Chap.4 Divide and Conquer

- Maximum subarray
- The substitution method
- The recursion-tree method
- The master method

Designing Algorithms

- There are a number of design paradigms for algorithms that have proven useful for many types of problems
- Insertion sort : incremental approach
- Other examples of design approaches
  - divide and conquer
  - dynamic programming
  - greedy algorithms…

Divide and Conquer

- A good divide and conquer algorithm generally implies an easy recursive version of the algorithm
- Three steps
  - **Divide** the problem into a number of subproblems
  - **Conquer** the subproblems by solving them recursively. When the subproblem size is small enough, just solve the subproblem.
  - **Combine** subproblems to form the solution of the original problem

Recurrence

- Definition
  a recurrence is an equation or inequality that describes a function in terms of its value on smaller inputs
- Ex)

$$T(n) = \begin{cases} \Theta(1) \\ aT(n/b) + D(n) + C(n) \end{cases}$$

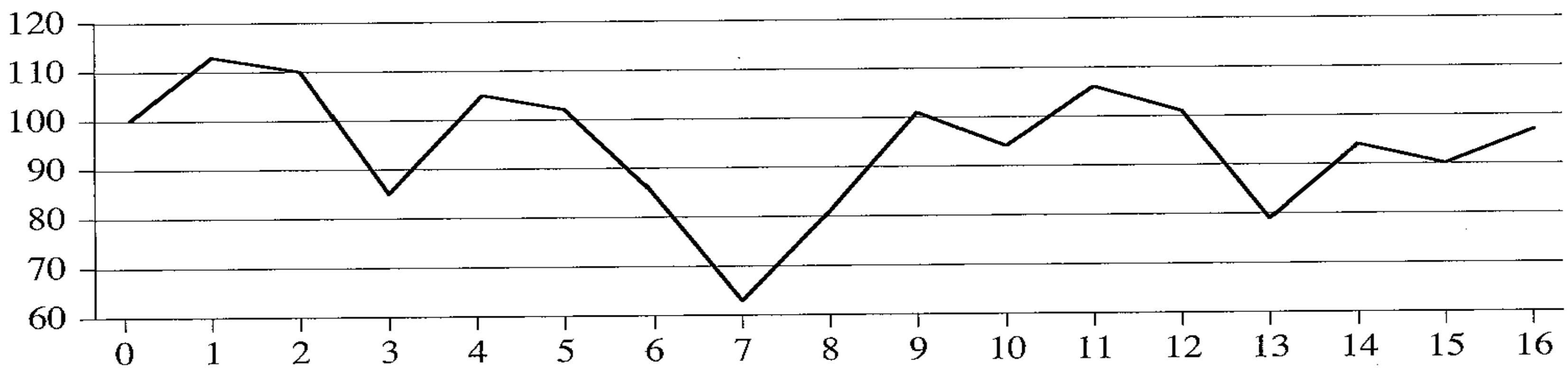
Conquer cost Divide cost Combine cost

Why Recurrences?

- The complexity of many interesting algorithms is easily expressed as a recurrence (especially divide and conquer algorithms)
- The complexity of recursive algorithms is readily expressed as a recurrence.

Maximum Subarray Problem

- Stock investment: Buy one unit of stock only one time and then sell it at a later date
- Goal: to maximize the profit

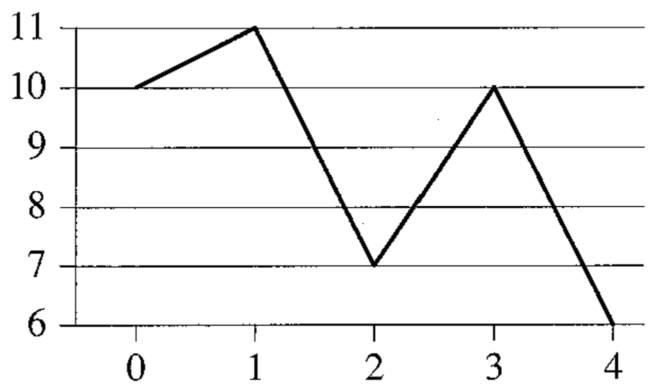


Day	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Price	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97
Change		13	−3	−25	20	−3	−16	−23	18	20	−7	12	−5	−22	15	−4	7

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.

One potential solution?

- Find the highest price and search left to find the lowest prior price
- Find the lowest price and search right to find the highest later price
- Take the pair with the greater difference
- Do not work! See counterexample below.



