Authors:

Brian Mitzel	893038547
Sorasit Wanichpan	897260477
Abeer Salam	899423594

Course:

CPSC 335

Assignment:

Project 3

Date:

April 18, 2014

I. Implementation 1: an O(n^2)-time sorting algorithm in Python

The *string sorting* problem is:
input: a list L of n words
output: the list L containing the same words in non-decreasing order
size: n

We will use the *in-place selection sort* algorithm given in Section 3.8 as a template to sort the strings from the provided text file.

Python in place selection sort result of n = 200

```
----------
Python in-place selection sort:
requested n = 200
loaded 200 lines from 'beowulf.txt'
first 10 unsorted words: ['The', 'Project', 'Gutenberg', 'EBook', 'of', 'Beowulf', 'This', 'eBook', 'is', 'for']
running in-place selection sort...
first 10 sorted words: ['AN', 'ANGLOSAXON', 'Act', 'An', 'AngloSaxon', 'Author', 'BEOWULF', 'BEOWULF', 'BOSTON', 'BY']
elapsed time: 0.0017098770003940444 seconds
----------
```

V3: Python in-place selection sort

```
1  #Brian Mitzel         893038547
2  #Sorasit Wanichpan 897260477
3  #Abeer Salam          899423594
4  #CPSC 335
5  #Project 3
6  #Version 3
7  #
8  #This script reads the first n lines of a text file
9  #and uses an in-place selection sort algorithm to sort the
10 #lines alphabetically. It also records the time that the
11 #in-place selection sort algorithm takes to execute.
12 #
13 #Note: Python = less code/longer time vs C++'s more code/less time.
14 #
15 #run: python3 ip_selection_sort.py <input file> <n value>
16
17 import sys
18 import time
19
20 #In place selection sort that was given in class
21 def in_place_selection_sort(L):
22        for k in range(len(L)-1):
23        least = k
24        for i in range(k+1, len(L)):
25               if L[i] < L[least]:
26               least = i
27
28        #swap elements
29        L[k], L[least] = L[least], L[k]
30        return L
31
32 def main():
33        if len(sys.argv) !=3:
34        print('error: you must supply exactly two arguments\n\n' +
35               'usage: python3 ip_selection_sort.py <input file> <n value>' )
36        sys.exit(1)
37
```

```
38          #Capture the command line arguments
39          input_file = sys.argv[1]
40          n_value = int(sys.argv[2])
41
42          #Introduction
43          print('----------')
44          print('Python in-place selection sort:')
45          print('requested n = ' + str(n_value))
46          to_sort = [line.rstrip('\n') for line in open(input_file)][0:n_value]
47
48          print('loaded ' + str(n_value) + ' lines from \'' + input_file + '\'')
49          print('first 10 unsorted words: ' + str(to_sort[0:10])) #Prints out the first
50  10
51
52          #Run the algorithm/start the timer
53          print('running in-place selection sort...')
54          start = time.perf_counter()
55          result = in_place_selection_sort(to_sort)
56          end = time.perf_counter()
57
58          #Results
59          print('first 10 sorted words: ' + str(result[0:10]))
60          print('elapsed time: ' + str(end - start) + ' seconds')
61          print('----------')
62
63  if __name__ == "__main__":
64          main()
```

II. Implementation 2: an O(n^2)-time sorting algorithm in C++

We will use the *in-place selection sort* algorithm given in Section 3.8 as a template to sort the strings from the provided text file. We will use the same in-place selection sort, but now implemented in C++. If our hypothesis was correct, our lower level C++ implementation will run faster than our higher level Python implementation.

C++ in place selection sort results of n = 200 run

```
----------
C++ in-place selection sort:
requested n = 200
first 10 unsorted words: [The] [Project] [Gutenberg] [EBook] [of] [Beowulf] [This] [eBook] [is] [for]
loaded 200 lines from 'beowulf.txt'
running in-place selection sort...
first 10 sorted words: [AN] [ANGLOSAXON] [Act] [An] [AngloSaxon] [Author] [BEOWULF] [BEOWULF] [BOSTON] [BY]
elapsed time: 0.001175 seconds

----------
```

Testing Highlights:
There is an interesting note regarding the C++ implementation. Our hypothesis states that the lower level languages such as C++ will run faster than higher-level languages such as Python by some multiplicative constant. In our initial testing with the C++ implementation, our timing results showed that the C++ sort was taking longer than the Python sort, which was the direct opposite result we expected!

