HW6实验报告

2252707 陈艺天

2024年3月1日

1. 排序

1.1 问题描述

使用不同排序算法进行测试, 总结各种算法特点.

1.2 问题分析与解决思路

使用自动化测试.

1.3 数据结构设计

本次所有排序均为基于向量的排序.

1.4 功能函数设计

自动化测试如下

```cpp
constexpr int SEED = 29;

static void generate_sequence(std::vector<int> &t, const int n,
                              const bool reverse = false,
                              const bool sorted = false) {
  t.resize(n);
  if (sorted) {
    for (int i = 0; i < n; i++)
      t[i] = i;
  } else if (!reverse)
    for (int i = 0; i < n; i++)
      t[i] = rand();
  else
    for (int i = 0; i < n; i++)
      t[i] = n - i - 1;
}

class SortTest {
  using func_type = std::function<void(int *, int, int)>;
  func_type sort_func;

public:
  SortTest(func_type sort) : sort_func(sort) {}
  double runtest(const int n, const bool reverse = false,
                 const bool sorted = false) {
    srand(SEED);
    std::vector<int> test;
    generate_sequence(test, n, reverse, sorted);
    using namespace std::chrono;
    const auto begin = high_resolution_clock::now();
    sort_func(test.data(), 0, n);
    const auto end = high_resolution_clock::now();
    auto duration = duration_cast<microseconds>(end - begin);
    double res = 0;
    std::cout << "Size : " << n
              << " Time : " << (res = duration.count() / 1000.0) << " ms"
              << std::endl;
    return res;
  }
};

static void basic_test(std::function<void(int *, int, int)> sort) {
  static auto print = [](const int elem) { std::cout << elem << ' '; };
```

```
  std::vector<int> test;
  srand(SEED);
  constexpr int N = 40;
  test.reserve(N);
  for (int i = 0; i < N; i++) {
    test.push_back(rand() % (3 * N));
  }

  std::for_each(test.begin(), test.end(), print);
  std::cout << '\n';
  sort(test.data(), 0, test.size());
  std::for_each(test.begin(), test.end(), print);
  std::cout << '\n';
}
```

冒泡排序

```
template <typename T>
void bubbleSort(T *const elem, const int low, const int high) {
  if (high - low < 2)
    return;
  bool sorted = false;
  int n = high;
  while (!sorted) {
    sorted = true;
    for (int i = low; i < n - 1; i++) {
      if (elem[i] > elem[i + 1]) {
        sorted = false;
        std::swap(elem[i], elem[i + 1]);
      }
    }
    n--;
  }
}
```

选择排序

```cpp
template <typename T>
void selectionSort(T *const elem, const int low, const int high) {
  if (high - low < 2)
    return;
  auto find_max = [elem](const int low, const int high) {
    auto max_elem = elem[low];
    int max_pos = 0;
    for (int i = low; i < high; i++) {
      if (elem[i] > max_elem) {
        max_elem = elem[i];
        max_pos = i;
      }
    }
    return max_pos;
  };

  int n = high;
  while (--n > low) {
    std::swap(elem[n], elem[find_max(low, n)]);
  }
}
```

插入排序

```cpp
template <typename T>
void insertionSort(T *const elem, const int low, const int high) {
  if (high - low < 2)
    return;
  for (int i = low + 1; i < high; i++) {
    const auto target = elem[i];
    auto pos = std::upper_bound(elem + low, elem + i, target);
    for (int j = i; j > pos - elem; j--) {
      elem[j] = elem[j - 1];
    }
    *pos = target;
  }
}
```

归并排序

```cpp
template <typename T>
static void merge(T *const elem, const int low, const int mid, const int high) {
  const int la = mid - low, lb = high - mid;
  T *const A = new T[la];
  T *const B = elem + mid;
  for (int i = 0; i < la; i++)
    A[i] = elem[i + low];

  int i = 0, j = 0, tot = low;
  while (i < la && j < lb) {
    if (A[i] <= B[j]) {
      elem[tot++] = A[i++];
    } else {
      elem[tot++] = B[j++];
    }
  }
  while (i < la)
    elem[tot++] = A[i++];
  delete[] A;
}

template <typename T>
void mergeSort(T *const elem, const int low, const int high) {
  if (high - low < 2)
    return;
  const auto mid = (high + low) / 2;
  mergeSort(elem, low, mid);
  mergeSort(elem, mid, high);
  merge(elem, low, mid, high);
}
```

堆排序-排序

```cpp
template <typename T>
void heapSort(T *const elem, const int low, const int high) {
  using namespace Heap;
  T *const A = elem + low;
  rank size = high - low;
  HeapBuild<T>::heapify(A, size);
  while (--size) {
    std::swap(A[0], A[size]);
    HeapBuild<T>::percolateDown(elem, 0, size);
  }
}
```

