06 - Divide and Conquer (part 2)

[KOMS119602] & [KOMS120403]

Design and Analysis of Algorithm (2021/2022)

Dewi Sintiari

Prodi S1 Ilmu Komputer
Universitas Pendidikan Ganesha

Week 21-25 March 2022

Table of contents

Master Theorem

How to deal with
long computation of time complexity?

Master Theorem (1)

When analyzing algorithms, recall that we only care about the asymptotic behavior.

The Master Theorem can be used to determine the asymptotic notation of time complexity in the form of a recurrence relation easily without having to solve it iteratively.

Theorem (Master Theorem)

Given the time complexity function: $T(n) = aT(n/b) + f(n)$*.*
If $f(n) \in \Theta(n^d)$ *where* $d \geq 0$*, then:*

$$T(n) \in \begin{cases} \Theta(n^d), & \text{if } a < b^d \\ \Theta(n^d \log n), & \text{if } a = b^d \\ \Theta(n^{\log_b a}), & \text{if } a > b^d \end{cases}$$

Analogous results also hold for the Ω *and* $\mathcal{O}$ *notations.*

Master Theorem (2): Example 1

In Merge Sort/Quick Sort,

$$T(n) = \begin{cases} t, & \text{for } n = 1 \\ 2T(n/2) + cn, & \text{for } n > 1 \end{cases}$$

- $T(n) = aT(n/b) + cn^d$
- $a = 2,\ b = 2,\ d = 1$
- $a = b^d$ is satisfied (namely $2 = 2^1$)

So the recurrence relation $T(n) = aT(n/b) + cn^d$ satisfies the *2nd case* of the following function.

$$T(n) \in \begin{cases} \mathcal{O}(n^d), & \text{if } a < b^d \\ \mathcal{O}(n^d \log n), & \text{if } a = b^d \\ \mathcal{O}(n^{\log_b a}), & \text{if } a > b^d \end{cases}$$

So, $T(n) \in \mathcal{O}(n \log n)$.

Master Theorem (2): Example 2

In Powering algorithm to compute X^n,

$$T(n) \in \begin{cases} 1, & \text{for } n = 0 \\ T(n/2) + 1, & \text{for } n > 0 \end{cases}$$

- $T(n) = aT(n/b) + cn^d$
- $a = 1,\ b = 2,\ d = 0$
- $a = b^d$ is satisfied (namely $1 = 2^0$)

So the recurrence relation $T(n) = aT(n/b) + cn^d$ satisfies the *2nd case* of the following function.

$$T(n) \in \begin{cases} \mathcal{O}(n^d), & \text{if } a < b^d \\ \mathcal{O}(n^d \log n), & \text{if } a = b^d \\ \mathcal{O}(n^{\log_b a}), & \text{if } a > b^d \end{cases}$$

So, $T(n) \in \mathcal{O}(n^0 \log n) = \mathcal{O}(\log n)$.

Master Theorem (2): Example 3

In divide-and-conquer array-sum-computation algorithm, given the input size is $n = 2^k$, we have time complexity function:

$$T(n) = 2T(n/2) + 1$$

because:

- at each step, the problem is divided into 2 sub-problems of equal size ($b = 2$), and both of them must be solved ($a = 2$).
- The DIVIDE and COMBINE complexity function is $f(n) \in \Theta(1) = \Theta(n^0)$

Hence,

$$T(n) \in \Theta\left(n^{\log_b a}\right) = \left(n^{\log_2 2}\right) = \Theta(n)$$

Let $T(n) = 2T(\frac{n}{4}) + \sqrt{n} + 42$. What are the parameters?

Master Theorem (2): Example 4

Let $T(n) = 2T(\frac{n}{4}) + \sqrt{n} + 42$. What are the parameters?

$$a = 2;\; b = 4;\; d = \frac{1}{2}$$

Therefore, which condition?

Master Theorem (2): Example 4

Let $T(n) = 2T(\frac{n}{4}) + \sqrt{n} + 42$. What are the parameters?

$$a = 2;\ \ b = 4;\ \ d = \frac{1}{2}$$

Therefore, which condition?

Since $2 = 4^{\frac{1}{2}}$, case 2 of Master Thm applies. Hence,

$$T(n) \in \Theta(n^d \log n) = \Theta(\sqrt{n} \log n)$$

Divide-and-Conquer + Master Theorem = ?

Divide-and-Conquer + Master Theorem = ?

The combination of the two gives you the ability to very quickly iterate between algorithm design and its runtime analysis.

