

对于一个序列可以采用二分查找或顺序查找，请结合实施查找的次数，确定在什么情况下使用二分查找效率更高？

问题分析：

对于这个问题我们首先要明确二分查找和顺序查找的算法。

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。

顺序查找：基本思想是从序列的第一个元素开始，依次与待查找元素进行比较，直到找到目标元素或者遍历完整个序列为止。时间复杂度为 $O(n)$ ，即查找次数与序列的长度 n 成线性关系。

由上述算法的描述可知，二分查找只能适用于有序序列，如果序列不有序，则无法将序列对半分开，即分开是没有意义的，在这样的情况下，非常容易得到错误结果或是运行时间变长。而顺序查找则没有这种限制，即序列不管有序还是无序，都能够实现。

因此，如果给定序列是有序的，二分查找的时间复杂度为 $O(\log n)$ 小于顺序查找的时间复杂度 $O(n)$ ，肯定选用二分查找更合适。如果给定序列是无序的，用二分查找可能无法得到正确的答案，所以尽管二分查找的时间复杂度 $O(\log n)$ 小于 $O(n)$ ，选择顺序查找的方法依旧更好。

代码举例

```

#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;
}

```

在这段代码中，我定义了一个100000个数字的有序序列，并在其中寻找数值为50001的下标。有运行结果如下：

Microsoft Visual Studio 调试控制台

```

找到目标元素 50001，下标为 50000
二分法耗时 826 微秒
找到目标元素 50001，下标为 50000
线性法耗时 2048 微秒

```

可以看到顺序查找的时间要大于二分查找，符合我们的预期。

然而这仅仅是在数据量较大的时候才可以得到，当数据量较小时：一个具有十个数字的有序序列，在其中寻找数值为6的下标，运行结果为：

选择 Microsoft Visual Studio 调试控制台

```

找到目标元素 6，下标为 5
二分法耗时 903 微秒
找到目标元素 6，下标为 5
线性法耗时 181 微秒

```

反而二分法的时间要大于线性法。

那如果序列为无序序列呢？

```

#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 = {10,6,8,7,2,4,9,1};
    int target = 1;
    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;
}

```

这里创建了一个无序列表{10,6,8,7,2,4,9,1}, 查找数字1返回下标

运行结果为：

 Microsoft Visual Studio 调试控制台

```

未找到目标元素 1
二分法耗时 625 微秒
找到目标元素 1, 下标为 7
线性法耗时 161 微秒

```

可以看到，二分法并没有找到元素，且时间长于线性法

因此这种情况下采用顺序查找更好。

总结

综上所述，在有序列表且列表元素个数较大时，采用二分法所消耗的时间更少，效率更高。在有序列表且列表元素较小和无序列表中，采用顺序查找的方法消耗时间更少，效率更高。

并且需要意识到，理论分析和实践往往有一定出入，所以一定要实践才可以得到更具体的答案。

现有一个序列，它是由一个有序系列绕着某个元素旋转得到的。请给出一个在此序列实施查找的有效算法，并对你的算法进行分析。

问题分析

首先我们要理解什么是旋转数组。例如，有序数组[1, 2, 3, 4, 5, 6, 7]，可以旋转处理成[4, 5, 6, 7, 1, 2, 3]等。在此题中，给定一个可能旋转过的有序数组arr，再给定一个数num，返回arr中是否含有num。

算法1

1. 遍历数组，如果发现当前元素大于下一个元素，则说明旋转点位于当前位置和下一个位置之间。
2. 如果目标元素在数组的第一个元素和旋转点之间，则在该范围内顺序查找目标元素。
3. 如果目标元素在旋转点和数组最后一个元素之间，则在该范围内顺序查找目标元素。
4. 如果目标元素不在数组的第一个元素和旋转点之间，并且不在旋转点和数组最后一个元素之间，则目标元素不存在于数组中。

