数据结构——排序

前言

本次报告包含：

- 测试流程；
- 排序算法的思考与比较；
- 个人心得；
- 源码；

测试流程

Intro.

对于本次数据结构report，我实现了以下排序：

- **insert sort**
- **binary insert sort**
- **shell sort**
- **bubble sort**
- **quick sort**(non-iterater version)
- **select sort**
- **heap sort**
- **merge sort**

> 其中除了`merge sort`我使用了递归，其他均为迭代版本。除了以上排序，我还将其结果和std::sort进行了比较，最后写出了一份测试代码。输出结果（删去了排好后的序列的显示）如下：

```
  ==>>  <时间校准>
程序设计了[1000.00] ms等待，实际等待
  ==>>  [1001.39] ms

[INSERT SORT]:
for 10000 elemets:
>>> time cost: 10.170700ms

[BI_INSERT SORT]:
for 10000 elemets:
```

```
>>> time cost: 2.010200ms

[SHELL SORT]:
for 10000 elemets:
>>> time cost: 0.791100ms

[BUBBLE SORT]:
for 10000 elemets:
>>> time cost: 119.200300ms

[SELECT SORT]:
for 10000 elemets:
>>> time cost: 22.280500ms

[QUICK SORT]:
for 10000 elemets:
>>> time cost: 0.562500ms

[HEAP SORT]:
for 10000 elemets:
>>> time cost: 0.699400ms

[MERGE SORT]:
for 10000 elemets:
>>> time cost: 4.959400ms

[STD SORT]:
for 10000 elemets:
>>> time cost: 0.442800ms


Process finished with exit code 0
```

测试设计的细节

在实现算法后，为了对算法进行测试，我考虑了以下几点：

1. 增加单次测试的loop

易知，对于一个程序$\mathcal{P}$，其耗时为$T(\mathcal{P})$。但在测试的时候，由于程序的启动部分（比如说常数级的函数调用，时钟的设定，比如看我代码中的时间校准部分就可以发现有大概0.1%的时间误差）会让我们最终得到的程序的时间$T(\mathcal{F})$加上一些常数时间b即：

$$T(\mathcal{F}) = T(\mathcal{P}) + b$$

我们想得到的是$T(\mathcal{P})$，但最终得到的是$T(\mathcal{F})$，如何让两者尽量相等呢？这里我们可以通过增加内部循环次数来进行计算：

$$T(\mathcal{F}) = \lim_{N \to \infty} \frac{T(\mathcal{P}) * N + b}{N} = T(\mathcal{P})$$

在我的程序中，限于自己的电脑的算力，我将N(对应程序中的LOOP_SIZE)设置为100。

2. 使用`chrono::steady_clock`

steady_clock的优点：

- 时间精度高；
- 稳；

排序算法的思考与比较

ranking

就“测试流程”处展示的测试结果而言，对我使用的算法进行时间升序排名可得：

- `std::sort`
- `quick_sort`
- `heap_sort`
- `shell_sort`
- `binary_insert_sort`
- `merge_sort`
- `insert_sort`
- `select_sort`
- `bubble_sort`

复杂度分析

> 下面讨论的都是大O标记的最坏情况的复杂度。

- `std::sort`

  > O(N logN)

- `quick_sort`

  > O(N logN)

- `heap_sort`

  > O(N logN)

- `shell_sort`

  > It depends on how we define the gap.
  >
  > 而本次代码中我使用的序列来自：
  >
  > > Pratt, V. Shellsort and sorting networks (Outstanding dissertations in the computer sciences). Garland. 1979. ISBN 0-824-04406-1. (This was originally presented as the author's Ph.D. thesis, Stanford University, 1971)

步长序列	最坏情况下复杂度
$2^i 3^j$	$\mathcal{O}(n \log^2 n)$
$2^k - 1$	$\mathcal{O}(n^{3/2})$
$n/2^i$	$\mathcal{O}(n^2)$

> source from wiki.

- `binary_insert_sort`

  > O(N)

- `merge_sort`

  > O(N logN)

- `insert_sort`

  > O(N)

- `select_sort`

  > O(N)

- `bubble_sort`

  > O(N)

一些关于快与慢的原因

std::sort

作为state-of-art的`std::sort`，首先对在数据规模大的时候，使用了快排，而且对于快排中的`partition`（即每次分界的那个数），快排使用了精心设计的随机数，随机设置partition的位置。这就比一般人只取头尾要简单。当快排得差不多了（某种程度的整体有序），这个时候就换插入排序了，对于基本有序的序列，插排的效率会很高。而且插排本身就存在一个常数小的优势，如果依然对几乎有序的序列使用快排，反而更加容易导致排序效率低下。除此之外，因为`std::sort`是编译器方的专业人员书写的，从代码本身到编译器对该算法的优化，都是为了加快整个排序流程，这也是为什么`std::sort`如此之快且稳定的原因。

merge sort

归并排序的复杂度低，但是常数较大，每次都存在大量的数组赋值操作，所以比较慢。

bubble sort

不管是比较次数还是位置切换次数都稳定在n^2…所以能不慢吗...

