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 average runtime with input in different sizes. Input of each size is repeated for five times. 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

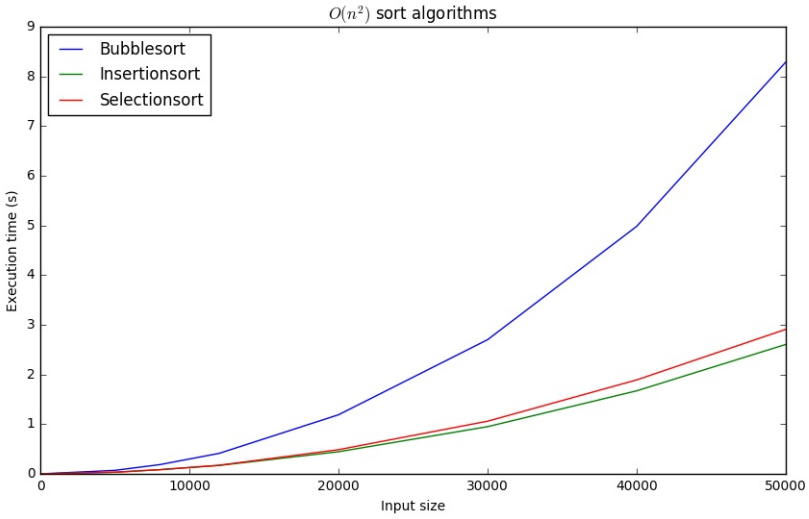

Figure 1: Performance of $O(n^2)$ sort algorithms

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

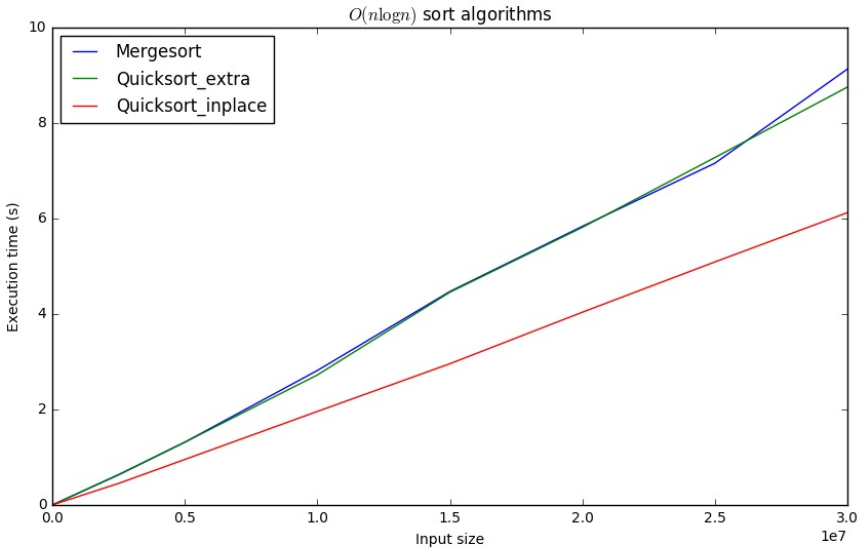

Figure 2: Performance of $O(n \log n)$ sort algorithms

In Fig. 2, 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. It shows that allocating memory is considerably time-consuming. 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 still separate them into two categories. We observed that all of them become more efficient. One possible explanation is that the descending data guarantees that each element differs. Elements with the same value probably lead to slowing down.

In addition, the gap between Bubble sort and the other two $O(n^2)$ algorithms is shrinking, saying that their times of swap are approaching. In general, the complexities essentially follows their average performance. Quick sort does not degenerate back to $O(n^2)$.

