二分法

笔记本:	0_leetcode		
创建时间:	2020/7/29 19:58	**更新时间:**	2020/8/27 22:44
作者:	415669592@qq.com		
URL:	https://leetcode-cn.com/problems/search-in-rotated-sorted-array/		

二分法

0. 介绍

关键字

1. 有序数组

二分法的唯一关键词虽然是有序数组。但是很多时候，给定的我们输入项需要我们稍微做下转换。
比如IP地址的查找。一般给定的是点分十进制的字符串。

再比如求开跟。这些算是经典题目。

基本的二分

对于一个升序数组的基本二分，没什么好说的。这个应该是要做到可以直接默写的阶段。这样我们后续魔改二分的时候才能走下去。

```
static int search( const vector<int> &arr, int key ) {
    // TODO: Write your code here
    int start = 0;
    int end = arr.size() - 1;
    while ( start <= end ) {
        int mid = start + ( end - start ) / 2;
        if ( arr[mid] == key ) {
            return mid;
        } else if ( arr[mid] < key ) {
            start = mid + 1;
        } else {
            assert( arr[mid] > key );
            end = mid - 1;
        }
    }
    return -1;
}
```

1. 实战

1.1 基本二分查找

> 给定一个有序数组，但不知道其是升序还是降序。查找某个key的index。

e.g.1

```
Input: [4, 6, 10], key = 10
Output: 2
```

e.g.2

```
Input: [10, 6, 4], key = 10
Output: 0
```

思路

这道题的关键点就是我们并不知道其是升序还是降序。因此需要对我们的二分进行一丢丢改动就可以了。

算是比较合适的开胃菜了。

```
using namespace std;

#include <iostream>
#include <vector>
#include <cassert>

class BinarySearch {
  public:
    static int search( const vector<int> &arr, int key ) {
        // TODO: Write your code here
        int start = 0;
        int end = arr.size() - 1;
        bool is_ascending = arr[start] < arr[end] ? true : false;

        while ( start <= end ) {
            int mid = start + ( end - start ) / 2;
            if ( arr[mid] == key ) {
                return mid;
            } else {

                if ( is_ascending ) {
                    if ( arr[mid] < key ) {
                        start = mid + 1;
```

```
                    } else {
                        assert( arr[mid] > key );
                        end = mid - 1;
                    }
                } else {
                    if ( arr[mid] > key ) {
                        start = mid + 1;
                    } else {
                        assert( arr[mid] < key );
                        end = mid - 1;
                    }
                }
            }

        }
        return -1;
    }
};

int main( int argc, char *argv[] ) {
    cout << BinarySearch::search( vector<int> {4, 6, 10}, 10 ) << endl;
    cout << BinarySearch::search( vector<int> {1, 2, 3, 4, 5, 6, 7}, 5 ) <<
endl;
    cout << BinarySearch::search( vector<int> {10, 6, 4}, 10 ) << endl;
    cout << BinarySearch::search( vector<int> {10, 6, 4}, 4 ) << endl;

    system( "pause" );
}
```

1.2 lower_bound实现

> C++lower_bound实现
> 即给定一个升序数组，找到不小于key的值的index

e.g.1

```
Input: [4, 6, 10], key = 6
Output: 1
```

e.g.2

```
Input: [1, 3, 8, 10, 15], key = 12
Output: 4
```

这道题实际上再邓俊辉的数据结构是作为二分的基本题作为讲解。

我们拿一道基本的二分来讲解

```
static int search( const vector<int> &arr, int key ) {
    // TODO: Write your code here
    int start = 0;
    int end = arr.size() - 1;
    while ( start <= end ) {
        int mid = start + ( end - start ) / 2;
        if ( arr[mid] == key ) {
            return mid;
        } else if ( arr[mid] < key ) {
            start = mid + 1;
        } else {
            assert( arr[mid] > key );
            end = mid - 1;
        }
    }
    return -1;
}
```

根据while循环，我们知道其退出的条件只有1种。

1. start > end 实际上是start == end + 1.

