Programming assignment 3

Sorting algorithms analysis

Zhihao Cao

Spring 2014

CSCI 36200

Index

Description

The objectives of this project are to get experience on coding different kinds of sorting algorithms and analyze their running time. Besides, it improves our programming skills on the usage of different kind of data like arrays, vectors and pointers. It also let us gets familiar with some challenging functions like unique random number generator, nanoseconds timer.

Insertion sort

It is an efficient algorithm for sorting a small number of elements, although its complexity is $O(n^2)$.
The data is sorted by repeatedly taking the next item and inserting it into the final data structure in its proper order with respect to items already inserted.

Quick sort

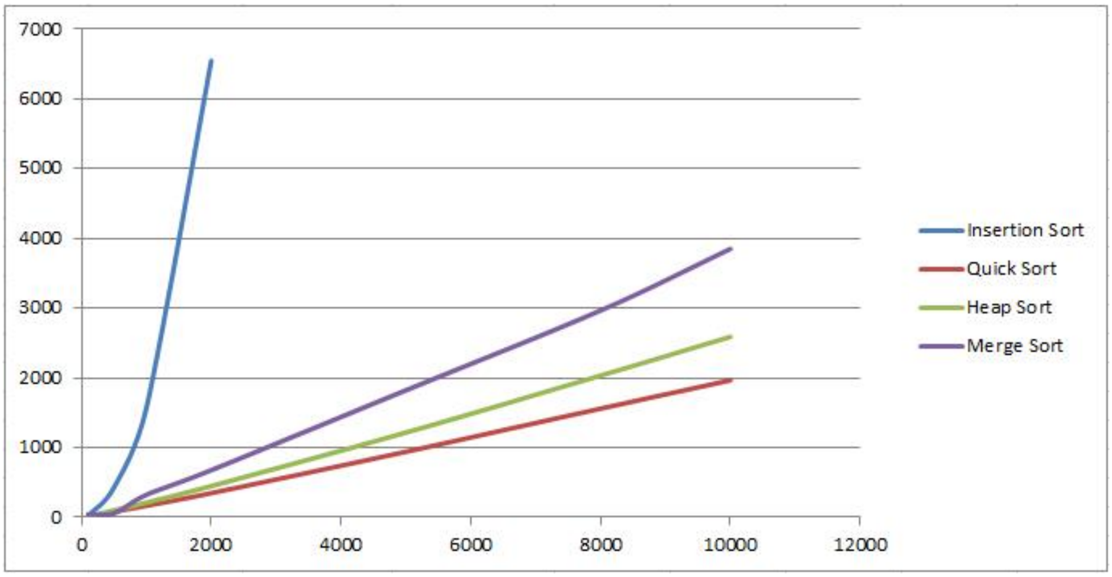
It is a fast sorting algorithm. On the average, it has O(n log n) complexity, making quicksort suitable for sorting big data volumes.
The divide-and-conquer strategy is used in quicksort. Below the recursion step is described:

1. Choose a pivot value.
2. Partition.
3. Sort both parts.

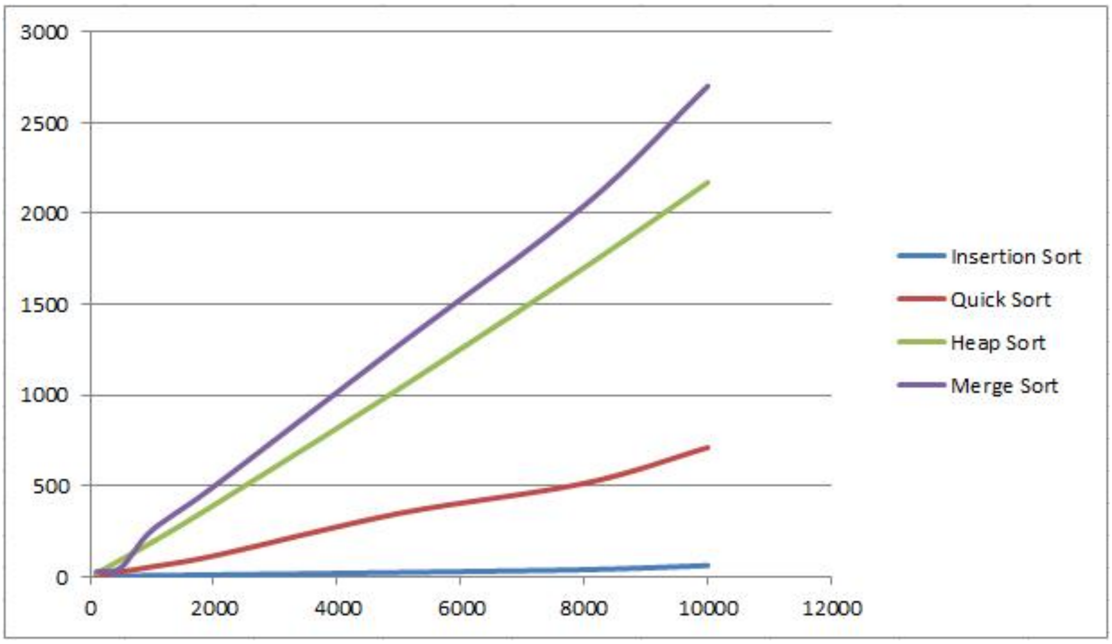
Heap sort

It is a comparison based sorting technique based on Binary Heap data structure. It is similar to selection sort where we first find the maximum element and place the maximum element at the end.
It follows these steps:
1. Build a max heap from the input data.
2. At this point, the largest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of tree.
3. Repeat above steps until size of heap is greater than 1.

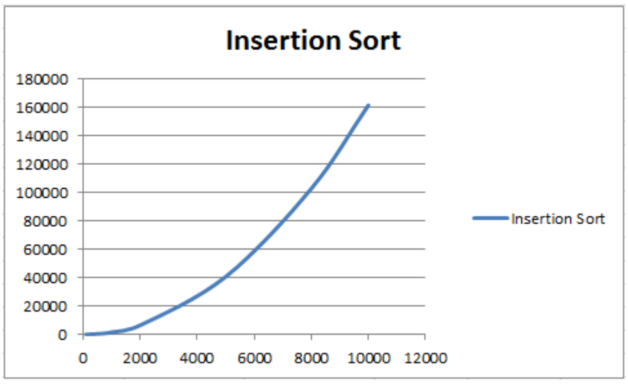
Merge sort
It is based on the divide-and-conquer paradigm.
It works as follows:
Divide the unsorted list into n sublists, each containing 1 element (a list of 1 element is considered sorted).
Repeatedly merge sublists to produce new sorted sublists until there is only 1 sublist remaining. This will be the sorted list.

