Algorithms, Data Structure and Software Engineering for Media Technology Mini Project: The Maximum-subarray problem

Nolan Sejr Fledelius Rux
20173907
Aalborg University
Medialogy 8th Semester

Abstract:

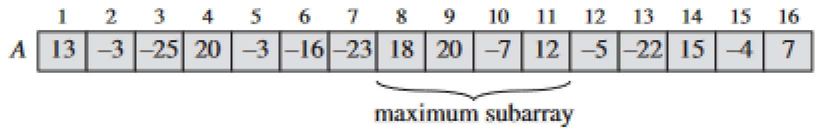
This Algorithms, Data Structures and Software Engineering for Media Technology Mini-Project is focused on the maximum subarray problem, wherein I was tasked with finding a contiguous subarray within a given array (for example A[i....j]), and finding the sum of the elements from said array such that a maximum is calculated. In order to solve this I implemented the Divide-and-Conquer and Kadane's algorithm to find the maximum in the most efficient manner possible. This paper will begin with an Introduction to the problem, the two methods for solving it (Divide-and-Conquer and Kadane's) along with explanations and pseudocode representing them both. Finally, both algorithms will run the same set of arrays, and afterwards I will determine with time-stamps and memory usage which is more efficient to utilize.

Table of Contents:

Introduction:

As stated in the abstract, this Mini-Project will be focusing on the “Maximum subarray problem”. Before we can discuss solving this solution, we should mention the ‘’contiguous subarray’’ which is in this case a set of values or numbers within a given array. When referring to the fluctuation of prices in a graph, we want to find the values of said contiguous subarray with the largest sum, referred to as the maximum subarray.

This array problem is described in Cormen et. al. as a method of determining the largest sum of a contiguous subarray. This type of problem is limited to only one-dimensional arrays (ex. [i...j]). According to Cormen et. al., the theory behind this is that the user is desiring to visualize the maximum value within a foreseeable period, such as the price of stocks in a market. Initially, it might make sense to ‘’brute force’’ the solution and make guesses on possible maximums, but this can be considered exponentially more time consuming, inaccurate, and inefficient. Therefore, the solution needs to be devised, transformation, with an $o(n^2)$ running time, $o$representing the complexity of the algorithm and n representing the input size. The difference from bute-forcing this solution is that we want to look at the changes, instead of the set values provided in the array (Cormen et al., 2009, 69).



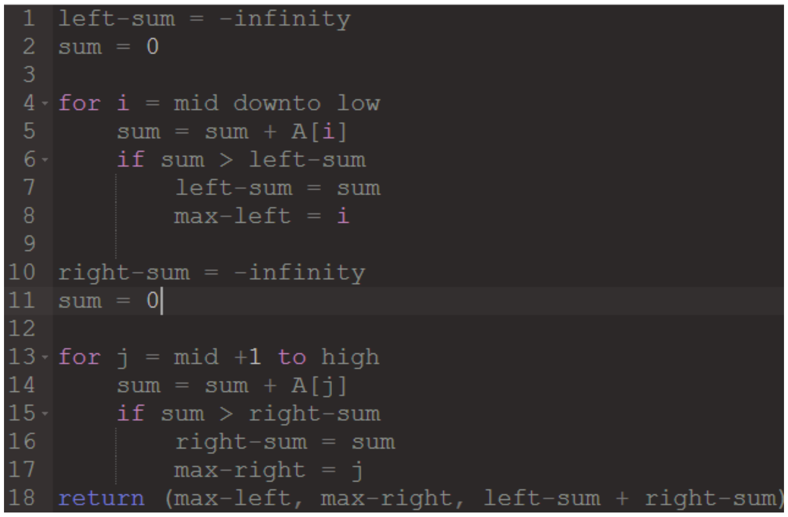
This example shows the maximum subarray,
and greatest sum possible 43 within this contiguous subarray (Cormen et al., 2009, 70).

Despite this, transformation will not accurately and efficiently output maximum subarrays, as it is not accurately checking the period of subarrays within n days, since we are, essentially, making guesses based on the information presented to us. More specifically, if an array had no negative values, then the entire array would be considered a maximum. Therefore, we must look for ‘a’ maximum subarray instead of the definitive one (Cormen et al., 2009, 70).

Below are the two algorithms I implemented in order to solve the maximum subarray: Divide-and-Conquer, and Kadane’s Algorithm.

