Algorithm Engineering

Md. Atiqur Rahman

Divide the Students and Conquer the Classroom

All the things taken from CLRS book

Buy Low, Sell High, Pray Harder: The Volatile Chemical Saga

A client has the chance to invest in the Volatile Chemical Corporation. Just like its products, the stock price changes a lot. He can buy one unit of the stock only once, and sell it later — both actions must happen after the market closes for the day. To help him decide, he is given the stock prices for n days in advance.

Now, the client wants you to write a program that tells him which day to buy and which day to sell the stock to make the most profit from those n days of prices.

Stocks for 17-day period (Here, n = 17)

Day	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Price	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97

Strategy 1:

1) Buy at the lowest and sell at the highest price.
2) Doesn't work since lowest is at day 7 where highest is at day 1.

Strategy 2:

1) Either buying at the lowest price or selling at the highest price.
2) Find the highest and lowest prices, and then work left from the highest price to find the lowest prior price, work right from the lowest price to find the highest later price, and take the pair with the greater difference.
3) Doesn't work for test case [10, 11, 7, 10, 6]

Tilt Your Head and Squint: It's a Whole New Problem Statement

We want to find a sequence of days over which the net change from the first day to the last is maximum. Instead of looking at the daily prices, let us instead consider the daily change in price, where the change on day i is the difference between the prices after day i 1 and after day i.

Day	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Price	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97
Change		13	−3	−25	20	−3	−16	−23	18	20	−7	12	−5	−22	15	−4	7

The table above shows these daily changes in the bottom row. If we treat this row as an array A, shown below, we now want to find the nonempty, contiguous subarray of A whose values have the largest sum. We call this contiguous subarray the **maximum subarray**. For example, in the array shown below, the maximum subarray of A[1,16] is A[8,11], with the sum 43. Thus, you would want to buy the stock just before day 8 (that is, after day 7) and sell it after day 11, earning a profit of $43 per share.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	13	−3	−25	20	−3	−16	−23	18	20	−7	12	−5	−22	15	−4	7

maximum subarray

Most of us, developers, will be satisfied with the below solution

Brute Force Approach

1) Take two nested for loops associated with i and j.
2) Maintain the condition i <= j and 1 <= i <= 16 and 1<= j <= 16
3) Now take another for loop inside the above two nested for loops to find the sum of subarray A[i, j].
4) Time Complexity is $O(n^3)$

In response to the client's concerns regarding the program's execution time, the maximum optimization we will implement is as follows:

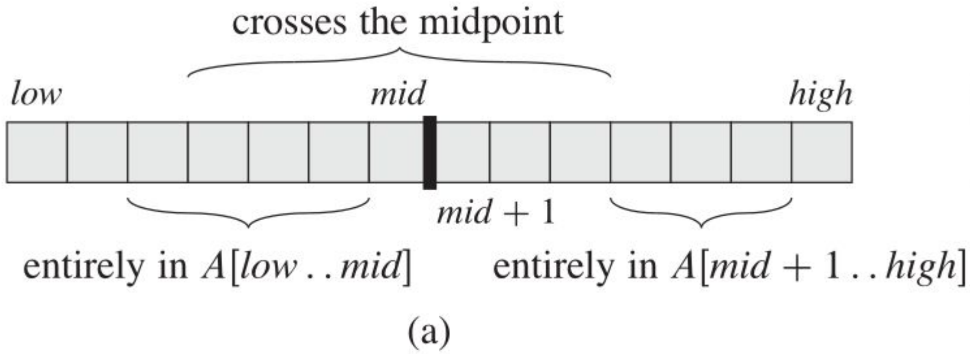
1) Make another array that has cumulative sum of the input array.
2) And do the 3rd step of the above algorithm in O(1).
3) No need for three nested loops, so time complexity is $O(n^2)$.

Ironically, the one person using DCC will end up working at FAANG

- **D**ivide the problem into a number of subproblems that are smaller instances of the same problem.
- **C**onquer the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.
- **C**ombine the solutions to the subproblems into the solution for the original problem. Sometimes, in addition to subproblems that are smaller instances of the same problem, we have to solve subproblems that are not quite the same as the original problem. We consider solving such subproblems as part of the combine step.

A solution using Divide, Conquer and Combine

Suppose we want to find a maximum subarray of the subarray A[low,high]. Divide-and-conquer suggests that we divide the subarray into two subarrays of as equal size as possible. That is, we find the midpoint, say mid, of the subarray, and consider the subarrays A[low,mid] and A[mid+1,high]. As below shows, any contiguous subarray A[i, j] of A[low,high] must lie in exactly one of the three places.