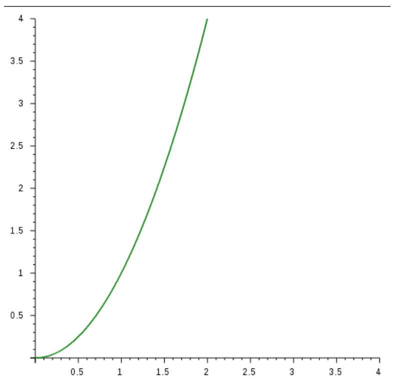


Unsorted arrays

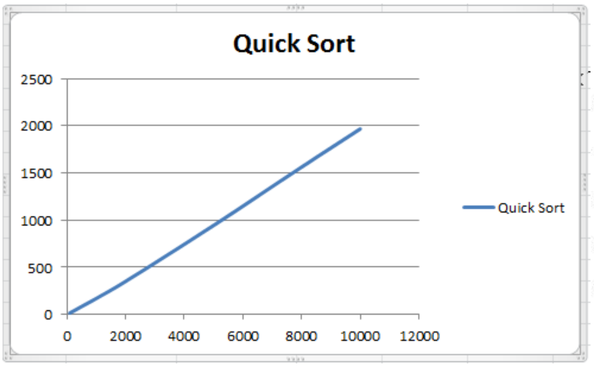


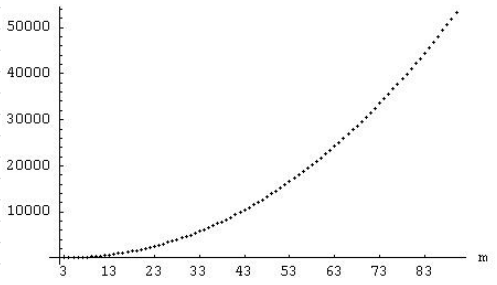
Sorted arrays



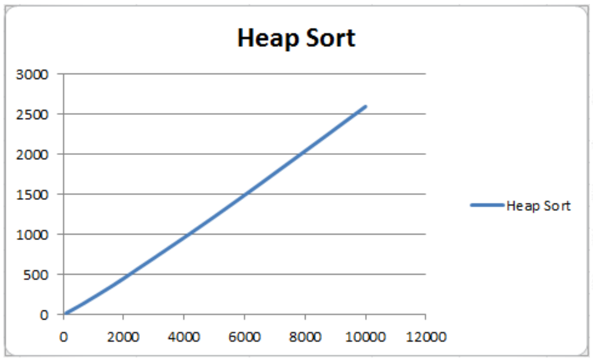


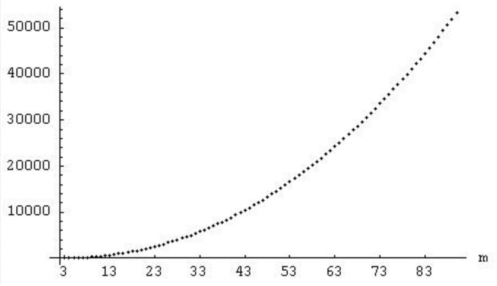
Insertion sort has O(n^2) complexity.



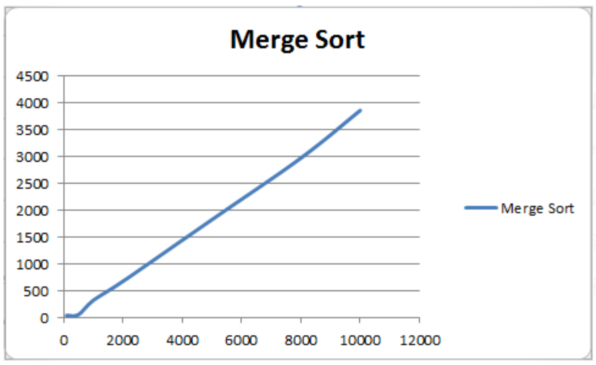


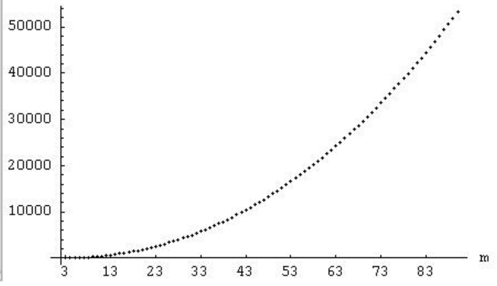
Quick sort has O(nlogn) complexity.





Heap sort also has O(nlogn) complexity.





Merge sort also has O(nlogn) complexity but with a little curve in the very first.

Source code

```
// Random_number_generator.cpp : Defines the entry point for the console
application.
//Author:         Zhihao Cao
//Last modify:     2014/4/10
//Subject:        CSCI 36200

#include "stdafx.h"
#include <stdlib.h>
#include <iostream>
#include <algorithm>
#include <vector>
#include <fstream>
#include <conio.h>
#include <Windows.h>
using namespace std;

double PCFreq = 0.0;//for timer
__int64 CounterStart = 0;// for timer

vector<int> n = { 100, 500, 1000, 2000, 5000, 8000, 10000 }; //Different
input size
const int N = 20000; //random scale
int n0[100], n1[500], n2[1000], n3[2000], n4[5000], n5[8000], n6[10000];
//initialize input arrays


void StartCounter() // Define a timer start function
{
    LARGE_INTEGER li;
    if (!QueryPerformanceFrequency(&li))
        cout << "QueryPerformanceFrequency failed!\n";

    PCFreq = double(li.QuadPart) / 1000000.0;

    QueryPerformanceCounter(&li);
    CounterStart = li.QuadPart;
}

double GetCounter() // Define a timer end function. return running time
in nanoseconds.
{
    LARGE_INTEGER li;
    QueryPerformanceCounter(&li);
    return double(li.QuadPart - CounterStart) / PCFreq;
}



void merge_helper(int *input, int left, int right, int *scratch)
{
    /* base case: one element */
    if (right == left + 1)
    {
        return;
    }
    else
    {
        int i = 0;
        int length = right - left;
        int midpoint_distance = length / 2;
        /* l and r are to the positions in the left and right subarrays
*/
        int l = left, r = left + midpoint_distance;

        /* sort each subarray */
        merge_helper(input, left, left + midpoint_distance, scratch);
        merge_helper(input, left + midpoint_distance, right, scratch);

        /* merge the arrays together using scratch for temporary storage
*/
        for (i = 0; i < length; i++)
        {
            if (l < left + midpoint_distance &&
                (r == right || max(input[l], input[r]) == input[r]))
            {
                scratch[i] = input[l];
                l++;
            }
            else
            {
                scratch[i] = input[r];
                r++;
            }
        }
        //Copy the sorted subarray back to the input
```

