605.621 Foundations of Algorithms Spring 2024
Project Assignment #3

1. **a:** Write pseudocode for a modification of PARTITION(A, p, r), pg. 184, using median-of-three partitioning.

```
MEDIAN-OF-THREE-PARTITON(A, p, r)
 1  k = ⌊(p + r)/2⌋
 2  if A[p] > A[k]
 3        exchange A[p] with A[k]
 4  if A[p] > A[r]
 5        exchange A[p] with A[r]
 6  if A[k] > A[r]
 7        exchange A[k] with A[r]
 8  exchange A[k] with A[r - 1]
 9  pivot = A[r-1]
10  i = p - 1
11  for j = p to r - 2
12        if A[j] ≤ pivot
13              i = i + 1
14              exchange A[i] with A[j]
15  exchange A[i + 1] with A[r-1]
16  return i + 1
```

Natural Language Description of Algorithm

The algorithm starts by accepting the inputs A, p, r. A is the array/set that is passed into the algorithm, p is the start index, and r is the end index. It begins by selecting a segment of an array between indices p and r and calculates the median of the first element, the last element, and the middle element to use as the pivot for partitioning. It then rearranges the chosen elements to ensure the median value is positioned correctly and then swaps the median element with the element just before the last, marking it as the pivot. Starting with an initial partition index i set just before the start index, it iterates through the array segment, moving elements smaller than the pivot to the left side, effectively partitioning the array around the pivot. After all elements are compared against the pivot, the algorithm swaps the pivot into its correct sorted position within the array. Finally the algorithm returns $i+1$ which is the final position of the pivot element post partitioning.

2. **b:** What is the worst-case asymptotic behavior of QUICKSORT(A, p, r) using median-of-three partitioning? Provide proofs supporting your conclusions.

   Lets start by examining what would make it a worst-case. A wost case would be where for each partition the pivot does not split the array into two even halfs. Not only that but it would split it into one large partition and one small/empty partition.

   With this we can begin to flesh out a recurrence relation for the worst case, lets call it T(n). The time to sort an array of size n can be expressed as the time to sort the subarray of size n - 1, T(n-1), plus the linear work O(n) needed to partition the array. Thus we can say $T(n) = T(n-1) + O(n)$. We can then expand this recurrance as such:
   $T(n) = T(n-2) + O(n-1) + O(n)$
   $T(n) = T(n-3) + O(n-2) + O(n-1) + O(n)$
   ...
   $T(n) = T(1) + O(2) + O(3) + ... + O(n-1) + O(n)$
   T(1) would run in O(1) time as this would be a base case of a single element so we have
   $T(n) = O(1) + O(2) + O(3) + ... + O(n-1) + O(n)$
   $T(n) = O(1 + 2 + 3 + ... + n - 1 + n)$
   $T(n) = O(\frac{n(n+1)}{2})$
   $T(n) = O(n^2)$

   Thus we have proven that the worst case time complexity of quicksort, even when using median of three partitioning is $O(n^2)$.

3. **c:** What is the worst-case asymptotic behavior of QUICKSORT(A, p, r) using median-of-three partitioning on an input set that is already sorted? Provide a proof supporting your conclusion.

Having an already sorted array definitely changes the worst-case asymptotic behavior of QUICKSORT utilizing median-of-three partitioning. Unlike in the previous question in the partitioning step median of three will always choose the middle element of the array. Due to this the partitioning will be split evenly or as close to even as you can get with a set (ie a set of length 7 cannot be split perfectly even). From this we can determine the recurrence relation. As we know when the array is divided it is divided into two separate subproblems of equal size, half the size of the original problem. This can be represented as 2T(n/2). Once it is divided it then has to be combined which would take $\Theta(n)$ time as we have to compare each element with the pivot. Thus we can find the recurrence relation to be $T(n) = 2T(\frac{n}{2}) + \Theta(n)$.

This relation may seem familiar, because it is a form of the master theorem, specifically this relation falls under case 2 of the Master theorem. This means we need to determine a and b. a would be the number of subproblems or 2. b would be the factor by which the subproblem is reduced also 2. f(n) would be the cost of dividing the problem and combining the solutions of the subproblems, which would be $\Theta(n)$. With that we can start plugging items into the master theorem
$f(n) = \Theta(n^{log_b a})$
$f(n) = \Theta(n^{log_2 2})$
$f(n) = \Theta(n)$
Which holds true meaning we can find T(n) by plugging into this equation $\Theta(n^{log_b a} logn)$, which gives us $T(n) = \Theta(nlogn)$. This means that the worst case time complexity is $O(nlogn)$.

4. **e:** Test your implementations to verify analytically derived complexity bounds or bounds developed on pgs 183-190:

