Algorithms and Data Structures

Module 2

Lecture 12

‘Divide-and-Conquer’ strategy. Multiplication.

Multiplication problem

Given two integer numbers x and y, calculate $z = x \cdot y$.

Numbers can be represented in either decimal or binary form.

Multiplication: standard algorithm

```
        23958233
×           5830
───────────────
        00000000  ( =     23,958,233 ×     0)
       71874699   ( =     23,958,233 ×    30)
     191665864    ( =     23,958,233 ×   800)
+ 119791165       ( =     23,958,233 × 5,000)
───────────────
    139676498390  ( = 139,676,498,390        )
```

Time complexity: $(O(n)\ multiplications + O(n)\ additions) \cdot n = O(n^2)$ for multiplying n-digit numbers.

Multiplication: Divide-and-Conquer

Lets apply ‘Divide-and-Conquer’ approach to the problem of multiplication.

x =	a	b
y =	c	d

2395	8233
0000	5830

$10^{n/2}a + b$

$10^{n/2}c + d$

Thus, $z = x \cdot y = \left(10^{\frac{n}{2}}a + b\right) \cdot \left(10^{\frac{n}{2}}c + d\right) = 10^{n}ac + 10^{\frac{n}{2}}(ad + bc) + bd.$

We have 4 multiplications and 3 additions.

Multiplication: Divide-and-Conquer

$z = x \cdot y = \left(10^{\frac{n}{2}}a + b\right) \cdot \left(10^{\frac{n}{2}}c + d\right) = 10^n ac + 10^{\frac{n}{2}}(ad + bc) + bd.$

We have 4 multiplications and 3 additions.
We can use a recursive function to compute the product.

Function RecursiveProduct(x, y, n):

1. Base case: if $n < 2$ then return $x \cdot y$.
2. Divide the factors into parts: a, b, c, d.
3. Recursively calculate:

    p = RecursiveProduct($a, c, n/2$)

    q = RecursiveProduct($a, d, n/2$)

    r = RecursiveProduct($b, c, n/2$)

    s = RecursiveProduct(b, d, $n/2$)

4. Return $10^n p + 10^{\frac{n}{2}}(q + r) + s$.

Multiplication: Divide-and-Conquer

$z = x \cdot y = \left(10^{\frac{n}{2}}a + b\right) \cdot \left(10^{\frac{n}{2}}c + d\right) = 10^n ac + 10^{\frac{n}{2}}(ad + bc) + bd.$

Let $T(n)$ denote the time for multiplying two *n*-digit numbers.

We have 4 multiplications and 3 additions in each recursive call.

Each multiplication takes $T\left(\frac{n}{2}\right)$ time, each addition takes $O(n)$ time.

Thus, $T(n) = 4 \cdot T\left(\frac{n}{2}\right) + O(n)$.

Divide-and-Conquer: Master Theorem (simplified version)

The **Master Theorem (simplified version)**: if $T(n)$ satisfies the following recurrence

$$T(n) = \begin{cases} c, & for \quad n = 1 \\ a \cdot T\left(\frac{n}{b}\right) + d \cdot n, & for \quad n > 1 \end{cases}$$

Then there are 3 cases:

1) If $a < b$, then $T(n) = \Theta(n)$.

2) If $a = b$, then $T(n) = \Theta(n \log n)$.

3) If $a > b$, then $T(n) = \Theta(n^{\log_b a})$.

Multiplication: Divide-and-Conquer

In $T(n) = 4 \cdot T\left(\frac{n}{2}\right) + O(n)$ we have: $a = 4, b = 2$.

Thus: $T(n) = \Theta\left(n^{\log_b a}\right) = \Theta\left(n^{\log_2 4}\right) = \Theta(n^2)$.

This version of Divide-and-Conquer algorithm does not reduce the overall time complexity. ☹

But the Master Theorem gives us the cue: to reduce time complexity, we need to reduce the number of *multiplications*, even at the expense of increasing the number of *additions*.

Multiplication: Karatsuba's algorithm

Standard multiplication scheme:

$$z = x \cdot y = \left(10^{\frac{n}{2}}a + b\right) \cdot \left(10^{\frac{n}{2}}c + d\right) = 10^n ac + 10^{\frac{n}{2}}(ad + bc) + bd.$$

We have 4 multiplications and 3 additions.

Karatsuba's multiplication scheme:

$$z = x \cdot y = 10^n ac + 10^{\frac{n}{2}}\big((a + b)(c + d) - ac - bd\big) + bd =$$
$$10^n ac + 10^{\frac{n}{2}}\big((a + b)(c + d) - ac - bd\big) + bd.$$

This scheme has 3 multiplications and 6 additions/subtractions.

Multiplication: Karatsuba’s algorithm

The time complexity of Karatsuba’s algorithm:

$T(n) = 3 \cdot T\left(\frac{n}{2}\right) + O(n).$

Thus: $a = 3, b = 2$ => $T(n) = \Theta\left(n^{\log_b a}\right) = \Theta\left(n^{\log_2 3}\right) = \Theta(n^{1.58496})$.

Fast exponentiation

After the problem of multiplication, let us consider the exponentiation problem.
Problem: given integers x and n, calculate $y = x^n$.

The naïve algorithm:

```
y = x;
for i=2 to n:
    y = y · x;
return y.
```

Time complexity: $O(n)$ multiplications.

Fast exponentiation

Time complexity of the naïve algorithm: $O(n)$ multiplications.

