复杂度分析

想不出来起什么名字 于 2024-12-08 12:22:15 发布

数据结构与算法 专栏收录该内容

0 订阅　1 篇文章

前言

我们进入数据结构和算法的学习环节。

什么是数据结构?

数据结构（Data Structure）：计算机储存数据的一种方式

为什么会有各种数据结构?

用户有不同的需求，我们解决实际问题时需要考虑到**时间成本**和**空间成本**，就需要我们更好地去管理和使用数据

基于解决实际问题，我们不难总结出数据结构有如下目标:

- 空间尽量占用少，省内存
- 数据操作尽可能快速，涵盖数据增删查改等操作
- 提供简洁的数据表示和逻辑信息，以便算法高效运行

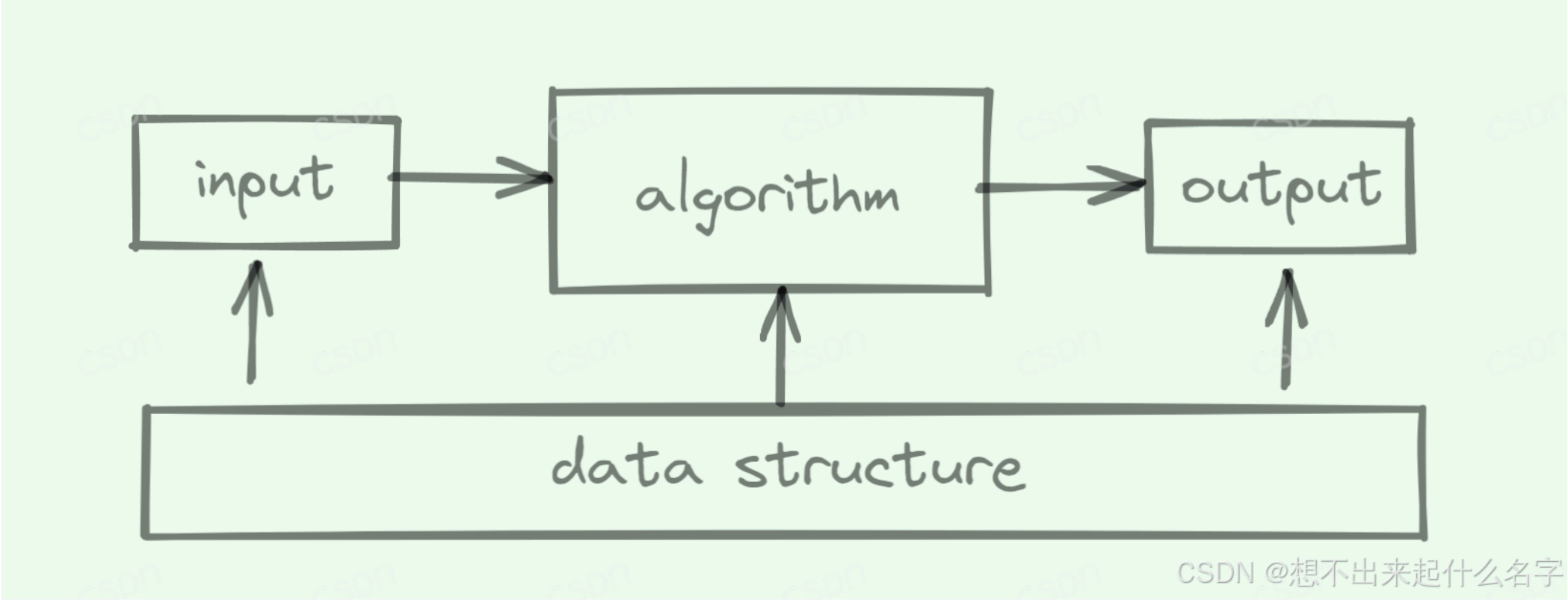
引出算法的概念，什么是算法?

