CSE- 207
Algorithms

Lecture: 08
Divide and Conquer

Fahad Ahmed

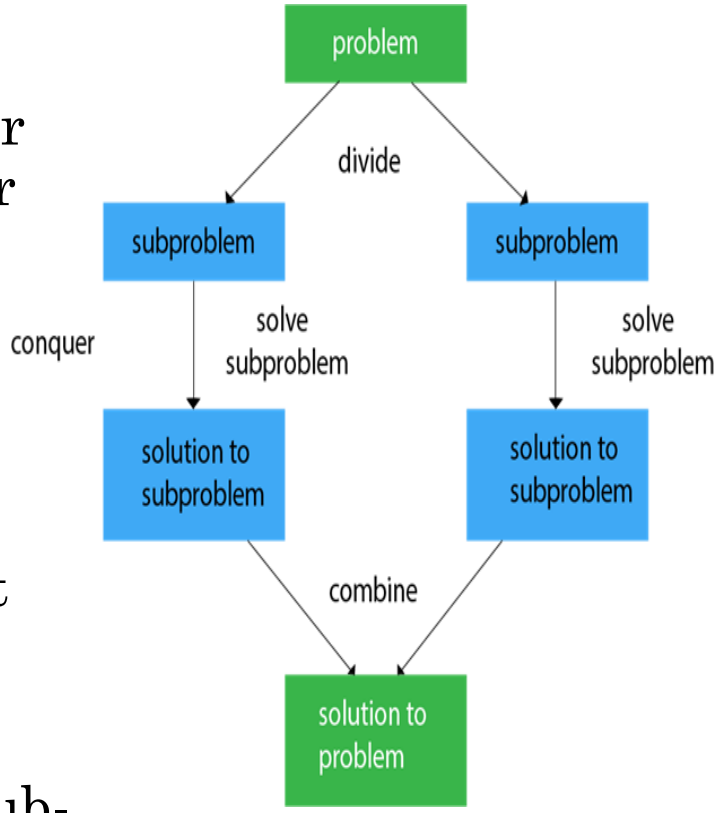
Lecturer, Dept. of CSE

E-mail: fahadahmed@uap-bd.edu

The Maximum Sub-Array

Divide and Conquer

- Divide:
  - **Divide** the problem into number of sub-problems that are smaller instances of the same problem.
- Conquer
  - **Conquer** the sub-problems by solving them recursively. If the sub-problem sizes are small enough, solve them in a straight forwards manner.
- Combine
  - **Combine** the solutions to the sub-problems into the solution for the original problem.



The Maximum Subarray

- The **maximum sub-array problem** is the task of finding the **contiguous sub-array** within a one-dimensional array of numbers which has the **largest sum**.
  - Input:
    - an array a[1...n] of (positive/negative) numbers.
  - Output:
    - Indices i and j such that A[i...j] has the greatest sum of any nonempty, contiguous subarray of a, along with the sum of the values in a[i...j].
  - Note:
    - Maximum subarray might not be unique, though its value is, so we speak of a maximum subarray, rather than the maximum subarray.

