

6.2.5 分离链接法

分离链接法也称拉链法、链地址法，将相应位置上有冲突的所有关键字存储在同一个单链表中。

例如关键字序列为 {47, 7, 29, 11, 16, 92, 22, 8, 3, 50, 37, 89, 94, 21}，哈希函数 h(key) = key % 11，用分离链接法处理冲突。

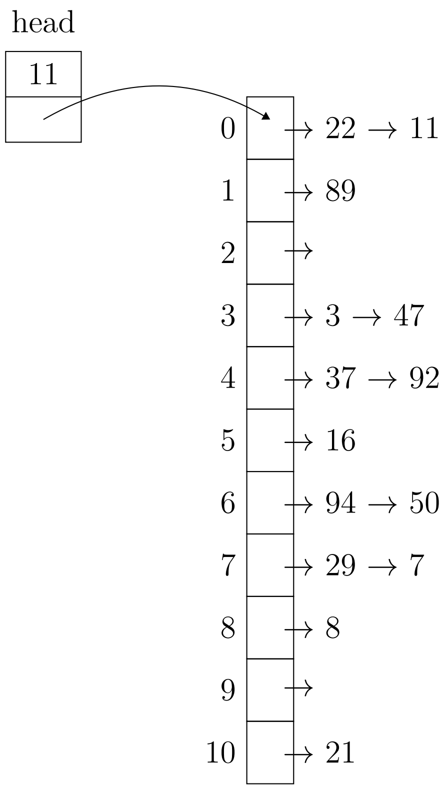

图 6.2: 分离链接法

6.2.6 性能分析

哈希表的平均查找长度（ASL, Average Search Length）用来度量哈希表查找效率。关键字的比较次数，取决于产生冲突的多少。影响产生冲突多少有三个因素：

1. 哈希函数是否均匀
2. 处理冲突的方法
3. 哈希表的装填因子 α

合理的最大装填因子 α 应该不超过 0.85，选择合适的哈希函数可以使哈希表的查找效率期望是常数 $O(1)$，它几乎与关键字的空间大小 n 无关。这是以较小的 α 为前提，因此哈希表是一个以空间换时间的结构。

哈希表的存储对关键字是随机的，因此哈希表不便于顺序查找、范围查找、最大值/最小值查找等操作。

Chapter 7 分治法

7.1 分治法

7.1.1 分治法（Divide and Conquer）

分治策略是将原问题分解为 k 个子问题，并对 k 个子问题分别求解。如果子问题的规模仍然不够小，则再划分为 k 个子问题，如此递归的进行下去，直到问题规模足够小，很容易求出其解为止。

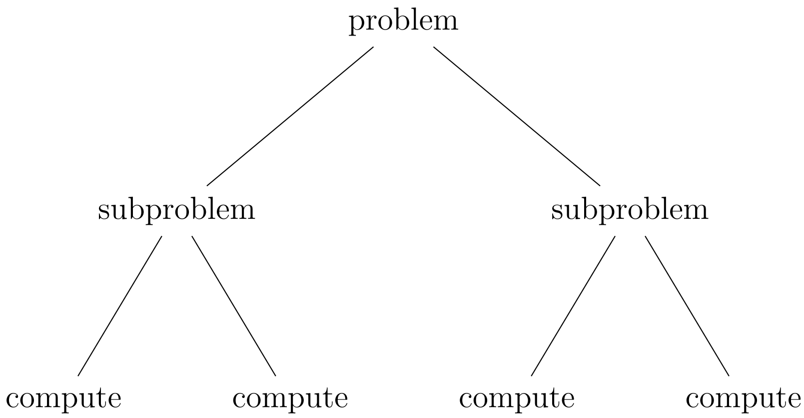

图 7.1: 分治法

将求出的小规模的问题的解合并为一个更大规模的问题的解，自底向上逐步求出原来问题的解。

分治法的适用条件有以下四点：

1. 该问题的规模缩小到一定的程度就可以容易地解决。
2. 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。
3. 利用该问题分解出的子问题的解可以合并为该问题的解。

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

人们在大量事件中发现，在用分治法设计算法时，最好使子问题的规模大致相同。即将一个问题分成大小相等的 k 个子问题的处理方法是行之有效的。这种使子问题规模大致相等的做法是出自一种平衡子问题的思想，它几乎总是比子问题规模不等的做法要好。

7.1.2 二分搜索

一个装有 16 个硬币的袋子，16 个硬币中有一个是伪造的，并且那个伪造的硬币比真的硬币要轻一些，要求找出这个伪造的硬币。只提供一台可用来比较两组硬币重量的仪器，利用这台仪器，可以知道两组硬币的重量是否相同。