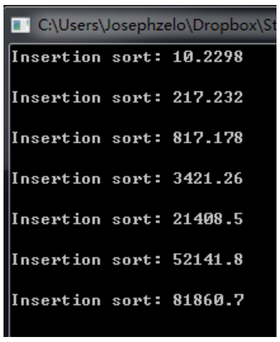
```
            for (i = left; i < right; i++)
            {
                input[i] = scratch[i - left];
            }
        }
    }

    void merge_sort(int *input, int size)
    {
        StartCounter(); //start timer

        int *scratch = (int *)malloc(size * sizeof(int));
        if (scratch != NULL)
        {
            merge_helper(input, 0, size, scratch);
            free(scratch);
        }

        cout << "Merge sort: " << GetCounter() << "\n" << endl; //end timer


    }

    void max_heapify(int *a, int i, int n) //Build a max heapify.
    {
        int j, temp;
        temp = a[i];
        j = 2 * i;
        while (j <= n)
        {
            if (j < n && a[j + 1] > a[j])
                j = j + 1;
            if (temp > a[j])
                break;
            else if (temp <= a[j])
            {
                a[j / 2] = a[j];
                j = 2 * j;
            }
        }
        a[j / 2] = temp;
        return;
    }
```

```cpp
void heapsort(int *a, int n)
{
    int i, temp;
    for (i = n; i >= 2; i--)
    {
        temp = a[i];
        a[i] = a[1];
        a[1] = temp;
        max_heapify(a, 1, i - 1);
    }
}

void build_maxheap(int *a, int n)
{
    int i;
    for (i = n / 2; i >= 1; i--)
    {
        max_heapify(a, i, n);
    }
}

void heap_sort(int *a, int n){
    StartCounter(); //start timer

    build_maxheap(a, n);
    heapsort(a, n);

    cout << "Heap sort: " << GetCounter() << "\n" << endl; //end timer
}

void quick_sort_helper(int arr[], int left, int right)
{



    int i = left, j = right;
    int tmp;
    int pivot = arr[(left + right) / 2];

    /* partition */
    while (i <= j) {
        while (arr[i] < pivot)
            i++;
        while (arr[j] > pivot)
            j--;
        if (i <= j) {
            tmp = arr[i];
            arr[i] = arr[j];
            arr[j] = tmp;
            i++;
            j--;
        }
    }
    /* recursion */
    if (left < j)
        quick_sort_helper(arr, left, j);
    if (i < right)
        quick_sort_helper(arr, i, right);


}

void quick_sort(int arr[], int left, int right) {
    StartCounter(); //start timer
    quick_sort_helper(arr, left, right);
    cout << "Quick sort: " << GetCounter() << "\n" << endl; //end timer
}

void insertion_sort(int arr[], int length){
    StartCounter(); //start timer

    int j, temp;

    for (int i = 0; i < length; i++){
        j = i;

        while (j > 0 && arr[j] < arr[j - 1]){
            temp = arr[j];
            arr[j] = arr[j - 1];
            arr[j - 1] = temp;
            j--;
        }
    }

    cout << "Insertion sort: " << GetCounter() << "\n" << endl; //end timer
}

void writeToFile(int array[], string fileName, int length) //output
necessary data.
```