We narrowed down the issue to Visual Studio's configuration mode for compiling the code. The debugger mode doesn't optimize the binary it produces and leaves open handles for debuggers to attach to, whereas the release mode enables optimization and generates less extra debugging data. In short, the debugger mode is unoptimized and produces extra data that adds an invisible overhead layer to the sort that skews our timing results. Compiling and executing the code in Linux or running the code in release mode eliminates this issue.

V3: C++ in-place selection sort

```
1  //Brian Mitzel        893038547
2  //Sorasit Wanichpan 897260477
3  //Abeer Salam         899423594
4  //CPSC 335
5  //Version 3
6  //Project 3, in-place selection sort
7  //
8  //This C++ program reads the first n lines of a text file and uses an in-place
9  //selection sort algorithm to sort the lines alphabetically. It also records the
10 //time that the in-place selection sort algorithm takes to execute.
11 //
12 //Note: Make sure the machine running the code is not running on battery (or //power
13 saving mode) and that the program is running on release mode (in Visual //Studios). If
14 run in debug mode, it will generate debugging files and impact //overall performance
15 (affects overall runtime). In addition, make sure you //compile the project with the
16 C++ 11 standards; otherwise, chrono will not work. //For Visual Studio, this requires
17 version 2012 or later.
18 //
19 //To compile in Linux: g++ std=c++11 in_place_selection.cpp -o ips.exe
20 //To run: ./ips.exe <filename> <# number of words to be sorted>
21
22 #include <iostream>
23 #include <cstdlib>
```

```cpp
#include <chrono>
#include <fstream>
#include <string>

using namespace std;

void in_place_selection_sortA(string[], int);
//Purpose: Performs in-place selection sort on an array of strings
//Precondition: The size of the array is passed as a nonnegative integer along with
the array itself
//Postcondition: The array of strings is sorted in alphabetical order

int main(int argc, char* argv[])
{
    if (argc != 3)
    {
        cout << "error: you must supply exactly two arguments\n\n"
                << "usage: python3 ip_selection_sort.py <input file> <n value>" <<
endl;
        exit(1);
    }

    //Capture the command line arguments
    ifstream fileToBeRead;
    fileToBeRead.open(argv[1]);
    int n_Val = atoi(argv[2]);

    //Introduction
    cout << "----------" << endl;
    cout << "C++ in-place selection sort:" << endl;
    cout << "requested n = " << n_Val << endl;

    //Read in file and print first 10 unsorted words
    cout << "first 10 unsorted words: ";
    string* words = new string[n_Val];
    for (int i = 0; i < n_Val; i++)
    {
        fileToBeRead >> words[i]; //Reads until end of line
        if (i < 10) //Prints out the first 10 words (Unsorted)
        {
                cout << "[" << words[i] << "] ";
        }
    }
    cout << "\nloaded " << n_Val << " lines from '" << argv[1] << "'" << endl;

    //Runs the timer and algorithm
    cout << "running in-place selection sort..." << endl;
    auto start = chrono::high_resolution_clock::now();
    in_place_selection_sortA(words, n_Val);
    auto end = chrono::high_resolution_clock::now();
    int microseconds = chrono::duration_cast<chrono::microseconds>(end -
start).count();
    double seconds = microseconds / 1E6;

    //Outputs the first 10 sorted words and the elapsed time
    cout << "first 10 sorted words: ";
    for (int i = 0; i < 10; i++)
    {
        cout << "[" << words[i] + "] ";
    }
    cout << "\nelapsed time: " << seconds << " seconds" << endl;
```

```
86      cout << "\n----------" << endl;
87
88      delete[] words;
89
90      return 0;
91  }
92
93  //In-place selection sort, implemented in C++
94  void in_place_selection_sortA(string token[], int n)
95  {
96      int least;
97      for (int i = 0; i < (n - 1); i++)
98      {
99          least = i;
100         for (int j = (i + 1); j < n; j++)
101         {
102                 if (token[j] < token[least])
103                 {
104                         least = j;
105                 }
106         }
107         swap(token[i], token[least]);
108     }
109 }
```

III. Implementation 3: an O(n log n)-time sorting algorithm in Python

We will use the *merge sort* algorithm given in Section 5.5 as a template to sort the strings from the provided text file. According to hypothesis 3, this implementation should outperform our other two implementations.

