1 算法复杂度

> 课程介绍

1. 比特数据结构课程分为初阶数据结构和高阶数据结构，当前课程是数据结构初阶
2. 在初阶课程中我们将掌握顺序表、链表、栈和队列、二叉树、常见排序算法等内容；高阶数据结构后续将以录播形式上传到班级课程中
3. 初阶数据结构中我们将继续使用C语言来实现基础的数据结构，在掌握数据结构的同时巩固了刚结束的C语法知识
4. 图、哈希表、红黑树等数据结构将在C++课程中学习

1. 数据结构前言

1.1 数据结构

数据结构(Data Structure)是计算机存储、组织数据的方式，指相互之间存在一种或多种特定关系的数据元素的集合。没有一种单一的数据结构对所有用途都有用，所以我们要学各式各样的数据结构，如：线性表、树、图、哈希等

1.2 算法

算法(Algorithm):就是定义良好的计算过程，他取一个或一组的值为输入，并产生出一个或一组值作为输出。简单来说算法就是一系列的计算步骤，用来将输入数据转化成输出结果。

1.3 数据结构和算法的重要性

校园招聘笔试必考

校园招聘面试必考

深信服校园招聘c/c 软件开发A卷 企业授权

匹配职位

C++工程师

题型数量

总题数	编程题
3	3

科大讯飞23校招研发类 企业授权

匹配职位

Java工程师、C++工程师、iOS工程师、安卓工程师、运维工程师、前端工程师、算法工程师、PHP工程师、测试工程师、安全工程师、数据库工程师、大数据开发工程师、数据分析师、游戏研发工程师、golang工程师、测试开发工程师、信息技术岗、区块链

题型数量

总题数	编程题
3	3

腾讯QQ部门 后台开发二面（C++实习）

发布于03-06 10:53

1. 什么时候开始接触计算的？平常是怎样学习的？
2. 了解过C++20的特性吗？（我说我了解过C++11）
3. 介绍一下智能指针是干什么用的？是怎么实现的？
4. 介绍一下你的两个项目
5. 说一说这个项目为什么比malloc函数快？
6. 你有没有思考过为什么C语言中要使用malloc而不使用这个项目？这个项目不是更优吗
7. 做这两个项目的时候有没有遇见什么难点？最后是怎样解决的？
8. 这两个项目有没有什么缺陷？使用起来有bug吗？
9. 手撕算法题：判断链表中是否存在环？（杭哥上课讲的原题，包会的）
10. 手撕算法题：找出两个数组中最长的相同的子数组（dp问题，看了算法课就能做）
11. 手撕算法题：全排列（算法课原题，秒了）
12. 公司在深圳，你能来吧？什么时候可以到岗？实习时长多久？
13. 平时有使用过GPT解决代码中出现的问题吗？（它们部门好像经常用）

面试官说他很讨厌八股文，所以问的问题全是和项目相关的，你根本水不了
最后反问环节他回答的：公司注重语言基础，独立思考和找到解决问题的方法的能力

猎豹科技-C++开发工程师(实习) - 24min

面试官

数组和链表的区别
数组和链表插入,删除,查找的时间复杂度
数组中随机删除一个元素的方法
链表中找到中间节点的方法
介绍一下最小堆
最小堆的插入删除查找的时间复杂度
TCP三次握手流程
Epoll的LT和ET的区别
ET工作模式下什么才算是有事件(边沿触发，缓冲区出现变化)
归并排序

