# HW7

522031910213 朱涵

April 23, 2024

## 1 算法实现

### 1.1 Quick Select

代码见下图。基本逻辑和课本上的示例代码一样，每一次选择数组开头元素作为pivot进行调整，不断把比它大的元素放到右边，把小元素放到左边，巧妙的通过一开始pivot空出的一个位置进行交换，达到了常数的空间复杂度。

```
int quickSelect(vector<int> &nums, int k)
{
    int n = nums.size();
    for (int lo = 0, hi = n - 1; lo < hi;)
    {
        int i = lo, j = hi;
        int pivot = nums[lo];
        while (i < j)
        {
            while (i < j && nums[j] >= pivot)
                j--;
            nums[i] = nums[j];
            while (i < j && nums[i] <= pivot)
                i++;
            nums[j] = nums[i];
        }
        nums[i] = pivot;
        if (k <= i)
            hi = i - 1;
        if (i <= k)
            lo = i + 1;
    }
    return nums[k];
}
```

Figure 1: 快速选择算法实现代码

### 1.2 Linear Select

线性选择代码的实现如下图。基本逻辑和课本上的伪代码一样。主要分为三部分工作:

1. 递归基：当数据规模小于Q时，直接进行排序并返回中位数索引；
2. 划分子序列&找到中位数序列的中位数：依照Q进行划分，每一个子序列都利用排序后计算出中位数的索引，然后把中位数交换到数组的开头位置，全部操作完之后，数组开头的子数组就是子序列的中位数数组。随后再次递归调用线性选择函数找到中位数数组的中位数，得到索引（设为M），存放到数组末尾；
3. 根据M元素对整个数组进行划分，得到L、E、G三个子集。我的代码中调用了partition函数（见图3）进行了划分，会将L部分放到左端，G部分放到右端，low指向了L部分的右端索引，high指向了G部分的开头索引，由此计算出L、G、E的大小，最后根据三种情况进行递归。

```cpp
// 线性选择算法
int linearSelect(vector<int> &nums, int begin, int end, int k, unsigned Q)
{
    // 递归基
    if (end - begin < Q){
        sort(nums.begin() + begin, nums.begin() + end);
        return begin + k;
    }
    // 划分数据集&选择中位数的中位数
    {   int i = begin, j = begin;
        unsigned n = (end - begin) / Q;
        while (n--){
            sort(nums.begin() + i, nums.begin() + i + Q);
            swap(nums[j++], nums[i + Q / 2]);
            i += Q;
        }
        if (i != end){
            sort(nums.begin() + i, nums.begin() + end);
            swap(nums[j++], nums[i + (end - i) / 2]);
        }
        int pivot = linearSelect(nums, begin, j, (j - begin) / 2, Q);
        swap(nums[end - 1], nums[pivot]);
    }
    // 划分为三个子集 减小规模
    {   int low = begin, high = end - 1, pivot = high;
        partition(nums, low, high, pivot);
        int l = low - begin, le = high - begin;
        if (l <= k && k < le)
            return begin + k;
        else if (k < l)
            return linearSelect(nums, begin, low, k, Q);
        else
            return linearSelect(nums, high, end, k - le, Q);
    }
}
```

Figure 2: 线性选择算法代码实现

```cpp
void partition(vector<int> &nums, int &low, int &high, int &pivot)
{
    for (int i = low; i != high;)
    {
        if (nums[i] > nums[pivot])
        {
            swap(nums[i], nums[--high]);
        }
        else
        {
            if (nums[i] < nums[pivot])
                swap(nums[i], nums[low++]);
            ++i;
        }
    }
    swap(nums[pivot], nums[high++]);
}
```

Figure 3: 划分函数实现

# 2 实验思路

## 2.1 数据集设计

- 第一部分:主要对两种算法在不同的规模及特性的数据集下的性能进行对比，设计共三种数据集（顺序、逆序、乱序）和十种数据规模（1w-10w，步长为1w）；
- 第二部分：主要对线性选择算法在不同的规模以及参数Q下的性能进行测试，设计一种数据集（乱序）、十九种参数（2-20，步长为1）以及三种数据规模（0.1w，1w，10w）。

## 2.2 实验思路

- 第一部分:在三种数据集和十种数据规模下分别对两种算法进行测试，输出数据并绘制三幅图（对应三种数据集），每幅图两条曲线（对应两种算法），每条曲线展示了算法性能随数据规模变化的曲线;
- 第二部分：主要对线性选择算法在不同的规模以及参数Q下的性能进行测试，输出数据并绘制一幅图，三条曲线（对应三种数据规模）。每条曲线展示了线性选择算法性能随参数Q变化的曲线;
- 注：由于第一部分的线性选择算法的参数Q需要进行合适的选择，因此在实际进行实验时，本人先进行了第二部分的实验并确定了最优性能下的Q=7，然后使用了此参数进行了第一部分的测试。