不论什么样的二分，如果没有找到值会变成如下俩种情况。

start or mid	end
val: y	val: z
idnex: x	index: x+1

如果我们要找的值在y和z中间。arr[mid] < key
此时start = mid + 1 = end。
然后因为key < arr[mid] = z = arr[start]
end = mid - 1 此时由end条件退出。（如果key出现在y的左边，算同一种情况，都是由end条件退出的）

如果我们要找的值在z的右边。 arr[mid] <key
start = mid + 1 = end;
此时 key > arr[mid] = z = arr[start - 1]

start = mid + 1
那么就会由start条件退出。

不难发现一个规律就是[0,start)这个区间的数全部小于key。
而arr[start] >= key
证明这个直接使用反证法即可。即认为在while循环结束时arr[start] <key。

对于第一种情况来说：arr[start] = z > key这是我们的假设。直接和我们的假设相反。
对于第二种情况来说：start = x + 2这个位置。这里我们假设end条件被触发过一次。即
assert(arr[mid] > key);
end = mid - 1;
end + 1 = mid = x + 2 而arr[mid] > key
所以
arr[start] = arr[x+2] = arr[mid] > key 推翻。

> Ps: 之所以有这个结论的原因是因为C++取整方式问题。比如7.5是直接取7的。

结论

也就是说，只要end条件被触发一次，或者如果end条件一次不触发，就必须保证arr[end] < key，那么我们就一定能保证我们的算法合理性。

```
using namespace std;

#include <iostream>
#include <vector>
#include <cassert>

class CeilingOfANumber {
  public:
    static int searchCeilingOfANumber( const vector<int> &arr, int key ) {
        // TODO: Write your code here
        int start = 0;
        int end = arr.size() - 1;
        if ( key > arr[end] )
            return -1;
        // 上述判断保证了我们的arr[end] <= key
        // ? 好像有点不太一样哦和结论。注意arr[end] = key
        // 那么就一定会触发由end结束的条件。
        int mid = -1;
        while ( start <= end ) {
            mid = start + ( end - start ) / 2;
            if ( arr[mid] == key ) {
                end = mid - 1;
            } else if ( arr[mid] < key ) {
```

```
                start = mid + 1;
            } else {
                assert( arr[mid] > key );
                end = mid - 1;
            }
            /*
            if ( arr[mid] <= key ) {
                end = mid - 1;
            } else ( arr[mid] < key ) {
                start = mid + 1;
            }
            */
        }
        return start;
    }

};

int main( int argc, char *argv[] ) {
    cout << CeilingOfANumber::searchCeilingOfANumber( vector<int> {4, 6, 6},
6 ) << endl;
    cout << CeilingOfANumber::searchCeilingOfANumber( vector<int> {1, 3, 8,
10, 15}, 12 ) << endl;
    cout << CeilingOfANumber::searchCeilingOfANumber( vector<int> {4, 6, 10},
17 ) << endl;
    cout << CeilingOfANumber::searchCeilingOfANumber( vector<int> {4, 6, 10},
-1 ) << endl;
    cout << CeilingOfANumber::searchCeilingOfANumber( vector<int> {4, 6, 6,
10}, 6 ) << endl;
    cout << CeilingOfANumber::searchCeilingOfANumber( vector<int> {4, 6, 6,
9, 10}, 6 ) << endl;
    system( "pause" );
}
```

upper_bound

那么如何实现upper_bound呢?
upper_bound就是查找第一个大于key的值。没有等于。

由前面的结论，我们已经知道了如果arr[start] >= key
那么只要让我们保证arr[start] >key就可以了。
很简单，就是修改条件就可以。只要让==的时候，start继续向右移动，直到不等于。

```
while ( start <= end ) {
    mid = start + ( end - start ) / 2;
     if ( arr[mid] <= key ) {
        start = mid + 1;
    } else {
        assert( arr[mid] > key );
        end = mid - 1;
    }
}
// 注意，这里因为我们是找大于，所以可能越界，即start = arr.size()
```

> 虽然我自己平常做题都是直接使用upper_bound和lower_bound的。但是了解其原理还是比较重要的。

1.3 寻找下一个字符

> 给定一系列字符，寻找比起它的第一个字符。假设数组是循环的。即找不到的话从头开始找。

e.g.1

```
Input: ['a', 'c', 'f', 'h'], key = 'f'
Output: 'h'
Explanation: The smallest letter greater than 'f' is 'h' in the given array.
```

e.g.2

```
Input: ['a', 'c', 'f', 'h'], key = 'm'
Output: 'a'
Explanation: As the array is assumed to be circular, the smallest letter
greater than 'm' is 'a'.
```