Divide-and-Conquer

As mentioned in the Abstract, I will be using the Divide-and-Conquer algorithm provided in Cormen et. al. in an attempt to solve the algorithm. The idea of this is that we want to distinguish a set of subarray's rather than a definitive subarray. In order to determine this, "Divide-and-Conquer"'s theory functions that a maximum subarray is divided between the middle (called the 'midpoint'), afterwards determining the minimum and maximum of each separate point (i.e from low to mid, then mid +1 to high). Pictured on the right is the 'pseudocode' for the functionality of finding the max-crossing subarray (Cormen et al., 2009, 71). Lines 1 - 8 determine the 'left' of the array (A[low-mid]), because the array requires a midpoint, the for loop beginning at line 4 gives an index 'i' and begins from the midpoint down to the lowest value. It is nearly identical from lines 10-17 for the 'right' side of the array. The difference is that the right index is labelled 'j' and functions by calling the midpoint from the previous lines, but adds a digit point, therefore starting past the midpoint, moving to the high point of the array.

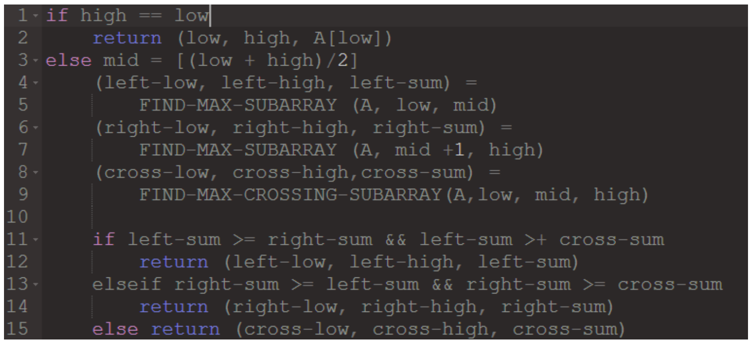


```
1  left-sum = -infinity
2  sum = 0
3
4  for i = mid downto low
5      sum = sum + A[i]
6      if sum > left-sum
7          left-sum = sum
8          max-left = i
9
10 right-sum = -infinity
11 sum = 0
12
13 for j = mid +1 to high
14     sum = sum + A[j]
15     if sum > right-sum
16         right-sum = sum
17         max-right = j
18 return (max-left, max-right, left-sum + right-sum)
```

Once both values are sorted, line 18 returns both the max points of both the left and right, as well as the summation of both the left and right, known as the maximum contiguous sum.

After determining the maximum-crossing subarray, we can implement the 'divide-and-conquer' algorithm to solve the maximum subarray! Pictured on the right is the pseudocode for the algorithm based on (Cormen et al., 2009, 72).

The first lines of this code is mainly for testing the base array (where there is only one element. Afterwards, lines 3-15 show the recursive case, starting with the 'divide' in line 3, by

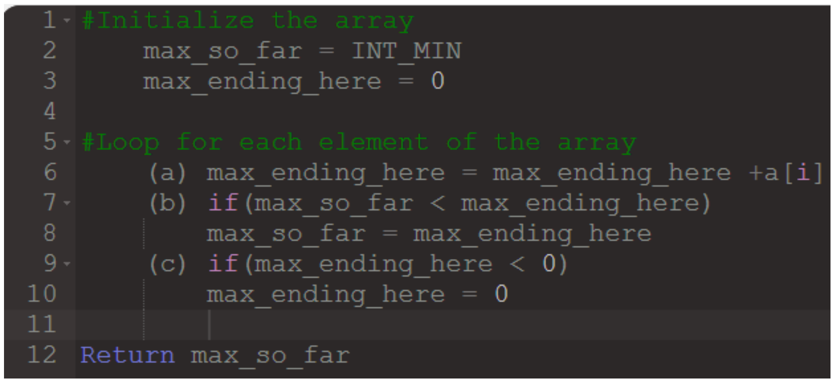


```
1  if high == low
2      return (low, high, A[low])
3  else mid = [(low + high)/2]
4      (left-low, left-high, left-sum) =
5          FIND-MAX-SUBARRAY (A, low, mid)
6      (right-low, right-high, right-sum) =
7          FIND-MAX-SUBARRAY (A, mid +1, high)
8      (cross-low, cross-high,cross-sum) =
9          FIND-MAX-CROSSING-SUBARRAY(A,low, mid, high)
10
11     if left-sum >= right-sum && left-sum >+ cross-sum
12         return (left-low, left-high, left-sum)
13     elseif right-sum >= left-sum && right-sum >= cross-sum
14         return (right-low, right-high, right-sum)
15     else return (cross-low, cross-high, cross-sum)
```

