D0012E Lab 2

Fredrik Larsson
email frelau-1@student.ltu.se

Mikkjal Erik Gardshorn
email mikgar-1@student.ltu.se

Josef Thurfjell
email jostuh-1@student.ltu.se

December 14, 2022

1 Introduction

The purpose of the lab assignment is to create three algorithms. Two algorithms that returns the three smallest elements in an array, using an incremental and a divide-and-conquer approach, as well as one algorithm that calculates the maximum sum that a consecutive subarray of a larger array can have.

2 Code

2.1 Three smallest elements Incremental

The incremental solution to the algorithm was created like this:

1. Create a list with 3 indexes to hold the smallest items.
2. Loop through the list and do one of the following:
   (a) If the data set's current index is smaller than the smallest element of the list, shift the first two forward and add the index in as the smallest element
   (b) Otherwise, if the list's current index is smaller than the middle element, shift the second element forward and add the data set index as the second element
   (c) Finally, if the previous two conditions are not fulfilled replace the final third element in the list with the index of the data set.
3. Return the computed three smallest elements.

2.2 Three smallest elements Divide and Conquer

The Divide and Conquer solution to the algorithm was created like this:

1. Check for the 3 cases when the length of the list is either 1, 2, 3. If one of the base cases happens the list is sorted in ascending order.
2. Split the list into two parts, left and right.
3. Create a new variable that is the sum of two recursions on the left and the right sides.
4. While the length of the list is greater than 3 go through the list and check if the numbers are larger or smaller than the previous numbers
5. Remove the largest number

2.3 Maximum sum in a contiguous subarray

The Maximum sum in a contiguous sub array algorithm was created like this:

1. Create a function that takes a *list*, *left*, *right* as inputs
2. Checks for the base case, when subarray has just one element. And returns a list with the four
3. **Then preform the divide part of the algorithm.**
4. Get the middle of the algorithm by dividing left side plus right side with two
5. Then get the left values [left left sum, left right sum, left max sum, left sum] by preforming recursion with the left as left value and middle as the right value
6. Then get the right values [right left sum, right right sum, right max sum, right sum] sum by preforming recursion with the middle + 1 as left value and right as the right value.
7. **Then preform the conquer part of the algorithm**
8. **Get max sum on left side**
9. First check left side: If left sum is better than merging entire left side with right, then left max is equal to left sum.
10. Else if merging entire left side with right left sum is best, left max = left sums[3] + right sums[0]
11. **Get max sum on right side**
12. Then check right side: If right sum is better than crossing entire right side with left right max
13. Else merging entire right side with left right max is best, right max = right sums[3] + left sums[1]
14. **Get the highest sum so far**
15. If left sum was better than right sum. Create a max temp variable and store left sum in it.

16. if right sum was better than left sum. Change max temp to right sum
17. Then check if the greatest sum was a subarray that crossed both the left and right side. Do this by checking if left sums + right sums are greater than the max temp
18. Then sum total = left sums max sum + right sums max sum
19. And lastly return [left max, right max, max sum, sum total]

3 Results

3.1 Three smallest elements

To test if the algorithms worked an array where the three smallest numbers were predetermined([1,2,3]) was used. In each case when the algorithm was tested it always returned the three smallest elements regardless of order, proving that the algorithm worked.

