# VE281 Homework I

## Performance Analysis for Sort Algorithms

**Name:** Tianyi Ge **Stu Number:** 516370910168

## 1 Introduction

In this report, we discuss the time complexity of six types of different sort algorithms including Bubble sort, Insertion sort, Selection sort, Merge sort and two types of Quick sorts, by testing their runtime with input in different sizes. The graph of time-size relationship intuitively demonstrates the time complexity, thus confirming our theoretical expectation.

Furthermore, this report entails several extreme situations like an integer array in descending order or an array of merely 1's. The result, interestingly, reflects the capacity of those six algorithms under different circumstances.

Since the time complexity of $O(n^2)$ and $O(n \log n)$ differs enormously with the increasing of input size, we separate the analysis into two parts and discuss respectively.

## 2 Performance analysis on $O(n^2)$ Sort Algorithms

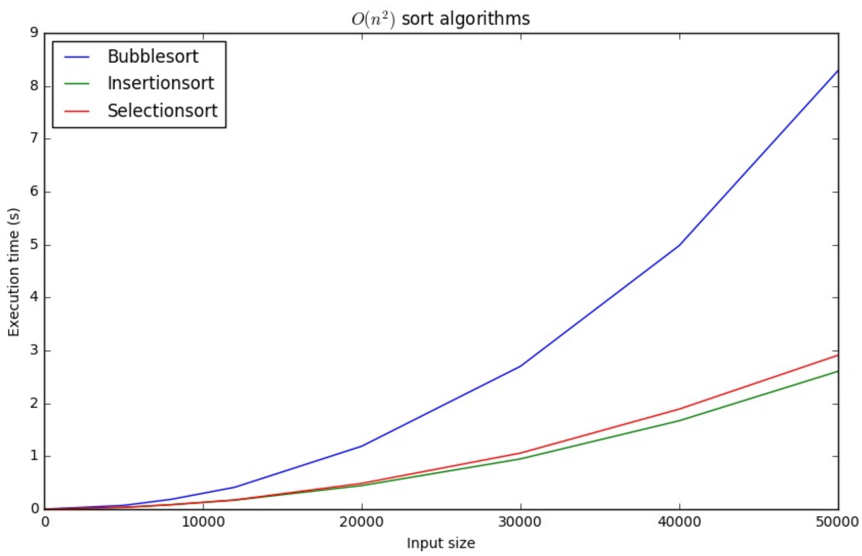


Figure 1: Performance of $O(n^2)$ sort algorithms

In Fig. 1, the curves are in shape of quadratic functions, among which the Bubble sort obtains a obvious larger constant. The runtime is considerably increasing non-linearly after the input size exceeds 25000.

# 3 Performance analysis on $O(n \log n)$ Sort Algorithms

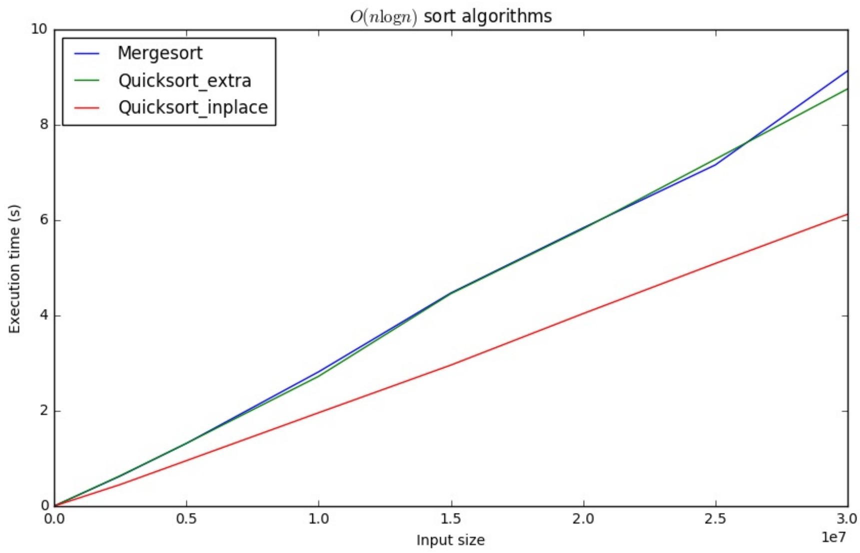


Figure 2: Performance of $O(n \log n)$ sort algorithms

In Fig. 3, the curves are almost linear, where the in-place Quick sort shows an undebatable edge. It's hard to tell whether it's $O(n \log n)$ or $O(n)$ because the input size is not large enough to give credit to the expectation. The distinction between two different Quick sort is intriguing. They followed almost the same procedure whereas a big gap occurred when even the input data is random. This reflects It also indicates that the randomized Quick sort works remarkably, even compared to other steady $O(n \log n)$ algorithms like Merge sort.

# 4 Extreme Input Situations

## 4.1 An Array in Descending Order

For descending data, we can see that Quick sort in-place is still advantageous, although all of them become more efficient. One possible explanation is that the descending data simultaneously reduces the cost on comparison between two elements.