FIND-MAX-CROSSING-SUBARRAY(*A*, *low*, *mid*, *high*)

```
1   left-sum = −∞
2   sum = 0
3   for i = mid downto low
4       sum = sum + A[i]
5       if sum > left-sum
6           left-sum = sum
7           max-left = i
8   right-sum = −∞
9   sum = 0
10  for j = mid + 1 to high
11      sum = sum + A[j]
12      if sum > right-sum
13          right-sum = sum
14          max-right = j
15  return (max-left, max-right, left-sum + right-sum)
```

We can easily find a maximum subarray crossing the midpoint in time linear in the size of the subarray A[low ,high] using the left pseudo code. This problem is not a smaller instance of our original problem, because it has the added restriction that the subarray it chooses must cross the midpoint.

Since this subarray must contain A[mid], the for loop of lines 3–7 starts the index i at mid and works down to low, so that every subarray it considers is of the form A[i,mid].

```
FIND-MAXIMUM-SUBARRAY(A, low, high)
1  if high == low
2      return (low, high, A[low])          // base case: only one
3  else mid = ⌊(low + high)/2⌋
4      (left-low, left-high, left-sum) =
           FIND-MAXIMUM-SUBARRAY(A, low, mid)
5      (right-low, right-high, right-sum) =
           FIND-MAXIMUM-SUBARRAY(A, mid + 1, high)
6      (cross-low, cross-high, cross-sum) =
           FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
7      if left-sum ≥ right-sum and left-sum ≥ cross-sum
8          return (left-low, left-high, left-sum)
9      elseif right-sum ≥ left-sum and right-sum ≥ cross-sum
10         return (right-low, right-high, right-sum)
11     else return (cross-low, cross-high, cross-sum)
```

Line 3 does the divide part.

Lines 4 and 5 conquer by recursively finding maximum subarrays within the left and right subarrays, respectively.

Lines 6–11 form the combine part.

Time Complexity

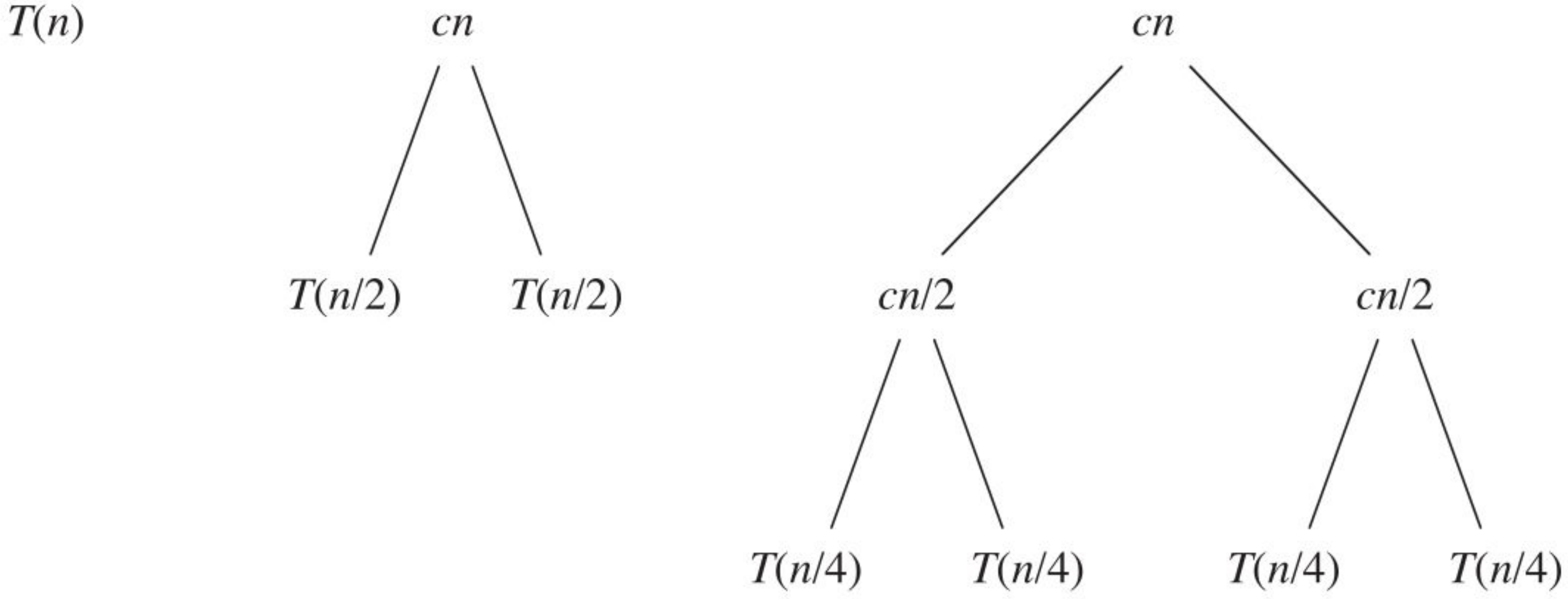
Line 1, 2 and 3 tasks constant time. Line 4 and 5 tasks $T(n/2)$ since it halves the input size n. Line 6 takes $O(n)$ time. Lines 7 to 11 takes again $O(1)$. Therefore,

$T(n) = 7xO(1) + T(n/2) = T(n/2) + O(n)$ where $n > 1$, and $T(n) = O(1)$ where $n = 1$.

