Maximum Subarray

Alexander Jonstrup
Ajonst17@student.aau.dk

ABSTRACT

This project worked with the maximum subarray problem where the aim is to find a contiguous subarray which gives the maximum sum in a one dimensional array of integer values. Therefore, Kadane's algorithm and the Divide and Conquer algorithm were described, analyzed, implemented and tested. The pseudo code for both algorithms were described in order to explain how the algorithms work. Then each of the algorithms were analyzed in terms of running time and memory usage where the highest order term was identified. It was found that Kadane's algorithm has a faster running time compared to Divide and Conquer. According to the results of the test of the implemented algorithms Kadane's algorithm took less time to find the maximum subarray compared to Divide and Conquer. Therefore, both the asymptotic running time and the test results shows that Kadane's algorithm is the fastest solution to this problem. However, non of the algorithms were found to be optimal in terms of running time or memory usage.

1 INTRODUCTION

This project focuses on the "maximum subarray problem" where the aim is to compute the maximum sum among contiguous elements in an array. In this project two different algorithms will be used for approaching this problem. These algorithms are "Kadane" and "divide and conquer". The structure of this report is the following. First both algorithms both algorithms will be analyzed, the pseudo code will be described and an implementation of both algorithms will be tested. The results of the test will be used for concluding the performance of the algorithms.

2 PSEUDO CODE

In this section the pseudo code for both Kadane's algorithm and the Divide and Conquer will be explained. The input array for both algorithms are assumed to be one dimensional and contain integer values.

2.1 Kadane's Algorithm

```
1  MaxSubArray(A){
2      local_max = 0
3      global_max = MinValue
4      for(i = 0; i < A.Length; i++){
5          local_max = max(A[i], A[i]+local_max)
6          if(local_max > global_max){
7              global_max = local_max
8          }
9      }
10     return global_max
11 }
```

Line 1 to 11 shows the pseudo code for Kadane's algorithm. On line 1 is the selector of the function and the parameter A is an array. On line 2 and 3 the variables "local_max" and "global_max" are initialized. The variable local_max refers to the current maximum sum among contiguous elements when iterating through A while global_max refers to the maximum sum in a subarray of A. Therefore, global_max is initialized to the minimum value of the value type of A thus any sum greater than the minimum value will be detected otherwise the minimum value is the greatest sum within A.

On line 4 to 9 shows the code for the for loop used for iterating through A and detecting the local and global maximum sum among contiguous elements within A. Line 4 initializes the variable i which is used as an index to access the elements of A. Therefore, i starts at a value of 0 and goes to length of elements of A and is incremented by one for each iteration. On line 5 the varibale local_max is assigned the maximum value between the element at index i of A and sum between index i of A and the local_max value. If the sum of the value of local_max and the current element of A then is less than the current element of A then the current element of A is so far the maximum subarray of A among contiguous elements. Otherwise if the sum of local_max and the current element of A is greater than the current element of A then the sum among contiguous elements continues thus local_max is then assigned to the maximum sum. On line 6 the value of local_max is compared with global_max and if local_max is greater than global_max then a new maximum sum has been detected. Therefore, global_max is assigned to the value of local_max on line 7. This also the reason mentioned earlier that global_max was initialized to a minimum value such that the maximum subarray sum in A always is greater than or equal to the minimum value that global_max is assigned to. Lastly, on line 10 the value of global_max is returned as the maximum sum among contiguous elements.