算法（Algorithm）：在有限时间内解决特定问题的一组指令或操作步骤，换句话说，我们利用算法来解决实际生活中的问题和需求

自然，我们也能总结出算法的特点:

- 问题是明确的，包含清晰的输入和输出定义（一个需求和问题不明确，怎么解决？）
- 具有可行性，能够在有限步骤、时间和内存空间下完成（保证实现的效率）
- 各步骤都有确定的含义，在相同的输入和运行条件下，输出始终相同（保证实现的准确性）

所以数据结构是基础，算法在数据结构的基础上进行操作，二者紧密联系



衡量指标

算法在编写成可执行程序后，运行时需要耗费**时间资源**和**空间(内存)资源**

因此衡量一个算法的好坏，一般是从**时间**和**空间**两个维度来衡量的，即**时间复杂度**和**空间复杂度**

时间复杂度主要衡量一个算法的运行快慢，而空间复杂度主要衡量一个算法运行所需要的额外空间

在计算机发展的早期，计算机的存储容量很小。所以对空间复杂度很是在乎。但是经过计算机行业的迅速发展，计 算机的存储容量已经达到了很高的程度。所以我们如今已经不需要再特别关注一个算法的空间复杂度

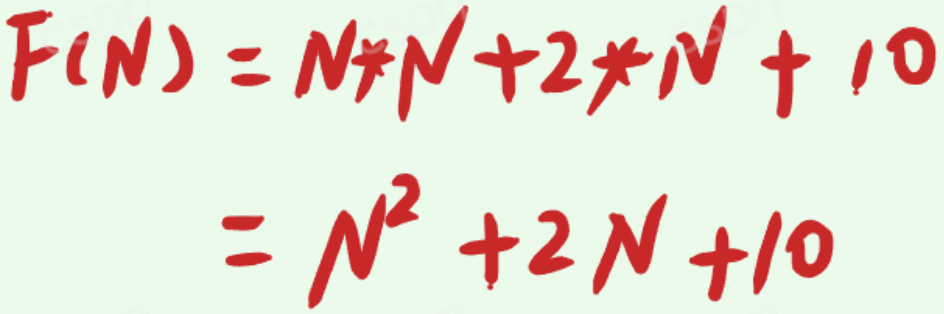
时间复杂度

算法的时间复杂度是一个函数，算法中的基本操作的执行次数，为算法的时间复杂度


```
1 void Func1(int N)
2 {
```

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

    for (int k = 0; k < 2 * N; ++k)
    {
        ++count;
    }

    int M = 10;
    while (M--)
    {
        ++count;
    }
    printf("%d\n", count);
}
```

```
void Func1(int N)
{
    int count = 0;
    for (int i = 0; i < N; ++i)
    {
        for (int j = 0; j < N; ++j)
        {
            ++count;
        }
    }

    for (int k = 0; k < 2 * N; ++k)
    {
        ++count;
    }

    int M = 10;
    while (M--)
    {
        ++count;
    }
    printf("%d\n", count);
}
```



这个函数表达式就是这段代码的时间复杂度，如果们每次要精确计算执行次数，会有些麻烦，所以我们只需要算个大概，找到这个代码执行次数的**量级**即可 ，这种方法称为**大O的渐进表示法**

常数阶O(1)

常数阶的操作数量与输入数据大小n无关，即不随着n的变化而变化。 在以下函数中，尽管操作数量 :size 可能很大，但由于其与输入数据大小n无关，因此时间复杂度仍为O(1)