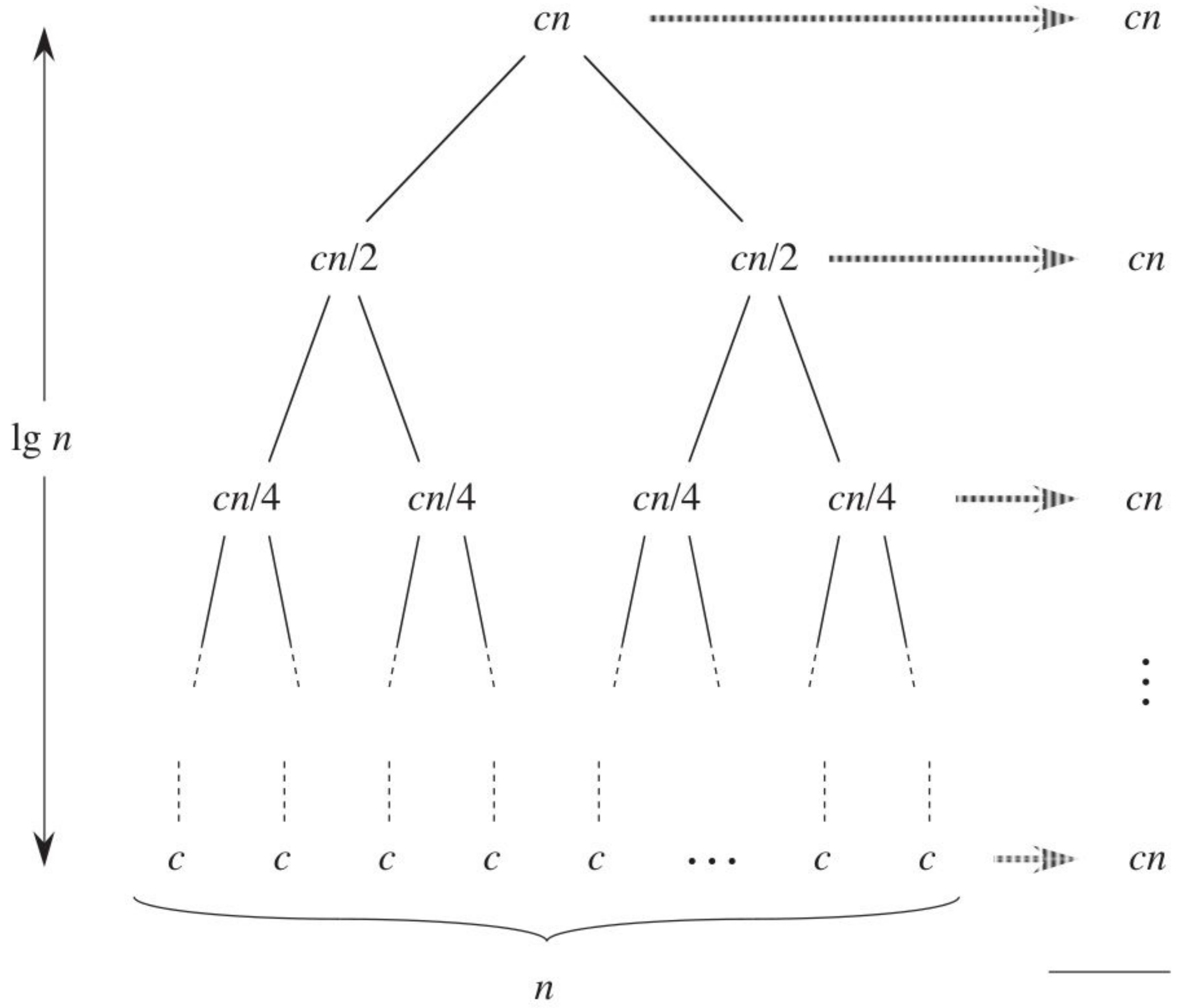
If we solve the above recurrence using Recursion Tree we will find $T(n) = O(n\log_2 n)$.

Because, one node can only have two nodes (two subarrays half of the size of an array). Thus, it becomes a binary tree. And the height of the binary tree is $\text{floor}(\log_2 n)+1$ and in each level total number of computation is n.

Hence, $n\log_2 n$ complexity.



(a) (b) (c)


cn
cn
cn/2
cn/2
cn
lg n
cn/4
cn/4
cn/4
cn/4
cn
c c c c c … c c
cn
n