Algorithms Analysis and Design

Chapter 4

Divide and Conquer
Part 3

Divide-and-Conquer Examples

- **Sorting**:
  - merge sort
  - quicksort
- **Binary tree traversals**
- **Mathematics**
  - Multiplication of large integers
  - Matrix multiplication: Strassen's algorithm
  - Exponentiation
- **Computational geometry**
  - Closest-pair
  - convex-hull algorithms
- **Searching**:
  - Binary search: decrease-by-half (or degenerate divide&conq.)

Matrix Multiplication

- If A is an $m \times n$ matrix and B is an $n \times p$ matrix

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{pmatrix}$$

- The *matrix product* **C** = **AB** is defined to be the $m \times p$ matrix

$$\mathbf{C} = \begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1p} \\ c_{21} & c_{22} & \cdots & c_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \cdots & c_{mp} \end{pmatrix}$$

- Such that:

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj} = \sum_{k=1}^{n} a_{ik}b_{kj}.$$ for i = 1 …. m and j = 1 …… p

That is C_{ij} is the dot product of the i^{th} row of A and the j^{th} column of B.

Example

"Dot Product"

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{bmatrix} = \begin{bmatrix} 58 & \\ & \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{bmatrix} = \begin{bmatrix} 58 & 64 \\ & \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{bmatrix} = \begin{bmatrix} 58 & 64 \\ 139 & 154 \end{bmatrix} ✓$$

Matrix Multiplication

You should read Chapter 2 to recall the **brute force algorithm** for matrix multiplications.

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

We must compute n^2 matrix entries, and each is the sum of n values.

Because each of the triply-nested **for** loops runs exactly n iterations, and each execution of line 7 takes constant time, the SQUARE-MATRIX-MULTIPLY procedure takes $\boldsymbol{\theta(n^3)}$ time.

Divide and Conquer for matrix multiplication

Divide and conquer strategy can be applied for multiplying matrices

- A simple divide and conquer algorithm
- **Strassen's method** (Enhanced DAC version)

Simple DAC algorithm

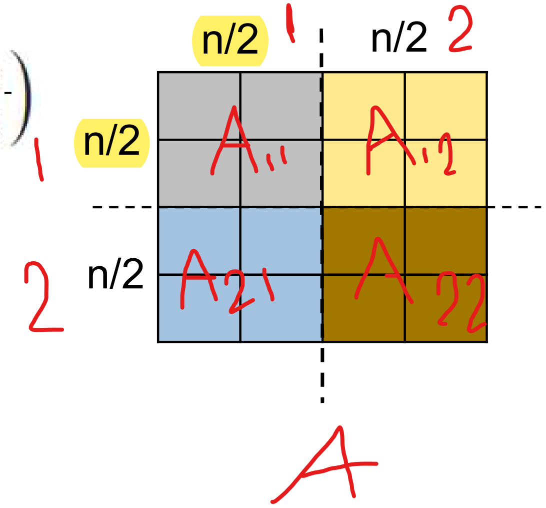
Suppose we want to design a divide and conquer algorithm multiply two matrices A and B, each of size $n\ x\ n$ To keep things simple, we assume that n is an exact power of 2 in each of the $n \times n$ matrices.

- Bases case: multiply two matrices, each containing *one element* (**n = 1**). $C = A_{11} \cdot B_{11}$
- Divide: in each divide step, we will divide $n \times n$ matrices into four $n/2 \times n/2$ matrices

نتج عندي 4 sub arrays

2D array

$$A = \left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right), \quad B = \left(\begin{array}{c|c} B_{11} & B_{12} \\ \hline B_{21} & B_{22} \end{array}\right), \quad C = \left(\begin{array}{c|c} C_{11} & C_{12} \\ \hline C_{21} & C_{22} \end{array}\right)$$



so that we rewrite the equation C = A . B as

نتجت من ضرب الصف الأول
بالعمود الاول

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

Divide and Conquer for matrix multiplication

- so that we rewrite the equation C = A . B as

بعد ال A

division

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

نتج عن هاي العملية

ضرب 8 مصفوفات

- The above equation corresponds to four equations:

matrix multiplication قانون ال

$$\begin{cases} C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21} \\ C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22} \\ C_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21} \\ C_{22} = A_{21} \cdot B_{12} + A_{22} \cdot B_{22} \end{cases}$$

بنحل مسألة الضرب ريكيرسيفلي 8 مرات

- Each of these four equations specifies two multiplications of $n/2 \times n/2$ matrices and the addition of their $n/2 \times n/2$ products.

