各位组员好，本年度的第一期算法学习于本周就正式开始了，希望大家积极参与每次的算法学习和讨论中！

算法学习会分为若干期进行，每期会指定一个主题，即某一类算法的学习，每期持续时间初步定为两个月，这段时间内会每两周学习一个算法，也就是每期会包含4个算法。

第一期算法学习是后续学习的开始，我打算首先从基础开始，而算法的基础就是数学，所以第一期学习算法的主题定位在“经典数学问题算法”，让大家切身感受到数学的威力，以及算法的魅力。

第一期的首个算法定为“浮点数开根号运算”算法，浮点数开根号对于我们来说再熟悉不过了，相信每一个Programmer一定用过无数次，但是就是这么一个我们最熟悉的问题实现起来其实并不是那么直接，如果考虑到算法实现的时间、空间复杂度可以说并不容易。所以第一期的第一个算法我们就从这个看似简单又不简单的问题出发，开启我们整个团队算法学习的新征程！

**第一期第一个算法学习时间**：2017年2月13日 – 2017年2月24日（共两周）

**这个期间对每个组员的要求**：提前对这个问题进行思考，查阅资料，尽可能多的提出解决这个问题更好更优的算法，同时不要仅仅只是给出自己方案的简单描述，建议可以将自己想到的算法写成完善的文档（包括算法的思路、公式，以及实例），以及自己将自己的算法编写成可以运行的程序。

**算法学习讨论的形式**：2017年2月24日也就是下周以前，我们会组织一次讨论会，会议上每个人都要求展示自己的算法学习文档，并运行算法实现的程序。同时在讨论会上各个组员之间也可以PK各自的算法，比较算法的优劣。

**备注**：以上为第一期第一个算法学习的暂定形式，后期可能根据运行效果进行调整。希望各位组员积极踊跃的提出对我们这个算法学习的建议，同时也祝愿我们这个活动能够办出成效，帮助每位组员，共同进步！

# 问题描述

我们通常会求一个数与它本身相乘的积，即求的平方，记为；类似的如果我们想知道一个正数可以表示为哪两个数的乘积，就需要对开根号，记为。开根号只是一个数学表示符号，真正想求得它的数值只能采用间接的方法（特殊正数的开根号是容易的，例如：4的开根号是2，9的开根号是3，等等），实际上它是一个求平方的逆过程，因此最直观的求解方式就是采用不同的数求平方，找出最相近的数作为开根号的近似值。在数值计算中通常采用逐步逼近的方式来求取，如何逼近就是我们自己需要设计的算法，逼近的效率和效果就决定了该算法的优劣。下面将分别介绍几种常用的开根号的数值求解算法，分析其原来，并比较它们的效率。

# 二分法

## 原理

已知一个**单调**函数，求所对应的？

求解之前需要确定所求的一个大致区间，算法比较简单，直接上代码：

|  |
| --- |
| float BinarySearch(float y, float x1, float x2)  {  float xmid = (x1+x2)/2;  float ymid = f(xmid);  for(int i=0; i<1000000 && fabs(y-ymid)>0.0000001; i++ )  {  if(ymid==y)  {  return ymid;  }  else if(ymid<y)  {  x1 = xmid;  }  else  {  x2 = xmid;  }  }    return (x1+x2)/2;  } |

## 实现

明显属于单调函数，我们求解，其值域也可以知道如下，可用于确定初始搜索区间，因此二分法完全可以用于开平方运算。

代码如下：

|  |
| --- |
| float SqrtByBisection(float n)  {  float low,up,mid,last;  low=0,up=(n<1?1:n);  mid=(low+up)/2;  do  {  if(mid\*mid>n)  up=mid;  else  low=mid;  last=mid;  mid=(up+low)/2;  }while(fabsf(mid-last) > eps);    return mid;  } |

# 牛顿法

## 原理

当在定义域范围内可导时，上式可用于快速迭代搜索一个近似的，使得。

【注意】

1. 为可导函数；

2. 需要确定一个初始的搜索值。

## 实现

当，我们可以转换成求。

此时牛顿公式变成

### 初值问题

不同于二分法给定一个初始区间，牛顿法则需要给定一个搜索的初始位置，然后在此基础上一步一步迭代，逼近最优解。通常不同的初始位置会产生不同的搜索速度，因此牛顿算法的关键之一就是确定一个初始搜索位置。

有没有一个最优的初始位置呢？That’s a question.

答案：有一个最容易接近最优解的值！就是一下这么一长串：

(((\*(int \*)&x)&0xff7fffff)>>1)+(64<<23); //其中x就是前面讨论的a

针对C语言中的32 bits的float类型，它的计算机表示法为：

|  |  |  |
| --- | --- | --- |
| s(1 bit) | e+127 (8 bits) | m (23 bits) |

【注】的值域变成。

开方过程『移位』：

（1）指数部分：除以2，向右移动1位；

（2）小数部分：除以2，向右移动1位；

（3）指数补位：指数部分移位的时候，127也参与了移位，即值变少了，应该补回来；

【注】指数部分最容易使得结果精确。

### 代码

|  |
| --- |
| float SqrtByNewton(float x)  {  int temp = (((\*(int \*)&x)&0xff7fffff)>>1)+(64<<23);  float val=\*(float\*)&temp;  float last;  do  {  last = val;  val =(val + x/val) / 2;  }while(fabsf(val-last) > eps);  return val;  } |

