The Maximum-subarray Problem

A comparison between Kadane's Algorithm and Divide & Conquer

Emil S. Valberg-Madsen, 20175075
ADSSE-MED8, Aalborg University 2021

Abstract

There are a multitude of different algorithms that can be used to solve the Maximum-subarray problem. In this report I aim to tackle this problem by implementing and analyzing two algorithms in Python. The two algorithms are Kadane's Algorithm and the Divide and Conquer Algorithm. Both algorithms were tested on the same set of data, and analyzed for their performance. When comparing the results of the experiments on the two algorithms, it was found that Kadane's algorithm performed significantly better than the Divide and Conquer algorithm when comparing their individual running times.

1 Introduction

For this project it was decided to work with the Maximum-subarray Problem first put forth by Ulf Grenander in 1977, as simplified model for estimating the maximum likelihood of patterns in digitized images. In the book "Introduction to Algorithms" by Cormen et al. (2001), an example to this problem is described as a way to compute the maximum profit in stock exchange. E.g. you are tasked with buying only one stock one time, and selling it later for the maximum profit. As Cormen et. al mentions in the book, you would think to just buy when lowest and sell when highest, but you might not be able to always sell at the highest price within a certain time period. It is therefore important to look at the change in value in a contiguous way.
The ***Maximum-subarray Problem*** can therefore be described as:
Input: an array A of numbers.
Output: a contiguous subarray of A whose sum is maximal.

Given an array **A** of **n** numbers, find a subarray **A[i..j]**, such that the **sum** of the elements from **A[i]** to **A[j]**, inclusive, is a **maximum**.

There are multiple ways to tackle this computational problem and different algorithms that can be used. In this project, I decided to use two different algorithms to solve the maximum-subarray problem and later analyze and compare them against each other to find which algorithm performs best in terms of running times and memory usage. The two algorithms chosen for this project is *Kadane's Algorithm* and *Divide and Conquer*.

2 Pseudo Code

As mentioned previously the maximum-subarray problem is the task of finding the largest possible sum of contiguous subarray, within a given two-dimensional array with **n** numbers. This section will be describing the two algorithms implemented and explain their implementation through pseudo code.

2.1 Kadane's Algorithm

The main purpose of this algorithm is to look through the given array A, and look for all the positive contigious sections of the array as well as keeping track of the maximum sum at all times. For this lets assume that a we have a given array A with n numbers of indices. We also assume we have two variables max_sum, which will be used to keep track of the maximum sum found so far, and max_sum_end, which is used to store the maximum sum found within a certain segment of the array. With this in mind, lets look at the pseudo code for this:

```
1      Initialize variables
2          max_sum = 0
3          max_sum_end = 0
4
5      Loop through each element of the array
6          max_sum_end = max_sum_end + A[i]
7          if max_sum_end > 0
8              max_sum = max_sum_end
9          if max_sum_end < 0
10             max_sum_end = 0
11
12         return max_sum
```

We first start by initializing our two variables and set them equals to 0. Then we start the looping function, that is used to go through each element of the array. On line 6 we set the maximum sum of the current contiguous segment equals to our variable plus the value of the current index. The two if statements is then used to check whether our maximum sum found within the current segment is greater or less than 0. If the value is greater than 0, that value will be added to the maximum sum found so far and the loop will move on to the next index in the array, and do the same again. This is until the value will go below 0, in which case the maximum sum with the current positive segment will be set to 0, and the algorithm will move on to the next index in the array, while starting on a new positive segment. Before the loop moves onto the next index it always returns the maximum sum found so far so it is stored within our max_sum variable until either the loop function is done or a new maximum sum is found.

2.2 Divide and Conquer

For the divide and conquer algorithm, it can be described as three main parts:
Divide: Which involves dividing the problem up into smaller instances of the same problem.
Conquer: Calling the sub problem recursively until the sub problem is solved. If they are small enough solve them straightforward.
Combine: Lastly combine the solutions from the sub problems together to find the solution of the original problem.