Very pro way of algorithm development!

Master Theorem (3): Importance & proof

Divide-and-Conquer + Master Theorem = ?

The combination of the two gives you the ability to very quickly iterate between algorithm design and its runtime analysis.

Very pro way of algorithm development!

The proof can be read in this lecture note (pages 3-4):

https://web.stanford.edu/class/archive/cs/cs161/cs161.1182/Lectures/Lecture3/CS161Lecture03.pdf

Master Theorem (5): Advantages & drawbacks

- Master theorem lets you go from the recurrence to the asymptotic bound very quickly.
- It typically works well for divide-and-conquer algorithms.
- But Master theorem *does not apply to all recurrences*.
  - $T(n)$ is not monotone, ex: $T(n) = \sin n$
  - $f(n)$ is not a polynomial, ex: $T(n) = 2T(\frac{n}{2}) + 2^n$
  - b cannot be expressed as a constant, ex: $T(n) = T(\sqrt{n})$
- When it does not apply, you can:
  - do some upper/lower bounding and get a potentially looser bound
  - use the substitution method

Matrix multiplication

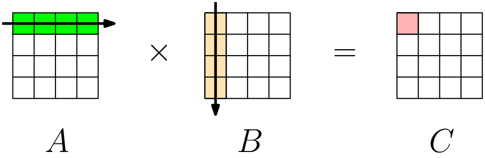
Square matrix multiplication (1)

Problem

Given two square matrices A and B. Compute $A \times B$

Let $A = [a_{ij}]$, $B = [b_{ij}]$ be $n \times n$ matrices, and $C = A \times B$.

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj} = \sum_{k=1}^{n} a_{ik}b_{kj}$$



Square matrix multiplication (2)

Brute-force approach: compute each element of C one-by-one by multiplying the corresponding row of A and column of B.

Algorithm 1 Square matrix multiplication (*brute force*)

```
procedure MatrixMult(A, B)
    for i ← 1 to n do
        for j ← 1 to n do
            C[i, j] ← 0
            for k ← 1 to n do
                C[i, j] ← C[i, j] + A[i, k] ∗ B[k, j]
            end for
        end for
    end for
    return C
end procedure
```

Time complexity: $\mathcal{O}(n^3)$

Square matrix multiplication (3)

Matrices A and B are each split into four submatrices of size $\frac{n}{2} \times \frac{n}{2}$.

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

Hence the component of matrix C can be computed as follows:

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

Square matrix multiplication (3)

Example

A square matrix can be split as follows:

$$A = \left[\begin{array}{cc|cc} 1 & 21 & 15 & 7 \\ 11 & 3 & 10 & 31 \\ \hline 52 & 31 & 2 & 17 \\ 2 & 9 & 23 & 3 \end{array}\right]$$

$$A_{11} = \begin{bmatrix} 1 & 21 \\ 11 & 3 \end{bmatrix} \quad A_{12} = \begin{bmatrix} 15 & 7 \\ 10 & 31 \end{bmatrix}$$

$$A_{21} = \begin{bmatrix} 52 & 31 \\ 2 & 9 \end{bmatrix} \quad A_{22} = \begin{bmatrix} 2 & 17 \\ 23 & 3 \end{bmatrix}$$

Square matrix multiplication (3): Pseudocode

Algorithm 2 Matrix multiplication

procedure $\textsc{MMul}(A, B\text{: matrices}, n\text{: integer})$
  if $n = 1$ **then** ▷ *The matrices are of size 1×1*
    return $A * B$ ▷ *Scalar multiplication*
  else
    $\textsc{Split}(A)$
    $\textsc{Split}(B)$
    $C_{11} \leftarrow \textsc{MSum}(\textsc{MMul}(A_{11}, B_{11}, \frac{n}{2}), \textsc{MMul}(A_{12}, B_{21}, \frac{n}{2}))$
    $C_{12} \leftarrow \textsc{MSum}(\textsc{MMul}(A_{11}, B_{12}, \frac{n}{2}), \textsc{MMul}(A_{12}, B_{22}, \frac{n}{2}))$
    $C_{21} \leftarrow \textsc{MSum}(\textsc{MMul}(A_{21}, B_{11}, \frac{n}{2}), \textsc{MMul}(A_{22}, B_{21}, \frac{n}{2}))$
    $C_{22} \leftarrow \textsc{MSum}(\textsc{MMul}(A_{21}, B_{12}, \frac{n}{2}), \textsc{MMul}(A_{22}, B_{22}, \frac{n}{2}))$
  end if
  return C ▷ *C is the union of $C_{11}, C_{12}, C_{21}, C_{22}$*
end procedure

Square matrix multiplication (3): Pseudocode

The procedure MSUM used in MMUL is as follows.

