CSC 440 Programming Assignment

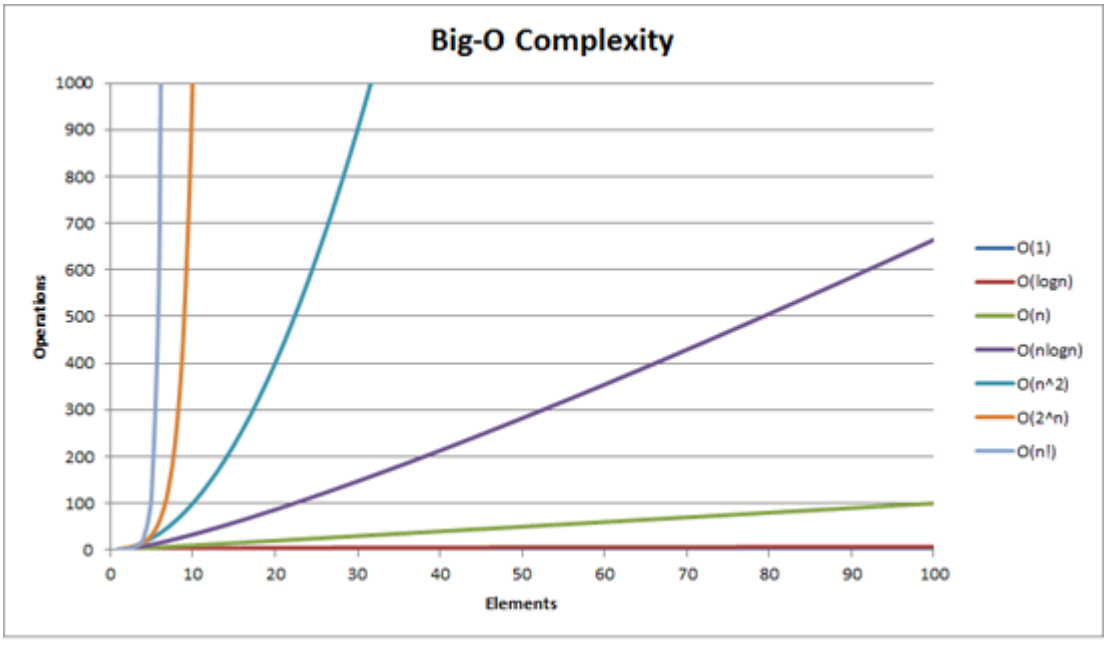
Nathan Larson

November 1, 2018

1 Introduction

The goal of this assignment is to implement five different searching algorithms to solve the maximum contiguous subvector problem using Python 3. Each of these five algorithms will be tested against the same **driver.py** script to compare execution times. In addition to testing and plotting the results required for the assignment, I ran my own tests using a linear increase in n to be able to plot the data on a graph and get a solid visual representation of the time complexity.

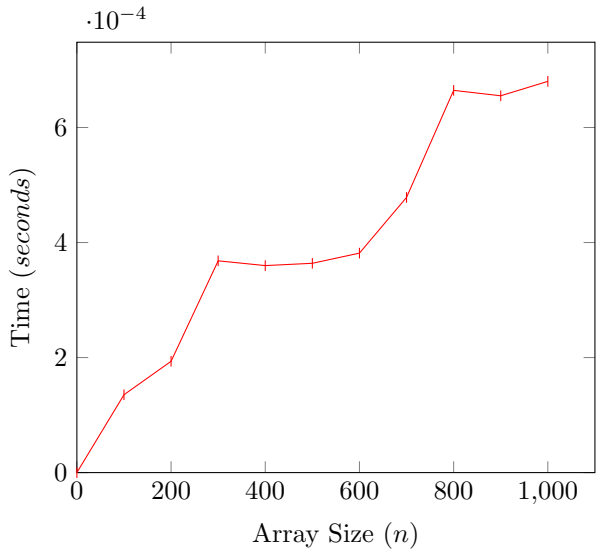
I used the following image to compare the run-time complexity of each of the algorithms from the tests to their theoretic complexity in order to gauge how similar they were.



2 Linear Algorithm: $O(n)$

The linear algorithm performed as expected for the original run times, although I was unable to test for the $1,000,000,000$ value of n due to getting a memory error when trying to run the command *python driver.py linear_algorithm 1000000000*. Besides this one problem, the algorithm increased very closely to linear time, from the values $100,000$ to $1,000,000$ the resulting time even increased almost exactly by a power of 10 from 0.0548 to 0.5408.

