Lab 2

Leo Man
Jesper Persson
Gustav Hammarström
Department of Engineering Sciences and Mathematics

January 9, 2025

1 Introduction and project layout

In this report will explain how our designed divide and conquer algorithm works, and that it works in linear time. Will will also show that the algorithm that was described in the assignment works.

In the project you will find the following. This report. The algorithm for part 1 named MaxSubArray.py and a script that runs some test on it named graph.py.

2 Part 1

2.1 How the Algorithm Works

Base Case

1. If the array size is 1 (only contains one element), return that element as:
    - The maximum sum value.
    - The maximum prefix sum value.
    - The maximum suffix sum value.
    - The total array sum value.

Divide

1. Calculate the middle of the array.
2. Recursively split the array into left and right subarrays. Where both left and right subarrays, has these as values:
    - The maximum sum value.
    - The maximum prefix sum value.
    - The maximum suffix sum value.
    - The total array sum value.

Combine

1. Calculate the maximum sum value that spans across the middle of the array. This is done by adding:
    - The suffix value from the left subarray.
    - The prefix value from the right subarray.
2. Determine the maximum sum value for the current array. This is the largest of:
    - The maximum sum value in the left subarray.
    - The maximum sum value in the right subarray.
    - The maximum sum value across the middle (calculated in the previous step).
3. Calculate the prefix sum for the current array. This is the greater of:
    - The prefix sum of the left subarray.
    - The total sum of the left subarray plus the prefix sum of the right subarray.
4. Calculate the suffix sum for the current array. This is the greater of:
    - The suffix sum of the right subarray.
    - The total sum of the right subarray plus the suffix sum of the left subarray.
5. Calculate the total sum for the current array by adding:
    - The total sum of the left subarray.
    - The total sum of the right subarray.
6. Return the following:

- The maximum sum value.
- The maximum prefix sum value.
- The maximum suffix sum value.
- The total array sum value.

2.2 Run time of the algorithm

This is our algorithm in pseudo code

Algorithm 1: Find Maximum Subarray

Input: Array arr, indices $head$ and $tail$
Output: Dictionary with **max**, **prefix**, **suffix**, and **total** for $arr[head : tail]$

```
1  function Solve(arr, head, tail):
2      if head = tail then
3          return { max: arr[head], prefix: arr[head],
4              suffix: arr[head], total: arr[head] };
5      middle ← ⌊(head + tail)/2⌋;
6      left ← Solve(arr, head, middle);
7      right ← Solve(arr, middle + 1, tail);
8      across ← left[suffix] + right[prefix];
9      maxValue ← max(left[max], right[max], across);
10     prefix ← max(left[prefix], left[total] + right[prefix]);
11     suffix ← max(right[suffix], right[total] + left[suffix]);
12     total ← left[total] + right[total];
13     return { max: maxValue, prefix: prefix,
14         suffix: suffix, total: total };
```

To find the runtime of this algorithm we firstly have to find the recurrence relation. Meaning an expression that looks like this

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n) \tag{1}$$

The algorithm recursively splits the problem into two subproblems of the size $\frac{n}{2}$ (At liine 5 to 7) meaning that the first term in equation is 1

$$2T\left(\frac{n}{b}\right) \tag{2}$$

and for the additional work after the subproblems are solved is constant since it is simple addition and comparisons therefore

$$f(n) = O(1) \tag{3}$$

and

$$T(n) = 2T\left(\frac{n}{2}\right) + O(1) \tag{4}$$

is the recurrence equation.

Now to find the runtime we use the *Master theorem* and from equation 1 and 4 we get that

$$\begin{cases} a = 2 \\ b = 2 \end{cases} \tag{5}$$

and with the first case of the master theorem, "if there exists a constant $\epsilon > 0$ such that $f(n) = O(n^{\log_b a - \epsilon})$, then $T(n) = \Theta(n^{\log_b a})$." And plugging in the values $a = b = 2$ gives

$$O(n^{\log_2 2 - \epsilon}) = O(1) = f(n) \quad if \quad \epsilon = 1 \tag{6}$$

therefor the runtime of our designed algorithm is

$$T(n) = \Theta(n) \tag{7}$$

according to the master theorem.

3 Induction proof of algorithm

3.1 The Base Case

If the array to be sorted is less or equal to 4, it is trivial that the array is correctly sorted.

3.2 The Inductive Hypothesis

Assuming that the algorithm correctly sorts an array of size m or less.

$$< a_1, a_2, ...a_m > \tag{8}$$

3.3 The Induction step

Proving that it can correctly sort an array of size $m + 1$.

At step 1 of the algorithm we recursively sort

$$< a_1, a_2, ...a_{\frac{3(m+1)}{4}} >, \tag{9}$$

which contains less elements then the array,

$$< a_1, a_2, ...a_m >,$$

and according to the inductive hypothesis this array can be sorted by the algorithm. Since

$$size(< a_1, a_2, ...a_{\frac{3(m+1)}{4}} >) \leq size(< a_1, a_2, ...a_m >) \tag{10}$$

The same logic holds true for both step 2 and step 3.

Now that step 1 to 3 is shown to return the inputed arrays sorted all that is left is to show that step 4 returns the entire array sorted.

After step 1 we have an array b which is the first 3/4 of the original array sorted.

At step 2 we sort the first 2/3 of array b together with the last 1/4 of the input array. Since b is sorted this means that the part that was left our from this sorting step (the first 1/3 of b), true right place cannot be the in the last 1/3 of c, since that would mean that array b was not correctly sorted.

Therefore what is left to do is to correctly sort in the first 1/3 of b, into the the first 1/3 of b and the the first 2/3 of c. Which is exactly what happens in step 3. When mereged at step 4 the array is then sorted.

Proven that the base case holds true and assuming that the algorithm correctly sorts an array of size m it follows that it correctly sorts an array of size $m + 1$. An according to the axiom of induction the algorithm correctly sorts an array of size $1 \leq n$

4 Recurrence Equation

The algorithm described in the assignment does 3 recursion calls, on a array of the size $\frac{3n}{4}$. The operations independent of the recursion are all constant.

Therefore the recurrence equation is

$$T(n) = 3T\left(\frac{3n}{4}\right) + O(1) \tag{11}$$