VE281 Selection Algorithms Comparison Report

Xinhao Liao 516370910037

October 2018

1 Introduction

There are various algorithms used to select an element from an array. For instance, one can firstly sort the array and then return the element with the given index. Another 2 common algorithms are randomized select algorithm and deterministic select algorithm. In this report, these algorithms are to be compared with randomly generated integer arrays. (Quicksorting is used to sort the array.)

2 Background

Consider an array A with n integers as elements. The i-th smallest element can be selected with following algorithms.

Select after Sorting

Sort the array first and then select the element with the required index. Since the time complexity of quicksorting should be $\Theta(nlogn)$ for the best case and the average case, and $\Theta(n^2)$ for the worst case, the complexity of this algorithm is the same. The C++ code for this algorithm can be found in Appendix C as the function *SortSelect* with the help of a function called *Quicksort* using in-place quicksorting given in Appendix B.

Randomized Select

As in quicksort, this algorithm partitions the input array recursively. The pivot for partitioning is randomly generated. If the size of the left subarray is larger than index of the wanted element plus 1, than work on the left subarray. Otherwise work on the right subarray until the size of the left subarray is equal to index plus 1. Unlike quicksort, which recursively processes both sides of the partition, this algorithm works on only one side of the partition. This difference shows up in the analysis: whereas quicksort has an expected average time complexity time of $\Theta(nlogn)$, the expected average complexity of RANDOMIZED-SELECT is $\Theta(n)$, assuming that the elements are distinct. And the worst-case runtime is $\Theta(n^2)$. The C++ code for this algorithm can be found in Appendix A as the function *RandomizedSort.*

Deterministic Select

Like Randomized Select, this algorithm finds the desired element by recursively partitioning the input array. Here, however, a good split upon partitioning the array is guaranteed by choosing a good pivot. Break the array with size n into $n/5$ groups of size 5 each, and sort each group (using insertion sort with the best performance when the array sorted has a small size). Next, copy $n/5$ medians into new array C and recursively compute median of C by calling the deterministic selection algorithm! The median of C is the good pivot required. Then partition the array with this pivot and recursively work on its left or right subarray as is done for Randomized Select. In the worst case, the required runtime is $\Theta(n)$ for this algorithm. The C++ code for this algorithm can be found in Appendix A as the function *DeterministicSelect.*

3 Procedures

1. Set i to be the size of the array, which is initially 10 in this report.
2. Generate i random integers in the array. Save a copy in an auxiliary array.

3. Generate 5 random integers in the range $[0, i-1]$ as the index of the elements to be selected. 4. Select the elements with the algorithms and measure the average time cost. Write the measured time cost to an output file. Remember to copy the integers in order back to keep the same order for every algorithm.
5. Multiply i with 10 and repeat steps 2 to 4 until i reaches 1000000.

The code implementing the above procedures is shown in Appendix C.

4 Result

Size	10	100	1000	10000	100000	1000000
RSelect	0.0006ms	0.001ms	0.0138ms	0.0632ms	0.6824ms	8.059ms
DSelect	0.0006ms	0.0032ms	0.0416ms	0.2578ms	2.5408ms	24.888ms
SortSelect	0.0008ms	0.0084ms	0.0972ms	0.5384ms	7.1226ms	71.9584ms

Table 1: Result for selecting with different algorithms.