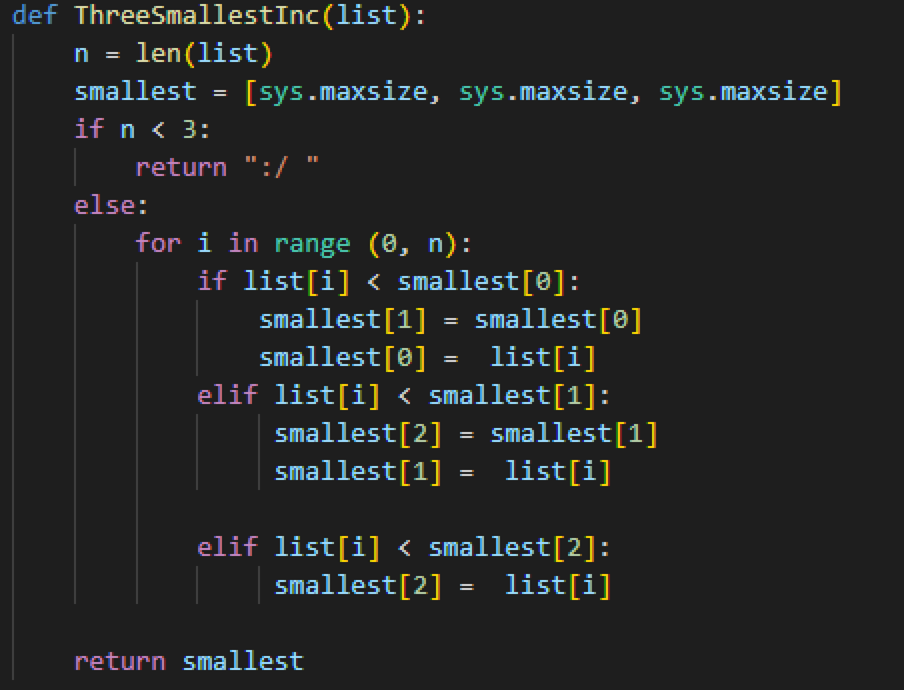
The incremental algorithm was deemed to be linear as it incrementally iterated through the array and checked if the element was smaller than the three saved values for the lowest values. As there was no repetition for already read elements the algorithm was deemed to be running in linear time.

The Divide and Conquer algorithm functioned through linear time by calculating through the arrays a set number of times based on the length of the array. The divided datasets were reduced and then merged for the merged lists to be reduced further until only three elements remained. Elements were not removed several times and the algorithm was deemed to be running in linear time.

