小马加编信息学教案(十七)

算法的时空复杂度

一. 课程内容

1. 算法时间复杂度的概念
2. 算法时间复杂度的估计
3. 算法空间复杂度的估计

二. 知识讲解

1. 算法时间复杂度的概念

> 时间复杂度是同一问题可用不同算法解决，而一个算法的质量优劣将影响到算法乃至程序的效率。
> **算法分析的目的在于选择合适算法和改进算法。**
> 这也正是算法竞赛所需要关注与考查的重中之重。

> 在计算机科学中，算法的时间复杂度是一个函数，它定性描述了该算法的运行时间。这是一个关于代表算法输入值的字符串的长度的函数。
> 简单地说,算法的时间复杂度是一个算法在面对大规模的数据增长时所需要花费时间的增长速度的衡量标准。

2. 算法的时间复杂度的估计

2.1 计算机运行速度

- 首先，我们要对计算机的运行速度有一个大致的了解。

> **NOI系列考试所用的计算机的运行速度大致为每秒可以运算10^8次(一亿次)**

2.2 算法时间复杂度的定义

> 我们假设计算机运行一行基础代码需要执行一次运算。

```
int aFunc(void) {
    printf("Hello, World!\n");      //  需要执行 1 次
    return 0;       // 需要执行 1 次
}
```

> 那么上面这个方法需要执行 2 次运算

```
int aFunc(int n) {
    for(int i = 0; i<n; i++) {          // 需要执行 (n + 1) 次
        printf("Hello, World!\n");       // 需要执行 n 次
    }
    return 0;       // 需要执行 1 次

}
```