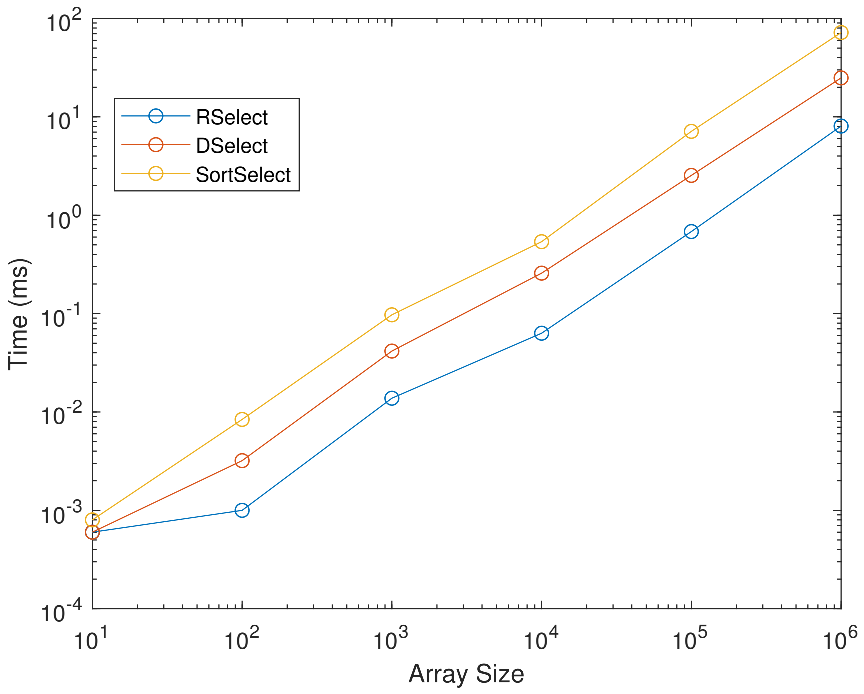

Figure 1: Curves corresponding to the 3 selecting algorithms.

The results for selecting from an array of 10, 100, 1000, 10000, 100000 and 1000000 random integers are shown in Table 1. The integers are randomly generated in the range $[-2^{31}, 2^{31}-1]$ and saved as *long int* in 32 bits. And we can plot all three curves corresponding to the three selecting algorithms in the same figure in Figure 1. The curves are plotted in log scales.

The results are porduced in Linux Ubuntu18.04 running in VMware Workstation 14.x virtual machine distributed with 4GB memory and 3 processors, with Intel CORE i9.

According to the results shown above, the three algorithms on average have similar performance. In general, randomized selecting has the best performance. deterministic selecting is the second best, and selecting after quicksorting has the worst performance.

5 Discussion and analysis

The performance of Selecting after Sorting is generally worse than that of Randomized Select and Deterministic Select. This conforms to the theoretical analysis that Selecting after Sorting has the time cost of $\Theta(nlogn)$ in the average case, which is worse than that of Randomized and Deterministic Select, which are $\Theta(n)$.

Randomized Select is faster than Deterministic Select on average. This is reasonable, although they theoretically both have the time complexity of $\Theta(n)$ for the average case, and the worst-case time complexity of Randomized Select ($\Theta(n^2)$) is even worse than that of Deterministic Select($\Theta(n)$). The reason may be that Deterministic Select requires extra time to sort the $n/5$ groups of size 5 and select the medians of the medians of the groups compared to Randomized Select.

Also, as we can see in Figure 1, the curves of Randomized Select and Deterministic Select are averagely close to a linear line. This conforms to the fact that their time complexities for the average case are both $\Theta(n)$.

