1 随机数生成

这里，我采用 Python 进行随机数生成（Python version: 3.9. 以生成一个容量为 1000000 的随机数文件为例（随机数范围 0 → 10000000 ），具体代码如下：

解释一下几行 core code:

1. 第三行表示将.py 生成的文件存储在所示地址上，所有随机数生成文件都在/lab1 下
2. 第 7→10 行在生成随机数，范围是 0→ num*10

```
1  import random
2  def random_generate(name, num):
3      desktop_path = '/Users/edith_lzh/Desktop/大三上/Algorithm/lab1'
4      full_path = desktop_path + '/'+name + '.txt'
5      file = open(full_path, 'w+')
6
7      for i in range(num):
8          random_number = random.randint(0, num*10)
9          file.write(str(random_number)+' ')
10     file.close()
11 random_generate('1000000', 1000000)
```

random generate.py

如果需要更改随机数的数量级和范围，只需要更改第 10 行 generate(a,b) 两个参数即可，a 是生成随机数文件名，b 是生成随机数的个数。

2 归并排序

2.1 归并排序的实现

编译环境：使用 UBUNTU 命令行界面编辑，编译器为 g++(Apple clang version 12.0.0 (clang-1200.0.32.27))，编译命令为：g++ ” /sort.cc” -o ” /sort.out” -W -Wall -O2 -std=c++17。硬件环境：Intel i9-9800H

归并排序（Merge sort）是建立在归并操作上的一种排序算法，归并排序对序列的元素进行逐层折半分组，然后从最小分组开始比较排序，合并成一个大的分组，逐层进行，最终所有的元素都是有序的。算法的时间复杂度为 O(nlogn)

具体代码如下：

解释一下几行 core code:

1. Merge() 是合并操作，将两个较小分组进行比较排序，合并成一个大的分组
2. MergeSort() 是归并算法本身，算法本省通过递归调用他自己实现，行参为数组名，左边界，右边界，每次算法先取中间值 $m = \frac{1+r}{2}$ 作为划分，递归调用前一半和后一半，最后将这两部分进行合并。
3. result() 是为了方便用户使用设计，输入行参 n 的范围是（1，6）分别代表运行随机数数据集的大小 (1:10.txt, 6:1000000.txt)

4. main() 面向用户，C++ 的 chrono 库用来测量程序运行时间，下面的代码块第 152 行到第 155 行就是在做这件事。

```
#include <chrono>
#include <fstream>
#include <iostream>
using namespace std;

void Merge(int *a, int p, int q, int r) {
  int n1 = q - p + 1;  //左部分的元素个数
  int n2 = r - q;      //同上
  int i, j, k;
  int *L = new int[n1 + 1];
  int *R = new int[n2 + 1];
  for (i = 0; i < n1; i++) L[i] = a[p + i];
  for (j = 0; j < n2; j++) R[j] = a[q + j + 1];
  L[n1] = 11111111;
  R[n2] = 11111111;
  // 数组L从0~n1-1存放，第n1个存放int型所能表示的最大数，即认为正无穷，这是为了
  //处理合并时，比如当数组L中的n1个元素已经全部按顺序存进数组a中，只剩下数组R的
  //部分元素，这时因为R中剩下的元素全部小于11111111,则执行else语句，直接将剩下的
  //元素拷贝进a中。
  for (i = 0, j = 0, k = p; k <= r; k++) {
    if (L[i] <= R[j])
      a[k] = L[i++];
    else
      a[k] = R[j++];
  }

  delete[] L;
  delete[] R;
}

void MergeSort(int *a, int l, int r) {
  if (l < r) {
    int m = (l + r) / 2;
    MergeSort(a, l, m);
    MergeSort(a, m + 1, r);
    Merge(a, l, m, r);
  }
}

void result(int n) {
  switch (n) {
    case 1: {
      ifstream fin;
      fin.open("/Users/edith_lzh/Desktop/大三上/Algorithm/lab1/10.txt");

      int i = 0;
      int data[10];
      for (i = 0; i < 10; ++i) {
        fin >> data[i];
      }
```

```
51        MergeSort(data, 0, 9);
52        cout << data[4] << endl;
53        // for (i = 0; i < 10; i++) cout << data[i] << " ";
54        // cout << endl;
55
56        fin.close();
57        break;
58      }
59
60      case 2: {
61        ifstream fin;
62        fin.open("/Users/edith_lzh/Desktop/大三上/Algorithm/lab1/100.txt");
63
64        int i = 0;
65        int data[100];
66        for (i = 0; i < 100; ++i) {
67          fin >> data[i];
68        }
69        MergeSort(data, 0, 99);
70        cout << data[49] << endl;
71        // for (i = 0; i < 100; i++) cout << data[i] << " ";
72         //cout << endl;
73
74        fin.close();
75        break;
76      }
77
78      case 3: {
79        ifstream fin;
80        fin.open("/Users/edith_lzh/Desktop/大三上/Algorithm/lab1/1000.txt");
81
82        int i = 0;
83        int data[1000];
84        for (i = 0; i < 1000; ++i) {
85          fin >> data[i];
86        }
87        MergeSort(data, 0, 999);
88        cout << data[499] << endl;
89        // for (i = 0; i < 10; i++) cout << data[i] << " ";
90        // cout << endl;
91
92        fin.close();
93        break;
94      }
95
96      case 4: {
97        ifstream fin;
98        fin.open("/Users/edith_lzh/Desktop/大三上/Algorithm/lab1/10000.txt");
99
100       int i = 0;
101       int data[10000];
102       for (i = 0; i < 10000; ++i) {
103         fin >> data[i];
```