The Maximum Subarray - example

- Example 1

Day	0	1	2	3	4
Price	10	11	7	10	6
Change a[…]		1	-4	3	-4

Max Sub Array : a[3] = 3

- Example 2

Day	0	1	2	3	4	5	6	7
Price	10	11	7	10	9	12	15	13
Change a[…]		1	-4	3	-1	3	3	-2

Max Sub Array : a[3…6] = 8

Maximum Subarray -Brute Force algorithm

3	-1	6	-2

SUM

3

3

2

3	-1

8

3	-1	6

6

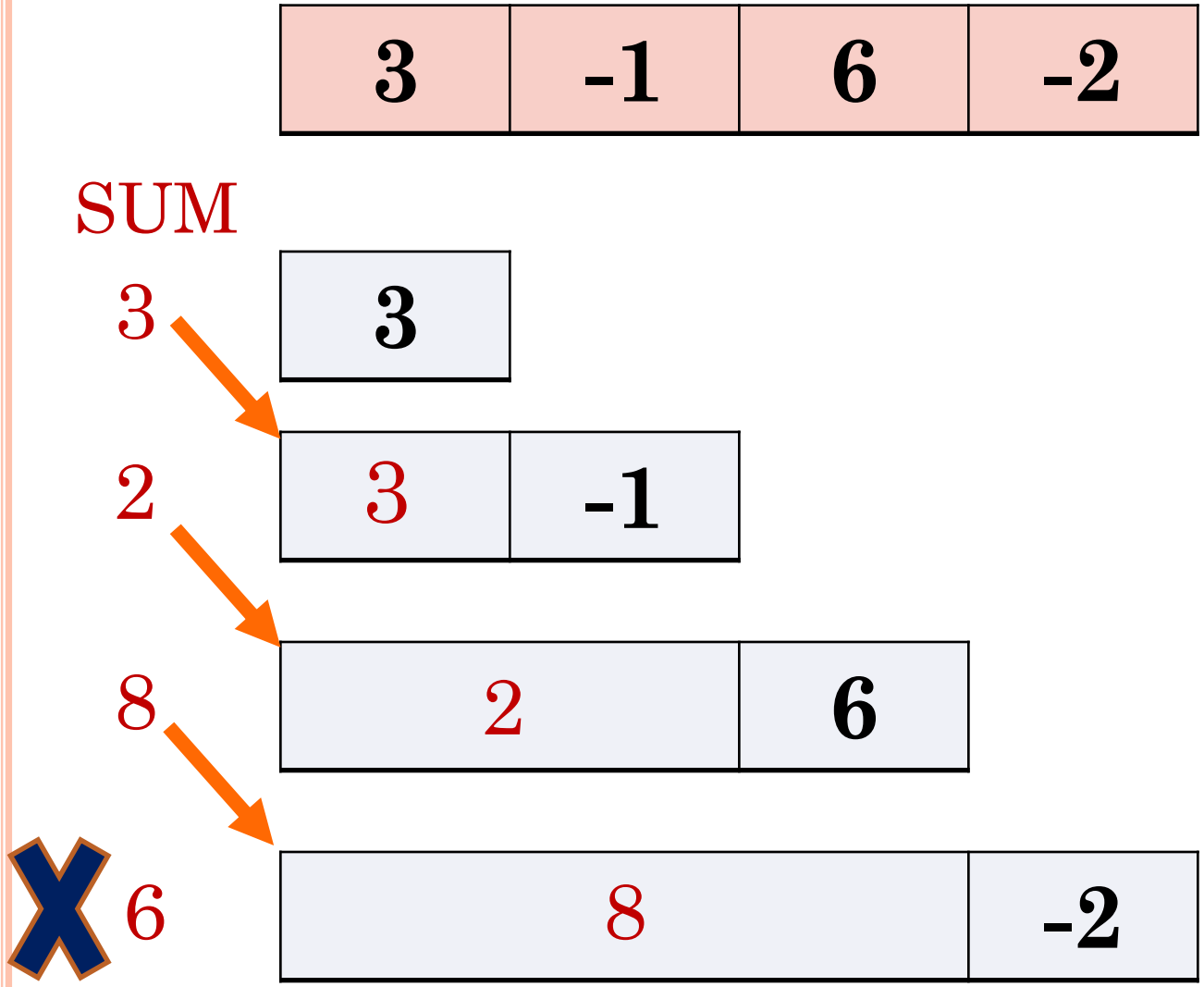
3	-1	6	-2

Maximum Subarray -Brute Force algorithm

```
int maxSubarraySum ( int A [] , int n)
{
   int max_sum = 0
   for(i = 0 to n-1)
   {
      for(j = i to n-1)
      {
         int sum = 0
         for(k = i to j)
            sum = sum + A[k]
         if(sum > max_sum)
            max_sum = sum
      }
   }
return max_sum
}
```

Time Complexity - $\Theta(n^3)$

Maximum Subarray -Brute Force algorithm



Maximum Subarray –Improvised Brute Force algorithm

```
int max_Subarray_Sum ( int A[] , int n)
{
   int max_sum = 0
   for ( i = 0 to n-1)
   {
      sum=0
      for( j = i to n-1)
      {
        sum = sum +  A[j]
        if (sum > max_sum)
            max_sum = sum
       }
   }
retun max_sum
}
```

Time Complexity - $\Theta(n^2)$

RRAY–

it necessary to iterate the inner
op until the end?

uld we determine a definitive
int, iterating past which wouldn't
ad to the correct answer?

Maximum Subarray– divide & conquer

You could divide the array into two equal parts and then recursively find the maximum subarray sum of the left part and the right part.

But what if the actual subarray with maximum sum is formed of some elements from the left and some elements from the right?

Maximum Subarray– divide & conquer

The sub-array we're looking for can be in only one of three places:

1. On the **left part** of the array (between 0 and the mid)
2. On the right part of the array (between the mid + 1 and the end)
3. Somewhere crossing the midpoint.

You could very easily find a cross-sum of elements from the left side and right side which cross the mid-element in linear time.

