VE281 Sorting Algorithms Comparison Report

Xinhao Liao 516370910037

September 2018

1 Introduction

There are various algorithms used to sort an array of integers. Among them, Bubble Sorting, Insertion Sorting, Selection Sorting, Merge Sorting and Quicksorting are all commonly used comparison sorting algorithms. In this report, these algorithms are to be compared with randomly generated integer arrays. (Quicksoting is respectively implemented with and without $\Omega(n)$ cost of space.)

2 Background

Consider an array A with n integers as elements. The array can be sorted with following algorithms.

Bubble Sorting

Compares two adjacent items and swap them to keep them in ascending order. From the beginning to the end, compare two adjacent items and swap them to keep them in ascending order. The last item will be the largest one. Then similarly find the second largest element, and so on. The time complexity should be $\Theta(n^2)$ for the best case, the average case, and the worst case. The C++ code for this algorithm can be found in Appendix A as the function *BubbleSort*.

Insertion Sorting

Notice that $A[0]$ alone is a sorted array. For i=1 to N-1, insert $A[i]$ into the appropriate location in the sorted array with the first $i-1$ elements, so that the first i elements are sorted. To do so, save $A[i]$ in a temporary variable *temp*, shift sorted elements greater than *temp* right, and then insert *temp* in the gap. The time complexity should be $\Theta(n)$ for the best case, and $\Theta(n^2)$ for the average case and the worst case. The C++ code for this algorithm can be found in Appendix A as the function *InsertionSort*.

Selection Sorting

For $i = 0$ to $n-2$, find the smallest item in the array $A[i], \ldots, A[n-1]$. Then, swap that item with A[i]. Finding the smallest item requires linear scan. The time complexity should be $\Theta(n^2)$ for the best case, the average case, and the worst case. The C++ code for this algorithm can be found in Appendix A as the function *SelectionSort*.

Merge Sorting

Spilt array into two (roughly) equal subarrays. Merge sort each subarray recursively. Then the two subarrays will be sorted. Finally, recursively merge the two sorted subarrays back into a sorted array. The time complexity should be $\Theta(nlogn)$ for the best case, the average case, and the worst case. The C++ code for this algorithm can be found in Appendix A as the function *MergeSort*, which uses a function *MergeSortHelper* to help make it recursive, and a function *Merge* to help merge subarrays back.

Quick Sorting

First, choose an array element as pivot. Put all elements less than pivot to the left of pivot. Put all elements greater than or equal to pivot to the right of pivot. Move pivot to its correct place in the array. Finally, sort left and right subarrays recursively (not including pivot).

The time complexity should be $\Theta(nlogn)$ for the best case and the average case, and $\Theta(n^2)$ for the worst case. The C++ code for this algorithm can be found in Appendix A as the function *QuickSort1* and *QuickSort2*, which uses a function *QuickSortHelper* to help make it recursive and a function *getpivot* to generate a pivot index. For *QuickSort1*, it used a function *partition_extra_place* to sort with $\Omega(n)$ space cost. And for *QuickSort2*, it used a function *partition_in_place* to sort, which costs on average only $\Theta(logn)$ stack space, which makes it weakly in-place.

Summary

	Worst Case Time	Average Case Time	In Place	Stable
Insertion	$O(N^2)$	$O(N^2)$	Yes	Yes
Selection	$O(N^2)$	$O(N^2)$	Yes	No
Bubble	$O(N^2)$	$O(N^2)$	Yes	Yes
Merge Sort	$O(N \log N)$	$O(N \log N)$	No	Yes
Quick Sort	$O(N^2)$	$O(N \log N)$	Weakly	No

Figure 1: Comparison sorting algorithms summary.

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. Sort the array with the algorithms in order and measure the 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.
4. Multiply i with 10 and repeat steps 2 to 3 until i reaches 100000.