computing the midpoint and splitting the high points in two separate parts (left and right subarray). After the division, lines 4 and onwards begin the conquer stage of the algorithm, by recursively finding the maximum subarrays within both left and right portions that we found in the max-crossing subarray. Afterwards, we combine the two, line 8 finds the max subarray that crosses the midpoint. Lines 11 and 13 test whether the left or right subarrays contain a maximum sum, and if it does, they both return that maximum. If neither of them contain the maximum, then that means a max subarray must cross the midpoint in either the left or right respectively, causing line 15 to run and return the value (Cormen et al., 2009, 73).

Kadane's Algorithm

As mentioned in the introduction, alongside the Divide-and-Conquer algorithm, I will also be utilizing "Kadane's Algorithm" to determine the maximum subarray. Unlike Divide and Conquer, which separates the one-dimensional array and sorts from low to high points, Kadane's functions as a sequential algorithm that runs through the array and checks for positive contiguous segments. Every time a positive integer occurs, it is compared with the maximum value found within the array at that given point (Geeks for Geeks, 2021).



```
1  #Initialize the array
2      max_so_far = INT_MIN
3      max_ending_here = 0
4
5  #Loop for each element of the array
6      (a) max_ending_here = max_ending_here +a[i]
7      (b) if(max_so_far < max_ending_here)
8          max_so_far = max_ending_here
9      (c) if(max_ending_here < 0)
10         max_ending_here = 0
11
12 Return max_so_far
```

Pictured on the left is an example of Kadane's pseudocode in action. As mentioned, Kadane's functions by checking whether a specific array contains a positive integer, then sets it at the maximum value so far (seen in line 2). Before the code is running, we set the 'max_ending_here' to 0 depending on a certain array, for example, if a[i] = -2, a being the array in question. The algorithm as seen in the loop will check for each integer and see how it stacks up with the current maximum, if that integer surpasses the current maximum, that value will override the 'max_so_far' as defined in line 2. As seen in lines 6 through 10 in the pseudocode, this will run through the array and check for values, first by checking negative values, and comparing them to the initial integer of 0, moving though the entire array and sorting them one by one, until ultimately returning the 'max_so_far' to the user once the entire array has been sorted.

Essentially, Kadane's functions are similar to that of the Brute-Force approach, but it counts backwards and calculates the sum of the possible array by A[n-1] (Singhal, 2018).

Analysis of Runtimes and Comparison of Results in Performance

Before I can discuss the comparison of both algorithms sorting an array, I need to discuss both of the ‘runtimes’ of each.

Divide and Conquer

Firstly, the Divide-and-conquer average running-time complexity is $O(nLogn)$. In essence, when dealing with smaller arrays that go up to a certain extent, it is optimal to use this type of algorithm. However, since Divide-and Conquer is expressed in an exponential manner, it becomes less efficient and time consuming to use when dealing with larger datasets. As mentioned earlier, the algorithm first has to determine the midpoint of the one-dimensional array, and if it has to sift through a large amount of values, the time and energy spent increases along with the margin of error.

Kadane’s Algorithm

Kadane’s algorithm has a simpler runtime complexity of $O(n)$. While this can be considered less complex than Divide-and-conquer, it still requires the algorithm to systematically check each number in a sequential manner, akin to the Brute-force solution. However, this will fall within the similar pitfall as the Divide-and-conquer, where as the amount of integers increase, the time and complexity it will take to sort through each individual value. However, for smaller sizes of arrays, both should be able to perform their functions with little to no time used.

Implementation

For the purpose of this paper, both algorithms I implement will be in Python. Moreover, I will be utilizing the same array of numbers in the exact same order for determining the efficiency of both the algorithms. I also implement a timer system which will record the elapsed time it takes for the systems to both output the maximum subarray, beginning from the algorithm instantiation to the end. Moreover, since both the algorithms are quite efficient, I will have to move the recorded time to the 7th decimal place. In order to measure the memory usage, I implemented ‘tracemalloc’ to monitor the bytes used to calculate the contiguous subarray (Keith-Magee, 2019).

