Programming Project 2

CS146 Data Structures and Algorithms

Benjamin Garcia Section 02

Maximum Subarray Problem

What Is It:

The maximum subarray problem (or largest sum contiguous subarray) is a problem to find the sum of a contiguous subarray within a one-dimensional array of numbers which has the largest sum. In this project, we will explore 3 different algorithms to solve this problem.

Application:

The application for this problem is if you are a free lancer planning to work at a resort for some part of the summer season. However, there isn't enough to pay you every day while requiring you to pay your expenses. But you know in advance how much you will earn on any given day. So, you want to maximize your earnings by choosing an arrival and a depart day that will give you the most money working through all those days.

Part A: Brute Force Algorithm

Task to be Completed:

In this part of the project, we are asked to implement the solution to the maximum subarray problem by using a brute force algorithm. This means considering every possible pair of arriving and departing dates and choosing which pair results in the largest sum. This algorithm is $O(n^2)$. It is the longest algorithm of the 3 we will be working with in this project.

Solution:

The way this is accomplished is by having two for loops, one nested in the other. We also need two variables: one to store the maximum sum starting at a specific arrival date and one to store the overall maximum so far. The outer loop picks the beginning element (the arrival date) and the inner loop finds the maximum possible sum based on that arrival date from the outer loop. The outer loop will start from index 0 of the array all the way to the end, while the inner loop starts at the index of the current instance in the outer loop all the way to the end of the array. For each new instance of the outer loop, the temp sum is set to 0 and for every instance of the inner loop, the element of the array at the given position is added to temp sum. It then calculates if the temp sum is greater than the overall max. If it is, it will first set the overall max to that temp max value and then save the current arrival date (given by the index of the outer loop), depart date (given by the index of the inner loop), and the max given by the value of the overall max that was just set. If not, it will move to the next instance of the inner loop doing the same thing until the end of the array. After this, the outer loop will move to the next arrival date and the process repeats until the arrival reaches the end of the array. Once the maximum is found throughout the entire array, a new object of type MaxSubArray is created that can hold the max sum, arrive date, and depart date. This entire object is then returned.

Cases to Consider:

In this algorithm, there were not too many cases to consider since the brute force algorithm is straight forward and will just compare every possible subarray. The only case I had to consider was if all elements were negative, then I would return a MaxSubArray object with the max sum = 0 and both the arrival and depart dates set to 0 which indicated no days were selected.

Problems I Ran Into:

The only problem I ran into was the professor wanting us to return the maximum sum along with the corresponding arrival and depart date. To do this, I created a new object called MaxSubArray which held 3 variables for each thing that needed to be returned. With this done, I could return the whole object and when needed to, I was able to access each of the 3 variables through methods of that object class (getMax, getArrive, getDepart).

Part B: Divide and Conquer Algorithm

Task to be Completed:

In this part of the project, we are asked to implement the solution to the maximum subarray problem by using a divide and conquer algorithm. This means that we continuously split the array into 2 parts until each side reaches its base case and compute each side on its own before comparing it to the other. This algorithm is O(nlgn). This algorithm's run time sits in the middle of the 3 we will be working with in this project.

Solution:

The way this is accomplished relies on the fact that the maximum sub array will either completely fall in the left half of the array, completely fall in the right half of the array, or will cross the midpoint and have the arrive on the left with the depart on the right. The first two cases are simple recursive calls which calls the divide and conquer algorithm again which keeps splitting the array down to its base case. The base case is when the index of the left most element equals the index of the right most element which means it is only looking at one element. When this happens, the max is the value of the element at that index in the array. This also means the arrive and depart date are both this index. The harder part is the crossing section. This part is done in linear time because we find the maximum sum starting from the midpoint and ending at some point to the left of the midpoint and then find the maximum sum starting from the midpoint + 1 and ending with sum point on the right of that. Then, combine the two and that is the max of the sub array that crosses the midpoint. The point on the left of the middle is the arrive date of the maximum sub array crossing the middle and the point on the right is the depart date. Now, compare all 3 cases and return whichever one is larger.

Cases to Consider:

The cases I had to consider were if two of the cases were the same because I originally had the logic setup so that it would return leftMax if it was greater than rightMax and crossMax, return rightMax if it was greater than leftMax and crossMax, return crossMax if it was greater than leftMax and rightMax, and return a MaxSubArray object with max sum, arrive date, and depart date all equal to 0 if it did not fall under one of those conditions. This would mean if two cases were equal, it would resort to picking no days. So, I changed it so all the if statements were greater than or equal to meaning that it will return the first max in my logic if there are two cases with the same sum.

Problems I Ran Into:

The problem I ran into was figuring out how to compute the third case (the maximum crossing array). I was not sure how the logic should work until I realized it had to include the middle so I would just find the max from the middle out to the left, the max from the middle + 1 out to the right and add the two together.

Part C: Kadane's Algorithm

Task to be Completed:

In this part of the project, we are asked to implement the solution to the maximum subarray problem by using Kadane's algorithm. This algorithm is O(n). This algorithm's run time is the fastest of all 3 algorithms we will be working with in this project.