```
//常数阶O(1)
int CountSize(int n) {
    int size = 100000;
    int count = 0;
    for (size_t i = 0; i < size; i++)
    {
        count++;
    }
    return count;
}
```

线性阶O(n)

操作数量相对于输入数据大小n以线性级别增长,线性阶通常出现在单层循环中

值得一提的是：遍历数组和遍历链表等操作,时间复杂度为O(n),n是数组或者链表的长度

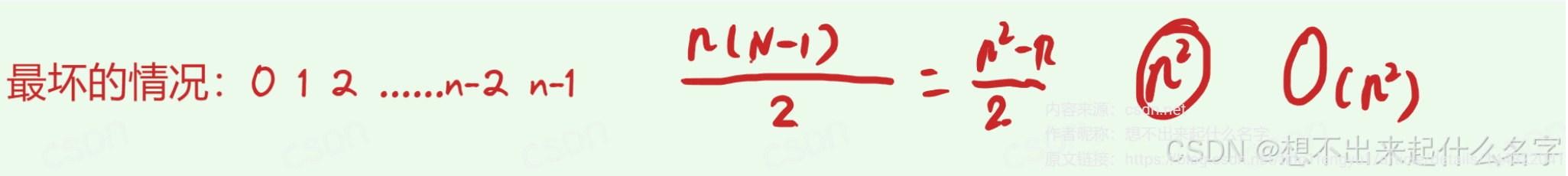
```
//线性阶O(n)
/*操作数量相对于输入数据大小n以线性级别增长,通常出现在单层循环中*/
//遍历数组和遍历链表等操作,O(n),n是数组或者链表的长度
int Test1(int n) {
    int count = 0;
    for (size_t i = 0; i < n; i++)
    {
        count++;
    }
    return count;
}
int Test2(int m) {
    int count = 0;
    while (count < m)
    {
        count++;
    }
    return count;
}
```

平方阶O(n^2)

平方阶的操作数量相对于输入数据大小n以平方级别增长

平方阶通常出现在嵌套循环中，外层和内层都为O(n)，因此总体为O(n^2)

```
void BubbleSort(int* arr, size_t size)
{
    for (size_t i = 0; i < size - 1; i++)//两两排序，有size个元素，则有size - 1趟
    {
        bool flag = 1;//假设这一趟已经有序
        for (size_t j = 0; j < size - 1 - i; j++)//排序完成的元素无需作比较，逐次往后比较
        {
            if (arr[j] > arr[j + 1])//顺序，如果降序则改为<
            {
                flag = 0;//发生了交换，说明无序
                int tmp = arr[j];//创建第三个变量方便排序
                arr[j] = arr[j + 1];
                arr[j + 1] = tmp;//升序完成
            }
        }
        if (flag) break;//未交换说明有序，直接跳出循环！
    }
}
```

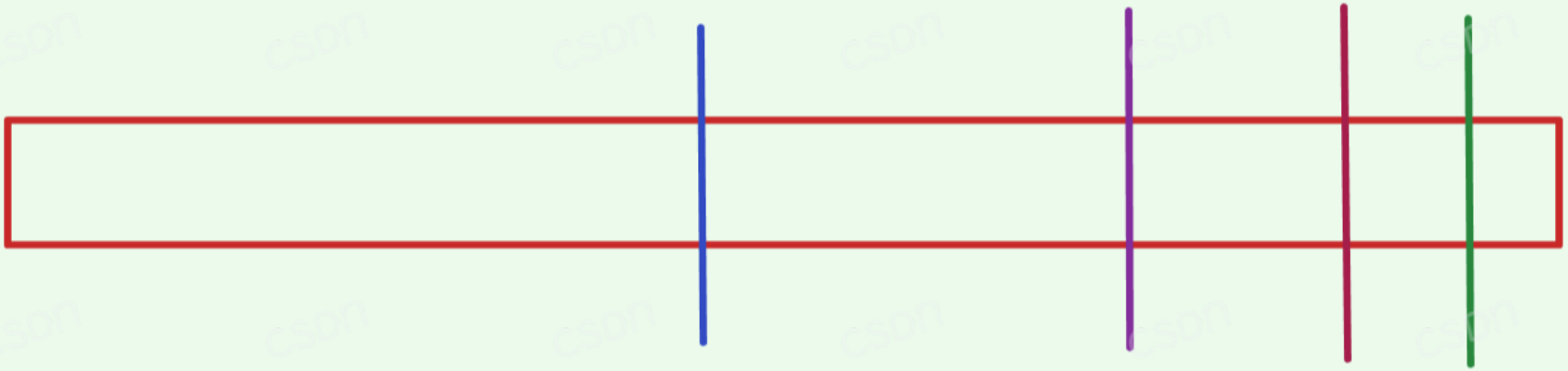


对数阶O(logn)

二分查找是典型的对数阶，**每轮缩减到一半**，再比如说将纸对折