Algorithm 3 Sum of two matrices

```
procedure MSUM(A, B: matrices, n: integer)
    for i ← 1 to n do
        for j ← 1 to n do
            C[i, j] ← A[i, j] + B[i, j]
        end for
    end for
end procedure
```

Time complexity: $\mathcal{O}(n^2)$

Square matrix multiplication (3): Time complexity

The recursive formula for TC is given by:

$$T(n) = \begin{cases} a, & n = 1 \\ 8T(n/2) + cn^2, & n > 1 \end{cases}$$

- By Master Thm:

  $$T(n) = aT\left(\frac{n}{b}\right) + cn^d$$

  where $a = 8,\ b = 2,\ d = 2.$
- The relation $a > b^d$ (namely $8 > 2^2$) is satisfied.
- So $T(n)$ satisfies 3rd case of Master Thm. Hence:

  $$T(n) = \mathcal{O}(n^{\log_2 8}) = \mathcal{O}(n^3)$$

This gives TC with *same order of magnitude as brute force*. So the algorithm is not so powerful. Can we do better?

Strassen Matrix multiplication

Figure: Volker Strassen (born in 1936, German mathematician)

Strassen matrix multiplication (1)

- Volker Strassen's idea is to reduce the number of 'multiplications' in the procedure. Since the 'multiplication' cost is more 'expensive' than the 'addition' (see `https://www.wikiwand.com/en/Computational_complexity_of_mathematical_operations`).
- The following operations consist of 8 multiplications and 4 additions:
  - $C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21}$
  - $C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22}$
  - $C_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21}$
  - $C_{22} = A_{21} \cdot B_{22} + A_{22} \cdot B_{22}$
- Strassen modifies the above equations to reduce it to 7 multiplications but with more additions

Strassen matrix multiplication (2)

The modification is as follows:

- $M_1 = (A_{11} - A_{22})(B_{21} + B_{22})$
- $M_2 = (A_{11} + A_{22})(B_{11} + B_{22})$
- $M_3 = (A_{11} - A_{21})(B_{11} + B_{12})$
- $M_4 = (A_{11} + A_{12})B_{22}$
- $M_5 = A_{11}(B_{12} - B_{22})$
- $M_6 = A_{22}(B_{21} - B_{11})$
- $M_7 = (A_{21} + A_{22})B_{11}$

Hence:

- $C_{11} = M_1 + M_2 - M_4 + M_6$
- $C_{12} = M_4 + M_5$
- $C_{21} = M_6 + M_7$
- $C_{22} = M_2 - M_3 + M_5 - M_7$

This operation consists of 7 multiplications and 18 additions.

Algorithm 4 Matrix multiplication

procedure STRASSEN(A, B: matrices, n: integer)
  if $n = 1$ **then return** $A * B$ ▷ *Scalar multiplication*
  else
    SPLIT(A)
    SPLIT(B)
    $M_1 \leftarrow$ STRASSEN$(A_{12} - A_{22}, B_{21} + B_{22}, \frac{n}{2})$
    $M_2 \leftarrow$ STRASSEN$(A_{11} + A_{22}, B_{11} + B_{22}, \frac{n}{2})$
    $M_3 \leftarrow$ STRASSEN$(A_{11} - A_{21}, B_{11} + B_{12}, \frac{n}{2})$
    $M_4 \leftarrow$ STRASSEN$(A_{11} + A_{12}, B_{22}, \frac{n}{2})$
    $M_5 \leftarrow$ STRASSEN$(A_{11}, B_{12} - B_{22}, \frac{n}{2})$
    $M_6 \leftarrow$ STRASSEN$(A_{22}, B_{21} - B_{11}, \frac{n}{2})$
    $M_7 \leftarrow$ STRASSEN$(A_{21} + A_{22}, B_{11}, \frac{n}{2})$
    $C_{11} \leftarrow M_1 + M_2 - M_4 + M_6$
    $C_{12} \leftarrow M_4 + M_5$
    $C_{21} \leftarrow M_6 + M_7$
    $C_{22} \leftarrow M_2 - M_3 + M_5 - M_7$
  end if
  return C ▷ *C is the union of $C_{11}, C_{12}, C_{21}, C_{22}$*
end procedure

Strassen matrix multiplication (3)

The recursive formula for TC is given by:

$$T(n) = \begin{cases} a, & n = 1 \\ 7T(n/2) + cn^2, & n > 1 \end{cases}$$

- By Master Thm, $T(n) = aT\left(\frac{n}{b}\right) + cn^d$, where $a = 7$, $b = 2$, $d = 2$.
- The relation $a > b^d$ (namely $7 > 2^2$) is satisfied.
- So $T(n)$ satisfies 3rd case of Master Thm. Hence:

$$T(n) = \mathcal{O}(n \log_2 7) = \mathcal{O}(n^{2.81})$$

This gives a better TC than the previous divide-and-conquer algorithm.