MAXIMUM SUBARRAY– DIVIDE & CONQUER

Solution Steps

1.Divide the array into two equal parts
2.Recursively calculate the maximum sum for **left** and **right** subarray
3.To find **cross-sum**:-

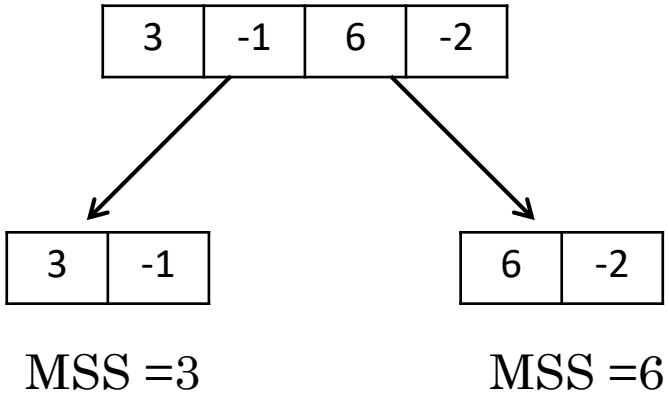
- Iterate from mid to the starting part of the left subarray and at every point, check the maximum possible sum till that point and store in the parameter lsum.

- Iterate from mid+1 to the ending point of right subarray and at every point, check the maximum possible sum till that point and store in the parameter rsum.

- Add **lsum** and **rsum** to get the **cross-sum**

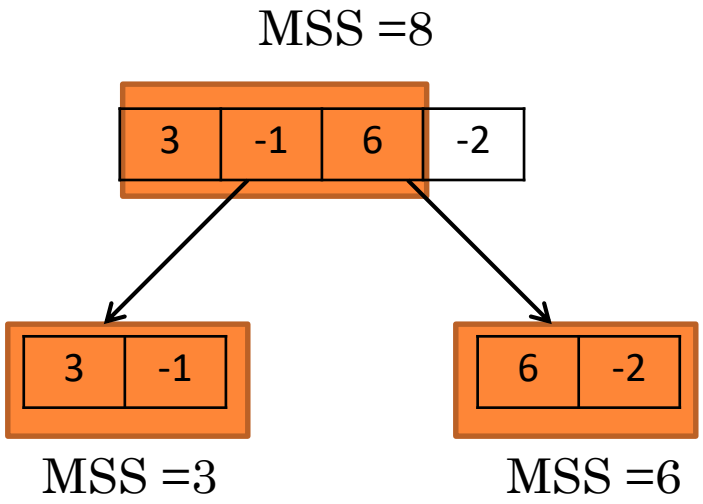
4. **Return the maximum** among (left, right, cross-sum)

Maximum Subarray–divide & conquer



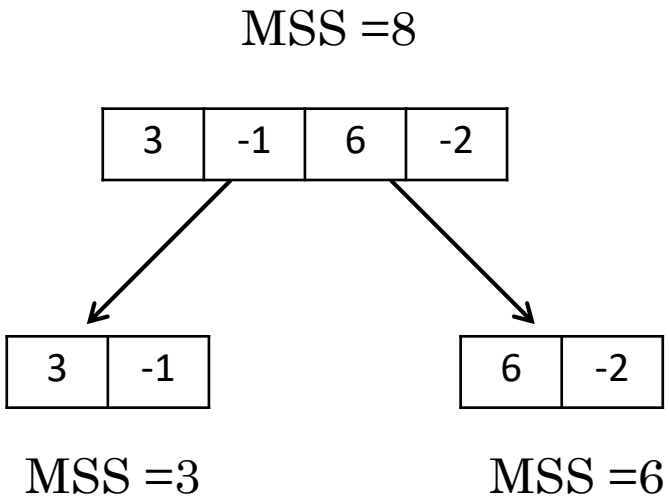
- Divide the array to 2 subarrays.
- Find the MSS of 2 subarrays.
  - Also need the find the MSS that crosses the 2 subarrays.
- Combine the Solution.

