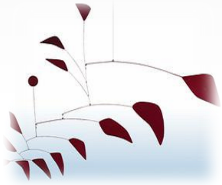


EECS 3101 - Design and Analysis of Algorithms

Shahin Kamali

Topic 2 - Divide & Conquer Technique

and Recursion

Overview

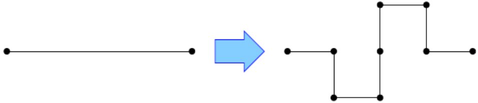
- what is recursion?
- recursion vs. iteration
- analyzing the running time of recursive algorithms

Recursion

- The term **recursion** refers to a method that calls itself.
- Recursion is a powerful programming technique that results in efficient algorithms with concise descriptions.

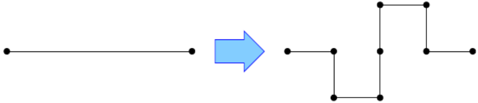
Recursion example

- Suppose we replace every line segment by eight shorter line segments according to the following geometric rule:

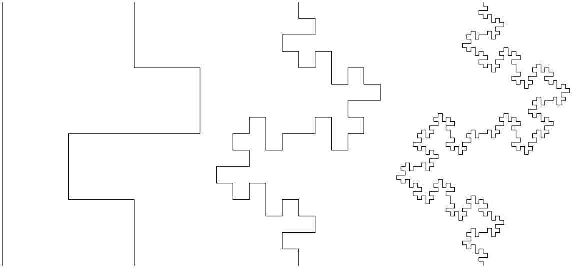


Recursion example

- Suppose we replace every line segment by eight shorter line segments according to the following geometric rule:

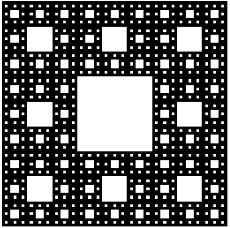


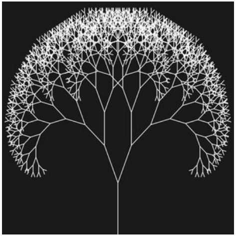
- By applying this rule recursively we obtain the following "fractal":

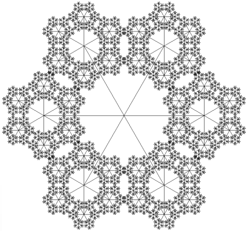


Fractals

- Fractals are beautiful "creatures" often built on the recursion principle:







Iteration versus Recursion

- An iterative algorithm for solving a problem P makes use of a loop to compute a sequence of analogous steps that solve P.

Iteration versus Recursion

- An iterative algorithm for solving a problem P makes use of a loop to compute a sequence of analogous steps that solve P.

$$n! = n(n-1)(n-2)\ldots 3 \cdot 2 \cdot 1$$

Iteration versus Recursion

- An iterative algorithm for solving a problem P makes use of a loop to compute a sequence of analogous steps that solve P.

$$n! = n(n-1)(n-2)\ldots 3 \cdot 2 \cdot 1$$

- A recursive algorithm for solving a problem P computes one step and calls itself to solve the remaining subproblem.

Iteration versus Recursion

- An iterative algorithm for solving a problem P makes use of a loop to compute a sequence of analogous steps that solve P.

$$n! = n(n-1)(n-2)\ldots 3 \cdot 2 \cdot 1$$

- A recursive algorithm for solving a problem P computes one step and calls itself to solve the remaining subproblem.

$$n! = \begin{cases} 1 & n \leq 1 \\ n(n-1)! & n > 1 \end{cases}$$

Iteration versus Recursion

- an iterative algorithm for computing $n!$:

```
FactorialIterative (n)
    result ← 1
    for i ← 1 to n do
        result ← result * i
    return result
```

- This clearly takes $\Theta(n)$ time.

Iteration versus Recursion

- a recursive algorithm for computing n!:

```
FactorialRecursive (n)
1.    result ← 1
2.    if n > 1
3.        result ← n* FactorialRecursive(n − 1)
4.    return result
```

- For the time complexity, we can write

$$f(n) = c + f(n-1) = 2c + f(n-2) = \ldots = c(n-1) + f(1) = \Theta(n)$$

- For the factorial problem, both iterative and recursive functions run in time linear to n.

Fibonacci numbers

- The Fibonacci numbers are the sequence

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \dots$$

Fibonacci numbers

- The Fibonacci numbers are the sequence

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$$

- The first two numbers in the sequence are 0 and 1.
- Each succeeding number (third, fourth, ...) is define as the sum of the two numbers that precede it in the sequence.

Fibonacci numbers

- The Fibonacci numbers are the sequence

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$$

- The first two numbers in the sequence are 0 and 1.
- Each succeeding number (third, fourth, ...) is define as the sum of the two numbers that precede it in the sequence.
- That is, Fibonacci numbers are defined recursively:

$$fib(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ fib(n) + fib(n-1) & n > 1 \end{cases}$$

A Recursive Solution

fib (n)

$result \leftarrow 0$
if $n \leq 1$
    $result \leftarrow 1$
else
    $result \leftarrow fib(n-1) + fib(n-2)$
return *result*

A Recursive Solution

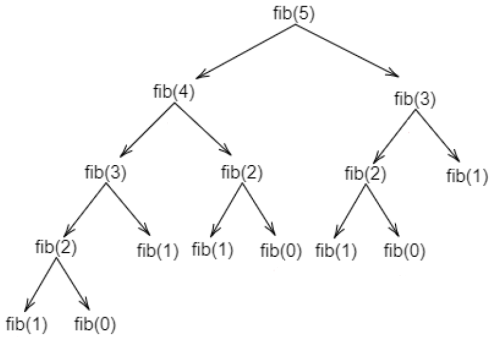
```
fib (n)
    result ← 0
    if n ≤ 1
        result ← 1
    else
        result ← fib(n − 1) + fib(n − 2)
    return result
```

- For the running time of this algorithm, we can write:

$$t(n) = t(n-1) + t(n-2) \geq 2T(n-2) \geq 4T(n-4) \geq 8T(n-6) \geq \ldots$$
$$\geq 2^k T(n-2k) = \Omega(2^{n/2})$$

Recursion Tree

- This implementation of Fibonacci numbers is inefficient because it recomputes values that have already been found.



