二分搜索

```
二分搜索不好记，左右边界让人迷；
小于等于变小于，mid加一再减一；
搜索一个元素时，搜索区间两端闭；
while条件带等号，否则需要打补丁；
if相等就返回,其他事情甭操心；
mid必须加减一，因为区间两端闭；
while结束就凉凉，凄凄惨惨返-1；
搜索左右边界时，搜索区间要阐明；
左闭右开最常见，其余逻辑自明白；
while要用小于号，这样才能不漏掉；
if相等别返回，利用mid锁边界；
mid加一或减一，要看区间开或闭；
while结束不算完，结果记得要返回；
索引可能出边界，if检查报平安.
```

二分查找框架

```cpp
int binarySearch(std::vector<int> &nums, int target) {
  int left = 0, right = ...;
  while(...) {
    int mid = left + (right - left) /2;
    if(nums[mid] == target) {
      ...
    } else if (nums[mid] > target) {
      right = ....;
    } else if (nums[mid] < target) {
      left = ...;
    }
  }

  return ...;
}
```

二分搜索逻辑统一：

1. 查找元素

> 因为我们初始化 right = nums.length - 1 所以决定了我们的「搜索区间」是 [left, right] 所以决定了 while (left <= right) 同时也决定了 left = mid+1 和 right = mid-1 因为我们只需找到一个 target 的索引即可 所以当 nums[mid] == target 时可以立即返回

2. 寻找左侧边界的二分查找

> 因为我们初始化 right = nums.length 所以决定了我们的「搜索区间」是 [left, right) 所以决定了 while (left < right) 同时也决定了 left = mid + 1 和 right = mid 因为我们需找到 target 的最左侧索引 所以当 nums[mid] == target 时不要立即返回 而要收紧右侧边界以锁定左侧边界

3. 寻找右侧边界的二分查找

> 因为我们初始化 right = nums.length 所以决定了我们的「搜索区间」是 [left, right) 所以决定了 while (left < right) 同时也决定了 left = mid + 1 和 right = mid 因为我们需找到 target 的最右侧索引 所以当 nums[mid] == target 时不要立即返回 而要收紧左侧边界以锁定右侧边界 又因为收紧左侧边界时必须 left = mid + 1 所以最后无论返回 left 还是 right，必须减一

搜索区间两端封闭的框架：

1. 查找元素

```
int binarySearch(std::vector<int> &nums, int target) {
  int left = 0, right = nums.size() - 1;
  while(left <= right) {
    int mid = left + (right - left) /2;
    if(target == nums[mid]) {
      return mid;
    } else if(target > nums[mid]) {
      left = mid + 1;
    } else if(target < nums[mid]) {
      right = mid - 1;
    }
  }

  return -1;
}
```

2.查找左边界

```
int left_bound(std::vector<int> &nums, int target) {
  int left = 0, right = nums.size()-1;
  while(left <= right) {
    int mid = left + (right - left) /2;
    if(target == nums[mid]) {
      right = mid - 1;
    } else if(target < nums[mid]) {
      left = mid + 1;
    } else if(target > nums[mid]) {
      right = mid - 1;
    }
  }

  if(left >= nums.size() || nums[left] != target) {
    return -1;
  }
  return mid;
}
```

3. 寻找右边界

```
int right_bound(std::vector<int> &nums, int target){
  int left = 0, right = nums.size() - 1;
  while(left <= right) {
    int mid = left + (right - left) /2;
    if(target == nums[mid]) {
      left = mid + 1;
    } else if(target < nums[mid]) {
      left = mid + 1;
    } else if(target > nums[mid]) {
      right = mid - 1;
    }
  }

  if(right < 0 || nums[right] != target) {
    return -1;
  }
  return mid;
}
```

分析： **二分选择算法的左边界理解： 1. 返回指定nums中大于等于target的最小元素索引； 2. 返回指定nums中应该插入在target的位置； 3. 返回指定nums中小于target元素的个数。 二分选择算法的右边界理解： 1. 返回指定nums中小于等于target的最小元素索引； 2. 返回指定nums中应该插入在target的位置； 3. 返回指定nums中大于target元素的个数。**

