算法和数据结构

1.1.算法复杂度

算法复杂度旨在计算在输入数据量 N 的情况下，算法的「时间使用」和「空间使用」情况；体现算法运行使用的时间和空间随「数据大小 N 」而增大的速度。

算法复杂度主要可从 时间 、空间 两个角度评价：

1. 时间： 假设各操作的运行时间为固定常数，统计算法运行的「计算操作的数量」 ，以代表算法运行所需时间；
2. 空间： 统计在最差情况下，算法运行所需使用的「最大空间」；

「输入数据大小 N 」指算法处理的输入数据量；根据不同算法，具有不同定义，例如：

- 排序算法： N 代表需要排序的元素数量；
- 搜索算法： N 代表搜索范围的元素总数，例如数组大小、矩阵大小、二叉树节点数、图节点和边数等；

接下来，本章将分别从概念定义、符号表示、常见种类、时空权衡、示例解析、示例题目等角度入手，介绍「时间复杂度」和「空间复杂度」。

1.1.1.时间复杂度

时间复杂度概念定义

根据定义，时间复杂度指输入数据大小为 N 时，算法运行所需花费的时间。需要注意：

- 统计的是算法的「计算操作数量」，而不是「运行的绝对时间」。计算操作数量和运行绝对时间呈正相关关系，并不相等。算法运行时间受到「编程语言 、计算机处理器速度、运行环境」等多种因素影响。例如，同样的算法使用 Python 或 C++ 实现、使用 CPU 或 GPU 、使用本地 IDE 或力扣平台提交，运行时间都不同。
- 体现的是计算操作随数据大小 N 变化时的变化情况。假设算法运行总共需要「 1 次操作」、「 100 次操作」，此两情况的时间复杂度都为常数级 $O(1)$ ；需要「 N 次操作」、「 $100N$ 次操作」的时间复杂度都为 $O(N)$

时间复杂度符号表示

根据输入数据的特点，时间复杂度具有「最差」、「平均」、「最佳」三种情况，分别使用 O , Θ , Ω 三种符号表示。以下借助一个查找算法的示例题目帮助理解。

题目：

输入长度为N的整数数组$nums$，判断此数组中是否有数字7，若有则返回$true$，否则返回$false$。

代码:

```java
boolean findSeven(int[] nums) {
    for (int num : nums) {
        if (num == 7)
            return true;
    }
    return false;
}
```

```python
def find_seven(nums):
    for num in nums:
        if num == 7:
            return True
    return False
```

```cpp
bool findSeven(vector<int>& nums) {
    for (int num : nums) {
        if (num == 7)
            return true;
    }
    return false;
}
```

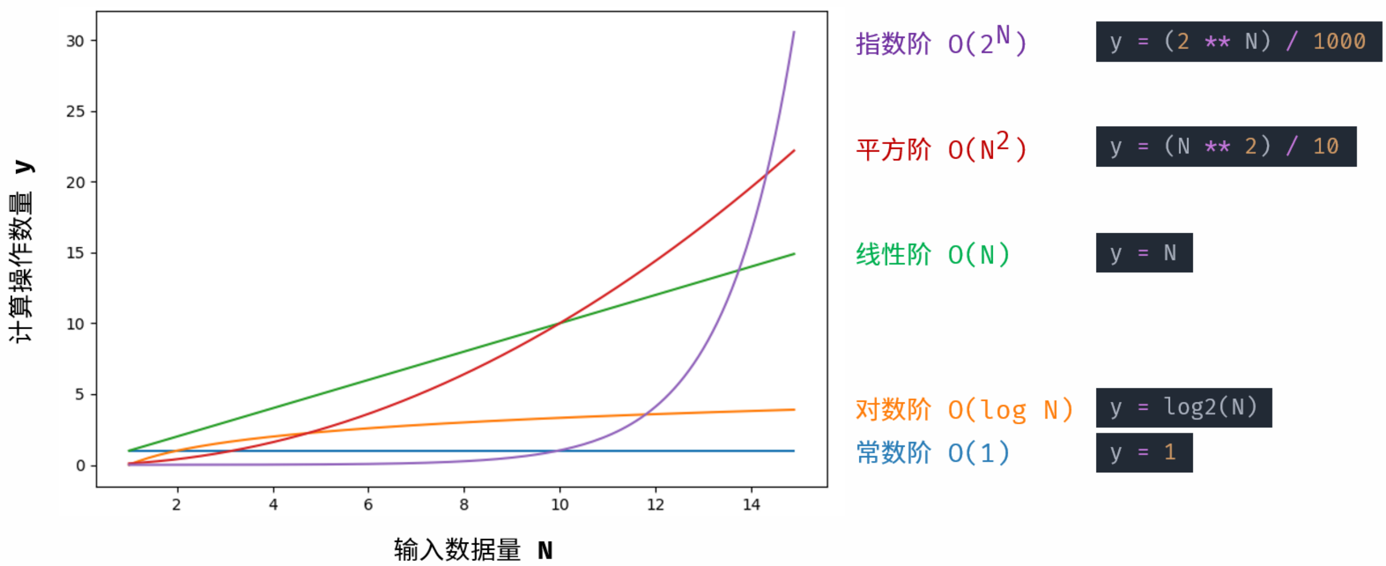
- **最佳情况** $\Omega(1)$： nums = [7, a, b, c, ...] ，即当数组首个数字为 7 时，无论 nums 有多少元素，线性查找的循环次数都为 1次；
- **最差情况** $O(N)$： nums = [a, b, c, ...]且 nums 中所有数字都不为 7 ，此时线性查找会遍历整个数组，循环 N 次；
- **平均情况** Θ : 需要考虑输入数据的分布情况，计算所有数据情况下的平均时间复杂度；例如本题目，需要考虑数组长度、数组元素的取值范围等；

大 O 是最常使用的时间复杂度评价渐进符号，下文示例与本 LeetBook 题目解析皆使用 O 。

时间复杂度常见种类

根据从小到大排列，常见的算法时间复杂度主要有：

$$O(1) < O(logN) < O(N) < O(NlogN) < O(N^2) < O(2^N) < O(N!)$$



示例解析

对于以下所有示例，设输入数据大小为 N ，计算操作数量为 $count$ 。图中每个「蓝色方块」代表一个单元计算操作。

常数 $O(1)$ ：

运行次数与 N 大小呈常数关系，即不随输入数据大小 N 的变化而变化。

```python
def algorithm(N):
    a = 1
    b = 2
    x = a * b + N
    return 1
```

```cpp
int algorithm(int N) {
    int a = 1;
    int b = 2;
    int x = a * b + N;
    return 1;
}
```

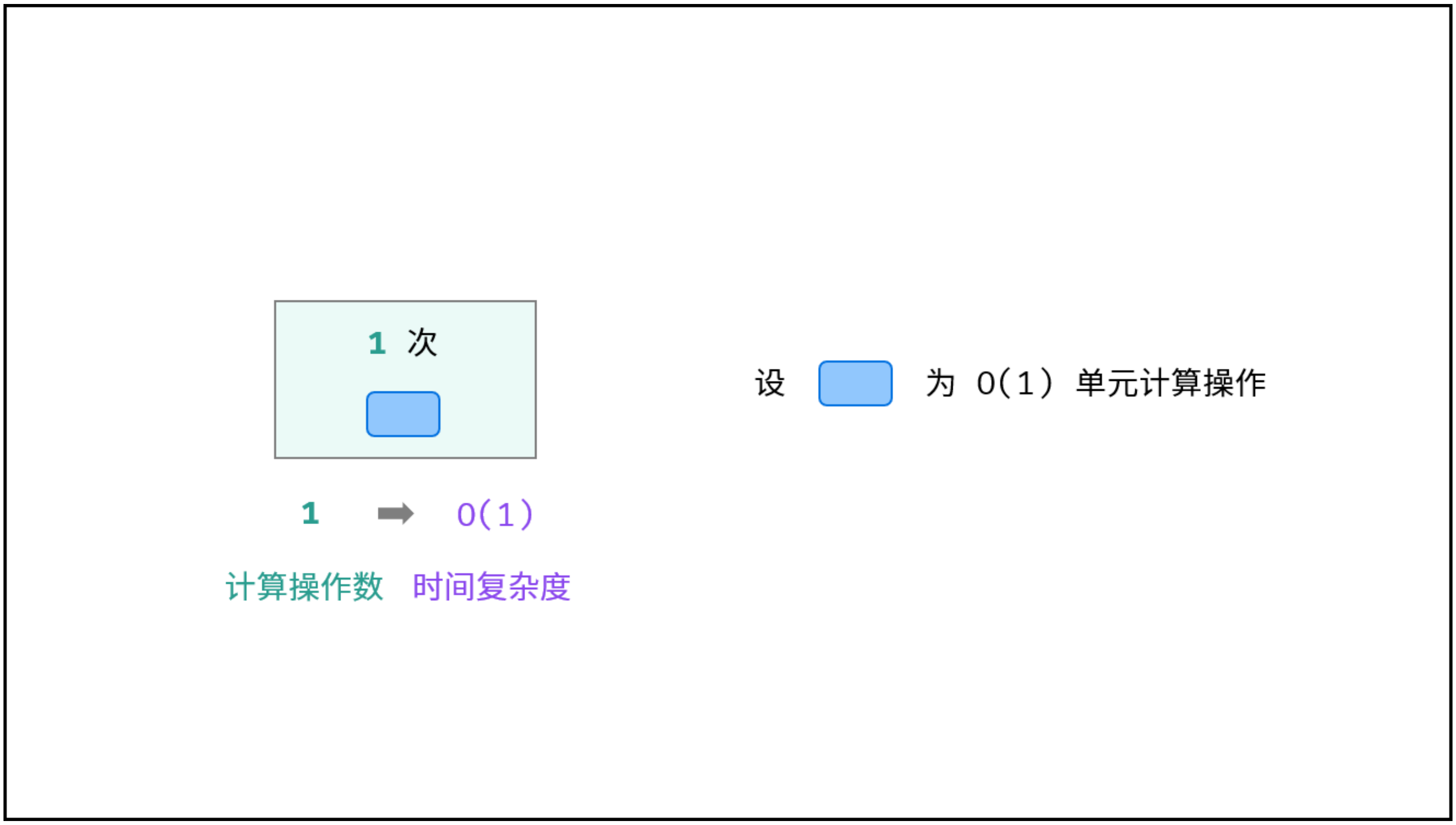
```java
int algorithm(int N) {
    int a = 1;
    int b = 2;
    int x = a * b + N;
    return 1;
}
```

