

selection algorithm

- 1.random selection 的实现
- 2.deterministic selection 的实现
- 3.debug记录(下面写的和上面的代码不太一样)
- 4.代码比较(每个规模50次运行取平均值)

selection algorithm

问题描述：在一组乱序的数组之中寻找任意第K小的元素

输入：数组指针与待查找的序号k

输出：返回第k小的元素

解决思路：

选取第k小的元素，最直观的思路就是先直接把数组排序，选取秩为k-1的数组元素就是第k小的数字。

如果选取的是mergesort, quicksort之类的排序算法，时间复杂度也能达到 $n\log n$ 的程度。

但是，还有更省时间的算法，把复杂度降低到 $\theta(n)$ 或者 $O(n)$ 的程度。

主体思路是：

quicksort每一层递归都可以在 $O(n)$ 的时间之内把一个元素摆到正确的位置上，且保证 {左边部分} $< \{a[p]\} < \{\text{右边部分}\}$

然后，如果k-1比p小，那么下一次只用处理左边；如果k-1等于p，那么直接返回p；如果k-1>p，那么下一次只用处理右边。

实现方式：

- random selection
- deterministic selection

1.random selection 的实现

算法流程：

1. 随机选择一个数组的元素，并把它和数组末尾元素交换
2. 把末尾元素放到正确的位置上，保证排在左边的元素全部小于它，排在右边的元素全部大于它
3. 比较k-1与末尾元素的位置r，三种可能：k-1 < r，目标一定在左边部分，返回左边的递归；k-1 == r，目标命中，直接返回；k-1 > r，目标一定在右边，返回右边的递归

伪代码

```
int random_selection(int a[], int first, int last, int k):
    if (first >= last) return a[first];
    在first到last，随机取第p个元素；
    把a[p]交换到末尾；
    p = partition(a, first, last);
    if (k-1 > p) return random_selection(a, p+1, last);
    else if (k-1 == p) return a[p];
    else return random_selection(a, first, p-1);
```

代码描述：

```
//假设k是合法的，不越界的
int random_selection(int a[], int first, int last, int k);
int partition(int a[], int first, int last);
void swap(int& a, int& b);

int random_selection(int a[], int first, int last, int k) {
    if (first >= last) return a[first]; //递归基，按照原理来说，其实不用它也不会出错
    int p = rand() % (last - first + 1) + first; //first不要漏加
    swap(a[p], a[last]);
    p = partition(a, first, last);
    if (k - 1 > p) {
        return random_selection(a, p + 1, last); //多个功能相似的函数一起写的时候，这个地方
        //不要不小心写错
    }
    else if (k - 1 == p) {
        return a[p];
    }
    else {
        return random_selection(a, first, p - 1);
    }
}

int partition(int a[], int first, int last) {
    int i = first, j = last - 1; //左边是first右边是last，绝对不要默认是从0到length
    while (true) {
        while (a[i] < a[last]) {
            i++;
            if (i == last) break; //不要漏写，因为对于升序的数组，是有可能到末尾的，要防止越界
        }
        while (a[j] >= a[last]) {
            j--;
            if (j == first) break; //不要漏写，因为对于降序的数组，是有可能到末尾的，要防止越界
        }
        if (i >= j) break; //注意跳出条件，也必须要写在交换前，否则会导致错误的交换
        swap(a[i], a[j]);
    }
    swap(a[i], a[last]);
    return i;
}

void swap(int& a, int& b) { //交换注意要使用引用
    int t = a;
    a = b;
    b = t;
}
```

random每次递归调用都可以减小一定的数据规模，但是在十分微小的概率下，也会出现很不巧每次选的都是最大或者最小元素，一次只能排除一个的情况。在这种极端情况下，时间复杂度会到达 $O(n^2)$ 。如果没有rand过程，对于有序数组几乎一定会出现这种情况。

于是，优化的思路是，如果保证每次选取的元素都是中位数，那么就几乎绝对安全了，不过对于所有元素同名的情况，似乎还是会很慢。

2.deterministic selection 的实现

伪代码:

```
int deterministic_selection(int[] a, int first, int last, int k):
    if(a中的元素小于等于10个):
        调用选择排序, 直接选出秩为k-1的元素;
        return a[k-1];
    else:
        把a切分, 每5个元素为一组, 多余的自成一组;
        每组调用基础排序, 取每个组的中位数;
        针对中位数数组调用deterministic_selection函数, 寻找到中位数;
        在原数组中找到此中位数位置为pivot, 交换它为末尾元素;
        p=partition(a,first,last);
        if(k-1<p) return deterministic_selection(a,first,p-1,k);
        else if(k-1==p) return a[p];
        else return deterministic_selection(a,p+1,last,k);
```