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

4 Result

Size	10	100	1000	10000	100000
Bubble Sort	0.004ms	0.123ms	5.721ms	777.441ms	80180.9ms
Insertion Sort	0.001ms	0.017ms	1.918ms	146.042ms	14213.4ms
Selection Sort	0.002ms	0.08ms	3.105ms	289.307ms	27408ms
Merge Sort	0.006ms	0.023ms	0.343ms	3.57ms	42.832ms
Quicksort not in place	0.004ms	0.048ms	0.392ms	4.19ms	50ms
Quicksort in place	0.002ms	0.019ms	0.238ms	3.167ms	36.442ms

Table 1: Result for sorting with different algorithms.

The results for sorting an array of 10, 100, 1000, 10000, and 100000 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 six curves corresponding to the six sorting algorithms in the same figure in Figure 1. The curves are plotted in log scales.

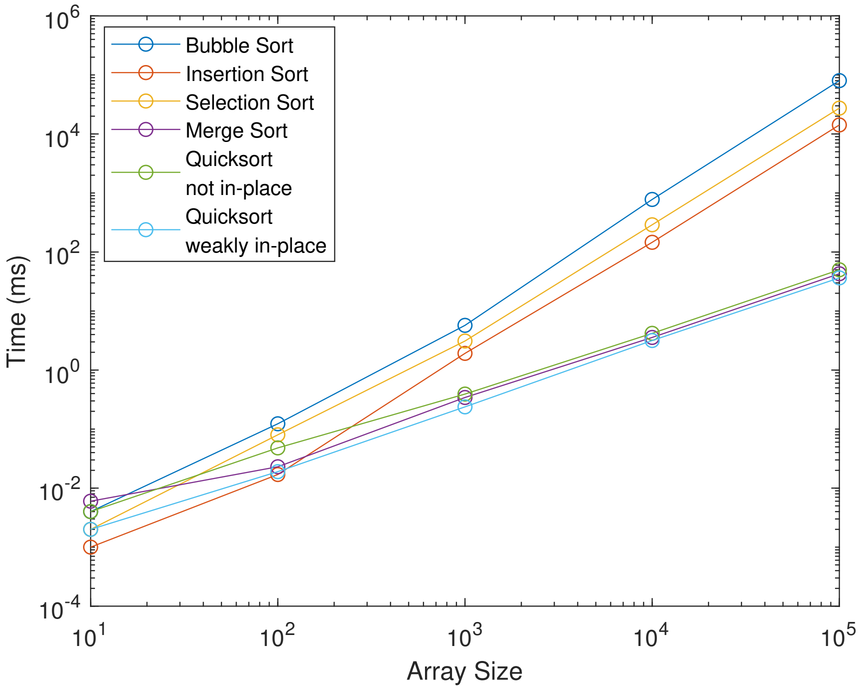

Figure 2: Curves corresponding to the six sorting algorithms.

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, Bubble Sorting, Insertion Sorting, and Selection Sorting on average have far worse performance than Merge Sorting and Quick Sorting. In general, Quicksorting in-place has the best performance. Quicksoting weakly-in-place is the second best, Merge Sorting the third, Insertion Sorting the fourth, Selection the fifth, and Bubble Sorting the last one. And when the size is small enough, Insertion sorting shows the best performance.

5 Discussion and analysis

The performance of Bubble Sorting, Insertion Sorting, and Selection Sorting is generally worse than that of Merge Sorting and Quick Sorting. This conforms to the theoretical analysis that Bubble Sorting, Insertion Sorting, and Selection Sorting have the time cost of $\Theta(n^2)$ in the average case, which are far worse than that of Merge Sorting and Quick Sorting, which are $\Theta(nlogn)$.

