Zachary Humphrey, Dakota Leslie, Cole McCauley

Dr. Chenyi Hu

CSCI 3330 Algorithms

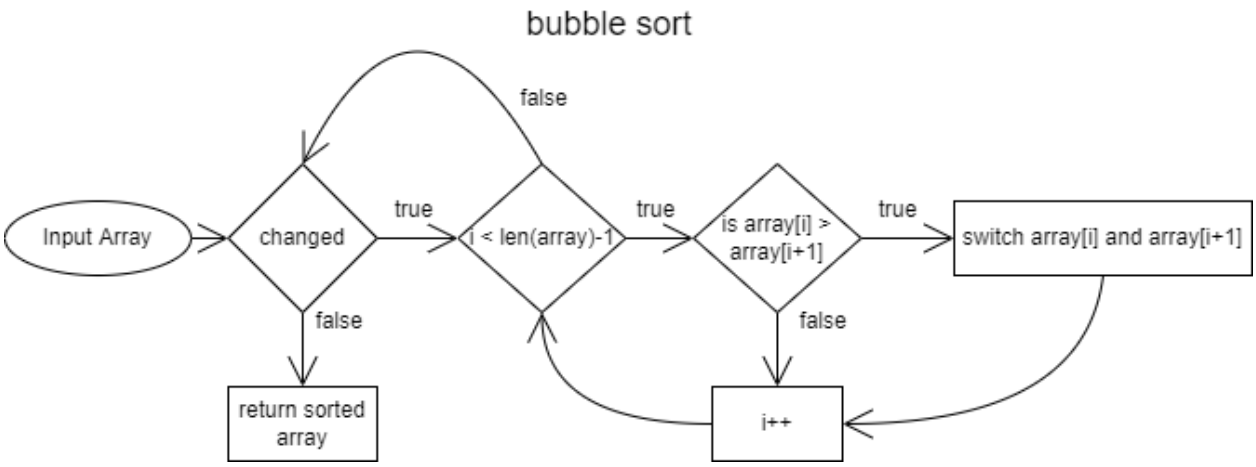
February 18, 2022

Asymmetric Cryptosystems: Project 1

Introduction:

For this project we were required to create a program that runs a bubble sort algorithm, a quick sort algorithm, a merge sort algorithm and one other algorithm of our choice. For our project we choose to do the pancake sort. Dakota chose and coded in the algorithms being used in the project as well as any error testing in the algorithms, Zach implemented the test suite and graphing functions in order for us to analyze our results, and Cole compiled all of our data into the report organizing our process and findings as well as helping with the test suite. In order to analyze and compare the time complexity of the algorithms, we created a time_sorting_fn function which takes in the algorithm to use and the list to sort, it then returns the time that it took for the algorithm to sort the list, allowing us to compare sorting times. To get data to sort we created a function that takes in the size of the array as an argument and fills the array with random values between 1 and 10000000000. Together out software solution, calls our time complexity function on each of our algorithms using three different array sizes and array orientations then graphing the data to give a visual of the time complexity of each algorithm for each array size, as well as making it easy to compare and contrast the times it took each algorithm compared to the other algorithms. Using this solution design we can analyze and verify the asymptotic time complexity of each sorting algorithm.

The sorting algorithms to be studied:



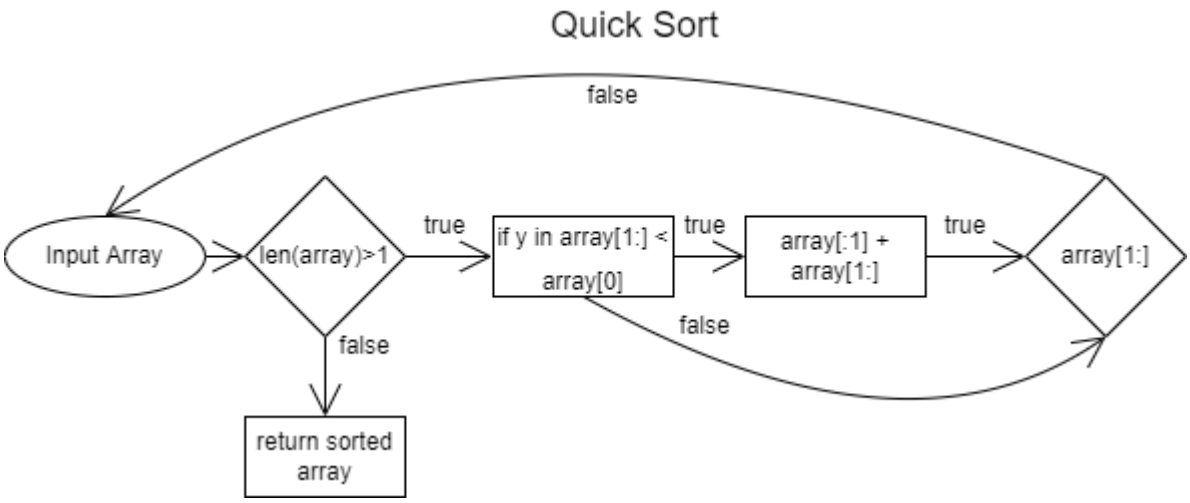
Time Complexity

Bubble sort is built on the concept of “bubbling” the highest value element to the back of the array. Then, the next highest element to the back.

Best **O(n)**: list is already sorted so no elements are carried to the back of the array

Average **O(n^2)**: most elements are “bubbled” to the back of the array, but some remain the same

Worst **O(n^2)**: array is reversed, so all elements must be “bubbled” to the back of the array, one by one



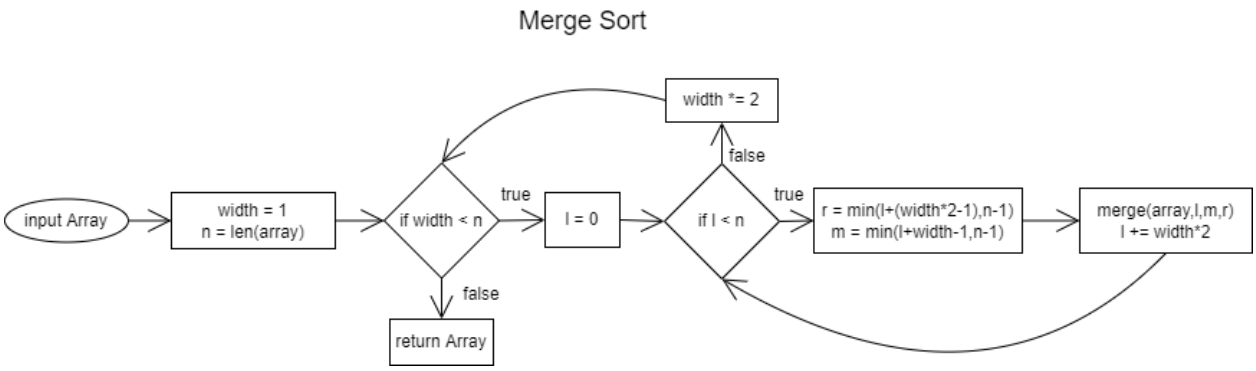
Time Complexity

Best **O(n log(n))**: sorted array will run through recursion process

Average **O(n log(n))**: array will run through recursion process so time complexity remains

Worst **O(n^2)**: each pivot is on either the smallest or largest number resulting in maximum number of recursive steps

Merge:



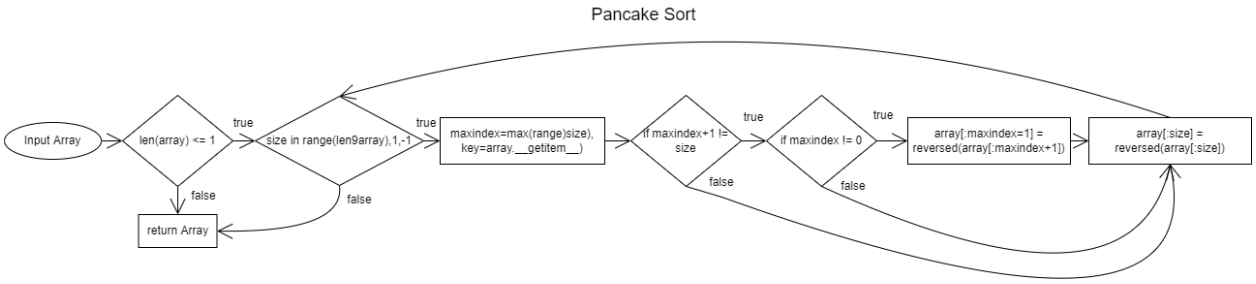
Time Complexity

Merge Sort divides arrays and takes linear time to merge back resulting in a time complexity of o(n log(n)) for all case scenarios.