对于以下代码，无论 a 取多大，都与输入数据大小 N 无关，因此时间复杂度仍为 $O(1)$ 。

```python
def algorithm(N):
    count = 0
    a = 10000
    for i in range(a):
        count += 1
    return count
```

```java
int algorithm(int N) {
    int count = 0;
    int a = 10000;
    for (int i = 0; i < a; i++) {
        count++;
    }
    return count;
}
```

```cpp
int algorithm(int N) {
    int count = 0;
    int a = 10000;
    for (int i = 0; i < a; i++) {
        count++;
    }
    return count;
}
```



线性 $O(N)$ ：

循环运行次数与 N 大小呈线性关系，时间复杂度为 $O(N)$ 。

```
def algorithm(N):
    count = 0
    for i in range(N):
        count += 1
    return count
```

```
int algorithm(int N) {
    int count = 0;
    for (int i = 0; i < N; i++)
        count++;
    return count;
}
```

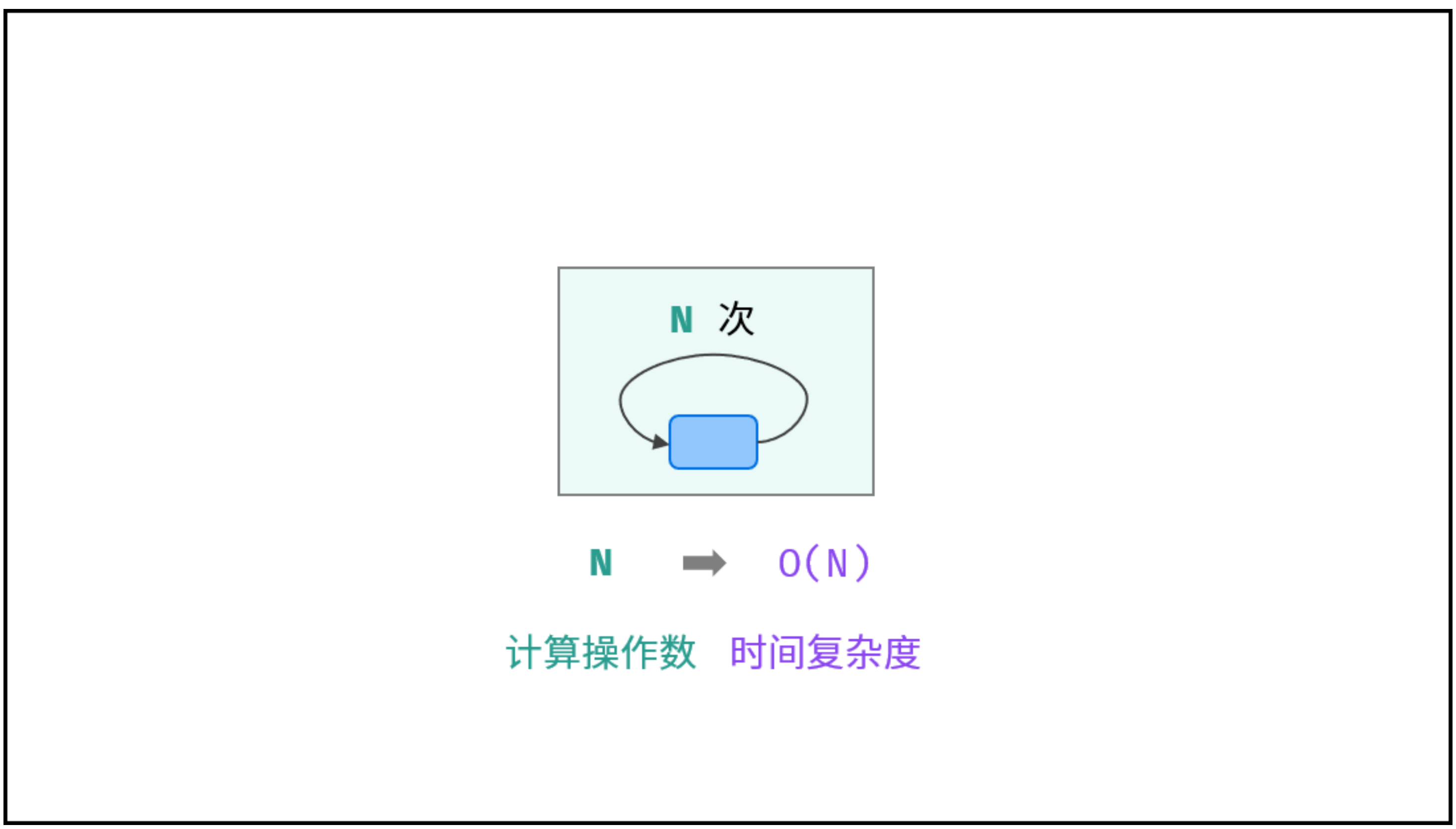
```
int algorithm(int N) {
    int count = 0;
    for (int i = 0; i < N; i++)
        count++;
    return count;
}
```

对于以下代码，虽然是两层循环，但第二层与 N 大小无关，因此整体仍与 N 呈线性关系。

```python
def algorithm(N):
    count = 0
    a = 10000
    for i in range(N):
        for j in range(a):
            count += 1
    return count
```

```java
int algorithm(int N) {
    int count = 0;
    int a = 10000;
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < a; j++) {
            count++;
        }
    }
    return count;
}
```

```cpp
int algorithm(int N) {
    int count = 0;
    int a = 10000;
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < a; j++) {
            count++;
        }
    }
    return count;
}
```



平方 $O(N^2)$：

两层循环相互独立，都与 N 呈线性关系，因此总体与 N 呈平方关系，时间复杂度为 $O(N^2)$ 。

```python
def algorithm(N):
    count = 0
    for i in range(N):
        for j in range(N):
            count += 1
    return count
```

```java
int algorithm(int N) {
    int count = 0;
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            count++;
        }
    }
    return count;
}
```

```cpp
int algorithm(int N) {
    int count = 0;
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            count++;
        }
    }
    return count;
}
```

以「冒泡排序」为例，其包含两层独立循环：

1. 第一层复杂度为$O(N)$ ；
2. 第二层平均循环次数为 $\frac{N}{2}$，复杂度为 $O(N)$ ，推导过程如下：

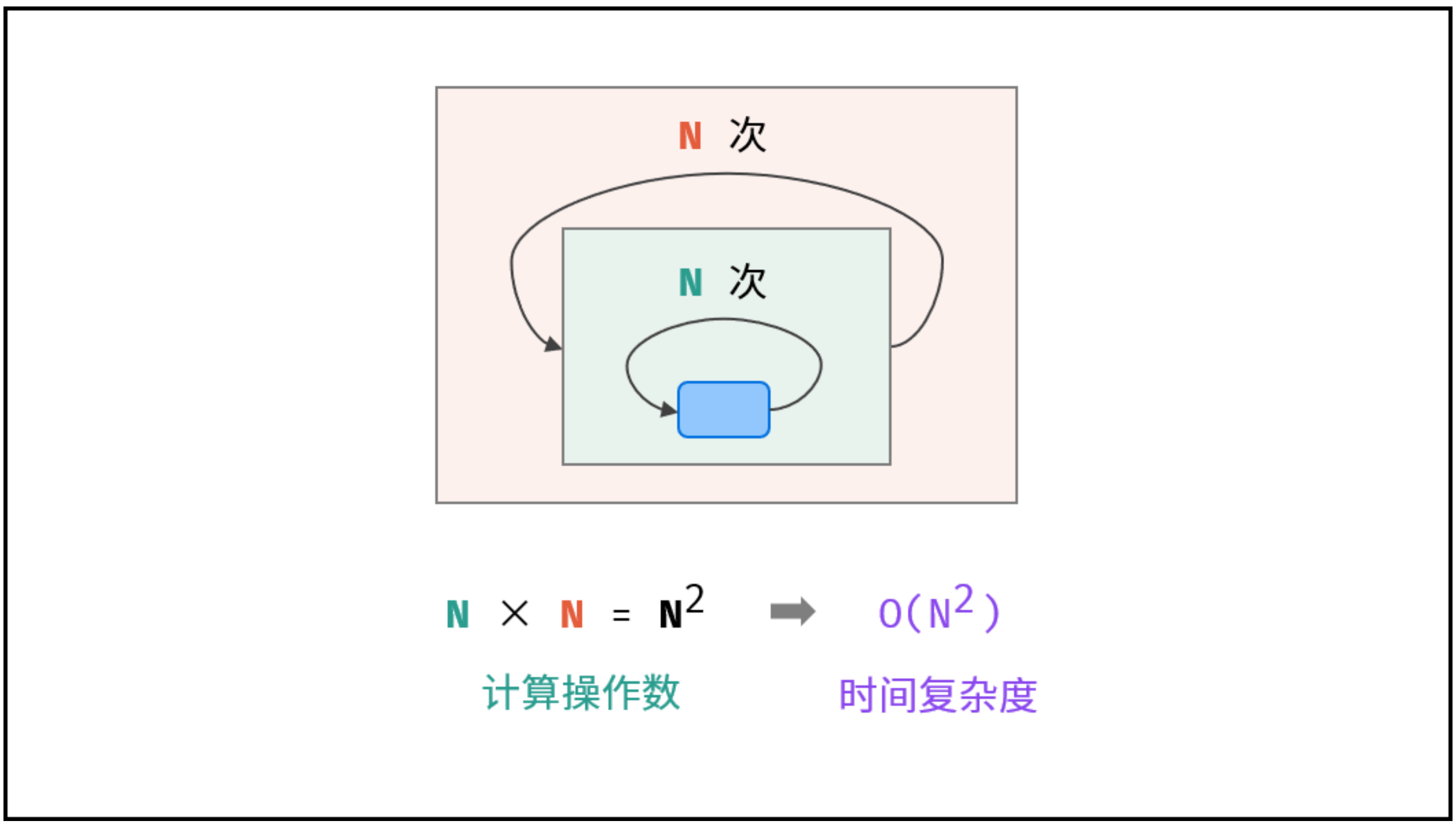
$$O(\frac{N}{2}) = O(\frac{1}{2})O(N) = O(1)O(N) = O(N)$$

