Largest Subarray Problem

Dr. Anand M
Id: 102763
Assistant Professor
Department of Computer Science and Engineering
SRM Institute of Science and Technology

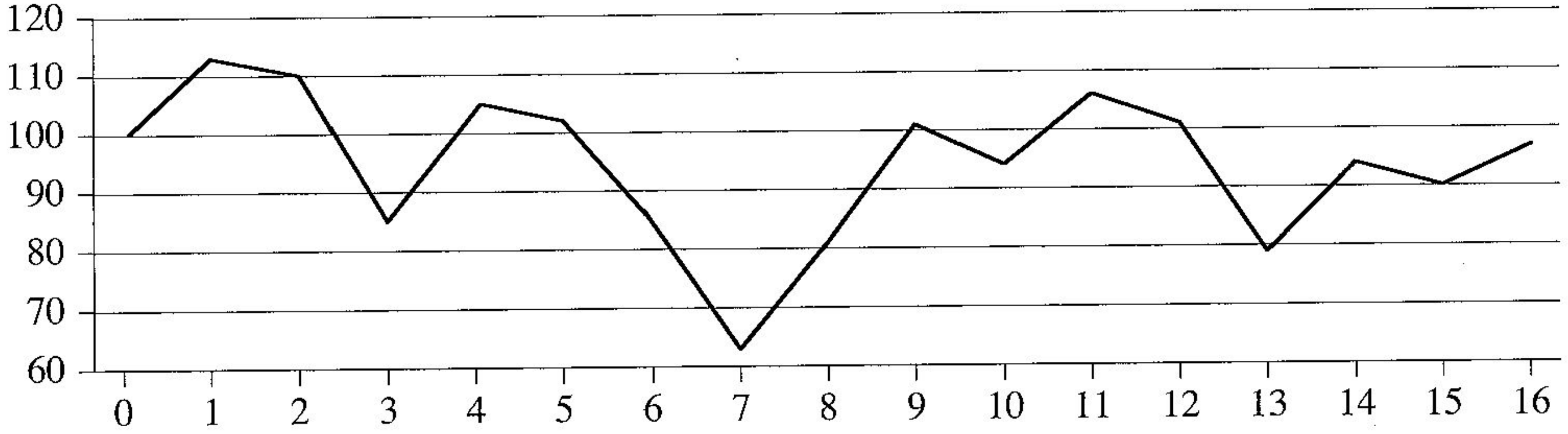
Largest Subarray Problem

Problem: given an array of n numbers, find the (a) contiguous subarray whose sum has the largest value.

Application: an unrealistic stock market game, in which you decide when to buy and see a stock, with full knowledge of the past and future. ***The restriction*** is that you can perform just one ***buy*** followed by a ***sell***. The buy and sell both occur right after the close of the market.

The interpretation of the numbers: each number represents the stock value at closing on any particular day.

Largest Subarray Problem

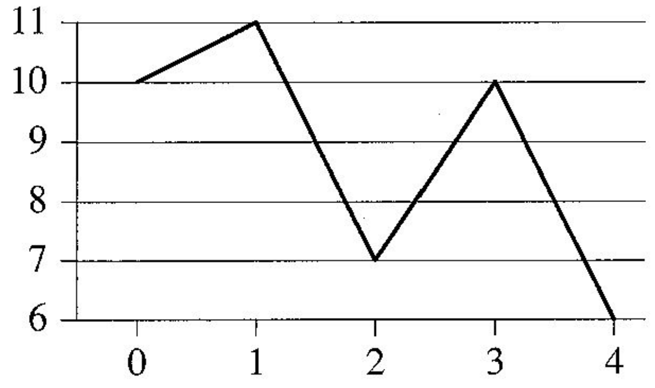


Day	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Price	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97
Change		13	−3	−25	20	−3	−16	−23	18	20	−7	12	−5	−22	15	−4	7

Figure 4.1 Information about the price of stock in the Volatile Chemical Corporation after the close of trading over a period of 17 days. The horizontal axis of the chart indicates the day, and the vertical axis shows the price. The bottom row of the table gives the change in price from the previous day.

Largest Subarray Problem

Another Example: buying low and selling high, even with perfect knowledge, is not trivial:



Day	0	1	2	3	4
Price	10	11	7	10	6
Change		1	−4	3	−4

Figure 4.2 An example showing that the maximum profit does not always start at the lowest price or end at the highest price. Again, the horizontal axis indicates the day, and the vertical axis shows the price. Here, the maximum profit of \$3 per share would be earned by buying after day 2 and selling after day 3. The price of \$7 after day 2 is not the lowest price overall, and the price of \$10 after day 3 is not the highest price overall.

