0.1 记号定义

定义 1： 如果存在正的常数 c 和 n_0 使得，对于任何 $N \geqslant n_0$ 时总有 $T(N) \leqslant cf(N)$，那么就使用记号 $T(N) = O(f(N))$。

定义 2： 如果存在正的常数 c 和 n_0 使得，对于任何 $N \geqslant n_0$ 时总有 $T(N) \geqslant cg(N)$，那么就使用记号 $T(N) = \Omega(g(N))$。

定义 3: 如果我们有关系式 $T(N) = O(h(N))$，并且同时还具有关系式 $T(N) = \Omega(h(N))$，那么我们使用记号 $T(N) = \Theta(h(N))$。

定义 4: 如果我们具有关系 $T(N) = O(p(N))$，但是不具有关系 $T(N) = \Theta(p(N))$，那么我们使用记号 $T(N) = o(p(N))$。

此处的记号定义非常类似于数字的比较，我们可以很简单地将定义 1 类比于数字比较的小于或等于，意即如果我们有关系 $T(N) = O(f(N))$，那么我们可以"认为" $T(N) \leqslant f(N)$，同样地，对于定义 2，我们类比于关系小于或等于；显然定义三就类比于等于，而定义 4 可以类比于小于。注意，此处的出现的 $T(N), f(N), g(N), p(N)$ 等，表明是一些关于 N 的函数，而 N 的取值一般限于整数。在算法分析中，我们会使用这些记号来度量算法之间的增长关系，即比较算法的运算时间的复杂程度。

这里的定义，直观地看来，与序列极限的定义十分类似；事实上，它们分别就是序列极限之比的另一种表述方式。我们来观察这样的极限表达式

$$\lim_{n\to\infty} \frac{T(N)}{f(N)} = r$$

如果 r
$= 0$，那么我们可以断言 $T(N) = o(f(N))$，如果 $r = c(0 < c < \infty)$，那么我们可以得到 $T(N) = \Theta(f(N))$，如果 $r = 0$，我们有 $f(N) = o(T(N))$。

0.2 T(N) 函数的意义

在算法分析中，我们不可能精确地估计算法需要执行的时间，这是因为，首先我们无法预计该算法会以什么样的编程语言形式，在什么样的计算机上运行；即便是我们在同一台计算机上进行编译，编译器的发行商甚至版本的不同，也会造成效率不一一的情况。所以，精确预计算法的运行时间是不可能的，但是，一个算法可能的运行时间快慢确实是可以量化估计的，

但不是以确定时间的形式，我们会代之以其他类似的形式——基本运算次数——来估计算法的运行效率。$T(N)$ 函数，就是这样一种函数，对于一个输入规摸 N，我们将会得到怎样量级的基本运算次数。

举例说明，如果 $T(N) = \Theta(N^3)$，此时，如果我们的输入规摸是 300，那么算法就需要 N^3 量级的运算次数，但是算法可能不一定就恰好进行 300^3 次基本运算，可能是它的某个常数倍，或者还有更多的出入，但是，当输入规摸 N 增加到非常大时，算法所需的基本运算次数会越来越趋近于 N^3。

我们进行算法分析时，一般讨论一个算法对于输入规模 N 的最坏情况，有时候我们也会研究算法的平均运算时间，不过这不总是可行的，因为对于很多算法，我们无法估计平均情况，甚至有时候我们根本无法定义每一层所谓的平均运算时间。这样虽然我们可能无法准确估计运行所需时间，但是我们可以给出一个上界，意味着对于这样的输入规摸，最坏的情况下，所需的最大运行时间不会超过我们得到的结果，但是可以小于这个结果；意味着我们可以估计，在这样的情况下，算法最久需要多长时间运行，这样也是很有用的情形。

0.3 算法分析例子

例子 1: 我们现在来对比两个排序算法的实例。

插入排序: 分析算法首先给出代码

```cpp
#include <iostream>
#include <random>
#include <time.h>
using namespace std;

int main()
{
	int n = 0;
	cout << "Please enter number of random numbers:";
	cin >> n;
	int *p = new int[n];
	srand(time(NULL));
	for(int i=0; i < n; i++){
	  p[i] = rand() % (2*n);
```

```
15          }

17            cout << "\n";
              for(int i = 0; i < n; i++){
19              cout << p[i] << " ";
            }
21            cout << "\n";

23            for(int i = 1; i < n; i++){
                int key = p[i];
25              int j = i - 1;
                while(j > -1 && p[j] > key){
27                  p[j+1] = p[j];
                    j -= 1;
29              }
                p[j+1] = key;
31          }

33          cout << "\n";
            for(int i = 0; i < n; i++){
35              cout << p[i] << " ";
            }
37          cout << "\n";
        }
```

