**斐波那契查找算法详解**

**说明**

1. 斐波那契查找算法核心思想类似于二分查找和插值查找，区别在于对标志值，即 mid 的设计算法不一样，二分查找直接重用中间值作为标杆，插值查找使用自适应确定mid,而斐波那契查找算法则使用黄金分割，使得mid总是处于查找数列的黄金分割点位置
2. 因为斐波那契数列越到后边，相邻两数的比值越发接近0.618，也就是黄金分割比，因为可以巧妙的使用斐波那契数列寻找数组中的黄金分割点，即mid对应的下标
3. 因此需要先构建一个斐波那契数列，可以使用递归的方法或者非递归的方式
4. 使用斐波那契数列寻找数组的黄金分割点公式为： mid = low + f (k - 1) - 1,k为当前斐波那契数对应的索引
5. 使用斐波那契数列查找，需要先将当前数组的长度构建为第一个比数组长度大的斐波那契数，这个数对应的索引就是 k ,可以使用循环的方法
6. 将构建的新数组后边补零的位置替换为数组中的最后一个位置，即最大值
7. 准备工作准备好后，就可以计算当前数组的黄金分割值，然后获取到当前黄金分割值对应的元素
8. 将这个元素和要查找的元素进行比较，然后重置左右指针和重置后数组对应的黄金分割点
9. 当查找完所有的元素后，如果没有找到，则返回 - 1
10. 注意斐波那契数列的特性 即 当索引 > 2时，当前位置元素 = 前两个位置元素之和，而前两个位置元素之比刚好是满足黄金分割，正是基于这样的特性，才有公式 mid = low + f (k - 1) - 1
11. 斐波那契查找算法不易理解，须慢慢体会
12. 源码及详解见下

**源码及分析**

//斐波那契数列的最大长度

public static int maxSize = 20;

public static void main(String[] args) {

int[] arr = {1, 23, 45, 66, 67, 88, 90, 100};

int index = fisSearch(arr, 88);

System.out.println("index = " +index);

}

//构建斐波那契数列

public static int[] fis() {

int[] f = new int[maxSize];

f[0] = 1;

f[1] = 1;

for (int i = 2; i < f.length; i++) {

f[i] = f[i - 1] + f[i - 2];

}

return f;

}

/\*\*

\* 斐波那契查找算法实现

\*

\* @param arr 要查找的原始数组

\* @param key 要查找的值

\* @return 查找的结果

\*/

public static int fisSearch(int[] arr, int key) {

//数组左侧索引

int low = 0;

//数组右侧索引

int high = arr.length - 1;

//比右侧索引大的第一个斐波那契数对应的索引

int k = 0;

//黄金分割点

int mid = 0;

//斐波那契数列

int[] f = fis();

//由数组最大值计算k

while (high > f[k] - 1) {

k++;

}

//因为f[k]的值可能大于数组的长度，因此需要给原数组扩容到长度 == f(k)

int[] tmp = Arrays.copyOf(arr, f[k]);

//调用copyOf方法后在扩容部分全部补了0，实际上需要补数组的最后一位

for (int i = high + 1; i < tmp.length; i++) {

tmp[i] = arr[high];

}

//使用while循环来查找需要找的数

while (low <= high) {

//先计算黄金分割点

mid = low + f[k - 1] - 1;

//判断黄金分割点的元素和要查找的元素的关系

//如果要查找的值在mid左边，重置high和k

if (tmp[mid] > key){

high = mid - 1;

k--;

//如果要查找的值在mid右边

}else if (tmp[mid] < key){

low = mid + 1;

k -= 2;

//否则找到该元素

}else {

if (mid <= high){

return mid;

}else {

return high;

}

}

}

//如果循环结束后还没有找到，说明没有

return -1;

}

## 2. 斐波那契数列

斐波那契数列，又称黄金分割数列，指的是这样一个数列：1、1、2、3、5、8、13、21、····，在数学上，斐波那契被递归方法如下定义：F(1)=1，F(2)=1，F(n)=f(n-1)+F(n-2) （n>=2）。该数列越往后相邻的两个数的比值越趋向于黄金比例值（0.618）。