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.

Maximum Subarray Problem

- Consider the daily change in price
- Maximum subarray problem: find the nonempty, contiguous subarray of A whose values have the largest sum.

	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

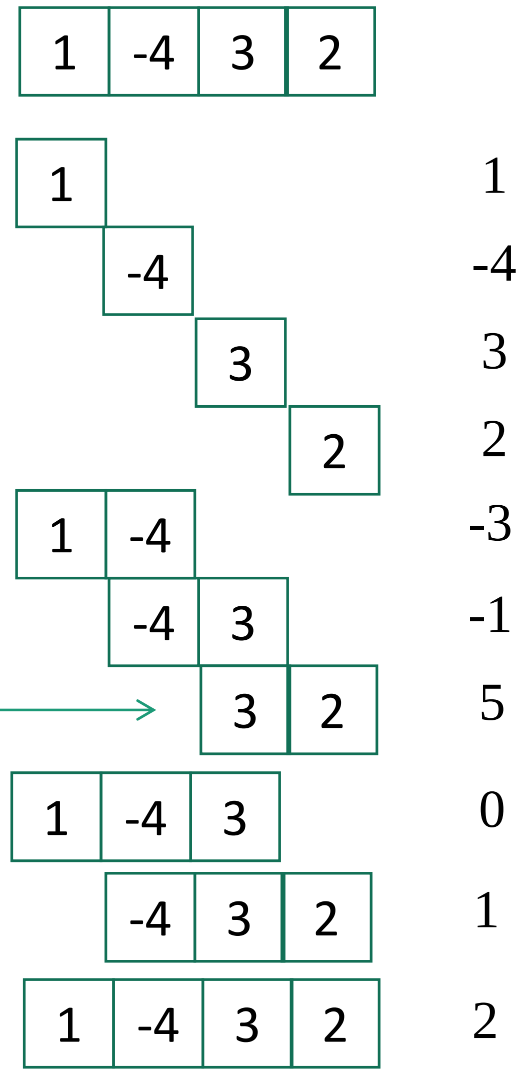
maximum subarray

What is a Maximum Subarray ?

Target array :

All the subarrays:

Maximum!



The subarray with the largest sum

What is the brute-force T(n)?

$O(n^2)$

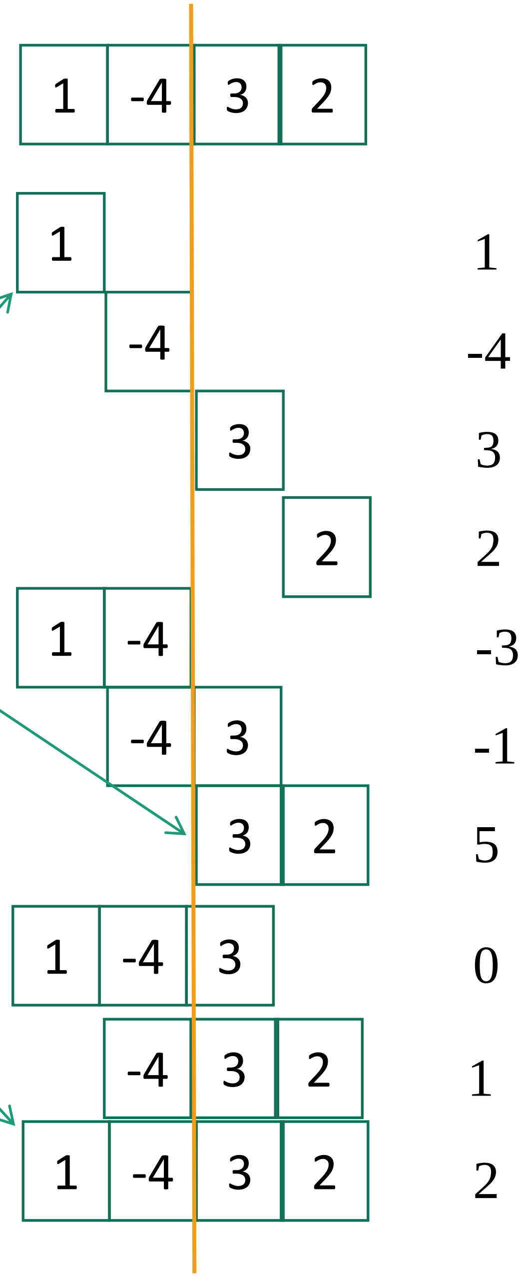
Maximum Subarray Problem

- We need to find the better algorithm than brute force algorithm.
- How about Divide & Conquer algorithm?