A brute-force solution

- $O(n^2)$ Solution
- Considering C_n^2 pairs
- Not a pleasant prospect if we are rummaging through long time-series (Who told you it was easy to get rich???), even if you are allowed to post-date your stock options…

A Better Solution: Max Subarray

Transformation: Instead of the daily price, let us consider the daily change: A[i] is the difference between the closing value on day i and that on day i-1.

The problem becomes that of finding a contiguous subarray the sum of whose values is maximum.

- On a first look this seems even worse: roughly the same number of intervals (one fewer, to be precise), and the requirement to add the values in the subarray rather than just computing a difference: $\Omega(n^3)$?

Max Subarray

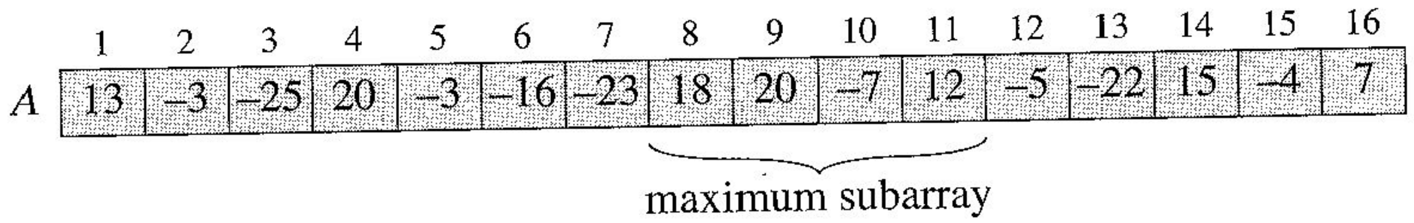


Figure 4.3 The change in stock prices as a maximum-subarray problem. Here, the subarray $A[8\,..\,11]$, with sum 43, has the greatest sum of any contiguous subarray of array A.

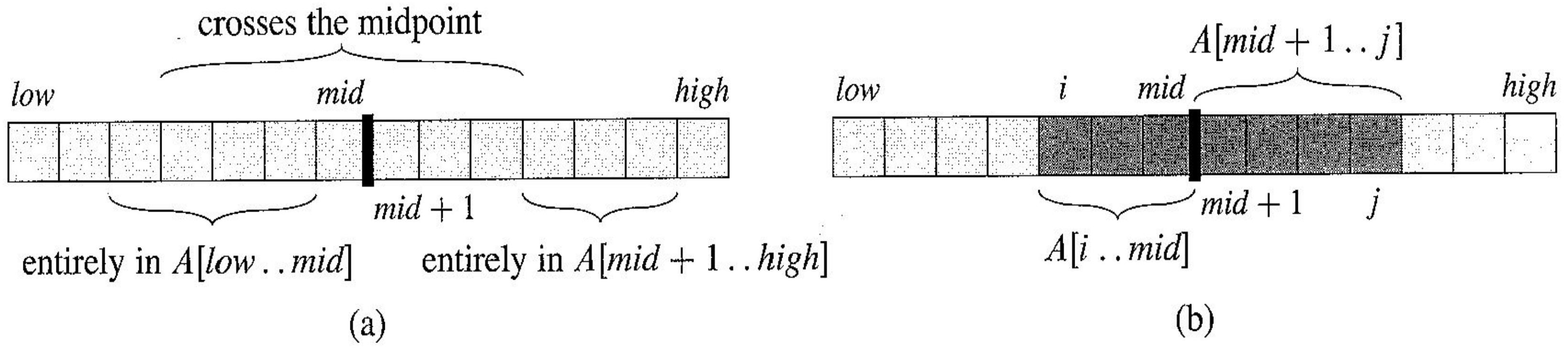


Figure 4.4 **(a)** Possible locations of subarrays of $A[low\,..\,high]$: entirely in $A[low\,..\,mid]$, entirely in $A[mid+1\,..\,high]$, or crossing the midpoint *mid*. **(b)** Any subarray of $A[low\,..\,high]$ crossing the midpoint comprises two subarrays $A[i\,..\,mid]$ and $A[mid+1\,..\,j]$, where $low \le i \le mid$ and $mid < j \le high$.

Max Subarray

How do we divide?

We observe that a maximum contiguous subarray A[i…j] must be located as follows:

1. It lies entirely in the left half of the original array: [low…mid];
2. It lies entirely in the right half of the original array: [mid+1…high];
3. It straddles the midpoint of the original array: i ≤ mid < j.