Maximum Subarray–divide & conquer



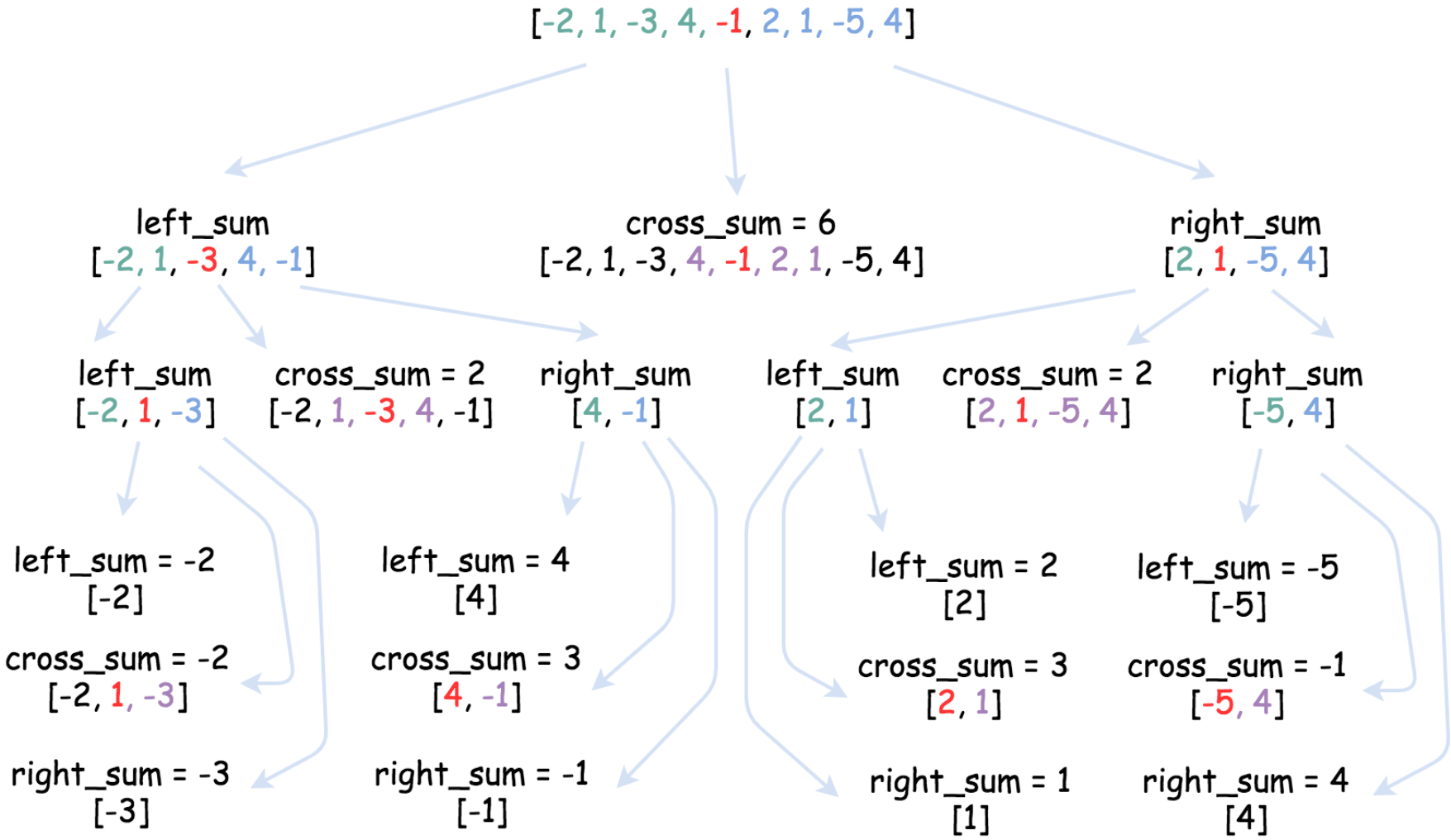
- Combine the solution
- MSS of the whole array could be
  - Entirely in left subarray, or
  - Entirely in right subarray, or
  - crossing the midpoint, some part lie in left subarray and the remaining in right

Maximum Subarray–divide & conquer



- Combine the Solution.
  - MSS of the whole array would be the biggest MSS among three(left, right an cross)

Maximum Subarray– divide & conquer



MAXIMUM SUBARRAY– DIVIDE & CONQUER

```
int maxSubarraySum (int A[], int low, int high)
{
   if (low == high)
      return A[low]
  else
      {
       int mid = low + (high - low)/2
       int left_sum = maxSubarraySum (A, low, mid)
       int right_sum = maxSubarraySum (A, mid+1, high)
       int crossing_Sum = maxCrossingSum(A, low, mid, high)

       return max (left_sum, right_sum, crossing_Sum)
      }
}
```

MAXIMUM SUBARRAY– DIVIDE & CONQUER

```
int maxCrossingSum(int A[], int l, int mid, int r)
{
  int sum = 0
  int lsum = INT_MIN
  for(i = mid to l)
   {
      sum = sum + A[i]
      If (sum > lsum)
      lsum = sum
   }
  sum = 0
  int rsum = INT_MIN
  for(i = mid+1 to r)
   {
      sum = sum + A[i]
      If (sum > rsum)
      rsum = sum
   }
return (lsum + rsum)
}
```

Time Complexity

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

- T(n/2) for each subarray of size n/
- Θ(n) for the cross array MSS
- Θ(1) for the comparing the 3 MSS(left, right and cross)

- This is similar to merge sort
- T(n) =θ(*n log n*)

RRAY– DIVIDE & CONQUER

)oes this algorithm
ppropriate handle
egative numbers?