Python merge sort results of n = 200

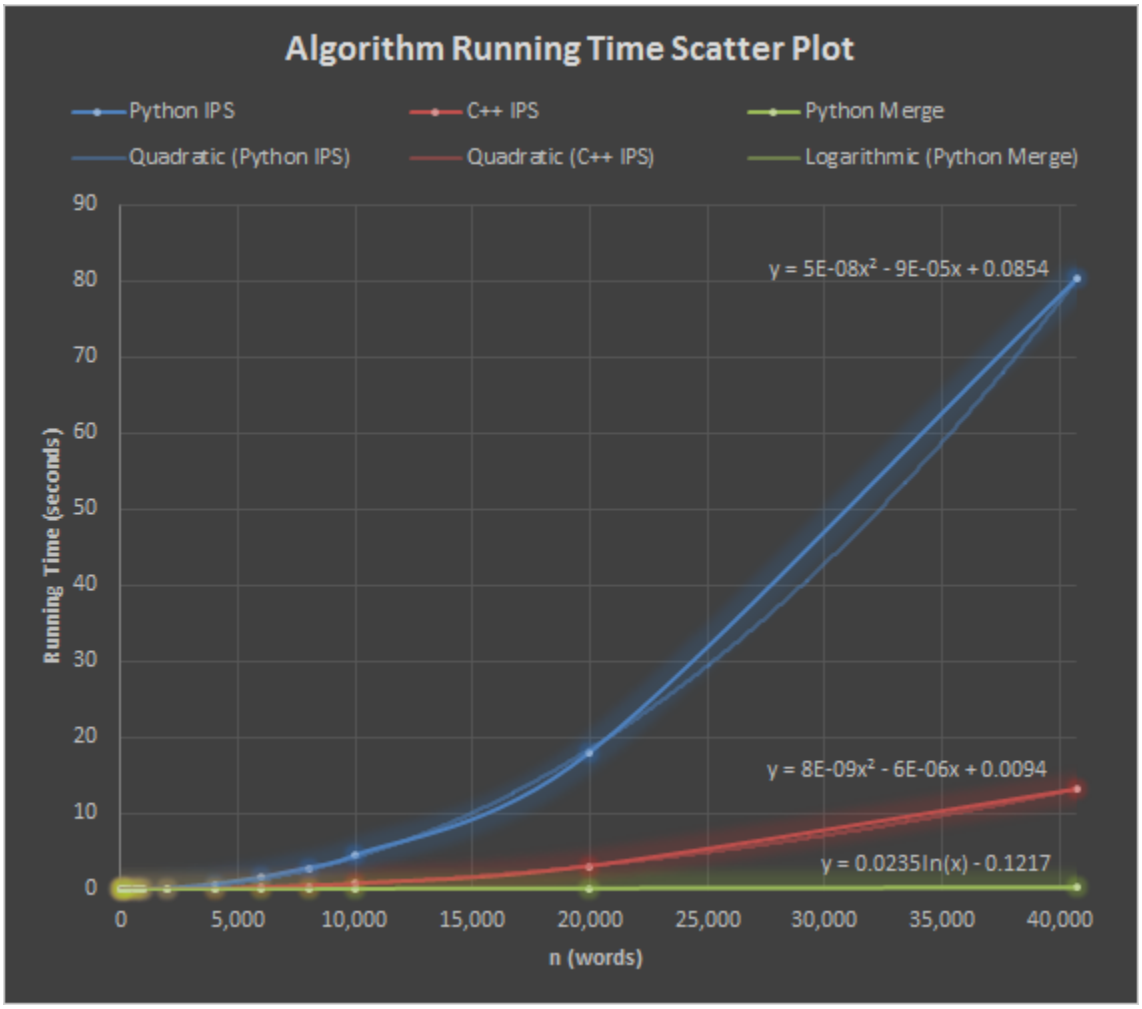
```
----------
Python merge sort:
requested n = 200
loaded 200 lines from 'beowulf.txt'
first 10 unsorted words: ['The', 'Project', 'Gutenberg', 'EBook', 'of', 'Beowulf', 'This', 'eBook', 'is', 'for']
running merge sort...
first 10 sorted words: ['AN', 'ANGLOSAXON', 'Act', 'An', 'AngloSaxon', 'Author', 'BEOWULF', 'BEOWULF', 'BOSTON', 'BY']
elapsed time: 0.0008873309998307377 seconds
----------
```