Among the 3 algorithms with average time cost of $\Theta(n^2)$, Bubble Sorting has the worst performance. Bubble Sorting, and Insertion Sorting has the best performance. This is reasonable, since in the best case, the time complexity of Insertion Sorting is $\Omega(n)$ which is better than all the other algorithms here. Insertion Sorting only needs to compare and swap when some order is not ascending. Once a comparison gives a correct answer, all the elements before doesn't need to be checked or swapped any more. And for Bubble Sorting, whatever order is given, it needs $\Theta(n^2)$ comparisons and can only swap with the neighboring element, which implies swapping $O(n^2)$ times. And for Selection sorting, it's similar to Bubble Sorting except that it only needs $\Theta(n)$ swaps. Comparing Insertion Sorting with Selection Sorting, Insertion Sorting may terminate in advance, and that may be the reason why it has better performance. However, notice that if Insertion Sorting is implemented by swapping the compared neighbors, rather than just copying the former ones as shown in Appendix B, Insertion Sorting may have worse performance than Selection Sorting. This is because a swap implies three assignments, which can make the Insertion Sorting cost much more.

Among the algorithms with average time cost of $\Theta(nlogn)$, as we can see, Quicksorting in place has the best performance, and Quicksorting not in place has the worst performance. The reason may be that reading and writing in memory are quite slow. And compared with Merge Sorting, though they both require $\Omega(n)$ extra place, Merge Sorting avoids the worst case of Quicksorting, which partitions only 1 element, by always partitioning in half.

Also notice that when the size is small enough, the Insertion Sorting algorithm shows the best performance, even better than the algorithms with $\Theta(nlogn)$ average time complexity. This is reasonable since in the best case, the time complexity of Insertion Sorting is only $\Theta(n)$, which is the best time complexity, and Insertion Sorting may terminate in advance in some cases. And when the size is small, there is more chance that it's the case that the algorithm can terminate in advance. There is always 1 best case, and there are $n!$ cases for an array of size n. The probability of the best case is $\frac{1}{n!}$, which increases quickly as n decreases.

6 Appendix A. Algorithms Implementation

sort.h

```
//sort.h
//Implementation of the algorithms
#ifndef __SORT_H__
#define __SORT_H__
#include <cstdlib>
static inline void swap(long int Array[],long int x,long 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]
  long int temp=Array[x];
  Array[x]=Array[y];
  Array[y]=temp;
}


//Bubble Sorting
void BubbleSort(long int Array[],long int number){
  //REQUIRES: Array initialized with at least 'number' elements
  //with elements in [-2^{31}, 2^{31}-1] and no more than 2^{31} elements,
  //number in [0,2^{31}-1]
  //MODIFIES: Array
  //EFFECTS: sort the Array
  for(long int i=number-1; i>0; i--){
    for(long int j=0; j<i; j++){
      if(Array[j]>Array[j+1])
        swap(Array, j, j+1);
    }
  }
}


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


//Selection Sorting
```

```
void SelectionSort(long int Array[], long 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
  long int min;
  for(long int i=0; i<(number-1); i++){
    min = i;
    for(long int j=i; j<number; j++){
      if(Array[j]<Array[min])
        min=j;
    }
    swap(Array, i, min);// similar to Bubble Sorting,
    //but instead of swaping every time only swap the minimum one
  }
}


//Merge Sorting
static void Merge(long int Array[], long int left, long int mid, long int right){
  //REQUIRES: Array initialized with left mid and right as legal index
  //with elements in [-2^{31}, 2^{31}-1] and no more than 2^{31} elements
  //MODIFIES: Array
  //EFFECTS: Merge the array with elements [left:mid]
  //and the array with elements [mid+1:right]
  long int i = left;
  long int j = mid+1;
  long int k =0;
  long int * Auxiliary = new long int [right - left + 1];
  //initizalize with new since there is a size limit for a static array
  while(i <= mid && j<= right){
    if(Array[i] <= Array[j])
      Auxiliary[k++]=Array[i++];
    else
      Auxiliary[k++]=Array[j++];
  }
  while(k <= (right - left)){
    if(i>mid)
      Auxiliary[k++]=Array[j++];
    else
      Auxiliary[k++]=Array[i++];
  }
  for(i=left; i<=right; i++){
    Array[i]=Auxiliary[i-left];
  }
  delete[] Auxiliary;
}

static void MergeSortHelper(long int Array[], long int left, long int right){
  //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 merge sort the array
  if (left >= right) return;
  long int mid = (left+right)/2;
  MergeSortHelper(Array, left, mid);
  MergeSortHelper(Array, mid+1, right);
  Merge(Array, left, mid, right);
}

void MergeSort(long int Array[], long 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
  MergeSortHelper(Array, 0, number-1);
}
```

```
//Quick Sorting
static long int getpivot(long int left, long int right){
  //REQUIRES: left and right in [0,2^{31}-1]
  //EFFECTS: return a number in the range [left, right]
  long 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 with extra place
static long int partition_extra_place(long int Array[], long int left, long 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
  long int partition_index = getpivot(left, right);
  long int * Auxiliary = new long int [right - left + 1];
  long int i,j,k;
  long int pivot=Array[partition_index];
  for(i=left,j=0,k=right-left; i<=right; i++){
    if(i==partition_index) continue;
    else if(Array[i]<=pivot)
      Auxiliary[j++]=Array[i];
    else
      Auxiliary[k--]=Array[i];
  }
  Auxiliary[j]=pivot;
  for(i=left; i<=right; i++){
    Array[i]=Auxiliary[i-left];
  }
  delete[] Auxiliary;
  return left+j;
}

//Partition in place
static long int partition_in_place(long int Array[], long int left, long 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
  long int pivot_index=getpivot(left, right);
  swap(Array, left, pivot_index);
  long int i=left;
  long int j=right;
  long 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(long int Array[], long int left, long int right, long int (*
    partition)(long int[], long int, long 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
  long 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 QuickSort1(long int Array[], long 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_extra_place);
}

void QuickSort2(long int Array[], long 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
```