Maximum Subarray

Efficient solutions

Five solutions for this problem:-

1. Brute force approach I : Using 3 nested loops
2. Brute force approach II : Using 2 nested loops
3. Divide and Conquer approach : Similar to merge sort
4. Dynamic Programming Approach I : Using an auxiliary array
5. Dynamic Programming Approach II : Kadanes's Algorithm

Maximum Subarray

Solution Approach	Time Complexity	Space Complexity
Brute Force approach 1	O(n^3)	O(1)
Brute Force approach 2	O(n^2)	O(1)
Divide and Conquer Approach	O(nlogn)	O(logn)
Dynamic Programming using auxiliary array	O(n)	O(n)
Kadane Algorithm	O(n)	O(1)

Maximum Subarray–divide & conquer

Agand's Algorithm

Time Complexity	O(n)
Algorithmic Paradigm	Divide and conquer
Required Storage	O(n)

Ref: https://upaspro.com/maximum-subarray-sum/

Maximum Subarray– Kadane's Algorithm

Kadane's Algorithm

Simple idea of the Kadane's algorithm is to look for all positive contiguous segments of the array (max_ending_here is used for this).

And keep track of maximum sum contiguous segment among all positive segments (max_so_far is used for this).

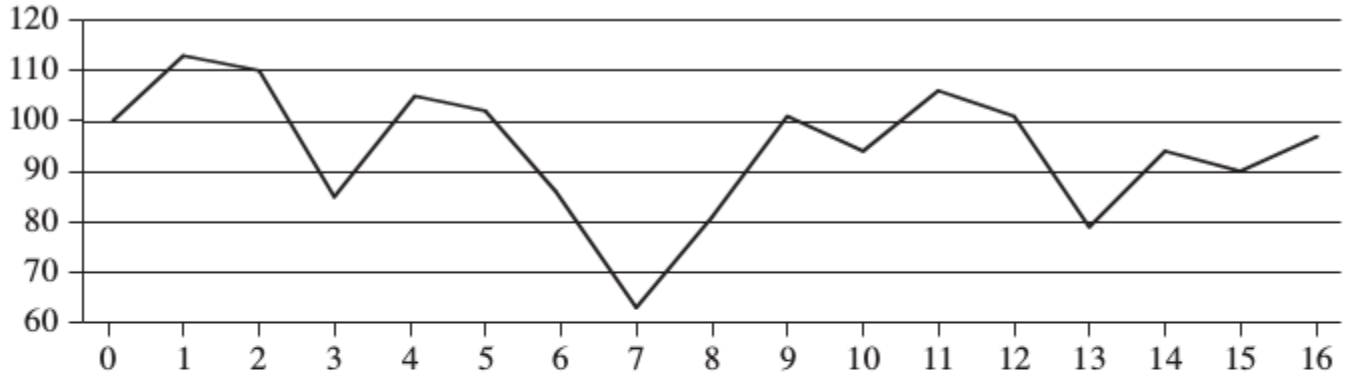
Each time we get a positive sum compare it with max_so_far and update max_so_far if it is greater than max_so_far

Application/example

Volatile Chemical Corporation – A Stock company.

- Suppose that you been offered the opportunity to invest in the Volatile Chemical Corporation.
- You are allowed to buy one unit of stock only one time and then sell it at a later date, buying and selling after the close of trading for the day.
- To compensate for this restriction, you are allowed to learn what the price of the stock will be in the future.

Volatile Chemical Corporation – A Stock company.



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.

The stock problem

- Your target is to maximize profit
- Find the lowest point/price?
- Choose the highest price?
  - The difference should give you maximum profit.
- But what if the highest comes before the lowest price.

Try Maximum SubArray

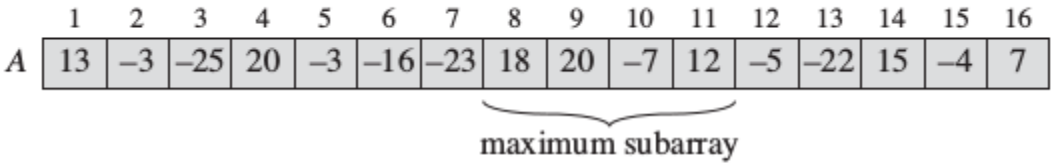
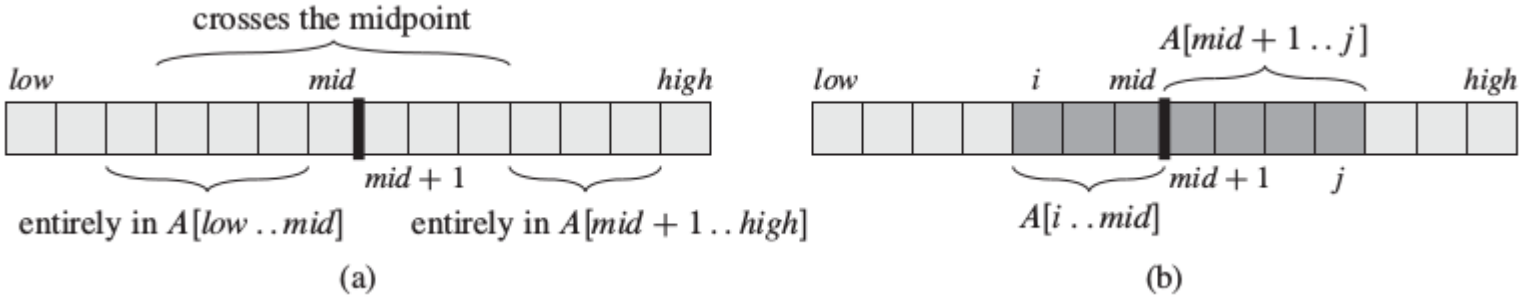