Efficient Fibonacci Computation

- A more efficient solution can be obtained by storing Fibonacci numbers that have already been computed in an array.

```
fibo (n)
1.    F ← an array of size n
2.    F[1] ← 1    F[2] ← 1
3.    for i = 3 to n
4.        F[i] ← F[i − 1] + F[i − 2]
return F[n]
```

- This runs in $\Theta(n)$

Maximum Subarray Problem

- given an array A of n numbers, find the a contiguous subarray whose sum has the largest value!

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7

  - In this example, it is $18, 20, -7, 12$ for a sum of 43.

Maximum Subarray Problem

- given an array A of n numbers, find the a contiguous subarray whose sum has the largest value!

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7

  - In this example, it is $18, 20, -7, 12$ for a sum of 43.
  - We denote the best solution with a triplet (lo, hi, sum), indicating, start index, end index, and sum of the numbers in the sub-array. In this example, it would be $(7, 10, 43)$

Maximum Subarray Problem

- given an array A of n numbers, find the a contiguous subarray whose sum has the largest value!

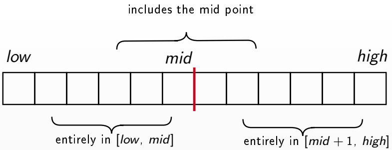
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7

  - In this example, it is $18, 20, -7, 12$ for a sum of 43.
  - We denote the best solution with a triplet (lo, hi, sum), indicating, start index, end index, and sum of the numbers in the sub-array. In this example, it would be $(7, 10, 43)$

- **Solution 1:** try all possible sub-arrays! There are $\binom{n}{2} = \Theta(n^2)$ sub-arrays; thus the running time of this "Brute-Force" solution is $\Omega(n^2)$.

Maximum Subarray Problem

- **Solution 2:** Use a divide and conquer approach!
- Suppose we want to find the sub-array with maximum sum from the input array from index *low* to index *hi*. There are three possibilities:
  - It is entirely in the left half of the range $[low, hi]$
  - It is entirely in the right half of the range $[low, hi]$
  - It straddles the midpoint $mid = \lfloor (low + high)/2$ of the range.
- We compute the optimal sub-array for all possible three cases and take the maximum!



Maximum Subarray Problem

- The recursive Find-Max-SubArray($A, low, high$) finds the sub array of A in the range $[low, high]$ with maximum sum.

Maximum Subarray Problem

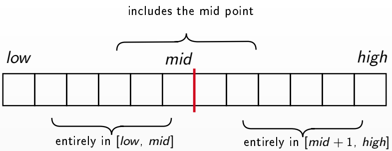
- The recursive Find-Max-SubArray($A, low, high$) finds the sub array of A in the range $[low, high]$ with maximum sum.
  - The output is (l, h, sum) and indicate the low-index, high-index, and the sum of the sub-array.

Maximum Subarray Problem

- The recursive Find-Max-SubArray($A, low, high$) finds the sub array of A in the range $[low, high]$ with maximum sum.
  - The output is (l, h, sum) and indicate the low-index, high-index, and the sum of the sub-array.
  - In the base, we have $low = high$, and the output is $(low, high, A[low])$.
- First, find the optimal solution that is entirely on the left:
  - recursively call Find-Max-SubArray(A, low, mid)

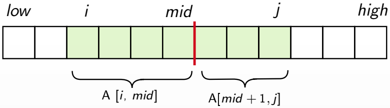
Maximum Subarray Problem

- The recursive Find-Max-SubArray($A, low, high$) finds the sub array of A in the range $[low, high]$ with maximum sum.
  - The output is (l, h, sum) and indicate the low-index, high-index, and the sum of the sub-array.
  - In the base, we have $low = high$, and the output is $(low, high, A[low])$.
- First, find the optimal solution that is entirely on the left:
  - recursively call Find-Max-SubArray(A, low, mid)
- Second, find the optimal solution that is entirely on the right:
  - recursively call Find-Max-SubArray($A, mid + 1, high$)



Maximum Subarray Problem

- Finally, find the sub-array with maximum sum, subject to it containing *mid*.
  - The subarray is made up of $A[i, mid]$ and $A[mid+1, j]$ for some $i \in [low, mid]$ and $j \in [mid+1, high]$.
  - Use a linear scan to find the values of i and j that give the sub-arrays with largest sums!
    - This can be done in linear time because one end (namely *mid*) of the subarrays is fixed.



Maximum Subarray Problem

- Finding the sub-array with maximum sum, subject to it containing *mid*.

```
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)
```

low		*i*			*mid*			*j*			*high*
3	-6	8	-1	-2	2	5	1	-2	-2	3	-1

i :

sum : **0**
leftSum : $-\infty$
maxLeft :