2.2 Divide and Conquer Algorithm

```
1  FindMaxCrossingSubarray(A, low, mid, high){
2      left_sum = −∞
3      sum = 0
4      for(i = mid; i ≥ low; i−−){
5          sum += A[i]
6          if(sum > left_sum){
7              left_sum = sum
8              max_left = i
9          }
10     }
11     right_sum = −∞
12     sum = 0
13     for(j = mid+1; j ≤ high; i++){
14         sum += A[j]
15         if(sum > right_sum){
16             right_sum = sum
17             max_right = j
18         }
19     }
20      return (max_left,max_right,left_sum+right_sum)
21 }
```

```
1  FindMaxSubarray(A,low,high){
2      if(low == high){
3          return (low,high,A[low])
4      } else{
5          mid = ⌊(low+high)/2⌋
6          (left_low,left_high,left_sum) =
7          FindMaxSubarray(A,low,mid)
8          (right_low,right_high,right_sum) =
```

```
9             FindMaxSubarray(A,mid+1,high)
10            (cross_low,cross_high,cross_sum) =
11            FindMaxCrossingSubarray(A,low,mid,high)
12            if(left_sum≥right_sum && left_sum≥cross_sum){
13                  return (left_low,left_high,left_sum)
14            }else if(right_sum≥left_sum && right_sum≥cross_sum){
15                  return (right_low,right_high,right_sum)
16            }else{
17                  return (cross_low,cross_high,cross_sum)
18            }
19      }
20  }
```

The idea behind the divide and conquer approach is to divide the input array A into smaller sub arrays until the smallest subarray has been reached and then use the element of the reached subarray as the local maximum. Then the local sum is compared with other local sums and the largest sum among contiguous elements in A is returned as the maximum sum.

First the function called "FindMaxCrossingSubaray" will be explained and then the function "FindMaxSubarray" will be explained. On line 1 in FindMaxCrossingSubarray the parameters are an array A and the low, mid and high range values of A. The parameters low and high specify the index range of A where the function should look for the maximum subarray sum. The parameter mid is then a value in between low and high. When dividing the input array into sub arrays the size of the sub arrays do not have to be of the same size but in this case the size of each of the sub arrays are almost the same. One of the sub arrays ranges from the value of low to the value of mid and the other ranges from the value of mid plus one to the value of high.

On line 2 and 3 the variables left_sum and sum are initialized similarly to the variables local_max and global_max in Kadane's algorithm. Then the for loop on line 4 to line 10 iterates through the left part of the input array A ranging from the mid value to the low index value. Therefore, the variable i is initialized to the value of mid and is decremented by one after each iteration. Line 5 adds the value of the element at position i in A to sum and then on line 6 the variable sum is compared with the variable left_sum. If the value of sum is greater than the value of left_sum then a new maximum sum has been detected in the left part of A. The variable max_left stores the index of the first element where the maximum sum has been detected. This means that max_left is the start index of the subarray with the maximum sum.

The sum for the right part of A is done in a similar way as the left part. The variable right_sum is initialized to negative infinity and sum is assigned to 0 in line 11 and 12. This time the for loop iterates through the right part of A therefore the index variable j is assigned to the value of mid plus on and is incremented by one for each iteration until it reaches the length of A, see line 13. The reason for j to be equal to the value of mid plus one has to do with the way that A was divided into a left and right part subarray. The variable mid is assigned to half the size of A rounded down to the closest integer. Therefore, by adding one to the mid value it does not include the element at that position which belongs to the left subarray. The lines 14 to 19 follows the same approach as line 5 to 10 in the left subarray for loop. Lastly, at line 20 the following three values are returned: the start and end index of the maximum subarray (max_left, max_right) and the sum between the left and right part of the maximum subarray). Therefore, the this function computes the maximum sum that is in between the left and right part of the given input array.