Figure 4.3 The change in stock prices as a maximum-subarray problem. Here, the subarray $A[8..11]$, with sum 43, has the greatest sum of any contiguous subarray of array A.

- Brute force
  - Try all pair
  - will give you $\theta(n^3)$ or $\theta(n^2)$ time complexity

TRY MAXIMUM SUBARRAY – DIVIDE & CONQUER



- We want to find a maximum subarray of A[low : high]
- Any contiguous subarray A[i : j] must lie:
  - Entirely in the subarray A[low : mid], so that low ≤ i ≤ j ≤ mid
  - entirely in the subarray A[mid + 1 : high], so that mid ≤ i ≤ j ≤ high, or
  - crossing the midpoint, so that low ≤ i ≤ mid ≤ j ≤ high

Try these

- What does this function return if all numbers are positive?

- What does this function return if all numbers are negative?

Matrix Multiplication

Matrix Multiplication

- Matrix Multiplication is one of the most fundamental operation in Machine Learning and optimizing it is the key to several optimizations.

- Here, A_{ij} and B_{ij} are 2 x 2 matrices.

A =
1 3
7 5

B =
6 8
4 2

C =
1*6+3*4 1*8+3*2
7*6+5*4 7*8+5*2

=
18 14
62 66

Matrix Multiplication

```
void multiply(int A[][N], int B[][N], int C[][N])
{
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
        {
            C[i][j] = 0;
            for (int k = 0; k < N; k++)
            {
                C[i][j] += A[i][k]*B[k][j];
            }
        }
    }
}
```

In this algorithm, the statement "C[i][j] += A[i][k] * B[k][j]" executes n^3 times as evident from the three nested for loops and is the most costly operation in the algorithm. So, the time complexity of the naive algorithm is $O(n^3)$.

Matrix Multiplication using recursion

Consider the following matrices A and B:

matrix
A = |a b|,
|c d|

matrix
B = |e f|
|g h|

matrix
C = |ae+bg af+bh|
|ce+dg cf+dh|

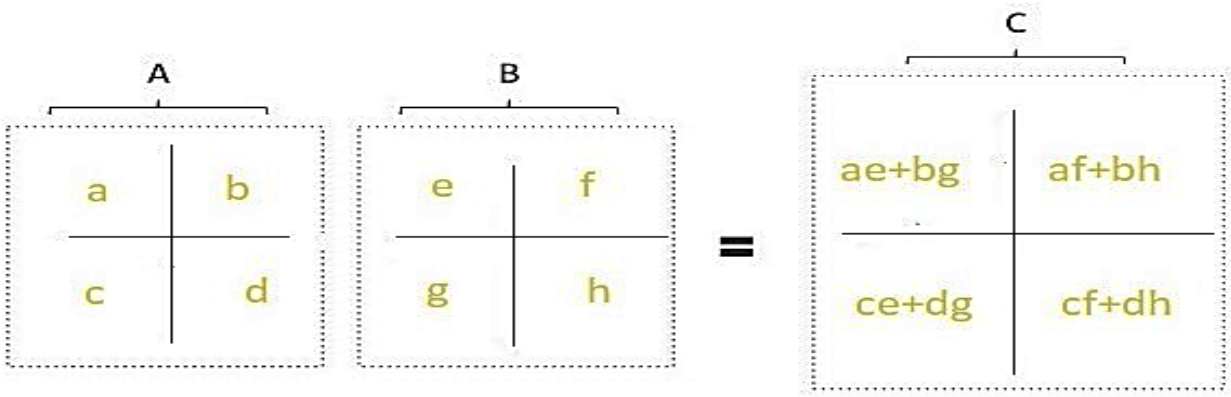
There will be 8 recursive calls:

a * e
b * g
a * f
b * h
c * e
d * g
c * f
d * h

Using the Master Theorem with **T(n) = 8T(n/2) + O(n^2)** we still get a runtime of O(n^3).

The above strategy is the basic O(N^3) strategy.