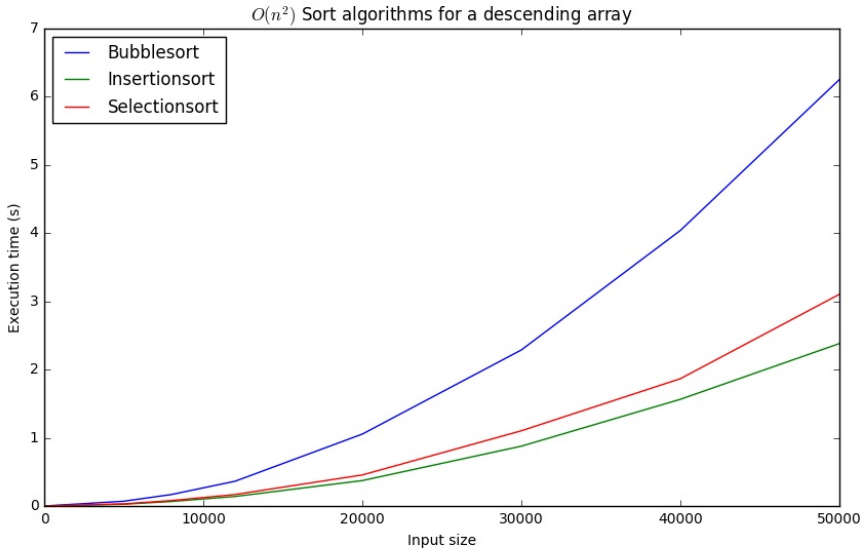

Figure 3: Performance of $O(n^2)$ sort algorithms

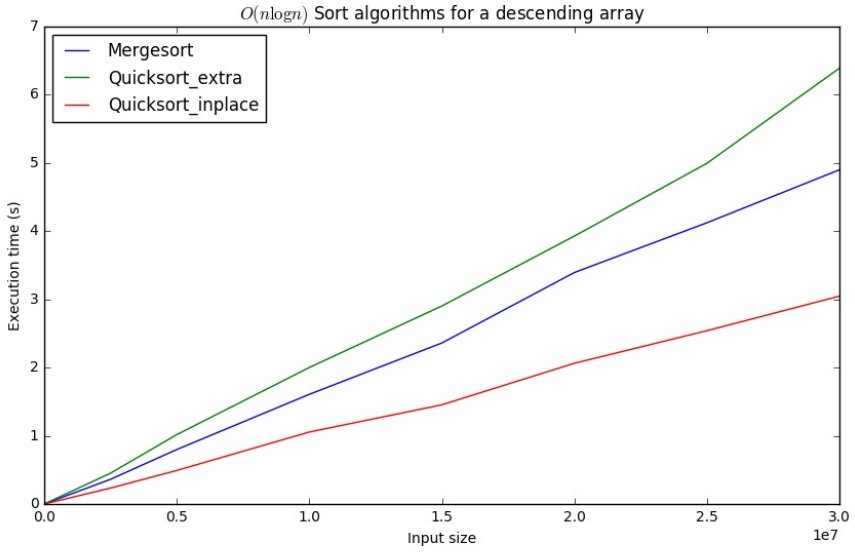

Figure 4: Performance of $O(n \log n)$ sort algorithms

4.2 An Array of 1's

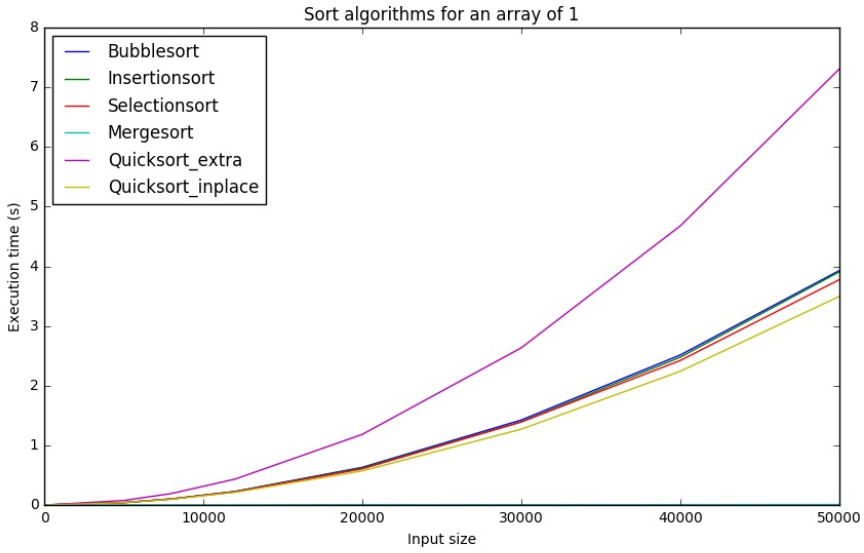

Figure 5: Performance of $O(n \log n)$ sort algorithms in 1's array

For this manipulated input, the performance of Quick sort precipitously degenerates, especially the version that requires extra space. Not only does it degenerate to $O(n^2)$ but also obtains a huge constant, even larger than Bubble sort. This phenomenon, again, indicates the flaw of frequently allocating memories.