因此，冒泡排序的总体时间复杂度为 $O(N^2)$ ，代码如下所示。

```python
def bubble_sort(nums):
    N = len(nums)
    for i in range(N - 1):
        for j in range(N - 1 - i):
            if nums[j] > nums[j + 1]:
                nums[j], nums[j + 1] = nums[j + 1], nums[j]
    return nums
```

```java
int[] bubbleSort(int[] nums) {
    int N = nums.length;
    for (int i = 0; i < N - 1; i++) {
        for (int j = 0; j < N - 1 - i; j++) {
            if (nums[j] > nums[j + 1]) {
                int tmp = nums[j];
                nums[j] = nums[j + 1];
                nums[j + 1] = tmp;
            }
        }
    }
    return nums;
}
```

```cpp
vector<int> bubbleSort(vector<int>& nums) {
    int N = nums.size();
    for (int i = 0; i < N - 1; i++) {
        for (int j = 0; j < N - 1 - i; j++) {
            if (nums[j] > nums[j + 1]) {
                swap(nums[j], nums[j + 1]);
            }
        }
    }
    return nums;
}
```



指数 $O(2^N)$ **：**

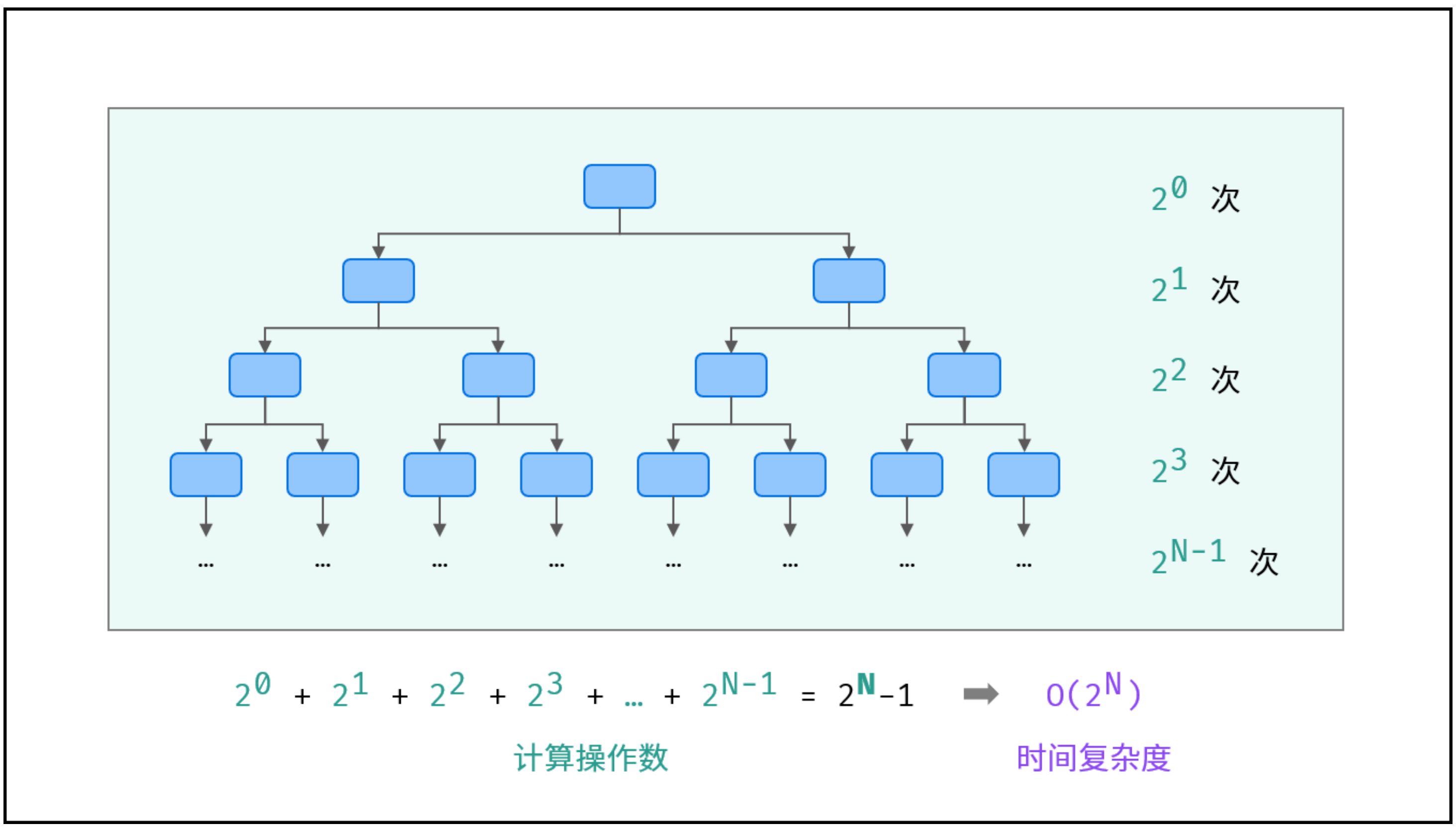
生物学科中的 “细胞分裂” 即是指数级增长。初始状态为 1 个细胞，分裂一轮后为 2 个，分裂两轮后为 4 个，……，分裂 N 轮后有 2^N 个细胞。

算法中，指数阶常出现于递归，算法原理图与代码如下所示。

```python
def algorithm(N):
    if N <= 0: return 1
    count_1 = algorithm(N - 1)
    count_2 = algorithm(N - 1)
    return count_1 + count_2
```

```java
int algorithm(int N) {
    if (N <= 0) return 1;
    int count_1 = algorithm(N - 1);
    int count_2 = algorithm(N - 1);
    return count_1 + count_2;
}
```

```cpp
int algorithm(int N) {
    if (N <= 0) return 1;
    int count_1 = algorithm(N - 1);
    int count_2 = algorithm(N - 1);
    return count_1 + count_2;
}
```