思路

额，直接upper_bound就解决了。

```
using namespace std;

#include <iostream>
#include <vector>

class NextLetter {
  public:
    static char searchNextLetter( const vector<char> &letters, char key ) {
        // TODO: Write your code here

        int start = 0;
        int end = letters.size() - 1;
        if ( key > letters[end] ) {
            return letters[0];
        }
        while ( start <= end ) {
            int mid = start + ( end - start ) / 2;
            if ( letters[mid] <= key ) {
                start = mid + 1;
            } else {
                end = mid - 1;
            }
        }
        if ( start == letters.size() )
            return letters[0];
        else
            return letters[start];
    }
};

int main( int argc, char *argv[] ) {
    cout << NextLetter::searchNextLetter( vector<char> {'a', 'c', 'f', 'h'},
'f' ) << endl;
    cout << NextLetter::searchNextLetter( vector<char> {'a', 'c', 'f', 'h'},
'b' ) << endl;
    cout << NextLetter::searchNextLetter( vector<char> {'a', 'c', 'f', 'h'},
'm' ) << endl;
    cout << NextLetter::searchNextLetter( vector<char> {'a', 'c', 'f', 'h'},
'h' ) << endl;
    system( "pause" );
}
```

1.4 寻找区间

> 给定有序数组，寻找key的range。这个range指的的第一个和最后一个key出现的位置。

e.g.1

```
Input: [4, 6, 6, 6, 9],
key = 60utput: [1, 3]
```

e.g.2

```
Input: [1, 3, 8, 10, 15],
key = 100utput: [3, 3]
```

e.g.3

```
Input: [1, 3, 8, 10, 15],
key = 120utput: [-1, -1]
```

思路

额，就是一个lower_bound，一个upper_bound的事情。

这里用了一个哨兵位置。如果熟悉C++迭代器的话，就是STL容器中的end位置。即如果key >arr[arr.size() - 1]。返回的是arr.size()位置。

然后我们知道lower_bound返回的是闭区间，而upper_bound则是闭区间。所以最后有点处理。

```
using namespace std;

#include <iostream>
#include <vector>

class FindRange {
  public:
    static int upper_search( const vector<int> &arr, int key, int start, int
end ) {
```

```
        if ( key > arr[end] )
            return end + 1;
        while ( start <= end ) {
            int mid = start + ( end - start ) / 2;
            if ( arr[mid] <= key ) {
                start = mid + 1;
            } else {
                end = mid - 1;
            }
        }
        return start;
    }
    static int lower_search( const vector<int> &arr, int key, int start, int
end ) {
        if ( key > arr[end] )
            return end + 1;
        while ( start <= end ) {
            int mid = start + ( end - start ) / 2;
            if ( arr[mid] < key )
                start = mid + 1;
            else
                end = mid - 1;
        }
        return start;
    }
    static pair<int, int> findRange( const vector<int> &arr, int key ) {
        pair<int, int> result( -1, -1 );
        // TODO: Write your code here
        int start = lower_search( arr, key, 0, arr.size() - 1 );
        int end = upper_search( arr, key, start, arr.size() - 1 );
        if ( start < end ) {
            result.first = start;
            result.second = end - 1;
        }
        return result;
    }
};

int main( int argc, char *argv[] ) {
    pair<int, int> result = FindRange::findRange( vector<int> {4, 6, 6, 6,
9}, 6 );
    cout << "Range: [" << result.first << ", " << result.second << "]" <<
endl;
    result = FindRange::findRange( vector<int> {1, 3, 8, 10, 15}, 10 );
```

```
    cout << "Range: [" << result.first << ", " << result.second << "]" <<
endl;
    result = FindRange::findRange( vector<int> {1, 3, 8, 10, 15}, 12 );
    cout << "Range: [" << result.first << ", " << result.second << "]" <<
endl;
    system( "pause" );
}
```

阶段总结

类似upper_bound和lower_bound的做法很多。作为最稳妥起见的手法（说实话，有时候会忘记二分的一些细节），自然是使用STL。
这里以上一题为例，使用STL示意下。