```
int BinarySearch(int* arr, int size, int x) {
    asseert(arr);

    int end = size - 1;//我们找的是索引
    int start = 0;

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

二分查找的本质是缩小区间，每次缩小到原来的一半



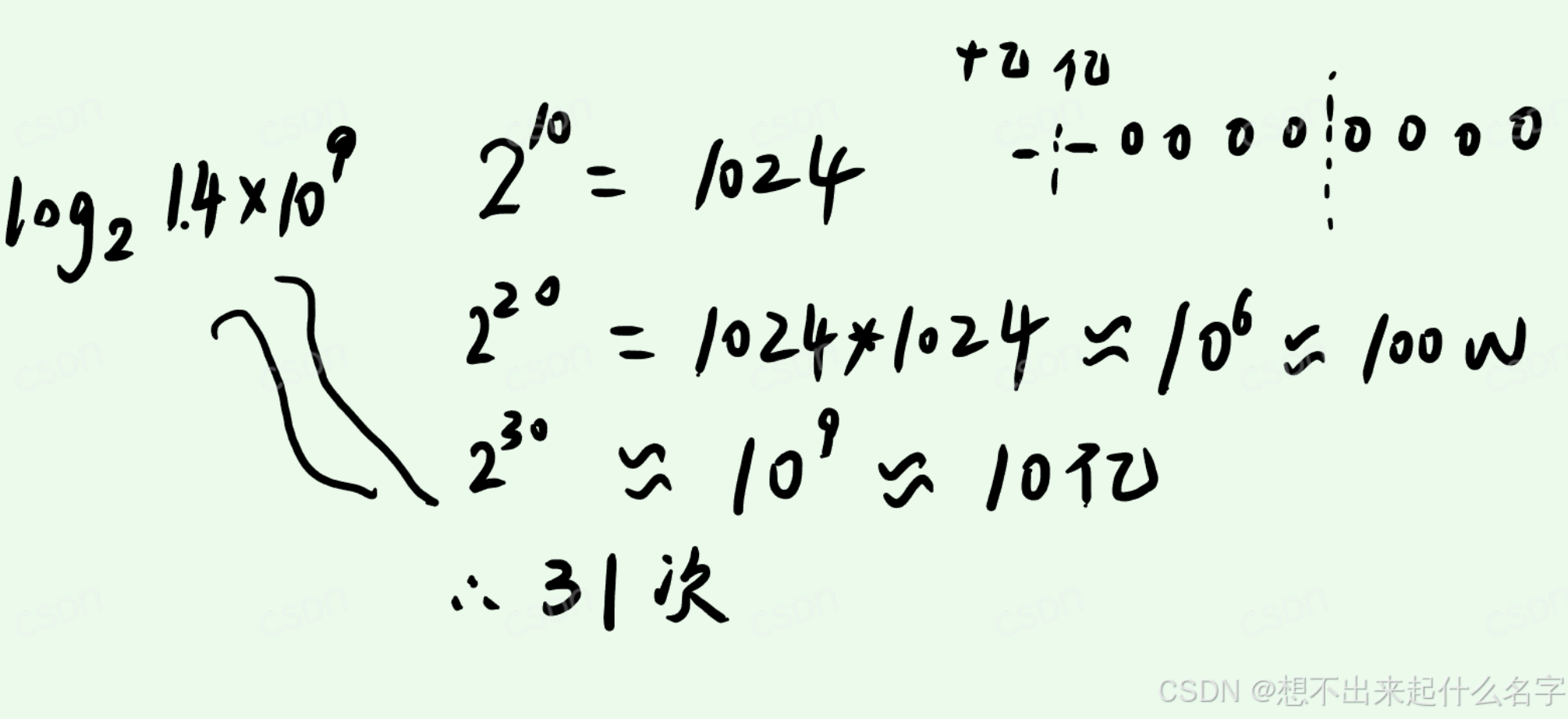
假设区间长度为n n/2/2/2/2/...../2 == 1, 假设除了x次2

n == 2^x

x == $log_2 n$

我们简写成logn

再来举个例子，假设我国14亿人口，放到数组中排序，找一个人，进行二分查找，最多要多少次?