Best: **O(n log(n))**

Average: **O(n log(n))**

Worst: **O(n log(n))**



Time Complexity

Best **O(n)**: sorted array results in inner loop being linear

Average **O(n^2)**: this algorithm is best thought as a stack of pancakes, and to sort those pancakes from widest to smallest the algorithm inverts pieces of the stack of pancakes until the smallest is on top and the largest is on bottom, leading to a sorted array

Worst **O(n^2):** the array alternates smallest to largest, which results in n iterations with n iterations inside of each of those

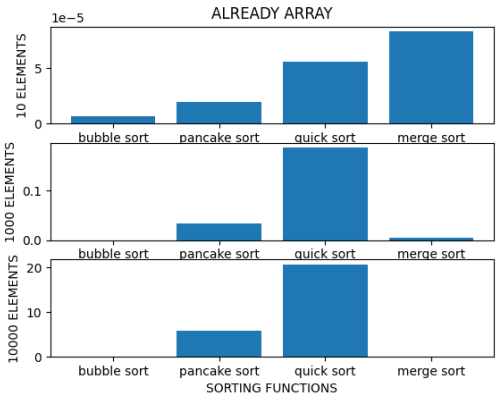
Design of experiments:

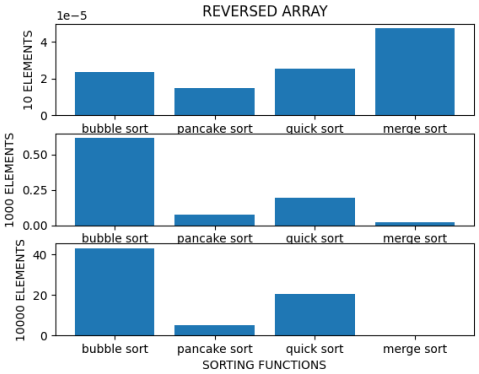
To design and test the time complexity of the algorithms we first needed a way of getting data into each of the algorithms. For this we created a few different array methods, the main one to be used will be the create_random_array which takes in a size parameter and returns an array of that size filled with numbers between 1-10000000000 in a random order. We also have array methods create_sorted_array_with_one_mistake, create_reversed_array, create_sorted_array, and create_array_of_one_value. These methods came in handy when testing our sorting algorithms under different conditions. To display our output data our function takes in an array method and a title. We call it with the different array creation functions. The create_graph function then calls each of the sorting algorithms on the array, it then changes the array size and repeats the process two more times. This results in three bar graphs comparing the time for each sorting algorithm, and each graph represents a different sized array. Other subtasks would include the merge method that is implemented in the merge sort. This method needed to be created individually so that it could then be called within the sorting algorithm.

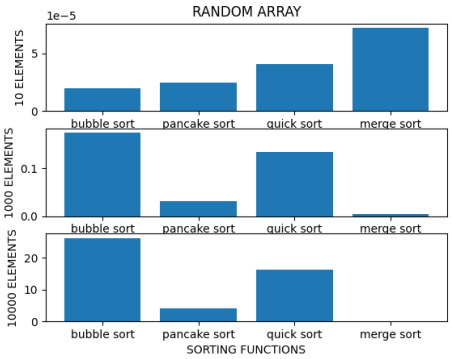
Implementation and testing:

For this project our team equally worked together to achieve our final product. Dakota and Zach took the time to handle most of the implementation. Cole handled most of the testing and would report back to Dakota and Zach, and was also in charge of doing the report.

Analyzing output data:







Looking at our output data visually we see a few details. With a low number of elements pancake sort and bubble sort both were faster than quick sort and merge sort. Once we reach 1000 elements, bubble sort takes much more time and merge sort takes much less time resulting in pancake sort and merge sort having faster times than bubble sort and quicksort. Quick sort surprisingly stayed consistent with its speed no matter the size of the array, however, it did consistently take the longest when the array was already sorted. Bubble sort has an average of $O(n^2)$ so as more elements are introduced it takes more time. Merge sort has a time complexity of O(nlog(n)) so it isn't surprising how it is the slowest when few elements are present, and it is easy to see that it quickly becomes more efficient than the other algorithms as

the number of elements increases. The pancake sort is surprising because as the elements increased the pancake sort got significantly better compared to the other algorithms.

Summary:

Overall, there are many different sorting algorithms, the question as to which is the best to use can have many factors. From this project we've learned that one major factor is the size of the array, with small arrays bubble sort is superior, however it quickly becomes obsolete with as much as 1000 elements and it is beat out by all of the other algorithms. With larger arrays, merge sort is superior, but on smaller arrays the merge sort is inefficient. Through this project we have grasped a better understanding on how algorithms work, the effect of time complexity, and how to implement both together to achieve a functioning test suite to examine the results of our program.