如何学好数据结构和算法

秘诀1：

死磕代码！！！

秘诀2：

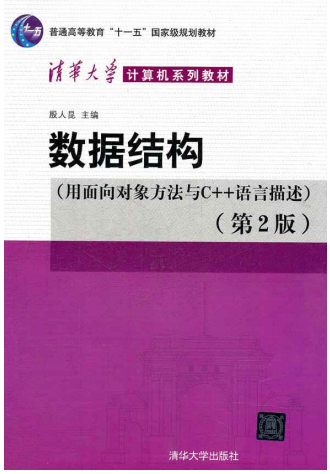
画图画图画图+思考

> 书籍推荐

《数据结构》严蔚敏

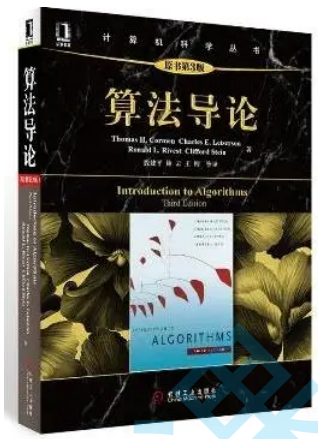
推荐理由：C语言版本，内容详尽，代码规整，各大院校指定教材

《数据结构》殷人昆



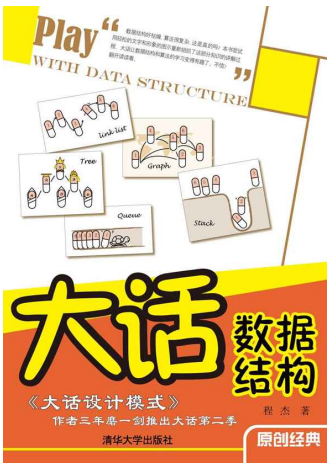
推荐理由：C++版本，内容详尽，代码规整

《算法导论》Thomas H. Cormen (托马斯·科尔曼)



推荐理由：叙述严谨，内容全面，深入讨论各类算法

《大话数据结构》程杰



推荐理由：

趣味易读，算法讲解细致深刻

2. 算法效率

> 如何衡量一个算法的好坏呢？
>
> 案例：旋转数组https://leetcode.cn/problems/rotate-array/description/
>
> 思路：循环K次将数组所有元素向后移动一位

```
1  void rotate(int* nums, int numsSize, int k) {
2      while(k--)
3      {
4          int end = nums[numsSize-1];
5          for(int i = numsSize - 1;i > 0 ;i--)
6          {
7              nums[i] = nums[i-1];
8          }
9          nums[0] = end;
10     }
11 }
```

代码点击执行可以通过，然而点击提交却无法通过，那该如何衡量其好与坏呢？

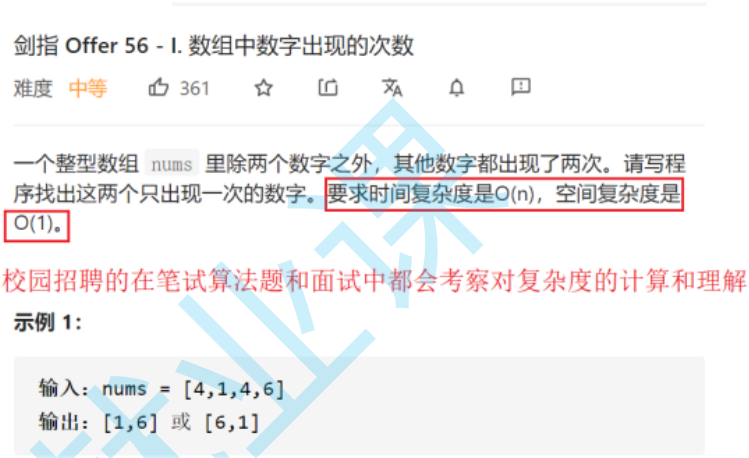
2.1 复杂度的概念

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。

2.2 复杂度的重要性

复杂度在校招中的考察已经很常见，如下：



3. 时间复杂度

定义：在计算机科学中，**算法的时间复杂度是一个函数式T(N)**，它定量描述了该算法的运行时间。时间复杂度是衡量程序的时间效率，那么为什么不去计算程序的运行时间呢？

1. 因为程序运行时间和编译环境和运行机器的配置都有关系，比如同一个算法程序，用一个老编译器进行编译和新编译器编译，在同样机器下运行时间不同。
2. 同一个算法程序，用一个老低配置机器和新高配置机器，运行时间也不同。
3. 并且时间只能程序写好后测试，不能写程序前通过理论思想计算评估。

那么算法的时间复杂度是一个函数式T(N)到底是什么呢？这个T(N)函数式计算了程序的执行次数。通过c语言编译链接章节学习，我们知道算法程序被编译后生成二进制指令，程序运行，就是cpu执行这些编译好的指令。那么我们通过程序代码或者理论思想计算出程序的执行次数的函数式T(N)，假设每句指令执行时间基本一样(实际中有差别，但是微乎其微)，那么执行次数和运行时间就是等比正相关，这样也脱离了具体的编译运行环境。执行次数就可以代表程序时间效率的优劣。比如解决一个问题的算法a程序T(N) = N，算法b程序T(N) = N^2，那么算法a的效率一定优于算法b。

案例：

```
// 请计算一下Func1中++count语句总共执行了多少次?
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;
    }
}
```