二分查找的应用：

1.有序旋转数组中找到最小值 分析：

- 1. 如果arr[low] < arr[high]表示数组没有旋转直接返回arr[low];
- 2. 令mid = low + (hihg - low) / 2,即mid为[low, right]的中间位置：
  - 如果arr[low] > arr[mid] 表明断点一定在arr[low...mid]上，令high = mid，重复1；
  - 如果arr[mid] > arr[high]表明断点一定在arr[mid...high]上，令low = mid+1, 重复1；
  - 如果arr[mid] == arr[low] == arr[high]：
    - 生成i=low,向右遍历：
      - 如果存在arr[i] < arr[low], arr[i]即为所求；
      - 如果存在arr[i] > arr[low],断点在[i...mid]，则令high = mid,继续二分查找；
      - 如果arr[i] == arr[low], i 从low到mid，则断点在[mid...high]上，令low = mid，继续二分。

```
int getMin(std::vector<int> &arr) {
  int low = 0,high = arr.size();
  int mid =0;
```

```
    while(low < high){
      if(low == high -1) {
        break;
      }

      if(arr[low] < arr[high]) {
        return arr[low];
      }

      mid = low + (high - low) / 2;
      if(arr[mid] >  arr[high]) {
        low = mid;
        continue;
      }

      if(arr[mid] < arr[low]){
        high = mid;
        continue;
      }

      while(low < mid ) {
        if(arr[low] == arr[mid]) {
          low++;
        } else if(arr[low] < arr[mid]){
          return arr[low];
        }else{
          high = mid;
          break;
        }
      }
    }

    return std::min(nums[low], nums[high]);
  }
```

2.有序旋转数组中找到一个数

- 1. 使用low和high表示arr上的一个范围，每次判断num是否在arr[low...high],初始时，low=0,high=arr.size()-1,进入步骤 2:
- 2. 如果low > high,直接进入步骤 5，令变量mid = low + (high - low)/2,如果arr[mid] == num，直接返回true;
- 3. 此时arr[num] != num,如果发现arr[low], arr[mid], arr[high]三个值都不相等,直接进入步骤 4.如果发现三个值相等，则无法判断断点位置在mid的哪一侧.处理方式如下:
  - 只要arr[num] == arr[mid],就将low向右移动，如果出现arr[low] != arr[mid]表示，arr[low ... mid ... right]上可以判断出断点位置，进入步骤 4:
- 4. 当arr[mid] != num 且 arr[low], arr[mid], arr[high]不相等， 表示一定可以二分，判断逻辑如下:
  - 情况一: arr[low] < arr[mid]则断点一定在arr[mid]的右侧，此时arr[low]和arr[mid]之间数据有序:
    - 当num >= arr[low] && num < arr[mid]说明num只需要在arr[low...mid]上寻找，令high = mid - 1,进入步骤 2;

        - 不满足上述条件，令`low = mid+1`,进入步骤 2；
    - 情况二，当不满足情况一时，`arr[mid ... high]`，一定有序：
        - 如果`num > arr[mid] && num <= arr[high]`， 令`low = mid+1`,进入步骤 2；
        - 否则`high = mid - 1`, 进入步骤 2.

代码如下：

```
bool isCantains(std::vector<int> &nums, int target){
  int low = 0, high = nums.size() - 1;
  int mid = 0;
  while(low <= high){
    mid = low + (high - low) / 2;
    if(nums[mid] == num) {
      return true;
    }

    if(arr[low] == arr[mid] && arr[mid] == arr[high]){
      while(low != mid && arr[low] == arr[mid]) {
        low++;
      }

      if(low == mid){
        low = mid+1;
        continue;
      }
    }

    if(arr[low] != arr[mid]){
      if(arr[mid] > arr[low]){
        if(num >= arr[low] && num < arr[mid]){
          high = mid  - 1;
        }else{
          low = mid + 1;
        }
      }else{
        if(num > arr[mid] && num <= arr[hihg]){
          low = mid + 1;
        }else{
          hihg = mid - 1;
        }
      }
    } else {
      if(arr[mid] < arr[high]){
        if(num > arr[mid] && num <= arr[high]){
          low = mid + 1;
        }else {
          high = mid - 1;
        }
      }else {
        if(num >= arr[low] && num < arr[mid]){
          high = mid - 1;
        }else{
```

```
            low = mid + 1;
          }
        }
      }

    }

    return false;
  }
```