What divide and conquer actually does is to divide the given array up into two subarrays with as equal size as possible. We therefore find the mid-point of the array A, and check the two subarrays created from low-mid, and mid-high. The maximum contiguous subarray within the array must therefor lie in either of three cases, which is entirely in the subarray A[low..mid], entirely in the subarray A[mid+1..high], or crossing the mid-point (Cormen et al., 2001).

Looking into the pseduo-code of the divide and conquer technique, the function is a combination of two individual function. First we define a function called *MaxCrossSubarray*, which is used to find the maximum subarray that crosses the midpoint. It takes the *array A* as an input, along with the indicies *low, mid, high.*

```
MAX_CROSS_SUB(arr, left, mid, high)
    left_sum = 0
    max_cross_sum = 0

    for i in mid downto low
```

```
6           max_cross_sum = max_cross_sum + A[i]
7           if max_cross_sum > left_sum
8               left_sum = max_cross_sum
9
10      right_sum = 0
11      max_cross_sum = 0
12
13      for j in mid downto low
14          max_cross_sum = max_cross_sum + A[j]
15          if max_cross_sum > right_sum
16              right_sum = max_cross_sum
17
18      return(left_sum + right_sum, left_sum, right_sum)
```

The code work as follows. Line 2-8 finds the maximum subarray of the left half A[low..mid]. The for loop starts with index i at mid and cycles down to low, to ensure that whichever maximum subarray it finds has the range A[i..mid]. The variables initialized in line 1-2, are used to store the maximum sum found so far, and the sum of the entries A[i..mid]. The for loop will then go through each entry in the subarray much like Kadane's algorithm and whenever a value greater is found it saves it into the left_sum. Lines 10-16 does the same but for the right half of the array A[mid+1..high]. Finally in line 17 we return the sum of the elements on left and right.

```
1   MAX_SUB_ARRAY(arr, low, mid, high)
2       if low == high
3           return A[low]
4
5       mid = (low + high) / 2
6
7       return maximum (MAX_SUB_ARRAY(arr, low, mid),
8                       MAX_SUB_ARRAY(arr, mid+1, high)
9                       MAX_CROSS_SUB(arr, low, mid, high))
```

This next part of the algorithm, *MAX_SUB_ARRAY* is the final part of the divide an conquer, and is where we piece back together the array to find the maximum contiguous subarray of array A. It takes the *array A* as input as well as the low, mid, and high indices. The if statement in line 2-3 test for the base case where the subarray has only one element - itself, and therefore the if statement would return a tuple with the start and end indices of just the one element. Line 5 simply finds the mid-point. Lastly the function will then line 7-9 checks the three cases mentioned earlier and returns the maximum value found.