Target array :

All the subarrays:

The problem can be then solved by:



1. Find the max in left subarrays

2. Find the max in right subarrays

3. Find the max in crossing subarrays

4. Choose the largest one from those 3 as the final result

Divide target array into 2 arrays.

We then have 3 types of subarrays:

1.The ones belong to the left array

2.The ones belong to the right array

3.The ones crossing the mid point

The whole algorithm

FindMaxSub ([1 | -4 | 3 | 2])

1. Find the max in left subarrays **FindMaxSub** ([1 | -4])

2. Find the max in right subarrays **FindMaxSub** ([3 | 2])

3. Find the max in crossing subarrays

   Scan [1 | -4] once, and scan [3 | 2] once

4. Choose the largest one from those 3 as the final result

Divide and Conquer

- Suppose we want to find a maximum subarray of A[*low..high]*
- Divide and conquer will find the midpoint, say mid, of the subarray, and consider the subarrays A[*low..mid] and A[mid+1..high]*
- Any contiguous subarray A[*i..j] must lie in* one area out of three possibilities

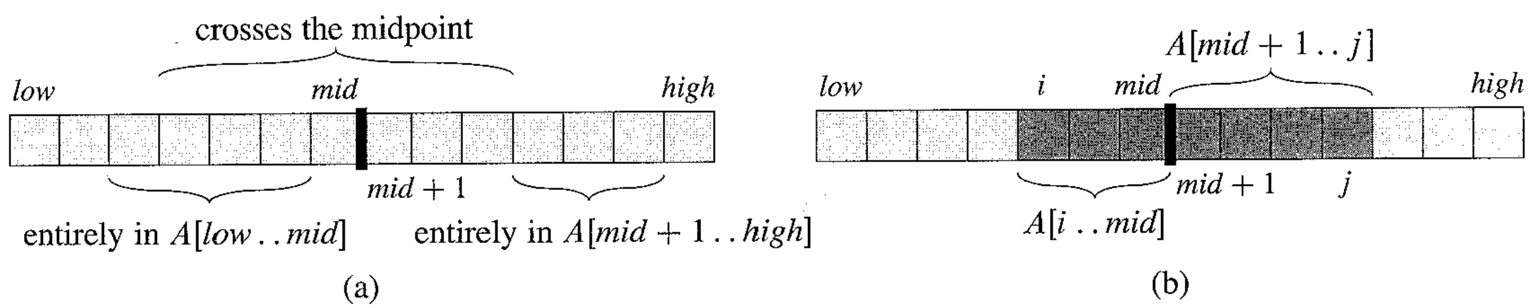


Figure 4.4 **(a)** Possible locations of subarrays of $A[low \,..\, high]$: entirely in $A[low \,..\, mid]$, entirely in $A[mid + 1 \,..\, high]$, or crossing the midpoint *mid*. **(b)** Any subarray of $A[low \,..\, high]$ crossing the midpoint comprises two subarrays $A[i \,..\, mid]$ and $A[mid +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

Find Max Crossing Subarray

- First, it is easy to find a maximum subarray crossing the midpoint
- We just need to find maximum subarrays of the form *A[i..mid] and A[mid+1..j]* and combine them

Find Max Crossing Subarray

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

Find Maximum subarray

- We can then write a divide and conquer algorithm to solve the maximum subarray problem.
- Divide into three cases, and choose the best solution
  - Left subarray
  - Crossing subarray
  - Right subarray

Find Maximum Subarray

```
FIND-MAXIMUM-SUBARRAY(A, low, high)
if high == low
    return (low, high, A[low])                    // base case: only one element
else mid = ⌊(low + high)/2⌋
    (left-low, left-high, left-sum) =
        FIND-MAXIMUM-SUBARRAY(A, low, mid)
    (right-low, right-high, right-sum) =
        FIND-MAXIMUM-SUBARRAY(A, mid + 1, high)
    (cross-low, cross-high, cross-sum) =
        FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
    if left-sum ≥ right-sum and left-sum ≥ cross-sum
        return (left-low, left-high, left-sum)
    elseif right-sum ≥ left-sum and right-sum ≥ cross-sum
        return (right-low, right-high, right-sum)
    else return (cross-low, cross-high, cross-sum)
```

Time complexity? $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(\mathrm{n})$ $O(nlogn)$