On the contrary, Merge sort is quite steady. Whether the input is manipulated or not does not provoke a notable change on its performance, even though in average case it's still slower than in-place Quick sort.

5 Conclusion

In this report, we have demonstrated the characteristics of six different sort algorithms as well as their performance when the input is descending or all the same. Quick sort with extra memory is especially unsteady whose performance greatly depends on the input pattern. While the Merge sort, on the contrary, is brilliant on steadiness.

Also, the Insertion sort implemented with linked-list is an interesting case. It avoids $O(n)$ time to move the sorted part but involves extra memory allocation. Since $O(n^2)$ sort algorithms are not practicable in reality, here we do not apply the time complexity analysis, although the code is finished in Appendix. If the reader is interested, the codes of which is listed in Appendix and easy to test.

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;
```

```
36      }
37  }
38
39  void insertionsort_list(int *d, int n) {
40      node_t *head = new node_t(0);
41      for (int i = 0; i < n; ++i) {
42          node_t *tmp = head;
43          while (tmp->next && tmp->next->key < d[i]) tmp = tmp->next;
44          add(tmp, d[i]);
45      }
46      node_t *tmp = head->next, *ptr;
47      while (tmp) {
48          cout << tmp->key;
49          ptr = tmp;
50          tmp = tmp->next;
51          delete ptr;
52      }
53      delete head;
54  }
55
56  void selectionsort(int *d, int n) {
57      for (int i = 0; i < n - 1; ++i) {
58          int flag = i;
59          for (int j = i + 1; j < n; ++j)
60              if (d[j] < d[flag])
61                  flag = j;
62          swap(d[flag], d[i]);
63      }
64  }
65
66  void merge(int *d, int l, int m, int r) {
67      int i = l, j = m + 1, k = 0;
68      int *tmp = new int[r-l+1];
69      while (i <= m && j <= r) {
70          if (d[i] <= d[j]) tmp[k++] = d[i++];
71          else tmp[k++] = d[j++];
72      }
73      while (i <= m) tmp[k++] = d[i++];
```

```
74      while (j <= r) tmp[k++] = d[j++];
75      for (i = l; i <= r; ++i) d[i] = tmp[i-l];
76      delete[] tmp;
77  }
78
79  void mergesort(int *d, int l, int r) { //close interval
80      if (l >= r) return;
81      int m = ((l + r) >> 1);
82      mergesort(d, l, m);
83      mergesort(d, m + 1, r);
84      merge(d, l, m, r);
85  }
86
87  void quicksort_extra(int *d, int l, int r) {
88      if (l >= r) return;
89      int p = rand()%(r-l+1)+l;
90      swap(d[l], d[p]);
91      int key = d[l];
92      int i = 0, j = r - l;
93      int *b = new int[r-l+1];
94      for (int k = l + 1; k <= r ; ++k) {
95          if (d[k] < key) b[i++] = d[k];
96          else b[j--] = d[k];
97      }
98      b[i] = key;
99      for (int k = 0; k <= r - l; ++k) d[k + l] = b[k];
100     delete[] b;
101     quicksort_extra(d, l, l + i - 1);
102     quicksort_extra(d, l + i + 1, r);
103 }
104
105 void quicksort_inplace(int *d, int l, int r) {
106     if (l >= r) return;
107     int p = rand()%(r-l+1)+l;
108     swap(d[l], d[p]);
109     int key = d[l];
110     int i = l, j = r;
111     while (i < j) {
```