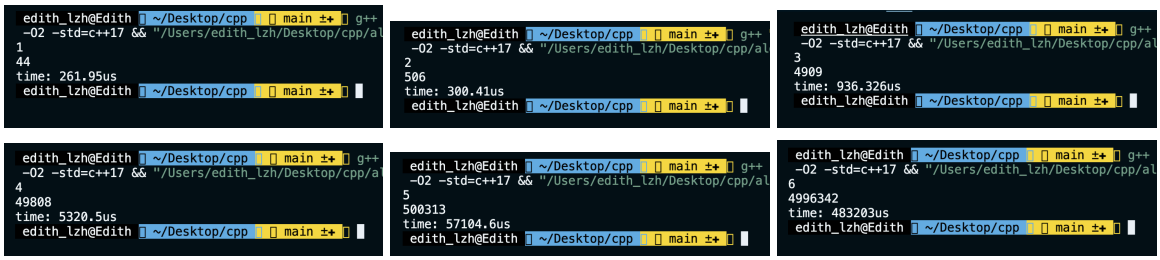
```
104       }
105       MergeSort(data, 0, 9999);
106       cout << data[4999] << endl;
107       // for (i = 0; i < 10; i++) cout << data[i] << " ";
108       // cout << endl;
109
110       fin.close();
111       break;
112     }
113
114     case 5: {
115       ifstream fin;
116       fin.open("/Users/edith_lzh/Desktop/大三上/Algorithm/lab1/100000.txt");
117
118       int i = 0;
119       int data[100000];
120       for (i = 0; i < 100000; ++i) {
121         fin >> data[i];
122       }
123       MergeSort(data, 0, 99999);
124       cout << data[49999] << endl;
125       // for (i = 0; i < 10; i++) cout << data[i] << " ";
126       // cout << endl;
127
128       fin.close();
129       break;
130     }
131
132     case 6: {
133       ifstream fin;
134       fin.open("/Users/edith_lzh/Desktop/大三上/Algorithm/lab1/1000000.txt");
135
136       int i = 0;
137       int data[1000000];
138       for (i = 0; i < 1000000; ++i) {
139         fin >> data[i];
140       }
141       MergeSort(data, 0, 999999);
142       cout << data[499999] << endl;
143       fin.close();
144       break;
145     }
146   }
147 }
148
149 int main() {
150   int n = 0;
151   cin >> n;
152   auto start = std::chrono::steady_clock::now();
153   result(n);
154   auto end = std::chrono::steady_clock::now();
155   std::chrono::duration<double, std::micro> elapsed =
156       end - start;  // std::micro 表示以微秒为时间单位
```

```
157     std::cout << "time: " << elapsed.count() << "us" << std::endl;
158 }
```

sort.cc

2.2 程序运行时间

在室内温度 25 摄氏度的环境下，快速进行了 6 次输入输出，最后的结果如下图所示:



3 线性时间选择

3.1 线性时间选择的实现

线性时间选择算法是一个非常精妙的算法。该算法基于分治算法的思想，在原数据集中合理地找出一个用来划分的元素 M，将数据集划分为大于 M，小于 M，等于 M 三类数据集，通过对目标元素的次序优先级的分类讨论，完成对原问题的巧妙剪枝。

关于划分元素 M 的选取也相当有趣。算法导论的做法是将数据分成 $[\frac{n}{5}]$ 的天花板函数，每一组至多有 5 各元素，对每一组排序（任何排序方法都可以，最多 5 个元素时间复杂度为常数），可以得到每一组中位数的序列，求中位数序列的中位数，将其作为 M。这样的话每次至少可以剪掉原规模 14 的数据量。

这让我想到了矩阵代数，上面求 M 的过程实际上就是对一个 m × n 的矩阵进行平均划分的过程，要求划分元素的左上方都是小于该元素的值，左下方都是大于该元素的值。所以事实上（我还没有仔细想过），应该划分不是唯一的（?），我可以随机选取划分的组数（比如每组至多 7 个，之所以是 7 而不是 6 的原因是因为奇数比较好定义中位数），应该也是行得通的。

线性选择算法的时间复杂度，很显然，是 O(n)，事实上，我看了 STL 的 Sort () 函数，他的思想和线性选择相仿。

具体代码如下:

解释一下几行 core code:

1. bubbleSort() 就是一个普普通通的冒泡函数，每冒一次泡都会把最大的元素冒到数组最后
2. Partition(a,p,r,val) 有四个行参，a 是输入数组，p、r 是进行 Partition 的范围，这个函数会对 a 进行重构，并且返回一个值 j，a[p] 到 a[j] 都是比 val 小或等于 val 的元素，a[j] 到 a[r] 都是比 val 大的元素
3. Select(a,p,r,k) 是线性选择算法本身，有四个行参，a 是输入数组，p、r 是进行选择的范围，该函数返回目标范围内第 k 小的数。函数的内部实现逻辑其实很明晰，具体可以见下面的各行注释，我已

经解释地很清楚了

4. Swap(a,b) 表示交换 a,b 的值

```
#include <chrono>
#include <fstream>
#include <iostream>
using namespace std;

void bubbleSort(int a[], int p, int r) {
  for (int i = p; i < r; i++) {
    for (int j = i + 1; j <= r; j++) {
      if (a[j] < a[i]) swap(a[i], a[j]);
    }
  }
}  // bubblesort(), can produce a list from small to large

int Partition(int a[], int p, int r, int val) {
  int pos;
  for (int q = p; q <= r; q++) {
    if (a[q] == val) {
      pos = q;
      break;
    }
  }
  swap(a[p], a[pos]);

  int i = p, j = r + 1, x = a[p];
  while (1) {
    while (a[++i] < x && i < r)
      ;
    while (a[--j] > x)
      ;
    if (i >= j) break;
    swap(a[i], a[j]);
  }
  a[p] = a[j];
  a[j] = x;
  return j;
}

int Select(int a[], int p, int r, int k) {  // larger k,  larger value
  if (r - p < 75) {
    bubbleSort(a, p, r);  // if not many data, just do sort()
    return a[p + k - 1];  // after it, a[n] is sorted, a[0] is the smallest,
    // for 10.txt, Select(a, 0, 9, 10) is the biggest,  Select(a, 0, 9, 1) is
    // the smallest Select(a, 0, 9, 5) is the mid-one
  }
  // divided [n/5] groups, each group has 1-5 items
  for (int i = 0; i <= (r - p) / 5; i++)
  //exchange every mid value with the head items in a[0:(r-p)/5]
  {
    int s = p + 5 * i, t = s + 4;
    for (int j = 0; j < 3; j++) {
```