Line 1 shows the selector "FindMaxSubarray" which has the following 3 parameters: the input array A and the start and end index of A called low and high. Line 2 checks if the value of the start index low and end index high are equal. If this is the case then the start index low, the end index high and element at position low in A are returned. Then the maximum sum is that particular element of A. If the start index low and end index high are different then some of the statements inside the else block will be executed, see line 5 to 18. On line 5 initializes the variable mid to the sum between low and high divided by 2 and then rounded down to the nearest integer as described earlier. Then on line 6 the variables left_low, left_high and left_sum are initialized to the returned values of the call of the function itself given only the left subarray as an argument, see line 7. The same goes for line 8 and 9 with the right subarray. This is why the same input array A is given as input array and low is given as the start index but the end index is the value of mid instead of high. Therefore, calling the function itself with the same start index but with varying end index the function will eventually encounter that the condition in the if statement at line 2 will be equal to true and the start and end index will be returned as well as the element of A at index low. This is the dividion and recursive parts of the divide and conquer algorithm where the input array is divided into sub arrays recursively until the subarray only consists of one element which is then the local maximum sum. Then on line 10 and 11 the variables cross_low, cross_high and cross_sum are initialized to the start and end index and maximum sum between the left and right subarray as described previously with the function "FindMaxCrossSubarray".

The last part of the function from line 12 to 18 checks whether the maximum among contiguous elements are in the left or right side of A or in between. If the sum in the left side is the greatest then left start and index are returned also with the left side sum. However, if the sum is in the right side of A then the start and end index of the right side is returned also with right side sum. Otherwise, the start and end index in between the left and right side of A is returned also with the sum.

3 Code Analysis

This section includes an analysis of the running time for both algorithms. The running time analysis is done such that both subsections refers to the pseudo code described in the previous section and looks for the highest order term which has the greatest impact on the running time. This is done by first presenting the lower order terms and then presenting the highest order term in the end. Based on the highest order term the asymptotic running time notation is then presented and the same goes for the space complexity.

3.1 Kadane's Algorithm

In Kadane's algorithm the variables local_max and global_max are both initialized once which takes constant time for each of them. The reason the initialization of the variables to run in constant time is because they do not depend on size of the input array as they are only initialized once. The for loop runs in linear time since the condition of the for loop must be checked once per iteration which goes from zero to the size of A and the incrimination of i is also done once per iteration. Then each statement inside the for loop takes constant time per iteration. This goes for the assignment of local_max and global_max and the if statement comparing the variables. However, this is done once per iteration therefore each statement or expression from line 5 to 7 in Kadane's algorithm also takes linear time. The reason for this to be linear is because these statements and expression inside the for loop depends on the size of the input array A. Lastly, the return statement also runs in constant time and is only do once. Therefore, the highest order term in Kadane's algorithm has to do with for loop which runs in linear time.

In terms of asymptotic running time notation the time complexity of Kadane's algorithm is denoted O(n). The big O of n denotes worst case scenario of the running time of the algorithm. This means that there is an upper bound of the running time as the size of the input array increases so does the running time. Thus, the running time of Kadane's algorithm will never decrease when the size of the input array increases. If the input array of Kadane's algorithm only

contains elements with a value less than or equal to zero then the maximum sum is 0 but the algorithm must still iterate through the entire input array in order to determine that 0 is the maximum sum in a subarray in the input array. Therefore, Kadane's algorithm is linear no matter the size of the input array and contains at least one element that is greater than or equal to zero. This is also why the running time of Kadane's algorithm is not optimal.

The amount of space needed for Kadane's algorithm depends on the size of input array A. If the size of A is 0 then the algorithm use constant memory for the variables local_max and global_max since the for loop condition evaluates to false and the statements inside will never be executed. However, for any size of the array that is greater than 0 the memory usage is linear. The memory usage is linear because it scales with the size of the input array A and can therefore be denoted O(N). The big O of N states that there is an upper bound to memory usage since the algorithm will never use any less memory as the size of the input array increases. Therefore, the memory usage for Kadane's algorithm is not optimal as the big O of N refers to the worst case scenario since there only exists an upper bound to the function of memory usage.