Maximum Subarray Problem

- Finding the sub-array with maximum sum, subject to it containing *mid*.

```
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)
```

low		*i*			*mid*			*j*			*high*
3	-6	8	-1	-2	2	5	1	-2	-2	3	-1

i : *mid*

sum : 2
leftSum : 2
maxLeft : *mid*

Maximum Subarray Problem

- Finding the sub-array with maximum sum, subject to it containing *mid*.

```
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)
```

low		*i*			*mid*			*j*			*high*
3	-6	8	-1	-2	2	5	1	-2	-2	3	-1

i : $mid - 1$

sum : 0

$leftSum$: 2

$maxLeft$: mid

Maximum Subarray Problem

- Finding the sub-array with maximum sum, subject to it containing *mid*.

```
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)
```

low		i			mid			j			high
3	-6	8	-1	-2	2	5	1	-2	-2	3	-1

i : $mid - 2$

sum : -1
$leftSum$: 2
$maxLeft$: mid

Maximum Subarray Problem

- Finding the sub-array with maximum sum, subject to it containing *mid*.

```
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)
```

low		*i*			*mid*			*j*			*high*
3	-6	8	-1	-2	2	5	1	-2	-2	3	-1

i : $mid - 3$

sum : **7**

$leftSum$: **7**

$maxLeft$: $mid - 3$

Maximum Subarray Problem

- Finding the sub-array with maximum sum, subject to it containing *mid*.

```
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)
```

low		*i*			*mid*			*j*			*high*
3	-6	8	-1	-2	2	5	1	-2	-2	3	-1

i : *mid* − **4**

sum : **1**
leftSum : **7**
maxLeft : *mid* − **3**

Maximum Subarray Problem

- Finding the sub-array with maximum sum, subject to it containing *mid*.

```
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)
```

low		*i*			*mid*			*j*			*high*
3	-6	8	-1	-2	2	5	1	-2	-2	3	-1

i : *mid* − **5**

sum : **4**
leftSum : **7**
maxLeft : *mid* − **3**

Maximum Subarray Problem

- Finding the sub-array with maximum sum, subject to it containing *mid*.

```
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)
```

low		*i*			*mid*			*j*			*high*
3	-6	8	-1	-2	2	5	1	-2	-2	3	-1

i : $mid - 5$

sum : 4
leftSum : 7
maxLeft : $mid - 3$

j :

sum : 0
rightSum : $-\infty$
maxRight :

Maximum Subarray Problem

- Finding the sub-array with maximum sum, subject to it containing *mid*.

```
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)
```

low		*i*			*mid*			*j*			*high*
3	-6	8	-1	-2	2	5	1	-2	-2	3	-1

i : $mid - 5$

sum : 4
$leftSum$: 7
$maxLeft$: $mid - 3$

j : $mid + 1$

sum : 5
$rightSum$: 5
$maxRight$: $mid + 1$

Maximum Subarray Problem

- Finding the sub-array with maximum sum, subject to it containing *mid*.

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)
```

low		*i*			*mid*			*j*			*high*
3	-6	8	-1	-2	2	5	1	-2	-2	3	-1

i : $mid - 5$

sum : 4
$leftSum$: 7
$maxLeft$: $mid - 3$

j : $mid + 2$

sum : 6
$rightSum$: 6
$maxRight$: $mid + 2$

Maximum Subarray Problem

- Finding the sub-array with maximum sum, subject to it containing *mid*.

```
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)
```

low		*i*			*mid*			*j*			*high*
3	-6	8	-1	-2	2	5	1	-2	-2	3	-1

i : $mid - 5$

sum : 4
$leftSum$: 7
$maxLeft$: $mid - 3$

j : $mid + 3$

sum : 4
$rightSum$: 6
$maxRight$: $mid + 2$

Maximum Subarray Problem

- Finding the sub-array with maximum sum, subject to it containing *mid*.

```
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)
```

low		*i*			*mid*			*j*			*high*
3	-6	8	-1	-2	2	5	1	-2	-2	3	-1

i : $mid - 5$

sum : 4
$leftSum$: 7
$maxLeft$: $mid - 3$

j : $mid + 4$

sum : 2
$rightSum$: 6
$maxRight$: $mid + 2$

Maximum Subarray Problem

- Finding the sub-array with maximum sum, subject to it containing *mid*.

```
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)
```

low		*i*			*mid*			*j*			*high*
3	-6	8	-1	-2	2	5	1	-2	-2	3	-1

i : $mid - 5$

sum : 4
$leftSum$: 7
$maxLeft$: $mid - 3$

j : $mid + 4$

sum : 5
$rightSum$: 6
$maxRight$: $mid + 2$

Maximum Subarray Problem

- Finding the sub-array with maximum sum, subject to it containing *mid*.

```
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)
```

low		*i*			*mid*			*j*			*high*
3	-6	8	-1	-2	2	5	1	-2	-2	3	-1

i : $mid - 5$

sum : 4
$leftSum$: 7
$maxLeft$: $mid - 3$

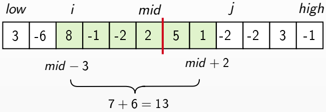
j : $mid + 5$

sum : 4
$rightSum$: 6
$maxRight$: $mid + 2$

Maximum Subarray Problem

- Finding the sub-array with maximum sum, subject to it containing *mid*.

```
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)
```



i : $mid - \mathbf{5}$

sum : **4**

$leftSum$: **7**

$maxLeft$: $mid - \mathbf{3}$

j : $mid + \mathbf{5}$

sum : **4**

$rightSum$: **6**

$maxRight$: $mid + \mathbf{2}$

Maximum Subarray Problem

- The recursive algorithm can be summarized as follows:

```
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)
```

Maximum Subarray Problem

- The recursive algorithm can be summarized as follows:

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

Maximum Subarray Problem

- For the running time of the recursive algorithm, we can run:

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

Maximum Subarray Problem

- For the running time of the recursive algorithm, we can run:

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

- The recurrence has the same form as that for MergeSort, and thus it has the same solution $T(n) = \Theta(n \log n)$.
- This algorithm is **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.

Divide & Conquer Paradigm

- The **divide and conquer paradigm** is important general technique for designing algorithms. In general, it follows the steps:
    - **Divide:** divide the problem into subproblems and recursively solve the subproblems
    - **Conquer:** combine solutions to subproblems to get solution to original problem

Divide & Conquer Paradigm

- The **divide and conquer paradigm** is important general technique for designing algorithms. In general, it follows the steps:
  - **Divide:** divide the problem into subproblems and recursively solve the subproblems
    - In merge sort, recursively sort two half-arrays on the left/right.
  - **Conquer:** combine solutions to subproblems to get solution to original problem
    - In merge sort, merge the two sorted half-arrays.

Divide & Conquer Paradigm

- The **divide and conquer paradigm** is important general technique for designing algorithms. In general, it follows the steps:
  - **Divide:** divide the problem into subproblems and recursively solve the subproblems
    - In merge sort, recursively sort two half-arrays on the left/right.
    - In maximum sub-array problem, recursively find the optimal sub-arrays that are entirely in the left/right half-arrays.
  - **Conquer:** combine solutions to subproblems to get solution to original problem
    - In merge sort, merge the two sorted half-arrays.
    - In maximum sub-array problem, find the optimal sub-array that crosses mid and take the best sub-array among three candidates.

Matrix Multiplication

- Consider two $n \times n$ matrices A and B.
- The matrix product $C = A \times B$ of two $n \times n$ matrices is defined as the $n \times n$ matrix that has the coefficient

$$c_{i,j} = \sum_{k=1}^{n} a_{i,k} \cdot b_{k,j}$$

$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} \\ a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} \end{bmatrix} \times \begin{bmatrix} b_{1,1} & b_{1,2} & b_{1,3} & b_{1,4} \\ b_{2,1} & b_{2,2} & b_{2,3} & b_{2,4} \\ b_{3,1} & b_{3,2} & b_{3,3} & b_{3,4} \\ b_{4,1} & b_{4,2} & b_{4,3} & b_{4,4} \end{bmatrix} = \begin{bmatrix} c_{1,1} & c_{1,2} & c_{1,3} & c_{1,4} \\ c_{2,1} & c_{2,2} & c_{2,3} & c_{2,4} \\ c_{3,1} & c_{3,2} & c_{3,3} & c_{3,4} \\ c_{4,1} & c_{4,2} & c_{4,3} & c_{4,4} \end{bmatrix}$$

A B C

Matrix Multiplication

- Consider two $n \times n$ matrices A and B.
- The matrix product $C = A \times B$ of two $n \times n$ matrices is defined as the $n \times n$ matrix that has the coefficient

$$c_{i,j} = \sum_{k=1}^{n} a_{i,k} \cdot b_{k,j}$$

$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} \\ a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} \end{bmatrix} \times \begin{bmatrix} b_{1,1} & b_{1,2} & b_{1,3} & b_{1,4} \\ b_{2,1} & b_{2,2} & b_{2,3} & b_{2,4} \\ b_{3,1} & b_{3,2} & b_{3,3} & b_{3,4} \\ b_{4,1} & b_{4,2} & b_{4,3} & b_{4,4} \end{bmatrix} = \begin{bmatrix} c_{1,1} & c_{1,2} & c_{1,3} & c_{1,4} \\ c_{2,1} & c_{2,2} & c_{2,3} & c_{2,4} \\ c_{3,1} & c_{3,2} & c_{3,3} & c_{3,4} \\ c_{4,1} & c_{4,2} & c_{4,3} & c_{4,4} \end{bmatrix}$$

$$c_{1,1} = a_{1,1}b_{1,1} + a_{1,2}b_{2,1} + a_{1,3}b_{3,1} + a_{1,4}b_{4,1}$$

Matrix Multiplication

- Consider two $n \times n$ matrices A and B.
- The matrix product $C = A \times B$ of two $n \times n$ matrices is defined as the $n \times n$ matrix that has the coefficient

$$c_{i,j} = \sum_{k=1}^{n} a_{i,k} \cdot b_{k,j}$$

$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} \\ a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} \end{bmatrix} \times \begin{bmatrix} b_{1,1} & b_{1,2} & b_{1,3} & b_{1,4} \\ b_{2,1} & b_{2,2} & b_{2,3} & b_{2,4} \\ b_{3,1} & b_{3,2} & b_{3,3} & b_{3,4} \\ b_{4,1} & b_{4,2} & b_{4,3} & b_{4,4} \end{bmatrix} = \begin{bmatrix} c_{1,1} & c_{1,2} & c_{1,3} & c_{1,4} \\ c_{2,1} & c_{2,2} & c_{2,3} & c_{2,4} \\ c_{3,1} & c_{3,2} & c_{3,3} & c_{3,4} \\ c_{4,1} & c_{4,2} & c_{4,3} & c_{4,4} \end{bmatrix}$$

$$c_{4,3} = a_{4,1}b_{1,3} + a_{4,2}b_{2,3} + a_{4,3}b_{3,3} + a_{4,4}b_{4,3}$$

Matrix Multiplication

- The straightforward algorithm takes $\Theta(n^3)$ time.

```
SQUARE-MATRIX-MULTIPLY(A, B)
n = A.rows
let C be a new n × n matrix
for i = 1 to n
    for j = 1 to n
        c_ij = 0
        for k = 1 to n
            c_ij = c_ij + a_ik · b_kj
return C
```

Matrix Multiplication

- The straightforward algorithm takes $\Theta(n^3)$ time.

