算法是为解一个问题需要遵循的、被清楚地指定的简单指令的集合。

# 1.数学基础：

## 1.1四个定义：

这四个定义的目的是在函数间建立一种相对的级别，比较其相对增长速率（即斜率）。T(N)为时间复杂度。

1.1.1如果存在常数c和n0使得当N>= n0时T(N)<=c·f(N)，则记为T(N)=O(f(N))。T(N)增长率<=f(N)增长率，f(N)是T(N)的一个上界。

1.1.2如果存在常数c和n0使得当N>= n0时T(N)>=c·g(N)，则记为T(N)=Ω(g(N))。

T(N)增长率>=f(N)增长率，f(N)是T(N)的一个下界。

1.1.3 T(N)=Θ(h(N))当且仅当T(N)=O(h(N))且T(N)= Ω(h(N))。T(N)增长率=f(N)增长率。

1.1.4如果T(N)=O(p(N))且T(N)≠Θ(h(N))，则T(N)= o(p(N))。T(N)增长率<f(N)增长率。

以定义一为例：最后总会存在某个点n0，从它以后c·f(N)总是至少与T(N)一样大。若忽略常数因子，则f(N)总是至少与T(N)一样大。对于f(N)=N2，T(N)=1000N，我们可以说1000N=O(N2)，即N平方级。

## 1.2三个结论：

1.2.1如果T1(N)=O(f(N))且T2(N)=O(g(N))，那么：

①T1(N)+T2(N)=max(Of(N),O(g(N))),

②T1(N)\*T2(N)=O(f(N)\*g(N))。

1.2.2如果T(N)是一个k次多项式，则T(N)= Θ(Nk)。

1.2.3对任意常数k，logkN=O(N)，这说明对数增长得十分缓慢。

## 1.3几点注意：

1.3.1将常熟或低阶项放进大O是非常坏的习惯（舍去）。在需要大O表示的分析中，各种简化都是可能发生的，要求的精度低。

1.3.2我们总能通过计算极限来确定两个函数的相对增长率：

①极限为0：意味f(N)=o(g(N))，

②极限为c，不为0：意味f(N)=Θ(g(N))，

③极限为无穷：g(N)=o(f(N))，

④极限摆动：二者无关。

# 2.模型：

计算模型是一台标准的计算机。①有标准的简单指令系统，②做任一件简单的工作都恰好花费一个时间单元，③有固定范围的整数，④不存在诸如矩阵求逆或排序等运算，⑤有无限内存。

# 3.要分析的问题：

主要分析运行时间，由使用的算法和该算法的输入数据量影响。

输入的大小是主要的考虑方面。①Tavg(N)：平均时间，②Tworst(N)：最坏情况下时间。Tavg(N)<= Tworst(N)。我们所需要分析的量是Tworst(N)。

## 最大子序列问题：

给定整数A1，A2，……，AN，求的最大值，如果所有整数均为负数，则最大子序列和为0。

该问题有很多算法，数据的读入是一个瓶颈，。

# 4.运行时间的运算：

## 4.1一个简单的例子——计算：

int sum(int n)

{

int sum = 0; //一个时间单元

for(int I = 1;I <= n;i++) //2\*n+2个时间单元，（一次初始化，n+1次判断合法性，n次自增运算）

sum += i\*i\*I; //4\*n个时间单元（2\*n次乘法，n次加法，n次赋值）

return sum; //一个时间单元

}

有的语句对结论无关紧要，所以花费时间去想是不必要的。

## 4.2一般法则：

分析的基本策略是从内部向外展开，计算大O的一般法则：

4.2.1FOR循环：

运行时间至多是该for循环内语句的运行时间乘以迭代的次数。

4.2.2嵌套的FOR循环：

从里向外分析这些循环，在一组嵌套循环内部的一条语句总的运行时间为该语句的运行时间乘以该组所有的for循环的大小和乘积。

4.2.3顺序语句：

将各个语句的运行时间求和即可，这意味着其中的最大值就是所得的运行时间。

4.2.4判断语句：

一个if/else语句的运行时间从不超过判断再加上S1和S2中运行时间长者的总的运行时间。

如果有函数调用，那么先分析函数，如果有递归，要看一下是不是被薄面纱遮住的for循环。

## 4.3斐波那契数列：

long Fib(int n)