## 斐波那契数和数据有什么关系？

斐波那契数列中(上表来自维基百科)，**从第三项开始，每一项都等于前两项之和**：

上述性质可以用于数据区间分割，将一个长度为F(n)数组看做前后两半，前面一半长度是F(n-1)，后面一半的长度是F(n-2)。正是这个想法将斐波那契数列和数组联系到一起，也是后面分析的基础。

斐波那契查找同样是查找算法家族中的一员，它要求数据是有序的（**升序**或**降序**）。斐波那契查找采用和二分查找/插值查找相似的区间分割策略，都是通过**不断的分割区间缩小搜索的范围**。

其中n的取值是任意长度的，即对任意长度的数组都能找到对应的斐波那契数，下面将具体介绍斐波那契查找的步骤。

## 原理分析

斐波那契查找的整个过程可以分为：

* 构建斐波那契数列；
* 计算数组长度对应的斐波那契数列元素个数；
* 对数组进行填充；
* 循环进行区间分割，查找中间值；
* 判断中间值和目标值的关系，确定更新策略；

1. 构建斐波那契数列如下：

[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377]

1. 计算数组长度对应的斐波那契数列元素个数  
   假设手中的数据如下：

[1, 2, 4, 6, 7, 9, 13,

16, 17, 21, 23, 25, 27,

31, 45, 56, 58, 61, 65,

67, 73, 75, 88, 93, 102]

可知上述数据共25个元素，不对应1.1节中的斐波那契数列中任何F(n)，这种情况是很常见的。此时，策略是采用“**大于数组长度的最近一个斐波那契数值**”。比如当前数组长度为25，斐波那契数列中大于25的最近元素为34。

1. 对数组进行填充  
   确定了斐波那契数值后，就要进行数值填充，**即将数组从25个元素填充到34个**。填充时，将第26到34个元素均采用第25个元素值进行填充，即最大值填充。
2. 循环进行区间分割，查找中间值  
   这一个步骤与前面介绍的二分查找和插值查找相似，都是不断的缩小搜索区间，进而确定目标值的位置。区间分割公式如“要点”所述，每次分割中间位置的计算如下：

image.png

此时**数组被分割为左右两个区间，左边区间含有F(n-1)个元素，-1是因为下标从0开始**（比如F(1)表示两个元素）。

1. 判断中间值和目标值的关系，确定更新策略  
   中间值和目标值有三种大小关系，分别对应三种处理方式：

* 相等，则查找成功，返回中间位置即可；
* 中间值小于目标值，则说明目标值位于中间值到右边界之间（即右区间），右区间含有F(n-2)个元素，所以n应该更新为n=n-2；
* 中间值大于目标值，这说明目标值位于左边界和中间值之间（即左区间），左区间含有F(n-1)个元素，所以n应更新为n=n-1；

**可能有人要问mid = low + F[k-1] - 1怎么来的？**  
接下来我们图解一下：

image.png

举例：  
对于斐波那契数列：1、1、2、3、5、8、13、21、34、55、89……（也可以从0开始），前后两个数字的比值随着数列的增加，越来越接近黄金比值0.618。比如这里的89，把它想象成整个有序表的元素个数，而89是由前面的两个斐波那契数34和55相加之后的和，也就是说把元素个数为89的有序表分成由前55个数据元素组成的前半段和由后34个数据元素组成的后半段，那么前半段元素个数和整个有序表长度的比值就接近黄金比值0.618，假如要查找的元素在前半段，那么继续按照斐波那契数列来看，55 = 34 + 21，所以继续把前半段分成前34个数据元素的前半段和后21个元素的后半段，继续查找，如此反复，直到查找成功或失败，这样就把斐波那契数列应用到查找算法中了。

## 查找算法

对于一个数组来说，如果数组长度为斐波那契数列中的某一个数字，那么我们就可以用黄金分割比例来分割该数组。当然，如果数组长度没有达到要求，那么我们可以尝试它扩大来满足要求，所以这就是算法的要求。

其实，该算法的本质也还是二分法，只不过跟插入排序法一样，也是将目标的mid值改变成其它的，以至于分割的结果不一样，查找的效果也不一样。

那么具体是怎样分割的呢？

这里用图片直观理解一下：

也就是说，真正需要我们做的是找到这个**mid**，这里给出公式：**mid = F(k-1)-1**，你也可以从图片中看出来，数组下标是从：**0~F[k]-1**，将原来的分成两半，再比较来查找。  
斐波那契查找原理与前两种相似，仅仅 改变了中间结点（mid）的位置，mid不 再是中间或插值得到，而是位于**黄金分割点**附近，即**mid=low+F(k-1)-1** （F代表斐波那契数列）