7 Appendix B. Testing different algorothms

compare.cpp

```
//compare.cpp
//Generate 10, 100 ,1000, 10000, and 100000 random integers in the range [-2^(31), 2^(31)-1]
//Sort them with different algorithms and measure the time cost.
//The output is written to a file called "CompareResult.txt".
#include <fstream>
#include <cstdlib>
#include <ctime>
#include <iostream>
#include <string>
#include "sort.h"
using namespace std;

const long int MAXSIZE=100000;  //the maximum size of the arrays to be tested
const long int MINSIZE=10;     //the minimum size of the arrays to be tested
const long int STEP=10;
const int METHODSNUMBER=6;
const string METHOSNAMES[6]={"Bubble Sort", "Insertion Sort", "Selection Sort",
            "Merge Sort", "Quick Sort Not-in-place", "Quick Sort In-place"};

int main(){
  ofstream output;
  output.open("CompareResult.txt");
  long int * integers= new long int [MAXSIZE];
  long int * auxiliary= new long int [MAXSIZE];
  clock_t start, end;
  void (*sort[METHODSNUMBER])(long int[], long)={BubbleSort, InsertionSort, SelectionSort,
            MergeSort, QuickSort1, QuickSort2};
  for(long int i=MINSIZE; i<=MAXSIZE; i*=STEP){
    for(long int j=0;j<i;j++){
      integers[j]=auxiliary[j]=mrand48();//save a copy in auxiliary
    }
    for(int choice=0; choice<METHODSNUMBER; choice++){
      start = clock();
      sort[choice](integers, (long)i);
      end = clock();
      output<<METHOSNAMES[choice]<<" with array size of "<<i<<" costs"
          <<(double)((end-start)*1000.0/CLOCKS_PER_SEC)<<"ms."<<endl;
      for(long int j=0;j<i;j++){
      //write back the saved numbers in order
      //to make it in the same order for every algorithm
        integers[j]=auxiliary[j];
      }
    }
  }
  output.close();
```

```
  delete[] auxiliary;
  delete[] integers;
}
```