```
SQUARE-MATRIX-MULTIPLY(A, B)
n = A.rows
let C be a new n × n matrix
for i = 1 to n
    for j = 1 to n
        c_ij = 0
        for k = 1 to n
            c_ij = c_ij + a_ik · b_kj
return C
```

- Can we design an algorithm with time $o(n^3)$?

Matrix Multiplication

- Partition each of A, B, and C into four $n/2 \times n/2$ matrices. We can write the product $A \times B = C$ as follows:

$$\begin{bmatrix} A_1 & A_2 \\ A_3 & A_4 \end{bmatrix} \times \begin{bmatrix} B_1 & B_2 \\ B_3 & B_4 \end{bmatrix} = \begin{bmatrix} A_1 \times B_1 + A_2 \times B_3 & A_1 \times B_2 + A_2 \times B_4 \\ A_3 \times B_1 + A_4 \times B_3 & A_3 \times B_2 + A_4 \times B_4 \end{bmatrix}$$

Matrix Multiplication

- Partition each of A, B, and C into four $n/2 \times n/2$ matrices. We can write the product $A \times B = C$ as follows:

$$\begin{bmatrix} A_1 & A_2 \\ A_3 & A_4 \end{bmatrix} \times \begin{bmatrix} B_1 & B_2 \\ B_3 & B_4 \end{bmatrix} = \begin{bmatrix} A_1 \times B_1 + A_2 \times B_3 & A_1 \times B_2 + A_2 \times B_4 \\ A_3 \times B_1 + A_4 \times B_3 & A_3 \times B_2 + A_4 \times B_4 \end{bmatrix}$$

$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} \\ a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} \end{bmatrix} \times \begin{bmatrix} b_{1,1} & b_{1,2} & b_{1,3} & b_{1,4} \\ b_{2,1} & b_{2,2} & b_{2,3} & b_{2,4} \\ b_{3,1} & b_{3,2} & b_{3,3} & b_{3,4} \\ b_{4,1} & b_{4,2} & b_{4,3} & b_{4,4} \end{bmatrix} = \begin{bmatrix} c_{1,1} & c_{1,2} & c_{1,3} & c_{1,4} \\ c_{2,1} & c_{2,2} & c_{2,3} & c_{2,4} \\ c_{3,1} & c_{3,2} & c_{3,3} & c_{3,4} \\ c_{4,1} & c_{4,2} & c_{4,3} & c_{4,4} \end{bmatrix}$$

A B C

Matrix Multiplication

- Partition each of A, B, and C into four $n/2 \times n/2$ matrices. We can write the product $A \times B = C$ as follows:

$$\begin{bmatrix} A_1 & A_2 \\ A_3 & A_4 \end{bmatrix} \times \begin{bmatrix} B_1 & B_2 \\ B_3 & B_4 \end{bmatrix} = \begin{bmatrix} A_1 \times B_1 + A_2 \times B_3 & A_1 \times B_2 + A_2 \times B_4 \\ A_3 \times B_1 + A_4 \times B_3 & A_3 \times B_2 + A_4 \times B_4 \end{bmatrix}$$

$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} \\ a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} \end{bmatrix} \times \begin{bmatrix} b_{1,1} & b_{1,2} & b_{1,3} & b_{1,4} \\ b_{2,1} & b_{2,2} & b_{2,3} & b_{2,4} \\ b_{3,1} & b_{3,2} & b_{3,3} & b_{3,4} \\ b_{4,1} & b_{4,2} & b_{4,3} & b_{4,4} \end{bmatrix} = \begin{bmatrix} c_{1,1} & c_{1,2} & c_{1,3} & c_{1,4} \\ c_{2,1} & c_{2,2} & c_{2,3} & c_{2,4} \\ c_{3,1} & c_{3,2} & c_{3,3} & c_{3,4} \\ c_{4,1} & c_{4,2} & c_{4,3} & c_{4,4} \end{bmatrix}$$

A B C

- How can we use this observation to design a D&Q algorithm?

Matrix Multiplication

- We have 8 smaller matrix multiplications and 4 additions.

$$\begin{bmatrix} A_1 & A_2 \\ A_3 & A_4 \end{bmatrix} \times \begin{bmatrix} B_1 & B_2 \\ B_3 & B_4 \end{bmatrix} = \begin{bmatrix} A_1 \times B_1 + A_2 \times B_3 & A_1 \times B_2 + A_2 \times B_4 \\ A_3 \times B_1 + A_4 \times B_3 & A_3 \times B_2 + A_4 \times B_4 \end{bmatrix}$$

```
SQUARE-MATRIX-MULTIPLY-RECURSIVE(A, B)
n = A.rows
let C be a new n × n matrix
if n == 1
    c11 = a11 · b11
else partition A, B, and C as in equations (4.9)
    C11 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A1 , B1 )
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A2 , B3 )
    C12 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A1 , B2)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A2 , B4 )
    C21 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A3 , B1 )
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A4 , B3 )
    C22 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A3 , B2)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A4 , B4 )
return C
```

Matrix Multiplication

- We have 8 smaller matrix multiplications and 4 additions.

$$\begin{bmatrix} A_1 & A_2 \\ A_3 & A_4 \end{bmatrix} \times \begin{bmatrix} B_1 & B_2 \\ B_3 & B_4 \end{bmatrix} = \begin{bmatrix} A_1 \times B_1 + A_2 \times B_3 & A_1 \times B_2 + A_2 \times B_4 \\ A_3 \times B_1 + A_4 \times B_3 & A_3 \times B_2 + A_4 \times B_4 \end{bmatrix}$$

