Lab07: Mergesort Comparison
CSE2050 Fall 2017
Instructor: Jeffrey Meunier & Wei Wei
TA: Jenny Blessing, Param Bidja, Yamuna Rajan & Zigeng Wang

1. Introduction

In this exercise, we will complete two Python functions in order to implement the mergesort sorting algorithm. In the second half of the assignment, you will also compare the time this algorithm takes to run with the time taken by other algorithms.

2. Objectives

The purpose of this assignment is to help you:

- Practice mergesort.
- See the efficiencies of various sorting algorithms in practice.

***Note**: Before you start, if you are not familiar with mergesort, bubble sort, selection sort, or insertion sort, you are recommended to review your lecture notes and the additional materials from Dr. Sheehy on common sorting algorithms and divide-and-conquer sorting algorithms.*

3. Background

3.1. Mergesort

Mergesort is a classic sorting algorithm which takes a divide-and-conquer approach. It is implemented recursively, where the function takes a list as input, divides the list into two halves, then calls itself recursively for the two halves. Once the smaller portions of the list cannot be divided any further, the algorithm builds the list back up again, in sorted form, by combining the sorted pieces. So, given a list of length *n*, the asymptotic running time of mergesort is *O(nlogn)*.

Wikipedia has an excellent gif that shows mergesort in action on a list of size 8.

4. Assignment

In this assignment, you will need to complete the mergesort(L) function as well as its merge(A, B, L) helper function, where **L** is the current list to be sorted and **A** and **B** are two smaller sub-lists.

4.1. mergesort(L)

First, we will implement the base case. In this case, if the size of the list is one or empty, this list is already sorted and we can return.

To divide, we simply want to create two new lists, **A** and **B**. **A** will contain the first half of the elements in **L**, and **B** will contain the second half of **L**.

To conquer, or solve the original problem using much smaller sub-problems, we recursively call **mergesort()** twice with the new lists **A** and **B** as the inputs. Once we have done this, we will now call the **merge()** function in order to combine the two lists **A** and **B**.

Note that the final line of the function, provided for you, contains three variables: **cost_A**, **cost_B**, and **cost_m**. The purpose of these variables is to determine the actual time, or cost, that the mergesort algorithm takes. **cost_A** and **cost_B** are the result of recursively calling **mergesort(A)** and **mergesort(B)**, respectively, while **cost_m** is the result of calling the **merge() function** (discussed in the following section).

4.2. merge(A, B, L)

Now, we will move to the **merge(A, B, L)** function. This is a helper function for the main **mergesort()** function, and this is where the actual merging of two lists will be done.
We want to iterate over the two lists A and B, each time comparing two elements, one from each list. The third parameter, **L**, is the new, combined list that we are about to build up.

We will need two index variables for each of the lists **A** and **B**, say i and j. With each comparison, we will add the smaller element to **L** in order to end up with a list that is ultimately sorted from smallest to largest. If the elements are equal, it does not matter in which order they are added to the list. If you reach the end of one list but there are still remaining elements in the other list, simply add all remaining elements into **L**. *Note that the input lists A and B are **already sorted**. The **merge()** function merely combines two sorted sub-lists.*

A	2	3	5
B	1	4	5
index	0	1	2

For example, let's look at the sample lists in the table above. We will begin by comparing **A[0]** and **B[0]**. 1 < 2, and so we have **L = [1]**. We will now compare **A[0]** and **B[1]**. 2 < 4, and so we will add 4 to the list to get **L = [1, 2]**. Next, we compare **A[1]** and **B[1]**; 3 < 4, and so we have **L = [1, 2, 3]**. We continue this same process to ultimately result in **L = [1, 2, 3, 4, 5, 5]**. We return the length of **L**, which in this case would be 6.

4.3 Comparison of Sorting Algorithms

Notice that implementations of bubble sort, selection sort, and insertion sort are also provided. The details of these sorting algorithms will not be described in great detail in this document, but please review your notes from class and this link from Dr. Sheehy's notes, which describes the implementations of these three functions.

Below these three additional sorting algorithms, you will see the **run()** function. This function uses Python's **time** module to calculate the precise time that each algorithm takes.

Once you have implemented the mergesort functions correctly, running the program should look something like this:

```
<function mergesort at 0x10559fc80>
  4.911
  2.353
  2.459
  2.260
  2.366
  2.287
  2.396
  2.406
  2.538
<function bubblesort at 0x105599d08>
  2.356
  2.271
  2.711
  3.058
  5.190
  3.069
  3.681
  3.947
  4.539
<function selectionsort at 0x105599d90>
  2.088
  2.162
  2.130
  2.454
  2.811
  3.092
  3.824
  3.618
  4.070
<function insertionsort at 0x105599e18>
  1.636
  3.878
  1.744
  3.694
  4.270
  3.640
  4.234
  3.669
  4.555
```

These numbers will all vary each time you run the program. Note that **run()** contains a line '**print(fun)**', where **fun** is the function name itself. This causes lines such as **<function selectionsort at 0x105599d90>** to be printed. Here, the **0x105599d90** is the memory address of the function. Memory storage is beyond the scope of this course, but it's useful to begin to recognize what memory addresses look like.

If you watch the program run, you will see that mergesort runs quite quickly, and that the three $O(n^2)$ algorithms—bubble sort, selection sort, and insertion sort—are the ones that take significantly longer.

5. Submit your work to Mimir

Submit your code to Mimir after you complete your code. Mimir will automatically your submission based on different unit test cases. These test cases **will only test the merge() and mergesort functions()**. Please comment out or delete any testing code at the bottom of your file you may have relating to the run() function. You can submit your code to Mimir up to **30 times** to refresh your existing score before the submission deadline.

6. Due date

This lab assignment is worth **2 points** in the final grade. It will be due by **11:59pm on Wednesday, November 1st, 2017**. A penalty of **10% per day** will be deducted from your grade, starting at 12:00am.

7. Getting help

Start your project early, because you will probably not be able to get timely help in the last few hours before the assignment is due.

- Go to the office hours of instructors and TAs.
    - Prof. Wei Wei: Mon. 2:30 - 3:15pm, Wed. 2:30 - 3:15pm, Thurs. 2:30 - 3:15pm, Fri. 2:30 - 3:15pm @ITE258
    - Jenny Blessing: Fri. 12pm - 2pm @ITE140
    - Param Bidja: Tues. 2pm - 3pm @ITE140
    - Yamuna Rajan: Tues. 11am - 12pm, Wed. 9:30am - 10:30am @ITE140
    - Zigeng Wang: Mon. 3pm - 4pm @ITE140
- Post your questions on Piazza. Instructors, TAs and many of your classmates may answer your questions.
- Search the answer of your unknown questions on the Internet. Many questions asked by you might have been asked and answered many times online already.