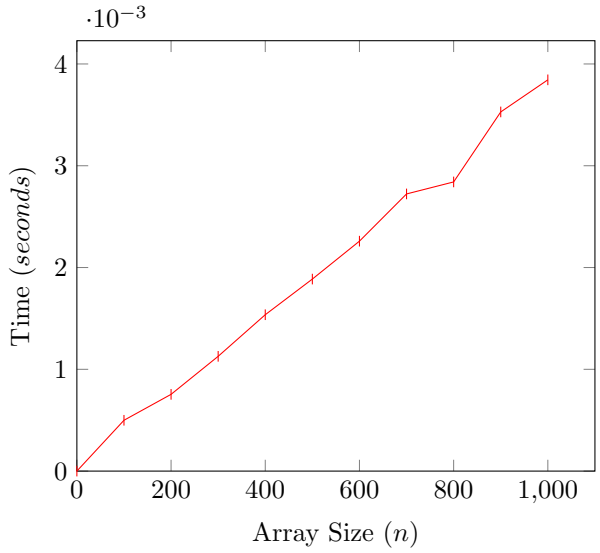
To graph the algorithm, I increased n linearly starting at 100 and increased by 100 each iteration up to 1000. Although the results were not a perfect straight line, it can be seen that the time increases relatively linearly as n increases linearly. I believe these results look skewed due to how small the changes in time are relative to changes in n, as well as unexpected computer-related occurrences such as random updates during certain tests.



3 Divide and Conquer Algorithm: $O(nlog(n))$

The divide and conquer algorithm performed as expected for the original run times. I implemented this algorithm using recursion, with one base function *div_and_con_algorithm(array)* and two additional functions to be called recursively. The first additional function *maxSubVectorSum(array, low, high)* is the main recursive function that returns the max sum contained in $(array[low], array[high])$. The second additional function *maxSubVectorSumWithMid(array, low, mid, high)* that returns the max sum in $(array[low], array[high])$ that also contains the element at $array[mid]$.

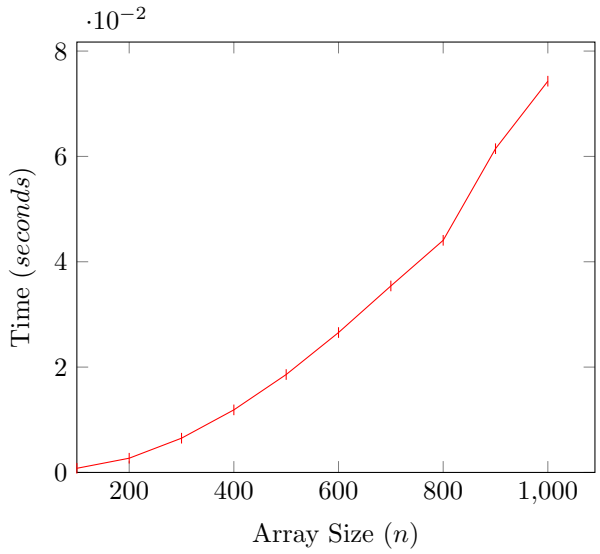
To graph the algorithm, I increased n linearly starting at 100 and increased by 100 each iteration up to 1000. The resulting graph followed the general path of an algorithm with a complexity of $O(nlog(n))$.



4 Quadratic Algorithms

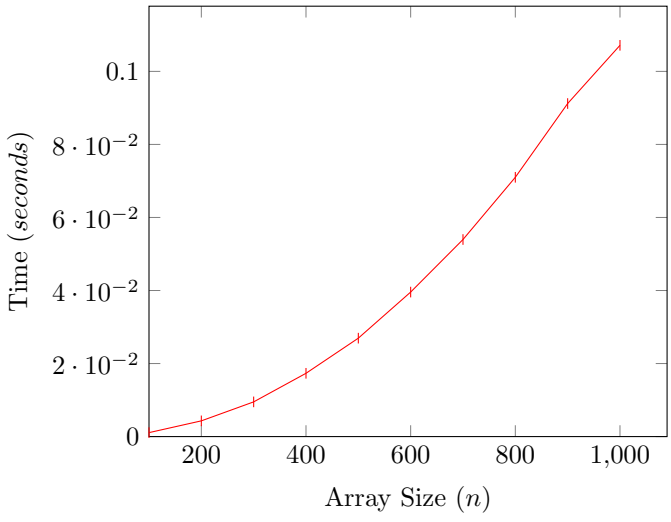
4.1 Quadratic Algorithm 1: $O(n^2)$

The first quadratic algorithm performed relatively similar to the theoretic performance, increasing very quickly as the value of n increased. To graph the algorithm, I increased n linearly starting at 100 and increased by 100 each iteration up to 1000. The resulting graph definitely followed the general shape of a quadratic curve, the run-time taken increasing faster as the input array size grew linearly.



