

重庆师范大学

实验报告

实验课程名称： 算法设计与分析

实验序号： 3

实验内容： 二分搜索

班级： 23计科6班

姓名： 周子依

学号： 2023051603162

2024年3月13日

实验目的与要求

一、学习目标

- 理解二分搜索核心思想：理解二分搜索算法的原理并能复现代码。
- 复杂度分析：能推导二分搜索算法的时间与空间复杂度
- 实际应用能力：通过实验案例，修改错误之处。

实验内容

一、实验原理

二分搜索（Binary Search）是一种在有序数组中高效查找目标元素的算法，其核心思想是通过不断将搜索区间减半来缩小查找范围，具体步骤如下：

- 前提条件：数组必须预先排序。
- 步骤：
 - 初始化左右指针，分别指向数组首尾。
 - 循环计算中间位置，将中间元素与目标值比较：
 - 若相等，返回中间位置。
 - 若目标值更大，调整左指针到中间位置右侧。
 - 若目标值更小，调整右指针到中间位置左侧。
 - 直到找到目标值或确定其不存在。

二、设计思路

1、学习书上的正确代码并复现

代码：

```
1  #include <iostream>
2  using namespace std;
3
4  template <typename Type>
5  int BinarySearch (Type a[], const Type& x, int n) {
6      int left = 0, right = n - 1;
7      while (left <= right) {
8          int middle = (left + right) / 2;
9          if (x == a[middle])
10             return middle;
11          if (x > a[middle])
12             left = middle + 1;
13          else
14             right = middle - 1;
15      }
16      return -1;
```

```

17 }
18
19 int main() {
20     int arr[] = {1, 3, 5, 7, 9};
21     int target = 5;
22     int size = sizeof(arr) / sizeof(arr[0]);
23     int result = BinarySearch(arr, target, size);
24     if (result != -1)
25         cout << "元素 " << target << " 在数组中的索引是: " << result << endl;
26     else
27         cout << "元素 " << target << " 不在数组中。" << endl;
28     return 0;
29 }

```

运行结果:

 stdout

元素 5 在数组中的索引是: 2

2、对代码逐一理解

(1) 函数定义和参数

```

1  template <typename Type>
2  int BinarySearch (Type a[], const Type& x, int n)

```

- `template <typename Type>`: 这是一个模板声明, 它允许函数处理不同数据类型的数组, 比如 `int`、`double` 等。
- `a[]`: 要进行搜索的已排序数组。
- `x`: 要查找的目标元素, 使用 `const Type&` 引用传递, 避免不必要的复制。
- `n`: 数组的长度。

(2) 初始化左右边界

```

1  int left = 0, right = n - 1;

```

- `left` 初始化为 0, 表示数组的起始位置。
- `right` 初始化为 `n - 1`, 表示数组的最后一个元素的位置。

(3) 循环条件

```

1  while (left <= right)

```

只要左边界小于等于右边界, 就继续在这个范围内进行搜索。

(4) 计算中间位置

```
1 int middle = (left + right) / 2;
```

计算当前搜索范围的中间位置。

(5) 比较目标元素和中间元素

```
1 if (x == a[middle])
2     return middle;
```

如果目标元素等于中间元素，说明找到了目标元素，返回中间元素的索引。

(6) 调整搜索范围

```
1 if (x > a[middle])
2     left = middle + 1;
3 else
4     right = middle - 1;
```

- 如果目标元素大于中间元素，说明目标元素在中间元素的右侧，更新左边界为 `middle + 1`。
- 如果目标元素小于中间元素，说明目标元素在中间元素的左侧，更新右边界为 `middle - 1`。

(7) 未找到目标元素

```
1 return -1;
```

如果循环结束后仍然没有找到目标元素，返回 -1 表示未找到。

3、复杂度分析

- **时间复杂度：** $O(\log n)$ ，因为每次迭代都将搜索范围缩小一半。
- **空间复杂度：** $O(1)$ ，只使用了常数级的额外空间。

二分搜索算法的前提是数组必须是有序的，否则无法保证搜索的正确性。


三、分析代码的错误性

1、错误代码一：

```

1  int BinarySearch (Type a[], const Type& x, int n) {
2      int left = 0, right = n - 1;
3      while (left <= right) {
4          int middle = (left + right) / 2;
5          if (x == a[middle])
6              return middle;
7          if (x > a[middle])
8              left = middle;
9          else
10             right = middle;
11     }
12     return -1;
13 }
14

