Efficient Algorithms

Ekkapot Charoenwanit

Software Systems Engineering
TGGS
KMUTNB

Lecture 6: Divide and Conquer

Divide and Conquer

In the last lecture,

we saw how ***mergesort*** sorts an array $A[1 \ldots n]$ by recursively sorting smaller subarrays of approximately equal size $\frac{n}{2}$ and combining the results from the two subarrays to produce a sorted array of the original length n.

Divide and Conquer: The Three-Step Process

Generally, in the divide-and-conquer paradigm , we solve a problem by recursively, applying the following three steps at each level of recursion:

- ***Divide*** the problem into a number of smaller subproblems that are smaller instances of the same problem
- ***Conquer*** the subproblems by solving them recursively. However, if the sizes of subproblems are small enough, simply solve these subproblems in a straightforward way without the need to divide any further
- ***Combine*** the solutions to the subproblems into the solution to the original problem

Divide and Conquer: Recursion

When the subproblems are large enough to solve recursively, we call such cases ***the recursive case***.

Once the subproblems become small enough that we do not recurse anymore, we say that the recursion ***"bottoms out"*** and that we have reached ***the base case***.

In addition to the subproblems that are smaller instances of the same problem, we sometimes need to solve subproblems that are not quite the same as the original subproblem. We consider solving such subproblems in the ***combine step.

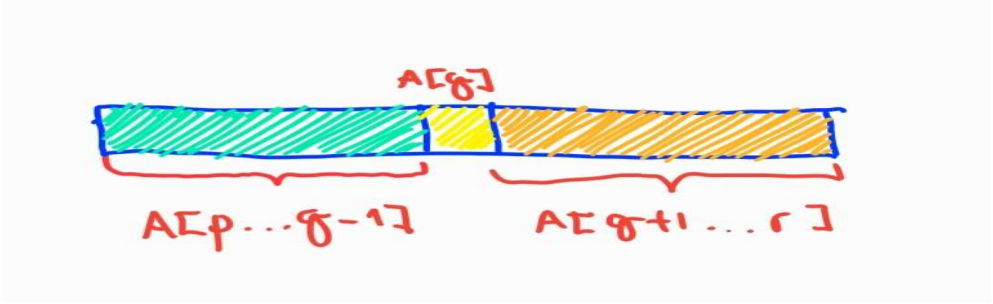
Quicksort

Like ***mergesort***, ***quicksort*** is a sorting algorithm based on ***the divide and conquer*** paradigm.

Typical of a divide-and-conquer algorithm, ***quicksort*** follows the following ***three-step*** divide-and-conquer process for sorting an array $A[p \ldots r]$: ***Divide, Conquer*** and ***Combine***

Quicksort: The Three-Step Process

Divide: Partition the array $A[p \ldots r]$ into two (possibly empty) subarrays $A[p \ldots q-1]$ and $A[q+1 \ldots r]$ such that each element in $A[p \ldots q-1]$ is smaller or equal to $A[q]$, which, in turn, is smaller or equal to $A[q+1 \ldots r]$.



Partition: Sort the two subarrays $A[p \ldots q-1]$ and $A[q+1 \ldots r]$ recursively.

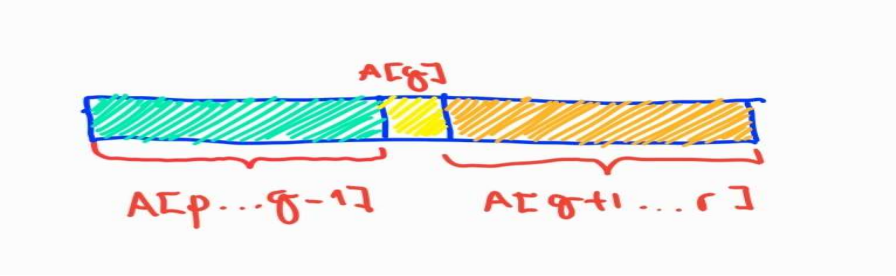
Combine: No work is needed to combine the two sorted arrays because the two subarrays are already sorted: the entire array $A[p \ldots r]$ is now sorted.