4.2 Quadratic Algorithm 2: $O(n^2)$

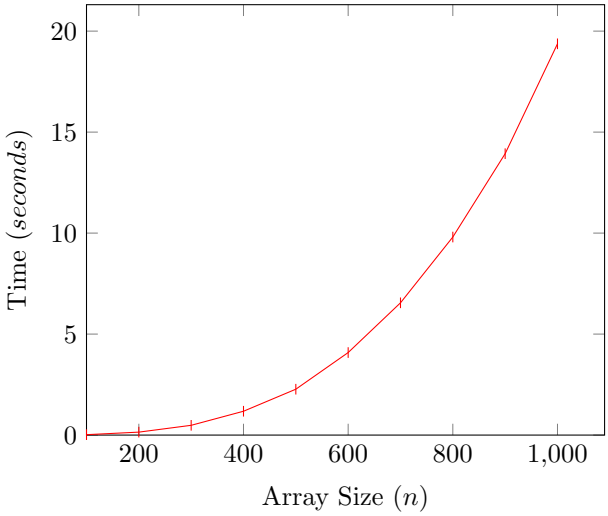
The second quadratic algorithm also performed similarly to the theoretical performance of an $O(n^2)$ algorithm. Although both quadratic algorithms have a time complexity of $O(n^2)$, this algorithm required an additional loop and computation resulting in a slightly larger run-time relative to the first quadratic algorithm. To graph the algorithm, I increased n linearly starting at 100 and increased by 100 each iteration up to 1000. This resulted in a very quadratic looking curve with slightly larger run-times than the previous quadratic algorithm, same as the other tests.



5 Cubic Algorithm

The cubic algorithm performed just as expected from the theoretical performance, with the run-time growing much quicker than any of the other algorithms. Even the slower quadratic algorithm was able to compute the max subvector of an array of size $n = 1000$ in 0.1128 seconds, while the cubic algorithm took almost 20 seconds.

To graph the algorithm, I increased n linearly starting at 100 and increased by 100 each iteration up to 1000. This resulted in an almost perfect cubic curve visualizing how quickly the run-time grows exponentially as the input array size grows linearly. I believe that this graph followed the theoretical performance the best since the run-time grows so quickly, the smaller input sizes used to test are able to represent the trend accurately.



6 Report

6.1 Run Times

n	10	100	1,000	10,000	100,000	1,000,000	10,000,000	100,000,000	1,000,000,000
Linear	0.0002	0.0003	0.0006	0.0057	0.0548	0.5408	5.2571	54.566	x
div_and_con	0.0001	0.0004	0.0036	0.0428	0.4759	5.4906	59.674	633.6	x
quadratic1	0.0001	0.0008	0.0722	7.1291	725.42	x	x	x	x
quadratic2	0.0001	0.0012	0.1128	11.412	1184.6	x	x	x	x
cubic	0.0001	0.0206	19.198	x	x	x	x	x	x

6.2 Questions

1. What CPU and RAM are you using to run your code? What are the speeds for the processor and the memory?

   I am using my desktop computer to build, run, and test all of the code used for this assignment:

   CPU: AMD Ryzen 5 1600x Six-Core Processor

   Processor speed: 3.60 GHz

   RAM: 16.0 GB

   Memory speed: 2134 MHz

2. What were the relative performances of each algorithm? Did you get the results you expected? Were the results in accordance with the theory (Did the run time of the linear algorithm increase linearly? What about the quadratic, cubic, and divide and conquer algorithms?). If not, why not?

   (Answered in corresponding sections above)

3. In python it is possible to implement the five algorithms almost exactly as they are written using the pseudo code in your book. Particularly it is possible to use the **max()** method to return the maximum of two numbers. How would you get around using this max function and how does implementing the algorithms without using the max function alter the run times of your methods? Can you explain the difference, if any?

   Without using the built-in **max()** function in python, I was able to implement all of the functions the same way with my own **myMax()** function(s). To be able to completely replace the **max()** function, I had to implement two different **myMax()** functions. One of functions takes two arguments as parameters, and the other takes three. It would be possible to override more **myMax()** functions, but only these two were needed to fully implement the algorithms for the assignment.