这里需要熟悉STL。

```
using namespace std;

#include <iostream>
#include <vector>
#include <algorithm>

class FindRange {
  public:
    static pair<int, int> findRange( const vector<int> &arr, int key ) {
        pair<int, int> result( -1, -1 );
        // TODO: Write your code here
        auto start_iter = lower_bound( arr.begin(), arr.end(), key );
        auto end_iter = upper_bound( start_iter, arr.end(), key );
        if ( distance( start_iter, end_iter ) > 0 ) {
            result.first = distance( arr.begin(), start_iter );
            result.second = distance(  arr.begin(), end_iter ) - 1;
        }
        return result;
    }
};

int main( int argc, char *argv[] ) {
    pair<int, int> result = FindRange::findRange( vector<int> {4, 6, 6, 6,
9}, 6 );
    cout << "Range: [" << result.first << ", " << result.second << "]" <<
```

```
endl;
    result = FindRange::findRange( vector<int> {1, 3, 8, 10, 15}, 10 );
    cout << "Range: [" << result.first << ", " << result.second << "]" <<
endl;
    result = FindRange::findRange( vector<int> {1, 3, 8, 10, 15}, 12 );
    cout << "Range: [" << result.first << ", " << result.second << "]" <<
endl;
    system( "pause" );
}
```

简单来说就是需要如下知识点

1. 了解迭代器。
2. 了解advance, distance, next等用法。
3. 了解begin,end迭代器的语义。

1-5 找相近的数

> 给定有序数组， 给定key，找到数组中和key绝对差最小的index

e.g.1

```
Input: [4, 6, 10], key = 7
Output: 6
Explanation: The difference between the key '7' and '6' is minimum than any
other number in the array
```

e.g.2

```
Input: [4, 6, 10], key = 4
Output: 4`
```

思路

感觉是直接使用lower_bound找到index，然后左右去查找的思路，因此需要构建一个例子

[1 2 3 4 10] key = 5
利用lower_bound 找到10.然后再找到4。
然后通过判断返回结果。

```
using namespace std;

#include <iostream>
#include <vector>
#include <algorithm>

class MinimumDifference {
  public:
    static int searchMinDiffElement( const vector<int> &arr, int key ) {
        // TODO: Write your code here
        auto iter = lower_bound( arr.begin(), arr.end(), key );
        if ( iter == arr.end() )
            return arr[arr.size() - 1];
        int right_index = distance( arr.begin(), iter );
        int left_index = right_index - 1;
        if ( left_index < 0 )
            return arr[right_index];
        return arr[right_index] - key > arr[left_index] - key ?
arr[left_index] : arr[right_index];
    }
};

int main( int argc, char *argv[] ) {
    cout << MinimumDifference::searchMinDiffElement( vector<int> {4, 6, 10},
7 ) << endl;
    cout << MinimumDifference::searchMinDiffElement( vector<int> {4, 6, 10},
4 ) << endl;
    cout << MinimumDifference::searchMinDiffElement( vector<int> {1, 3, 8,
10, 15}, 12 ) << endl;
    cout << MinimumDifference::searchMinDiffElement( vector<int> {4, 6, 10},
17 ) << endl;
    system( "pause" );
}
```

```
using namespace std;

#include <iostream>
#include <vector>
#include <algorithm>

class MinimumDifference {
  public:
```

```
    static int mylower_bound( const vector<int> &arr, int key ) {
        int start = 0;
        int end = arr.size() - 1;
        if ( key > arr[end] )
            return arr[end];
        while ( start <= end ) {
            int mid = start + ( end - start ) / 2;
            if ( arr[mid] <= key )
                start = mid + 1;
            else
                end = mid - 1;
        }
        int right = start;
        int left = start - 1;
        if ( left < 0 )
            return arr[right];

        return arr[right] - key > arr[left] - key ? arr[left] : arr[right];
    }
    static int searchMinDiffElement( const vector<int> &arr, int key ) {
        return mylower_bound( arr, key );
    }
};

int main( int argc, char *argv[] ) {
    cout << MinimumDifference::searchMinDiffElement( vector<int> {4, 6, 10},
7 ) << endl;
    cout << MinimumDifference::searchMinDiffElement( vector<int> {4, 6, 10},
4 ) << endl;
    cout << MinimumDifference::searchMinDiffElement( vector<int> {1, 3, 8,
10, 15}, 12 ) << endl;
    cout << MinimumDifference::searchMinDiffElement( vector<int> {4, 6, 10},
17 ) << endl;
    system( "pause" );
}
```