Quicksort: Pseudocode

To sort an entire array, $QuickSort(A, 1, A.length)$ is initially called.

```
1: procedure QUICK-SORT(A, p, r)
2:     if p < r then
3:         q = PARTITION(A, p, r)
4:         QUICKSORT(A, p, q − 1)
5:         QUICKSORT(A, q + 1, r)
```

The key part of the algorithm is to partition the array as shown in the figure.



Quicksort: Partitioning

In each recursive call, the algorithm always chooses $A[r]$ as the so called ***pivot***.

The $Partition$ routine computes the pivot element q in such a way that:

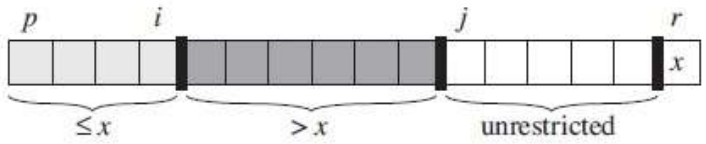
- all elements in $A[p \ldots q-1]$ are smaller or equal to q
- all elements in $A[q+1 \ldots r]$ are strictly larger than q

```
1: procedure PARTITION(A, p, r)
2:     x = A[r]
3:     i = p − 1
4:     for j = p → r − 1 do
5:         if A[j] ≤ x then
6:             i = i + 1
7:             A[i] ⟺ A[j]
8:     A[i + 1] ⟺ A[r]
9:     return i + 1
```

Quicksort: Pictorially

The $Partition$ routine partitions the array A into ***four*** (***possibly empty***) regions:

- The elements in the ***lightly-shaded*** region are no larger than the pivot $x = A[r]$.
- The elements in the ***heavily-shaded*** region are strictly greater than the pivot $x = A[r]$.
- The elements in the unshaded region have not yet been classified and hence put into the first two shaded regions so their values have no particular relationship to the pivot $x = A[r]$.
- The last element, which is the ***pivot***, is in the ***singleton*** region at the end of the array.



Observation:

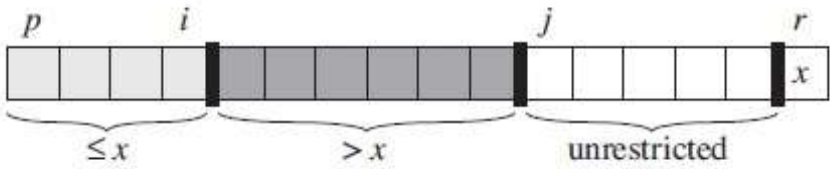
- i points to the last element of the yellow-shaded region.
- j points to the first element of the unrestricted region.

****Picture taken from CLRS***

Quicksort: Loop Invariant

At the start of each iteration of lines ***4-7*** for any array index k,

1. If $p \leq k \leq i$, then $A[k] \leq x$.
2. If $i + 1 \leq k \leq j - 1$, then $A[k] > x$.
3. If $k = r$, $A[k] = x$.




```
1: procedure PARTITION(A, p, r)
2:     x = A[r]
3:     i = p − 1
4:     for j = p → r − 1 do
5:         if A[j] ≤ x then
6:             i = i + 1
7:             A[i] ⟺ A[j]
8:     A[i + 1] ⟺ A[r]
9:     return i + 1
```

******Picture taken from CLRS***

Quicksort: Correctness

To prove ***correctness***, we need to show that:

- the ***loop invariant*** is true prior to the start of the first iteration
- the ***loop invariant*** is maintained at each iteration
- the ***loop invariant*** provides a useful property to show correctness when the loop terminates

Quick sort: Correctness

Initialization:

Prior to the start of the first iteration, $i = p - 1$ and $j = p$. The first two regions are initially empty so the first two invariant properties trivially hold, i.e., $F \rightarrow T \equiv T$. The third invariant property initially holds because of the assignment $x = A[r]$ in ***Line 2***.

```
1: procedure PARTITION(A, p, r)
2:     x = A[r]
3:     i = p − 1
4:     for j = p → r − 1 do
5:         if A[j] ≤ x then
6:             i = i + 1
7:             A[i] ⟺ A[j]
8:     A[i + 1] ⟺ A[r]
9:     return i + 1
```

Quicksort: Correctness

Maintenance: There are ***two cases*** to consider, depending on the test result of $A[j] \leq x$ in ***Line 4***.

Case I: $A[j] > x$

The only action performed is to increment j.