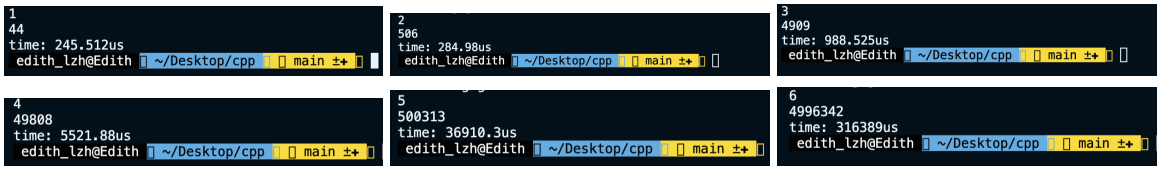
```
51        for (int n = s; n < t - j; n++) {
52          if (a[n] > a[n + 1]) swap(a[n], a[n + 1]);
53        }
54        // bubblesort for 3 times, then we get 3 largest values in each group
55      }
56      swap(a[p + i], a[s + 2]);  //exchange every mid items pf each group to the front
57    }
58    // get the mid value of mid values ( from a[p] : a[p+(r - p) / 5])
59    int x = Select(a, p, p + (r - p) / 5, (r - p + 5) / 10);  //求中位数的中位数
60    /*
61    (r-p+5)/10 = (p+(r+p)/5-p+1)/2
62    */
63    int i = Partition(a, p, r, x), j = i - p + 1;
64    if (k <= j)
65      return Select(a, p, i, k);
66    else
67      return Select(a, i+1, r, k - j);
68  }
69
70  void result(int n) {
71    switch (n) {
72      case 1: {
73        ifstream fin;
74        fin.open("/Users/edith_lzh/Desktop/大三上/Algorithm/lab1/10.txt");
75
76        int i = 0;
77        int data[10];
78        for (i = 0; i < 10; ++i) {
79          fin >> data[i];
80        }
81        cout << Select(data, 0, 9, 5) << endl;
82        break;
83      }
84
85      case 2: {
86        ifstream fin;
87        fin.open("/Users/edith_lzh/Desktop/大三上/Algorithm/lab1/100.txt");
88
89        int i = 0;
90        int data[100];
91        for (i = 0; i < 100; ++i) {
92          fin >> data[i];
93        }
94        cout << Select(data, 0, 99, 50) << endl;
95        break;
96      }
97
98      case 3: {
99        ifstream fin;
100       fin.open("/Users/edith_lzh/Desktop/大三上/Algorithm/lab1/1000.txt");
101
102       int i = 0;
103       int data[1000];
```

```
104       for (i = 0; i < 1000; ++i) {
105         fin >> data[i];
106       }
107       cout << Select(data, 0, 999, 500) << endl;
108       break;
109     }
110
111     case 4: {
112       ifstream fin;
113       fin.open("/Users/edith_lzh/Desktop/大三上/Algorithm/lab1/10000.txt");
114
115       int i = 0;
116       int data[10000];
117       for (i = 0; i < 10000; ++i) {
118         fin >> data[i];
119       }
120       cout << Select(data, 0, 9999, 5000) << endl;
121       break;
122     }
123
124     case 5: {
125       ifstream fin;
126       fin.open("/Users/edith_lzh/Desktop/大三上/Algorithm/lab1/100000.txt");
127
128       int i = 0;
129       int data[100000];
130       for (i = 0; i < 100000; ++i) {
131         fin >> data[i];
132       }
133       cout << Select(data, 0, 99999, 50000) << endl;
134       break;
135     }
136
137     case 6: {
138       ifstream fin;
139       fin.open("/Users/edith_lzh/Desktop/大三上/Algorithm/lab1/1000000.txt");
140
141       int i = 0;
142       int data[1000000];
143       for (i = 0; i < 1000000; ++i) {
144         fin >> data[i];
145       }
146       cout << Select(data, 0, 999999, 500000) << endl;
147       break;
148     }
149   }
150 }
151
152 int main() {
153   int n = 0;
154   cin >> n;
155   auto start = std::chrono::steady_clock::now();
156   result(n);
```

```
157   auto end = std::chrono::steady_clock::now();
158   std::chrono::duration<double, std::micro> elapsed =
159       end - start;  // std::micro 表示以微秒为时间单位
160   std::cout << "time: " << elapsed.count() << "us" << std::endl;
161 }
```