1-6 找最值

> 在波峰序列中找最值。arr[i] != arr[i+1]

e.g.1

```
Input: [1, 3, 8, 12, 4, 2]
Output: 12
Explanation: The maximum number in the input bitonic array is '12'.
```

e.g.2

```
Input: [3, 8, 3, 1]
Output: 8
```

思路

根据以上题目，我们已经知道，这些魔改的二分查找主要就是去修改二分。修改二分主要就是添加一些对应题目的判断条件。

1. 先想办法找到有序数组。很自然地想到要去根据start, mid, end进行判断。
2. 根据start和end进行比较， 我们发现根据这俩个值，好像无法判断有序。但是，我们知道，使用一个mid区分数组后，俩个中，一定有一个是有序的。我们可以根据arr[mid],arr[mid+1]这俩个值来判断升降。
3. 答案已经出来了。剩下的就是一些细节问题。

```
using namespace std;

#include <iostream>
#include <vector>
#include <assert.h>

class MaxInBitonicArray {
  public:
    static int findMax( const vector<int> &arr ) {
        // TODO: Write your code here
        int start = 0;
        int end = arr.size() - 1;
        while ( start < end ) {
            int mid = start + ( end - start ) / 2;
            if ( arr[mid] > arr[mid + 1] ) {
                end = mid;
            } else {
                start = mid + 1;
            }
        }
        return arr[start];
```

```
    }

};

int main( int argc, char *argv[] ) {
    cout << MaxInBitonicArray::findMax( vector<int> {1, 3, 8, 12, 4, 2} ) <<
endl;
    cout << MaxInBitonicArray::findMax( vector<int> {3, 8, 3, 1} ) << endl;
    cout << MaxInBitonicArray::findMax( vector<int> {1, 3, 8, 12} ) << endl;
    cout << MaxInBitonicArray::findMax( vector<int> {10, 9, 8} ) << endl;
    system( "pause" );
}
```

1. while结束条件发生了变化，根据题意，我们知道，这个最大值是一定存在的。所以start == end的时候，就是找到了值。
2. 其次，要注意边界值。arr[mid] > arr[mid+1] 因为arr[mid]有可能是最大值，所以end = mid。同理对于start。即我们通过二分多滤掉一定不是我们要的区间。最后留下的就是我们要的区间。

1.7 在峰值区间查找

> 同样是在峰值区间，这次是查找值，值不存在，则返回-1。

e.g.1

```
Input: [1, 3, 8, 4, 3], key=4
Output: 3
```

e.g.2

```
Input: [1, 3, 8, 12], key=12
Output: 3
```

思路

俩张解法

1. 直接二分查找，靠判断出来的有序区间进行二分。

```
using namespace std;

#include <iostream>
#include <vector>

class SearchBitonicArray {
  public:
    static int bsearch( const vector<int> &arr, int start, int end, int key,
bool is_ascending  = true ) {
        while ( start <= end ) {
            int mid = start + ( end - start ) / 2;
            if ( arr[mid] == key )
                return mid;
            else {
                if ( is_ascending ) {
                    if ( arr[mid] > key )
                        end = mid - 1;
                    else
                        start = mid + 1;
                } else {
                    if ( arr[mid] < key )
                        end = mid - 1;
                    else
                        start = mid + 1;
                }
            }
        }
        return -1;
    }
    static int search( const vector<int> &arr, int key ) {
        // TODO: Write your code here
        int start = 0;
        int end = arr.size() - 1;
        while ( start <= end ) {
            int mid = start + ( end - start ) / 2;
            if ( arr[mid] == key )
                return mid;
            if ( arr[mid] > arr[mid+ 1] ) {// 判断[mid, end]为降序
                if( key < arr[mid] )
                    return bsearch( arr, mid, end, key, false );
                end = mid - 1;
            } else {// 判断[start, mid + 1]为升序
                if ( key < arr[mid + 1] )
                    return bsearch( arr, start, mid + 1, key );
```

```
                start = mid + 1;
            }
        }
        return -1;
    }
};

int main( int argc, char *argv[] ) {
    cout << SearchBitonicArray::search( vector<int> {1, 3, 8, 4, 3}, 4 ) <<
endl;
    cout << SearchBitonicArray::search( vector<int> {3, 8, 3, 1}, 8 ) <<
endl;
    cout << SearchBitonicArray::search( vector<int> {1, 3, 8, 12}, 12 ) <<
endl;
    cout << SearchBitonicArray::search( vector<int> {10, 9, 8}, 10 ) << endl;
    system( "pause" );
}
```