心得体会

排序算法要考虑2个因素：

- 比较次数；
- 位置变动次数；

从算法设计层面优化代码有下面这些points：

- 分治；
- 排序算法的组合；

从编程角度：

- 对于连续迁移的元素，不要一个一个swap（3次操作），而是全都往后推一位，再把多的一位放前头（1次操作）。（详情请见我写的`insert_sort`）
- 尽量用迭代的方法来书写，便于代码的优化，减少函数调用的时间。

源码

```
#include <iostream>

#include <array>
#include <stack>
#include <vector>

#include <limits>
#include <random>
#include <algorithm>
#include <thread>

#include <iomanip>
#include <chrono>

/* Written by ganler. Use this code to feel the pipelines of diverse sorts.
*/

// 分别测试：插入排序√，折半插入排序√，希尔排序√，冒泡排序√，快速排序√，选择排序√，归并排序√，堆排序√
// 以及最强排序 STL 排序
// 随机数范围[0, 1000 000]
// 随机数个数10 000个
```

```cpp
// 测试次数50次（减少常数的影响）

constexpr uint32_t TEST_SIZE = 10000;
constexpr uint32_t LOOP_SIZE = 10;
constexpr uint32_t PRECISION = 6;
constexpr uint32_t SHELL_FACTOR = 3; // For guys who want use a gap
function with $n/SHELL_FACTOR^i$

using namespace std;

void time_test()
{
    auto start = chrono::steady_clock::now();
    this_thread::sleep_for(std::chrono::milliseconds(1000));
    auto end = chrono::steady_clock::now();

    auto diff = end-start;

    cout << "  ==>>  <时间校准>" << endl;
    cout << "程序设计了[1000.00] ms等待，实际等待" << endl << "  ==>>  [" <<
chrono::duration<double, milli>(diff).count() << "] ms" << endl;
}

void print(const array<int, TEST_SIZE>& arr)
{
    for(const auto& x : arr)
        cout << x << ' ';
    cout << endl;
}

void insert_sort(const array<int, TEST_SIZE>& data_)
{
    auto start = chrono::steady_clock::now();
    auto data = data_;

    for (int i = 0; i < LOOP_SIZE; ++i)
    {
        data = data_;
        for (int j = 1; j < TEST_SIZE; ++j)
        {
            auto tmp = data[j];
            int k = j;
            for (; k > 0 && data[k - 1] > tmp; --k) // Must be "data[k-1] >
tmp" not "data[k-1] > data[k]"
                data[k] = data[k - 1];              // "data[k-1] >
data[k]" is used in constant swap operation.
            data[k] = tmp;
        }
    }

    auto end = chrono::steady_clock::now();
    auto diff = (end-start);
```

```
    cout << "[INSERT SORT]: " << endl << "for " << TEST_SIZE << " elemets:"
<< endl;
    cout << ">>>\t";
    print(data);
    cout << ">>> time cost: " << fixed << setprecision(PRECISION)
         << chrono::duration<double, milli>(diff).count()/LOOP_SIZE << "ms"
<< endl << endl;
}


void bi_insert_sort(const array<int, TEST_SIZE>& data_)
{
    auto start = chrono::steady_clock::now();

    auto data = data_;

    for (int i = 0; i < LOOP_SIZE; ++i)
    {// 用swap会慢一点（std::swap有2次操作（手写swap有3次）），直接往后拉就只有一次。
        data = data_;
        for (int j = 1; j < TEST_SIZE; ++j)
        {
            auto tmp = data[j];
            auto beg = 0;
            auto end = j-1;

            while(beg <= end)
            {
                auto mid = (beg+end)/2;
                if(data[mid] == tmp)
                {
                    beg = mid;
                    break;
                }
                else if(data[mid] < tmp)
                    beg = mid+1;
                else
                    end = mid-1;
            }

            for (int k = j; k > beg; --k)
                data[k] = data[k-1];

            data[beg] = tmp;
        }
    }

    auto end = chrono::steady_clock::now();
    auto diff = (end-start);

    cout << "[BI_INSERT SORT]: " << endl << "for " << TEST_SIZE << "
elemets:" << endl;
```

```
    cout << ">>>\t";
    print(data);
    cout << ">>> time cost: " << fixed << setprecision(PRECISION)
         << chrono::duration<double, milli>(diff).count()/LOOP_SIZE << "ms"
<< endl << endl;
}

void shell_sort(const array<int, TEST_SIZE>& data_)
{
    auto start = chrono::steady_clock::now();

    auto data = data_;

    for (int k = 0; k < LOOP_SIZE; ++k)
    {
        data = data_;
        /* This part is related to the parameter SHELL_FACTOR */
//        for (int gap = TEST_SIZE / SHELL_FACTOR; gap > 0; gap /=
SHELL_FACTOR)
//            for (int i = gap; i < TEST_SIZE; ++i)