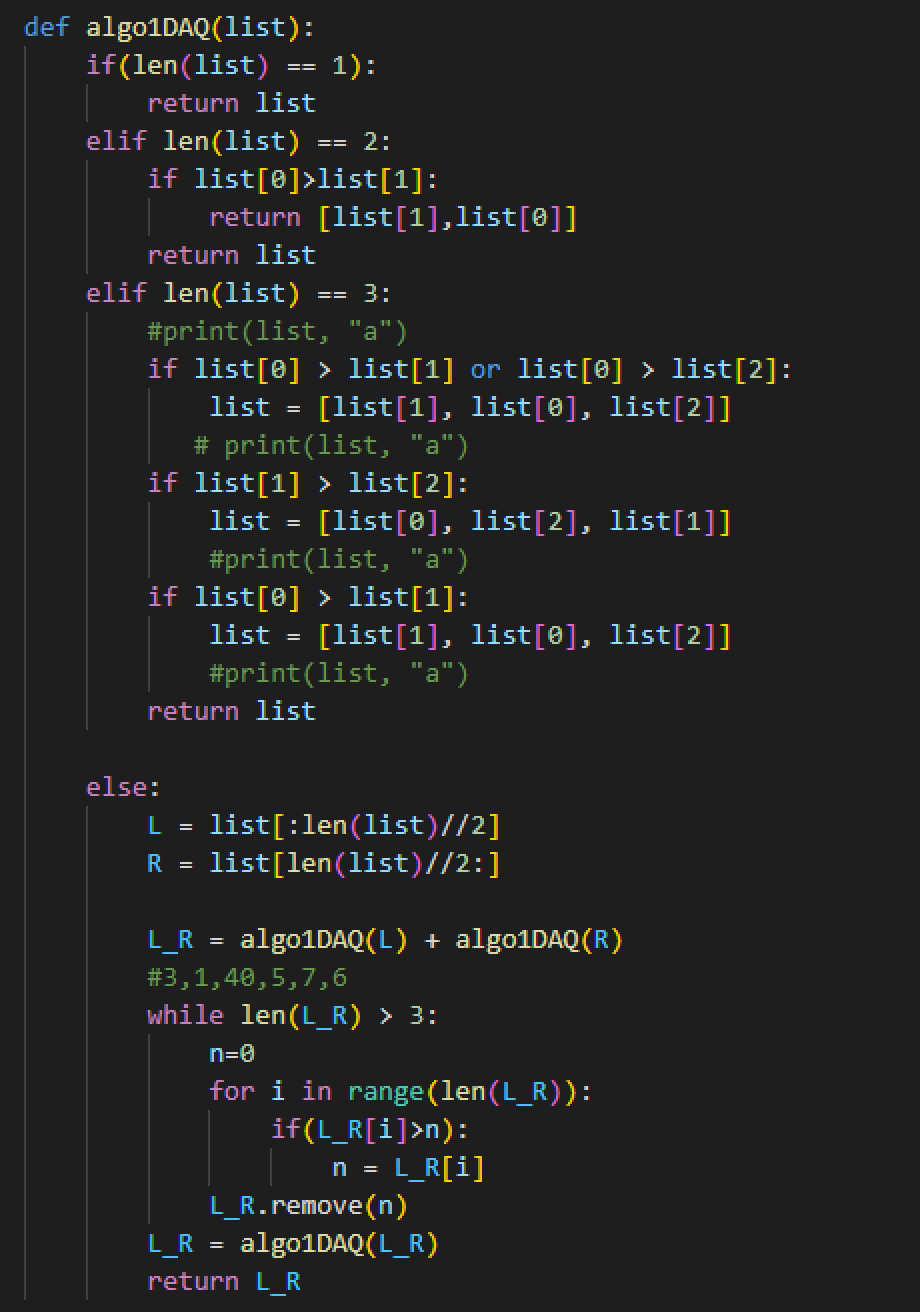
3.2 Maximum sum in subarray

To test if the algorithms worked, arrays where the maximum subarray sum was predetermined was used. With each array the algorithm returned the predetermined maximum sum proving that the algorithm worked.

