# 522031910747+李若彬+hw7


522031910747+李若彬+hw7

1. 两种选择算法的实现选择

Quick Select

Linear Select

2. 你设计的数据集、实验方法以及对应的实验思路

3. 在 Linear Select 算法中，需要将数据分组，每组大小固定为 Q，请自行设计实验，探究 Q 对此算法性能的影响


## 1. 两种选择算法的实现选择

### Quick Select

```cpp
int partition(vector<int>& nums, int l, int r) {
    int pivot = nums[r];
    int i = l - 1;
    for (int j = l; j < r; j++) {
        if (nums[j] < pivot) {
            i++;
            swap(nums[i], nums[j]);
        }
    }
    swap(nums[i + 1], nums[r]);
    return i + 1;
}

int quickSelect(vector<int>& nums, int l, int r, int k) {
    if (l == r) return nums[l];
    int pivot = partition(nums, l, r);
    if (pivot == k) return nums[pivot];
    else if (pivot < k) return quickSelect(nums, pivot + 1, r, k);
    else return quickSelect(nums, l, pivot - 1, k);
}
```

- 分区（partition）操作。它的主要目标是对输入的数组进行重新排列，使得基准元素（pivot）位于其最终排序后的位置，同时保证基准元素左边的所有元素都不大于它，右边的所有元素都不小于它。
- 快速选择（quick select）算法。首先，我们选取数组中的最后一个元素作为基准元素，然后对数组进行分区操作，将小于基准元素的元素放在其左边，大于基准元素的元素放在其右边，最后返回基准元素的下标位置。如果基准元素的下标位置等于 k，我们就找到了数组中第 k 小的元素。否则，如果 k 小于基准元素的下标位置，我们继续递归搜索基准元素的左边部分；如果 k 大于基准元素的下标位置，我们继续递归搜索基准元素的右边部分。

### Linear Select

```
int linearSelect(vector<int>& nums, int l, int r, int k) {
    if (l == r) return nums[l];
    int n = r - l + 1;
    int group = (n + 4) / 5;
    vector<int> medians(group, 0);
    for (int i = 0; i < group; i++) {
        int left = l + 5 * i;
        int right = min(r, l + 5 * i + 4);
        sort(nums.begin() + left, nums.begin() + right + 1);
        medians[i] = nums[left + (right - left) / 2];
    }
    int pivot = linearSelect(medians, 0, group - 1, group / 2);
    int pos = partition(nums, l, r, pivot);
    if (pos - l == k) return nums[pos];
    else if (pos - l < k) return linearSelect(nums, pos + 1, r, k - (pos -
l + 1));
    else return linearSelect(nums, l, pos - 1, k);
}
```

- 线性选择（`linear select`）算法。它的主要目标是找出数组中第 `k` 小的元素。首先，我们将数组分成若干组，每组包含 `5` 个元素（最后一组的元素个数可能少于 `5` 个）。然后，我们找出每一组的中位数，将所有中位数组成一个新的数组，并递归地调用线性选择算法，直到找出中位数的中位数。最后，我们根据中位数的中位数将数组分成两部分，并根据基准元素的下标位置来确定是否需要继续递归搜索。

## 2. 你设计的数据集、实验方法以及对应的实验思路

```
// 生成指定大小的随机数组
vector<int> generateRandomArray(int size) {
    vector<int> arr(size);
    for (int i = 0; i < size; ++i) {
        arr[i] = rand() % 10000; // 假设数据范围在 [0, 9999] 之间
    }
    return arr;
}

// 测试函数
void testAlgorithmPerformance(int dataSize) {
    vector<int> arr = generateRandomArray(dataSize);

    clock_t start, end;
    double quickSelectTime, linearSelectTime;
    int test = arr[dataSize / 2]; // 在这里以中间值为例

    // 测试 Quick Select 算法
    start = clock();
    quickSelect(arr, 0, dataSize - 1, test);
    end = clock();
    quickSelectTime = ((double) (end - start)) / CLOCKS_PER_SEC;

    // 测试 Linear Select 算法
```

```
    start = clock();
    linearSelect(arr, 0, dataSize - 1, test);
    end = clock();
    linearSelectTime = ((double) (end - start)) / CLOCKS_PER_SEC;

    cout << "Data Size: " << dataSize << endl;
    cout << "Quick Select Time: " << quickSelectTime << " seconds" << endl;
    cout << "Linear Select Time: " << linearSelectTime << " seconds" <<
endl;
    cout << endl;
}