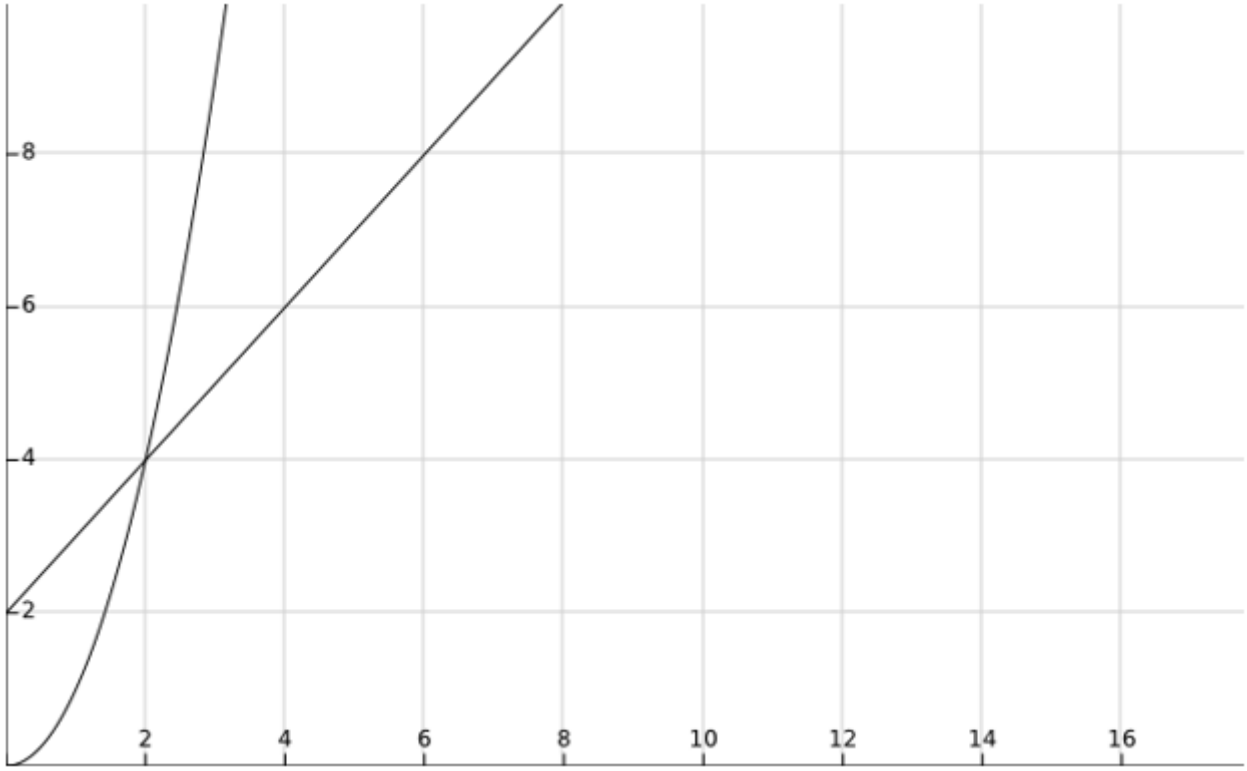
> 这个方法需要 $(n+1+n+1)=2n+2$ 次运算。

- **我们把算法需要执行的运算次数用输入大小 n 的函数 表示，即 $T(n)$ 。**

> 此时为了 **估算算法需要的运行时间** 和 **简化算法分析**，我们引入时间复杂度的概念。

- 以下内容有些专业化，但只需要了解其大概含义与使用方法即可。

> 定义： 存在常数 c，使得当 $N \geq c$ 时 $T(N) \leq f(N)$，表示为 $T(n)=O(f(n))$ 。



> 当 $N \geq 2$ 的时候，$f(n) = n^2$ 总是大于 $T(n) = n + 2$ 的，于是**我们说 $f(n)$ 的增长速度是大于或者等于 $T(n)$ 的**，也说 $f(n)$ 是 $T(n)$ 的上界，可以表示为 $T(n) = O(f(n))$。

> 因为 $f(n)$ 的增长速度是大于或者等于 $T(n)$ 的，即 $T(n) = O(f(n))$ ，所以我们可以用 $f(n)$ 的增长速度来度量 $T(n)$ 的增长速度，所以我们说这个算法的时间复杂度是 $O(f(n))$。

> 所以我们知道：
> **算法的时间复杂度，用来度量算法的运行时间，记作: $T(n) = O(f(n))$。它表示随着 输入大小 n 的增大，算法执行需要的时间的增长速度可以用 $f(n)$ 来描述。**

> 显然如果 $T(n) = n^2$，那么 $T(n) = O(n^2)$，$T(n) = O(n^3)$，$T(n) = O(n^4)$ 都是成立的，但是因为第一个 $f(n)$ 的增长速度与 $T(n)$ 是最接近的，所以第一个是最好的选择，所以我们说这个算法的复杂度是 $O(n^2)$ 。

2.3 算法时间复杂度的估算

- 那么当我们拿到算法的执行次数函数 T(n) 之后怎么得到算法的时间复杂度呢？

> (1) 我们知道常数项并不影响函数的增长速度，所以当 $T(n) = c$ ，c 为一个常数的时候，我们说这个算法的时间复杂度为 $O(1)$ ；如果 $T(n)$ 不等于一个常数项时，直接将常数项省略。
> 例如:
> 第一个 `Hello, World` 的例子中 $T(n) = 2$，所以我们说那个函数(算法)的时间复杂度为 $O(1)$。
> 若$T(n) = n + 29$ ，此时时间复杂度为 $O(n)$ 。

> (2) 我们知道高次项对于函数的增长速度的影响是最大的。n^3 的增长速度是远超 n^2 的，同时 n^2 的增长速度是远超 n 的。 同时因为要求的精度不高，所以我们直接忽略低此项。
> 例如:
> $T(n) = n^3 + n^2 + 29$ ，此时时间复杂度为 $O(n^3)$。

> (3) 因为函数的阶数对函数的增长速度的影响是最显著的，所以我们忽略与最高阶相乘的常数。
> 例如:
> $T(n)=3n^3$ ，此时时间复杂度为 $O(n^3)$。

> 综合起来：如果一个算法的执行次数是 $T(n)$，那么只保留最高次项，同时忽略最高项的系数后得到函数 $f(n)$，此时算法的时间复杂度就是 $O(f(n))$。为了方便描述，下文称此为 大O推导法。