算法思想是将 16 个硬币等分成 A、B 两份，将 A 放仪器的一边，B 放另一边，如果 A 袋轻，则表明伪币在 A，解子问题 A 即可，否则解子问题 B。

二分搜索每执行一次循环，待搜索数组的大小减少一半，在最坏情况下，循环被执行了 $O(logn)$ 次，循环体内运算需要 $O(1)$ 时间。因此，整个算法在最坏情况下的计算时间复杂性为 $O(logn)$。

7.2 大整数加法

7.2.1 大整数加法

如果有两个很大的整数，如何求出它们的和?

这还不简单? 直接用 long 类型存储，在程序里相加不就行了?

C/C++ 中的 int 类型能表示的范围是 $-2^{31} \sim 2^{31}-1$，unsigned 类型能表示的范围是 $0 \sim 2^{32}-1$，所以 int 和 unsigned 类型变量都不能保存超过 10 位的整数。

有时需要参与运算的数可能会远远不止 10 位，例如计算 100! 的精确值。即便使用能表示很大数值范围的 double 变量，但是由于 double 变量只有 64 位，精度也不足以表示一个超过 100 位的整数。我们称这种基本数据类型无法表示的整数为大整数。

在小学的时候，老师教我们用列竖式的方式计算两个整数的和。

$$\begin{array}{rr} & 426709752318 \\ + & 95481253129 \\ \hline = & 522191005447 \end{array}$$

不仅仅是人脑，对于计算机来说同样如此。对于大整数，我们无法一步到位直接算出结果，所以不得不把计算拆解成一个一个子步骤。

可是，既然大整数已经超出了 long 类型的范围，我们如何来存储这样的整数呢?

存放大整数最简单的方法就是使用数组，可以用数组的每一个元素存储整数的每一个数位。如果给定大整数的最长位数是 n，那么按位计算的时间复杂度是 $O(n)$。

大整数加法

```
def big_int_add(num1, num2):
```

```python
    # 其中一个数为0，直接返回另一个数
    if num1 == "0":
        return num2
    elif num2 == "0":
        return num1

    # 计算两个数中较长的整数位数
    max_len = max(len(num1), len(num2))
    # 让位数较短的整数前面补0对齐
    num1 = '0' * (max_len - len(num1)) + num1
    num2 = '0' * (max_len - len(num2)) + num2

    result = ""          # 结果
    carry = 0            # 保存进位
    # 从右往左逐位相加
    for i in range(max_len - 1, -1, -1):
        s = int(num1[i]) + int(num2[i]) + carry
        result = str(s % 10) + result
        carry = s // 10

    # 判断最高位是否有进位
    if carry > 0:
        result = str(carry) + result

    # 去除结果前面多余的0
    i = 0
    while result[i] == '0':
        i += 1
    return result[i:]
```

这种思路其实还存在一个可优化的地方。我们之前是把大整数按照每一个十进制数位来拆分，比如较大整数的长度有 50 位，那么需要创建一个 51 位的数组，数组的每个元素存储其中一位。

真的有必要把原整数拆分得那么细吗？显然不需要，只需要拆分到可以被直接计算的程度就够了。int 类型的取值范围是 $-2147483648 \sim 2147483647$，最多有 10

位整数。为了防止溢出，可以把大整数的每 9 位作为数组的一个元素，进行加法运算。如此一来，占用空间和运算次数，都被压缩了 9 倍。

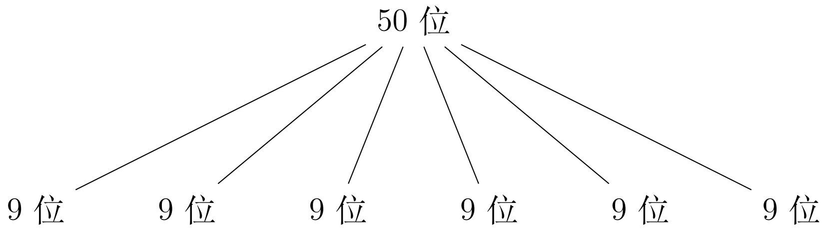

图 7.2: 大整数加法优化

在 Java 中，工具类 BigInteger 和 BidDecimal 的底层实现也是把大整数拆分成数组进行运算，与此处的思路大体类似。

7.3 大整数乘法

7.3.1 逐位相乘

起初对于大整数乘法，认为只要按照大整数相加的思路稍微做一下变形，就可以轻松实现。但是随着深入的学习，才发现事情并没有那么简单。如果沿用大整数加法的思路，通过列竖式求解⋯⋯

$$
\begin{array}{r}
93281 \\
\times \qquad 2034 \\
\hline
373124 \\
279843 \\
000000 \\
186562 \\
\hline
189733554
\end{array}
$$