STRASSEN'S MATRIX MULTIPLICATION ALGORITHM



Strassen's insight was that **we don't actually need 8 recursive calls** to complete this process. **We can finish the call with 7 recursive calls** and a little bit of addition and subtraction.

Following is simple **Divide and Conquer method** to multiply two square matrices.
1) Divide matrices A and B in 4 sub-matrices of size N/2 x N/2 as shown in the below diagram.
2) Calculate following values recursively. ae + bg, af + bh, ce + dg and cf + dh.

Strassen's Matrix Multiplication algorithm

p1 = a(f-h)
p2 = (a+b)h
p3 = (c+d)e
p4 = d(g-e)
p5 = (a+d)(e+h)
p6 = (b-d)(g+h)
p7 = (a-c)(e+f)

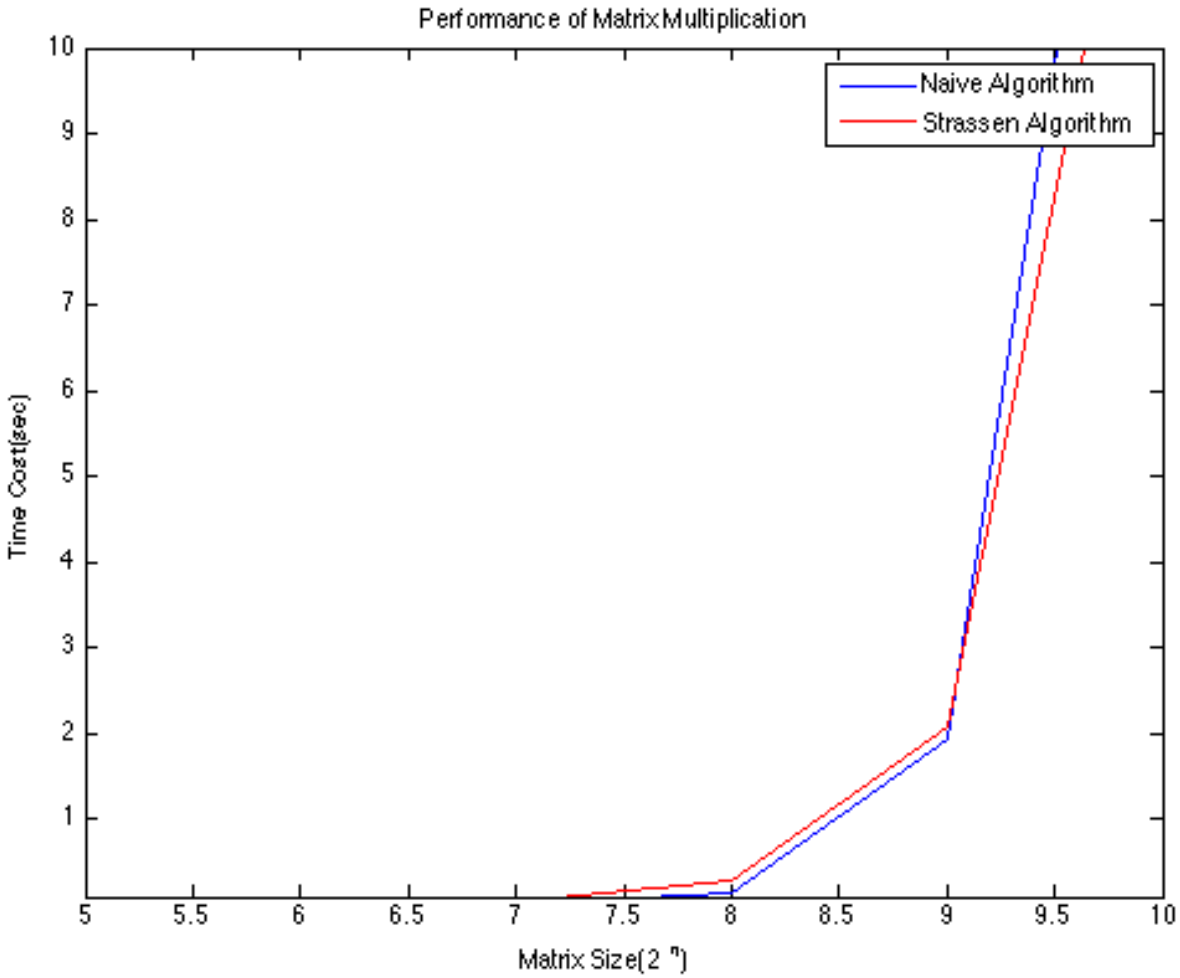
$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} p5+p4-p2+p6 & p1+p2 \\ p3+p4 & p1+p5-p3p7 \end{bmatrix}$$