具体来说，如果目标元素在数组中存在，那么该算法会返回目标元素的下标；如果目标元素不存在于数组中，则返回-1。

该算法的时间复杂度为 $O(n)$ ，其中 n 为数组的长度。由于需要遍历整个数组才能确定旋转点的位置，因此该算法的效率较低，适用于数组长度较小的情况。

代码实现

```

#include <iostream>
#include <vector>
#include <chrono>
using namespace std;

int find(vector<int>& nums, int target) {
    for (int i = 0; i < nums.size(); i++) {
        if (nums[i] > nums[i + 1]) {
            if ((target <= nums[i]) && (target >= nums[0])) {
                for (int j = 0; j <= i; j++) {
                    if (nums[j] == target)
                        return j;
                }
                return -1;
            }
            else {
                for (int j = i; j < nums.size(); j++) {
                    if (nums[j] == target)
                        return j;
                }
                return -1;
            }
        }
    }
}

int main() {
    vector<int> nums = {4,5,6,1,2,3};
    int target = 2;
    int index = find(nums, target);
    if (index != -1)
        cout << "找到元素" << target << endl << "下标是" << index << endl;
    else
        cout << "没有找到元素" << target << endl;
    return 0;
}

```

算法2

1. 定义左右指针，初始化为数组两端。
2. 使用二分查找的方式找到旋转点，即数组中的最小值所在位置。每次将数组的中间元素与右端元素进行比较，如果中间元素大于右端元素，则旋转点一定在右半部分，否则旋转点在左半部分。最终，当左右指针重合时，此时的指针位置即为旋转点。
3. 重新定义左右指针，初始化为数组两端。
4. 在旋转有序数组中进行二分查找，每次将数组的中间元素与目标元素进行比较，如果中间元素等于目标元素，则返回其下标；如果中间元素小于目标元素，则目标元素一定在右半部分，将左指针右

移；否则目标元素在左半部分，将右指针左移。直到找到目标元素或左右指针相遇时，返回-1表示未找到目标元素。

5. 返回查找结果。

该算法的时间复杂度为 $O(\log n)$ ，其中 n 为数组的长度。

算法实现


```

#include <iostream>
#include <vector>
#include <chrono>
using namespace std;

int search(vector<int>& nums, int target) {
    int n = nums.size(); // 数组长度
    int left = 0, right = n - 1; // 定义左右指针，初始化为数组两端
    while (left < right) { // 二分查找旋转点，直到left和right重合
        int mid = (left + right) / 2; // 中间位置
        if (nums[mid] > nums[right]) // 如果中间值大于右端值，则旋转点在右半部分
            left = mid + 1; // left右移
        else // 否则旋转点在左半部分
            right = mid; // right左移
    }
    int rot = left; // 记录旋转点位置
    left = 0, right = n - 1; // 重新定义左右指针
    while (left <= right) { // 在旋转有序数组中二分查找目标元素
        int mid = (left + right) / 2; // 中间位置
        int realmid = (mid + rot) % n; // 真实中间位置
        if (nums[realmid] == target) // 如果中间值等于目标值，则找到目标元素
            return realmid;
        if (nums[realmid] < target) // 如果中间值小于目标值，则目标值在右半部分
            left = mid + 1; // left右移
        else // 否则目标值在左半部分
            right = mid - 1; // right左移
    }
    return -1; // 没有找到目标元素
}

int main() {
    vector<int> nums = {4,5,6,1,2,3};
    int target = 4;
    int index = search (nums, target);
    if (index != -1)
        cout << "找到元素" << target << endl << "下标是" << index << endl;
    else
        cout << "没有找到元素" << target << endl;
    return 0;
}

```

算法2难点分析

在寻找旋转点的过程中，如果 $\text{nums}[\text{mid}] > \text{nums}[\text{right}]$ 则旋转点在该序列的右半部分。比如{3, 4, 5, 6, 7, 1, 2}

mid为 $(0+6) / 2 = 3$, $nums[mid] = 6 > 2$, 可以知道围绕的旋转点1肯定在比较靠右的位置, 这样处于中间的mid的nums值才可以大于右边。这时, 我们将left赋值为mid+1, 这样可以将旋转点包在left和right内。经过不断循环, 可以得到循环点rot, 即1的下标。

在寻找target的时候, 出了得到mid, 我们还用了realmid, 其含义是mid下标在原正序序列中所对应的数。在此算法中可能不太好理解, 我们可以换思路理解。假设一个正序序列中寻找target元素, 每次将mid与target比较, 然后不断缩小比较区间。在此算法中, 将realmid类比正序序列的mid, 就可以容易理解该算法。

总结

在实现旋转序列的两种算法中, 第一种时间复杂度为 $O(n)$, 第二种时间复杂度为 $O(\log n)$, 第二种的性能更优越。