Func1 执行的基本操作次数：

$T(N) = N^2 + 2 * N + 10$

- N = 10　　T(N) = 130
- N = 100　　T(N) = 10210
- N = 1000　　T(N) = 1002010

通过对N取值分析，对结果影响最大的一项是 N^2

实际中我们计算时间复杂度时，计算的也不是程序的精确的执行次数，精确执行次数计算起来还是很麻烦的(不同的一句程序代码，编译出的指令条数都是不一样的)，计算出精确的执行次数意义也不大，因为我们计算时间复杂度只是想比较算法程序的增长量级，也就是当N不断变大时T(N)的差别，上面我们已经看到了当N不断变大时常数和低阶项对结果的影响很小，所以我们只需要**计算程序能代表增长量级的大概执行次数，复杂度的表示通常使用大O的渐进表示法。**

3.1 大O的渐进表示法

大O符号（Big O notation）：是用于描述函数渐进行为的数学符号

推导大O阶规则

1. 时间复杂度函数式T(N)中，只保留最高阶项，去掉那些低阶项，因为当N不断变大时，低阶项对结果影响越来越小，当N无穷大时，就可以忽略不计了。
2. 如果最高阶项存在且不是1，则去除这个项目的常数系数，因为当N不断变大，这个系数对结果影响越来越小，当N无穷大时，就可以忽略不计了。
3. T(N)中如果没有N相关的项目，只有常数项，用常数1取代所有加法常数。

通过以上方法，可以得到 `Func1` 的时间复杂度为： $O(N^2)$

3.2 时间复杂度计算示例

3.2.1 示例1

```
// 计算Func2的时间复杂度?
void Func2(int N)
{
    int count = 0;
    for (int k = 0; k < 2 * N ; ++ k)
    {
        ++count;
    }
    int M = 10;
    while (M--)
    {
        ++count;
    }
    printf("%d\n", count);
}
```

Func2执行的基本操作次数：

$T(N) = 2N + 10$

根据推导规则第3条得出

Func2的时间复杂度为： $O(N)$

3.2.2 示例2

```
// 计算Func3的时间复杂度?
void Func3(int N, int M)
{
    int count = 0;
    for (int k = 0; k < M; ++ k)
    {
        ++count;
    }
    for (int k = 0; k < N ; ++
k)
    {
        ++count;
    }
    printf("%d\n", count);
}
```

Func3执行的基本操作次数：

$T(N) = M + N$

因此：Func2的时间复杂度为： $O(N)$

3.2.3 示例3

Func4执行的基本操作次数：

```c
// 计算Func4的时间复杂度?
void Func4(int N)
{
    int count = 0;
    for (int k = 0; k < 100; ++ k)
    {
        ++count;
    }
    printf("%d\n", count);
}
```

$T(N) = 100$

根据推导规则第1条得出

Func2的时间复杂度为：$O(1)$

3.2.4 示例4

```c
// 计算strchr的时间复杂度?
const char * strchr ( const char
* str, int character)
{
    const char* p_begin = s;
    while (*p_begin != character)
    {
    if (*p_begin == '\0')
        return NULL;
    p_begin++;
}
    return p_begin;
}
```

strchr执行的基本操作次数：

1）若要查找的字符在字符串第一个位置，则：

$T(N) = 1$

2）若要查找的字符在字符串最后的一个位置，则：

$T(N) = N$

3）若要查找的字符在字符串中间位置，则：

$T(N) = \frac{N}{2}$

因此：**strchr的时间复杂度分为：**

最好情况：$O(1)$

最坏情况：$O(N)$

平均情况：$O(N)$

总结

通过上面我们会发现，有些算法的时间复杂度存在最好、平均和最坏情况。

最坏情况：任意输入规模的最大运行次数(上界)

平均情况：任意输入规模的期望运行次数

最好情况：任意输入规模的最小运行次数(下界)

大O的渐进表示法在实际中一般情况关注的是算法的上界，也就是最坏运行情况。

3.2.5 示例5

BubbleSort执行的基本操作次数：

```
// 计算BubbleSort的时间复杂度?
void BubbleSort(int* a, int n)
{
    assert(a);
    for (size_t end = n; end > 0; --end)
    {
        int exchange = 0;
        for (size_t i = 1; i < end; ++i)
        {
            if (a[i-1] > a[i])
            {
                Swap(&a[i-1], &a[i]);
                exchange = 1;
            }
        }
        if (exchange == 0)
                break;
    }
}
```