//                for (int j = i; j - gap >= 0 && data[j - gap] > data[j];
j -= gap)// Must be j>=gap. Especially "=".
//                    swap(data[j], data[j - gap]);
        int step[] = { 1, 5, 19, 41, 109, 209, 505, 929, 2161, 3905 };
        for (int k = 9; k >= 0; k--)
        {
            auto gap = step[k];
            for (int i = gap; i < TEST_SIZE; ++i)
                for (int j = i; j - gap >= 0 && data[j - gap] > data[j]; j
-= gap)// Must be j>=gap. Especially "=".
                    swap(data[j], data[j - gap]);
        }
    }

    auto end = chrono::steady_clock::now();
    auto diff = (end-start);

    cout << "[SHELL SORT]: " << endl << "for " << TEST_SIZE << " elemets:"
<< endl;
    cout << ">>>\t";
    print(data);
    cout << ">>> time cost: " << fixed << setprecision(PRECISION)
         << chrono::duration<double, milli>(diff).count()/LOOP_SIZE << "ms"
<< endl << endl;
}

void bubble_sort(const array<int, TEST_SIZE>& data_)
{
    auto start = chrono::steady_clock::now();

    auto data = data_;
```

```
    for (int k = 0; k < LOOP_SIZE; ++k)
    {
        data = data_;
        for (int i = TEST_SIZE-1; i > 0; --i)
            for (int j = 0; j < i; ++j)
                if (data[j] > data[j + 1])
                    swap(data[j], data[j + 1]);
    }

    auto end = chrono::steady_clock::now();
    auto diff = (end-start);

    cout << "[BUBBLE SORT]: " << endl << "for " << TEST_SIZE << " elemets:"
<< endl;
    cout << ">>>\t";
    print(data);
    cout << ">>> time cost: " << fixed << setprecision(PRECISION)
         << chrono::duration<double, milli>(diff).count()/LOOP_SIZE << "ms"
<< endl << endl;
}


void quick_sort(const array<int, TEST_SIZE>& data_)
{
    auto start = chrono::steady_clock::now();

    auto data = data_;
    random_device rd_;

    for (int i = 0; i < LOOP_SIZE; ++i)
    {
        data = data_;
        stack<pair<uint32_t, uint32_t>> range_stack;
        range_stack.push(pair<uint32_t, uint32_t>(0, TEST_SIZE));

        while (!range_stack.empty())
        {
            auto range = range_stack.top();
            range_stack.pop();

            uint32_t partition = data[range.first];

            uint32_t i = range.first, j;
            for (j = range.first+1; j < range.second; j++)
                if(data[j] <= partition)
                    swap(data[++i], data[j]);

            swap(data[i], data[range.first]);
            if(i > range.first)
                range_stack.push(pair<uint32_t, uint32_t>(range.first, i));
            if(i+1 < range.second)
```

```
                range_stack.push(pair<uint32_t, uint32_t>(i+1,
range.second));
        }
    }

    auto end = chrono::steady_clock::now();
    auto diff = (end-start);

    cout << "[QUICK SORT]: " << endl << "for " << TEST_SIZE << " elemets:"
<< endl;
    cout << ">>>\t";
    print(data);
    cout << ">>> time cost: " << fixed << setprecision(PRECISION)
         << chrono::duration<double, milli>(diff).count()/LOOP_SIZE << "ms"
<< endl << endl;
}


void select_sort(const array<int, TEST_SIZE>& data_)
{
    auto start = chrono::steady_clock::now();

    auto data = data_;

    for (int i = 0; i < LOOP_SIZE; ++i)
    {
        data = data_;
        for (int j = 0; j < TEST_SIZE - 1; ++j)
        {
            auto min_ind = j;
            for (int k = j; k < TEST_SIZE; ++k) {
                if(data[k] < data[min_ind])
                    min_ind = k;
            }
            swap(data[min_ind], data[j]);
        }
    }

    auto end = chrono::steady_clock::now();
    auto diff = (end-start);

    cout << "[SELECT SORT]: " << endl << "for " << TEST_SIZE << " elemets:"
<< endl;
    cout << ">>>\t";
    print(data);
    cout << ">>> time cost: " << fixed << setprecision(PRECISION)
         << chrono::duration<double, milli>(diff).count()/LOOP_SIZE << "ms"
<< endl << endl;
}

void merge(array<int, TEST_SIZE>& arr, uint32_t beg, uint32_t mid, uint32_t
end)
```