6 Appendix A. Select Algorithms Implementation

select.h

```
//select.h
//Implementation of the select algorithms
#ifndef __SELECT_H__
#define __SELECT_H__
#include <cstdlib>

static void InsertionSort(int Array[],int left, int right){
  //REQUIRES: Array initialized with index x and y,
  //with elements in [-2^{31}, 2^{31}-1] and no more than 2^{31} elements
  //MODIFIES: Array
  //EFFECTS: sort the Array
  int number = right - left + 1;
  int j;
  for(int i=1; i<number; i++){
    int temp=Array[left+i];
    for(j=i; j>0 && temp<Array[left+j-1]; j--){
      //once Array[j-1]<=Array[j] stop checking since all elements before have been sorted
      Array[left+j]=Array[left+j-1];
      //copy Array[j-1] to Array[j] instead of swap everytime to save time
    }
    Array[left+j]=temp;
  }
}

static inline void swap(int Array[],int x,int y){
  //REQUIRES: Array initialized with index x and y,
  //with elements in [-2^{31}, 2^{31}-1] and no more than 2^{31} elements
  //MODIFIES: Array[x] and Array[y]
  //EFFECTS: swap Array[x] with Array[y]
  int temp=Array[x];
  Array[x]=Array[y];
  Array[y]=temp;
}

//Partition in place
static void partition_in_place(int Array[], int left, int right, int & pivot_index){
  //REQUIRES: Array initialized with with left and right as legal index
  //with elements in [-2^{31}, 2^{31}-1] and no more than 2^{31} elements
  //MODIFIES: Array
  //EFFECTS: rearrange the array to make it partitioned by a pivot
  //and return the index of the pivot
  swap(Array, left, pivot_index);
  int i=left;
  int j=right;
  int pivot=Array[left];
  while(i < j){
        while(i<j && pivot<=Array[j])
      j--;
```

```
        while(i<j && Array[i]<=pivot)
      i++;
        swap(Array,i,j);
    }
  swap(Array,left,i);
  pivot_index = i;
}

//Get a random pivot
static int RandPivot(int left, int right){
  //REQUIRES: left and right in [0,2^{31}-1]
  //EFFECTS: return a number in the range [left, right]
  int result = (rand() % (right - left) ) + left;
  return result;
  //here returns a random index in the range as the pivot index
}

//Randomized Select
int RandomizedSelect(int Array[],int left, int right, int i){
  //REQUIRES: Array of integers initialized with index [left,right]
  //EFFECTS: return the (i+1)-th smallest element of the array
  if (left==right)
    return Array[left];
  int pivot_index = RandPivot(left, right);
  partition_in_place(Array, left, right, pivot_index);
  int number = pivot_index - left + 1; // size of the left subarray
  if(i== (number - 1))
    return Array[pivot_index];
  else if(i < (number-1))
    return RandomizedSelect(Array, left, pivot_index-1, i);
  else return RandomizedSelect(Array, pivot_index+1, right, i - number);
}

//Deterministic Select
int DeterministicSelect(int Array[],int left, int right, int i){
  //REQUIRES: Array of integers initialized with index [left,right]
  //EFFECTS: return the (i+1)-th smallest element of the array
  if (left==right)
    return Array[left];
  int index = left;
  int num = (right - left + 1 + 5) / 5; //number of groups
  int * C = new int [num];
  int cnt;
  for(cnt = 0; cnt < num-1; cnt ++){
    InsertionSort(Array, index, index+4);
    C[cnt] = Array[index+2];
    index += 5;
  }
  InsertionSort(Array, index, right);
  C[cnt] = ((index+2)<=right) ? Array[index+2] : Array[right]; //The last median
  int pivot = DeterministicSelect(C, 0, num-1, num/2); // find out the median of medians
  delete[] C;
  for(index = left; Array[index]!=pivot; index++){} //'index' is the index of pivot
  partition_in_place(Array, left, right, index);
  if (i == (index - left))
    return pivot;
  else if (i < (index - left) )
    return DeterministicSelect(Array, left, index-1, i);
  else
    return DeterministicSelect(Array, index+1, right, i - (index - left + 1));
}


#endif
```