指数阶O(2^n)

“细胞分裂”是指数阶增长的典型例子，（假设细胞不死）初始状态为1个细胞，分裂一轮后变为2个，分裂两轮后变为4个，以此类推，分裂n轮后有2n个细胞 又比如说汉诺塔问题

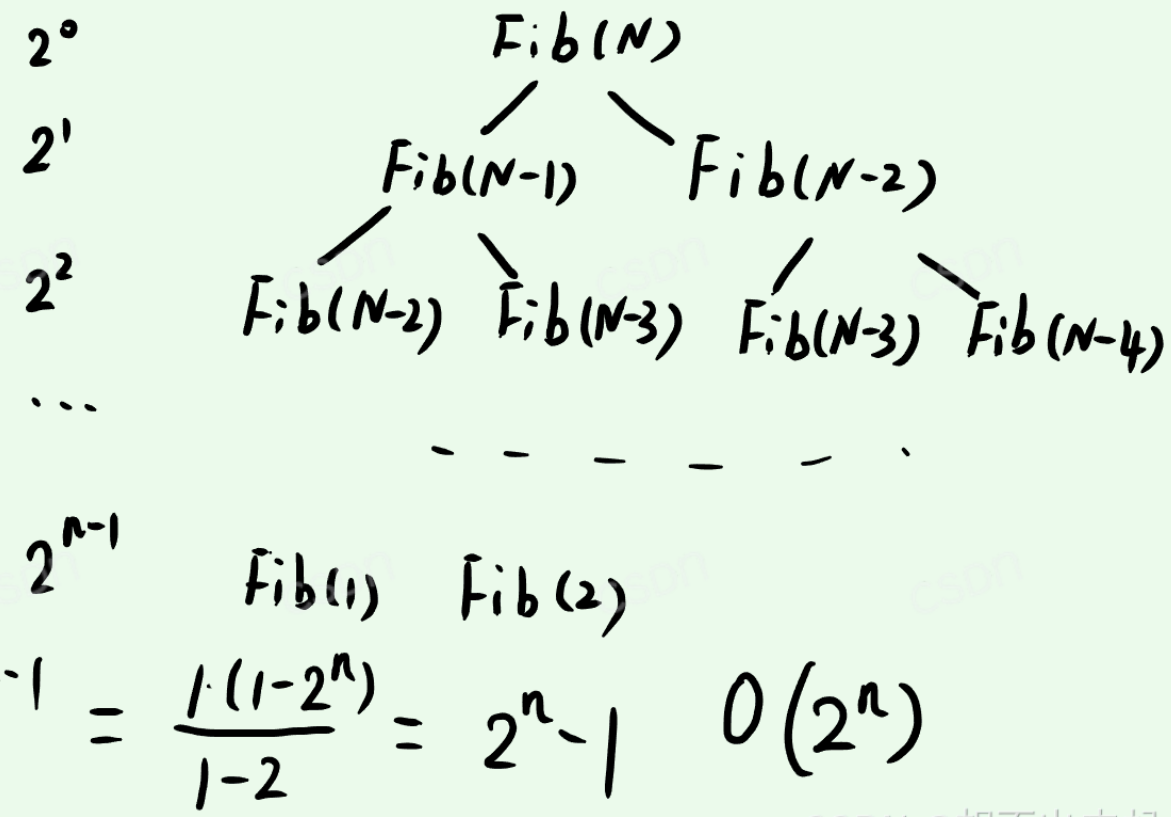
指数阶常出现于递归函数

对数阶和指数阶是互为反函数关系的

```
long long Fib(size_t N)
 {
    if(N < 3)
        return 1;
```

```
5       6       return Fib(N-1) + Fib(N-2);
7   }
```

```
long long Fib(size_t N)
{
    if (N < 3)
        return 1;

    return Fib(N - 1) + Fib(N - 2);
}
```