乘法竖式的计算过程可以大体分为两步：

1. 整数 B 的每一个数位和整数 A 所有数位依次相乘，得到中间结果。
2. 所有中间结果相加，得到最终结果。

这样的做法确实可以实现大整数乘法，由于两个大整数的所有数位都需要一一彼此相乘。如果整数 A 的长度为 m，整数 B 的长度为 n，那么时间复杂度就是 $O(m * n)$。如果两个大整数的长度接近，那么时间复杂度也可以写为 $O(n^2)$。

那么有没有优化方法，可以让时间复杂度低于 $O(n^2)$ 呢?

7.3.2 分治法

利用分治法可以简化问题的规模，可以把大整数按照数位拆分成两部分。

$$\text{整数 } 1 = \underbrace{81325}_{A}\underbrace{79076}_{B}$$

$$\text{整数 } 2 = \underbrace{9213}_{C}\underbrace{52184}_{D}$$

即：

$$整数\ 1 = A \times 10^5 + B$$
$$整数\ 2 = C \times 10^5 + D$$

如果把两个大整数的长度抽象为 m 和 n，那么：

$$整数\ 1 = A \times 10^{m/2} + B$$
$$整数\ 2 = C \times 10^{n/2} + D$$

因此：

$$\begin{aligned}
&整数\ 1 \times 整数\ 2 \\
&= (A \times 10^{m/2} + B) \times (C \times 10^{n/2} + D) \\
&= AC \times 10^{\frac{m+n}{2}} + AD \times 10^{\frac{n}{2}} + BC \times 10^{\frac{m}{2}} + BD
\end{aligned}$$

如此一来，原本长度为 n 的大整数的 1 次乘积，被转化成了长度为 $n/2$ 的大整数的 4 次乘积。

通过递归把大整数不断地对半拆分，一直拆分到可以直接计算为止。

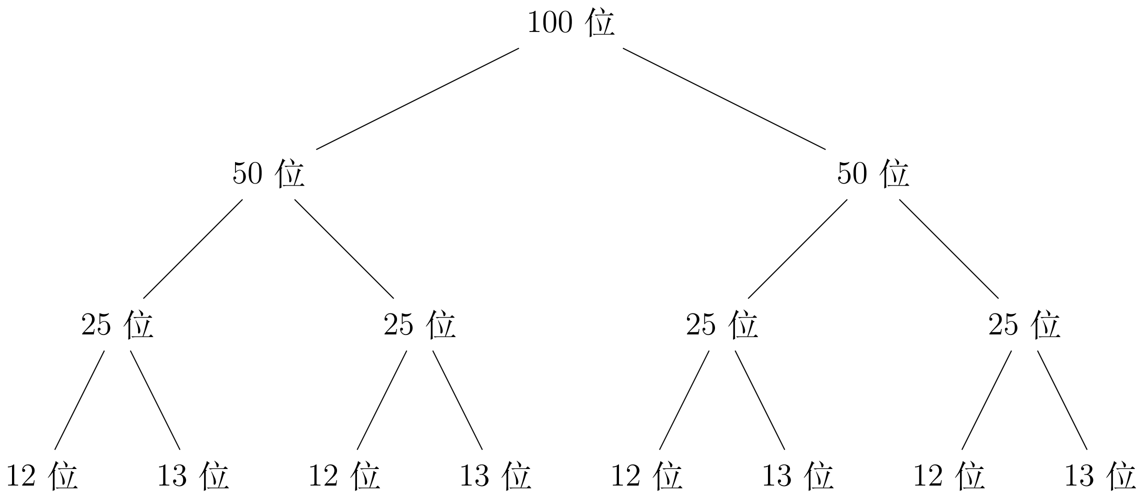

图 7.3: 大整数拆分

但是先别高兴地太早，这个方法真的提高了效率吗？

$$T(n)=\begin{cases} O(1) & n=1 \\ 4T(n/2)+O(n) & n>1 \end{cases}$$

求解得到：

$$T(n)=O(n^2)$$

闹了半天，时间复杂度还是 $O(n^2)$ 啊，白高兴一场。但是努力的方向并没有白费，这里还是有优化的方法的。

7.3.3 优化方法

在大整数乘法运算中，如果只简单地利用分治法将大整数的位置减半，并不能降低时间复杂度的阶。分治法把两个大整数相乘到的问题转化为四个较小整数相乘，性能的瓶颈仍旧在乘法上。