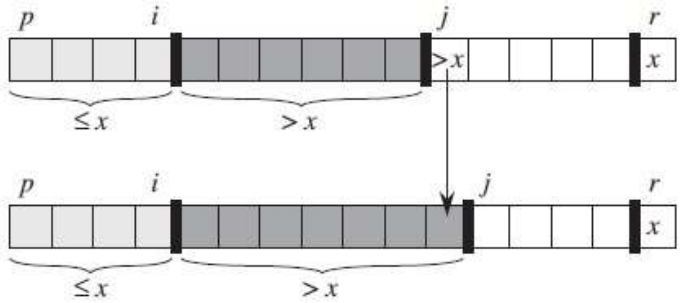
Since we assume the invariant holds at the start of the current iteration,

we have:

all elements in $A[p \ldots i] \leq x$

all elements in $A[i+1 \ldots j-1] > x$

$A[r] = x$



******Picture taken from CLRS***

Quicksort: Correctness

Maintenance: There are ***two cases*** to consider, depending on the test result of $A[j] \leq x$ in ***Line 4***.

Case I: $A[j] > x$

At the end of the iteration,

we increment j while i remains the same.

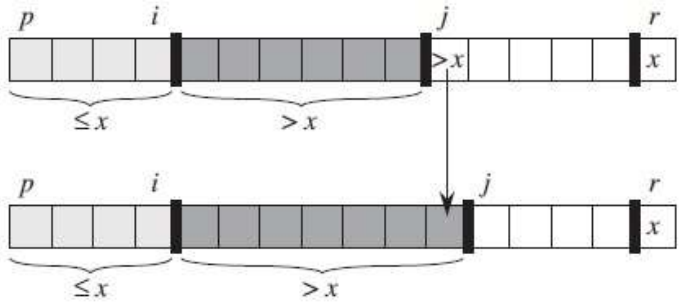
That is, $j' = j + 1$ and $i' = i$ and we have

all elements in $A[p \ldots i'] \leq x$

all elements in $A[i' + 1 \ldots j' - 1] > x$

$A[r] = x$

Therefore, the invariant still holds at the start of the next iteration j'.



******Picture taken from CLRS***

Quicksort: Correctness

Case II: $A[j] \leq x$

Since we assume the invariant holds at the start of the current iteration,
we have:

all elements in $A[p \ldots i] \leq x$

all elements in $A[i+1 \ldots j-1] > x$

$A[r] = x$

The actions performed are

to increment i by one and swap $A[i] \Leftrightarrow A[j]$.

After the increment, $i' = i + 1$ in ***Line 6***.

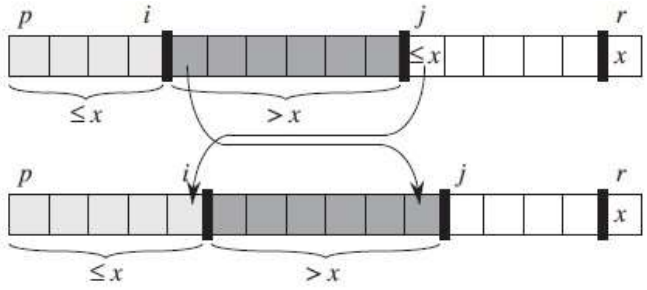
Since $A[j] \leq x$, after the swap $A[i'] \Leftrightarrow A[j]$ in ***Line 7***,

we have $A[i'] \leq x$

Since $A[i'] > x$ ***previously*** (by the loop invariant),

after the swap $A[i'] \Leftrightarrow A[j]$ in in ***Line 7***,

we have $A[j] > x$



******Picture taken from CLRS***

Quicksort: Correctness

Case II: $A[j] \leq x$

At the end of the iteration,

we increment j.

That is, $j' = j + 1$ and $i' = i + 1$ and we have

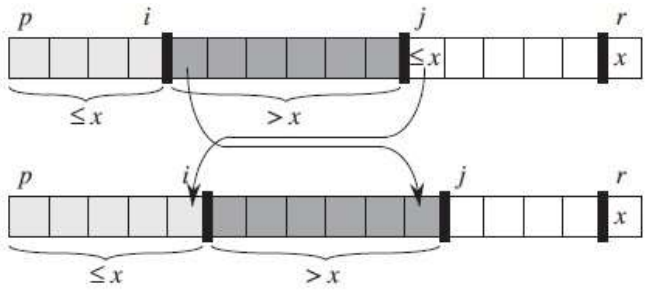
all elements in $A[p \ldots i'] \leq x$

all elements in $A[i' + 1 \ldots j' - 1] > x$

$A[r] = x$

Therefore, the invariant still holds at the start of the next iteration j'.