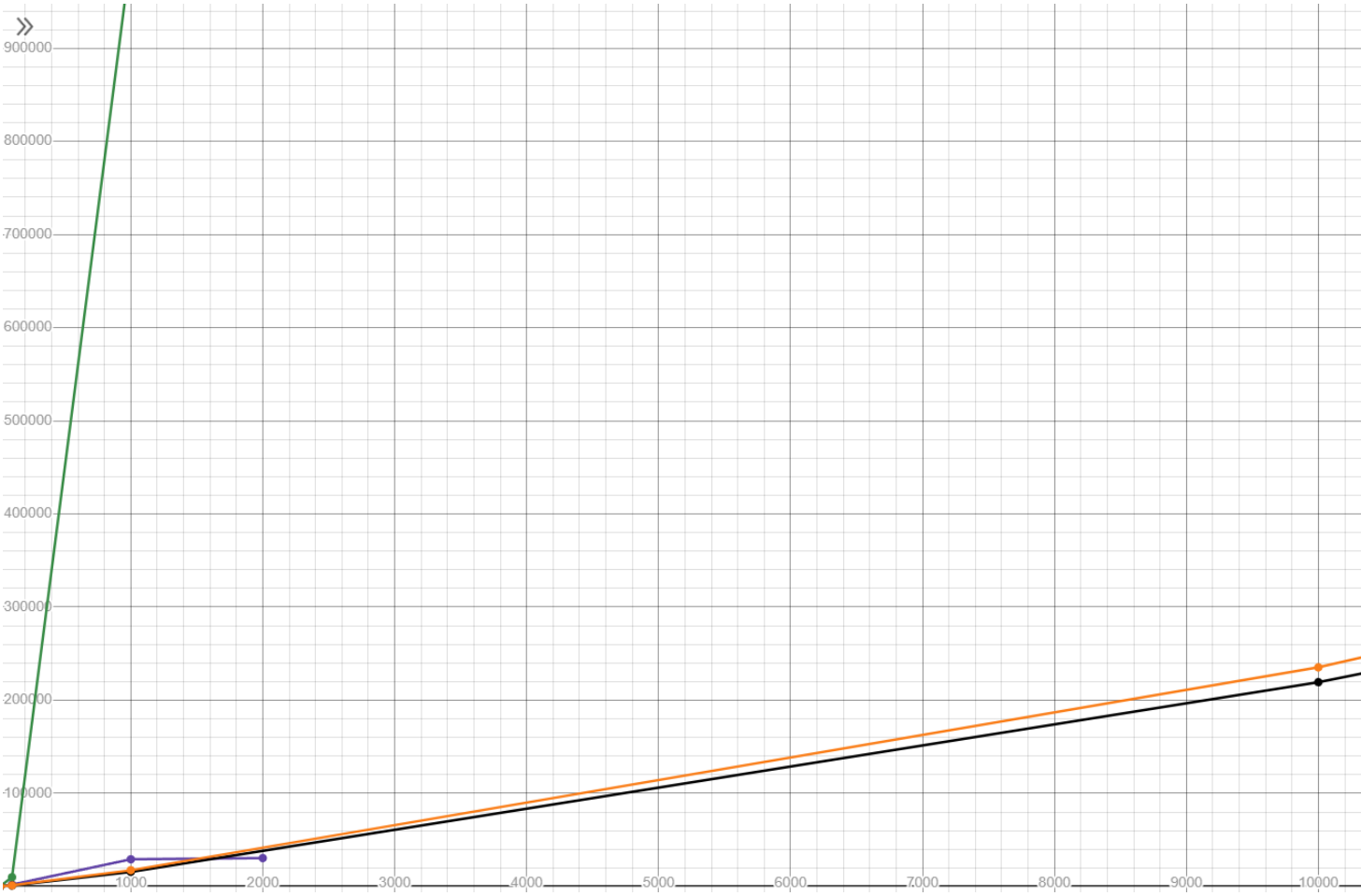
   There isn't too much to write for this part I just wanted to explain how I went about this. To create a worst case scenario is not the easiest to do manually. However, for the basic quick sort this wasn't as hard. Giving Quicksort with the basic Partitioning Scheme an already sorted array, is its worst case scenario (or at least one of them). In contrast this is not specifically the worst case for Quicksort with median of three partitioning. However, as we did do a worst-case analysis on Quicksort with median of three partitioning with an already sorted array I felt that doing this for the code would be sufficient. These outputs can be found in Tests/BB_Already_Assorted_Tests.txt, and graphs below.

5. **g:** Analysis comparing your algorithm's worst-case asymptotic behavior to the achieved asymptotic behavior of your programs [1(b), 1(e)(ii)] .

   I will be doing this with two separate graphs just to get the whole picture. To start lets look at the Worst case running times that I found against the actual running time of the program.

In this graphic it is good to note which lines which.
Red: $O(n^2)$
Blue: $O(nlgn)$
Purple: Quicksort with MOT (Sorted Arrays so "worst case)
Green: Quicksort No MOT (Sorted Arrays so "worst case)

Now lets look at the worst case against the achieved asymptotic behavior.



In this graphic it is good to note which lines which.
Black: Quicksort with MOT (Random Arrays)
Orange: Quicksort No MOT (Random Arrrays)
Purple: Quicksort with MOT (Sorted Arrays so "worst case)
Green: Quicksort No MOT (Sorted Arrays so "worst case)

So what does all this mean? How do they compare?

Well as you can see in graph one, the already sorted arrays really affect Quicksort that doesn't utilize MOT. The graph also proves what I determined earlier. A Quicksort utilizing MOT can mitigate the effects of the worst case and ends up running close to $O(nlgn)$ time. One without that does run at its normal worst case time of $O(n^2)$. Unfortunately due to recursive calls I could only run up to 2000 data points before the code stopped functioning (ie my computer told me that it hit max recursive calls, I changed max recursive calls, nothing would print out to my file). Despite that I think a good enough sense of how the program is running can be seen on the graph.

Finally how does the worst case against the achieved asymptotic match up. For normal quicksort without MOT we see a drastic impact from worst case to average case. Its worst case graph is close to $O(n^2)$ but when running against normal unsorted arrays it was keeping up with Quicksort with MOT at around a $O(nlgn)$ time. This is much different than Quicksort with MOT as it seems it was able to mitigate the worst case to maintain a close to $O(nlgn)$ time. It also continued to be able to achieve that time with the normal data.

References:

1. Desmos. (2011). Desmos Graphing Calculator. Desmos Graphing Calculator; Desmos. https://www.desmos.com/calculator