```
SQUARE-MATRIX-MULTIPLY-RECURSIVE(A, B)
n = A.rows
let C be a new n × n matrix
if n == 1
    c11 = a11 · b11
else partition A, B, and C as in equations (4.9)
    C11 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A1, B1)
        + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A2, B3)
    C12 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A1, B2)
        + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A2, B4)
    C21 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A3, B1)
        + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A4, B3)
    C22 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A3, B2)
        + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A4, B4)
return C
```

- What is the running time of this algorithm?

Matrix Multiplication

- We have 8 smaller matrix multiplications and 4 additions.

$$\begin{bmatrix} A_1 & A_2 \\ A_3 & A_4 \end{bmatrix} \times \begin{bmatrix} B_1 & B_2 \\ B_3 & B_4 \end{bmatrix} = \begin{bmatrix} A_1 \times B_1 + A_2 \times B_3 & A_1 \times B_2 + A_2 \times B_4 \\ A_3 \times B_1 + A_4 \times B_3 & A_3 \times B_2 + A_4 \times B_4 \end{bmatrix}$$

- For the time complexity $T(n)$ we can write:

$$T(n) = \begin{cases} 8T(n/2) + \Theta(n^2) & \text{if} \quad n \geq 2 \\ c \quad \text{if} \quad n = 1 \end{cases}$$

Matrix Multiplication

- We have 8 smaller matrix multiplications and 4 additions.

$$\begin{bmatrix} A_1 & A_2 \\ A_3 & A_4 \end{bmatrix} \times \begin{bmatrix} B_1 & B_2 \\ B_3 & B_4 \end{bmatrix} = \begin{bmatrix} A_1 \times B_1 + A_2 \times B_3 & A_1 \times B_2 + A_2 \times B_4 \\ A_3 \times B_1 + A_4 \times B_3 & A_3 \times B_2 + A_4 \times B_4 \end{bmatrix}$$

- For the time complexity $T(n)$ we can write:

$$T(n) = \begin{cases} 8T(n/2) + \Theta(n^2) & \text{if} \quad n \geq 2 \\ c \quad \text{if} \quad n = 1 \end{cases}$$

- This is Case 1 of Master theorem; the time complexity is $n^{\log_2 8} = \Theta(n^3)$.

Matrix Multiplication

- We have 8 smaller matrix multiplications and 4 additions.

$$\begin{bmatrix} A_1 & A_2 \\ A_3 & A_4 \end{bmatrix} \times \begin{bmatrix} B_1 & B_2 \\ B_3 & B_4 \end{bmatrix} = \begin{bmatrix} A_1 \times B_1 + A_2 \times B_3 & A_1 \times B_2 + A_2 \times B_4 \\ A_3 \times B_1 + A_4 \times B_3 & A_3 \times B_2 + A_4 \times B_4 \end{bmatrix}$$

- For the time complexity $T(n)$ we can write:

$$T(n) = \begin{cases} 8T(n/2) + \Theta(n^2) & \text{if} \quad n \geq 2 \\ c \quad \text{if} \quad n = 1 \end{cases}$$

- This is Case 1 of Master theorem; the time complexity is $n^{\log_2 8} = \Theta(n^3)$.
- How can we improve this? **Strassen's Algorithm**

Strassen's Algorithm

- To get $A \times B$, it suffices to find C_1, C_2, C_3, and C_4

$$\begin{bmatrix} A_1 & A_2 \\ A_3 & A_4 \end{bmatrix} \times \begin{bmatrix} B_1 & B_2 \\ B_3 & B_4 \end{bmatrix} = \begin{bmatrix} \overbrace{A_1 \times B_1 + A_2 \times B_3}^{C_1} & \overbrace{A_1 \times B_2 + A_2 \times B_4}^{C_2} \\ \underbrace{A_3 \times B_1 + A_4 \times B_3}_{C_3} & \underbrace{A_3 \times B_2 + A_4 \times B_4}_{C_4} \end{bmatrix}$$

Strassen's Algorithm

- To get $A \times B$, it suffices to find C_1, C_2, C_3, and C_4

$$\begin{bmatrix} A_1 & A_2 \\ A_3 & A_4 \end{bmatrix} \times \begin{bmatrix} B_1 & B_2 \\ B_3 & B_4 \end{bmatrix} = \begin{bmatrix} \overset{C_1}{A_1 \times B_1 + A_2 \times B_3} & \overset{C_2}{A_1 \times B_2 + A_2 \times B_4} \\ \underset{C_3}{A_3 \times B_1 + A_4 \times B_3} & \underset{C_4}{A_3 \times B_2 + A_4 \times B_4} \end{bmatrix}$$

- **Divide:** compute the following seven $n/2 \times n/2$ matrices by calling the multiplication function recursively **seven times**.

$$P_1 = A_1 \times (B_2 - B_4)$$
$$P_2 = (A_1 + A_2) \times B_4$$
$$P_3 = (A_3 + A_4) \times B_1$$
$$P_4 = A_4 \times (B_3 - B_1)$$
$$P_5 = (A_1 + A_4) \times (B_1 + B_4)$$
$$P_6 = (A_2 - A_4) \times (B_3 + B_4)$$