# 3 实验结果及分析

## 3.1 实验结果

实验结果见下图。

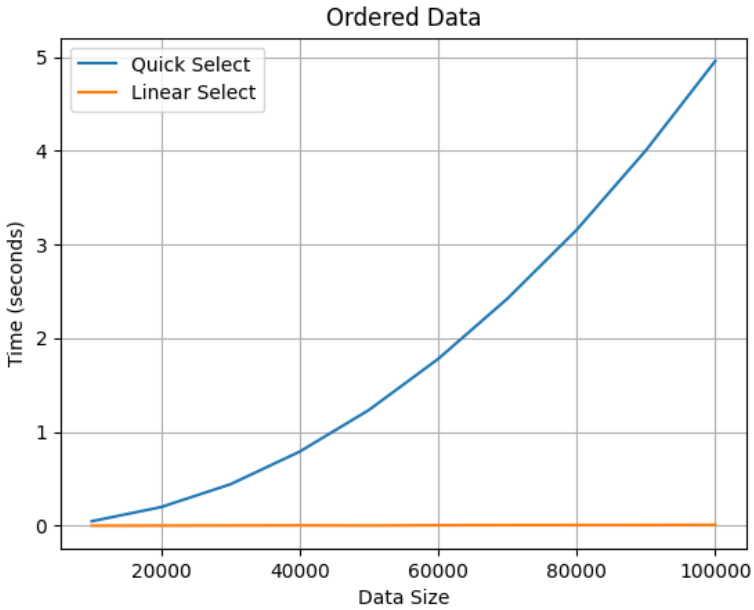


Figure 4: 顺序数据集下两种算法性能的变化曲线

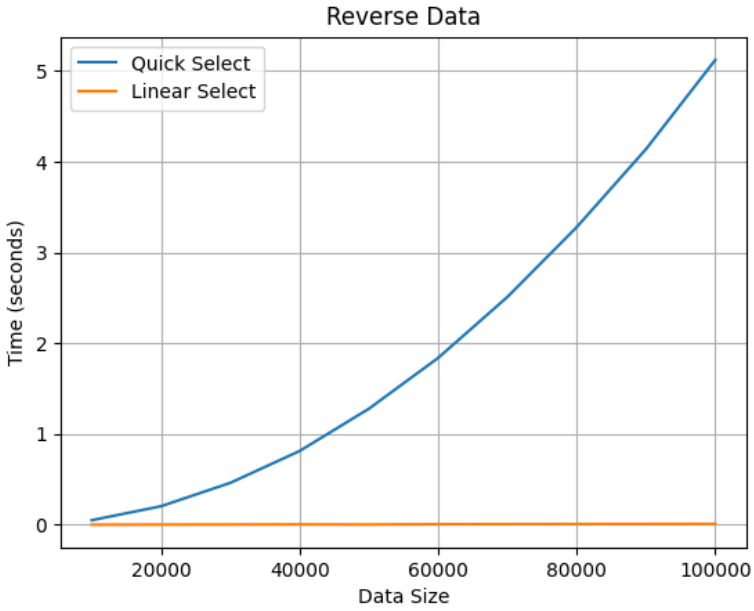


Figure 5: 逆序数据集下两种算法性能的变化曲线


Figure 6: 乱序数据集下两种算法性能的变化曲线

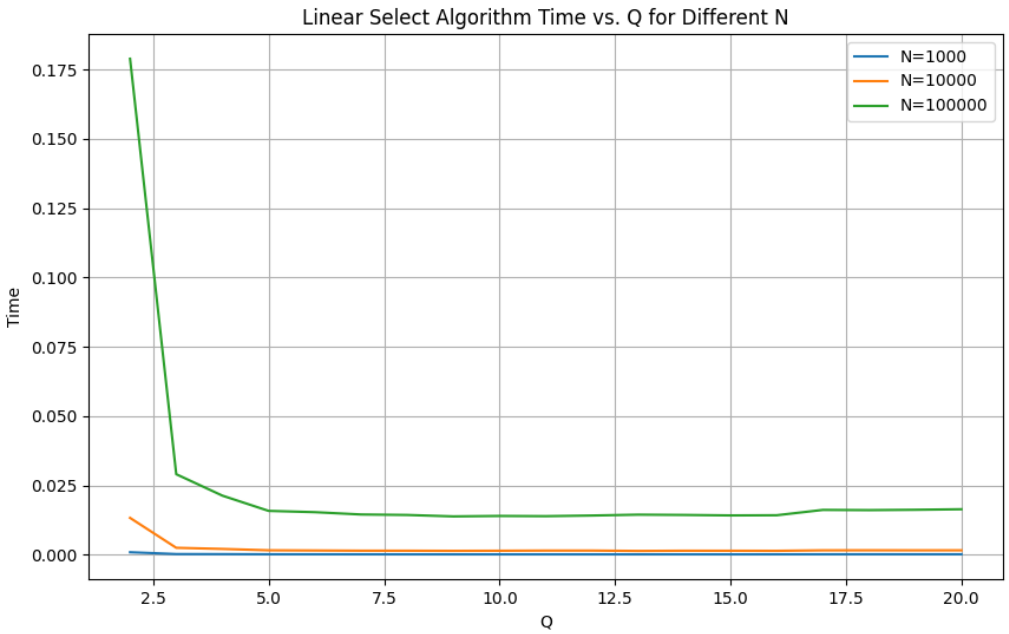


Figure 7: 不同数据规模下线性选择算法性能随Q变化曲线

## 3.2 结论

- 第一部分:观察图像可以发现，当数组有序（顺序或逆序）时，由于快速选择算法总是选取开头元素作为pivot，区间的划分十分低效，达到了最差复杂度$O(n^2)$,性能远低于线性选择算法；当数组乱序时，两种算法的耗时均与数据规模N呈线性关系，而线性选择算法的性能要高于快速选择，这与数据规模或许是有关系的，在更大或更小的规模可能会因为线性选择算法复杂度的常数项而有不同的差别，这一点有待继续深入探究。
- 第二部分：观察图像可以发现，在不同的数据规模下线性选择算法的性能都呈先下降后趋于稳定的趋势，基本上都在Q=7时达到了最优性能。
- 总结：线性选择算法的性能在本实验的数据规模下要明显优于快速选择，而快速选择的适用数据也比较少，只能用于乱序随机数据。此外，线性选择算法大概在参数Q=7时有最优性能。