V4: Python merge sort

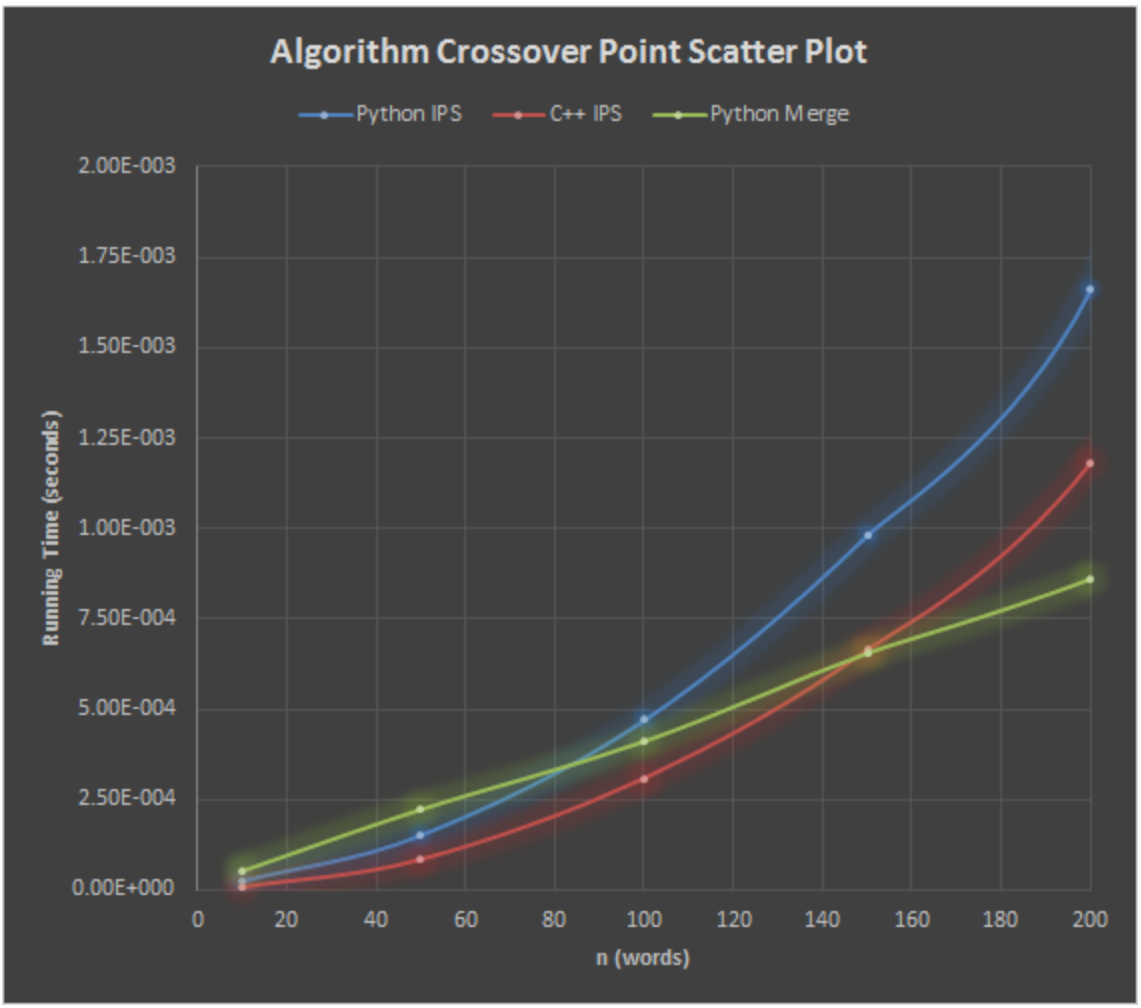
```
#Brian Mitzel      893038547
#Sorasit Wanichpan 897260477
#Abeer Salam       899423594
#CPSC 335
#Project 3
#Version 4

#This script reads the first n lines of a text file
#and uses a merge sort algorithm to sort the lines
#alphabetically. It also records the time that the
#merge sort algorithm takes to execute.

#run: python3 merge_sort.py <input file> <n value>

import sys
import time

# Merges two sorted lists A and B into one sorted list S
# as shown in the lecture notes
def merge(A, B):
    S = []
    i = 0
    j = 0
    while i < len(A) and j < len(B):
        if A[i] <= B[j]:
            x = A[i]
            i += 1
        else:
            x = B[j]
            j += 1
        S.append(x)
    return S + A[i:] + B[j:]

# Sorts a list L using the Merge Sort decrease-by-half algorithm
# as shown in the lecture notes
def merge_sort(L):
    if len(L) <= 1:
        return L
    else:
        middle = int(len(L) / 2)
        left = L[:middle]
        right = L[middle:]