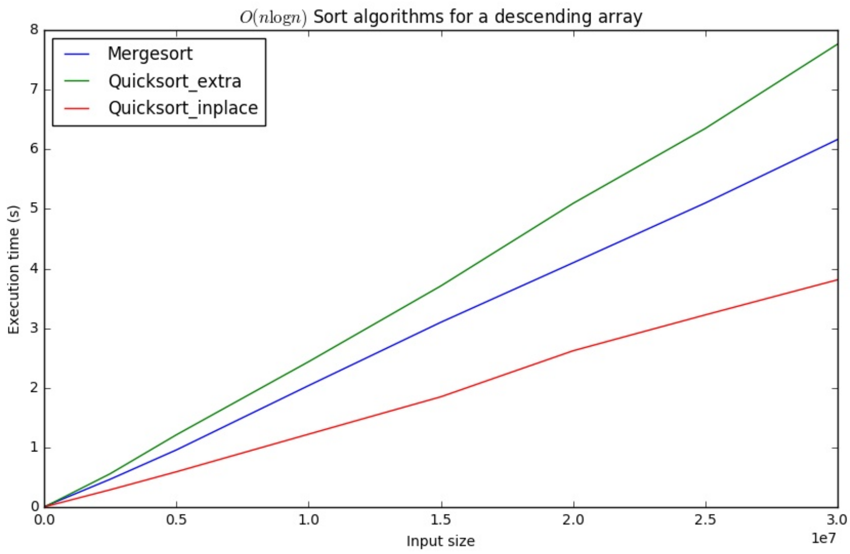


Figure 3: Performance of $O(n \log n)$ sort algorithms

# A Source Codes

Program code 1: Sort algorithms

```
#include <iostream>
#include <ctime>
#include <cstdlib>

using namespace std;

typedef struct node_t {
    int key;
    struct node_t *next;
    node_t(int v) {key = v; next = NULL;}
} node_t;

void add(node_t *prev, int key) {
    node_t *tmp = new node_t(key);
    tmp->next = prev->next;
    prev->next = tmp;
}

void swap(int &a, int &b) {
```

```
    int t = a; a = b; b = t;
}

void bubblesort(int *d, int n) {
    for (int i = 0; i < n; ++i)
        for (int j = n - 1; j > i; --j)
            if (d[j] < d[j-1])
                swap(d[j], d[j-1]);
}

void insertionsort_array(int *d, int n) {
    for (int i = 1; i < n; ++i) {
        int key = d[i], j;
        for (j = 0; j < i && d[j] <= key; ++j);
        for (int k = i; k > j; --k) d[k] = d[k-1];
        d[j] = key;
    }
}

void insertionsort_list(int *d, int n) {
    node_t *head = new node_t(0);
    for (int i = 0; i < n; ++i) {
        node_t *tmp = head;
        while (tmp->next && tmp->next->key < d[i]) tmp = tmp->next;
        add(tmp, d[i]);
    }
    node_t *tmp = head->next, *ptr;
    while (tmp) {
        cout << tmp->key;
        ptr = tmp;
        tmp = tmp->next;
        delete ptr;
    }
    delete head;
}

void selectionsort(int *d, int n) {
    for (int i = 1; i < n - 1; ++i) {
```

```
        int flag = i;
        for (int j = i + 1; j < n; ++j)
            if (d[j] < d[flag])
                flag = j;
        swap(d[flag], d[i]);
    }
}

void merge(int *d, int l, int m, int r) {
    int i = l, j = m + 1, k = 0;
    int *tmp = new int[r-l+1];
    while (i <= m && j <= r) {
        if (d[i] <= d[j]) tmp[k++] = d[i++];
        else tmp[k++] = d[j++];
    }
    while (i <= m) tmp[k++] = d[i++];
    while (j <= r) tmp[k++] = d[j++];
    for (i = l; i <= r; ++i) d[i] = tmp[i-l];
    delete[] tmp;
}

void mergesort(int *d, int l, int r) { //close interval
    if (l >= r) return;
    int m = ((l + r) >> 1);
    mergesort(d, l, m);
    mergesort(d, m + 1, r);
    merge(d, l, m, r);
}

void quicksort_extra(int *d, int l, int r) {
    if (l >= r) return;
    int p = rand()%(r-l+1)+l;
    swap(d[l], d[p]);
    int key = d[l];
    int i = 0, j = r - l;
    int *b = new int[r-l+1];
    for (int k = l + 1; k <= r ; ++k) {
        if (d[k] < key) b[i++] = d[k];
```

```cpp
        else b[j--] = d[k];
    }
    b[i] = key;
    for (int k = 0; k <= r - l; ++k) d[k + l] = b[k];
    delete[] b;
    quicksort_extra(d, l, l + i - 1);
    quicksort_extra(d, l + i + 1, r);
}

void quicksort_inplace(int *d, int l, int r) {
    if (l >= r) return;
    int p = rand()%(r-l+1)+l;
    swap(d[l], d[p]);
    int key = d[l];
    int i = l, j = r;
    while (i < j) {
        while(d[j] >= key && i < j) --j; // make sure finally i == j and d[j]=d[i] < key so that you could put it on the left
        while(d[i] <= key && i < j) ++i;
        if (i < j) swap(d[i], d[j]);
    }
    d[l] = d[i];
    d[i] = key;
    quicksort_inplace(d, l, i - 1);
    quicksort_inplace(d, i + 1, r);
}

void rd(int *d, int n){
    for (int i = 0; i < n; ++i)
        cin >> d[i];
}

void prt(int *d, int n) {
    for (int i = 0; i < n; ++i)
        cout << d[i] << "\n";
}

int main() {
```

```
    ios::sync_with_stdio(false);
    srand(time(NULL));
    int cmd, n;
    cin >> cmd >> n;
    int *d = new int[n];
    rd(d, n);
    int start = clock();
    switch(cmd) {
        case 0: bubblesort(d, n); break;
        case 1: insertionsort_array(d, n); break;
        case 2: selectionsort(d, n); break;
        case 3: mergesort(d, 0, n-1); break;
        case 4: quicksort_extra(d, 0, n-1); break;
        case 5: quicksort_inplace(d, 0, n-1); break;
        default: return 0;
    }
    //prt(d, n);
    cout << (clock() - start)*1.0/CLOCKS_PER_SEC << "\n";
    delete[] d;
    return 0;
}
```