2. 靠上题的思路，先找到最大值。然后将数组划分成为俩个有序数组。进行二分查找。

```
using namespace std;

#include <iostream>
#include <vector>

class SearchBitonicArray {
  public:
    static int search( const vector<int> &arr, int key ) {
        int maxIndex = findMax( arr );
        int keyIndex = binarySearch( arr, key, 0, maxIndex );
        if ( keyIndex != -1 ) {
            return keyIndex;
        }
        return binarySearch( arr, key, maxIndex + 1, arr.size() - 1 );
    }

    // find index of the maximum value in a bitonic array
    static int findMax( const vector<int> &arr ) {
        int start = 0, end = arr.size() - 1;
        while ( start < end ) {
            int mid = start + ( end - start ) / 2;
            if ( arr[mid] > arr[mid + 1] ) {
                end = mid;
```

```
            } else {
                start = mid + 1;
            }
        }

        // at the end of the while loop, 'start == end'
        return start;
    }

  private:
    // order-agnostic binary search
    static int binarySearch( const vector<int> &arr, int key, int start, int
end ) {
        while ( start <= end ) {
            int mid = start + ( end - start ) / 2;

            if ( key == arr[mid] ) {
                return mid;
            }

            if ( arr[start] < arr[end] ) { // ascending order
                if ( key < arr[mid] ) {
                    end = mid - 1;
                } else {  // key > arr[mid]
                    start = mid + 1;
                }
            } else {  // descending order
                if ( key > arr[mid] ) {
                    end = mid - 1;
                } else {  // key < arr[mid]
                    start = mid + 1;
                }
            }
        }
        return -1;  // element is not found
    }
};

int main( int argc, char *argv[] ) {
    cout << SearchBitonicArray::search( vector<int> {1, 3, 8, 4, 3}, 4 ) <<
endl;
    cout << SearchBitonicArray::search( vector<int> {3, 8, 3, 1}, 8 ) <<
endl;
    cout << SearchBitonicArray::search( vector<int> {1, 3, 8, 12}, 12 ) <<
```

```
endl;
    cout << SearchBitonicArray::search( vector<int> {10, 9, 8}, 10 ) << endl;
    system( "pause" );
}
```

1.8 搜索旋转排序数组(leetcode 33)

> 数组rotate有序。不存在重复元素。查询key的index

e.g.1

```
Input: [10, 15, 1, 3, 8], key = 15
Output: 1
```

思路

这里有个隐藏条件，即如果数组有序arr[start] < arr[end]可以直接二分。
否则assert(arr[start] > arr[end]);

俩种思路：

1. 利用二分寻找到起始点。然后对其做虚拟节点的转换。即认为自己在0，end上做二分。但在实际取数据的时候要进行转换为实际值。

这里的思路是

1. 利用二分先找到最值点。（这也是一道leetcode题目，leetcode-153）

说白了还是去玩start, mid, end这三个index。
我们一定要保证在循环中，[start, end]这个区间内存在那个值的坠落。

我们首先要想个条件来过滤一部分。这个条件实际上可以随便取一个。这里举来个例子。
arr[mid] > arr[start] 这里我们不知道mid出现在哪个区间。但是第一个坑定是出现在左边这个区间的。那么我们就让start永远呆在左边界。当mid出现在右边的时候，这个条件就不会被触发。
start = mid还是mid + 1呢?
前面说了，我们的start,end区间一定要存在那个值的坠落。如果选取mid+1，有可能start就会进入右边的区间。所以start = mid
arr[mid] <= arr[start] 这个条件发生时。mid一定在右边的区间。
这里选取end = mid还是end = mid - 1呢?
选取end = mid。