We have just shown that the invariant is maintained across iterations.



******Picture taken from CLRS***

Quicksort: Correctness

Termination: At termination, we have $j = r$.

Plugging $j = r$ into the three invariant properties, we have

1. If $p \leq k \leq i$, then $A[k] \leq x$.
2. If $i + 1 \leq k \leq r - 1$, then $A[k] > x$.
3. If $k = r$, $A[k] = x$.

Therefore, every entry in the array is in one of the three regions described by the invariant. We have partitioned the elements in the array into ***three sets*** as required: those smaller or equal to the pivot x, those strictly larger than x and a singleton set containing the pivot x.

The final two lines swaps the pivot with the leftmost element larger than the pivot, thereby placing the pivot into the correct position, and returns the pivot index. ■

Quicksort: Analysis of Partitioning

Let n be the number of elements in $A[p \ldots r]$.

Therefore, we have $n = r - p + 1$.

The number of iterations executed by $Partition$ is $r - p = n - 1$.

Since the loop body consists of only constant-time operations,
the running time of $Partition$ is $\Theta(n - 1) \equiv \Theta(n)$. ■

Quick sort: Worst-Case Analysis

The ***Worst-case*** behavior for quicksort occurs when the $Partition$ routine produces one subproblem with $n-1$ elements and one with 0 elements.

Let us assume that such an ***unbalanced partitioning*** happens in each recursive call.

Since the recursive call on an array of size 0 returns immediately,

$$T(0) = \Theta(1).$$

Quicksort: Worst-Case Analysis

Therefore, the running time of quicksort is

$$T(n) = T(n-1) + T(0) + \Theta(n)$$

Since $T(0) = \Theta(1)$,

$$T(n) = T(n-1) + \Theta(1) + \Theta(n)$$

The $\Theta(1)$ term can be absorbed into the more asymptotically dominant $\Theta(n)$ term.

Therefore,

$$T(n) = T(n-1) + \Theta(n)$$

Solving the recurrence, we have $T(n) = \Theta(n^2)$.

Quicksort

Although quicksort is $\Theta(n^2)$ in the worst case,

it performs fairly well in the average case as we will see in a future lecture on ***Randomized Algorithms***.

We will see that the randomized version of quicksort runs in ***expected time*** $O(n \log n)$.

Another crucial property of quicksort is that it is an ***in-place*** sorting algorithm.

Matrix Multiplication: Naïve Algorithm

Naïve matrix multiplication can be implemented in an iterative fasion as follows:

```
1: procedure NAIVE-SQUARE-MAT-MUL(A, B)
2:     n = A.rows
3:     C = new MATRIX(n)
4:     for i = 1 → n do
5:         for j = 1 → n do
6:             C[i][j] = 0
7:             for k = 1 → n do
8:                 C[i][j]+ = A[i][k] · B[k][j]
9:     return C
```

Matrix Multiplication: Naïve Algorithm

Because each of the ***three nested loops*** runs exactly n times and each execution of ***Line 8*** runs in constant time $\Theta(1)$,

the running time of the naïve algorithm takes $T(n) = \Theta(n^3)$.

We will see shortly that we can multiply two square matrices ***asymptotically better*** than $\Theta(n^3)$ since there is a ***divide-and-conquer*** algorithm called ***Strassen***'s algorithm that runs in $\Theta(n^{\log_2 7}) \approx \Theta(n^{2.81})$ time.

Matrix Multiplication: Naïve DQ Algorithm

Before we talk about ***Strassen's***, we shall try to implement MM in a straightforward, recursive, divide-and-conquer manner.

To keep things simple, we shall assume that n is an ***exact power of two*** in each of the $n \times n$ matrices.

We make this assumption because in each divide step, we will divide $n \times n$ matrices into ***four*** $\frac{n}{2} \times \frac{n}{2}$ matrices.