插入排序

我们假定每个基本命令都将消耗一个单位的时间，如果遇到函数调用，除去调用的时间开销一个单位外，函数内部的时间开销也要进行计算。依照此约定，我们来分析这个代码，对于输入规摸是 N 时，需要消耗的时间。当然，在实际的计算机中，不可能每个基本命令只消耗一个单位的时间，而且在实际的计算机中，一个单位的时间没有意义，我们只是为了分析算法的时间复杂性而引入的这样一种度量方式，它会和实际运算时间成正相关关系。

每一层首先我们看看第 8 12 行代码，无论对于多大的输入规摸 N，我们都将执行 5 次基本命令，意味着我们会消耗（在理论算法上）5 个单位时

间;

$$T_1 = 5$$

而第 13——15 行代码是一个 for 循环，此时我们预计最坏的情形是要运行 N 次，而初始化一个内部循环变量 i 需要消耗一个时间单位，每次运行时将 i 与 n 进行比较需要消耗一个时间单位，共最多 N 次，而 i 在每次内 for 循环部语句运行完成后需要自增 1，每次也需要消耗 1 个单位时间，共最多 N 次，因此;

$$T_{2_1} = 2N + 1$$

再向内分析，这个 for 循环包含一个赋值语句，显然它们消耗时间多于一个单位，因为它同时包含多个操作这些操作有，取随机数 rand()，乘法 2*n，以及求余数%，数组下标操作，以及赋值；这些操作每次需要 5 个单位时间，因此经过 N 次循环需要的总时间为

$$T_{2_2} = 5N$$

每一层从而对于整个循环我们有

$$T_2 = T_{2_1} + T_{2_2} = 2N + 1 + 5N = 7N + 1$$

继续我们的分析，接下来第 17 行是一个流输出操作，我们认为它需要一个单位时间，因此

$$T_3 = 1$$

此处又是一个 for 循环，从而基本时间量同 T_{2_1}，而内部语句不同；此时的内部语句实质上是两次流输出操作，因此它每次需要 2 个单位时间，从而

$$T_4 = 2N + 1 + 2N = 4N + 1$$

类似的，对于第 21 行，有

$$T_5 = 1$$

第 23——31 行，是算法的核心部分，我们对它进行分析；首先是一个 for 循环，它需要基本时间量（最坏的情况）是 2N（因为是从 i=1 开始，比前述情况少执行一次，故而少 1），然后内部代码分为三部分，前两部分是定义变量，但是操作并非单一的，24 行有一个赋值操作与数组下标操作，消

耗 2 个时间，第 25 行有一个赋值操作与减法操作，也需要消耗 2 个时间，故而这一部分将消耗

$$t_{6_1} = (N-1)(2+2) = 4N-4$$

接下来是一个 while 循环，在最坏的情况下，循环每次判断需要两次比较操作以及一次求逻辑与的操作，因此在最坏的情况下，每次循环判断需要 3 个单位时间，而在最坏的情况下，显然需要进行 i-(-1)=i+1 次判断，至少在最后一次判断时条件必然为假，否则将造成死循环，而我们分析代码，它将不会死循环，因为每次循环进行，j 至少会少一个，而根据自然数的子序列必然存在有限下界的公理，因而循环次数必然为有限次，所以 while 循环体内部至多会执行 i 次。再看看 while 循环的内部情况，简要分析，我们知道第 27 行需要进行两次下标操作与一次赋值操作，28 行需要一次自减操作，每次循环需要执行 3 次基本操作，因而需要 3 个单位时间，从而整个 while 循环（在每次 for 循环内）需要至多 3i 个单位时间，从而对于全部的 for 循环而言，第二部分，即 while 循环部分，共需要时间

$$T_{6_2} = \sum_{i=1}^{N-1} 3i = 3\sum_{k=1}^{N-1} i = 3\frac{(N-1)(N-1+1)}{2} = \frac{3}{2}N(N-1)$$

第三部分只有一个语句，但是每次它都需要两个单位时间，因为需要下标操作与赋值操作，因此

$$T_{6_3} = 2(N-1) = 2N-2$$

因此，这整个 for 循环需要最多

$$T_6 = T_{6_1} + T_{6_2} + T_{6_3} = 4N-4+\frac{3}{2}N(N-1)+2N-2 = \frac{3}{2}N^2+\frac{9}{2}N-6$$

