1. 对比分析折半查找与斐波那契查找的性能与应用场景。

斐波拉契查找

算法思路

1. 首先，将待查找的数组按照升序排列。
2. 确定一个斐波那契数列，使得其最后一个数不小于待查找数组的长度。
3. 初始化两个指针：left为数组起始位置，right为数组结束位置。
4. 找到F[k]-1中的临界k，使n>F[k]，若n=F[k]则mid = low + F[k]-1 否则将待查数组扩充到F[k]个
5. 比较当前位置的值与待查找的值：
   - 如果当前位置的值等于待查找的值，则返回该位置。
   - 如果当前位置的值小于待查找的值，则将left指针移动到当前位置的下一个位置，同时将right指针向右移动k个位置。
   - 如果当前位置的值大于待查找的值，则将left指针向左移动k个位置，同时将right指针向左移动k+1个位置。
6. 重复步骤5，直到找到待查找的值或者left指针大于等于right指针为止。

时间复杂度为O(log n)

代码

```
#include <memory>
#include  <iostream>
using namespace std;

const int max_size = 20;//斐波那契数组的长度

/*构造一个斐波那契数组*/
void Fibonacci(int* F)
{
	F[0] = 0;
	F[1] = 1;
	for (int i = 2; i < max_size; ++i)
		F[i] = F[i - 1] + F[i - 2];
}

/*定义斐波那契查找法*/
int Fibonacci_Search(int* a, int n, int key)  //a为要查找的数组,n为要查找的数组长度,key为要查找的关键
{
	int low = 0;
	int high = n - 1;

	int F[max_size];
	Fibonacci(F);//构造一个斐波那契数组F

	int k = 0;
	while (F[k]<n)//计算n位于斐波那契数列的位置
		++k;

	int* temp;//将数组a扩展到F[k]-1的长度
	temp = new int[F[k]];
	memcpy(temp, a, n * sizeof(int));

	for (int i = n; i < F[k]; ++i)
		temp[i] = a[n - 1];

	while (low <= high)
	{
		int mid = low + F[k - 1] - 1;
		if (key < temp[mid])
		{
			high = mid - 1;
			k -= 1;
		}
		else if (key > temp[mid])
		{
			low = mid + 1;
			k -= 2;
		}
		else
		{
```

```
                if (mid < n)
                        return mid; //若相等则说明mid即为查找到的位置
                else
                        return n - 1; //若mid>=n则说明是扩展的数值,返回n-1
            }
        }
        delete[] temp;
        return -1;
}

int main()
{
        int a[] = { 0,16,24,35,47,59,62,73 };
        int key = 24;
        int index = Fibonacci_Search(a, sizeof(a) / sizeof(int), key);
        cout << key << " is located at:" << index;
        return 0;
}
```

二分查找

算法思路

二分查找：基本思想是将待查找的元素与序列的中间元素进行比较，如果待查找元素比中间元素小，则在序列的左半边继续查找，否则在序列的右半边查找，直到找到目标元素或者序列为空为止。时间复杂度为O(log n)，即查找次数与序列的长度n成对数关系。

```cpp
#include <iostream>
#include <vector>
#include <chrono>
using namespace std;
// 二分查找函数
int binarySearch(vector<int>& nums, int target) {
int left = 0, right = nums.size() - 1;
while (left <= right) {
int mid = (left + right) / 2;
if (nums[mid] == target) {
return mid; // 找到目标元素，返回其下标
}
else if (nums[mid] < target) {
left = mid + 1; // 目标元素在右半边
}
else {
right = mid - 1; // 目标元素在左半边
}
}
return -1; // 没有找到目标元素，返回-1
}
int linearSearch(vector<int>& nums, int target) {
for (int i = 0; i < nums.size(); i++) {
if (nums[i] == target) {
return i; // 找到目标元素，返回索引
}
}
return -1; // 未找到目标元素，返回-1
}
int main() {
vector<int> nums(100000);
for (int i = 0; i < 100000; i++) {
nums[i] = i + 1;
}
int target = 50001;
auto start1 = chrono::high_resolution_clock::now();
int index = binarySearch(nums, target);
if (index != -1) {
cout << "找到目标元素 " << target << ", 下标为 " << index << endl;
}
else {
cout << "未找到目标元素 " << target << endl;
}
auto end1 = chrono::high_resolution_clock::now();
auto duration1 = chrono::duration_cast<chrono::microseconds>(end1 - start1); // 计算耗时
cout << "二分法耗时 " << duration1.count() << " 微秒" << endl; // 输出耗时
auto start2 = chrono::high_resolution_clock::now();
index = linearSearch (nums, target);
if (index != -1) {
cout << "找到目标元素 " << target << ", 下标为 " << index << endl;
```

```
}
else {
cout << "未找到目标元素 " << target << endl;
}
auto end2 = chrono::high_resolution_clock::now();
auto duration2 = chrono::duration_cast<chrono::microseconds>(end2 - start2); // 计算耗时
cout << "线性法耗时 " << duration2.count() << " 微秒" << endl; // 输出耗时
return 0;
}
```

