Algorithms: Selection in Linear Time

2017-18570 Sungchan Yi
Dept. of Computer Science and Engineering

April 6th, 2019

Contents

1 Introduction

In this homework, we implement the randomized and deterministic selection algorithms with C++ language. The deterministic selection algorithm must run in $O(n)$ time. Given n elements in an array and an integer k ($0 \le k < n$), the algorithm must find the k-th smallest element in the array.[1] Additionally, we implement a checker program for the correctness for the implemented algorithms. The following are required:

- Implement the randomized and deterministic selection algorithms.
- Run each algorithm for given inputs, measure the running time, print the k-th smallest element.
- Check the correctness of each algorithm by using a checker program that runs in $O(n)$ time. Print the result of checking.

[1] For simplicity in programming, we will count from 0.

- After measuring the running time for inputs of various sizes, compute the ratios of the constant hidden in the asymptotic complexities of each algorithm.

The following are the restrictions on given inputs.

- n, k are integers, with $1 \le n \le 1,000,000,000$, $0 \le k < n$.
- All the elements given in the array are distinct.[2]

2 Implementation

In this section, we will cover the implementation of each algorithm. The basic idea is divide and conquer. Select a pivot and partition the given array into subproblems. The two algorithms discussed here differs only on the method of selecting the pivot.
For the codes, assume the necessary header files are already included.

2.1 Randomized Select

```
int randomized_select(int a[], int n, int k) {
    return randkth(a, 0, n - 1, k + 1);
}
```

The RANDOMIZED-SELECT algorithm chooses a pivot randomly from the array. The algorithm is implemented in `randkth` method. `randkth(arr, st, ed, k)` will return the k-th smallest element from `arr[st..ed]`.[3]

```
int randkth(int arr[], int st, int ed, int k) {
    if(st == ed) // base case for recursion
        return arr[st];
    int pivot = randomized_partition(arr, st, ed); // index of pivot

    // index of pivot from strarting from st
    int idx = pivot - st + 1;

    // Divide and Conquer
    if(k < idx) // search left
        return randkth(arr, st, pivot - 1, k);
    else if(k > idx) // search right
        return randkth(arr, pivot + 1, ed, k - idx);
    else // element is found
        return arr[pivot];
}
```

[2] This condition is required for the deterministic selection algorithm to run in $O(n)$ time.
[3] `A[i..j]` denotes the elements from `A[i]`, ..., `A[j]`.

For `randomized_partition`, select a random index from `st..ed` and use it as a pivot.

```
int randomized_partition(int arr[], int st, int ed) {
    srand(time(NULL));
    int pIdx = st + rand() % (ed - st + 1); // random index
    swap(arr, ed, pIdx); // put it at the end of array
    int x = arr[ed]; // pivot element
    int i = st - 1; // track the correct place of pivot
    for(int j = st; j < ed; ++j) {
        if(arr[j] <= x)
            swap(arr, ++i, j); // increment i then swap
    }
    swap(arr, ++i, ed); // put pivot at correct place
    return i;
}
```

`swap(A, i, j)` will swap `A[i]` and `A[j]`. Its implementation is trivial, so we omit it here.

According to the CLRS textbook, the above algorithm will run in expected $O(n)$ time. But the worst case complexity is $O(n^2)$, since we get extremely unlucky by the random function choosing the smallest/largest element from `A[st..ed]`. That choice of pivot will lead to an unbalanced partition of input array.

2.2 Deterministic Select

```
int deterministic_select(int a[], int n, int k) {
    return detkth(a, 0, n - 1, k + 1);
}
```

The deterministic selection algorithm chooses a pivot by using "median of medians". First divide the array into $\lceil n/5 \rceil$ sub-arrays, each with 5 elements, except for the last sub-array. Then choose a median from each sub-array and from the chosen medians, choose a median and use it as the pivot for partition. `detkth(arr, st, ed, k)` will return the k-th smallest element from `arr[st..ed]`.

```
int detkth(int arr[], int st, int ed, int k) {
    if(st == ed) // base case
        return arr[st];
    int n = ed - st + 1; // number of elements

    // array for storing medians of each sub-array
    int* med = (int*) malloc((n + 4) / 5 * sizeof(int));
    // find medians of each subarray
    int i;
```

```
10        for(i = 0; i < n / 5; ++i)
11            med[i] = median(arr + st + 5 * i, 5);
12        if(5 * i < n) {
13            med[i] = median(arr + st + 5 * i, n % 5);
14            i++;
15        }
16
17        // choose median of medians
18        int medOfMed = (i == 1) ? med[i-1] : detkth(med, 0, i-1, i/2);
19
20        // partition the array with median of medians
21        int pivot = partition(arr, st, ed, medOfMed);
22        int idx = pivot - st + 1;
23
24        // divide and conquer
25        if(idx == k)
26            return arr[pivot];
27        else if(idx > k)
28            return detkth(arr, st, pivot - 1, k);
29        else
30            return detkth(arr, pivot + 1, ed, k - idx);
31    }
```