```



(1) 分析错误原因：

这段代码的错误在于 **边界更新逻辑错误**，具体分析如下：

1. left 和 right 更新不当：

- 当 $x > a[middle]$ 时，说明目标值在右半区，此时应将 `left` 设为 `middle + 1`（而非 `middle`），因为 `middle` 位置已确认不是目标值，需排除。
- 当 $x < a[middle]$ 时，目标值在左半区，应将 `right` 设为 `middle - 1`（而非 `middle`），同样是为了排除已检查的 `middle` 位置。

2. 后果：

若按代码中 `left = middle` 或 `right = middle`，会导致搜索区间无法有效缩小，可能陷入死循环。

运行结果如下，显示超时：

超過時間限制 #stdin #stdout 5s 5284KB

[comments](#)

 stdin

[c](#)

Standard input is empty

 stdout

[c](#)

Standard output is empty

(2) 修正方法：

修改更新边界的条件如下：

```

1  if (x > a[middle])
2      left = middle + 1; // 右半区，更新左边界
3  else
4      right = middle - 1; // 左半区，更新右边界

```

2、错误代码二：

将每个数逐一查找验证，再查找一个不在数组中的元素

```
1  #include <iostream>
2  using namespace std;
3
4  template<class Type>
5  int BinarySearch(Type a[],const Type& x,int n){
6      int left=0;
7      int right=n-1;
8      while(left<right-1){
9          int middle = (left+right)/2;
10         if (x<a[middle]){
11             right=middle;
12         }
13         else
14             left=middle;
15     }
16     if(x==a[left])
17         return left;
18     else
19         return -1;
20 }
21
22 int main() {
23     int arr[] = {1, 3, 5, 7, 9};
24     int size = sizeof(arr) / sizeof(arr[0]);
25
26     // 逐一验证数组内元素
27     for (int num : arr) {
28         int result = BinarySearch(arr, num, size);
29         if (result != -1)
30             cout << "查找元素 " << num << ", 索引: " << result << endl;
31         else
32             cout << "元素 " << num << " 不在数组中 (理论不应出现) " << endl;
33     }
34
35     // 查找不在数组中的元素
36     int targetNotInArr = 4;
37     int result = BinarySearch(arr, targetNotInArr, size);
38     if (result != -1)
39         cout << "查找元素 " << targetNotInArr << ", 索引: " << result << endl;
40     else
41         cout << "元素 " << targetNotInArr << " 不在数组中" << endl;
42
43     return 0;
44 }
```

运行结果如下，查找边界元素9时出现了错误。

查找元素 1, 索引: 0
 查找元素 3, 索引: 1
 查找元素 5, 索引: 2
 查找元素 7, 索引: 3
 元素 9 不在数组中 (理论不应出现)
 元素 4 不在数组中

(1) 分析错误原因:

```

5  int BinarySearch(Type a[],const Type& x,int n){
6      int left=0;
7      int right=n-1;
8      while(left<right-1){ 1
9          int middle = (left+right)/2;
10         if (x<a[middle]){
11             right=middle; 2
12         }
13         else
14             left=middle; 3
15     }
16     if(x==a[left])
17         return left; 4
18     else
19         return -1;
20 }
    
```

• 循环条件错误

`while(left < right - 1)` 的条件限制过严, 会导致循环提前终止。 , `left < right - 1` 会跳过许多情况, 例如当目标元素恰好在 `right` 位置时, 循环可能提前结束, 无法完成查找。

• 边界更新逻辑错误

当前代码中 `right = middle` 和 `left = middle` 会导致搜索区间无法有效缩小, 可能遗漏目标元素或陷入死循环。

• 结果判断不全面

循环结束后, 仅判断 `x == a[left]`, 忽略了目标元素可能在 `right` 位置的情况。例如, 当数组只剩两个元素时, 循环结束后还需检查 `right` 位置是否为目标元素。

以数组 {1, 3, 5, 7, 9} 查找元素 9 为例:

初始 `left=0`, `right=4`, 进入循环计算 `middle=2` (值为 5), 因 `9 > 5`, 执行 `left = middle = 2`。下一轮循环, `left=2`, `right=4`, 仍满足 `left < right - 1` (`2 < 4-1=3`), 计算 `middle=3` (值为 7), 因 `9 > 7`, 执行 `left = middle = 3`。循环结束后, 仅检查 `a[left]` (即 `a[3] = 7`), 错过真正的目标元素 9 (位于 `right=4`), 最终错误返回 -1。