> This approach requires
> - 8 multiplications to be solved recursively (Conquer)
> - 4 additions (combine resulting solution from the recursive call)

لأني بستدعي الفنكشن 8 مرات مشان يحل ال 8 المسائل الصغيرة

$$T(n) = 8T\left(\frac{n}{2}\right) + O(n^2) + O(1)$$

لأني قسمت من النص

Pseudocode of Merge sort

8 مرات

استدعاء الفنكشن لضرب مصفوفتين

ال base case اذا بدي أضرب مصفوفتين كل وحده فيها عنصر ا

```
SQUARE-MATRIX-MULTIPLY-RECURSIVE(A, B)
1  n = A.rows
2  let C be a new n × n matrix
3  if n == 1
4      c11 = a11 · b11
5  else partition A, B, and C as in equations
6      C11 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B11)
           + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B21)
7      C12 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B12)
           + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B22)
8      C21 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B11)
           + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B21)
9      C22 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B12)
           + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B22)
10 return C
```

أقل اشي لازم يكون 4elements

we can partition the matrices by **index calculations**. We identify a submatrix by a range of row indices and a range of column indices of the original matrix.

Executing line 5 takes only $O(1)$ time

Analysis of DAC algorithm

- Let T(n) be the time to multiply two $n \times n$ matrices
- In the base case, when n = 1, we perform just the one scalar multiplication T(1) = O(1)
- The recursive case occurs when n > 1
  - The cost of partitioning the matrices using index calculations is ***O*(1)** time.
  - We recursively call the procedure a total of 8 times, each recursive call multiplies two $n/2 \times n/2$ matrices. The time taken by all eight recursive calls is **8T(n/2)**
  - We also must account for the four matrix additions. Each of these matrices contains $n^2/4$ entries, and so each of the four matrix additions takes ***O*(n²)** time.

The total time for the recursive case, therefore, is the sum of the **partitioning time**, **the time for all the recursive calls**, and the **time to add the matrices** resulting from the recursive calls

Analysis of DAC algorithm

The total time for the recursive case, therefore, is the sum of the **partitioning time**, **the time for all the recursive calls**, and the **time to add the matrices** resulting from the recursive calls

$$T(n) = 8T(n/2) + O(1) + O(n^2) \qquad \text{if } n > 1$$

a = 8 , b = 2 , d = 2

Since $a > b^d \rightarrow \quad T(n) \in O(n^{\log_b a})$

$T(n) \in O(n^{\log_2 8})$

$T(n) \in O(n^3)$

How does this algorithm compare with the brute-force algorithm for this problem??

This simple divide-and-conquer approach is **no faster** than the straightforward brute-force approach

Matrix Multiplication using Strassen's Method

- **Strassen** suggested a <u>divide and conquer strategy</u>-based **matrix multiplication** technique that requires fewer multiplications than the traditional method.
- The key of Strassen's method is to **perform only seven recursive multiplications** of $n/2 \times n/2$ matrices instead of eight.

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

Matrix Multiplication using Strassen’s Method

- **Strassen’s method has four main steps:**

- **Step 1**: Divide the input matrices A and B and output matrix C into $n/2 \times n/2$ submatrices by index calculation .

- **Step 2**: Create 10 matrices S1, S2, …, S10, each of which is $\frac{n}{2} \times \frac{n}{2}$

$$
\begin{aligned}
S_1 &= B_{12} - B_{22} \\
S_2 &= A_{11} + A_{12} \\
S_3 &= A_{21} + A_{22} \\
S_4 &= B_{21} - B_{11} \\
S_5 &= A_{11} + A_{22} \\
S_6 &= B_{11} + B_{22} \\
S_7 &= A_{12} - A_{22} \\
S_8 &= B_{21} + B_{22} \\
S_9 &= A_{11} - A_{21} \\
S_{10} &= B_{11} + B_{12}
\end{aligned}
$$

Matrix Multiplication using Strassen's Method

- **Strassen's method has four main steps:**

- **Step3**: Using the submatrices created in step 1 and the 10 matrices created in step 2, recursively compute **seven matrix products** P1, P2, …… , P7. Each matrix P_i is $n/2 \times n/2$.