```

```
44          left = merge_sort(left)
45          right = merge_sort(right)
46          return merge(left, right)
47
48  def main():
49      if len(sys.argv) !=3:
50          print('error: you must supply exactly two arguments\n\n' +
51                'usage: python3 merge_sort.py <input file> <n value>' )
52          sys.exit(1)
53
54      #Capture the command line arguments
55      input_file = sys.argv[1]
56      n_value = int(sys.argv[2])
57
58      #Introduction
60      print('----------')
61      print('Python merge sort:')
62      print('requested n = ' + str(n_value))
63      to_sort = [line.rstrip('\n') for line in open(input_file)][0:n_value]
64      print('loaded ' + str(n_value) + ' lines from \'' + input_file + '\'')
65      print('first 10 unsorted words: ' + str(to_sort[0:10])) #Prints out the first 10
66
67      #Run the algorithm/start the timer
68      print('running merge sort...')
69      start = time.perf_counter()
70      result = merge_sort(to_sort)
71      end = time.perf_counter()
72
73      #Results
74      print('first 10 sorted words: ' + str(result[0:10]))
75      print('elapsed time: ' + str(end - start) + ' seconds')
76      print('----------')
77
78  if __name__ == "__main__":
79      main()
80
```

IV. Scatter Plots



This scatter plot shows the plot points of the empirical data we collected during the executions of our three implementations. The implementations for the Python and C++ in-place selection sorts both fit on quadratic curves, while the implementation for the Python merge sort fits a logarithmic curve.



This scatter plot shows the crossover point n_c where the Python merge sort implementation becomes faster than both of the in-place selection sort implementations. The approximate value of n_c is 148.

V. Questions

a) Which $O(n^2)$ algorithm did you choose to implement, and why?

We decided to implement an in-place selection sort for our $O(n^2)$ algorithm because of its relative simplicity and because of our familiarity with the algorithm after having covered it in class. It also has a slight performance advantage over the out-of-place selection sort. Finally, we were able to avoid having to create a separate list to store and return the sorted results.

b) Which $O(n \log n)$ algorithm did you choose to implement, and why?

We chose to use merge sort as our $O(n \log n)$ algorithm because we wanted to compare a decrease-by-half implementation with the decrease-by-one implementations of in-place selection sort. As an additional note, merge sort also provides a bit more consistency in its performance when compared with the alternative, quick sort. Both the average case and worst case time complexity for merge sort are in $O(n \log n)$. By comparison, while the average case time complexity for quick sort is also $O(n \log n)$, its worst case is only $O(n^2)$.

c) Which of the three algorithms did you find most difficult to implement, and why?