代码正文:

```
int deterministic_selection(int* a, int first, int last, int k);
int partition(int a[], int first, int last); //使用上面写的
void swap(int& a, int& b); //使用上面写的, 不过algorithm库似乎就有swap。。。
void select_pivot(int* a, int first, int last);

int deterministic_selection(int* a, int first, int last, int k) {
    if (first >= last) return a[first];
    if (last - first + 1 < 10) {
        _selection_sort(a, first, last);
        return a[k - 1];
    }
    select_pivot(a, first, last);
    int p = partition(a, first, last);
    if (k - 1 < p) return deterministic_selection(a, first, p - 1, k);
    else if (k - 1 == p) return a[p];
    else return deterministic_selection(a, p + 1, last, k);
}

void select_pivot(int* a, int first, int last) {
    int n_num = ceil((last - first + 1) / 5.0); //除以5然后向上取整, 注意是5.0
    int* b = new int [n_num]; //b中装的将是小组中位数
    int k = 0; //k指示b的元素;

    for (int i = first; i <= last; i += 5) { //i指示每一组的头部
        int j = i + 4;
        if (j <= last) {
            _selection_sort(a, i, j);
            b[k++] = a[i + 2];
        }
        else {
            _selection_sort(a, i, last);
            b[k++] = a[(i + last) / 2];
        }
    }
}
```

```
//selection_sort(b,0,n_num-1); 这种做法是有问题的，万一b很长，时间很长

int* c = new int[n_num];
for (int i = 0; i < n_num; i++) c[i] = b[i];
int element = determinstic_selection(b, 0, n_num - 1, (n_num - 1) / 2);
int rank = 0; //指示中位数在原始的b中的位置，借此可以求出中位数在a中的位置
for (int i = 0; i < n_num; i++) {
    if (c[i] == element) rank = i;
}
if (first + rank * 5 + 4 > last) {
    rank = (first + rank * 5 + last) / 2; //这是中位数在数组a中的位置
}
else {
    rank = (first + rank * 5 + 2);
}
swap(a[rank], a[last]);
}
```

3.debug记录(下面写的和上面的代码不太一样)

- 递归函数名写错

```
int _determinstic_selection(int* a, int first, int last, int rth) {
    if (rth > last) return -1;
    if (first >= last) return a[first]; //有点不清晰
    select_pivot(a, first, last);

    int p = partition(a, first, last);
    if (rth < p) return _selection(a, first, p - 1, rth);
    else if (rth == p) return a[p];
    else if (rth > p) return _selection(a, p + 1, last, rth);
}
```

函数主体叫determinstic_selection但是递归调用的时候怎么写成了selection

感想：1.这段代码是从_selection那里复制过来修改的，这个地方忘记修改了

2.精神状态不好的时候容易恍惚，然而对于代码来说是致命的

- 函数中的数组不一定是从0开始的，要提醒自己是不是忘记加左端了

```

void select_pivot(int* a, int first, int last) {
    int n_num = ceil((last - first + 1) / 5);
    int* b = new int[n_num];
    int k = 0;
    for (int i = 0; i <= last; i+=5) {
        int j = i + 4;
        if (j <= last) {
            _insertion_sort(a, i, j);
            b[k] = a[i + 2]; k++;
        }
        else if (j > last) {
            b[k] = a[i];
        }
    }

    int* c = new int[n_num];
    for (int i = 0; i < n_num; i++) {
        c[i] = b[i];
    }

    _insertion_sort(b, 0, n_num-1);
    int pivot = n_num / 2;
    int i = 0;
    for (; i < n_num; i++) {
        if (c[i] == b[pivot]) break;
    }

    pivot = i * 5 + 2;
    swap(a, pivot, last);
}