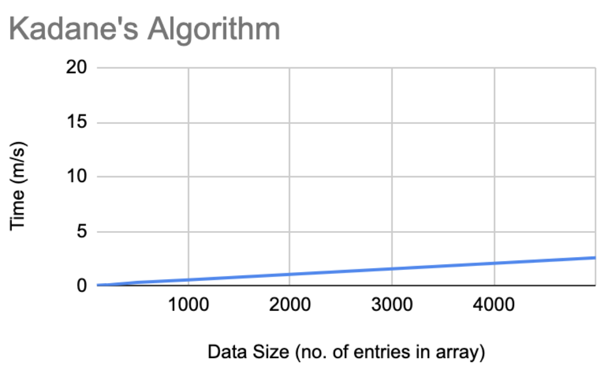
3 Performance & Comparison

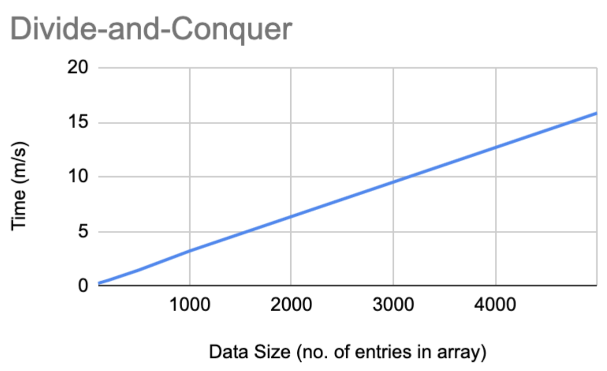
In order to determine which of the two previously describe algorithms performs best, they were both implemented in Python and used to test on the same data set. The following table shows the given results of the experiments for easy comparison. Each algorithm was run 5 times on the same array, to measure the running times, and an average was found. Five different arrays was tested of different sizes to measure performance with larger and larger data sets. The arrays was randomly generated to contain random integers with the range of -50 to 50.

	Kadane's Algorithm	Divide-and-Conquer
Data Size	Avg. Time (m/s)	Avg. Time (m/s)
100 Entries	0.052356718	0.24151802
200 Entries	0.099658966	0.510358808
500 Entries	0.32658577	1.4523983
1000 Entries	0.562191008	3.216123582
5000 Entries	2.59103775	15.83924294

Table 1: Experimental Results of the two algorithms

The table above has been set to contain the average running times of the five experiments run on each data entry. This means that for each set of data sizes, each algorithm has been run five times, and the average running time has been logged.
Looking at the table we can see that Kadane's algorithm performs faster than the divide and conquer. To better understand what happens with these algorithms performance time, and understanding the time complexity of the two, below you can see two graphs which depicts the data from the table.





As can be seen, Kadane's algorithm has a linear scaling of its running time, which suggests that Kadane's algortihm has the Big-O notation of O(n). The Big-O notation is used to describe how well an algorithm handles varible sizes of inputs. Having a Big-O of O(n) means, that the algorithm scales linearly as sizes in data inputs rises.
Looing at the graph for the divide and conquer algorithm, even though it might seem like it is linear in its time complexity, it has a much steeper scaling than the one seen in Kadane's algorithm. According to Cormen et al. (2001), the time complexity of a divide an conquer algorithm is O(n log(n)). Knowing this and looking at our graph for the divide and conquer, we can assume that this is correct due to steeper scaling. The reason it seems linear can be due to the lack of data tested on, or the size of the data. A logarithmic scale will always scale faster the higher we get, but in this case, the data sets points most likely only to the beginning of the (n log(n) logarithmic scale. So if we were to test on data size of 10.000 and even more entries within the array, the graph for the divide and conquer would most likely start showing that rise in scaling.
This suggests that Kadane's algorithm should perform better than the divide and conquer algorithm, as the latter will scale logarithmic in its running time if we only have the Big-O notation in mind. Comparing that with the results from the experiments, we can safely assume that this is correct, as Kadane's algorithm has a much lower running time.

4 Conclusion

This project worked with the problem of finding the maximum contiguous sum of a subarray within a give array. This problem was described, and it was chosen to implement and test two different solution to this problem. The two solutions or algorithms chosen was the Kadane's algorithm and the Divide and Conquer. The algorithms were implemented in Python, and tested for their performance in terms of running times. The same data set was used, were the amount of entries within the arrays would vary. The results of the experiments suggested that Kadane's performs significantly better than the divide and conquer approach in terms of its running times. This was due to Kadane's O(n) time complexity being a much better option than the divide and conquers O(n log(n)).

It was also the purpose of this project to test the two algorithms in terms of not only the running times, but also their memory usage. I was struggling figuring out how to do it, and it was ultimately scrapped, since I could not get it to work. While it would have helped solidify that Kadane's algorithm performs better, the results we do have definitely shows this, as well as proving that O(n) time complexity is better than O(n log(n). It can therefore be concluded that when solving the maximum subarray problem, Kadane's algorithm will be the better choice out of the two algorithms implemented in this project.

References

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2001). *Introduction to algorithms* (2nd ed.). The MIT Press.