3.2 Divide and Conquer

The divided and conquer algorithm runs in constant time if the parameters low and high are equal or the size of the input array is equal to 1 since the element at position low is returned as the maximum subarray sum and low and high are returned as the start and end index. In this case the running time does not depend on the size of the input array A. However, for any size of the input array A that is greater than one and the parameters low and high are not equal the running time depends on the size of A. Since the divide and conquer algorithm runs recursively each initialization and assignment runs in constant time for each recursive call but the amount of times that the function calls itself depends on the size of the input array A. Another factor which affects the running time is the fact that the function calls itself twice when the function has been invoked. However, the each recursive call only takes half the size of the input array since it has been subdivided into a left and right subarray. Additionally, the function "FindMaxCrossingSubarray" is also called which goes through the entire input array and therefore this runs in linear time. Then the asymptotic notation can be stated as T(n) = 2*T(n/2) + $\theta(n)$. The function T(n) describes the total running time of the divided and conquer algorithm where the first part on the right of the equal sign describes the running time for the recursive part of the algorithm. As mentioned earlier, the function calls itself twice for each call but only goes through half of the input array. Therefore, the first part ti two times the running time for half the size of the input array. The second part of the running time equation is denoted big Theta meaning that the running time has an upper and lower bound for an input array with one or more elements.

In terms of the space complexity of the divide and conquer algorithm the memory usage is constant if the parameters low and high are equal and or if the input array only contains one element. Similar to the running time, the memory usage does not depend on the size of the input array in this case. However, if these conditions are not the case such that the size of A is greater than 1 and low and high are not equal the memory usage then depends on the size of A. Since the memory usage depends on the size of A and the two recursive calls goes through half the size of A it becomes linear in total. The same goes for the call of the function FindMaxCrossingSubarray which goes through the entier input array. Then the highest order term in relation to memory usage is linear which scales with size of A. Therefore, the growth of memory usage can be denoted as O(n). This is big O of n which describes the worst case scenario of memory usage since it only has an upper bound. This also means that the memory usage is not optimal since the upper bound tells that this is the worst case scenario.

4 Results

In this section the results of running the implemented algorithms will be presented. Both algorithms were implemented using the programming language C#. When testing the implemented algorithms they were both given the same input array which consisted of 32 bit integer values ranging from 0 to 2,147,483,647. The integer values in the input array was randomly generated. The tests has been run on multiple sizes of the input array which gave different results in terms of execution time. These execution times for different sizes of the input array has been inserted into the following tables: 1 and 2. The running time column shows the running time in milliseconds and the size of input array column is the size of the input array.

Running Time in milliseconds	Size of input array
0	10
0	100
0	1000
1	10000
11	100000
126	1000000
1335	10000000
13428	100000000

Table 1: Running Time Table: Kadane's Algorithm

Running Time in milliseconds	Size of input array
0	10
0	100
0	1000
1	10000
12	100000
134	1000000
1395	10000000
14036	100000000

Table 2: Running Time Table: Divide and Conquer Algorithm

When comparing the running time for Kadane's algorithm and Divide and Conquer it can be seen that Kadane's algorithm runs faster than the divide and conquer algorithm. This also correlates with the asymptotic notation stated in the previous section. Another reason why Kadane's algorithm runs faster than Divide and Conquer could also be that Kadane's algorithm only iterates through the input array. The divide and Conquer algorithm is called recursively and also calls the function FindMaxCrossingSubarray which runs in linear time.

5 Conclusion

In this mini project Kadane's algorithm and the Divide and Conquer algorithm have been described, analyzed, implemented and tested. The asymptotic running time for Kadane's algorithm was stated to be O(n) and for Divide and Conquer the running time was stated to be 2T(n/2)+θ(n). According to the asymptotic running time notation Kadane's algorithm has a faster running time than divide and Conquer. This was also found to be the case when testing the implementation of the two algorithms on multiple sizes of an input array which elements were random integer values ranging from 0 to 2,147,483,647. Therefore, it is concluded that Kadane's algorithm had the fastest running time in this project.

References

[1] C. Stein, C. E. Leiserson, T. H. Cormen, and R. L. Rivest. *Introduction to algorithms*. The MIT Press. The MIT Press, 2009.