```
{
    auto left_vec = vector<int>(arr.begin()+beg, arr.begin()+mid);
    auto right_vec = vector<int>(arr.begin()+mid, arr.begin()+end);

    uint32_t left_ind = 0;
    uint32_t right_ind = 0;

    left_vec.insert(left_vec.end(), numeric_limits<int>::max());
    right_vec.insert(right_vec.end(), numeric_limits<int>::max());

    for(; beg<end; beg++)
        if(left_vec[left_ind] > right_vec[right_ind])
            arr[beg] = right_vec[right_ind++];
        else
            arr[beg] = left_vec[left_ind++];
}

void merge_sort_(array<int, TEST_SIZE>& arr, int beg=0, int end=TEST_SIZE)
{
    if(beg >= end-1)
        return;
    int mid = beg+(end-beg)/2;
    merge_sort_(arr, beg, mid);
    merge_sort_(arr, mid, end);
    merge(arr, beg, mid, end);
}


void merge_sort(const array<int, TEST_SIZE>& data_)
{
    auto start = chrono::steady_clock::now();

    auto data = data_;

    for (int i = 0; i < LOOP_SIZE; ++i) {
        data = data_;
        merge_sort_(data);
    }

    auto end = chrono::steady_clock::now();
    auto diff = (end-start);

    cout << "[MERGE SORT]: " << endl << "for " << TEST_SIZE << " elemets:"
<< endl;
    cout << ">>>\t";
    print(data);
    cout << ">>> time cost: " << fixed << setprecision(PRECISION)
         << chrono::duration<double, milli>(diff).count()/LOOP_SIZE << "ms"
<< endl << endl;
}
```

```
inline void max_heapify(array<int, TEST_SIZE>& arr, uint32_t begin,
uint32_t end)
{
    uint32_t father = begin;
    uint32_t left_son = 2*father+1;

    while(left_son <= end)
    {
        if(left_son+1 <= end && arr[left_son] < arr[left_son+1])
            left_son++;

        if(arr[father] > arr[left_son])
            return;
        else
        {
            swap(arr[father], arr[left_son]);
            father = left_son;
            left_son = 2*father+1;
        }
    }
}

void heap_sort(const array<int, TEST_SIZE>& data_)
{
    auto start = chrono::steady_clock::now();

    auto data = data_;

    if(TEST_SIZE <= 1){ return; }

    for (int k = 0; k < LOOP_SIZE; ++k) {
        data = data_;
        for (int i = TEST_SIZE / 2 - 1; i >= 0; i--)
            max_heapify(data, i, TEST_SIZE - 1);
        for (int i = TEST_SIZE - 1; i > 0; i--)
        {
            swap(data[0], data[i]);
            max_heapify(data, 0, i - 1);
        }
    }

    auto end = chrono::steady_clock::now();
    auto diff = (end-start);

    cout << "[HEAP SORT]: " << endl << "for " << TEST_SIZE << " elemets:"
<< endl;
    cout << ">>>\t";
    print(data);
    cout << ">>> time cost: " << fixed << setprecision(PRECISION)
         << chrono::duration<double, milli>(diff).count()/LOOP_SIZE << "ms"
<< endl << endl;
}
```

```
void std_sort(const array<int, TEST_SIZE>& data_)
{
    auto start = chrono::steady_clock::now();

    auto data = data_;

    for (int i = 0; i < LOOP_SIZE; ++i) {
        data = data_;
        sort(data.begin(), data.end());
    }

    auto end = chrono::steady_clock::now();
    auto diff = (end-start);

    cout << "[STD SORT]: " << endl << "for " << TEST_SIZE << " elemets:" <<
endl;
    cout << ">>>\t";
    print(data);
    cout << ">>> time cost: " << fixed << setprecision(PRECISION)
         << chrono::duration<double, milli>(diff).count()/LOOP_SIZE << "ms"
<< endl << endl;
}

int main(){
    ios_base::sync_with_stdio(false);

    random_device rd;
    default_random_engine e{rd()};
    uniform_int_distribution<int> u{0, 1000000};

    array<int, TEST_SIZE> test_array;

    for (int k = 0; k < TEST_SIZE; ++k)
        test_array[k] = u(e);

    time_test(); // 时间测试

    cout << endl << "--- 原始数组序列 ---" << endl << ">>>\t";
    print(test_array);
    cout << endl;

    insert_sort(test_array);
    bi_insert_sort(test_array);
    shell_sort(test_array);
    bubble_sort(test_array);
    select_sort(test_array);
    quick_sort(test_array);
    heap_sort(test_array);
    merge_sort(test_array);
    std_sort(test_array);
}
```