堆排序-建堆

```cpp
//下滤算法
template <typename T>
rank HeapBuild<T>::percolateDown(T* const elem, const rank start,
                                 const rank n) {
  int i = start, j = -1;
  const auto target = elem[start];
  while (i != (j = maxIn3(elem, i, target, n))) {
    elem[i] = elem[j];
    i = j;
  }
  elem[j] = target;
  return j;
}
//Floyd建堆法
template <typename T>
void HeapBuild<T>::heapify(T* const elem, const rank n) {
  for (int i = Parent(n - 1); InHeap(i, n); i--) {
    percolateDown(elem, i, n);
  }
}
```

快速排序

```cpp
//分割算法
template <typename T> static int partition(T *const elem, int low, int high) {
  std::swap(elem[low], elem[low + rand() % (high - low)]);
  const auto pivot = elem[low];
  while (low < high) {
    do
      high--;
    while (low < high && pivot <= elem[high]);
    if (low < high)
      elem[low] = elem[high];
    do
      low++;
    while (low < high && elem[low] < pivot);
    if (low < high)
      elem[high] = elem[low];
  }
  elem[high] = pivot;
  return high;
}

//快速排序 递归版
template <typename T>
void quickSort(T *const elem, const int low, const int high) {
  if (high - low < 2)
    return;
  const auto mid = partition(elem, low, high);
  quickSort(elem, low, mid);
  quickSort(elem, mid + 1, high);
}
```

希尔排序

```
template <typename T>
void shellSort(T *const elem, const int low, const int high) {
  if (high - low < 2)
    return;
// 使用PS序列
  for (int d = INT_MAX; d > 0; d >>= 1) {
    // 步长为 d，矩阵宽度为d
    for (int j = low + d; j < high; j++) {
      const auto x = elem[j];
      int i{j};
      while (i - d >= low && elem[i - d] > x) {
        elem[i] = elem[i - d];
        i -= d;
      }
      elem[i] = x;
    }
  }
}
```

1.5 调试与分析

无且略.

1.6 总结

assert: 以下时间单位为ms

	10	100	1000	10000	100000	1000000	10k 逆序	10k正序
堆排序	0.001	0.008	0.142	1.992	19.658	227.235	1.107	1.155
归并排序	0.001	0.009	0.129	1.149	14.563	147.915	0.589	0.49
快速排序	0	0.005	0.071	0.714	9.343	100.241	0.384	0.348
希尔排序	0	0.005	0.088	1.144	18.911	229.019	0.28	0.178
插入排序	0	0.017	0.276	22.737	2180.79	223089	44.481	0.427
冒泡排序	0	0.006	0.364	32.323	3084.22	317779	42.283	54.273
选择排序	0	0.02	1.461	159.873	20732.3	>223089	164.606	0.013

复杂度分析

排序名称	时间复杂度	空间复杂度
堆排序	$\Theta(n \log n)$	原地
归并排序	$\Theta(n \log n)$	$O(n)$
快速排序	$O(n^2)$期望$O(n \log n)$	栈空间$O(\log n)$
希尔排序	$O(n^2)$输入敏感最优$O(n)$	原地
插入排序	$O(n^2)$输入敏感最优$O(kn)$, k为逆序对最大间距	原地
冒泡排序	$O(n^2)$	原地
选择排序	$O(n^2)$	原地

稳定性

排序名称	稳定性
堆排序	不稳定
归并排序	稳定
快速排序	不稳定
希尔排序	不稳定
插入排序	稳定
冒泡排序	稳定
选择排序	不稳定

2. 逆序对

2.1 问题描述

请求出整数序列A的所有逆序对个数

2.2 问题分析与解决思路

如果直接求逆序对,Brute-Force算法需要$O(n^2)$复杂度 .不足以通过本题. 因此需要更优秀的算法. 可以借助归并排序, 在排序中统计逆序对, 可以实现$\Theta(n \log n)$的复杂度.

2.3 数据结构设计

基于向量的归并排序.

2.4 功能函数设计

归并函数:

```
int mergeSortCount(const int low, const int high) {
  if (high - low < 2) return 0;
  int res = 0;
  const int mid = (low + high) / 2;
  res += mergeSortCount(low, mid);//分别计算
  res += mergeSortCount(mid, high);//分别计算

  res += mergeCount(low, mid, high);//向量归并
  return res; //返回逆序对
}
```