```
112             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
113             while(d[i] <= key && i < j) ++i;
114             if (i < j) swap(d[i], d[j]);
115         }
116         d[l] = d[i];
117         d[i] = key;
118         quicksort_inplace(d, l, i - 1);
119         quicksort_inplace(d, i + 1, r);
120     }
121
122     void rd(int *d, int n){
123         for (int i = 0; i < n; ++i)
124             cin >> d[i];
125     }
126
127     void prt(int *d, int n) {
128         for (int i = 0; i < n; ++i)
129             cout << d[i] << "\n";
130     }
131
132     int main() {
133         ios::sync_with_stdio(false);
134         srand(time(NULL));
135         int cmd, n;
136         cin >> cmd >> n;
137         int *d = new int[n];
138         rd(d, n);
139         int start = clock();
140         switch(cmd) {
141             case 0: bubblesort(d, n); break;
142             case 1: insertionsort_array(d, n); break;
143             case 2: selectionsort(d, n); break;
144             case 3: mergesort(d, 0, n-1); break;
145             case 4: quicksort_extra(d, 0, n-1); break;
146             case 5: quicksort_inplace(d, 0, n-1); break;
147             default: return 0;
148         }
```

```
149     //prt(d, n);
150     cout << (clock() - start)*1.0/CLOCKS_PER_SEC << "\n";
151     delete[] d;
152     return 0;
153 }
```

Program code 2: Test case generator

```
1  #!/usr/bin/python
2
3  import sys
4  reload(sys)
5  sys.setdefaultencoding('utf-8')
6
7  import random
8  import os
9  import commands
10 import time
11
12 MAX = 15
13 INT_MAX = 2**31;
14 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)]
15 if __name__ == "__main__":
16     for cases in range(MAX):
17         for k in range(5):
18             with open('input{}{}'.format(cases,k), 'w') as w:
19                 n = size[cases]
20                 w.write(str(n) + '\n')
21                 for i in range(n):
22                     w.write(str(random.randint(-INT_MAX, INT_MAX-1)) + '\n')
23         print("Testcase{}".format(cases))
```

Program code 3: Cases runner

```
1  #!/bin/zsh
2
```

```
for ((i=0; i<3; i++)); do
    echo $i
    for ((j=0; j<8; j++)); do
        for ((k=0; k<5; k++)); do
            echo $i | ./a1_test < input$j$k >> $i$k.out
            echo $j $?
        done
    done
done

for ((i=3; i<6; i++)); do
    echo $i
    for ((j=8; j<15; j++)); do
        for ((k=0; k<5; k++)); do
            echo $i | ./a1_test < input$j$k >> $i$k.out
            echo $j $?
        done
    done
done
```

Program code 4: Plotting program

```
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']
def myplot(Range, N, Out, Label, Title, name):
    plt.figure(figsize=(10,6))
    for i in Range:
        time = [0]*len(N)
        for k in range(5):
            with open(Out.format(i,k), 'r') as f:
                data = f.read()
                tmp = data.split('\n')
```

```
                time = time + map(eval, tmp[:-1]) //eliminate the ending ' '
        time = [t/5.0 for i in time];
        plt.plot(N, time)
    plt.legend(Label, loc = 'upper left')
    plt.xlabel('Input size')
    plt.ylabel('Execution time (s)')
    plt.title(Title)
    plt.savefig(name)
    plt.show()

def __name__ = "__main__":
    myplot(range(3), n[:8], '{}{}.out', label[:3], '$O(n^2)$ sort algorithms',
    ↪ '012.jpg')
    myplot(range(3,6), n[8:], '{}{}.out', label[3:6], '$O(n\log n)$ sort
    ↪ algorithms', '345.jpg')
    myplot(range(3), n[:8], 'reverse{}{}.out', label[:3], '$O(n^2)$ Sort
    ↪ algorithms for a descending array', 'reverse012.jpg')
    myplot(range(3,6), n[8:], 'reverse{}{}.out', label[3:6], '$O(n\log n)$
    ↪ Sort algorithms for a descending array', 'reverse345.jpg')
    myplot(range(6), n[:8], 'same{}{}.out', label, 'Sort algorithms for an
    ↪ array of 1', 'same.jpg')

```