性能对比

斐波那契查找相对于二分查找的优点是：

1. 在查找长度较大时，斐波那契查找的效率更高，因为它能够利用斐波那契数列的特点，更快地逼近查找目标，减少比较次数，从而提高查找效率。
2. 斐波那契查找的查找效率更高，因为斐波那契查找中直用到了加减，而二分查找则用了除法。查找资料显示，斐波那契查找算法大约较二分查找算法快17%

斐波那契查找相对于二分查找的缺点是：

1. 斐波那契查找需要预处理出斐波那契数列，这样需要一定的额外空间。
2. 当要查找的数组长度不是斐波那契数时，还需要将数组扩展到最接近的斐波那契数，并将扩展部分填充为原数组的最后一个元素，这也会浪费一些空间
3. key在数组的末尾时，斐波那契查找的效率不如二分查找

应用场景

斐波那契查找和二分查找都需要序列有序。

斐波那契查找适用于大数据查找。由于斐波那契数列在数据量较大时数据增加很快，所以在处理大数据时斐波那契查找可以快速缩小待查序列，这样使查找效率高于二分查找。在数据量较小时，因为斐波拉契数列分布较为密集，反而更不利于缩小查找范围。

二分查找的算法不仅仅可以用于一位数组的查找，还可以用于二维数组的查找。在之前的思考题中做过此类题目。而斐波那契查找不能实现多维数组的查找。

2.调研学习Bloom过滤器的原理。

背景

现代计算机用二进制（bit，位）作为信息的基础单位，1 个字节等于 8 位。许多开发语言都提供了操作位的功能，合理地使用位能够有效地提高内存使用率和开发效率。

Bit-map 的基本思想就是用一个 bit 位来标记某个元素对应的 value，而 key 即是该元素。由于采用了 bit 为单位来存储数据，因此在存储空间方面，可以大大节省。

举个例子：假设网站有 1 亿用户，每天独立访问的用户有 5 千万，如果每天用集合类型和 BitMap 分别存储活跃用户：

集合类型：假如用户 id 是 int 型，4 字节，32 位，则集合类型占据的空间为 50 000 000 * 4/1024/1024 = 200M；

BitMap：.如果按位存储，5 千万个数就是 5 千万位，占据的空间为 50 000 000/8/1024/1024 = 6M。

运用场景

1、目前有 10 亿数量的自然数，乱序排列，需要对其排序。限制条件在 32 位机器上面完成，内存限制为 2G。如何完成?

2、如何快速在亿级黑名单中快速定位 URL 地址是否在黑名单中？(每条 URL 平均 64 字节)

3、需要进行用户登陆行为分析，来确定用户的活跃情况?

4、网络爬虫-如何判断 URL 是否被爬过?

5、快速定位用户属性（黑名单、白名单等）？

6、数据存储在磁盘中，如何避免大量的无效 IO?

7、判断一个元素在亿级数据中是否存在?

8、缓存穿透。

实现原理

假设我们有个元素。利用**k个哈希散列**函数，将元素映射到一个长度为 a 位的数组 B中的不同位置上，这些位置上的二进制数均设置为 1。

如果待检查的元素，经过这 k个哈希散列函数的映射后，发现其 k 个位置上的二进制数全部为 1，这个元素很可能属于集合A，反之，一定不属于集合A。

演示一下：

分别用7个hash函数对e取hash值，假设结果如下：

r1 = h1(e) = 5,

r2 = h2(e) = 6,

r3 = h3(e) = 1,

r4 = h4(e) = 7,

r5 = h5(e) = 6,

r6 = h6(e) = 10,

r7 = h7(e) = 9

根据上面的结果，把bits对应位置为1(重复只需置一次就可以了)：

bits[5] = 1

bits[6] = 1,

bits[1] = 1,

bits[7] = 1,

bits[10] = 1,

bits[9] = 1.

现在假设又来一个元素e2，要添加到集合中来。假设用7个hash函数做hash得：

r1 = h1(e2) = 3,

r2 = h2(e2) = 1,

r3 = h3(e2) = 6,

r4 = h4(e2) = 14,

r5 = h5(e2) = 11,

r6 = h6(e2) = 13,

r7 = h7(e2) = 3.

再把对应bits的位置为1，

bits[3] = 1,

bits[1] = 1,

bits[6] = 1,

bits[14] = 1,

bits[11] = 1,

bits[13] = 1,

bits[3] = 1.

现在要判断元素e3是否在这个集合内。

首先分别用上面的7个hash函数对e3求hash，结果假设如下：

r1 = h1(e3) = 3,

r2 = h2(e3) = 1,

r3 = h3(e3) = 6,

r4 = h4(e3) = 14,

r5 = h5(e3) = 11,

r6 = h6(e3) = 13,

r7 = h7(e3) = 3.

如果e3在集合内，那么上面结果对应的bits位要全都为1才行，如果有一个为0，那么e3就不在集合内。很明显，bits对应的位都为1，所以我们可以说e3很有可能在集合内，但不是百分之百。（这个应该很好理解）

所以Bloom过滤器可能会造成误判现象。