   After creating my own max functions, I replaced pythons **max()** functions with the ones I created and ran the same tests. Although there may be other system related occurrences playing a part, the tests resulted in a slightly faster runtime for the linear algorithm, which used the **myMax2()** function. On the other hand the divide & conquer algorithm used the **myMax3()** function and resulted in just about the same run-times versus using pythons built-in **max()** function. I believe that the linear function may have run slightly faster due to less overhead without an external function call, but comparing three different values for the divide and conquer algorithm was a little more intensive resulting in very similar results as from using the **max()** function.

```
# calculates the max value of 2 values
def myMax2(num1, num2):
   if(num1 > num2):
      return num1
   else:
      return num2

# calculates the max value of 3 values
def myMax3(num1, num2, num3):
    temp_max = num1
    if(num2 > temp_max):
        temp_max = num2
    if(num3 > temp_max):
        temp_max = num3
        if(num2 > num3):
            temp_max = num2
    return temp_max
```

4. What are the storage requirements of each algorithm?

   Linear: Since the linear algorithm uses the same amount of memory regardless of input size, it has a space complexity of $O(1)$.

   Divide & Conquer: The amount of memory required for the divide and conquer algorithm increases linearly with the size of the input array. This is because as the size of the input array grows the number of recursive calls and frames on the stack grows at the same rate to store the elements, resulting in a space complexity of $O(n)$.

   Quadratic 1: The first quadratic algorithm uses the same constant amount of storage regardless of the input size since it simply iterates through the array, calculating and comparing the max subvector on the fly. This results in a space complexity of $O(1)$.

   Quadratic 2: Since the second quadratic algorithm calculates and stores the cumulative sum of the array before calculating the max subvector, it requires additional space for this array that will grow as the input array grows, resulting in a space complexity of $O(n)$.

   Cubic: The cubic algorithm, although it has a terribly inefficient time complexity, requires the same amount of storage regardless of the size of the input since it only stores a running max throughout its run-time. This results in a space complexity of $O(1)$.

   (a) One of the quadratic algorithms requires extra store, explain which one and why.

   The implementation of **quadratic_algorithm2** requires extra storage relative to the other quadratic algorithm. This is because it required another array of size $(n+1)$ in order to store the cumulative sum at each index in the original array, ie.

$$[\ A[0], \sum_{i=0}^{1} A[i], \sum_{i=0}^{2} A[i], ..., \sum_{i=0}^{n} A[i]\ ]$$

   (b) The divide and conquer algorithm also requires extra storage, explain why.

   The divide and conquer algorithm requires extra storage because it is a recursive algorithm, thus requires more memory on the stack. This is due to the fact that at each recursive call a new call stack frame is creating, requiring more and more memory as the array grows and potentially leading to a Stack Overflow.

   (c) Finally, supposing you were reading input from a file, describe how it would be possible for the linear algorithm to use less storage than loading the full data into an array. How would this change in loading the data affect the run time of the algorithm?

   The linear algorithm could potentially use less storage than loading the full data into an array because it only requires one element at a time for each iteration. Applying the linear algorithm to reading a file, it would only need keep track of the element index that it is reading and storing the element that is at the current index, rather that reading and storing the entire file into an array. This change in loading the data would not change the complexity of the algorithm, but would most likely increase the run-time as it must load each element from the file individually rather than simply getting the element by index from an array.

5. Research the python library **timeit** (docs.python.org/3/library/timeit.html). The documentation states that that the **timeit** library "avoids a number of common traps for measuring execution times". Discuss what these common traps are. What system effects might cause the measurement of the run time of an algorithm to be inaccurate. How does the **timeit** library overcome these traps and system effect? How does the **timeit** library fail to control for system effects? You must cite at least one source (for instance a discussion from stack overflow, or Wikipedia article).

   One of the common traps for measuring execution time is not considering compiler processes in the background such as *garbage collection* that can have an impact on the run-time of an algorithm. The **timeit** documentation states that it "temporarily turns of garbage collection during timing" in order to get a more accurate run-time. Another common trap that **timeit** takes care of is setting up the run-time code and not including the time taken to setup in the final measured time.

   The documentation also mentions that various system effects can cause inaccuracies in the run-time. These system effects can include other processes running on the computer interfering with the accuracy of timing as well as the hardware the program is running on. If one device has a different CPU frequency or architecture than another, it can have a noticeable effect on how quickly processes can be executed resulting in different execution times for the same program.

   Another common trap stemming from the system is how time is measured. On most platforms the default timing function uses the "wall clock time" to measure time rather than the CPU time. This can result in the same effect of other processes potentially interfering with the measured time of the program. On windows the **time.clock()** function would be the best timer while most other platforms would use the **time.time()** function.

   Citations

   1.) https://stackoverflow.com/questions/1685221/accurately-measure-time-python-function-takes

   2.) https://stackoverflow.com/questions/7304875/does-the-frequency-of-machine-effect-execution-time-of-my-code