Large number multiplication

Large number multiplication (1): definition

A large number is a number that contains n digits or n bits.

Example: 564389018149014329871520, 1000011011010100100110010101, ...

Issues with large numbers

- Programming languages have limitation in representing large numbers
- In C, number types are `char` (8 bit), `int` (6 bit), and `long` (32 bit)
- For the numbers that are greater than 32 bits, we have to define *new type* and define the primitive arithmetic operations $(+, -, *, /,$ etc.)

Large number multiplication (2): problem statement

We will discuss how an algorithm can perform multiplication with large numbers

Example: 17654208752083451 86 × 754711199736308361736432

Problem

Given two integers X and Y of n digits (or n bits):

$$X = x_1 x_2 x_3 \ldots x_n$$
$$Y = y_1 y_2 y_3 \ldots y_n$$

Compute $X \times Y$

Large number multiplication (3): classical multiplication

Example

$$X = 1234 \quad (n = 4)$$

$$Y = 5678 \quad (n = 4)$$

Classical way to perform $X \times Y$:

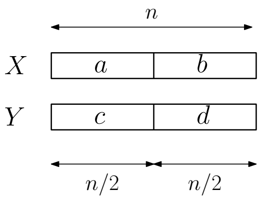
$$
\begin{array}{r}
X \times Y = 1234 \\
\underline{5678 \times} \\
9872 \\
8368 \\
7404 \\
\underline{6170 +} \\
7006652
\end{array}
$$

Large number multiplication (3): pseudocode

Algorithm 5 Large number multiplication (*brute force*)

procedure MULT(X, Y: long integer, n: integer)
    declaration
        temp, unit, tens: **integer**
    end declaration
    for every digit y_i of $y_n, y_{n-1}, \ldots, y_1$ **do**
        tens $\leftarrow 0$
        for every digit x_j of $x_n, x_{n-1}, \ldots, x_1$ **do**
            temp $\leftarrow x_j * y_i$
            temp $\leftarrow$ temp $+$ tens
            unit $\leftarrow$ temp mod 10
            tens $\leftarrow$ temp **div** 10
            print(unit)
        end for
    end for
    $Z \leftarrow$ add all results of the multiplication from top to bottom
    return Z
end procedure

Large number multiplication (4): DnC approach



$a = X$ **div** $10^{n/2}$ $\quad b = X$ **mod** $10^{n/2}$

$c = Y$ **div** $10^{n/2}$ $\quad d = Y$ **mod** $10^{n/2}$

X and Y can be represented as a, b, c, and d:

$$X = a \cdot 10^{n/2} + b \quad \text{and} \quad Y = c \cdot 10^{n/2} + d$$

The multiplication of X and Y is represented as:

$$\begin{aligned} X \cdot Y &= (a \cdot 10^{n/2} + b) \cdot (c \cdot 10^{n/2} + d) \\ &= ac \cdot 10^{n} + ad \cdot 10^{n/2} + bc \cdot 10^{n/2} + bd \\ &= ac \cdot 10^{n} + (ad + bc) \cdot 10^{n/2} + bd \end{aligned}$$

Large number multiplication (5): DnC approach

Example

Let $n = 6$, $X = 346769$ and $Y = 279431$. Then:

$$X = 346769 \rightarrow a = 346, b = 769 \rightarrow X = 346 \cdot 10^3 + 769$$
$$Y = 279431 \rightarrow c = 279, d = 431 \rightarrow Y = 279 \cdot 10^3 + 431$$

The multiplication of X and Y can be written as:

$$\begin{aligned} X \cdot Y &= (346 \cdot 10^3 + 769) \cdot (279 \cdot 10^3 + 431) \\ &= (346)(279) \cdot 10^6 + ((346)(431) + (769)(279)) \cdot 10^3 + (769)(431) \end{aligned}$$

This operation involves **four** large numbers multiplication.