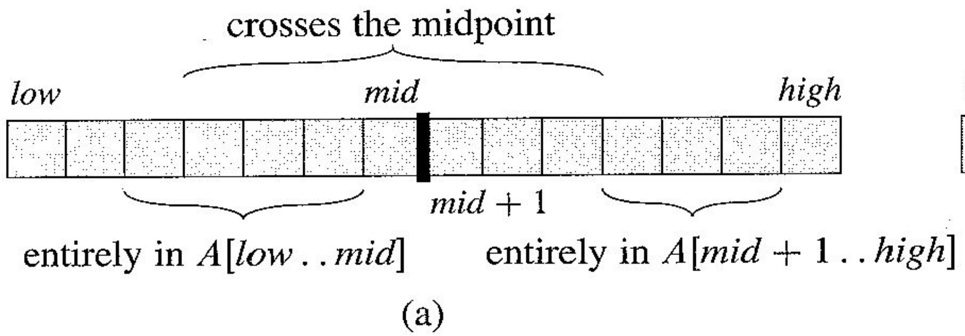
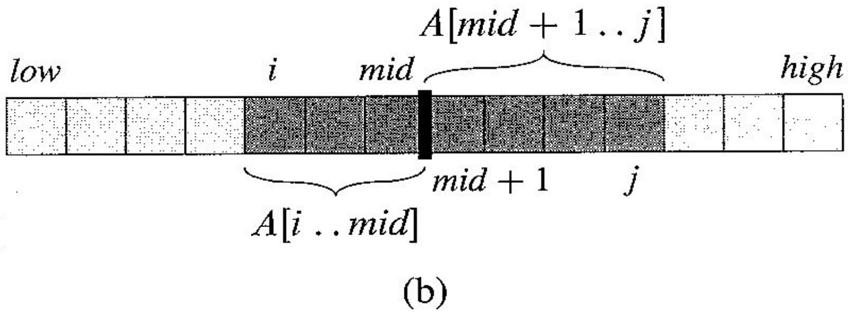


Figure 4.4 **(a)** Possible locations of subarrays of $A[low\,..\,high]$: entirely in $A[low\,..\,mid]$, entirely in $A[mid+1\,..\,high]$, or crossing the midpoint *mid*. **(b)** Any subarray of $A[low\,..\,high]$ crossing the midpoint comprises two subarrays $A[i\,..\,mid]$ and $A[mid+1\,..\,j]$, where $low \le i \le mid$ and $mid < j \le high$.



Max Subarray: Divide & Conquer

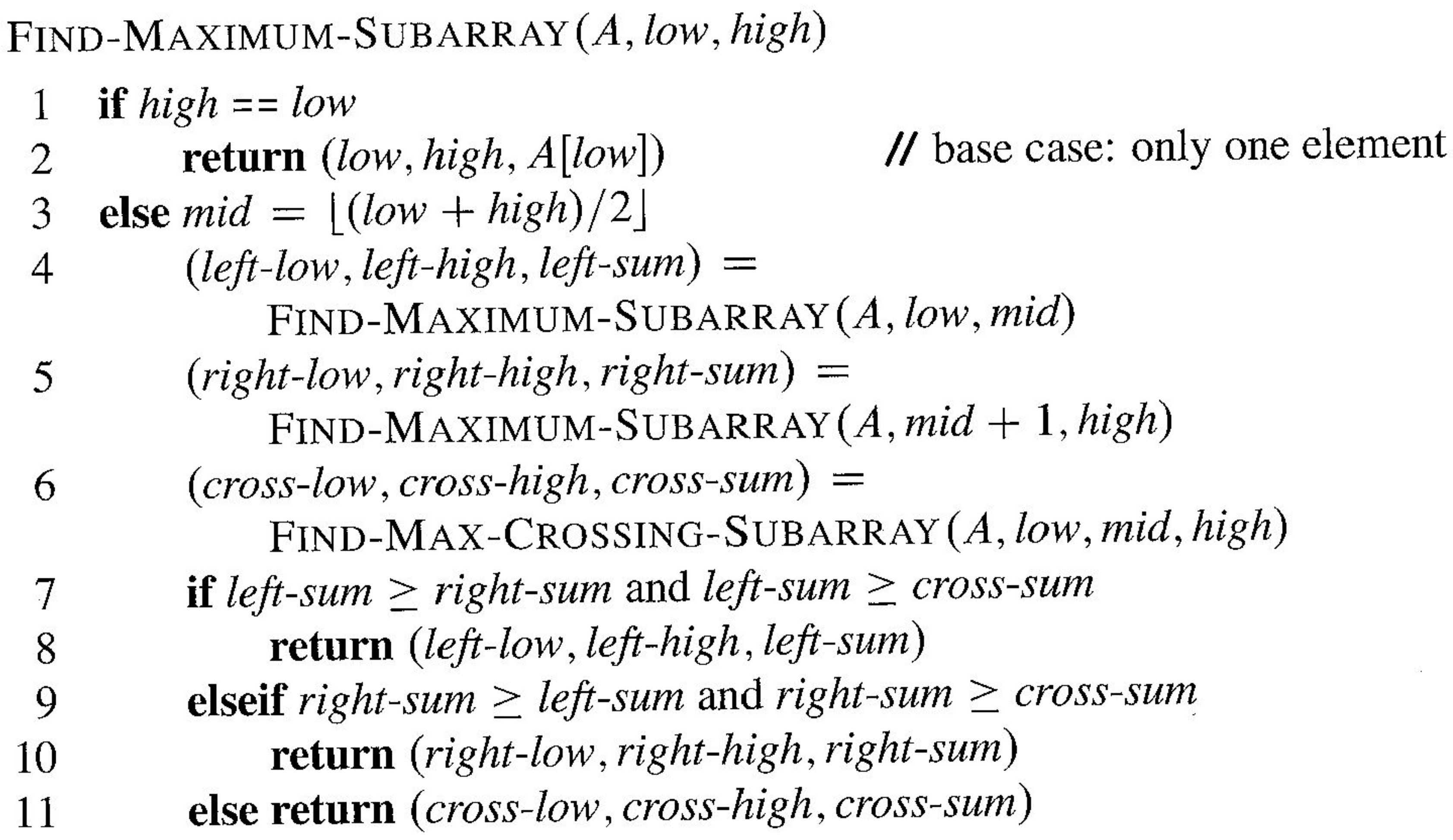
- The “left” and “right” subproblems are smaller versions of the original problem, so they are part of the standard Divide & Conquer recursion.
- The “middle” subproblem is not, so we will need to count its cost as part of the “combine” (or “divide”) cost.
  - The crucial observation (and it may not be entirely obvious) is that **we can find the maximum crossing subarray in time linear in the length of the A[low…high] subarray**.