Solution:

The way this is accomplished is by using 3 variables: one to hold the overall max sum, one to hold a temporary max, and one to hold a temporary arrive date. We initially assume that the maximum subarray starts with index 0. There is a for loop starting from index 0 up to the last element. The element at the index specified in the for loop is added to the temporary max. If that temporary max goes below zero, we reset the temporary max and set the new arrive date to the next index. It is then calculated if the max sum is less than the temporary max, set the max sum to the value in the temporary max, set the depart date to the current index and set the temporary arrive to the value of the arrive date. After this for loop repeats until the last element, then set the arrive date to the value of the temporary arrive. The max sum of the entire array will be stored in the max sum variable and the depart date will be stored. Then, return a MaxSubArray object with these 3 variables saved.

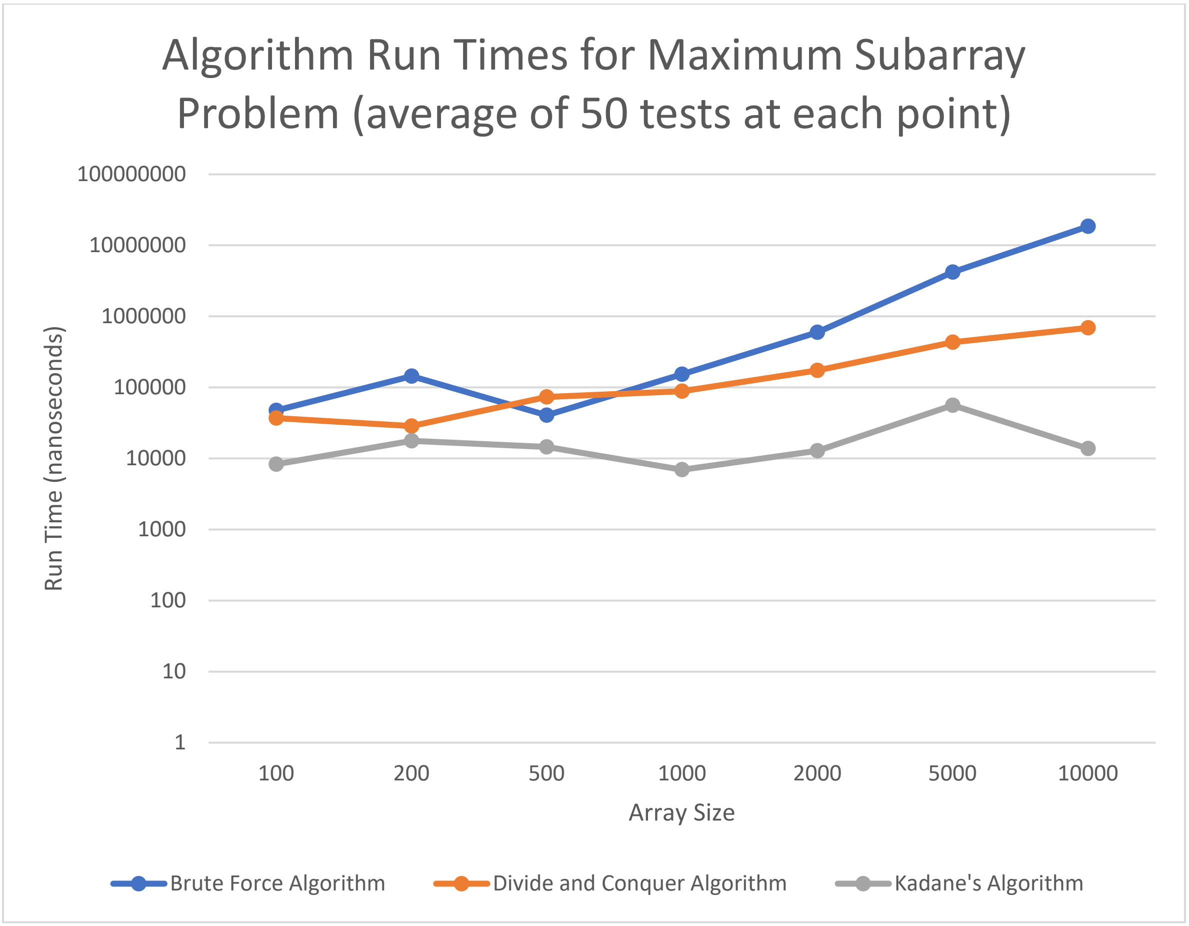
Cases to Consider:

There weren't any cases I had to consider since the algorithm worked in linear time and was very straight forward to follow for all cases. The only one as previously mentioned is if all subarrays resulted in a negative sum, return a MaxSubArray object with the max sum, arrive date, and depart date all set to 0.

Problems I Ran Into:

The only problem I ran into when writing this algorithm was understanding why it worked. After I wrote it out and traced the values it made sense, but it was not very intuitive when I first learned the algorithm.

Part D: Algorithm Run Times



Array Size	Brute Force Algorithm Runtime (nanoseconds)	Divide and Conquer Algorithm Runtime (nanoseconds)	Kadane's Algorithm Runtime (nanoseconds)
100	47714	37146	8280
200	143632	28644	17794
500	40646	73434	14488
1000	153192	88910	6948
2000	599556	174480	12802
5000	4181862	434622	56184
10000	18470960	689438	13802