1）若数组有序，则：

$T(N) = N$

2）若数组有序且为降序，则：

$T(N) = \frac{N * (N+1)}{2}$

3）若要查找的字符在字符串中间位置，则：

因此：BubbleSort**的时间复杂度取最差情况为：** $O(N^2)$

3.2.6 示例6

```
void func5(int n)
{
    int cnt = 1;
    while (cnt < n)
    {
        cnt *= 2;
    }
}
```

当n=2时，执行次数为1

当n=4时，执行次数为2

当n=16时，执行次数为4

假设执行次数为 x ，则 $2^x = n$

因此执行次数： $x = \log n$

因此：func5**的时间复杂度取最差情况为：** $O(\log_2 n)$

> 注意课件中和书籍中 $\log_2 n$ 、 $\log n$、 $\lg n$ 的表示
>
> 当n接近无穷大时，底数的大小对结果影响不大。因此，一般情况下不管底数是多少都可以省略不写，即可以表示为 $\log n$
>
> 不同书籍的表示方式不同，以上写法差别不大，我们建议使用 $\log n$

3.2.7 示例7

```
// 计算阶乘递归Fac的时间复杂度?
long long Fac(size_t N)
{
    if(0 == N)
```

调用一次Fac函数的时间复杂度为 $O(1)$

而在Fac函数中，存在n次递归调用Fac函数

因此：

```
5         return 1;
6
7     return Fac(N-1)*N;
8 }
```

阶乘递归的**时间复杂度为**：$O(n)$

4. 空间复杂度

空间复杂度也是一个数学表达式，是对一个算法在运行过程中因为算法的需要**额外临时开辟的空间**。

空间复杂度不是程序占用了多少bytes的空间，因为常规情况每个对象大小差异不会很大，所以空间复杂度算的是变量的个数。

空间复杂度计算规则基本跟实践复杂度类似，也使用**大O渐进表示法**。

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

4.1 空间复杂度计算示例

4.1.1 示例1

```
1  // 计算BubbleSort的时间复杂度?
2  void BubbleSort(int* a, int n)
3  {
4      assert(a);
5      for (size_t end = n; end > 0; --end)
6      {
7          int exchange = 0;
8          for (size_t i = 1; i < end; ++i)
9          {
10             if (a[i-1] > a[i])
11             {
12                 Swap(&a[i-1], &a[i]);
13                 exchange = 1;
14             }
15         }
16         if (exchange == 0)
17                 break;
18     }
19 }
```