对F(k-1)-1的理解：

1. 由斐波那契数列 **F[k]=F[k-1]+F[k-2]** 的性质，可以得到 **（F[k]-1）=（F[k-1]-1）+（F[k-2]-1）+1** 。该式说明：**只要顺序表的长度为F[k]-1，则可以将该表分成长度为F[k-1]-1和F[k-2]-1的两段**，即如上图所示。从而中间位置为**mid=low+F(k-1)-1**
2. 类似的，每一子段也可以用相同的方式分割
3. **但顺序表长度n不一定刚好等于F[k]-1，所以需要将原来的顺序表长度n增加至F[k]-1**。这里的k值只要能使得F[k]-1恰好大于或等于n即可，由以下代码得到,顺序表长度增加后，新增的位置（从n+1到F[k]-1位置），都赋为n位置的值即可。

javascript代码：

/\*\*

\*

斐波那契（黄金分割）查找算法

斐波那契(黄金分割法)原理:

斐波那契查找原理与前两种相似，仅仅 改变了中间结点（mid）的位置，mid不 再是中间或插值得到，而是位于黄金分 割点附近，即mid=low+F(k-1)-1 （F代表斐波那契数列），如下图所示

对F(k-1)-1的理解：

1.由斐波那契数列 F[k]=F[k-1]+F[k-2] 的性质，可以得到 （F[k]-1）=（F[k-1]-1）+（F[k-2]-1）+1 。该式说明：只要顺序表的长度为F[k]-1，则可以将该表分成长度为F[k-1]-1和F[k-2]-1的两段，即如上图所示。从而中间位置为mid=low+F(k-1)-1

2.类似的，每一子段也可以用相同的方式分割

3.但顺序表长度n不一定刚好等于F[k]-1，所以需要将原来的顺序表长度n增加至F[k]-1。这里的k值只要能使得F[k]-1恰好大于或等于n即可，由以下代码得到,顺序表长度增加后，新增的位置（从n+1到F[k]-1位置），都赋为n位置的值即可。

\*

\*/

let arr = [1, 8, 10, 89, 1000, 1234, 1555, 1777, 1888, 1999];

console.log('查找1：', fibSearch(arr, 1));

console.log('查找1999：', fibSearch(arr, 1999));

console.log('查找2999：', fibSearch(arr, 2999));

//因为后面我们mid = mid=low+F(k-1)-1,需要使用到斐波那契数列，

//因此我们需要先获取到一个斐波那契数列

function fib(maxSize) {

let f = new Array(maxSize);

f[0] = 1;

f[1] = 1;

for (let i = 2; i < maxSize; i++) {

f[i] = f[i - 1] + f[i - 2];

}

return f;

}

//编写斐波那契查找算法

/\*\*

\*

\* @param {数组} arr

\* @param {我们需要查找的关键码} key

\* @return 返回对应的下标，没找到返回-1

\*/

function fibSearch(arr, key) {

let low = 0;

let high = length = arr.length - 1;

let k = 0;//表示斐波那契分割值得下标

let mid = 0;//存放mid值

let f = fib(20);

//获取斐波那契分割值得下标

while (high > f[k] - 1) {

k++;

};

//因为f[k]值，可能大于a的长度，因此我们需要构建一个新的数组

//用arr数组的最后的数组填充arr数组

arr = [...arr];

for (let i = high + 1; i < f[k]; i++) {

arr.push(arr[high]);

}

//使用while来循环处理，找到我们的数key

while (low <= high) {//只要这个条件满足，就可以找

mid = low + f[k - 1] - 1;

if (key < arr[mid]) {

//我们应该继续向数组的前面查找（左边）

high = mid - 1;

//1.全部元素 = 前面的元素 + 后面的元素

//2.f[k] = f[k-1] + f[k-2]

//因为前面 f[k-1]个元素，所以可以继续拆分f[k-1] = f[k-2]+f[k-3]