我们结束条件应该是end + 1 - start <= 2即 start >= end - 1 ，那么循环条件就是start <end - 1

```
using namespace std;

#include <iostream>
#include <vector>
#include <cassert>

class SearchRotatedArray {
  public:
    static int bsearch( const vector<int> &arr, int start, int end, int key )
{
        while ( start <= end ) {
            int mid = start + ( end - start ) / 2;
            if ( arr[mid] == key )
                return mid;
            else if ( arr[mid] > key )
                end = mid - 1;
            else
                start = mid + 1;
        }
        return -1;
    }
    static int search( const vector<int> &arr, int key ) {
        // TODO: Write your code here
        int start = 0;
        int end = arr.size() - 1;
        while ( start < end - 1 ) {
            int mid = start + ( end - start ) / 2;
            // 根据mid来判断哪边有序
            if ( arr[mid] > arr[start] ) {
                start = mid;
            } else {
                // arr[mid] <= arr[start]
                // mid, end 中间一定存在最大值。
                end = mid;
            }
        }
        return start ;
    }
};

int main( int argc, char *argv[] ) {
```

```
    cout << SearchRotatedArray::search( vector<int> {10, 15, 1, 3, 8}, 15 )
<< endl;
    cout << SearchRotatedArray::search( vector<int> {4, 5, 7, 9, 10, -1, 2},
10 ) << endl;
    system( "pause" );
}
```

换一个
arr[mid] > arr[end] , [0, mid] 一定在左边。
start = mid;
arr[mid] <= arr[end] 第一次一定出现在右边。因为我们让end不越过左右俩个升序数组的界。所以，mid出现在左边时。这个条件不会被触发。
end = mid;

```
static int search( const vector<int> &arr, int key ) {
    // TODO: Write your code here
    int start = 0;
    int end = arr.size() - 1;
    while ( start < end - 1 ) {
        int mid = start + ( end - start ) / 2;
        // 根据mid来判断哪边有序
        if ( arr[mid] > arr[end] ) {
            start = mid;
        } else {
            // arr[mid] <= arr[start]
            // mid, end 中间一定存在最大值。
            end = mid;
        }
    }
    return start ;
}
```

当你把题吃透了之后，只要你有大体思路，后续就是修修补补。怎么都能通。

现在我们知道了数组真正end的位置。
接下来使用一套映射规则做二分即可。就是将实际的start，end映射到理想的0,arr.size() - 1。所有数字都通过此映射。在bsearch看来，就是工作在一个正常的有序数组上。

```
using namespace std;

#include <iostream>
```

```cpp
#include <vector>
#include <cassert>

class SearchRotatedArray {
  public:
    int bsearch( const vector<int> &arr, int start, int end, int key ) {
        while ( start <= end ) {
            int mid = start + ( end - start ) / 2;
            if ( arr[ideal_to_real( mid, relative, arr.size() )] == key )
                return mid;
            else if ( arr[ideal_to_real( mid, relative, arr.size() )] > key )
                end = mid - 1;
            else
                start = mid + 1;
        }
        return -1;
    }
    int search_max( const vector<int> &arr, int key ) {
        // TODO: Write your code here
        int start = 0;
        int end = arr.size() - 1;
        while ( start < end - 1 ) {
            int mid = start + ( end - start ) / 2;
            // 根据mid来判断哪边有序
            if ( arr[mid] > arr[end] ) {
                start = mid;
            } else {
                // arr[mid] <= arr[start]
                // mid, end 中间一定存在最大值。
                end = mid;
            }
        }
        return start ;
    }
    int relative = 0;
    int search( const vector<int> &arr, int key ) {
        int real_end = search_max( arr, key );
        int real_start = ( real_end + 1 ) % arr.size();
        relative = arr.size() - 1 - real_end;
        int index = bsearch( arr, real_to_ideal( real_start, relative,
arr.size() ), real_to_ideal( real_end, relative, arr.size() ), key );
        if ( index == -1 )
            return index;
        return ideal_to_real( index, relative, arr.size() );
```

```
    }
    int real_to_ideal( int real_index, int relative, int size ) {
        return ( real_index + relative ) % size;
    }
    int ideal_to_real( int ideal_index, int relative, int size ) {
        return ( ideal_index - relative + size ) % size;
    }
};

int main( int argc, char *argv[] ) {
    cout << SearchRotatedArray{} .search( vector<int> {10, 15, 1, 3, 8}, 15 )
<< endl;
    cout << SearchRotatedArray{} .search( vector<int> {10, 15, 1, 3, 8}, 10 )
<< endl;
    cout << SearchRotatedArray{} .search( vector<int> {10, 15, 1, 3, 8}, 9 )
<< endl;
    cout << SearchRotatedArray{} .search( vector<int> {4, 5, 7, 9, 10, -1,
2}, 10 ) << endl;
    system( "pause" );
}
```