阶乘 $O(N!)$：

阶乘阶对应数学上常见的 “全排列” 。即给定 N 个互不重复的元素，求其所有可能的排列方案，则方案数量为：

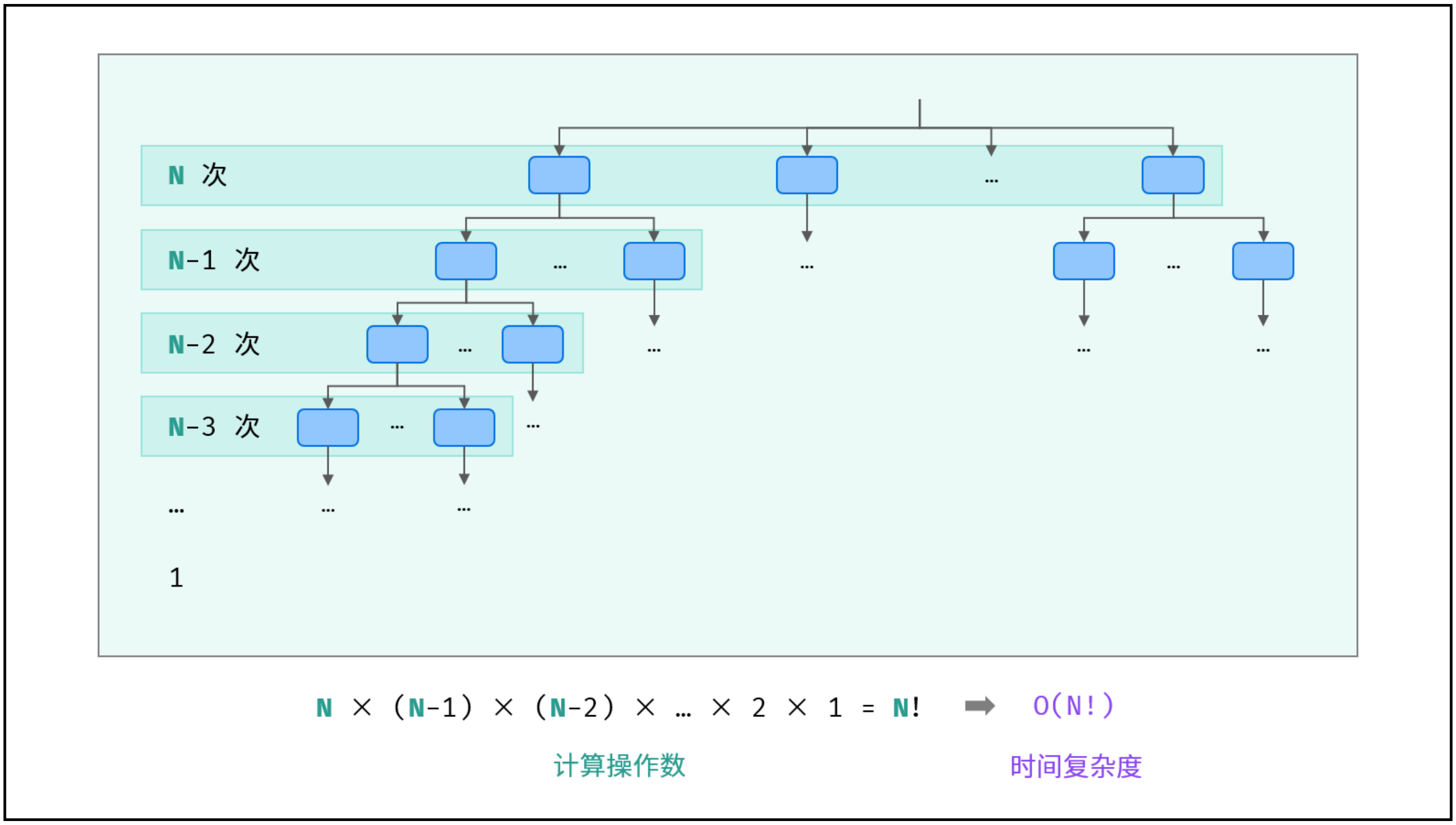
$$N \times (N-1) \times (N-2) \times \cdots \times 2 \times 1 = N!$$

如下图与代码所示，阶乘常使用递归实现，算法原理：第一层分裂出 N 个，第二层分裂出 $N-1$ 个，……，直至到第 N 层时终止并回溯。

```python
def algorithm(N):
    if N <= 0: return 1
    count = 0
    for _ in range(N):
        count += algorithm(N - 1)
    return count
```

```java
int algorithm(int N) {
    if (N <= 0) return 1;
    int count = 0;
    for (int i = 0; i < N; i++) {
        count += algorithm(N - 1);
    }
    return count;
}
```

```cpp
int algorithm(int N) {
    if (N <= 0) return 1;
    int count = 0;
    for (int i = 0; i < N; i++) {
        count += algorithm(N - 1);
    }
    return count;
}
```



对数 $O(\log N)$ ：

对数阶与指数阶相反，指数阶为 “每轮分裂出两倍的情况” ，而对数阶是 “每轮排除一半的情况” 。对数阶常出现于「二分法」、「分治」等算法中，体现着 “一分为二” 或 “一分为多” 的算法思想。

设循环次数为 m ，则输入数据大小 N 与 2^m 呈线性关系，两边同时取 log_2 对数，则得到循环次数 m 与 $log_2 N$ 呈线性关系，即时间复杂度为 $O(\log N)$ 。

```python
def algorithm(N):
    count = 0
    i = N
    while i > 1:
        i = i / 2
        count += 1
    return count
```

```cpp
int algorithm(int N) {
    int count = 0;
    float i = N;
    while (i > 1) {
        i = i / 2;
        count++;
    }
    return count;
}
```

```
int algorithm(int N) {
    int count = 0;
    float i = N;
    while (i > 1) {
        i = i / 2;
        count++;
    }
    return count;
}
```

如以下代码所示，对于不同 a 的取值，循环次数 m 与 $\log_a N$ 呈线性关系，时间复杂度为 O($\log_a N$)。而无论底数 a 取值，时间复杂度都可记作 $O(\log N)$，根据对数换底公式的推导如下：

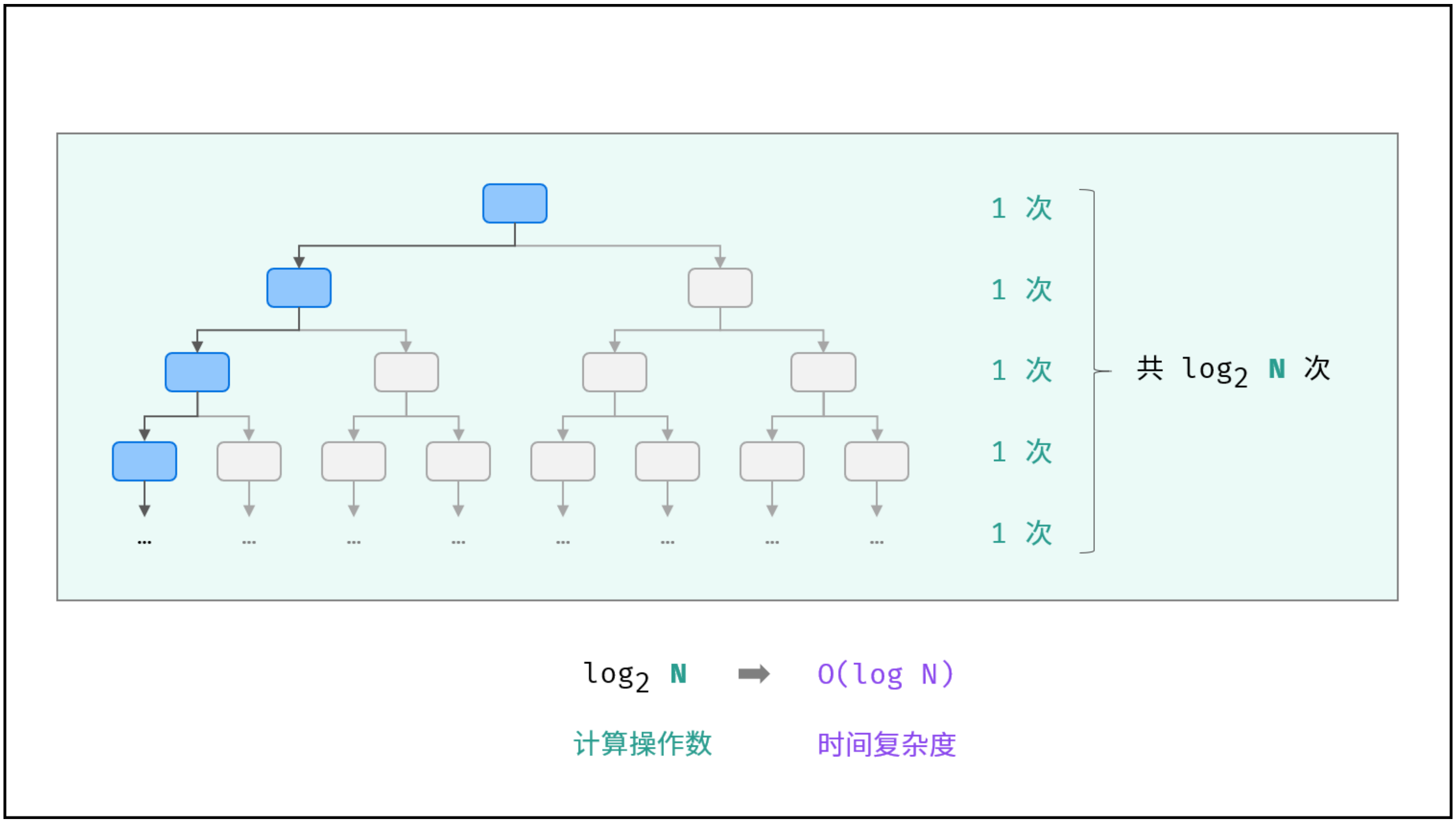
$$O(\log_a N) = \frac{O(\log_2 N)}{O(\log_2 a)} = O(\log N)$$

```
def algorithm(N):
    count = 0
    i = N
    a = 3
    while i > 1:
        i = i / a
        count += 1
    return count
```

```
int algorithm(int N) {
    int count = 0;
    float i = N;
    int a = 3;
    while (i > 1) {
        i = i / a;
        count++;
    }
    return count;
}
```

```
int algorithm(int N) {
    int count = 0;
    float i = N;
    int a = 3;
    while (i > 1) {
        i = i / a;
        count++;
    }
    return count;
}
```