//即下次循环mid = f[k-1-1] -1

k--;

} else if (key > arr[mid]) {

//我们应该继续向数组的后面查找(右边)

low = mid + 1;

//1.全部元素 = 前面的元素 + 后面的元素

//2.f[k] = f[k-1] + f[k-2]

//3，因为后面我们有f[k-2]，所以可以继续拆分 f[k-2] = f[k-3]+f[k-4]

//4.即在f[k-2]的前面进行查找 k-=2

//5.即下次循环mid = f[k-1-2] -1

k -= 2;

} else {

//找到

//需要确定，返回的是哪个下标,因为可能找到扩充的元素的下标

if (mid <= length) {

return mid;

} else {

return length;

}

}

}

return -1

}

作者：老鼠AI大米\_Java全栈  
链接：https://www.jianshu.com/p/ca639e01c5a0  
来源：简书  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

## 1.查找算法概述

查找是在大量的信息中寻找一个特定的信息元素，在计算机应用中，查找是常用的基本运算，例如编译程序中符号表的查找。本文简单概括性的介绍了常见的七种查找算法，说是七种，其实二分查找、插值查找以及斐波那契查找都可以归为一类——插值查找。插值查找和斐波那契查找是在二分查找的基础上的优化查找算法。树表查找和哈希查找会在后续的博文中进行详细介绍。

### （1）查找定义

根据给定的某个值，在查找表中确定一个其关键字等于给定值的数据元素（或记录）。

### （2）查找算法分类

#### 1）静态查找和动态查找

注：静态或者动态都是针对查找表而言的。动态表指查找表中有删除和插入操作的表。

#### 2）无序查找和有序查找

无序查找：被查找数列有序无序均可；

有序查找：被查找数列必须为有序数列。

### （3）查找算法分析

**平均查找长度（Average Search Length，ASL）**：需和指定key进行比较的关键字的个数的期望值，称为查找算法在查找成功时的平均查找长度。

对于含有n个数据元素的查找表，查找成功的平均查找长度为：ASL = Pi\*Ci的和。

Pi：查找表中第i个数据元素的概率。

Ci：找到第i个数据元素时已经比较过的次数。

## 2.斐波那契查找

### （1）斐波那契查找说明

在介绍斐波那契查找算法之前，我们先介绍一下很它紧密相连并且大家都熟知的一个概念——黄金分割。

黄金比例又称黄金分割，是指事物各部分间一定的数学比例关系，即将整体一分为二，较大部分与较小部分之比等于整体与较大部分之比，其比值约为1:0.618或1.618:1。

0.618被公认为最具有审美意义的比例数字，这个数值的作用不仅仅体现在诸如绘画、雕塑、音乐、建筑等艺术领域，而且在管理、工程设计等方面也有着不可忽视的作用。因此被称为黄金分割。

大家记不记得斐波那契数列：1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89…….（从第三个数开始，后边每一个数都是前两个数的和）。然后我们会发现，随着斐波那契数列的递增，前后两个数的比值会越来越接近0.618，利用这个特性，我们就可以将黄金比例运用到查找技术中。

### （2）基本思想

也是二分查找的一种提升算法，通过运用黄金比例的概念在数列中选择查找点进行查找，提高查找效率。同样地，斐波那契查找也属于一种有序查找算法。

斐波那契查找与折半查找很相似，他是根据斐波那契序列的特点对有序表进行分割的。他要求开始表中记录的个数为某个斐波那契数小1，及n=F(k)-1;

斐波那契查找的核心是：

1）当key=a[mid]时，查找成功；

2）当key<a[mid]时，新的查找范围是第low个到第mid-1个，此时范围个数为F[k-1] - 1个，即数组左边的长度，所以要在[low, F[k - 1] - 1]范围内查找；

3）当key>a[mid]时，新的查找范围是第mid+1个到第high个，此时范围个数为F[k-2] - 1个，即数组右边的长度，所以要在[F[k - 2] - 1]范围内查找。