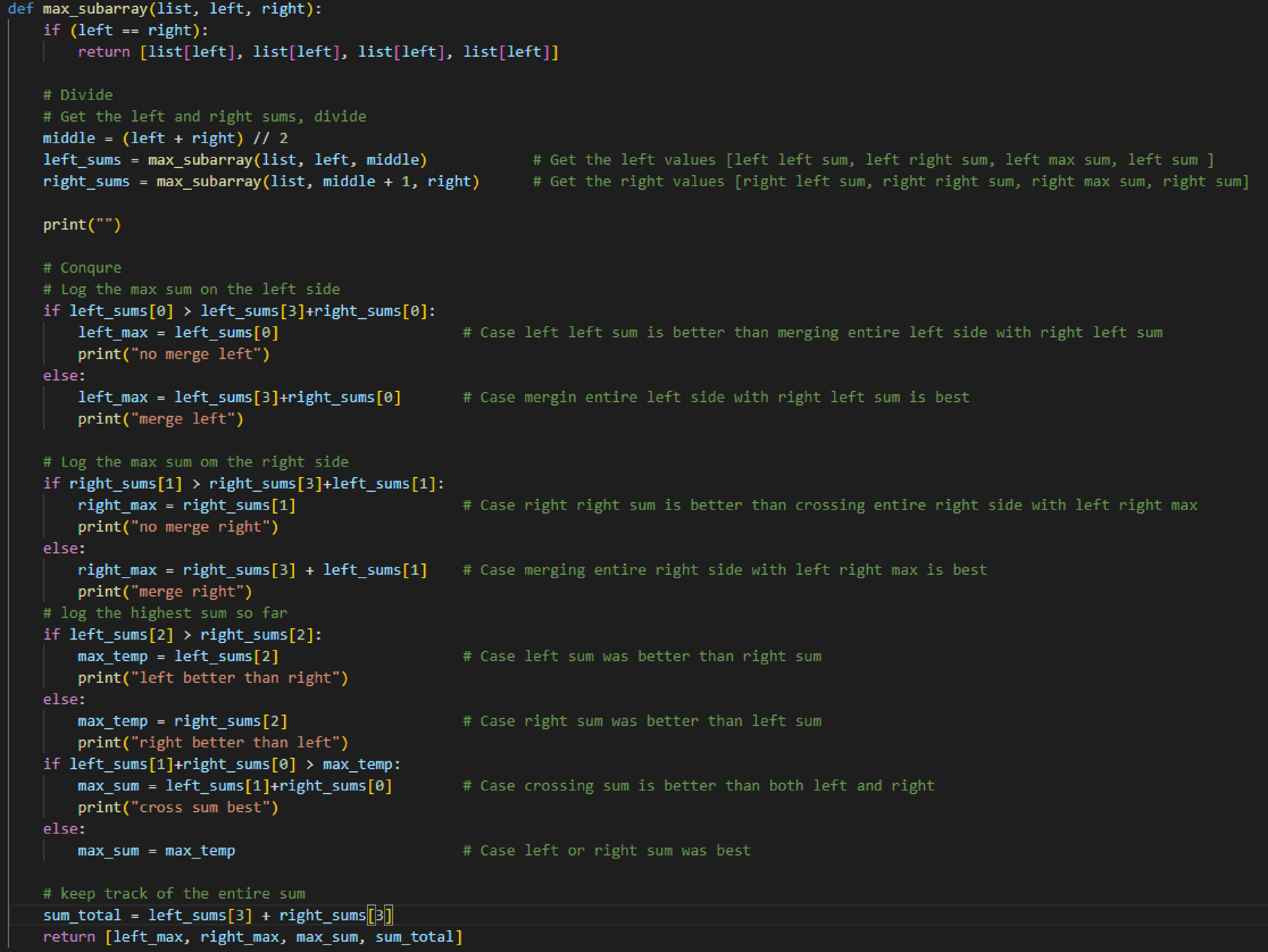
4 Discussion



```python
def ThreeSmallestInc(list):
    n = len(list)
    smallest = [sys.maxsize, sys.maxsize, sys.maxsize]
    if n < 3:
        return ":/ "
    else:
        for i in range (0, n):
            if list[i] < smallest[0]:
                smallest[1] = smallest[0]
                smallest[0] =  list[i]
            elif list[i] < smallest[1]:
                 smallest[2] = smallest[1]
                 smallest[1] =  list[i]

            elif list[i] < smallest[2]:
                 smallest[2] =  list[i]

    return smallest
```



```python
def algo1DAQ(list):
    if(len(list) == 1):
        return list
    elif len(list) == 2:
        if list[0]>list[1]:
            return [list[1],list[0]]
        return list
    elif len(list) == 3:
        #print(list, "a")
        if list[0] > list[1] or list[0] > list[2]:
            list = [list[1], list[0], list[2]]
           # print(list, "a")
        if list[1] > list[2]:
            list = [list[0], list[2], list[1]]
            #print(list, "a")
        if list[0] > list[1]:
            list = [list[1], list[0], list[2]]
            #print(list, "a")
        return list

    else:
        L = list[:len(list)//2]
        R = list[len(list)//2:]

        L_R = algo1DAQ(L) + algo1DAQ(R)
        #3,1,40,5,7,6
        while len(L_R) > 3:
            n=0
            for i in range(len(L_R)):
                if(L_R[i]>n):
                    n = L_R[i]
            L_R.remove(n)
        L_R = algo1DAQ(L_R)
        return L_R
```



```
def max_subarray(list, left, right):
    if (left == right):
        return [list[left], list[left], list[left], list[left]]

    # Divide
    # Get the left and right sums, divide
    middle = (left + right) // 2
    left_sums = max_subarray(list, left, middle)            # Get the left values [left left sum, left right sum, left max sum, left sum ]
    right_sums = max_subarray(list, middle + 1, right)      # Get the right values [right left sum, right right sum, right max sum, right sum]

    print("")

    # Conqure
    # Log the max sum on the left side
    if left_sums[0] > left_sums[3]+right_sums[0]:
        left_max = left_sums[0]                     # Case left left sum is better than merging entire left side with right left sum
        print("no merge left")
    else:
        left_max = left_sums[3]+right_sums[0]       # Case mergin entire left side with right left sum is best
        print("merge left")

    # Log the max sum om the right side
    if right_sums[1] > right_sums[3]+left_sums[1]:
        right_max = right_sums[1]                   # Case right right sum is better than crossing entire right side with left right max
        print("no merge right")
    else:
        right_max = right_sums[3] + left_sums[1]    # Case merging entire right side with left right max is best
        print("merge right")
    # log the highest sum so far
    if left_sums[2] > right_sums[2]:
        max_temp = left_sums[2]                     # Case left sum was better than right sum
        print("left better than right")
    else:
        max_temp = right_sums[2]                    # Case right sum was better than left sum
        print("right better than left")
    if left_sums[1]+right_sums[0] > max_temp:
        max_sum = left_sums[1]+right_sums[0]        # Case crossing sum is better than both left and right
        print("cross sum best")
    else:
        max_sum = max_temp                          # Case left or right sum was best

    # keep track of the entire sum
    sum_total = left_sums[3] + right_sums[3]
    return [left_max, right_max, max_sum, sum_total]
```