函数栈帧在编译期间已经确定好了，只需要关注函数在运行时额外申请的空间。

BubbleSort额外申请的空间有exchange等有限个局部变量，使用了常数个额外空间

因此空间复杂度为 $O(1)$

4.1.2 示例2

```
// 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;
}
```

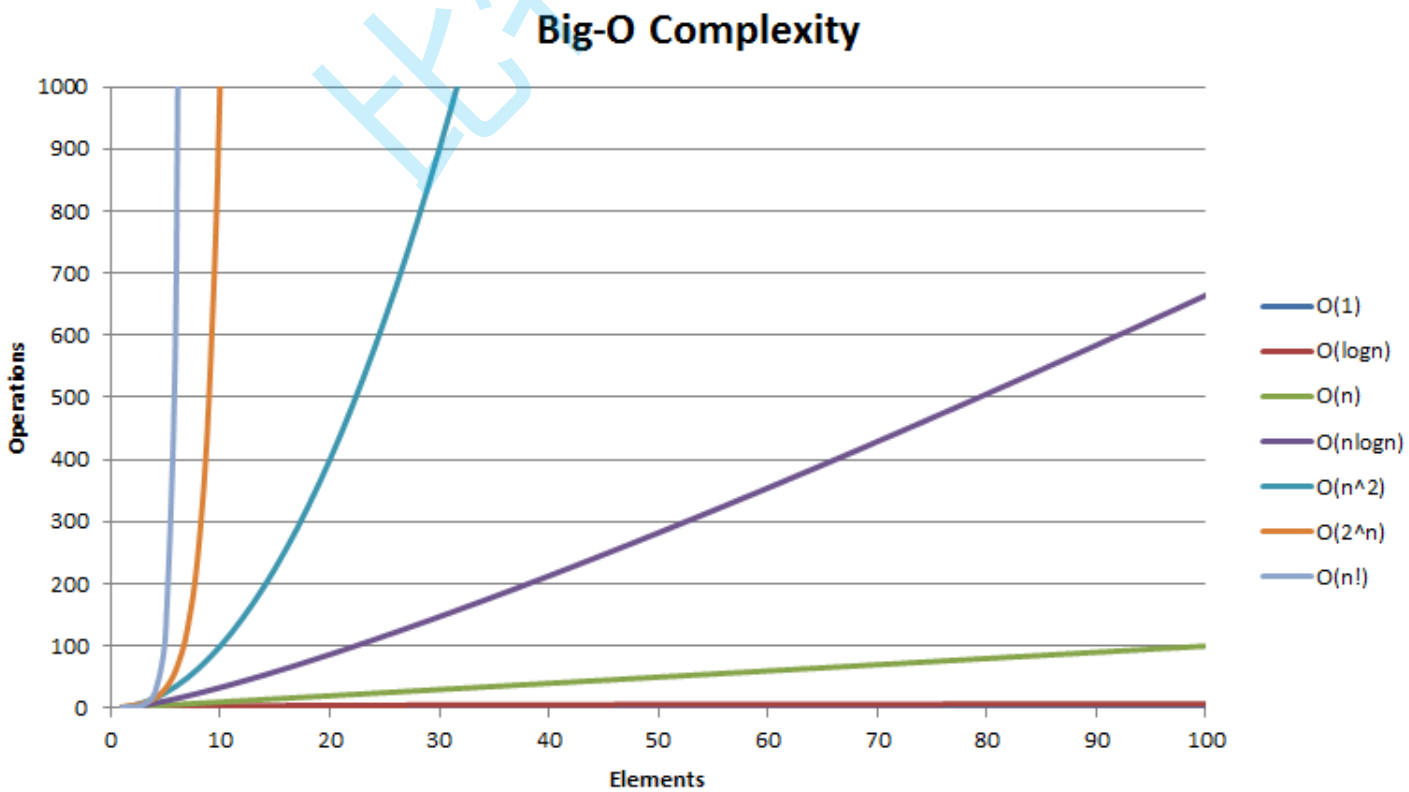
Fac递归调用了N次，额外开辟了N个函数栈帧，每个栈帧使用了常数个空间

因此空间复杂度为：$O(N)$

5. 常见复杂度对比

表 1.4　各个函数随 n 的增长函数值的变化情况

n	$\log_2 n$	$n\times\log_2 n$	n^2	n^3	2^n	$n!$
4	2	8	16	64	16	24
8	3	24	64	512	256	80320
10	3.32	33.2	100	1000	1024	3628800
16	4	64	256	4096	65536	2.1×10^{13}
32	5	160	1024	32768	4.3×10^{9}	2.6×10^{35}
128	7	896	16384	2097152	3.4×10^{38}	∞
1024	10	10240	1048576	1.07×10^{9}	∞	∞
10000	13.29	132877	10^{8}	10^{12}	∞	∞



6. 复杂度算法题

6.1 旋转数组

https://leetcode.cn/problems/rotate-array/description/

思路1

> 时间复杂度 $O(n^2)$
>
> 循环K次将数组所有元素向后移动一位（代码不通过）

```
void rotate(int* nums, int numsSize, int k) {
    while(k--)
    {
        int end = nums[numsSize-1];
        for(int i = numsSize - 1;i > 0 ;i--)
        {
            nums[i] = nums[i-1];
        }
        nums[0] = end;
    }
}
```

思路2：

> 空间复杂度 $O(n)$
>
> 申请新数组空间，先将后k个数据放到新数组中，再将剩下的数据挪到新数组中

```
void rotate(int* nums, int numsSize, int k)
{
    int newArr[numsSize];
    for (int i = 0; i < numsSize; ++i)
    {
        newArr[(i + k) % numsSize] = nums[i];
    }
    for (int i = 0; i < numsSize; ++i)
    {
        nums[i] = newArr[i];
    }
}
```

思路3：

> 空间复杂度 $O(1)$

- 前n-k个逆置：4321567
- 后k个逆置 ：4321765
- 整体逆置 ：5671234

```c
void reverse(int* nums,int begin,int end)
{
    while(begin<end){
        int tmp = nums[begin];
        nums[begin] = nums[end];
        nums[end] = tmp;

        begin++;
        end--;
    }
}
void rotate(int* nums, int numsSize, int k)
{
    k = k%numsSize;
    reverse(nums,0,numsSize-k-1);
    reverse(nums,numsSize-k,numsSize-1);
    reverse(nums,0,numsSize-1);
}
```