如下图所示，为二分查找的时间复杂度示意图，每次二分将搜索区间缩小一半。



线性对数 $O(N \log N)$：

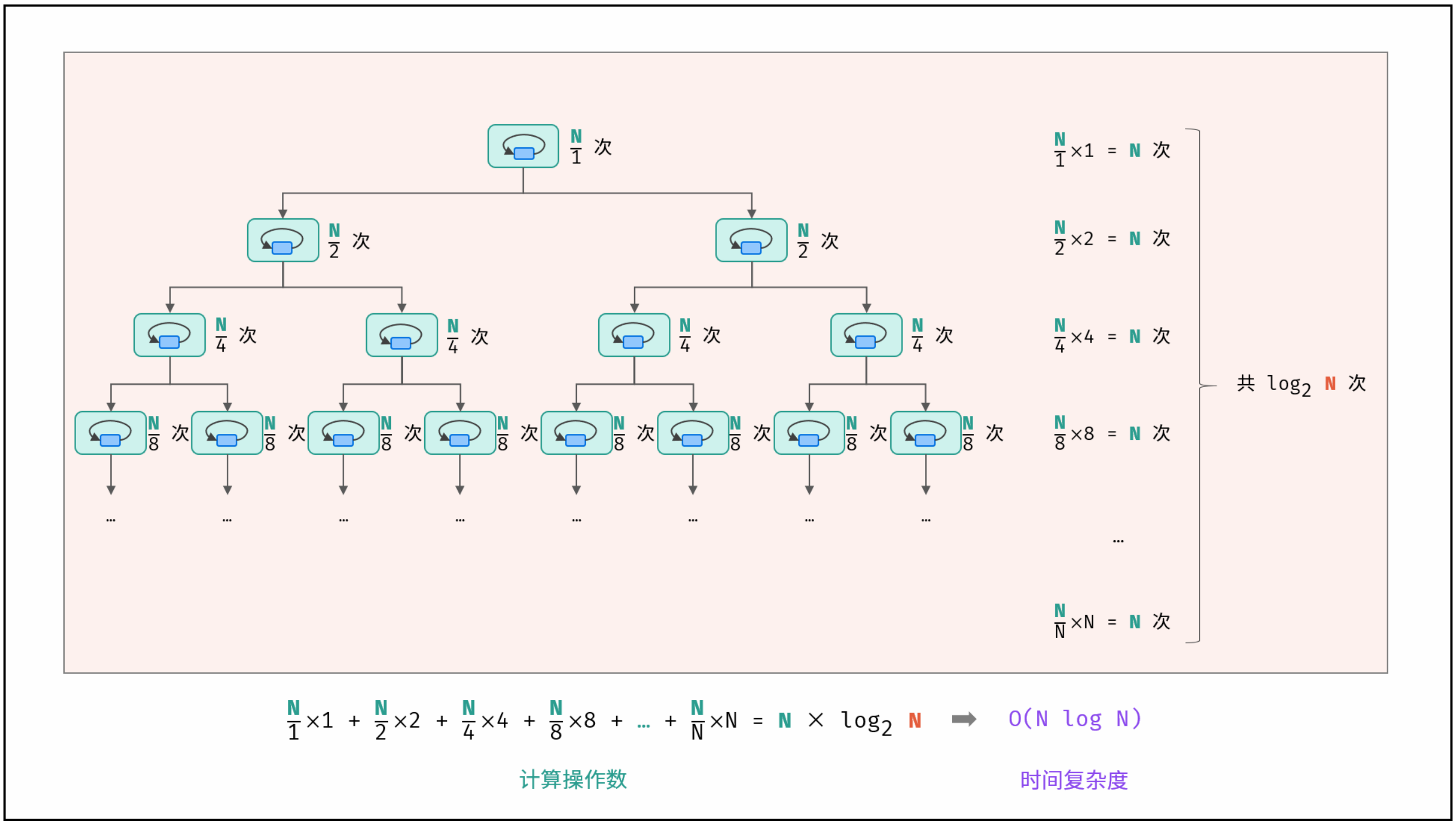
两层循环相互独立，第一层和第二层时间复杂度分别为 $O(\log N)$ 和 $O(N)$ ，则总体时间复杂度为 $O(N \log N)$ ；

```python
def algorithm(N):
    count = 0
    i = N
    while i > 1:
        i = i / 2
        for j in range(N):
            count += 1
```

```java
int algorithm(int N) {
    int count = 0;
    float i = N;
    while (i > 1) {
        i = i / 2;
        for (int j = 0; j < N; j++)
            count++;
    }
    return count;
}
```

```c
int algorithm(int N) {
    int count = 0;
    float i = N;
    while (i > 1) {
        i = i / 2;
        for (int j = 0; j < N; j++)
            count++;
    }
    return count;
}
```

线性对数阶常出现于排序算法，例如「快速排序」、「归并排序」、「堆排序」等，其时间复杂度原理如下图所示。



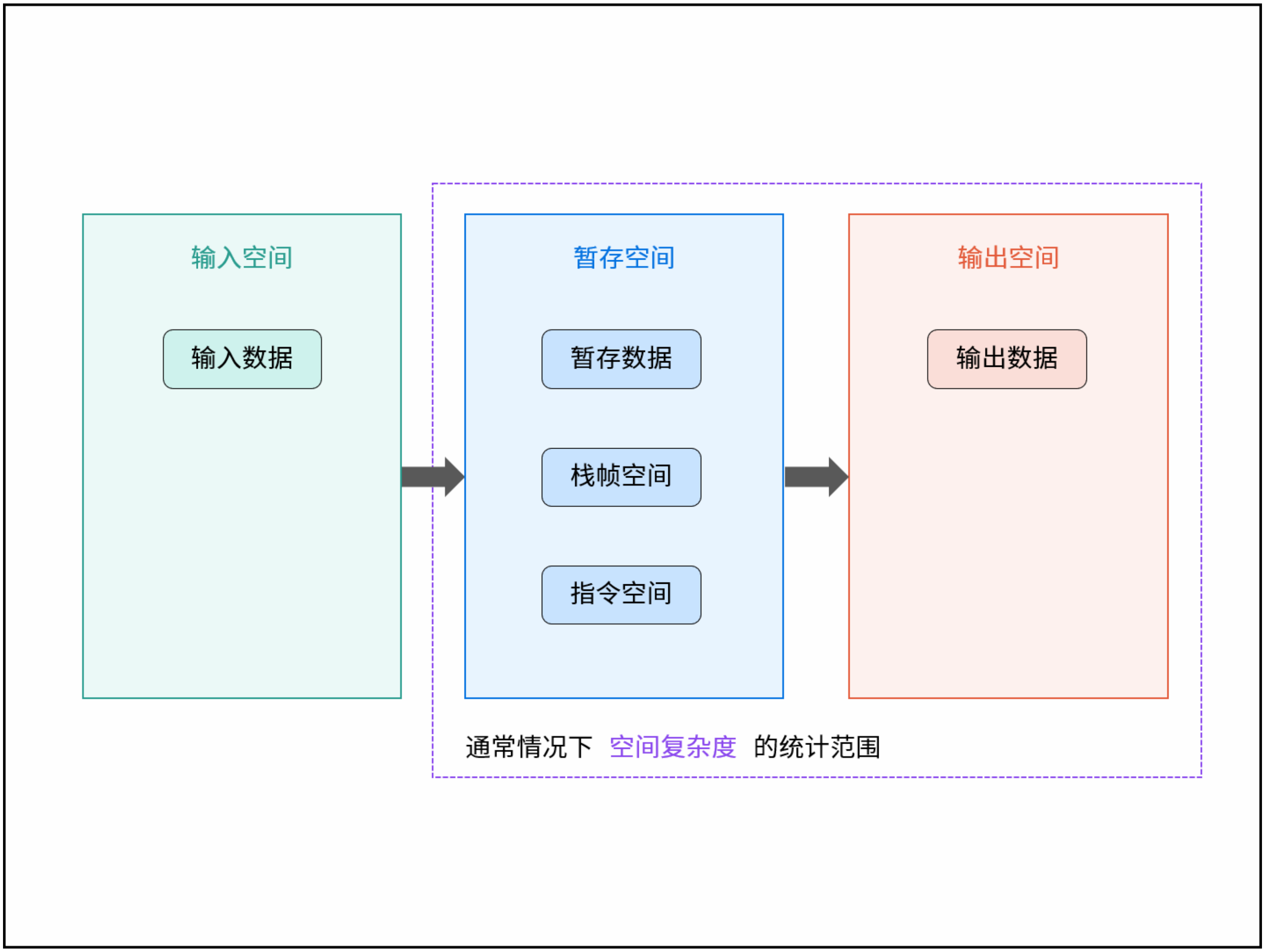
1.1.2.空间复杂度

空间复杂度概念定义

空间复杂度涉及的空间类型有：

- 输入空间： 存储输入数据所需的空间大小；
- 暂存空间： 算法运行过程中，存储所有中间变量和对象等数据所需的空间大小；
- 输出空间： 算法运行返回时，存储输出数据所需的空间大小；

通常情况下，空间复杂度指在输入数据大小为 N 时，算法运行所使用的「暂存空间」+「输出空间」的总体大小。



而根据不同来源，算法使用的内存空间分为三类：

指令空间：

编译后，程序指令所使用的内存空间。

数据空间：

算法中的各项变量使用的空间，包括：声明的常量、变量、动态数组、动态对象等使用的内存空间。

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

def algorithm(N):
    num = N          # 变量
    nums = [0] * N   # 动态数组
    node = Node(N)   # 动态对象
```

```java
class Node {
    int val;
    Node next;
    Node(int x) { val = x; }
}

void algorithm(int N) {
    int num = N;              // 变量
    int[] nums = new int[N]; // 动态数组
    Node node = new Node(N); // 动态对象
}
```