### （3）复杂度分析

最坏情况下，时间复杂度为O(log2n)，且其期望复杂度也为O(log2n)。

网上都没有讲清楚，导致外行很难直接直观的理解斐波那契查找。

我认为，斐波那契的核心在于两点：

1）斐波那契是一种特殊的分割方法，和二分、插值本质上是一样的，都是分割方法；

2）利用了斐波那契数列特殊的性质，一个长度只要可以被黄金分割，那么分割后的片段仍然可以继续进行黄金分割，循环。

首先，我们构造一个完整的斐波那契数列，然后开始分，小于就取左边的分割，F变为F-1；大于就取右边的分割，F变为F-2。

很好理解，F-1 和 F-2，自己画个图就理解了，这是菲波那切数列特殊的性质。

### （4）斐波那契查找算法C语言源代码



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70 | /\*\*  \* 算法君：一个专业的算法学习分享网站  \* 算法君：专业分享--数据剖析--算法精解  \* 算法君：http://www.suanfajun.com  \*/      #include <stdio.h>  #include <stdlib.h>  #define MAXN 20    //产生斐波那契数列  void Fibonacci(int \*f)  {      int i;      f[0] = 1;      f[1] = 1;      for (i = 2; i < MAXN; ++i)          f[i] = f[i - 2] + f[i - 1];  }    // 查找--有序查找  int Fibonacci\_Search(int \*a, int key, int n)  {      int i, low = 0, high = n - 1;      int mid = 0;      int k = 0;      int F[MAXN];      Fibonacci(F);      while (n > F[k] - 1)          //计算出n在斐波那契中的数列          ++k;      for (i = n; i < F[k] - 1; ++i) //把数组补全          a[i] = a[high];      while (low <= high)      {          mid = low + F[k - 1] - 1;  //根据斐波那契数列进行黄金分割          if (a[mid] > key)          {              high = mid - 1;              k = k - 1;          }          else if (a[mid] < key)          {              low = mid + 1;              k = k - 2;          }          else          {              if (mid <= high) //如果为真则找到相应的位置                  return mid;              else                  return -1;          }      }      return 0;  }    int main()  {      int a[MAXN] = { 5, 15, 19, 20, 25, 31, 38, 41, 45, 49, 52, 55, 57 };      int k, res = 0;      printf("请输入要查找的数字:\n");      scanf("%d", &k);      res = Fibonacci\_Search(a, k, 13);      if (res != -1)          printf("在数组的第%d个位置找到元素:%d\n", res + 1, k);      else          printf("未在数组中找到元素:%d\n", k);      return 0;  } |

### （5）斐波那契查找算法C++源代码



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77 | /\*\*  \* 算法君：一个专业的算法学习分享网站  \* 算法君：专业分享--数据剖析--算法精解  \* 算法君：http://www.suanfajun.com  \*/    /\*斐波那契查找法，前提是线性表必须有序,时间复杂度是O(logn)\*/  #include<iostream>    const int MAXSIZE = 20;    /\*定义斐波那契查找法\*/  int Fibonacci\_Search(int \*a, int n, int key)  {      int low, high, mid, i, k;      int F[MAXSIZE];        Fibonacci(F); //构造一个斐波那契数组F      low = 1;   //最低下标记录为首位      high = n;  //最高下标记录为末位      k = 0;        while(n > F[k]-1)  //计算n位于斐波那契数列的位置      {          k++;      }        for(i=n; i<F[k]-1; i++)  //将a的元素扩展到(某斐波那契数 - 1)，即F[k]-1      {          a[i] = a[n];      }        while(low <= high)      {          mid = low + F[k-1] - 1;   //计算当前分割的下标          if(key < a[mid])          {              high = mid - 1;              k -= 1;          }          else if(key > a[mid])          {              low = mid + 1;              k -= 2;          }          else          {              if(mid <= n)                  return mid;   //若相当则说明mid即为查找到的位置              else                  return n;     //若mid>n则说明是扩展的数值，返回n          }      }      return -1;  }      /\*用非递归法构造一个斐波那契数组\*/  void Fibonacci(int \*f)  {      f[0] = 1;      f[1] = 1;        for(int i=2; i<MAXSIZE; i++)      {          f[i] = f[i-1] + f[i-2];      }  }    int main()  {      int array[] = {0,16,24,35,47,59,62,73,88,99};        int number = Fibonacci\_Search(array, sizeof(array)/sizeof(int), 73);      std::cout<<"位置是：array["<<number<<"]\n";      return 0;  } |