2. 利用二分判断哪边有序，如果有序，就可以判断值是否在这里。

如果画了图，实际上一目了然。
对于上题来说，是俩个有序的，不过一个升序，一个降序。而在这里则是均为升序。

关键就是如何判断哪个区间是有序的。

```
using namespace std;

#include <iostream>
#include <vector>
#include <cassert>

class SearchRotatedArray {
  public:
    static int bsearch( const vector<int> &arr, int start, int end, int key )
{
        while ( start <= end ) {
            int mid = start + ( end - start ) / 2;
            if ( arr[mid] == key )
                return mid;
            else if ( arr[mid] > key )
                end = mid - 1;
```

```
            else
                start = mid + 1;
        }
        return -1;
    }
    static int search( const vector<int> &arr, int key ) {
        // TODO: Write your code here
        int start = 0;
        int end = arr.size() - 1;
        while ( start <= end ) {
            int mid = start + ( end - start ) / 2;
            // 根据mid来判断哪边有序
            if ( arr[mid] == key )
                return mid;
            if ( arr[mid] < arr[end] ) {
                // start - mid+1升序
                if ( key >= arr[mid] && key <= arr[end] ) {
                    return bsearch( arr, mid, end, key );
                }
                end = mid - 1;
            } else {
                assert( arr[mid] >= arr[start] );
                if ( key >= arr[start ] && key <= arr[mid] ) {
                    return bsearch( arr, start, mid, key );
                }
                start = mid + 1;
            }
        }
        return -1;
    }
};

int main( int argc, char *argv[] ) {
    cout << SearchRotatedArray::search( vector<int> {10, 15, 1, 3, 8}, 15 )
<< endl;
    cout << SearchRotatedArray::search( vector<int> {4, 5, 7, 9, 10, -1, 2},
10 ) << endl;
    system( "pause" );
}
```

最后是题目给出的解法

这个方法更加简单直白。额自己理解吧。作为开阔眼界使用吧。个人喜欢自己的第二种方法。

```
using namespace std;

#include <iostream>
#include <vector>

class SearchRotatedArray {
 public:
  static int search(const vector<int>& arr, int key) {
    int start = 0, end = arr.size() - 1;
    while (start <= end) {
      int mid = start + (end - start) / 2;
      if (arr[mid] == key) {
        return mid;
      }

      if (arr[start] <= arr[mid]) {  // left side is sorted in ascending
order
        if (key >= arr[start] && key < arr[mid]) {
          end = mid - 1;
        } else {  // key > arr[mid]
          start = mid + 1;
        }
      } else {  // right side is sorted in ascending order
        if (key > arr[mid] && key <= arr[end]) {
          start = mid + 1;
        } else {
          end = mid - 1;
        }
      }
    }

    // we are not able to find the element in the given array
    return -1;
  }
};

int main(int argc, char* argv[]) {
  cout << SearchRotatedArray::search(vector<int>{10, 15, 1, 3, 8}, 15) <<
endl;
  cout << SearchRotatedArray::search(vector<int>{4, 5, 7, 9, 10, -1, 2}, 10)
<< endl;
}
```

2. 总结

二分法总结

1. 因为二分法的边界判断常常出问题。所以建议先使用纸和笔画图，模拟一遍。
2. 魔改二分比较重要的点是寻找有序区间。一般题目都是那种Like有序的那种状态。根据start, mid, end这三个来判断。
3. 注意结束条件，即while循环的编写。一般模拟剩下2个元素，或者1个元素。这俩种情况基本就能保证循环条件不错。
4. 学会使用STL中的俩个二分查找。lower_bound，upper_bound。