```cpp
struct Node {
    int val;
    Node *next;
    Node(int x) : val(x), next(NULL) {}
};

void algorithm(int N) {
    int num = N;              // 变量
    int nums[N];              // 动态数组
    Node* node = new Node(N); // 动态对象
}
```

栈帧空间：

程序调用函数是基于栈实现的，函数在调用期间，占用常量大小的栈帧空间，直至返回后释放。如以下代码所示，在循环中调用函数，每轮调用 test() 返回后，栈帧空间已被释放，因此空间复杂度仍为 $O(1)$ 。

```python
def test():
    return 0

def algorithm(N):
    for _ in range(N):
        test()
```

```java
int test() {
    return 0;
}

void algorithm(int N) {
    for (int i = 0; i < N; i++) {
        test();
    }
}
```

```
int test() {
    return 0;
}

void algorithm(int N) {
    for (int i = 0; i < N; i++) {
        test();
    }
}
```

算法中，栈帧空间的累计常出现于递归调用。如以下代码所示，通过递归调用，会同时存在 N 个未返回的函数 algorithm() ，此时累计使用 $O(N)$ 大小的栈帧空间。

```
def algorithm(N):
    if N <= 1: return 1
    return algorithm(N - 1) + 1
```

```
int algorithm(int N) {
    if (N <= 1) return 1;
    return algorithm(N - 1) + 1;
}
```

```
int algorithm(int N) {
    if (N <= 1) return 1;
    return algorithm(N - 1) + 1;
}
```

空间复杂度符号表示

通常情况下，空间复杂度统计算法在 “最差情况” 下使用的空间大小，以体现算法运行所需预留的空间量，使用符号 O 表示。

最差情况有两层含义，分别为「最差输入数据」、算法运行中的「最差运行点」。例如以下代码：

输入整数 N ，取值范围 $N \geq 1$ ；

- 最差输入数据： 当 $N \leq 10$ 时，数组 $nums$ 的长度恒定为 10 ，空间复杂度为 $O(10) = O(1)$ ；当 $N > 10$ 时，数组 $nums$ 长度为 N ，空间复杂度为 $O(N)$ ；因此，空间复杂度应为最差输入数据情况下的 $O(N)$ 。
- 最差运行点： 在执行 $nums = [0] * 10$ 时，算法仅使用 $O(1)$ 大小的空间；而当执行 $nums = [0] * N$ 时，算法使用 $O(N)$ 的空间；因此，空间复杂度应为最差运行点的 $O(N)$ 。

```python
def algorithm(N):
    num = 5              # O(1)
    nums = [0] * 10      # O(1)
    if N > 10:
        nums = [0] * N  # O(N)
```

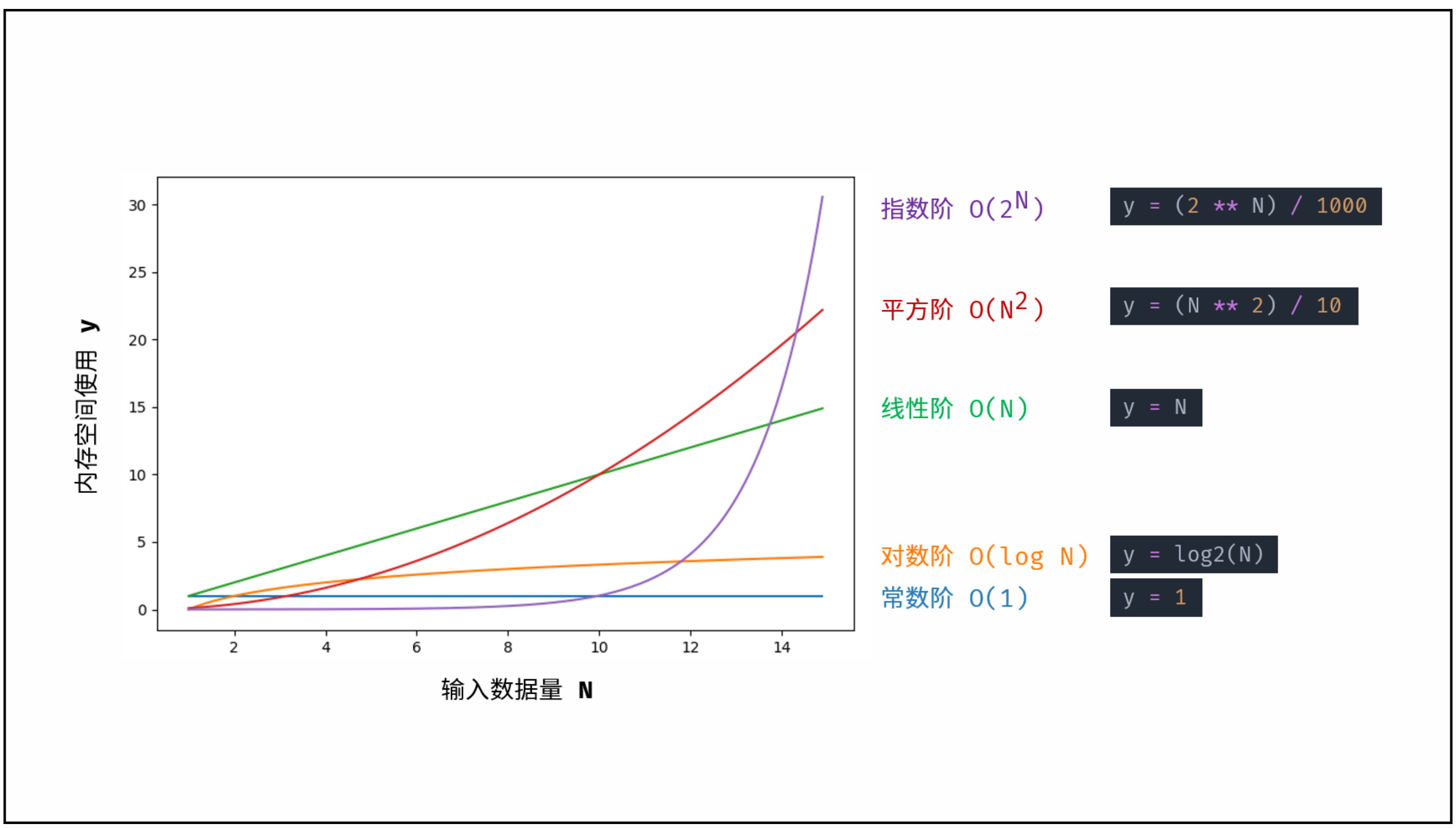
```java
void algorithm(int N) {
    int num = 5;              // O(1)
    int[] nums = new int[10]; // O(1)
    if (N > 10) {
        nums = new int[N];    // O(N)
    }
}
```

```cpp
void algorithm(int N) {
    int num = 5;            // O(1)
    vector<int> nums(10);   // O(1)
    if (N > 10) {
        nums.resize(N);     // O(N)
    }
}
```

空间复杂度常见种类

据从小到大排列，常见的算法空间复杂度有：

$$O(1) < O(\log N) < O(N) < O(N^2) < O(2^N)$$



示例解析

对于以下所有示例，设输入数据大小为正整数 N ，节点类 Node 、函数 test() 如以下代码所示。

```
# 节点类 Node
class Node:
    def __init__(self, val):
        self.val = val
        self.next = None

# 函数 test()
def test():
    return 0
```

```
// 节点类 Node
class Node {
    int val;                    // 变量
    Node next;                  // 动态数组
    Node(int x) { val = x; } // 动态对象
}

// 函数 test()
int test() {
    return 0;
}
```

```
// 节点类 Node
struct Node {
    int val;
    Node *next;
    Node(int x) : val(x), next(NULL) {}
};

// 函数 test()
int test() {
    return 0;
}
```

常数 $O(1)$ ：

普通常量、变量、对象、元素数量与输入数据大小 N 无关的集合，皆使用常数大小的空间。

```
def algorithm(N):
    num = 0
    nums = [0] * 10000
    node = Node(0)
    dic = { 0: '0' }
```

```java
void algorithm(int N) {
    int num = 0;
    int[] nums = new int[10000];
    Node node = new Node(0);
    Map<Integer, String> dic = new HashMap<>() {{ put(0, "0"); }};
}
```

```cpp
void algorithm(int N) {
    int num = 0;
    int nums[10000];
    Node* node = new Node(0);
    unordered_map<int, string> dic;
    dic.emplace(0, "0");
}
```

如以下代码所示，虽然函数 test() 调用了 N 次，但每轮调用后 test() 已返回，无累计栈帧空间使用，因此空间复杂度仍为 $O(1)$ 。

```python
def algorithm(N):
    for _ in range(N):
        test()
```

```java
void algorithm(int N) {
    for (int i = 0; i < N; i++) {
        test();
    }
}
```

```cpp
void algorithm(int N) {
    for (int i = 0; i < N; i++) {
        test();
    }
}
```

线性 $O(N)$ ：

元素数量与 N 呈线性关系的任意类型集合（常见于一维数组、链表、哈希表等），皆使用线性大小的空间。

```python
def algorithm(N):
    nums_1 = [0] * N
    nums_2 = [0] * (N // 2)

    nodes = [Node(i) for i in range(N)]

    dic = {}
    for i in range(N):
        dic[i] = str(i)
```

```
void algorithm(int N) {
    int[] nums_1 = new int[N];
    int[] nums_2 = new int[N / 2];