$$P_7 = (A_1 - A_3) \times (B_1 + B_2)$$

Strassen's Algorithm

- **Conquer:** Use matrices P_i to compute C_1, C_2, C_3, and C_4.

$$\begin{bmatrix} A_1 & A_2 \\ A_3 & A_4 \end{bmatrix} \times \begin{bmatrix} B_1 & B_2 \\ B_3 & B_4 \end{bmatrix} = \begin{bmatrix} \overbrace{A_1 \times B_1 + A_2 \times B_3}^{C_1} & \overbrace{A_1 \times B_2 + A_2 \times B_4}^{C_2} \\ \underbrace{A_3 \times B_1 + A_4 \times B_3}_{C_3} & \underbrace{A_3 \times B_2 + A_4 \times B_4}_{C_4} \end{bmatrix}$$

$$\begin{aligned} P_1 &= A_1 \times (B_2 - B_4) \\ P_2 &= (A_1 + A_2) \times B_4 \\ P_3 &= (A_3 + A_4) \times B_1 \\ P_4 &= A_4 \times (B_3 - B_1) \\ P_5 &= (A_1 + A_4) \times (B_1 + B_4) \\ P_6 &= (A_2 - A_4) \times (B_3 + B_4) \\ P_7 &= (A_1 - A_3) \times (B_1 + B_2) \end{aligned}$$

Strassen's Algorithm

- **Conquer:** Use matrices P_i to compute C_1, C_2, C_3, and C_4.

$$\begin{bmatrix} A_1 & A_2 \\ A_3 & A_4 \end{bmatrix} \times \begin{bmatrix} B_1 & B_2 \\ B_3 & B_4 \end{bmatrix} = \begin{bmatrix} \overbrace{A_1 \times B_1 + A_2 \times B_3}^{C_1} & \overbrace{A_1 \times B_2 + A_2 \times B_4}^{C_2} \\ \underbrace{A_3 \times B_1 + A_4 \times B_3}_{C_3} & \underbrace{A_3 \times B_2 + A_4 \times B_4}_{C_4} \end{bmatrix}$$

$$\begin{aligned} P_1 &= A_1 \times (B_2 - B_4) \\ P_2 &= (A_1 + A_2) \times B_4 \\ P_3 &= (A_3 + A_4) \times B_1 \\ P_4 &= A_4 \times (B_3 - B_1) \\ P_5 &= (A_1 + A_4) \times (B_1 + B_4) \\ P_6 &= (A_2 - A_4) \times (B_3 + B_4) \\ P_7 &= (A_1 - A_3) \times (B_1 + B_2) \end{aligned}$$

$$\begin{aligned} C_1 &= P_5 + P_4 - P_2 + P_6 \\ &= (A_1 B_1 + A_1 B_4 + A_4 B_1 + A_4 B_4) + \\ &\quad (A_4 B_3 - A_4 B_1) + \\ &\quad (-A_1 B_4 - A_2 B_4) + \\ &\quad (A_2 B_3 + A_2 B_4 - A_4 B_3 - A_4 B_4) \\ &= A_1 B_1 + A_2 B_3 \end{aligned}$$

Strassen's Algorithm

- **Conquer:** Use matrices P_i to compute C_1, C_2, C_3, and C_4.

$$\begin{bmatrix} A_1 & A_2 \\ A_3 & A_4 \end{bmatrix} \times \begin{bmatrix} B_1 & B_2 \\ B_3 & B_4 \end{bmatrix} = \begin{bmatrix} \overset{C_1}{A_1 \times B_1 + A_2 \times B_3} & \overset{C_2}{A_1 \times B_2 + A_2 \times B_4} \\ \underset{C_3}{A_3 \times B_1 + A_4 \times B_3} & \underset{C_4}{A_3 \times B_2 + A_4 \times B_4} \end{bmatrix}$$

$$\begin{aligned}
P_1 &= A_1 \times (B_2 - B_4) \\
P_2 &= (A_1 + A_2) \times B_4 \\
P_3 &= (A_3 + A_4) \times B_1 \\
P_4 &= A_4 \times (B_3 - B_1) \\
P_5 &= (A_1 + A_4) \times (B_1 + B_4) \\
P_6 &= (A_2 - A_4) \times (B_3 + B_4) \\
P_7 &= (A_1 - A_3) \times (B_1 + B_2)
\end{aligned}$$

$$\begin{aligned}
C_1 &= P_5 + P_4 - P_2 + P_6 \\
&= (A_1B_1 + A_1B_4 + A_4B_1 + A_4B_4) + \\
&\quad (A_4B_3 - A_4B_1) + \\
&\quad (-A_1B_4 - A_2B_4) + \\
&\quad (A_2B_3 + A_2B_4 - A_4B_3 - A_4B_4) \\
&= A_1B_1 + A_2B_3
\end{aligned}$$

$$\begin{aligned}
C_2 &= P_1 + P_2 \\
&= (A_1B_2 - A_1B_4) + \\
&\quad (A_1B_4 + A_2B_4) \\
&= A_1B_2 + A_2B_4
\end{aligned}$$

Strassen's Algorithm

- **Conquer:** Use matrices P_i to compute C_1, C_2, C_3, and C_4.

$$\begin{bmatrix} A_1 & A_2 \\ A_3 & A_4 \end{bmatrix} \times \begin{bmatrix} B_1 & B_2 \\ B_3 & B_4 \end{bmatrix} = \begin{bmatrix} \overbrace{A_1 \times B_1 + A_2 \times B_3}^{C_1} & \overbrace{A_1 \times B_2 + A_2 \times B_4}^{C_2} \\ \underbrace{A_3 \times B_1 + A_4 \times B_3}_{C_3} & \underbrace{A_3 \times B_2 + A_4 \times B_4}_{C_4} \end{bmatrix}$$

$$\begin{aligned}
P_1 &= A_1 \times (B_2 - B_4) \\
P_2 &= (A_1 + A_2) \times B_4 \\
P_3 &= (A_3 + A_4) \times B_1 \\
P_4 &= A_4 \times (B_3 - B_1) \\
P_5 &= (A_1 + A_4) \times (B_1 + B_4) \\
P_6 &= (A_2 - A_4) \times (B_3 + B_4) \\
P_7 &= (A_1 - A_3) \times (B_1 + B_2)
\end{aligned}$$

$$\begin{aligned}
C_1 &= P_5 + P_4 - P_2 + P_6 \\
&= (A_1B_1 + A_1B_4 + A_4B_1 + A_4B_4) + \\
&\quad (A_4B_3 - A_4B_1) + \\
&\quad (-A_1B_4 - A_2B_4) + \\
&\quad (A_2B_3 + A_2B_4 - A_4B_3 - A_4B_4) \\
&= A_1B_1 + A_2B_3
\end{aligned}$$

$$\begin{aligned}
C_2 &= P_1 + P_2 \\
&= (A_1B_2 - A_1B_4) + \\