归并排序函数:

```
int mergeCount(const int low, const int mid, const int high) {
    int res = 0;
    int i = 0, j = mid, tot = low;
    const int la = mid - low;
    //这部分采用A作为公共缓冲区,避免多次的动态内存分配,提高效率
    if (A.size() < la) A.resize(la);
    for (int t = 0; t < la; t++) A[t] = elem[t + low];

    // 归并
    while (i < la && j < high) {
      if (A[i] <= elem[j]) {
        //如果满足左半部分小，那么不对逆序对有贡献
        elem[tot++] = A[i++];
      } else {
        elem[tot++] = elem[j++];
        //如果左半部分大
        //那么从这个元素开始后面的元素都会比右半段大
        //我们把逆序对的帐记到后者.
        res += mid - (i + low);
      }
    }
    //拷贝剩余部分元素
    while (i < la) {
      elem[tot++] = A[i++];
    }

    return res;
  }
```

2.5 调试与分析

```
res += mid - (i + low);//这一步的贡献数目容易出错，因为从low开始计算
还是从0开始计算,需要注意边界条件.
```

2.6 总结

本题利用了分治的思想, 将原问题$T(n)$转化为子问题$2T(\frac{n}{2})$. 实现了复杂度的降低. 但同时缺陷在于空间复杂度提高为$\Theta(n)$.

3. 最大数

3.1 问题描述

给定一组非负整数 nums，重新排列每个数的顺序（每个数不可拆分）使之组成一个最大的整数.

3.2 问题分析与解决思路

本题本质就是一个排序问题, 如何确定每个数的序. 进一步地说, 也就是给定a, b判断$a > b$的条件是什么? 经过分析可以发现只要保证$\{ab\} > \{ba\}$即可.

3.3 数据结构设计

基于向量的快速排序

3.4 功能函数设计

```cpp
std::string largestNumber(std::vector<int>& nums) {
  // 这里填写你的代码
  std::string res;
  res.reserve(nums.size() * 4);
  //这里cmp函数直接借助string字典序的性质. 由于 s1 + s2 与 s2 + s1是
  //等长的, 因此字典序也就是数字大小的顺序
  auto cmp = [](const int a, const int b) {
    const auto s1 = std::to_string(a), s2 = std::to_string(b);
    if (s1 + s2 > s2 + s1) return true;
    return false;
  };
  //排序
  std::sort(nums.begin(), nums.end(), cmp);
  //加入答案
  for (auto iter : nums) {
    res.append(std::to_string(iter));
  }
  return res;
}
```

3.5 调试与分析

过于简单, 无需调试.

3.6 总结

关键在于提取出关于不同数字的全序关系, 划归为排序问题.

4. 三数之和

4.1 问题描述

给你一个整数数组 nums ，判断是否存在三元组 [nums[i], nums[j], nums[k]] 满足 i != j、i != k 且 j != k ，同时还满足 nums[i] + nums[j] + nums[k] == 0 。请你返回所有和为 0 且不重复的三元组，每个三元组占一行。

4.2 问题分析与解决思路

BF算法是不可取的, 直接达到$O(n^3)$的复杂度; 借鉴两数之和的哈希表方法也不可取, 因为无法进行不重复筛查.重复的源头就是, 选定元素a, b, c, 首先选取a, 那么在从小到大选取的过程中, a可能相同, 因此首先做的第一步就是排除第一个数相同的. 同时再利用双指针确定b, c, 确定之后排除相同的. 可以全局去重.

4.3 数据结构设计

基于向量的排序

4.4 功能函数设计

```
void threeSum() {
    std::sort(A.begin(), A.end());

    for (int n1 = 0; n1 < A.size() - 2; n1++) {  // n1最大是size()-3,
    // 进行a的去重
      if (n1 > 0 && A[n1] == A[n1 - 1]) continue;

      int n2 = n1 + 1, n3 = A.size() - 1;

      //转变为两数之和问题
      const int target = -A[n1];
      while (n2 < n3) {
        const auto sum = A[n2] + A[n3];
        //根据<target, >target进行分类
        if (sum < target) {
          n2++;
        } else if (sum > target) {
          n3--;
        } else {
          // assert 此时相等输出答案并且剔除等号
          std::cout << A[n1] << ' ' << A[n2++] << ' ' << A[n3--] << '\n';

          //剔除等号
          while (n2 < A.size() && A[n2] == A[n2 - 1]) {
            n2++;
          }
          //剔除等号
          while (n3 >= 0 && A[n3] == A[n3 + 1]) {
            n3--;
          }
        }
      }
    }
  }
```

4.5 调试与分析

两个边界条件, 剔除等号的循环什么时候停止. 首先是n2,n3的合法性,其次是与上一个相等, 因此采用while循环而不是do while.

```
//剔除等号
while (n2 < A.size() && A[n2] == A[n2 - 1]) {
n2++;
}
//剔除等号
while (n3 >= 0 && A[n3] == A[n3 + 1]) {
n3--;
}
```

4.6 总结

通过排序+双指针在复杂度中剔除掉一个$O(n)$, 从两端分别扫描, 实现了$O(n^2)$的时间复杂度. 因此本题需要提取出不变性, 也就是序列的单调性, 由此就可以通过双指针进行优化.