    List<Node> nodes = new ArrayList<>();
    for (int i = 0; i < N; i++) {
        nodes.add(new Node(i));
    }

    Map<Integer, String> dic = new HashMap<>();
    for (int i = 0; i < N; i++) {
        dic.put(i, String.valueOf(i));
    }
}
```

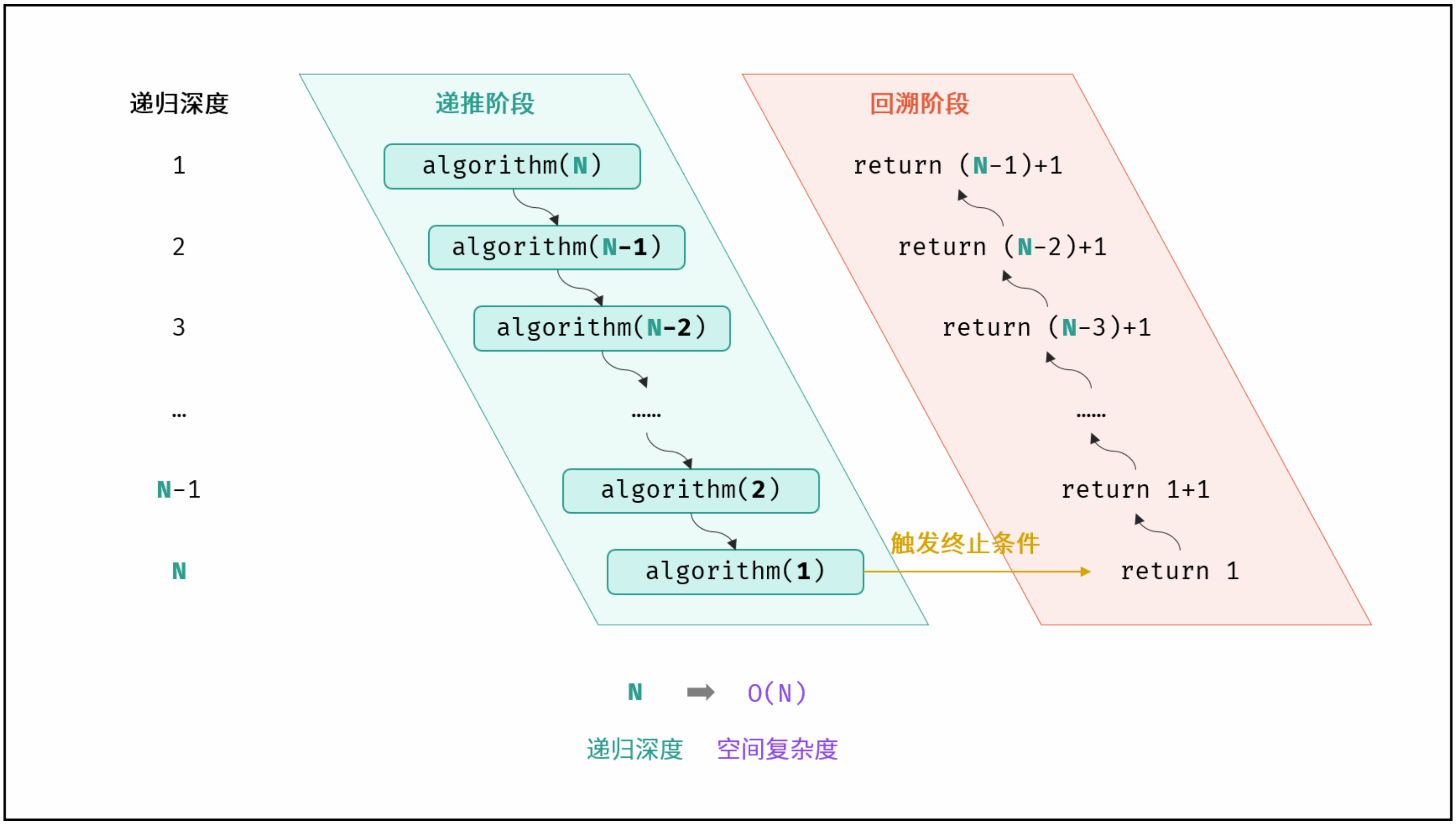
```
void algorithm(int N) {
    int nums_1[N];
    int nums_2[N / 2 + 1];

    vector<Node*> nodes;
    for (int i = 0; i < N; i++) {
        nodes.push_back(new Node(i));
    }

    unordered_map<int, string> dic;
    for (int i = 0; i < N; i++) {
        dic.emplace(i, to_string(i));
    }
}
```

如下图与代码所示，此递归调用期间，会同时存在 N 个未返回的 algorithm() 函数，因此使用 $O(N)$ 大小的栈帧空间。

```
def algorithm(N):
    if N <= 1: return 1
    return algorithm(N - 1) + 1
```

```
int algorithm(int N) {
    if (N <= 1) return 1;
    return algorithm(N - 1) + 1;
}
```

```
int algorithm(int N) {
    if (N <= 1) return 1;
    return algorithm(N - 1) + 1;
}
```



平方 $O(N^2)$：

元素数量与 N 呈平方关系的任意类型集合（常见于矩阵），皆使用平方大小的空间。

```
def algorithm(N):
    num_matrix = [[0 for j in range(N)] for i in range(N)]
    node_matrix = [[Node(j) for j in range(N)] for i in range(N)]
```

```
void algorithm(int N) {
    int num_matrix[][] = new int[N][N];

    List<List<Node>> node_matrix = new ArrayList<>();
    for (int i = 0; i < N; i++) {
        List<Node> nodes = new ArrayList<>();
        for (int j = 0; j < N; j++) {
            nodes.add(new Node(j));
        }
        node_matrix.add(nodes);
    }
}
```

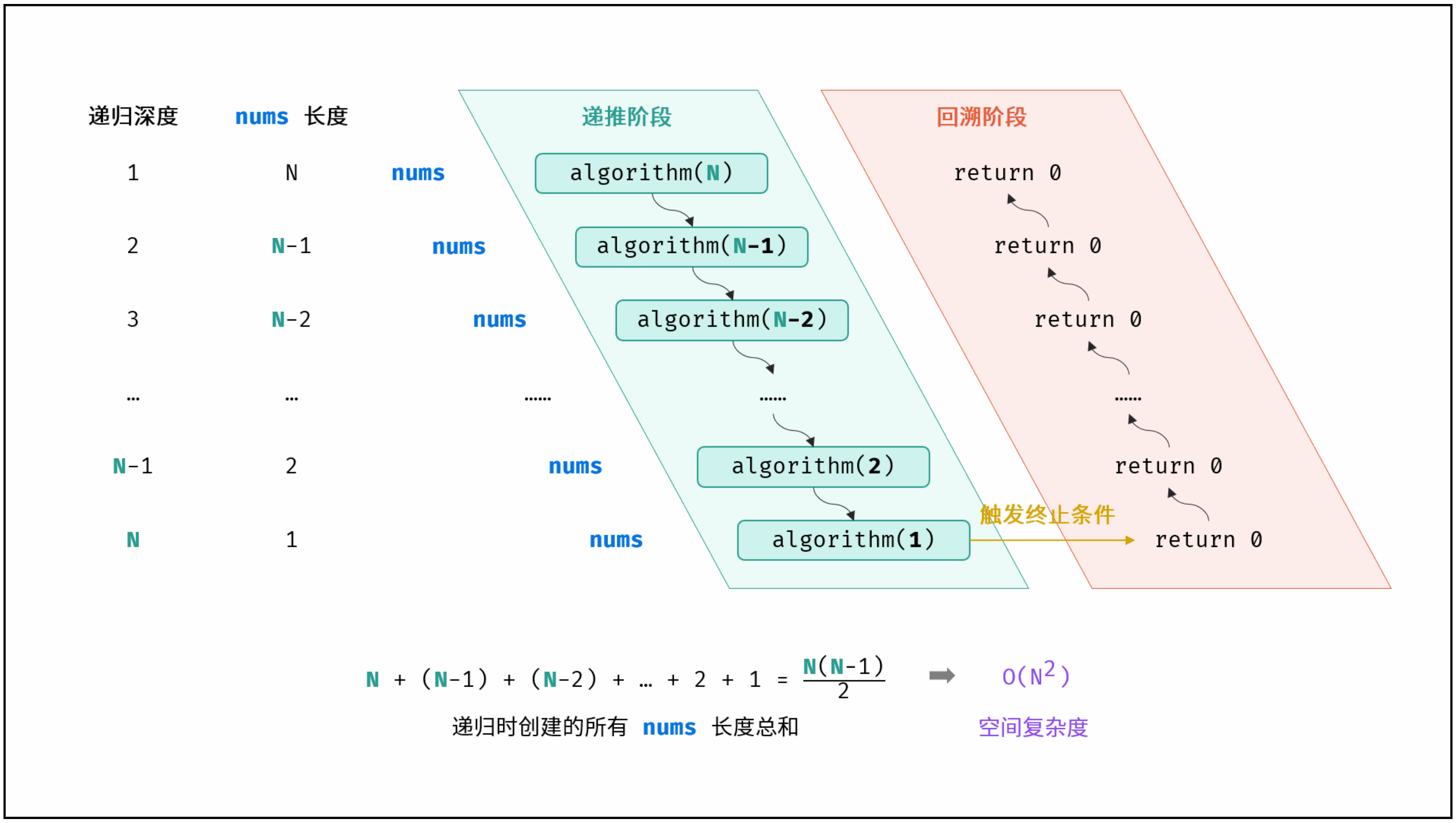
```
void algorithm(int N) {
    vector<vector<int>> num_matrix;
    for (int i = 0; i < N; i++) {
        vector<int> nums;
        for (int j = 0; j < N; j++) {
            nums.push_back(0);
        }
        num_matrix.push_back(nums);
    }

    vector<vector<Node*>> node_matrix;
    for (int i = 0; i < N; i++) {
        vector<Node*> nodes;
        for (int j = 0; j < N; j++) {
            nodes.push_back(new Node(j));
        }
        node_matrix.push_back(nodes);
    }
}
```

如下图与代码所示，递归调用时同时存在 N 个未返回的 algorithm() 函数，使用$O(N)$ 栈帧空间；每层递归函数中声明了数组，平均长度为 $\frac{N}{2}$ ，使用$O(N)$ 空间；因此总体空间复杂度为 $O(N^2)$ 。