对于代码余下的部分，我们可以很轻松地发现

$$T_7 = 1+2N+1+3N = 5N+1$$

对于整个算法的运行时间上界，显然有

$$
\begin{aligned}
T(N) &= \sum_{i+1}^{7} T_i = 5 + 7N + 1 + 1 + 4N + 1 + 1 + 4N - 4 + \frac{3}{2}N^2 + \frac{9}{2}N - 6 + 5N + 1 \\
&= \frac{3}{2}N^2 + 5N - \frac{9}{2}N \\
&= \frac{3}{2}N^2 + \frac{1}{2}N \\
&= \frac{3N^2 + N}{2}
\end{aligned}
$$

这样，根据前述定义我们可以很显然地看出来，有关系

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

因此我们已经得到了这个算法的复杂性，称为“$O(N^2)$”的。

0.4 算法分析法则

我们在定义算法时间复杂性时(那些 $T(N) = O(f(N)), \Omega(g(N)), \Theta(h(N))$)，已经分析了它们和极限的内在关系，这是当 N 足够大时（大到能满足我们形容的性质时，因为这些表达式具有这样的性质，即存在某些极限，所以必然能在某个值，从那样的 N 之后的一切 N 都满足某种性质），复杂性函数的值的最主要部分，忽略其余的与它相比微不足道的部分（即除以它之后，分式表达式的极限将为 0)，并且，不看主要部分的系数，只看它的量级，我们就会得到算法在足够大的输入规模下的时间复杂性的渐进方式。

在这种意义下，我们对每个算法都进行例子中的分析将是及其不明智的，因为，有很多操作，它们是固定大小，这些值在充分大的输入规模下是微不足道，比如上述的 T_3 与 T_5，以及很多情况下，系数也是不重要的，上述中一切 T_i 的系数，这些都没必要进行分析。进而我们发现，在上述算法中，存在一个 for 循环和 while 循环的嵌套，这里的复杂性估算得到将是 N^2 量级的，因为每次循环在最坏的情况下，必然各自不会超过 N 次，因此对比与其他部分，在充分大的 N 的情况下，只有这里最重要，从而得到 $T(N) = O(N^2)$。

现在我们来简述一些一般的法则：

法则 1——for 循环：

非嵌套 for 循环的运行时间，最多消耗循环体内部加上 for 循环条件测试以及变量自增时间总和乘以迭代循环的次数。如果内部总消耗（加上 for 循环本身的测试等消耗）为函数 $f(N)$，那么 for 循环的时间复杂性 $T(N) = O(Nf(N))$。

法则 2——嵌套的 for 循环：

对于多个 for 循环的嵌套，假定一共有 m 层嵌套循环，第 i 层循环的非下一层 for 循环部分时间消耗为 $f_i(N)$，且进行 P_i 次迭代，那么，从内向外进行分析，第 $j(1 \leqslant J \leqslant m)$ 内部非下一个 for 循环部的时间消耗为第 1 到 j 层的所有循环次数 $P_k(1 \leqslant k \leqslant j)$ 的乘积，即 $T_j = O(P_1 P_2 \cdots P_j f_j(N))$，而总的时间消耗为

$$T(N) = \sum_{k=1}^{n} T_k = \sum_{k=1}^{n} O(f_k(N) \prod_{i=1}^{k} P_i)$$

对于 $O(f(N))$ 记号的性质，我们接下来会谈到，现在需要认识到，如果 $f(N)$ 包含很多项，那么其中最大量级的项将成为主要部分，其余部分与他相比将成为无穷小；因此，在此处，嵌套循环的过程中，一般来说最内层的量级最大，因而它将成为整个复杂性的决定因素。不过这并不绝对，如果在某个中间层中包含一个很复杂的函数调用，那么也有可能它成为决定性因素。

法则 3——顺序语句：

这是很显然的，各个顺序执行的部分时间消耗求和就是总时间消耗；其中如果存在相比于其他语句高阶大的（在输入规模趋于无穷时它与其他语句复杂性的比值趋于无穷）语句，那么它将起决定性作用假定有 m 个语句，每个语句的复杂性 $T_i(N) = O(f_i(N))$，而第 j 个语句的复杂性 $T_j(N)$ 相比与其他语句高阶大，那么就有

$$T(N) = \sum_{k=1}^{m} T_i = \sum_{k=1}^{m} O(f_k(N)) = O(\sum_{k=1}^{m} f_k(N)) = O(f_j(N))$$

法则 *4——if/else* 语句：

假设条件判断需要消耗时间为 $O(r(N))$，if 语句内部代码消耗时间为 $O(f_1(N))$，else 语句消耗时间为 $f_2(N)$，那么总消耗时间不会超过条件判断加上后二者中的最大值，即

$$T(N) = \max(O(r(N) + f_1(N)), O(r(N) + f_2(N)))$$