{

if(n <= 1) return 1;

else return Fib(N - 1) + Fib(N - 2);

}

令T(N)为函数Fib(N)的运行时间。N>=2时，有如下关于Fib(N)的运行时间公式：T(N)=T(N-1)+T(N-2)+2。

这个程序缓慢是因为存在大量多余的工作要做，第一次调用Fib(N-1)实际上计算了Fib(N-2)，这个信息被抛弃，在第二次调用时又重新计算了一遍。抛弃的信息量递归地合成起来导致巨大的运行时间。

## 4.4最大子序列和问题的解：

四种算法：

4.4.1算法一：

int maxsubsequencesun(int a[],int n)

{

int maxsum = 0;

for(int i = 0;i < n;i++)

for(int j = i;j < n;j++)

{

int sum = 0;

for(int k = i;k < j;k++)

sum += a[k];

if(sum > maxsum) maxsum = sum;

}

return maxsum;

}

由三层循环，精确的分析是：。

4.4.2算法二：

通过撤除一个for循环来避免立方运行时间，不过这不总是可能的。

int maxsubsequencesum(int a[],int n)

{

int maxsum = 0,thissum;

for(int i = 0;I < n;i++)

{

thissum = 0;

for(int j = I;j < n;j++)

{

thissum += a[j];//用thissum存储

if(thissum > maxsum)

maxsum = thissum;

}

}

return maxsum;

}

4.4.3算法三：

该方法采用了一种分治策略，“分”是指把问题分成两个大致相等的字问题，然后递归地为他们求解，“治”是指将两个字问题的解合并到一起并可能再做些少量的附加工作，最后得到问题的解。

该问题中，最大子序列和肯呢个在三处出现，或者出现在数据左半部，或者出现在数据右半部，或者跨越输入数据的中部从而占据左右两半部分。第三种情况的最大和可以通过求出前半部分包括最后一个元素的最大和和后半部分包括第一个元素的最大和而得到。

#include<stdio.h>

#include<iostream>

#include<string.h>

using namespace std;

int maxsubsequencesum(const int \*num,int left,int right)

{

//基准情况

if(left == right)

if(num[left] > 0)//why?求解求解！！！！！

return num[left];

else return 0;

//计算两边

int middle = (right - left)/2 + left;

int maxleft = maxsubsequencesum(num,left,middle);//左半边，注意这里函数有返回值

int maxright = maxsubsequencesum(num,middle + 1,right);//右半边

//计算跨越

int maxleftboarder = 0,maxrightboarder = 0,maxthissum = 0;;

for(int i = middle; i >= left; i--)

{

maxthissum += num[i];

if(maxthissum > maxleftboarder)

maxleftboarder = maxthissum;

}

maxthissum = 0;

for(int i = middle + 1; i <= right; i++)

{

maxthissum += num[i];

if(maxthissum > maxrightboarder)

maxrightboarder = maxthissum;

}

return max(max(maxleft,maxright),maxleftboarder + maxrightboarder);//这里算是程序的出口，一定要搞明白返回值！

}

int a[1006];

int main()

{

int n;

cin>>n;

for(int i = 0; i < n; i++)

cin>>a[i];

cout<<maxsubsequencesum(a,0,n - 1);

return 0;

}

当N=1时，T(1)=1。当N!=1时，两个for循环均为O(N)，两个递归函数由于要求解大小为N/2的子序列问题（假设N为偶数），这两个递归函数各花费T(N/2)个时间单位，合起来就是：T(N)=2\*T(N/2)+O(N)。用N代替方程中的O(N)项，因为T(N)最终还是要用大O表示，因此这么做不影响答案。由两式的规律可得：T(N)=N\*(k+1)=NlogN+N=O(NlogN)。

