 2017年12月23日*16:12:34*[6](https://cuijiahua.com/blog/2017/12/algorithm_2.html#comments) 5,095 °C



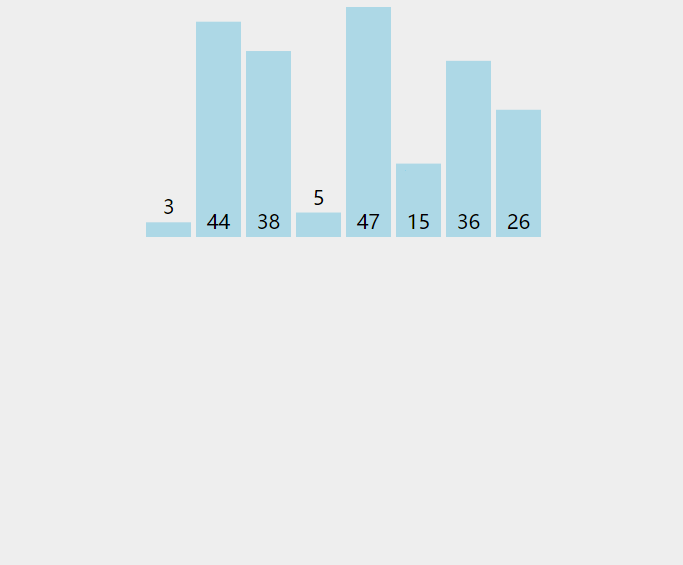
**一、前言**

直接插入排序（Insertion Sort）序是一种最简单的**插入排序**。为简化问题，我们下面只讨论升序排序。

**二、算法思想**

**插入排序**：每一趟将一个待排序的记录，按照其关键字的大小插入到有序队列的合适位置里，知道全部插入完成。

**动态效果示意图：**

[](https://cuijiahua.com/wp-content/uploads/2017/12/algorithm_2_1.gif)

以上的过程，其实就是典型的**直接插入排序，每次将一个新数据插入到有序队列中的合适位置里。**

很简单吧，接下来，我们要将这个算法转化为编程语言。

假设有一组无序序列 R0, R1, ... , RN-1。

(1) 我们先将这个序列中下标为 0 的元素视为元素个数为 1 的有序序列。

(2) 然后，我们要依次把 R1, R2, ... , RN-1 插入到这个有序序列中。所以，我们需要一个**外部循环**，从下标 1 扫描到 N-1 。

(3) 接下来描述插入过程。假设这是要将 Ri 插入到前面有序的序列中。由前面所述，我们可知，插入Ri时，前 i-1 个数肯定已经是有序了。

所以我们需要将Ri 和R0 ~ Ri-1 进行比较，确定要插入的合适位置。这就需要一个**内部循环**，我们一般是从后往前比较，即从下标 i-1 开始向 0 进行扫描。

**1、代码**

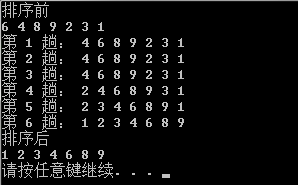
C++：

C++



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42 | #include <iostream>  #include <vector>    using namespace std;    vector<int> insertSort(vector<int> list){      vector<int> result;      if (list.empty()){          return result;      }      result = list;      // 第1个数肯定是有序的，从第2个数开始遍历，依次插入有序序列      for (int i = 1; i < result.size(); i++){          // 取出第i个数，和前i-1个数比较后，插入合适位置          int temp = result[i];          // 因为前i-1个数都是从小到大的有序序列，所以只要当前比较的数(list[j])比temp大，就把这个数后移一位          int j = i - 1;          for (j; j >= 0 && result[j] > temp; j--){              result[j + 1] = result[j];          }          result[j + 1] = temp;      }      return result;  }    void main(){      int arr[] = { 6, 4, 8, 9, 2, 3, 1 };      vector<int> test(arr, arr + sizeof(arr) / sizeof(arr[0]));      cout << "排序前" << endl;      for (int i = 0; i < test.size(); i++){          cout << test[i] << " ";      }      cout << endl;      vector<int> result;      result = insertSort(test);      cout << "排序后" << endl;      for (int i = 0; i < result.size(); i++){          cout << result[i] << " ";      }      cout << endl;      system("pause");  } |

运行结果：

[](https://cuijiahua.com/wp-content/uploads/2017/12/algorithm_2_2.png)