分治算法的时间复杂度方程为 $W(n)=aW(n/b)+f(n)$，其中 a 为子问题数，n/b 为子问题规模，$f(n)$ 为划分与合并工作量。当 a 较大、b 较小、$f(n)$ 不大时，$W(n)=\Theta(n^{log_b a})$。减少 a 是降低 $W(n)$ 的阶的一种途径。利用子问题的依赖问题，可以使某些子问题的解通过组合其它子问题的解而得到。

那么，怎样才能减少乘法运算的次数呢？哪怕由四次乘法变成三次乘法也好呀。通过对之前的乘法等式做一些调整，可以减少乘法的次数。

$$\begin{aligned}
&\text{整数 } 1\times\text{整数 } 2\\
&=(A\times 10^{n/2}+B)\times(C\times 10^{n/2}+D)\\
&=AC\times 10^{n}+AD\times 10^{n/2}+BC\times 10^{n/2}+BD\\
&=AC\times 10^{n}+(AD+BC)\times 10^{n/2}+BD\\
&=AC\times 10^{n}+(AD-AC-BD+BC+AC+BD)\times 10^{n/2}+BD\\
&=AC\times 10^{n}+((A-B)(D-C)+AC+BD)\times 10^{n/2}+BD
\end{aligned}$$

这样一来，原本的 4 次乘法和 3 次加法，转变成了 3 次乘法和 6 次加法。

骗人！最后式子里明明包含五次乘法啊！

AC 出现了两次，BD 也出现了两次，这两个乘积分别只需计算一次就行了，所以总共只需要三次乘法。

$$T(n) = \begin{cases} O(1) & n = 1 \\ 3T(n/2) + O(n) & n > 1 \end{cases}$$

求解得到：

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

大整数乘法

```python
def big_int_mul(num1, num2):
    # 有一个为空，结果为0
    if not num1 or not num2:
        return '0'
    # 终止条件
    elif len(num1) == 1 and len(num2) == 1:
        return str(int(num1) * int(num2))

    mid1 = len(num1) // 2
    mid2 = len(num2) // 2

    # 将num1分成两部分
    a = num1[:mid1]
    b = num1[mid1:]
    # 将num2分成两部分
    c = num2[:mid2]
    d = num2[mid2:]

    m = len(b)       # m次幂
    n = len(d)       # n次幂
```

```python

    # 分治计算，分别补上幂次
    x1 = big_int_mul(a, c) + '0' * (m + n)
    x2 = big_int_mul(b, c) + '0' * n
    x3 = big_int_mul(a, d) + '0' * m
    x4 = big_int_mul(b, d)

    # 将计算结果根据最长的补零，方便之后直接相加
    max_len = max(len(x1), len(x2), len(x3), len(x4))
    x1 = '0' * (max_len - len(x1)) + x1
    x2 = '0' * (max_len - len(x2)) + x2
    x3 = '0' * (max_len - len(x3)) + x3
    x4 = '0' * (max_len - len(x4)) + x4

    # 计算x1 + x2 + x3 + x4的值，也就是原问题的解
    result = ""
    carry = 0           # 保存进位
    for i in range(max_len - 1, -1, -1):
        s = int(x1[i]) + int(x2[i])
            + int(x3[i]) + int(x4[i])
            + carry
        result = str(s % 10) + result
        carry = s // 10
    # 判断是否存在进位
    if carry > 0:
        result = str(carry) + result

    # 去除结果前面多余的0
    i = 0
    while i < len(result) and result[i] == '0':
        i += 1
    return result[i:]
```

7.4 快速幂

7.4.1 快速幂（Fast Exponentiation）

使用传统算法计算 a^n 的时间复杂度为 $\Theta(n)$，然而利用快速幂的算法时间复杂度为 $\Theta(logn)$。

$$a^n = \begin{cases} a^{n/2} * a^{n/2} & n\text{为偶数} \\ a^{(n-1)/2} * a^{(n-1)/2} * a & n\text{为奇数} \end{cases}$$

例如计算 2^{18} 只需要 4 步即可：

$$2^{18} = 2^9 * 2^9$$
$$2^9 = 2^4 * 2^4 * 2$$
$$2^4 = 2^2 * 2^2$$
$$2^2 = 2^1 * 2$$