(2) 修正方法

修改为和原始正确代码一致, 如下:

```
1 int BinarySearch (Type a[], const Type& x, int n) {
2     int left = 0, right = n - 1;
3     while (left <= right) {
4         int middle = (left + right) / 2;
5         if (x == a[middle])
6             return middle;
7         if (x > a[middle])
8             left = middle + 1;
9         else
10            right = middle - 1;
11    }
12    return -1;
13 }
```

3、错误代码三：

```
1 template<class Type>
2 int BinarySearch3(Type a[], const Type& x, int n) {
3     int left = 0;
4     int right = n - 1;
5     while (left + 1 != right) { // 循环条件错误
6         int middle = (left + right) / 2;
7         if (x >= a[middle])
8             left = middle;
9         else
10            right = middle;
11    }
12    if (x == a[left]) // 仅检查 left, 遗漏 right
13        return left;
14    else
15        return -1;
16 }
```

(1) 分析错误原因：

- 主要错误是循环条件错误：

`while (left + 1 != right)` 限制过严，导致循环提前终止。例如，当数组只剩两个元素（`left` 和 `right` 直接相邻），循环会跳过关键判断，无法覆盖所有可能的搜索区间。其他错误与前面一致。

总结与体会

一、学习总结

- 系统掌握了二分搜索算法的原理、代码实现及复杂度分析，明确其在有序数组中查找的高效性，对模板编程的应用也有了更深刻的理解。
- 通过分析错误代码，深入理解了二分搜索中循环条件、边界更新逻辑的关键作用，提升了对算法细节的把控能力。

二、实验体会

好的，用户需要补充实验报告的“总结与体会”部分的最后两点。首先，分析学习总结和实验体会的常见内容。学习总结可以包括对算法的深入理解、代码实现的掌握。实验体会则侧重实践中的收获，比如调试错误的经验，对二分搜索应用的认识。

现在组织内容：

学习总结：

- 深入理解二分搜索原理，掌握代码实现，能分析复杂度。
- 学会排查错误代码，理解边界条件和逻辑的重要性。

实验体会：

- 调试中认识到细节的关键，如循环条件、边界更新，提升调试能力。
- 明白二分搜索在有序数据中的高效性，拓展算法应用思维。

现在整理成规范的内容：

总结与体会

一、学习总结

- 通过本次实验，深入理解了二分搜索算法的核心思想，不仅能熟练复现代码，还能清晰推导其时间复杂度（ $O(\log n)$ ）和空间复杂度（ $O(1)$ ），对算法的高效性有了更直观的认识。
- 在分析错误代码的过程中，掌握了二分搜索中边界条件、循环逻辑的关键作用，学会从循环条件、搜索区间更新等角度排查问题，强化了对算法细节的理解。

二、实验体会

- 实践中深刻体会到代码细节的重要性，如循环条件设置、边界更新逻辑稍有错误，就会导致算法失效。通过调试错误代码，提升了逻辑分析和问题排查能力。
- 认识到二分搜索在有序数据场景下的高效性，也意识到算法理论与实践结合的必要性。未来在解决类似问题时，会更注重算法前提条件（如数据有序性）和代码实现细节，培养严谨的编程思维。

附正确完整代码：

```
1  #include <iostream>
2  using namespace std;
3
4  template <typename Type>
5  int BinarySearch (Type a[], const Type& x, int n) {
6      int left = 0, right = n - 1;
7      while (left <= right) {
8          int middle = (left + right) / 2;
9          if (x == a[middle])
10             return middle;
11          if (x > a[middle])
12             left = middle + 1;
13          else
14             right = middle - 1;
15      }
16      return -1;
17  }
18
```

```
19 int main() {
20     int arr[] = {1, 3, 5, 7, 9};
21     int size = sizeof(arr) / sizeof(arr[0]);
22
23     // 逐一验证数组内元素
24     for (int num : arr) {
25         int result = BinarySearch(arr, num, size);
26         if (result != -1)
27             cout << "查找元素 " << num << ", 索引: " << result << endl;
28         else
29             cout << "元素 " << num << " 不在数组中 (理论不应出现) " << endl;
30     }
31
32     // 查找不在数组中的元素
33     int targetNotInArr = 4;
34     int result = BinarySearch(arr, targetNotInArr, size);
35     if (result != -1)
36         cout << "查找元素 " << targetNotInArr << ", 索引: " << result << endl;
37     else
38         cout << "元素 " << targetNotInArr << " 不在数组中" << endl;
39
40     return 0;
41 }
```

运行结果：

🔧 stdout

 [copy](#)

查找元素 1, 索引: 0
查找元素 3, 索引: 1
查找元素 5, 索引: 2
查找元素 7, 索引: 3
查找元素 9, 索引: 4
元素 4 不在数组中