# 卡马克法

## 原理

卡马克算法其实核心公式也是牛顿公式，只是在构建函数时有所区别，但是产生的效果是惊人的。

令，那么牛顿公式变成

初始值为

0x5f3759df - ( i >> 1 ) // i为a的int表示

## 代码

|  |
| --- |
| float Q\_rsqrt( float number )  {  long i;  float x2, y;  const float threehalfs = 1.5F;    x2 = number \* 0.5F;  y = number;  i = \* ( long \* ) &y; // evil floating point bit level hacking  i = 0x5f3759df - ( i >> 1 ); // what the fuck?  y = \* ( float \* ) &i;  y = y \* ( threehalfs - ( x2 \* y \* y ) ); // 1st iteration  // y = y \* ( threehalfs - ( x2 \* y \* y ) ); // 2nd iteration, this can be removed    return y;  } |

# 汇编求开方函数

不在讨论之列

故略

# 算法分析

## 时间复杂度

定义：如果一个问题的规模是n，解这一问题的某一算法所需要的时间为T(n)，它是n的某一函数 T(n)称为这一算法的“时间复杂性”。

常用大O表示法表示时间复杂性，注意它是某一个算法的时间复杂性。大O表示只是说有上界，由定义如果f(n)=O(n)，那显然成立f(n)=O(n^2)，它给你一个上界，但并不是上确界，但人们在表示的时候一般都习惯表示前者。

时间复杂度是总运算次数表达式中受n的变化影响最大的那一项(不含系数)。

例子：

(1)

for(i=1;i<=n;i++) //循环了n\*n次，当然是O(n^2)

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

s++;

(2)

for(i=1;i<=n;i++) //循环了(n+n-1+n-2+...+1)≈(n^2)/2，因为时间复杂度是不考虑系数的，所以也是O(n^2)

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

s++;

(3)

for(i=1;i<=n;i++) //循环了(1+2+3+...+n)≈(n^2)/2,当然也是O(n^2)

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

s++;

(4)

i=1;k=0;

while(i<=n-1)

{

k+=10\*i;

i++;

}

//循环了n-1≈n次，所以是O(n)

常见的算法时间复杂度由小到大依次为：

Ο(1)＜Ο(log2n)＜Ο(n)＜Ο(nlog2n)＜Ο(n2)＜Ο(n3)＜…＜Ο(2^n)＜Ο(n!)

Ο(1)、Ο(log2n)、Ο(n)、Ο(nlog2n)、Ο(n2)和Ο(n3)称为多项式时间，而Ο(2^n)和Ο(n!)称为指数时间。计算机科学家普遍认为前者是有效算法，把这类问题称为P类问题，而把后者称为NP问题。

Ο(1)： 简单赋值、比较

Ο(log2n)： 树深度的遍历

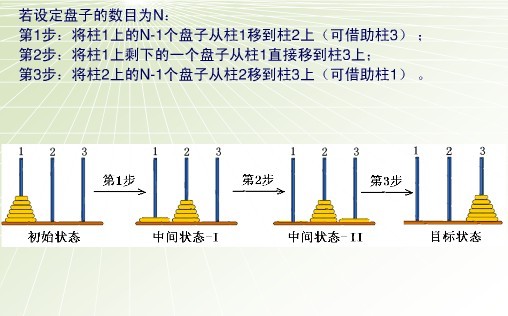
Ο(n)： 一层for循环

Ο(nlog2n)： 块排、堆排序

Ο(n2)，Ο(n3)，…：多层循环

Ο(2^n)： Hanoi Tower问题

Ο(n!)：



Hanoi Tower问题

## 空间复杂度

空间复杂度(Space Complexity)是对一个算法在运行过程中临时占用存储空间大小的量度。

一个算法在计算机存储器上所占用的存储空间，包括：

（1）存储算法本身所占用的存储空间

存储算法本身所占用的存储空间与算法书写的长短成正比，要压缩这方面的存储空间，就必须编写出较短的算法。

（2）算法的输入输出数据所占用的存储空间

算法的输入输出数据所占用的存储空间是由要解决的问题决定的，是通过参数表由调用函数传递而来的，它不随本算法的不同而改变。

（3）算法在运行过程中临时占用的存储空间（**关注的重点**）

算法在运行过程中临时占用的存储空间随算法的不同而异，有的算法只需要占用少量的临时工作单元，而且不随问题规模的大小而改变，是节省存储的算法；有的算法需要占用的临时工作单元数与解决问题的规模n有关，它随着n的增大而增大，当n较大时，将占用较多的存储单元。

## 算法比较

|  |  |
| --- | --- |
| **算法** | **时间（秒）** |
| 二分法 | 016.0860 |
| 牛顿法 | 002.3130 |
| 卡马顿法 | 000.7790 |
| 系统函数 | 000.5070 |

【注】40 million次循环，不开启编译优化

|  |  |  |
| --- | --- | --- |
| **算法** | **时间（秒）** | **有无循环** |
| 二分法 | 012.6390 | 有 |
| 牛顿法 | 001.3800 | 有 |
| 卡马顿法 | 000.0000 | 无 |
| 系统函数 | 000.0000 | 未知 |

【注】40 million次循环，开启编译优化

即使是4 billion次循环，卡马顿法和系统函数计算时间都是0。