```

两处的错误都是一样的i不是从0开始的，而是从first开始的，pivot也不能直接是i*5+2而应该再加上first

感想：或许要养成一种从first到last的习惯，在草稿上或是什么别的东西上都要先有first开头的意识；

- partition最后的i位置就是last元素应该交换的位置，写成i-1就错了

```

int partition(int* a, int first, int last) {
    int pivot = a[last];
    int i = first, j = last - 1;
    while (true)
    {
        while (a[i] < pivot) {
            i++;
            if (i == last) break;
        }
        while (a[j] >= pivot) {
            j--;
            if (j == first) break;
        }
        if (i >= j) break;
        swap(a, i, j);
    }

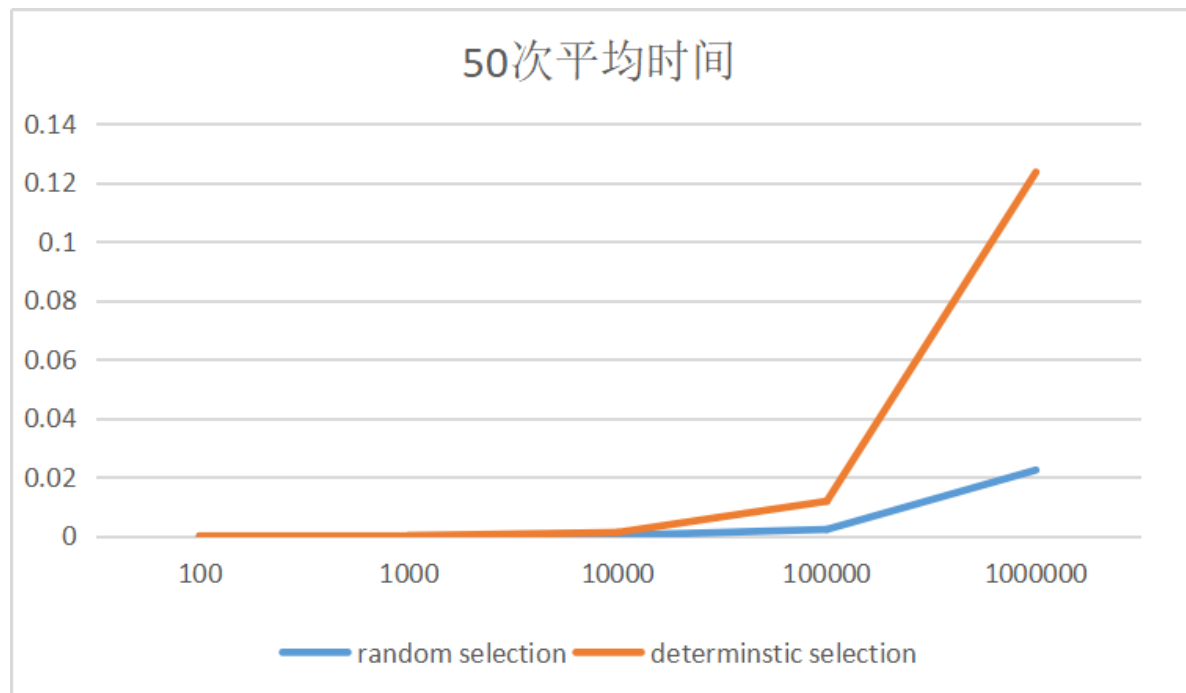
    swap(a, i-1, last);
    return i-1;
}

```

我写的partition所做的工作是把last元素选作pivot，放置到合适的位置，使得左边的元素全部小于pivot，右边的元素大于pivot。方法是双指针，i指针指示比pivot小的元素，j指针指示大于等于pivot的元素，一旦 $i \geq pivot$ 而 $j < pivot$ ，那么交换i与j，循环的条件是 $i > j$ 。那么可以直接确定i最后所指的元素必然大于等于last，j最后所指位置必然小于last。

我犯的错误就是把last与i-1互换，想当然认为既然i所指比last大，那么肯定last元素正确的位置是i-1，实际上，信息还有i左边的元素一定比last小，也就是说，一旦把i-1与last互换，就把一个小于last的元素交换到最右端。

4.代码比较(每个规模50次运行取平均值)



Microsoft Visual Studio 调试控制台

```
数据规模为: 100
random selection      平均时间为 0
deterministic selection 平均时间为 0
数据规模为: 1000
random selection      平均时间为 2e-05
deterministic selection 平均时间为 0.00012
数据规模为: 10000
random selection      平均时间为 0.00022
deterministic selection 平均时间为 0.00122
数据规模为: 100000
random selection      平均时间为 0.00222
deterministic selection 平均时间为 0.0118
数据规模为: 1000000
random selection      平均时间为 0.02236
deterministic selection 平均时间为 0.12346
```

F:\C++\selection_sort\Debug\selection_sort.exe (进程 5336) 已退出，代码为 0。
按任意键关闭此窗口。 . .

结论：

selection 中random比deterministic要快

原因推测是：

random中的pivot是很不对称的pivot，有些时候可以一下子舍弃很多数据，然后只在很小的数据之中寻找元素

而deterministic的pivot是中位数，太过于保守了，虽然避免了random可能出现的最坏情况

n^2 (极其小的概率), 但是中规中矩, 很少会出现一次性舍弃大量的幸运情况(除非重复的元素很多).