A B C

The time complexity using the Master Theorem.
T(n) = 7T(n/2) + O(n^2) which leads to **O(n^log(7))** runtime.
This comes out to approximately **O(n^2.8074)** which is better than O(n^3)

Strassen's Matrix Multiplication algorithm

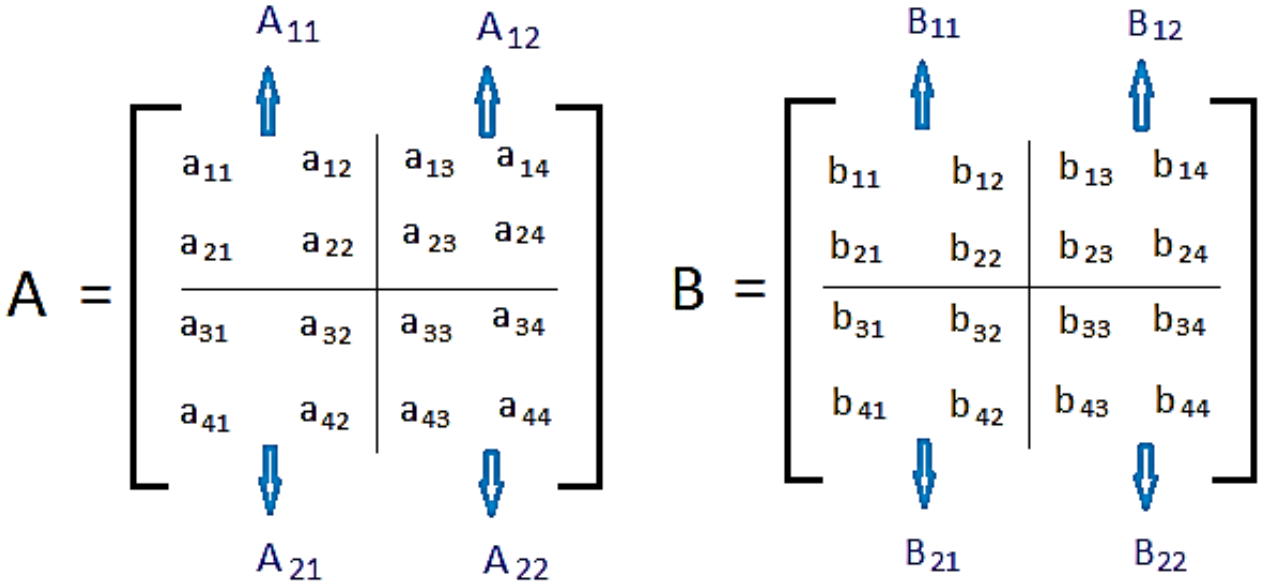
- **Worst case time complexity: Θ(n^2.8074)**



Strassen's Matrix Multiplication algorithm

What if the size is greater then 2×2?

We assume that the matrices are having the dimensions in powers of 2 like 2×2, 4×4, 8×8, 16×16, 256×256 and e.t.c. If it is not of power 2×2 then we can fill zeros and makes it as square matrix of 2×2.



$$A = \left[\begin{array}{cc|cc} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ \hline a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{array}\right] \quad B = \left[\begin{array}{cc|cc} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ \hline b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{array}\right]$$

Strassen's Matrix Multiplication algorithm

Generally Strassen's Method is not preferred for practical applications for following reasons.

1.The constants used in Strassen's method are high and for a typical application Naive method works better.

2.For Sparse matrices, there are better methods especially designed for them.

3.The submatrices in recursion take extra space.

4.Because of the limited precision of computer arithmetic on non-integer values, larger errors accumulate in Strassen's algorithm than in Naive Method

Example of Divide and Conquer

- Closest pair (points)
- Cooley–Tukey Fast Fourier Transform (FFT) algorithm
- Karatsuba algorithm for fast multiplication

Disadvantages of Divide and Conquer

Disadvantages of Divide and Conquer

- Problem decomposition may be very complex and thus not really suitable to divide and conquer.
- Recursion into small/tiny base cases may lead to huge recursive stacks, and efficiency can be lost by not applying solutions earlier for larger base cases.
- The main problem which arises with this algorithm is the recursion is slow that increases the time complexity.
- Since most of its algorithms are designed by incorporating recursion, so it necessitates high memory management. An explicit stack may overuse the space.
- Recursive nature of the solution may end up duplicating sub-problems, dynamic/memoized solutions may be better in some of these cases, like Fibonacci.

ıat will be the solution?

t means we need to know
ther approach of Algorithm.

Greedy Approach

Reference

- Chapter 4.1 (Cormen)
- https://upaspro.com/maximum-subarray-sum/
- https://www.geeksforgeeks.org/largest-sum-contiguous-subarray/
- https://afteracademy.com/blog/maximum-subarray-sum