快速幂

```
/**
 * @brief  快速幂计算a^n
 */
int fastExp(int a, int n) {
    int result = 1;
    while(n) {
        if(n & 1) {
            result *= a;
        }
        a *= a;
        n >>= 1;
    }
    return result;
}
```

7.4.2 矩阵快速幂

Fibonacci 数列 $\{0, 1, 1, 2, 3, 5, 8, 13, 21, \dots\}$ 可以通过递归公式 $F_n = F_{n-1} + F_{n-2}$ 计算出第 n 项的值，时间复杂度为 $\Theta(n)$。但是利用矩阵快速幂的算法可以在将时间复杂度降低为 $\Theta(logn)$。

令 $M = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$，通过计算 M^n 即可计算出 F_n 的值。即：

$$\begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n$$

通过数学归纳法可以证明该性质：

当 $n = 1$ 时，

$$\begin{bmatrix} F_2 & F_1 \\ F_1 & F_0 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

当 $n \geq 2$ 时，

$$\begin{aligned} & \begin{bmatrix} F_{n+2} & F_{n+1} \\ F_{n+1} & F_n \end{bmatrix} \\ = & \begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \\ = & \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \\ = & \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n+1} \end{aligned}$$

矩阵快速幂

```python
N = 2

def matrix_multiply(a, b):
    c = [
```

```
        [0, 0],
        [0, 0]
    ]
    for i in range(N):
        for j in range(N):
            for k in range(N):
                c[i][j] += a[i][k] * b[k][j]
    return c

def matrix_fast_exp(n):
    result = [
        [1, 1],
        [1, 0]
    ]
    M = [
        [1, 1],
        [1, 0]
    ]

    while n > 0:
        if n & 1:
            result = matrix_multiply(result, M)
        M = matrix_multiply(M, M)
        n >>= 1

    return result[0][0]
```

7.5 矩阵乘法

7.5.1 矩阵乘法

假设 A 和 B 为 n 阶矩阵（$n = 2^k$），计算时，对于 C 中 n^2 个元素，每个元素都需要做 n 次乘法，因此 $W(n) = O(n^3)$。

利用简单的分治策略，可以将矩阵分块计算计算。

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

其中，

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\ C_{12} &= A_{12}B_{12} + A_{12}B_{22} \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21} \\ C_{22} &= A_{21}B_{12} + A_{21}B_{22} \end{aligned}$$

这样就把原问题转换为了 8 个子问题，递推方程为：

$$W(n) = \begin{cases} 1 & n = 1 \\ 8W(n/2) + cn^2 & n > 1 \end{cases}$$

求解得到：

$$W(n) - O(n^3)$$

简单的分治算法并不能降低时间复杂度的阶，但是矩阵乘法可以通过减少子问题的个数进行优化。

7.5.2 Strassen 矩阵乘法

让 M_1，M_2，...，M_7 分别对应矩阵乘法的 7 个子问题：

$$
\begin{aligned}
M_1 &= A_{11}(B_{12} - B_{22}) \\
M_2 &= (A_{11} + A_{12})B_{22} \\
M_3 &= (A_{21} + A_{22})B_{11} \\
M_4 &= A_{22}(B_{21} - B_{11}) \\
M_5 &= (A_{11} + A_{22})(B_{11} + B_{22}) \\
M_6 &= (A_{12} - A_{22})(B_{21} + B_{22}) \\
M_7 &= (A_{11} - A_{21})(B_{11} + B_{12})
\end{aligned}
$$

利用这些中间矩阵，可以得到结果矩阵：

$$
\begin{aligned}
C_{11} &= M_5 + M_4 - M_2 + M_6 \\
C_{12} &= M_1 + M_2 \\
C_{21} &= M_3 + M_4 \\
C_{22} &= M_5 + M_1 - M_3 - M_7
\end{aligned}
$$

在这些运算中，一共有 7 个子问题和 18 次矩阵加减法，时间复杂度为：

$$
W(n) = \begin{cases} 1 & n = 1 \\ 7W(n/2) + 18(n/2)^2 & n > 1 \end{cases}
$$

求解得到：

$$
W(n) = O(n^{log7}) \approx O(n^{2.8075})
$$

Coppersmith-Winograd 算法是目前已知最好的矩阵乘法算法，时间复杂度为 $O(n^{2.376})$。矩阵乘法可以应用在科学计算、图形处理、数据挖掘等方面，在回归、聚类、主成分分析、决策树等挖掘算法中常常涉及大规模矩阵运算。