$$
\begin{aligned}
P_1 &= A_{11} \cdot S_1 &&= A_{11} \cdot B_{12} - A_{11} \cdot B_{22} \\
P_2 &= S_2 \cdot B_{22} &&= A_{11} \cdot B_{22} + A_{12} \cdot B_{22} \\
P_3 &= S_3 \cdot B_{11} &&= A_{21} \cdot B_{11} + A_{22} \cdot B_{11} \\
P_4 &= A_{22} \cdot S_4 &&= A_{22} \cdot B_{21} - A_{22} \cdot B_{11} \\
P_5 &= S_5 \cdot S_6 &&= A_{11} \cdot B_{11} + A_{11} \cdot B_{22} + A_{22} \cdot B_{11} + A_{22} \cdot B_{22} \\
P_6 &= S_7 \cdot S_8 &&= A_{12} \cdot B_{21} + A_{12} \cdot B_{22} - A_{22} \cdot B_{21} - A_{22} \cdot B_{22} \\
P_7 &= S_9 \cdot S_{10} &&= A_{11} \cdot B_{11} + A_{11} \cdot B_{12} - A_{21} \cdot B_{11} - A_{21} \cdot B_{12}
\end{aligned}
$$

- **Step4**: Compute the desired submatrices C_{11} , C_{12} , C_{21} , C_{22} of the result matrix C by adding and subtracting various combinations of the Pi matrices

$$
\begin{pmatrix}
C_{11} = P_5 + P_4 - P_2 + P_6 & C_{12} = P_1 + P_2 \\
C_{21} = P_3 + P_4 & C_{22} = P_5 + P_1 - P_3 - P_7
\end{pmatrix}
$$

Analysis of Strassen's Algorithm

- **To evaluate the asymptotic efficiency of Strassen's algorithm**
  - The cost of partitioning the matrices using index calculations is $O(1)$ time.
  - The number of recursive multiplications = 7. The time taken by all seven recursive calls is **7 T(n/2)**
  - Number of additions/subtractions: **18 additions/subtractions** of matrices of size $n/2$ takes $O(n^2)$ time.

$$\mathbf{T(n) = 7T(n/2) + 18\, n^2/4 \qquad \text{if } n > 1}$$

$$\mathbf{T(n) = 7T(n/2) + O(1) + O(n^2) \qquad \text{if } n > 1}$$

a = 7 , b = 2 , d = 2

Since a > b^d $\rightarrow$ $T(n) \in O(n^{\log_b a})$

$$T(n) \in O(n^{\log_2 7})$$

$$T(n) \approx O(n^{2.807})$$

How does this algorithm compare with the simple DAC algorithm??

Strassen's method is **asymptotically faster** than the straightforward brute-force approach and the simple divide-and-conquer approach

Notes

- The efficiency class of Strassen's algorithm $\theta(n^{\log_2 7})$ is a better efficiency class than $\theta(n^3)$ of the brute-force and simple DAC methods.
- several other algorithms for multiplying two $n \times n$ matrices of real numbers in $\boldsymbol{O}(\boldsymbol{n}^{\boldsymbol{\alpha}})$ time with progressively smaller constants $\boldsymbol{\alpha}$ have been invented.
- The fastest algorithm so far is that of Coopersmith and Winograd with its efficiency in $O(n^{2.376})$

Divide-and-Conquer Examples

- **Sorting**:
  - merge sort
  - quicksort
- **Binary tree traversals**
- **Mathematics**
  - Multiplication of large integers
  - Matrix multiplication: Strassen's algorithm
  - Exponentiation problem
- **Computational geometry**
  - Closest-pair
  - convex-hull algorithms
- **Searching**:
  - Binary search: decrease-by-half (or degenerate divide&conq.)

Exponentiation problem

- ❑ The divide-and-conquer technique can be successfully applied to handle the exponentiation problem

- ❑ Recall that we solved the same problem in linear time [$\theta\ (n)$] using two Conventional methods:
  - Brute force
  - Simple recursive algorithm / Decrease and conquer [**unbalanced** partitioning]

Exponentiation problem

- **Problem**: exponentiation problem [Computing a^n (a > 0, n a nonnegative integer)]

$$a^n = a * \underbrace{a * a \ldots\ldots\ldots *}_{\text{n times}} a$$

- Time complexity: $\theta\ (n)$

- **Can we find a better (faster) algorithm?**

Naive algorithm

```
ALGORITHM pow ( a , n )
{
    result = 1
    for i = 1 to n
        result = result * a
    return result
}
```

Decrease and conquer for solving a^n

$$a^n = a * \underbrace{\boxed{a * a * a \ldots\ldots\ldots * a}}_{n-1}$$

$$a^n = a * a^{n-1}$$

1. Decrease the problem by **1**, and

   solve the smaller instance of size $n-1$ recursively.