表8-1　运算量随着规模的变化

运算量	$n!$	2^n	n^3	n^2	$n\log_2 n$	n
最大规模	11	26	464	10000	$4.5*10^6$	100000000
速度扩大两倍后	11	27	584	14142	$8.6*10^6$	200000000

	T（n）= 100n X 100	T（n）= 5n^2
n = 1	10 000	5
n = 5	50 000	125
n = 10	100 000	500
n = 100	1 000 000	50000
n = 1 000	10 000 000	5 000 000
n = 10 000	100 000 000	500 000 000
n = 100 000	1 000 000 000	50 000 000 000
n = 1 000 000	10 000 000 000	5 000 000 000 000

2.4 计算次数$T(n)$的估算方法

> 很多时候，由执行次数 $T(n)$ 得到时间复杂度并不困难，很多时候困难的是从算法通过分析和数学运算得到 $T(n)$。
> 对此，提供下列四个便利的法则，这些法则都是可以简单推导出来的，总结出来以便提高效率。

> (1) 对于一个循环，假设循环体的时间复杂度为 $O(n)$ ，循环次数为 m，则这个循环的时间复杂度为 $O(n\times m)$。

```
void aFunc(int n) {
    for(int i = 0; i < n; i++) {          // 循环次数为 n
        printf("Hello, World!\n");       // 循环体时间复杂度为 O(1)
    }
}
```

> 此时时间复杂度为 $O(n\times 1)$，即 $O(n)$。

> (2) 对于多个循环，假设循环体的时间复杂度为 $O(n)$，各个循环的循环次数分别是$a, b, c...$，则这个循环的时间复杂度为 $O(n \times a \times b \times c...)$。
> 分析的时候应该由里向外分析这些循环。

```
void aFunc(int n) {
    for(int i = 0; i < n; i++) {          // 循环次数为 n
        for(int j = 0; j < n; j++) {        // 循环次数为 n
            printf("Hello, World!\n");       // 循环体时间复杂度为 O(1)
        }
    }
}
```

> 此时时间复杂度为 $O(n \times n \times 1)$，即 $O(n^2)$。

> (3) 对于顺序执行的语句或者算法，总的时间复杂度等于其中最大的时间复杂度。

```
void aFunc(int n) {
    // 第一部分时间复杂度为 O(n^2)
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < n; j++) {
            printf("Hello, World!\n");
        }
    }
    // 第二部分时间复杂度为 O(n)
    for(int j = 0; j < n; j++) {
        printf("Hello, World!\n");
    }
}
```

> 此时时间复杂度为 $max(O(n^2), O(n))$，即 $O(n^2)$。

> (4) 对于条件判断语句，总的时间复杂度等于其中 时间复杂度最大的路径 的时间复杂度。

```
void aFunc(int n) {
    if (n >= 0) {
        // 第一条路径时间复杂度为 O(n^2)
        for(int i = 0; i < n; i++) {
            for(int j = 0; j < n; j++) {
                printf("输入数据大于等于零\n");
            }
        }
    } else {
        // 第二条路径时间复杂度为 O(n)
        for(int j = 0; j < n; j++) {
            printf("输入数据小于零\n");
        }
    }
}
```

> 此时时间复杂度为 $max(O(n^2), O(n))$，即 $O(n^2)$。

- 时间复杂度分析的基本策略是：从内向外分析，从最深层开始分析。如果遇到函数调用，要深入函数进行分析。

2.5 时间复杂度的应用

- 根据题目中的数据范围，我们可以根据算法时间复杂度大致估计我们的算法是否会超时。

算法复杂度	建议不超过的n的范围
$O(logn)$	很大,$long\ long$ 以内都可以
$O(n)$	10^7
$O(n\ logn)$	10^5 ~ $5*10^5$
$O(n^2)$	1000 ~ 5000
$O(n^3)$	200 ~ 500
$O(2^n)$	20 ~ 24
$O(n!)$	12