要求N是偶数，因为否则N/2就不确定了。当N不是2的幂的时候，我们多少需要更加复杂一些的分析，但是大O的结果是不变的。

4.4.4算法四：

是算法二的一个改进，该算法的一个附带优点是，它只对数据进行一次扫描，一旦a[i]被读入并被处理，它就不再需要被记忆。在任意时刻算法都能对它已经读入的数据给出子序列问题的正确答案，具有这种特性的算法叫做联机算法，仅需要常量空间并以线性时间运行的联机算法几乎是完美的算法。

重点的一个思想是：如果a[i]是负数那么它不可能代表最小序列的起点，因为任何包含a[i]的作为起点的子序列都可以通过用a[i+1]作为起点来改进。类似的有，任何的负的子序列不可能是最优子序列的前缀。例如说，循环中我们检测到从a[i]到a[j]的子序列是负数，那么我们就可以推进i。关键的结论是我们不仅可以把i推进到i+1，而且我们实际可以把它一直推进到j+1。

举例来说，令p是i+1到j之间的任何一个下标，由于前面假设了a[i]+…+a[j]是负数，则开始于下标p的任意子序列都不会大于在下标i并且包含从a[i]到a[p-1]的子序列对应的子序列（j是使得从下标i开始成为负数的第一个下标）。因此，把i推进到j+1是安全的，不会错过最优解。注意的是：虽然，如果有以a[j]结尾的某序列和是负数就表明了这个序列中的任何一个数不可能是与a[j]后面的数形成的最大子序列的开头，但是并不表明a[j]前面的某个序列就不是最大序列，也就是说不能确定最大子序列在a[j]前还是a[j]后，即最大子序列位置不能求出。但是能确保maxSum的值是当前最大的子序列和。这个算法还有一个有点就是，它只对数据进行一次扫描，一旦a[j]被读入处理就不需要再记忆。它是一个联机算法。

long maxSubSum4(const vector<int>& a)

{

long maxSum = 0, thisSum = 0;

for (int j = 0; j < a.size(); j++)

{

thisSum += a[j];

if (thisSum > maxSum)

maxSum = thisSum;

else if (thisSum < 0)

thisSum = 0;

}

return maxSum;

}

## 4.4运行时间中的对数：

如果一个算法用常数时间O(1)将问题的大小削减为其一部分（通常是1/2），那么该算法就是O(logN)。另一方面，如果使用常数时间只是把问题减少一个常数，那么该算法就是O(N)的。

若输入N个数，则一个算法只是把这些数读入就必须耗费Ω(N)的时间，因此当我们谈到这类问题的O(logN)算法时，通常都是假设输入数据已经提前读入。

4.4.1对分查找：

给定一个整数X和整数A0,A1,……,AN-1,后者已经预先排序并在内存中，求使得Ai=X的下标，如果X不在数据中，则返回i=-1。

#include<stdio.h>

#include<iostream>

using namespace std;

int BinarySearch(int \*num,int n,int sear)

{

int left = 0,right = n - 1,middle;

while(left <= right)

{

middle = (right - left)/2 + left;//注意这里

if(num[middle] == sear) return middle;

if(num[middle] > sear) right = middle - 1;

if(num[middle] < sear) left = middle + 1;

}

}

int main()

{

int n,num[10010],sear;

cin>>n;

for(int i = 0;i < n;i++)

cin>>num[i];

cin>>sear;

cout<<1 + BinarySearch(num,n,sear);

return 0;

}

循环从High-Low=N-1开始并在High-Low>=-1结束，每次循环后High-Low的值至少将该次循环前的值折半，于是循环次数最多为┌log(N-1)┐+2。