How? A[i,…,j] must be made up of A[i…mid] and A[m+1…j] – so we find the largest A[i…mid] and the largest A[m+1…j] and combine them.

The middle subproblem

```
FIND-MAX-CROSSING-SUBARRAY($A$, $low$, $mid$, $high$)
$left\text{-}sum = -\infty$
$sum = 0$
for $i = mid$ downto $low$
    $sum = sum + A[i]$
    if $sum > left\text{-}sum$
        $left\text{-}sum = sum$
        $max\text{-}left = i$
$right\text{-}sum = -\infty$
$sum = 0$
for $j = mid + 1$ to $high$
    $sum = sum + A[j]$
    if $sum > right\text{-}sum$
        $right\text{-}sum = sum$
        $max\text{-}right = j$
return ($max\text{-}left$, $max\text{-}right$, $left\text{-}sum + right\text{-}sum$)
```

Algorithms: Max Subarray



```
FIND-MAXIMUM-SUBARRAY($A, low, high$)
if $high == low$
    return ($low, high, A[low]$)                    // base case: only one element
else $mid = \lfloor (low + high)/2 \rfloor$
    ($left\text{-}low, left\text{-}high, left\text{-}sum$) =
        FIND-MAXIMUM-SUBARRAY($A, low, mid$)
    ($right\text{-}low, right\text{-}high, right\text{-}sum$) =
        FIND-MAXIMUM-SUBARRAY($A, mid + 1, high$)
    ($cross\text{-}low, cross\text{-}high, cross\text{-}sum$) =
        FIND-MAX-CROSSING-SUBARRAY($A, low, mid, high$)
    if $left\text{-}sum \geq right\text{-}sum$ and $left\text{-}sum \geq cross\text{-}sum$
        return ($left\text{-}low, left\text{-}high, left\text{-}sum$)
    elseif $right\text{-}sum \geq left\text{-}sum$ and $right\text{-}sum \geq cross\text{-}sum$
        return ($right\text{-}low, right\text{-}high, right\text{-}sum$)
    else return ($cross\text{-}low, cross\text{-}high, cross\text{-}sum$)
```

Max Subarray: Algorithm Efficiency

We finally have:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 2T(n/2) + \Theta(n) & \text{if } n > 1. \end{cases}$$

The recurrence has the same form as that for MERGESORT, and thus we should expect it to have the same solution T(n) = Θ(n lg n).

This algorithm is clearly substantially faster than any of the brute-force methods. It required some cleverness, and the programming is a little more complicated – but the payoff is large.

Assignment Work

1. Write a pseudocode for a divide-and-conquer algorithm for finding the position of the largest element in an array of n numbers.
2. Set up and solve a recurrence relation for the number of key comparisons made by your algorithm.
3. How does this algorithm compare with the brute-force algorithm for this problem?