7 Appendix B. Quicksorting Implementation

QSort.h

```
//QSort.h
//Implementation of in-place quicksorting
#ifndef __QSORT_H__
#define __QSORT_H__
#include <cstdlib>

//Quick Sorting
static int getpivot(int left, int right){
  //REQUIRES: left and right in [0,2^{31}-1]
  //EFFECTS: return a number in the range [left, right]
  int result = (rand() % (right - left) ) + left;
  return result; //here returns a random index in the range as the pivot index, this can be
    modified for a better performance
}

//Partition in place
static int partition_in_place( int Array[], int left, int right){
  //REQUIRES: Array initialized with with left and right as legal index
  //with elements in [-2^{31}, 2^{31}-1] and no more than 2^{31} elements
  //MODIFIES: Array
  //EFFECTS: rearrange the array to make it partitioned by a pivot
  int pivot_index=getpivot(left, right);
  swap(Array, left, pivot_index);
  int i=left;
  int j=right;
  int pivot=Array[left];
  while(i < j){
        while(i<j && pivot<=Array[j])
      j--;
        while(i<j && Array[i]<=pivot)
      i++;
        swap(Array,i,j);
    }
  swap(Array,left,i);
  return i;
}

static void QuickSortHelper(int Array[], int left, int right, int (*partition)(int[], int,
    int)) {
  //REQUIRES: Array initialized with left and right as legal index
  //with elements in [-2^{31}, 2^{31}-1] and no more than 2^{31} elements
  //MODIFIES: Array
  //EFFECTS: recursively quicksort the array
  int pivotat; // index of the pivot
  if(left >= right) return;
  pivotat = partition(Array, left, right);
  QuickSortHelper(Array, left, pivotat-1, partition);
  QuickSortHelper(Array, pivotat+1, right, partition);
  }


void QuickSort(int Array[], int number){
  //REQUIRES: Array initialized with at least 'number' elements
  //with elements in [-2^{31}, 2^{31}-1] and no more than 2^{31}-1 elements, number in
    [0,2^{31}-1]
  //MODIFIES: Array
  //EFFECTS: sort the Array
  QuickSortHelper(Array, 0, number-1, partition_in_place);
}

#endif
```

8 Appendix C. Testing different algoroithms

compare.cpp

```
//compare.cpp
```

```cpp
//Generate 10, 100 ,1000, 10000, 100000 and 1000000 random integers in the range [-2^(31),
    2^(31)-1] and 5 random index
//Select the elements with index and measure the average time cost for the 3 algorithms.
//The output is written to a file called "CompareResult.txt".
#include <fstream>
#include <cstdlib>
#include <ctime>
#include <iostream>
#include <string>
#include "select.h"
#include "QSort.h"
using namespace std;

const int MAXSIZE=1000000;  //the maximum size of the arrays to be tested
const int MINSIZE=10;   //the minimum size of the arrays to be tested
const int STEP=10;
const int METHODSNUMBER=3;
const int INDEXNUMBER=5;
const string METHOSNAMES[3]={"RSelect", "DSelect","SortSelect"};

int RSelect(int Array[], int n, int i){
  return RandomizedSelect(Array, 0, n-1, i);
}

int DSelect(int Array[], int n, int i){
  return DeterministicSelect(Array, 0, n-1, i);
}

int SortSelect(int Array[], int n, int i){
  QuickSort(Array, n);
  return Array[i];
}

int main(){
  ofstream output;
  output.open("CompareResult.txt");
  int * integers= new int [MAXSIZE];
  int * auxiliary= new int [MAXSIZE];
  clock_t start, end;
  double time[METHODSNUMBER]={0,0,0};
  int (*select[METHODSNUMBER])(int[], int,int)={RSelect, DSelect, SortSelect};
  int index[INDEXNUMBER];
  for(int i=MINSIZE; i<=MAXSIZE; i*=STEP){
    for(int j=0;j<i;j++){
      integers[j]=auxiliary[j]=mrand48();
    }
    for(int j=0; j<INDEXNUMBER; j++){
      index[j]= rand()%i; //randomly generate the index of the wanted elements
    }
    for(int choice=0; choice<METHODSNUMBER; choice++){
      for(int j=0; j<INDEXNUMBER; j++){
        start = clock();
        select[choice](integers, i, index[j]);
        end = clock();
        time[choice] += (double) (( end - start )* 1000.0 / CLOCKS_PER_SEC);
        for(int k=0;k<i;k++){
          integers[k]=auxiliary[k];  //change the array back
        }
      }
      time[choice] /= INDEXNUMBER;
      output<<METHOSNAMES[choice]<<" with size "<<i<<": "<<time[choice]<<"ms"<<endl;
      time[choice] = 0;
    }
  }
  delete[] integers;
  output.close();
}
```