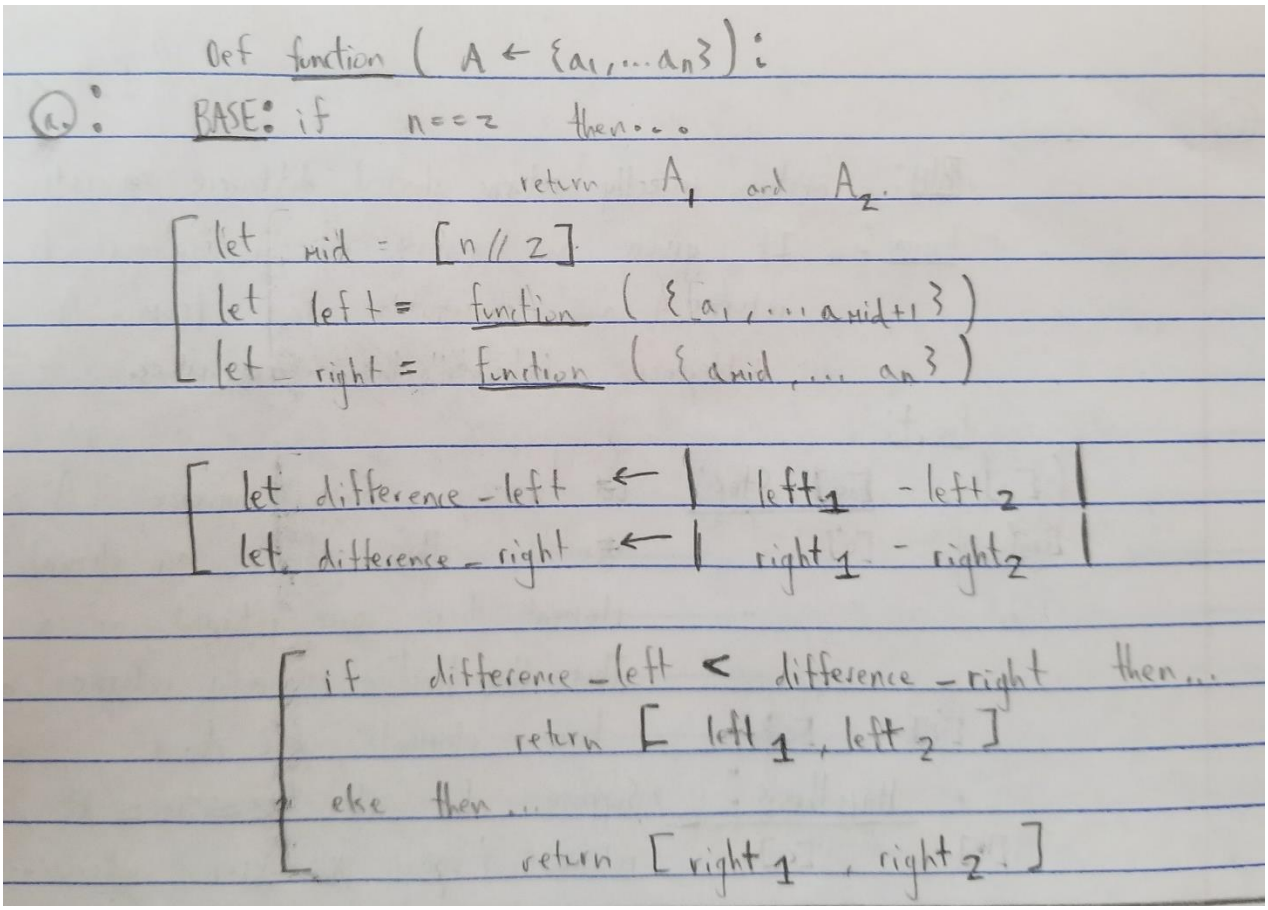
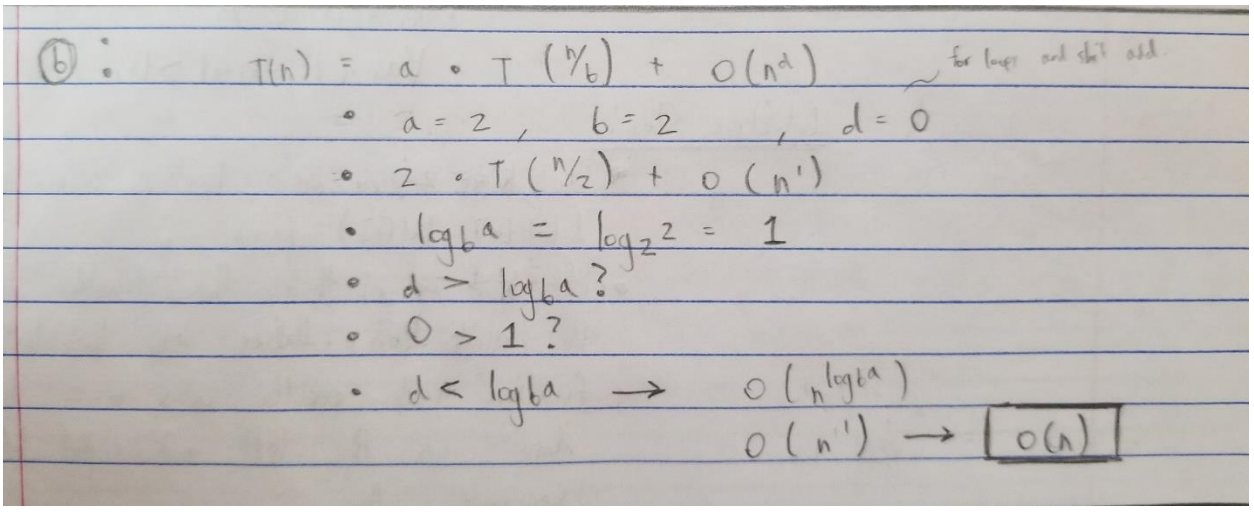
(1A)