Naïve DQ MM Algorithm

Suppose we partition A ,B and C into ***four*** $\frac{n}{2} \times \frac{n}{2}$ matrices as follows:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

So we can write A , B and C as:

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

Naïve DQ MM Algorithm

Therefore, we have four ***matrix-level*** equations as follows:

$$C_{11} = A_{11}\, B_{11} + A_{12} B_{21}$$
$$C_{12} = A_{11}\, B_{12} + A_{12} B_{22}$$
$$C_{21} = A_{21}\, B_{11} + A_{22} B_{21}$$
$$C_{22} = A_{21}\, B_{12} + A_{22} B_{22}$$

Each of these four equations specifies ***two matrix-level multiplications*** of $\frac{n}{2} \times \frac{n}{2}$ matrices and ***one matrix-level addition*** of their products.

Naïve DQ MM Algorithm

Therefore, we can use these equations to create a straightforward recursive divide-and-conquer algorithm for MM.

```
1:  procedure NAIVE-DQ-MM(A, B)
2:      n = A.rows
3:      C = new MATRIX(n)
4:      if n = 1 then
5:          c_{1,1} = a_{1,1} · b_{1,1}
6:      else
7:          Partition A, B and C
8:          C_{1,1} = NAIVE-DQ-MM(A_{1,1}, B_{1,1}) + NAIVE-DQ-MM(A_{1,2}, B_{2,1})
9:          C_{1,2} = NAIVE-DQ-MM(A_{1,1}, B_{1,2}) + NAIVE-DQ-MM(A_{1,2}, B_{2,2})
10:         C_{2,1} = NAIVE-DQ-MM(A_{2,1}, B_{1,1}) + NAIVE-DQ-MM(A_{2,2}, B_{2,1})
11:         C_{2,2} = NAIVE-DQ-MM(A_{2,1}, B_{1,2}) + NAIVE-DQ-MM(A_{2,2}, B_{2,2})
12:     return C
```

Naïve DQ MM Algorithm: Running Time

The pseudocode glosses over one subtle but important implementation detail: How do we partition A ,B and C in ***Line 7***?

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

This way, we can compute ***Line 7*** in $\Theta(1)$ time.

Naïve DQ MM Algorithm: Running Time

Let $T(n)$ be the running time of the naïve DQ MM algorithm multiplying two $n \times n$ matrices.

In the ***base case***, when $n = 1$, we perform only the one scalar multiplication in ***Line 5*** and so $T(1) = \Theta(1)$.

The ***recursive case*** occurs when $n > 1$. As discussed, the partitioning in ***Line 7*** takes $\Theta(1)$ time using ***index calculations***.

Naïve DQ MM Algorithm: Running Time

In ***Lines 8-11***, we recursively call $NaiveDQMM$ ***8*** times.

Each recursive call multiplies ***two*** $\frac{n}{2} \times \frac{n}{2}$ matrices, thereby contributing $T(\frac{n}{2})$ to the overall running time. The time taken by the ***8*** recursive calls is then $8T(\frac{n}{2})$.

Additionally, we must also take into account the ***four matrix-level additions*** in ***Lines 8-11***.

Each of these matrices has $\frac{n^2}{4}$ entries and requires $\frac{n^2}{4}$ ***scalar additions***.

In total, there are $4 \times \frac{n^2}{4} = n^2 = \Theta(n^2)$ scalar additions.

Naïve DQ MM Algorithm: Running Time

The total running time is the sum of the time taken to ***partition***, the time for the ***eight recursive calls*** and the time to perform ***scalar additions***.

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

Since the constant-time term $\Theta(1)$ can be absorbed into the more asymptotically dominating term $\Theta(n^2)$, we have

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

Naïve DQ MM Algorithm: Running Time

Solving the recurrence,

$$T(1) = \Theta(1) \qquad \text{for } n = 1$$
$$T(n) = 8T\left(\frac{n}{2}\right) + \Theta(n^2) \text{ for } n > 1$$

we have

$$T(n) = \Theta(n^3)$$

,which is ***asymptotically identical*** to the running time of the naïve algorithm implemented using loop iterations discussed previously.

Strassen's Algorithm

The key idea of ***Strassen's algorithm*** is to make the recursion tree ***slightly less bushy***.

Instead of performing ***eight*** matrix-level multiplications in each recursive call, it performs only ***seven***.