1. Derive a recurrence relation and solve it.
2. Find the time complexity.
3. Is it a good algorithm for solving this problem?

Recursive algorithm

```
ALGORITHM Rec_pow ( a , n )
{



}
```

Divide and conquer approach for solving a^n

a^8 = a * a * a * a * a * a * a * a

= $a^4 \cdot a^4$

$a^n = a^{n/2} \cdot a^{n/2}$

n is even

a^9 = a * a^8

= a * $a^4 \cdot a^4$

a^n = a * $a^{(n-1)/2} \cdot a^{(n-1)/2}$

n is odd

1. Divide and conquer approach is achieved by creating sub problems of size n/2 [**Balanced partitioning**]
2. Each subproblem is solved recursively until n = 1 [**base case**]
3. The recursive solution:

$$a^n = \begin{cases} a^{n/2} \cdot a^{n/2} & \text{if } n \text{ is even} \\ a * a^{(n-1)/2} \cdot a^{(n-1)/2} & \text{if } n \text{ is odd} \end{cases}$$

Pseudocode of DAC approach

ALGORITHM $Pow\_DAC\ (a, n)$

{

$\boldsymbol{if}\ (n = 1)\ \boldsymbol{then}$

$return\ a$

else

$\boldsymbol{return}\ Pow\_DAC(a, n/2) * Pow\_DAC(a, n/2)$

}

- **Note**: This algorithm does not check whether n is even or odd

- **Overlapping problem: same subproblem solved two times**

- Complexity analysis:

  T(n) = 2 T(n/2) + C n > 1

  using master theorem, T(n) ∈ O(n)

How does this algorithm compare with the conventional approach??

Improved DAC approach

ALGORITHM $Pow\_DAC\ (a, n)$

{

$\boldsymbol{if}\ (n = 1)\ \boldsymbol{then}$

$return\ a$

else

$\mathbf{y} = Pow\_DAC(a, n/2)$

$\boldsymbol{return}\ y * y$

}

- **To handle overlapping problem:**

Solve one subproblem and store its solution.

Multiply the partial solution by itself

This approach can be considered decrease by a half technique.

- Complexity analysis:

  T(n) = T(n/2) + C   n > 1

  using master theorem, T(n) ∈ **O(log n)**

This approach is asymptotically faster than previous methods.

Improved DAC approach

- **Rules to have an efficient Divide and Conquer method:**

1. Balanced partitioning.

2. Subproblems are independent (no overlapping)

Exercise

Modify the following pseudocode to handle both even and odd values of n

ALGORITHM $Pow\_DAC\ (\ a\ , n)$

{

$\boldsymbol{if}\ (n = 1)\ \boldsymbol{then}$

$return\ a$

else

$\boldsymbol{return}\ Pow\_DAC(a\ , n/2) * Pow\_DAC(a\ , n/2)$

}

$$a^n = \begin{cases} a^{n/2} \,.\, a^{n/2} & if\ n\ is\ even \\ a * a^{(n-1)/2} \,.\, a^{(n-1)/2} & if\ n\ is\ odd \end{cases}$$

Divide-and-Conquer Examples

- **Sorting**:
  - merge sort
  - quicksort
- **Binary tree traversals**
- **Mathematics**
  - Multiplication of large integers
  - Matrix multiplication: Strassen's algorithm
  - Exponentiation problem
- **Computational geometry**
  - Closest-pair
  - convex-hull algorithms
- **Searching**:
  - Binary search: decrease-by-half (or degenerate divide&conq.)

Multiplication of large integers

- Why it is essential to investigate efficient algorithms for efficient multiplication of large integers:
  - Some applications, notably modern **cryptography**, require manipulation of integers that are over 100 decimal digits long.
  - Such integers are too long to fit in a single word of a modern computer.
- Problem:

  Consider the problem of multiplying two (large) n-digit integers represented by arrays of their digits such as:

A = 12345678901357986429 B = 87654321284820912836

The grade-school algorithm:

$$
\begin{array}{r}
a_1\ a_2 \ldots\ a_n \\
\underline{b_1\ b_2 \ldots\ b_n} \\
(d_{10})\ d_{11} d_{12} \ldots\ d_{1n} \\
(d_{20})\ d_{21} d_{22} \ldots\ d_{2n} \\
\ldots\ \ldots\ \ldots\ \ldots\ \ldots\ \ldots\ \ldots \\
(d_{n0})\ d_{n1} d_{n2} \ldots\ d_{nn}
\end{array}
$$