&\quad (A_1B_4 + A_2B_4) \\
&= A_1B_2 + A_2B_4
\end{aligned}$$

$$\begin{aligned}
C_3 &= P_3 + P_4 \\
&= (A_3B_1 + A_4B_1) + \\
&\quad (A_4B_3 - A_4B_1) \\
&= (A_3B_1 + A_4B_3)
\end{aligned}$$

Strassen's Algorithm

- **Conquer:** Use matrices P_i to compute C_1, C_2, C_3, and C_4.

$$\begin{bmatrix} A_1 & A_2 \\ A_3 & A_4 \end{bmatrix} \times \begin{bmatrix} B_1 & B_2 \\ B_3 & B_4 \end{bmatrix} = \begin{bmatrix} \overbrace{A_1 \times B_1 + A_2 \times B_3}^{C_1} & \overbrace{A_1 \times B_2 + A_2 \times B_4}^{C_2} \\ \underbrace{A_3 \times B_1 + A_4 \times B_3}_{C_3} & \underbrace{A_3 \times B_2 + A_4 \times B_4}_{C_4} \end{bmatrix}$$

$$\begin{aligned}
P_1 &= A_1 \times (B_2 - B_4) \\
P_2 &= (A_1 + A_2) \times B_4 \\
P_3 &= (A_3 + A_4) \times B_1 \\
P_4 &= A_4 \times (B_3 - B_1) \\
P_5 &= (A_1 + A_4) \times (B_1 + B_4) \\
P_6 &= (A_2 - A_4) \times (B_3 + B_4) \\
P_7 &= (A_1 - A_3) \times (B_1 + B_2)
\end{aligned}$$

$$\begin{aligned}
C_1 &= P_5 + P_4 - P_2 + P_6 \\
&= (A_1 B_1 + A_1 B_4 + A_4 B_1 + A_4 B_4) + \\
&\quad (A_4 B_3 - A_4 B_1) + \\
&\quad (-A_1 B_4 - A_2 B_4) + \\
&\quad (A_2 B_3 + A_2 B_4 - A_4 B_3 - A_4 B_4) \\
&= A_1 B_1 + A_2 B_3
\end{aligned}$$

$$\begin{aligned}
C_2 &= P_1 + P_2 \\
&= (A_1 B_2 - A_1 B_4) + \\
&\quad (A_1 B_4 + A_2 B_4) \\
&= A_1 B_2 + A_2 B_4
\end{aligned}$$

$$\begin{aligned}
C_3 &= P_3 + P_4 \\
&= (A_3 B_1 + A_4 B_1) + \\
&\quad (A_4 B_3 - A_4 B_1) \\
&= (A_3 B_1 + A_4 B_3)
\end{aligned}$$

$$\begin{aligned}
C_4 &= P_5 + P_1 - P_3 - P_7 \\
&= (A_1 B_1 + A_1 B_4 + A_4 B_1 + A_4 B_4) + \\
&\quad (A_1 B_2 - A_1 B_4) + \\
&\quad (-A_3 B_1 - A_4 B_1) + \\
&\quad (-A_1 B_1 - A_1 B_2 + A_3 B_1 + A_3 B_2) \\
&= A_3 B_2 + A_4 B_4
\end{aligned}$$

Strassen's Algorithm Summary

- We make 7 recursive calls to multiply matrices of size $n/2 \times n/2$.
  - The additional work involves adding/subtracting matrices of size $n/2 \times n/2$ several times; this takes $\Theta(n^2)$.

$$\begin{bmatrix} A_1 & A_2 \\ A_3 & A_4 \end{bmatrix} \times \begin{bmatrix} B_1 & B_2 \\ B_3 & B_4 \end{bmatrix} = \begin{bmatrix} \overset{C_1}{A_1 \times B_1 + A_2 \times B_3} & \overset{C_2}{A_1 \times B_2 + A_2 \times B_4} \\ \underset{C_3}{A_3 \times B_1 + A_4 \times B_3} & \underset{C_4}{A_3 \times B_2 + A_4 \times B_4} \end{bmatrix}$$

$$\begin{aligned} P_1 &= A_1 \times (B_2 - B_4) \\ P_2 &= (A_1 + A_2) \times B_4 \\ P_3 &= (A_3 + A_4) \times B_1 \\ P_4 &= A_4 \times (B_3 - B_1) \\ P_5 &= (A_1 + A_4) \times (B_1 + B_4) \\ P_6 &= (A_2 - A_4) \times (B_3 + B_4) \\ P_7 &= (A_1 - A_3) \times (B_1 + B_2) \end{aligned}$$

$$\begin{aligned} C_1 &= P_5 + P_4 - P_2 + P_6 \\ C_2 &= P_1 + P_2 \\ C_3 &= P_3 + P_4 \\ C_4 &= P_5 + P_1 - P_3 - P_7 \end{aligned}$$

The time complexity of the Strassen's algorithm is: $T(n) = \begin{cases} 7T(n/2) + \Theta(n^2) & \text{if } n \geq 2 \\ c \quad \text{if} \quad n = 2 \end{cases}$

This is case 1 of Master theorem, and $T(n) = \Theta(n^{\log_2 7})$

Matrix Multiplication Summary

- A naive iterative algorithm runs in $\Theta(n^3)$.

Matrix Multiplication Summary

- A naive iterative algorithm runs in $\Theta(n^3)$.
- A simple D&Q does not improve the running time (it stays $\Theta(n^3)$).

Matrix Multiplication Summary

- A naive iterative algorithm runs in $\Theta(n^3)$.
- A simple D&Q does not improve the running time (it stays $\Theta(n^3)$).
- Strassen algorithm is a D&Q algorithm with improved running time of $\Theta(n^{\log_2 7})$.

Matrix Multiplication Summary

- A naive iterative algorithm runs in $\Theta(n^3)$.
- A simple D&Q does not improve the running time (it stays $\Theta(n^3)$).
- Strassen algorithm is a D&Q algorithm with improved running time of $\Theta(n^{\log_2 7})$.
- The best existing algorithm has running time $O(n^{2.373})$ [Alman 2020]
  - We know we cannot do better than $\Omega(n^2)$ (why?)
  - Finding the best running time is still an open problem!