Algorithm 6 Large number multiplication (DnC)

```
procedure MULT2(X, Y: long integer, n: integer)
    declaration
        a, b, c, d: Long integer,  s: integer
    end declaration
    if n = 1 then
        return X * Y                                   ▷ scalar multiplication
    else
        s ← n div 2
        a ← X div 10^s
        b ← X mod 10^s
        c ← Y div 10^s
        d ← Y mod 10^s
        return MULT2(a, c, s)*10^2s + MULT2(b, c, s)*10^s + MULT2(a, d, s)*10^s + MULT2(b, d, s)
    end if
end procedure
```

Large number multiplication (6): time complexity

Time complexity of MULT2

$$T(n) = \begin{cases} a & \text{for } n = 1 \\ 4T(n/2) + cn & \text{for } n > 1 \end{cases}$$

Remark. Computing 10^s and 10^{2s} in the algorithm can be done by adding s or $2s$ zeros.

By Master Thm, we obtain (**prove it!**):

$$T(n) = \mathcal{O}(n^2)$$

This algorithm has the same complexity (asymptotically) as the brute force algorithm. Can we do better?

Karatsuba multiplication

Figure: Anatoly Alexeyevich Karatsuba (1937-2008, Russian mathematician)

Improvement of the previous multiplication algorithm

The idea is similar to the *Strassen matrix multiplication*, by reducing the number of multiplication.

The previous algorithm gives:

$$X \cdot Y = ac \cdot 10^n + (ad + bc) \cdot 10^{n/2} + bd$$

Karatsuba manipulates the above equation such that it needs only 3 multiplications, but consequently, it needs more addition.

Let

$$r = (a+b)(c+d) = ac + (ad+bc) + bd$$

Then

$$(ad+bc) = r - ac - bd = (a+b)(c+d) - ac - bd$$

So, the multiplication $X \cdot Y$ can be written as:

$$\begin{aligned} X \cdot Y &= ac \cdot 10^n + (ad+bc) \cdot 10^{n/2} + bd \\ &= \underbrace{ac}_{p} \cdot 10^n + (\underbrace{(a+b)(c+d)}_{r} - \underbrace{ac}_{p} - \underbrace{bd}_{q}) \cdot 10^{n/2} + \underbrace{bd}_{q} \end{aligned}$$

Now the algorithm only contains 3 multiplications, to compute p, q, and r.

Karatsuba multiplication (3): pseudocode

Algorithm 7 Karatsuba multiplication

```
procedure MULT3(X, Y: long integer, n: integer)
    declaration
        a, b, c, d, p, q, r: Long integer,  s: integer
    end declaration
    if n = 1 then
        return X * Y                                  ▷ scalar multiplication
    else
        s ← n div 2
        a ← X div 10^s
        b ← X mod 10^s
        c ← Y div 10^s
        d ← Y mod 10^s
        p ← MULT3(a, c, s)
        q ← MULT3(b, d, s)
        r ← MULT3(a + b, c + d, s)
        return p * 10^(2s) + (r − p − q) * 10^s + q
    end if
end procedure
```

Time complexity of Mult3

$T(n)$: three multiplications of integers of $n/2$ digits + addition of integers of $n/2$ digits

$$T(n) = \begin{cases} a & \text{for } n = 1 \\ 3T(n/2) + cn & \text{for } n > 1 \end{cases}$$

From $T(n) = 3T(n/2) + cn$, we have $a = 3$, $b = 2$, $d = 1$, and $a > b^d$ (namely $3 > 2^1$).

So the recurrence formula satisfies the 3rd case of Master Thm (namely $a > b^d$). So:

$$T(n) = \mathcal{O}(n^{\log_2 3}) = \mathcal{O}(n^{1.59})$$

This is better than $\textsc{Mult2}$ (which is $\mathcal{O}(n^2)$).

Summary

Advantages of DnC method

- **Solving difficult problems:** It is a powerful method for solving difficult problems. Dividing the problem into subproblems so that subproblems can be combined again is a major difficulty in designing a new algorithm. For many such problem this algorithm provides a simple solution.
- **Parallelism:** Since it allows us to solve the subproblems independently, this allows for execution in multi-processor machines, especially shared-memory systems where the communication of data between processors does not need to be planned in advance, because different subproblems can be executed on different processors.

Drawbacks of DnC method

- **Recursion is slow:** This is because of the overlap of the repeated subproblem calls. Also the algorithm need stack for storing the calls. (But actually this depends upon the implementation style.)