Def function (A ← {a1, ... an}):
(a):
BASE: if n==2 then...
return A1 and A2.
let mid = [n // 2]
let left = function ({a1, ... amid+1})
let right = function ({amid, ... an})
let difference_left ← | left1 - left2 |
let difference_right ← | right1 - right2 |
if difference_left < difference_right then...
return [left1 , left2]
else then...
return [right1 , right2]

(1B)


(b):
T(n) = a • T(n/b) + O(n^d)
for loop and the add
• a = 2, b = 2, d = 0
• 2 • T(n/2) + O(n^1)
• log_b a = log_2 2 = 1
• d > log_b a ?
• 0 > 1 ?
• d < log_b a → O(n^(log_b a))
O(n^1) → O(n)

(1C)

Thm: function correctly returns shortest difference consecutive pair from A.

lemma: If given a sequence of n integers A, function returns a consecutive pair from A with minimal difference between the two values.

Proof:

Basis Step: Let $n=2$. Since sequence A contains two elements, this means a_1 element one and a_2 element two are returned as a pair from A. Thus the function correctly returns a pair of two elements $x \in A$.

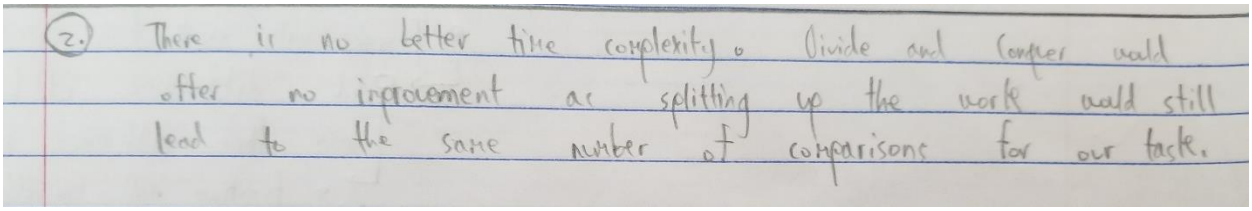
Hypothesis: Suppose for all $2 \leq n \leq k$, function will return pair x_i, x_{i+1} where...

- $x_i, x_{i+1} \in A$
- $\forall a_i \; (|x_i - x_{i+1}| \leq |a_i - a_{i+1}|)$

Inductive Step:

- if $k>2$, then the function will recursively be called on halves of the sequence until each function call results in a $n=2$ list. This is done on the left and right halves of Sequence A
- Each function call producing two possibilities:
  - if difference left (a_i, a_{i+1}) is less than difference right $(a_i + a_{i+1})$ then the function will correctly return left, WLOG, would return right.
- Since each call returns the correct smaller pair, we can conclude that the output is correct for size $N/2$ and $N/4$ and so on... This deduction will continue until we are left with one $n=2$ list. At our base case for which it is correct from the base case proof step.
- By the SMI for all $n \geq 2$, the function correctly returns pair (x_i, x_{i+1}) where distance of pair is less than all other pairs in sequence. ☐

(2)


2. There is no better time complexity. Divide and Conquer would offer no improvement as splitting up the work would still lead to the same number of comparisons for our task.