Efficiency: $O(n^2)$ one-digit multiplications

First Divide-and-Conquer Algorithm

An example of two-digit integers: A = 23 B = 14

$A = (2 \,.\, 10^1 + 3 \,.\, 10^0\ )$ $B = (1 \,.\, 10^1 + 4 \,.\, 10^0)$

$A = (2 \,.\, 10^1 + 3)$ $B = (1 \,.\, 10^1 + 4)$

$A * B = (2 \,.\, 10^1 + 3) * (1 \,.\, 10^1 + 4)$

$= (2 * 1) \cdot 10^2 + (2 * 4 + 3 * 1)\ 10^1 + 3 * 4$

An example of four-digit integers: A = 2135 and B = 4014

$A = (21 \cdot 10^2 + 35)$ $B = (40 \cdot 10^2 + 14)$

$A * B = (21 \cdot 10^2 + 35) * (40 \cdot 10^2 + 14)$

$= 21 * 40 \cdot 10^4 + (21 * 14 + 35 * 40) \cdot 10^2 + 35 * 14$

First Divide-and-Conquer Algorithm

- In general, if $A = A_1A_2$ and $B = B_1B_2$ (where A and B are n-digit, A_1, A_2, B_1, B_2 are $n/2$-digit numbers)

$$A * B = (A_1 * B_1)\cdot 10^{n} + (A_1 * B_2 + A_2 * B_1)\cdot 10^{n/2} + (A_2 * B_2)$$

- **Analysis** of the first divide-and-conquer algorithm
  - Divide: The cost of partitioning $\boldsymbol{O(1)}$ time.
  - Conquer: The number of recursive multiplications = 4. The time taken by all four recursive calls is **4 T(n/2)**
  - Combine: Additions $\boldsymbol{O(n)}$

Recurrence for the number of one-digit multiplications T(n):

$$T(n) = 4T(n/2) + cn, \quad T(1) = 1$$

a = 4 , b = 2 , d = 1 (Use master theorem)

$T(n) \in O(n^2)$

How does this algorithm compare with the conventional approach??

Improved Divide-and-Conquer Algorithm

The idea of the improved version is to decrease the number of multiplications form 4 to 3

$$A * B = A_1 * B_1 \cdot 10^n + (A_1 * B_2 + A_2 * B_1) \cdot 10^{n/2} + A_2 * B_2$$

The product A*B can be written by the formula:

$$A * B = C_2 \cdot 10^n + C_1 \cdot 10^{n/2} + C_0$$

- $C_2 = A_1 * B_1$ (1 matrix multiplication)
- $C_0 = A_2 * B_2$ (1 matrix multiplication)
- $C_1 = (A_1 * B_2 + A_2 * B_1)$ (2 matrix multiplication)

❑ **Can we decrease the number of multiplications for C_1**

Improved Divide-and-Conquer Algorithm

C_1

$$(A_1 + A_2) * (B_1 + B_2) = A_1 * B_1 + \mathbf{(A_1 * B_2 + A_2 * B_1)} + A_2 * B_2$$

$$(A_1 * B_2 + A_2 * B_1) = (A_1 + A_2) * (B_1 + B_2) - A_1 * B_1 - A_2 * B_2$$

$$C_1 = (A_1 + A_2) * (B_1 + B_2) - C_2 * C_0$$ (3 matrix multiplications)

But what about additions and subtractions? Increase or decrease?

- **Analysis** of the enhanced divide-and-conquer algorithm
  - The number of recursive multiplications = 3. The time taken by all three recursive calls is **3 T(n/2)**
  - Additions and subtractions O(n)

Recurrence for the number of one-digit multiplications T(*n*):

$$T(n) = 3T(n/2) + cn, \quad T(1) = 1$$

a = 3 , b = 2 , d = 1 (Use master theorem)

$$T(n) = n^{log3} \approx n^{1.585}$$

How does this algorithm compare with the conventional approach??

Exercises

1) Compute 2101 $*$ 1130 by applying the improved divide-and-conquer algorithm.
2) Why did we not include multiplications by 10^n in the multiplication count $M(n)$ of the large-integer multiplication algorithm?

Divide-and-Conquer Examples

- **Sorting**:
  - merge sort
  - quicksort
- **Binary tree traversals**
- **Mathematics**
  - Multiplication of large integers
  - Matrix multiplication: Strassen's algorithm
  - Exponentiation problem
- **Computational geometry**
  - Closest-pair
  - convex-hull algorithms
- **Searching**:
  - Binary search: decrease-by-half (or degenerate divide&conq.)