总结

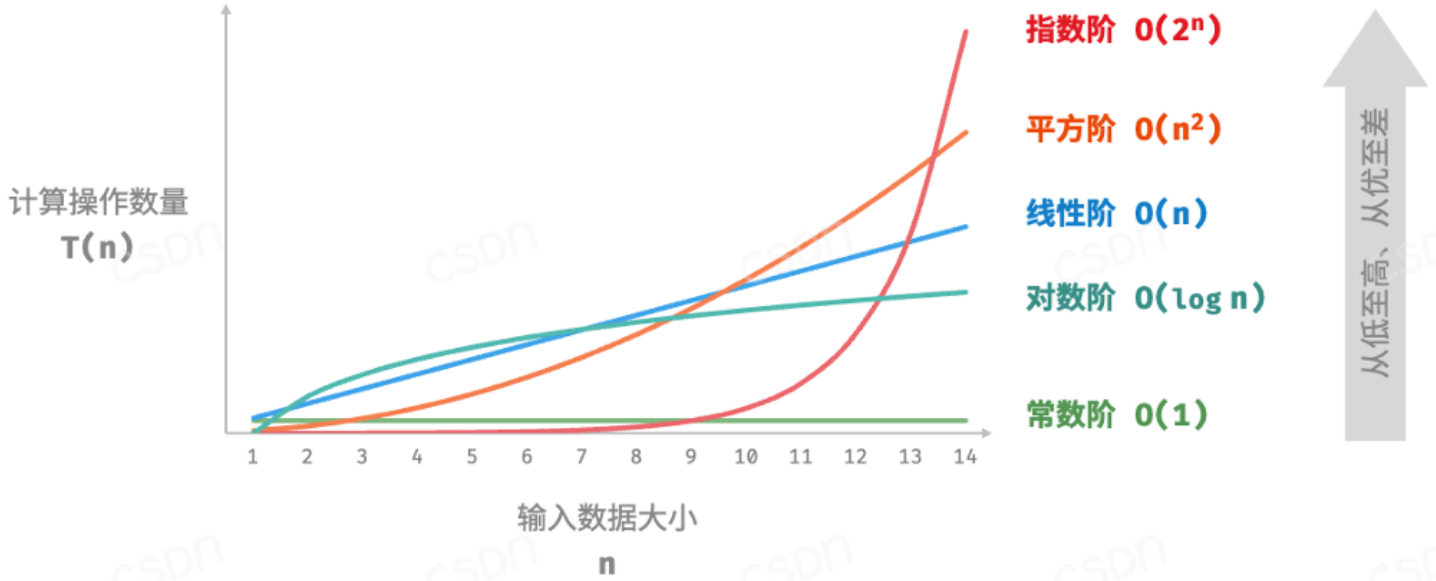


图 2-9　常见的时间复杂度类型

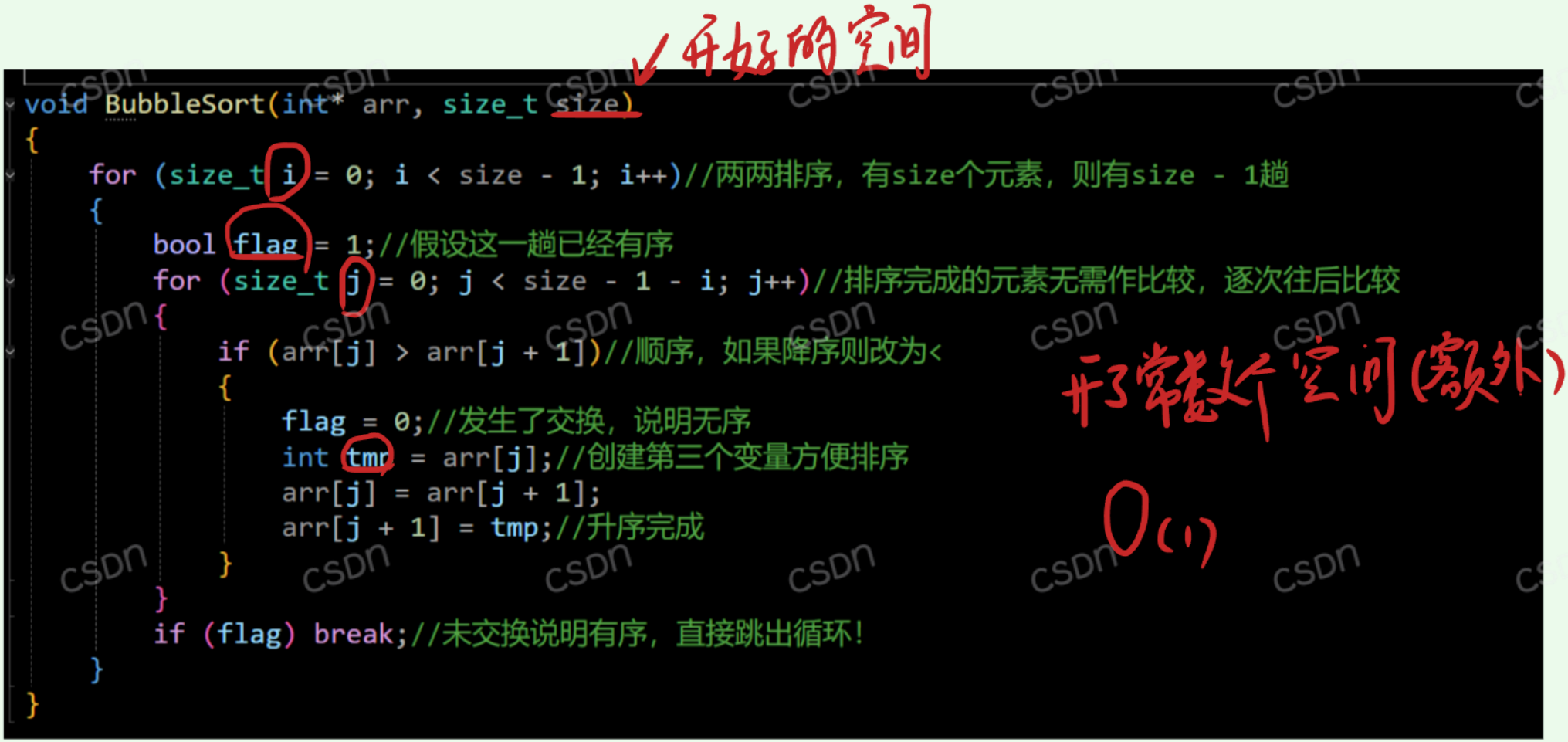
空间复杂度

算法的空间复杂度是一个函数，描述一个算法实现临时占用存储空间大小的量度

注意：**函数运行时所需要的栈空间(存储参数、局部变量、一些寄存器信息等)在编译期间已经确定好了，因此空间复杂度主要通过函数在运行时候显式申请的额外空间来确定**

```
1 void BubbleSort(int* arr, size_t size)
2 {
3     for (size_t i = 0; i < size - 1; i++)//两两排序，有size个元素，则有size - 1趟
```

```
    {
        bool flag = 1;//假设这一趟已经有序
        for (size_t j = 0; j < size - 1 - i; j++)//排序完成的元素无需作比较，逐次往后比较
        {
            if (arr[j] > arr[j + 1])//顺序，如果降序则改为<
            {
                flag = 0;//发生了交换，说明无序
                int tmp = arr[j];//创建第三个变量方便排序
                arr[j] = arr[j + 1];
                arr[j + 1] = tmp;//升序完成
            }
        }
        if (flag) break;//未交换说明有序，直接跳出循环！
    }
}
```




```
// 计算阶乘递归Fac的空间复杂度？
long long Fac(size_t N)
{
if(N == 0)
return 1;
return Fac(N-1)*N;
```

```
//计算阶乘递归Fac的空间复杂度?
long long Fac(size_t N)
{
    if (N == 0)
        return 1;
    return Fac(N - 1) * N;
}
```

O(n)

开辟的栈帧空间是可以复用的！！！
但时间不行 时间一去不复返
只有在函数生命周期结束时，
这个函数的栈帧空间才会被销毁

总结

- 算法和数据结构的关系
- 常见复杂度分类
- 时间复杂度和空间复杂度具体计算方法