4.4.2计算最大公因数的欧几里德算法：

因数：一整数被另一整数整除,后者即是前者的因数。最大公因数：若干个数它们公共的因数中最大的一个（GCD）。最小公倍数：若干个数它们公共的因数中最小的一个（LCM）。

欧几里德算法通过连续计算余数直到余数为0为止，最后的非零余数就是最大公因数。GCD(a,b)=GCD(b,a%b)。

#include<iostream>

using namespace std;

//要求a>b

int gcd1(int a,int b)//递归

{

if(!b) return a;

return gcd1(b,a%b);

}

int gcd2(int a,int b)//迭代

{

while(b)

{

int temp = a;

a = b;

b = temp % a;

}

return a;

}

int main()

{

int a,b;

cin>>a>>b;

cout<<gcd1(a,b)<<endl;

cout<<gcd2(a,b)<<endl;

return 0;

}

估计算法的整个运行时间依赖于确定玉树序列究竟有多长，事实上在一次迭代中余数并不按照一个常数因子递减，然而可以证明：在两次迭代后，余数最多是原始值的一半（相当于二分）。可以从如下定理推出：

定理：如果M>N，则M mod N <M/2。

证明：

a可以表示成a = kb + r，则r = a mod b，

　　假设d是a,b的一个公约数，则有：

　　d|a, d|b，而r = a - kb，因此d|r，

　　因此d是(b,a mod b)的公约数，

　　假设d 是(b,a mod b)的公约数，则：

　　d | b , d |r ，但是a = kb +r，

　　因此d也是(a,b)的公约数，

因此(a,b)和(b,a mod b)的公约数是一样的，其最大公约数也必然相等，得证。

4.4.3幂运算：

计算XN。运用递归，如果N是偶数，有XN=XN/2·X N/2;如果N是奇数，有XN=X(N-1)/2·X(N-1)/2·X。

#include<iostream>

using namespace std;

unsigned long pow(int x,int n)

{

if(!n) return 1; //注意这里！！！！！！！！

if(n == 1) return x;

if(n%2)//奇数

return pow(x,(n-1)/2)\*pow(x,(n-1)/2)\*x;

else //偶数

return pow(x,n/2)\*pow(x,n/2);

}

int main()

{

int x,n;

cin>>x>>n;

cout<<pow(x,n);

return 0;

}

原来计算XN需要进行N-1次运算，现在最多用2logN次，因为把问题分半最多需要两次乘法（如果N是奇数）。这里我们又写出一个递归公式并将其解出。简单的直觉避免了盲目的强行处理。

## 4.5检验你的分析：

4.5.1编程并比较实际观察到的运行时间与通过分析所描述的运行时间是否相匹配：

当N扩大一倍时，线性程序的运行时间乘以因子2，二次程序乘以4，三次程序乘8，对数程序时间多加一个常数，以O(NlogN)运行的程序花费比在相同环境下运行时间的两倍多一些时间。如果低阶项的系数相对地大，并且N又不是足够大，那么运行时间的变化量很难观察清楚。

4.5.2对N的某个范围（通常用2的倍数隔开）计算比值T(N)/f(N)：

其中T(N)是凭经验观察到的运行时间，如果f(N)是运行时间的理想近似，那么所算出的值收敛于一个正常数；如果f(N)估计过大，则算出的值收敛于0；如果f(N)估计过低而程序不是O(f(N))的，那么算出的值发散。

4.5.3计算两个随机选取出小于或等于N的互异正整数互素的概率：

#include<iostream>

using namespace std;

double percentage(int n)

{

tot = 0;

for(int i = 1;i <= n;i++)

for(int j = i + 1;j <= n;j++)

{

if(gcd(i,j)) rel++;//伪代码

tot++;

}

return rel\*1.0/tot

}

int main()

{

int n;cin>>n;

cout<<percentage(n);

return 0;

}

当n增大时，结果将趋向于6/∏。

## 4.6分析结果的正确性：