Program code 2: Test case generator

```
#!/usr/bin/python

import sys
reload(sys)
sys.setdefaultencoding('utf-8')

import random
import os
import commands
import time

MAX = 15
INT_MAX = 2**31;
size = [5, 5000, 8000, 12000, 20000, 30000, 40000, 50000, int(0.25e7),
↪ int(0.5e7), int(1e7), int(1.5e7), int(2.0e7), int(2.5e7), int(3e7)]
```

```python
if __name__ == "__main__":
    for cases in range(MAX):
        with open('input{}'.format(cases), 'w') as w:
            n = size[cases]
            w.write(str(n) + '\n')
            for i in range(n):
                w.write(str(random.randint(-INT_MAX, INT_MAX-1)) + '\n')
        print("Testcase{}".format(cases))
```

Program code 3: Cases runner

```
#!/bin/zsh

for i in `seq 0 2`
do
    echo $i
    for j in `seq 0 7`
    do
        echo $i | ./a1_test < input$j >> $i.out
        echo $j $?
    done
done

for i in `seq 3 5`
do
    echo $i
    for j in `seq 8 14`
    do
        echo $i | ./a1_test < input$j >> $i.out
        echo $j $?
    done
done
#rm -rf Testcase*
```

Program code 4: Plotting program

```python
import pandas as pd
import matplotlib.pyplot as plt
from matplotlib.patches import *
import seaborn as sns
n = [5, 5000, 8000, 12000, 20000, 30000, 40000, 50000, int(0.25e7),
    int(0.5e7), int(1e7), int(1.5e7), int(2.0e7), int(2.5e7), int(3e7)]
label = ['Bubblesort', 'Insertionsort', 'Selectionsort', 'Mergesort',
    'Quicksort_extra', 'Quicksort_inplace']
plt.figure(figsize=(10,6))
for i in range(3):
    with open('{}.out'.format(i), 'r') as f:
        data = f.read();
        time = data.split('\n')
        plt.plot(n[:8], time[:-1])
plt.legend(label[:3], loc = 'upper left')
plt.xlabel('Input size')
plt.ylabel('Execution time (s)')
plt.title('$O(n^2)$ sort algorithms')
plt.savefig('012.jpg')
plt.show()

plt.figure(figsize=(10,6))
for i in range(3,6):
    with open('{}.out'.format(i), 'r') as f:
        data = f.read();
        time = data.split('\n')
        nn = n[8:]
        ttime = time[:-1]
        nn.insert(0,0)
        ttime.insert(0,'0.0')
        plt.plot(nn, ttime)
plt.legend(label[3:6], loc='upper left')
plt.xlabel('Input size')
plt.ylabel('Execution time (s)')
plt.title('$O(n\log n)$ sort algorithms')
plt.savefig('345.jpg')
plt.show()
plt.figure(figsize=(10,6))
```

```
for i in range(3,6):
    with open('reverse{}.out'.format(i), 'r') as f:
        data = f.read();
        time = data.split('\n')
        plt.plot(n, time[:-1])
plt.legend(label[3:6], loc = 'upper left')
plt.xlabel('Input size')
plt.ylabel('Execution time (s)')
plt.title('$O(n\log n)$ Sort algorithms for a descending array')
plt.savefig('reverse.jpg')
plt.show()

plt.figure(figsize=(10,6))
for i in range(6):
    with open('same{}.out'.format(i), 'r') as f:
        data = f.read();
        time = data.split('\n')
        plt.plot(n[:8], time[:-1])
plt.legend(label, loc = 'upper left')
plt.xlabel('Input size')
plt.ylabel('Execution time (s)')
plt.title('Sort algorithms for an array of 1')
plt.savefig('same.jpg')
plt.show()
```