The number generated will originate from random.org, a website where it can randomly generate an array of integers in both the positive and negative spectrum of values. For this example, I implemented several arrays within the code, containing between 100, 500, 1000, and 10,000 entries, each with a range between -100 and 100, and ran the program to print within the console. Moreover, I will implement a data collection to understand how much memory is taken from each algorithm set. In the case of divide-and conquer, I opted to make changes to the Python code itself, such as instead of returning each point of the array in a sum of three parts, I feed all three within the function 'max' in Python (seen in line 44 of the code). Then, both Kadane's and Divide and Conquer, for the sake of a fair comparison will be given the same exact array, then analyzed for the elapsed time it took to return the maximum subarray sum.

After each algorithm runs through the array fed, I will perform it five times, then average out the time taken for each of the arrays (100, 500, 1000, 10000). I unfortunately could not find an efficient way to automatically select and run each array efficiently, therefore I opted to hard code the array via the driver program located at the end of the python script. The result will be found within the table below:

Average Elapsed Time and Data Usage (Seconds and in Bytes)	**Divide-and Conquer** $O(nLogn)$	**Kadane's Algorithm** $O(n)$
100 Entries	0.02560844 (s) with 272 bytes	0.02546270 (s) with 120 byes
500 Entries	0.02662478 (s) with 272 bytes	0.02690106 (s) with 152 bytes
1000 Entries	0.0276815 (s) with 272 bytes	0.02698754 (s) with 152 bytes
10,000 Entries	0.04059656 (s) with 272 bytes	0.04606926 (s) with 152 bytes

Based on the information provided, and the runtime of both, It is clear that Divide-and-Conquer is more efficient in terms of both memory usage and time taken to discover the maximum contiguous subarray. As seen in the table above, in every average instance, it was faster than Kadane's (albeit in real time this could be considered insignificant). When looking at the tables, Divide-and conquer had an nearly identical average runtime of roughly 0.02 seconds across each of the array entries. However, it should be noted that once the entries go up to 10,000 integers, the calculation runtime grows to almost double. When looking at Kadane's output, it uses less memory to calculate the entries at nearly the exact speed, but when the entries surpass 500, it uses more consistently afterwards. It should be noted that once both algorithms process 10,000 entries, the speed of the algorithm is nearly identical, but Kadane's used slightly less bytes in order to calculate the maximum contiguous sum.

Conclusion

In conclusion, Divide-and-Conquer, subverting my expectations, was nearly operating at the same speed at Kadane’s algorithm, albeit with slight variations of memory usage. However, the time elapsed to calculate maximum contiguous subarray is dependent on the speed of PCs, so when purely basing the algorithm efficacy on memory usage alone, Kadane’s algorithm outputs the same contiguous sum as Divide-and-Conquer with less memory usage. Therefore, in terms of the Maximum subarray problem, Kadane’s algorithm can be considered the best choice for outputting the sum of a massive array, even at thousands of entries.

Despite this, Kadane’s should not be considered the ‘one-and only’ algorithm that must be implemented in each maximum subarray, as a researcher wants to rule out any possibilities of a false answer derived from the algorithm. Therefore, it can be considered impervious that, when calculating sums in this context, multiple algorithms should be implemented to rule out any possibility of error.

Bibliography

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). The MIT Press.

freeCodeCamp. (2019, November 26). *Divide and Conquer Algorithm Meaning: Explained with Examples*. freeCodeCamp. https://www.freecodecamp.org/news/divide-and-conquer-algorithms/

Geeks for Geeks. (2021, April 15). *Largest Sum Contiguous Subarray*. Geeks for Geeks. https://www.geeksforgeeks.org/largest-sum-contiguous-subarray/

Keith-Magee, R. (2019, November 22). *Monitoring memory usage of a running Python program*. Medium. https://medium.com/survata-engineering-blog/monitoring-memory-usage-of-a-running-python-program-49f027e3d1ba

Maximum subarray problem. (n.d.). Wikipedia. https://en.wikipedia.org/wiki/Maximum_subarray_problem

Singhal, R. (2018, December 31). *Kadane’s Algorithm — (Dynamic Programming) — How and Why does it Work?* Medium. https://medium.com/@rsinghal757/kadanes-algorithm-dynamic-programming-how-and-why-does-it-work-3fd8849ed73d