linear.cc

3.2 程序运行时间

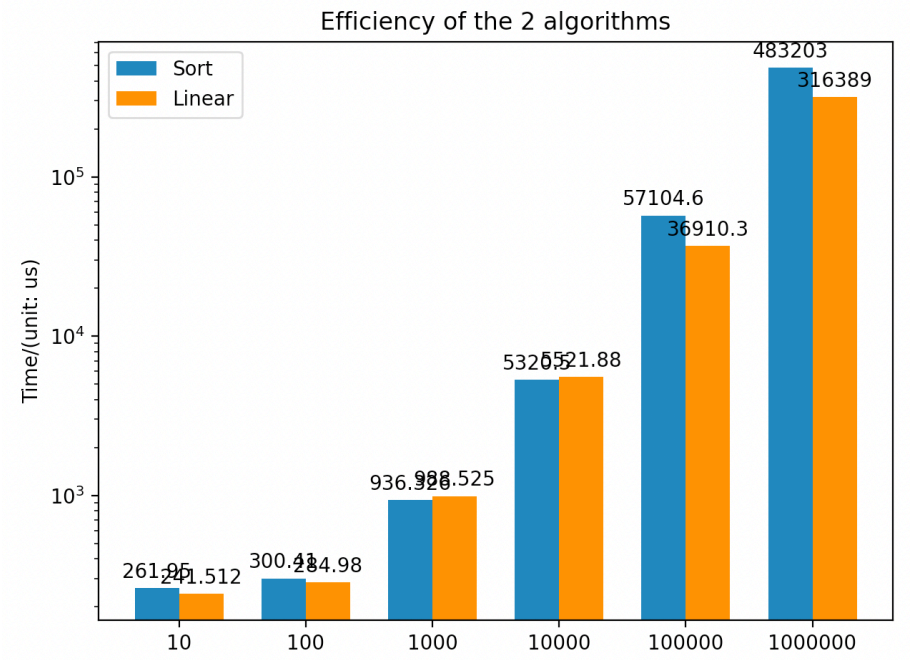


4 性能比较

我们可以从上面的结果清晰地发现，在数据集大小比较小的时候（小于 10000）的时候，直接归并排序的性能是优于线性时间选择算法的，但是当数据增多到 100000 甚至 1000000 的时候，线性时间选择算法明显优于归并排序。

有理由相信，性能拐点大概出现在 10000

用 python.matplotlib 画图，如下



画图的代码我也附在下面了：

```
1
2 import matplotlib.pyplot as plt
```

```python
import numpy as np

labels = ['10', '100', '1000', '10000', '100000','1000000']
sorttime = [261.95, 300.41,936.326 , 5320.5, 57104.6,483203]
lineartime = [241.512, 284.98, 988.525, 5521.88, 36910.3,316389]

x = np.arange(len(labels))  # the label locations
width = 0.35  # the width of the bars

fig, ax = plt.subplots()
rects1 = ax.bar(x - width/2, sorttime, width, label='Sort')
rects2 = ax.bar(x + width/2, lineartime, width, label='Linear')

# Add some text for labels, title and custom x-axis tick labels, etc.
ax.set_ylabel('Time/(unit: us)')
ax.set_title('Efficiency of the 2 algorithms')
ax.set_xticks(x)
ax.set_xticklabels(labels)
ax.legend()
plt.yscale("log")

ax.bar_label(rects1, padding=3)
ax.bar_label(rects2, padding=3)

fig.tight_layout()

plt.show()
```

graph.py