Python：

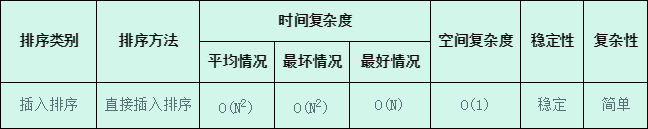
Python



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | # -\*- coding:utf-8 -\*-    def insertSort(input\_list):      if len(input\_list) == 0:          return []      sorted\_list = input\_list        for i in range(1, len(sorted\_list)):          temp = sorted\_list[i]          j = i - 1          while j >=0 and temp < sorted\_list[j]:              sorted\_list[j + 1] = sorted\_list[j]              j -= 1          sorted\_list[j + 1] = temp      return sorted\_list    if \_\_name\_\_ == '\_\_main\_\_':      input\_list = [6, 4, 8, 9, 2, 3, 1]      print('排序前:', input\_list)      sorted\_list = insertSort(input\_list)      print('排序后:', sorted\_list) |

**三、算法分析**

**1、直接插入排序的算法性能**

[](https://cuijiahua.com/wp-content/uploads/2017/12/algorithm_2_3_modify.png)

**2、时间复杂度**

当数据**正序**时，执行效率**最好**，每次插入都不用移动前面的元素，时间复杂度为**O(N)**。

当数据**反序**时，执行效率**最差**，每次插入都要前面的元素后移，时间复杂度为**O(N^2)**。

所以，**数据越接近正序，直接插入排序的算法性能越好**。

**3、空间复杂度**

由直接插入排序算法可知，我们在排序过程中，需要一个临时变量存储要插入的值，所以空间复杂度为 1 。

**4、算法稳定性**

直接插入排序的过程中，不需要改变相等数值元素的位置，所以它是**稳定的**算法。

**四、优化**

因为在一个有序序列中查找一个插入位置，以保证有序序列的序列不变，所以可以使用二分查找，减少元素比较次数提高效率。

二分查找是对于有序数组而言的，假设如果数组是升序排序的。那么，二分查找算法就是不断对数组进行对半分割，每次拿中间元素和目标数字进行比较，如果中间元素小于目标数字，则说明目标数字应该在右侧被分割的数组中，如果中间元素大于目标数字，则说明目标数字应该在左侧被分割的数组中。

**1、代码**

C++：

C++



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62 | #include <iostream>  #include <vector>    using namespace std;    // 给定一个有序的数组，查找第一个大于等于value的下标，不存在返回-1  int BinarySearch(vector<int> list, int n, int value){      int left = 0;      int right = n - 1;        while (left <= right){          int middle = left + ((right - left) >> 1);            if (list[middle] >= value){              right = middle - 1;          }          else{              left = middle + 1;          }      }      return (left < n) ? left : -1;  }    vector<int> BinaryInsertSort(vector<int> list){      vector<int> result = list;      for (int i = 1; i < result.size(); i++){          int insert\_index = BinarySearch(result, i, result[i]);          if (insert\_index != -1){              int temp = result[i];              int j = i - 1;              while (j >= insert\_index){                  result[j + 1] = result[j];                  j--;              }              result[j + 1] = temp;          }          printf("第 %d 趟： ", i);          for (int i = 0; i < result.size(); i++){              cout << result[i] << " ";          }          cout << endl;      }      return result;  }    void main(){      int arr[] = { 6, 4, 8, 9, 2, 3, 1 };      vector<int> test(arr, arr + sizeof(arr) / sizeof(arr[0]));      cout << "排序前" << endl;      for (int i = 0; i < test.size(); i++){          cout << test[i] << " ";      }      cout << endl;      vector<int> result;      result = BinaryInsertSort(test);      cout << "排序后" << endl;      for (int i = 0; i < result.size(); i++){          cout << result[i] << " ";      }      cout << endl;      system("pause");  } |

运行结果没有改变，只是在查找插入位置的次数减少了，提高了算法的效率。

Python：

Python



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | # -\*- coding:utf-8 -\*-    def BinarySearch(input\_list, end, value):      left = 0      right = end - 1      while left <= right:          middle = left + (right - left) // 2          if input\_list[middle] >= value:              right = middle - 1          else:              left = middle + 1        return left if left < end else -1    def BinaryInsertSort(input\_list):      if len(input\_list) == 0:          return []      result = input\_list      for i in range(1, len(input\_list)):          j = i - 1          temp = result[i]          insert\_index = BinarySearch(result, i, result[i])          if insert\_index != -1:              while j >= insert\_index:                  result[j + 1] = result[j]                  j -= 1              result[j + 1] = temp      return result      if \_\_name\_\_ == '\_\_main\_\_':      input\_list = [6, 4, 8, 9, 2, 3, 1]      print('排序前:', input\_list)      sorted\_list = insertSort(input\_list)      print('排序后:', sorted\_list) |