`i == 1` in line 18 was to handle the case with the input array size less than 6. To find the median for each sub-array, `median(arr, n)` method was called, and it will find the median of n elements starting from `arr` (pointer) by insertion sort.

```
int median(int arr[], int n) {
    for(int i = 1; i < n; i++) {
        for(int j = i; j > 0; j--) {
            if(arr[j] < arr[j - 1]) swap(arr, j, j - 1);
            else break;
        }
    }
    return arr[n / 2];
}
```

`partition` process is done similarly.

```
int partition(int arr[], int st, int ed, int x) {
    // search for x in arr, and move it to the end
    int i;
    for(i = st; i < ed; ++i) {
        if(arr[i] == x) break;
    }
    swap(arr, i, ed);
```

```
        i = st - 1;
        for(int j = st; j < ed; ++j) {
            if(arr[j] <= x) // increment i then swap arr[i] and arr[j]
                swap(arr, ++i, j);
        }
        swap(arr, ++i, ed);
        return i;
}
```

According to the textbook, the above algorithm will run in $O(n)$ time.

2.3 Checker

To check the correctness, we implement a checker program. The basic idea is to count the number of elements less than or equal to the return value of above algorithms.

```
bool checker(int a[], int n, int k, int ans){
    int cnt = 0;
    for(int i = 0; i < n; ++i) {
        if(ans >= a[i]) cnt++; // count elements
    }
    if(cnt == k) // cnt must be k to be true
        return true;
    else return false;
}
```

This checker runs in $O(n)$ time because it sweeps the given array and compares each element with `ans`.

3 Experiments

Here is how the test was done.

- Test environment
  - OS: Ubuntu 18.04.2 LTS
  - CPU: Intel Core i5-6200U CPU @ 2.30GHz $\times$ 4
  - RAM: 8GB
- Checker Code

  `chrono` header file was used to check the wall clock.

```
#include <chrono>
using namespace std::chrono;
```

```
... // get input and etc. omitted here
auto start = high_resolution_clock::now(); // start time
int ans1 = randomized_select(arr, n, k);
auto stop = high_resolution_clock::now(); // end time

// check correctness
if(checker(arr, n, k, ans1)) printf("%s","correct");
else printf("%s","incorrect");

auto duration = duration_cast<microseconds>(stop - start);
printf(", Execution time ");
std::cout << duration.count() / 1000.0 << " ms\n";
```

Similar code was also run for `deterministic_select` algorithm.

- The checker code was saved to `checker.cpp`.
  - Compile: `g++ check.cpp -o check`
  - Run: `./check.cpp < 1.txt`

There were no reports of incorrectness during the following experiments.

3.1 Randomized Select

Since the running time of randomized algorithms differ for every execution, we ran the algorithm 10,000 times and take the average execution time.

Array Size (n)	20,000	60,000	200,000	600,000
Average Time (ms)	0.363094	0.926259	1.955946	9.627423

Table 1: Average running time of RANDOMIZED-SELECT algorithm

The estimated complexity is $T(n) = 0.01615n - 335.4$ (μs), with $R^2 = 0.9789$. The R^2 value is less than the deterministic select due to the randomness of the algorithm.

3.2 Deterministic Select

The running time for deterministic algorithm does not differ very much. We ran it once here.

Array Size (n)	20,000	60,000	200,000	600,000
Time (ms)	1.922	6.265	17.785	54.067

Table 2: Running time of deterministic select algorithm

The estimated complexity is $T(n) = 0.09034n + 269.3$ (μs), with $R^2 = 0.9996$.
Just for curiosity, we ran the program 1,000 times to see the average.

Array Size (n)	20,000	60,000	200,000	600,000
Average Time (ms)	0.902151	2.726879	10.423119	31.067172

Table 3: Average running time of deterministic select algorithm

It can be easily seen that the deterministic algorithm show linearity.

4 Conclusion

We calculated the ratio of running times by dividing the results of Table 2 by Table 1, for each n.

Array Size (n)	20,000	60,000	200,000	600,000
Ratio	5.293	6.764	9.093	5.616

Table 4: Ratio of running times for each n

With only the leading coefficients of the complexities for each algorithm, we can see that the ratio is $0.09034/0.01615 = 5.593808$. We conclude that the deterministic selection algorithm is around 5.5 times slower than the randomized selection algorithm.