The cost of eliminating ***one*** matrix-level multiplication will increase the number of matrix-level additions/subtractions to ***18***.

Strassen's Algorithm

Strassen's algorithm makes use of the following fact for multiplying two 2×2 matrices:

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

$$= \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix}$$

Strassen's Algorithm

$$\begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix}$$

where

$$m_1 = (a_{11}+a_{22})(b_{11}+b_{22})$$
$$m_2 = (a_{21}+a_{22})(b_{11})$$
$$m_3 = (a_{11})(b_{12}-b_{22})$$
$$m_4 = (a_{22})(b_{21}-b_{11})$$
$$m_5 = (a_{11}+a_{12})(b_{22})$$
$$m_6 = (a_{21}-a_{11})(b_{11}+b_{12})$$
$$m_7 = (a_{12}-a_{22})(b_{21}+b_{22})$$

Strassen's Algorithm: Analysis

In the ***base case,*** when $n = 1$, only one scalar multiplications is performed so $T(1) = \Theta(1)$.

In the ***recursive case***,
Partitioning takes $\Theta(1)$ time using index calculations.

There are now ***seven*** matrix-level multiplications, each of which multiplies two $\frac{n}{2} \times \frac{n}{2}$ matrices, thereby contributing $7T\left(\frac{n}{2}\right)$ to the overall running time.

There are now ***18*** matrix-level additions/subtractions, each adding or subtracting two $\frac{n}{2} \times \frac{n}{2}$ matrices, thereby contributing to $18 \times \frac{n^2}{4}$ to the overall running time.

Summing up all the contributions, we have

$$T(n) = 7T\left(\frac{n}{2}\right) + \Theta(n^2)$$

Strassen's Algorithm: Analysis

Solving the recurrence,

$$T(1) = \Theta(1) \qquad \text{for } n = 1$$

$$T(n) = 7T\left(\frac{n}{2}\right) + \Theta(n^2) \qquad \text{for } n > 1$$

we have

$$T(n) = \Theta(n^{\log_2 7}) \cong \Theta(n^{2.81})$$

Since $n^{2.81} \in O(n^3)$,

Strassen's algorithm is asymptotically better than the naïve algorithm.

Strassen's Algorithm: Analysis

Since $n^{2.81} \in O(n^3)$,

Strassen's algorithm is asymptotically better than the naïve algorithm.

We know that any matrix multiplication algorithm must take at least $\Omega(n^2)$ time.

***The question whether there exists a matrix multiplication algorithm that runs in $\Theta(n^2)$ time still remains unsolved.

Strassen's Algorithm: No Practical Use

Because of its complexity, it has a ***very large multiplicative constant*** so Strassen's algorithm has no real practical use.

It is only interesting from the theoretical perspective.

Since the invention of Strassen's algorithm, there have been several other algorithms that run in $O(n^{\alpha})$ time with ***progressively smaller constants*** α.

Master Theorem

In a more general case of divide-and-conquer, a problem's instance of size n is divided into a instances of size $\frac{n}{b}$, where $a \geq 1$ and $b > 1$.

That is, the running time can be written as a recurrence:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

where $f(n)$ is a function that accounts for the time spent on dividing the problem into smaller ones and on combining their solutions.

Theorem: If $f(n) \in \Theta(n^d)$ with $d \geq 0$, then

$$T(n) \in \Theta(n^d) \quad \text{if } a < b^d$$
$$T(n) \in \Theta(n^d \log n) \quad \text{if } a = b^d$$
$$T(n) \in \Theta(n^{\log_b a}) \quad \text{if } a > b^d$$

Master Theorem: Example

Applying ***Master Theorem*** to the running time of ***mergesort***

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

we have $a = 2$,$b = 2$ and $f(n) = O(n)$ i.e. $d = 1$.

Since $a = 2 \ = 2^1 = b^d$, this is the ***second case*** of Master theorem.

Therefore, $T(n) = O\left(n^d \log n\right) = O(n^1 \log n) = O(n \log n)$.

Summary

We have covered the following topics:

- The Three-Step Process in the Divide-and-Conquer Paradigm.
- Quick Sort
- Matrix Multiplication: Naïve and Strassen's Method
- Master Theorem

In the next lecture, we will cover ***dynamic programming***.