3. 算法空间复杂度的估计

- $NOIP$系列赛事一般考查的是算法的时间复杂度，而一般不会卡空间

> 但我们对于空间复杂度也需要有一个大致的了解

> 空间复杂度与我们之前讨论的时间复杂度大致相同

- 类似于时间复杂度的讨论，一个算法的空间复杂度(Space Complexity)S(n)定义为该算法所耗费的存储空间，它也是问题规模n的函数。渐近空间复杂度也常常简称为空间复杂度。

> 一般来说我们的程序中的空间消耗主要在于:

- 静态数组的内存大小
- 函数调用的空间大小

> 我们一般考虑数组内存大小并留出一定空间给函数调用以及程序运行即可

> 总体来讲，就是多开了一个辅助的数组：$O(n)$，多开了一个辅助的二维数组:$O(n^2)$,多开常数空间：$O(1)$

> 题目中的空间范围一般是$256MB$或$512MB$
> 而一个 `int` 的内存大小为$4B$,所以在不考虑别的空间消耗时，我们可以分配67108864或134217728个 `int` 的内存。

> 但实际上，我们一般保证我们的程序中所开的数组大小不超过千万级别。
> 具体大小可根据数据类型大小以及所开规模计算会不会爆空间。

三. 经典例题

- 尝试阅读观察下面程序，并估算其时间复杂度大小

```
void aFunc(int n) {
    for (int i = 0; i < n; i++) {
        for (int j = i; j < n; j++) {
            printf("Hello World\n");
        }
    }
}
```

```
void aFunc(int n) {
    for(i = 0; i < n; i++) {
        for(j = 0; j < i; j++) {
            for(k = 0; k < j; k++)
                x += 2;
        }
    }
}
```

```
void aFunc(int n) {
    for (int i = 2; i < n; i++) {
        i *= 2;
        printf("%i\n", i);
    }
}
```

```
long aFunc(int n) {
    if (n <= 1) {
        return 1;
    } else {
        return aFunc(n - 1) + aFunc(n - 2);
    }
}
```

四. 提高巩固

最大连续和问题

> 给出一个长度为 n 的序列$A_1, A_2, \ldots, A_n$，求最大连续和。
> 换句话说，要 求找到$1 \leq i \leq j \leq n$，使得$A_i + A_{i+1} + \ldots + A_j$ 尽量大。

> 尝试在老师的帮助下阅读理解下面代码，并分析得出它们的时间复杂度

```
tot = 0;
best = A[1]; //初始最大值
for(int i = 1; i <= n; i++)
    for(int j = i; j <= n; j++) {        //检查连续子序列A[i],..., A[j]
        int sum = 0;
        for(int k = i; k <= j; k++) {
            sum += A[k];
            tot++;
        }                        //累加元素和
        if(sum > best)
            best = sum;         //更新最大值
    }


S[0] = 0;
for(int i = 1; i <= n; i++)
    S[i] = S[i - 1] + A[i];              //递推前缀和S
for(int i = 1; i <= n; i++)
    for(int j = i; j <= n; j++)
        best = max(best, S[j] - S[i - 1]); //更新最大值


int maxsum(int A[], int x, int y) { //返回数组在左闭右开区间[x,y)中的最大连续和
    int v, L, R, maxs;
    if(y - x == 1)
        return A[x];                          //只有一个元素，直接返回

    int m = x + (y - x)/2;         //分治第一步：划分成[x, m)和[m, y)

    int maxs = max(maxsum(A, x, m),maxsum(A, m, y));    //分治第二步：递归求解

    int v, L, R; v = 0; L = A[m - 1];          //分治第三步：合并(1)——从分界点开始往左的最大连续和L

    for(int i = m-1; i >= x; i--)
        L = max(L, v += A[i]);
    v = 0; R = A[m];              //分治第三步：合并(2)——从分界点开始往右的最大连续和R

    for(int i = m; i < y; i++)
        R = max(R, v += A[i]);
    return max(maxs, L+R);     //把子问题的解与L和R比较
}
```