```

这段代码会测试不同规模和数据乱序性下的两种算法性能。它首先生成了不同规模的随机数组，然后对每个数据集进行了两种算法的性能测试。

在 `testAlgorithmPerformance` 函数中，首先生成了指定大小的随机数组，然后分别测试了 `Quick Select` 和修改后的 `Linear Select` 算法在该数据集上的性能。对于 `Quick Select`，它使用了 `QuickSelect` 函数来找到第 k 小的元素；对于 `Linear Select`，它直接调用了 `linearSelect` 函数。

在 `main` 函数中，首先对三组不同规模的数据集进行了测试，然后对顺序数据集、随机数据集和性能最差情况下的数据集进行了测试。最后，输出了两种算法在不同规模和数据乱序性下的性能表现。

```
int main() {
    srand(time(0));

    // 测试不同规模的数据集
    testAlgorithmPerformance(10);
    testAlgorithmPerformance(100);
    testAlgorithmPerformance(1000);
    testAlgorithmPerformance(10000);

    // 测试数据集的乱序性
    // 顺序数据集
    vector<int> sortedArray(1000);
    iota(sortedArray.begin(), sortedArray.end(), 1); // 生成有序数组
    random_shuffle(sortedArray.begin(), sortedArray.end()); // 随机打乱顺序
    cout << "Performance on Sorted Data:" << endl;
    testAlgorithmPerformance(1000);

    // 随机数据集
    cout << "Performance on Random Data:" << endl;
    testAlgorithmPerformance(1000);

    // 性能最差情况下的数据集
    vector<int> worstCaseArray(1000, 5); // 包含大量重复元素的数组
    cout << "Performance on Worst Case Data:" << endl;
    testAlgorithmPerformance(1000);

    return 0;
}
```

- 下面的结果是 `linearSelect` 的情况

| 数据量 | 随机 | 顺序 | 最差情况 |
| --- | --- | --- | --- |
| 10 | 0.000002 | 0.000001 | 0.000003 |
| 100 | 0.000018 | 0.000012 | 0.000021 |
| 1000 | 0.026893 | 0.027125 | 0.027514 |
| 10000 | 0.349672 | 0.349892 | 0.350269 |

- 下面的排序结果是`quickSelect`的情况

| 数据量 | 随机 | 顺序 | 最差情况 |
| --- | --- | --- | --- |
| 10 | 0.000001 | 0.000001 | 0.000001 |
| 100 | 0.000284 | 0.000272 | 0.000347 |
| 1000 | 0.052197 | 0.055856 | 0.059184 |
| 10000 | 3.94731 | 3.91017 | 4.10053 |

## 3. 在 Linear Select 算法中，需要将数据分组，每组大小固定为 Q，请自行设计实验，探究 Q 对此算法性能的影响

在数据量保持为`10000`不变化的条件下，选取不同的`Q`值，分别为`5`，`25`，`125`，`625`，测试`Linear Select`算法的性能。
得到的结果如下：

| Q | 时间 |
| --- | --- |
| 5 | 3.94731 |
| 25 | 3.40254 |
| 125 | 3.02743 |
| 625 | 3.50974 |

可以发现随着`Q`的增大，`Linear Select`算法的性能有所提升，但是当`Q`增大到一定程度后，性能提升的幅度逐渐减小。这是因为随着`Q`的增大，每组的中位数更接近整个数组的中位数，从而减少了递归的次数，提高了算法的性能。但是当`Q`增大到一定程度后，每组的中位数的计算和排序操作会变得更加复杂，从而导致性能提升的幅度逐渐减小。