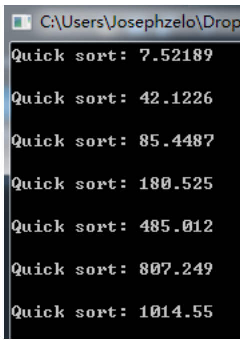
```
{
    ofstream myfile;
    myfile.open(fileName);
    for (int i = 0; i < length; ++i)
    {
        myfile << array[i] << endl;
    }
    myfile.close();
}

bool isAlreadyAdded(int value, int index, int *pointer) // check if
generated integer already in array.
{
    for (int i = 0; i < index; i++)
    {
        if (*pointer == value)
            return true;
        pointer++;
    }
    return false;
}

void generator_code(int *p, int *p2, int input_size)  //Generate numbers
according to the input size.
{
    for (int x = 0; x != input_size; ++x) {

        int tmp = 1 + (rand() % N);
        while (x != 0 && isAlreadyAdded(tmp, x, p2))
            tmp = 1 + (rand() % N);
        *p = tmp;
        p++;
    }
}

void generator(int size){

    int input_size = n[size];
    int *p, *p2; //create a pointer point to the arrays which we want to
manipulate.

    switch (input_size) //use different cases to decide what size of array
is going to deal with.
    {
    case 100:
        p = n0;
        p2 = n0;
        generator_code(p, p2, input_size);
        break;

    case 500:
        p = n1;
        p2 = n1;
        generator_code(p, p2, input_size);
        break;
    case 1000:
        p = n2;
        p2 = n2;
        generator_code(p, p2, input_size);
        break;
    case 2000:
        p = n3;
        p2 = n3;
        generator_code(p, p2, input_size);
        break;
    case 5000:
        p = n4;
        p2 = n4;
        generator_code(p, p2, input_size);
        break;
    case 8000:
        p = n5;
        p2 = n5;
        generator_code(p, p2, input_size);
        break;
    case 10000:
        p = n6;
        p2 = n6;
        generator_code(p, p2, input_size);
        break;

    default:
        cout << "Invalid input_size" << endl;
    } //end swith cases.
}
```

```
int main() {

    for (int i = 0; i < n.size(); ++i){
        generator(i); // Parameter is the input size
    }

    //Uncomment one sorting algorithm at a time.
    //Default, we are going to sort the arrays by using Insertion sort
algorithm.

    insertion_sort(n0, 100);
    insertion_sort(n1, 500);
    insertion_sort(n2, 1000);
    insertion_sort(n3, 2000);
    insertion_sort(n4, 5000);
    insertion_sort(n5, 8000);
    insertion_sort(n6, 10000);

    /*
    quick_sort(n0, 0, 99);
    quick_sort(n1, 0, 499);
    quick_sort(n2, 0, 999);
    quick_sort(n3, 0, 1999);
    quick_sort(n4, 0, 4999);
    quick_sort(n5, 0, 7999);
    quick_sort(n6, 0, 9999);
    */

    /*
    heap_sort(n0, 100);
    heap_sort(n1, 500);
    heap_sort(n2, 1000);
    heap_sort(n3, 2000);
    heap_sort(n4, 5000);
    heap_sort(n5, 8000);
    heap_sort(n6, 10000);
    */

    /*
    merge_sort(n0, 100);
    merge_sort(n1, 200);
    merge_sort(n2, 1000);
    merge_sort(n3, 2000);
    merge_sort(n4, 5000);
    merge_sort(n5, 8000);
    merge_sort(n6, 10000);
    */

    //Output a specific array.
    //writeToFile(n0, "n0", sizeof(n0) / sizeof(n0[0]));


    cin.ignore();
    return 0;
}

int _tmain(int argc, _TCHAR* argv[])
{
    return 0;
}
```

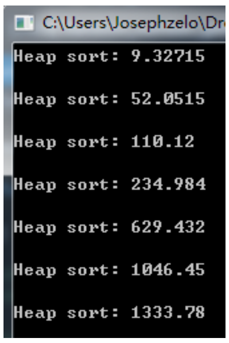
Output



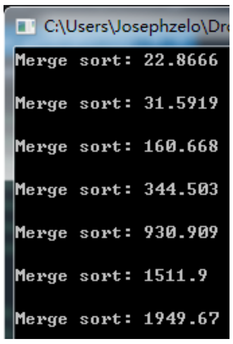
```
C:\Users\Josephzelo\Dropbox\St
Insertion sort: 10.2298
Insertion sort: 217.232
Insertion sort: 817.178
Insertion sort: 3421.26
Insertion sort: 21408.5
Insertion sort: 52141.8
Insertion sort: 81860.7
```



```
C:\Users\Josephzelo\Drop
Quick sort: 7.52189
Quick sort: 42.1226
Quick sort: 85.4487
Quick sort: 180.525
Quick sort: 485.012
Quick sort: 807.249
Quick sort: 1014.55
```



```
C:\Users\Josephzelo\Dr
Heap sort: 9.32715
Heap sort: 52.0515
Heap sort: 110.12
Heap sort: 234.984
Heap sort: 629.432
Heap sort: 1046.45
Heap sort: 1333.78
```



```
C:\Users\Josephzelo\Dr
Merge sort: 22.8666
Merge sort: 31.5919
Merge sort: 160.668
Merge sort: 344.503
Merge sort: 930.909
Merge sort: 1511.9
Merge sort: 1949.67
```

Conclusion

Through this project, we understand why insertion sort is not suitable to sort big number of data. Quick sort is the most efficient way among these 4 algorithms. Also, it is the most practical sorting algorithm. Heap sort is also quick but not stable. Merge sort cause more memory then other algorithms.