```
def algorithm(N):
    if N <= 0: return 0
    nums = [0] * N
    return algorithm(N - 1)
```

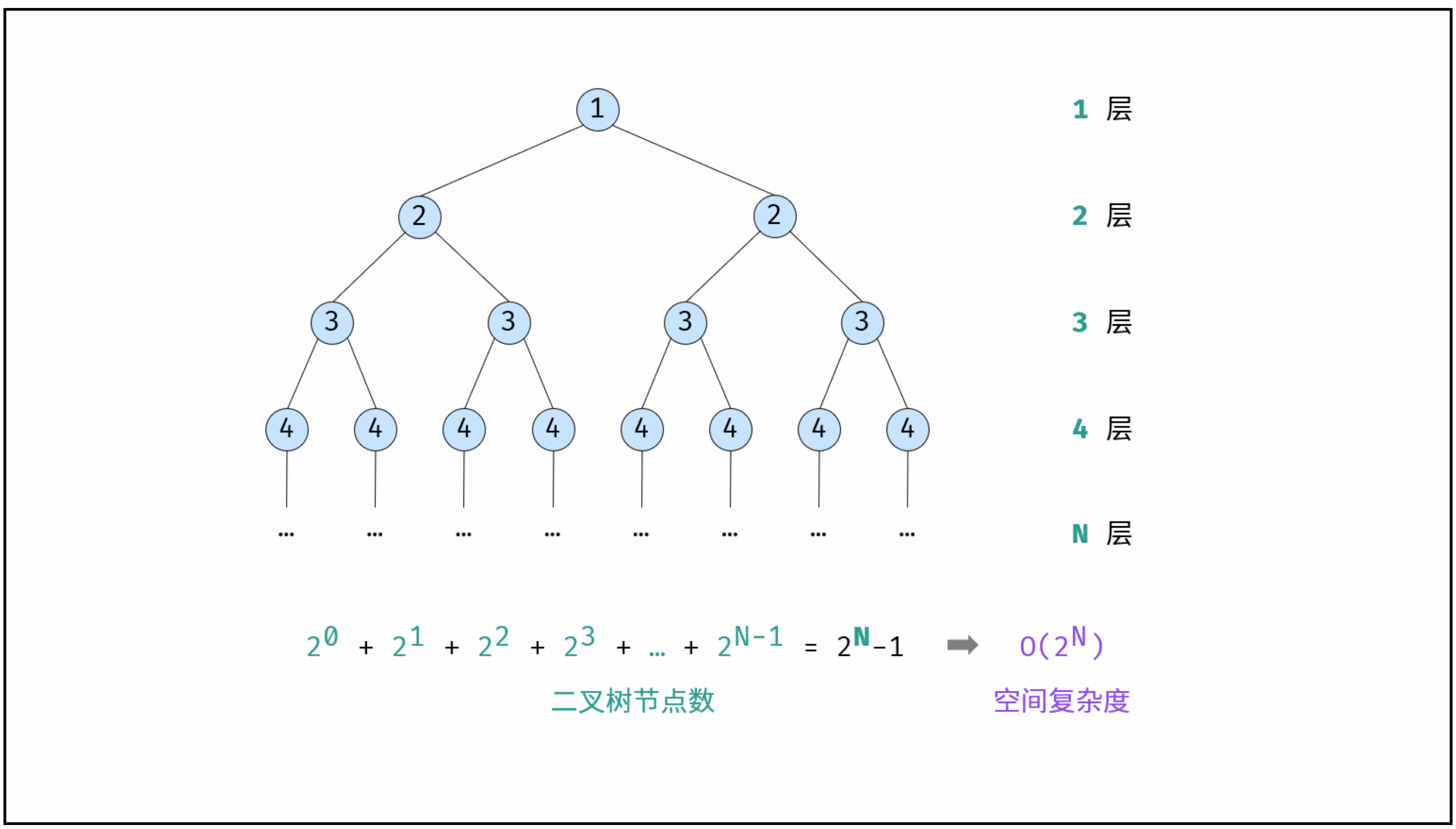
```
int algorithm(int N) {
    if (N <= 0) return 0;
    int[] nums = new int[N];
    return algorithm(N - 1);
}
```

```
int algorithm(int N) {
    if (N <= 0) return 0;
    int nums[N];
    return algorithm(N - 1);
}
```



指数 $O(2^N)$:

指数阶常见于二叉树、多叉树。例如，高度为 N 的「满二叉树」的节点数量为 2^N，占用 $O(2^N)$ 大小的空间；同理，高度为 N 的「满 m 叉树」的节点数量为 m^N，占用 $O(m^N) = O(2^N)$ 大小的空间。



对数 $O(\log N)$：
对数阶常出现于分治算法的栈帧空间累计、数据类型转换等，例如：

- 快速排序，平均空间复杂度为 $\Theta(\log N)$，最差空间复杂度为 $O(N)$。拓展知识：通过应用 Tail Call Optimization，可以将快速排序的最差空间复杂度限定至 $O(N)$。
- 数字转化为字符串，设某正整数为 N，则字符串的空间复杂度为 $O(\log N)$。推导如下：正整数 N 的位数为 $\log_{10} N$，即转化的字符串长度为 $\log_{10} N$，因此空间复杂度为 $O(\log N)$。

时空权衡

对于算法的性能，需要从时间和空间的使用情况来综合评价。优良的算法应具备两个特性，即时间和空间复杂度皆较低。而实际上，对于某个算法问题，同时优化时间复杂度和空间复杂度是非常困难的。降低时间复杂度，往往是以提升空间复杂度为代价的，反之亦然。

由于当代计算机的内存充足，通常情况下，算法设计中一般会采取「空间换时间」的做法，即牺牲部分计算机存储空间，来提升算法的运行速度。

以 LeetCode 全站第一题 两数之和 为例，「暴力枚举」和「辅助哈希表」分别为「空间最优」和「时间最优」的两种算法。

方法一：暴力枚举

时间复杂度 $O(N^2)$，空间复杂度 $O(1)$；属于「时间换空间」，虽然仅使用常数大小的额外空间，但运行速度过慢。

```python
class Solution:
    def twoSum(self, nums: List[int], target: int) -> List[int]:
        for i in range(len(nums) - 1):
            for j in range(i + 1, len(nums)):
                if nums[i] + nums[j] == target:
                    return i, j
        return
```

```java
class Solution {
    public int[] twoSum(int[] nums, int target) {
        int size = nums.length;
        for (int i = 0; i < size - 1; i++) {
            for (int j = i + 1; j < size; j++) {
                if (nums[i] + nums[j] == target)
                    return new int[] { i, j };
            }
        }
        return new int[0];
    }
}
```

```cpp
class Solution {
public:
    vector<int> twoSum(vector<int>& nums, int target) {
        int size = nums.size();
        for (int i = 0; i < size - 1; i++) {
            for (int j = i + 1; j < size; j++) {
                if (nums[i] + nums[j] == target)
                    return { i, j };
            }
        }
        return {};
    }
};
```

方法二：辅助哈希表

时间复杂度 $O(N)$ ，空间复杂度 $O(N)$ ；属于「空间换时间」，借助辅助哈希表 dic ，通过保存数组元素值与索引的映射来提升算法运行效率，是本题的最佳解法。

```python
class Solution:
    def twoSum(self, nums: List[int], target: int) -> List[int]:
        dic = {}
        for i in range(len(nums)):
            if target - nums[i] in dic:
                return dic[target - nums[i]], i
            dic[nums[i]] = i
        return []
```

```java
class Solution {
    public int[] twoSum(int[] nums, int target) {
        int size = nums.length;
        Map<Integer, Integer> dic = new HashMap<>();
        for (int i = 0; i < size; i++) {
            if (dic.containsKey(target - nums[i])) {
                return new int[] { dic.get(target - nums[i]), i };
            }
            dic.put(nums[i], i);
        }
        return new int[0];
    }
}
```

```cpp
class Solution {
public:
    vector<int> twoSum(vector<int>& nums, int target) {
        int size = nums.size();
        unordered_map<int, int> dic;
        for (int i = 0; i < size; i++) {
            if (dic.find(target - nums[i]) != dic.end()) {
                return { dic[target - nums[i]], i };
            }
            dic.emplace(nums[i], i);
        }
        return {};
    }
};
```