Of the three algorithms, we found the C++ in-place selection sort algorithm to be the trickiest to implement. One reason that the two Python implementations were simpler was because we had already developed the Python pseudocode for those algorithms in our lectures. On the other hand, the C++ implementation first had to be adapted from the Python code. Then, we also encountered an issue getting accurate timing results when building our program in Microsoft Visual Studio. See the testing highlights in Section II above for more details.

d) Are your empirical results consistent or inconsistent with hypothesis 1? In other words, do the run times of both implementation 1 and 2 fit quadratic curves?

Our empirical results are consistent with hypothesis 1 because our scatter plot shows both implementations 1 and 2 fitting quadratic curves. Therefore, the efficiency class that was mathematically derived for in-place selection sort, $O(n^2)$, accurately predicted the run time of the algorithm's implementation for both the Python and C++ implementations.

e) Are your empirical results consistent or inconsistent with hypothesis 2? In other words, are the run times of your implementation 1 greater than those of implementation 2 by a constant factor? If so, approximately what is that factor, as a percentage? Does this result surprise you?

Our empirical results are inconsistent with hypothesis 2, which states that C++ is faster than Python by some multiplicative constant. We can see this by analyzing some of our run times in the table below:

n (words)	**Python IPS sort run time (seconds)**	**C++ IPS sort run time (seconds)**	**Python/C++ Performance Factor (%)**
200	0.00166	0.00118	141%
800	0.02878	0.01195	241%
4,000	0.71589	0.12665	565%
10,000	4.50347	0.06556	582%
40,707	80.39747	13.282	605%

This table shows that the C++ implementation of the in-place selection sort becomes increasingly faster as the value of n becomes increasingly larger. It is definitely surprising to us to see just how much faster the C++ implementation of the same algorithm in Python can be.

f) Are your empirical results consistent or inconsistent with hypothesis 3? In other words, is there a crossover point n_c for which implementation 3 is faster than implementations 1 and 2? If so, what is the approximate value of n_c? How much faster is implementation 3 over implementation 2, as a percentage, for the full $n = 40{,}707$? Does this result surprise you?

Our empirical results are consistent with hypothesis 3. Our second scatter plot in Section IV shows that there is a crossover point n_c for which implementation 3 is faster than implementations 1 and 2. By looking at our scatter plot, we can see that the approximate value of n_c is 148. After this point, implementation 3 is always faster than implementations 1 and 2. For the full value of $n = 40{,}707$,

$$\frac{13.282}{0.3013102} = 4{,}408\%$$

This result is not as surprising, since it was expected that an algorithm with a time complexity of $O(n \log n)$ would be significantly faster than one of $O(n^2)$ for large values of n.

g) Based on these results, which approach do you think is a better way of implementing algorithms efficiently: implementing a slow algorithm in a fast low level language (implementation 2), or implementing a fast algorithm in a slow high level language (implementation 3)? Why? What are the implications on software development in general?

The members of our group were unable to come to a consensus on this question, so we have included arguments for both sides.

1) The argument for implementing a slow algorithm in a fast low level language:

To develop algorithms most efficiently, we should implement them in a lower-level language. With a lower-level language, programmers have greater control over the code. This is extremely crucial for programs that require an extreme amount of precision and timing performance, e.g. Curiosity MSL skycrane landing. The performance benefits from implementing an $O(n \log n)$ algorithm in Python can be beaten by the same implementation in C++. This saves valuable processing time for other mission critical tasks (terrain sensing, altitude (XYZ) control, etc.) on a very limited hardware platform.

2) The argument for implementing a fast algorithm in a slow high level language:

On the other hand, high level language implementations allow programmers to pump out working code in less time and using fewer lines of code in comparison to lower level languages. This means that implementations in high-level languages save time at the cost of performance. The cost benefit ratio from helping to reduce bugs during software development is usually a sufficient enough reason for developers to use HLLs for implementation. In general everyday programs, the performance lost by using a high level language can be curtailed by utilizing efficient algorithms such as merge sort. In today's world, we have an ever-increasing demand for software, and having the ability to write shorter and correct code can definitely outweigh the cost of reduced performance.