For numeric exponentiation, the complexity of one multiplication operation grows with the size of y. But there are many practical problems for which the complexity of a multiplication operation depends on the size of x only: matrix multiplication, multiplication in modular arithmetic, etc.

Can we calculate powers with less than $O(n)$ multiplications?

Fast exponentiation

Let's consider binary representation of *n*:

$$n = \sum_{i=0}^{L-1} n_i \cdot 2^i ,$$

where $L = [\log_2 n]$, $n_i \in \{0,1\}$.

Thus,

$$x^n = x^{\sum_{i=0}^{L-1} n_i \cdot 2^i} = \prod_{i=0}^{L-1} x^{n_i \cdot 2^i} = \prod_{n_i=1} x^{2^i}$$

This expression contains only $L = \log_2 n$ operations of *squaring*.

Fast exponentiation

Let's consider binary representation of *n*:

$$n = \sum_{i=0}^{L-1} n_i \cdot 2^i \,,$$

where $L = [\log_2 n]$, $n \in \{0,1\}$.

Thus,

$$x^n = x^{\sum_{i=0}^{L-1} n_i \cdot 2^i} = \prod_{i=0}^{L-1} x^{n_i \cdot 2^i} = \prod_{n_i=1} x^{2^i}$$

This expression contains only $L = \log_2 n$ operations of *squaring*.

Fast exponentiation

$$x^n = \prod_{n_i=1} x^{2^i}$$

The ‘exponentiation by squaring’ algorithm:

```
    y = 1;
    s = x;
    for i=0 to L-1:
        if n_i = 1 then y = y · s;
        s = s · s;
    return y.
```

Time complexity: $O(\log n)$ multiplications.

Fast exponentiation

We can rewrite this algorithm in a recursive form:

```
FastExponentiationRecursive(x,n)
if n=1 than return x;
   else
         s = FastExponentiationRecursive(x,⌊n/2⌋);
         if n is even than return s·s;
              else return s·s·x.
```

Time complexity: $O(\log n)$ multiplications.

Matrix multiplication

Let us consider one more multiplication problem.
Problem: given two $n \times n$ matrices X and Y, calculate their dot-product $Z = X \cdot Y$.

Direct calculation

$$z_{ij} = \sum_{k=1}^{n} x_{ik} \cdot y_{kj}$$

needs $O(n^3)$ time (n^2 entries, each entry is calculated in $O(n)$ time).

Matrix multiplication

Let us apply the 'Divide-and-Conquer' approach.
We represent both tables in block form:

$$X = \begin{pmatrix} A & B \\ C & D \end{pmatrix}, Y = \begin{pmatrix} E & F \\ G & H \end{pmatrix}$$

where $A, \ldots, H$ are $\frac{n}{2} \times \frac{n}{2}$ matrices.

The direct formula leads to the form

$$\mathbf{X} \cdot \mathbf{Y} = \begin{pmatrix} \mathbf{A} \cdot \mathbf{E} + \mathbf{B} \cdot \mathbf{G} & \mathbf{A} \cdot \mathbf{F} + \mathbf{B} \cdot \mathbf{H} \\ \mathbf{C} \cdot \mathbf{E} + \mathbf{D} \cdot \mathbf{G} & \mathbf{C} \cdot \mathbf{F} + \mathbf{D} \cdot \mathbf{H} \end{pmatrix}$$

which also has $O(n^3)$ time complexity, since $T(n) = 8 \cdot T\left(\frac{n}{2}\right) + O(n)$.

Matrix multiplication: Strassen algorithm

Strassen algorithm calculates block matrix multiplication using 7 (instead of 8) submatrix multiplications:

$$\mathbf{P}_1 = \mathbf{A} \cdot (\mathbf{F} - \mathbf{H})$$
$$\mathbf{P}_2 = (\mathbf{A} + \mathbf{B}) \cdot \mathbf{H}$$
$$\mathbf{P}_3 = (\mathbf{C} + \mathbf{D}) \cdot \mathbf{E}$$
$$\mathbf{P}_4 = \mathbf{D} \cdot (\mathbf{G} - \mathbf{E})$$
$$\mathbf{P}_5 = (\mathbf{A} + \mathbf{D}) \cdot (\mathbf{E} + \mathbf{H})$$
$$\mathbf{P}_6 = (\mathbf{B} - \mathbf{D}) \cdot (\mathbf{G} + \mathbf{H})$$
$$\mathbf{P}_7 = (\mathbf{A} - \mathbf{C}) \cdot (\mathbf{E} + \mathbf{F}).$$

$$\mathbf{X} \cdot \mathbf{Y} = \left(\begin{array}{c|c} \mathbf{A} \cdot \mathbf{E} + \mathbf{B} \cdot \mathbf{G} & \mathbf{A} \cdot \mathbf{F} + \mathbf{B} \cdot \mathbf{H} \\ \hline \mathbf{C} \cdot \mathbf{E} + \mathbf{D} \cdot \mathbf{G} & \mathbf{C} \cdot \mathbf{F} + \mathbf{D} \cdot \mathbf{H} \end{array}\right) = \left(\begin{array}{c|c} \mathbf{P}_5 + \mathbf{P}_4 - \mathbf{P}_2 + \mathbf{P}_6 & \mathbf{P}_1 + \mathbf{P}_2 \\ \hline \mathbf{P}_3 + \mathbf{P}_4 & \mathbf{P}_1 + \mathbf{P}_5 - \mathbf{P}_3 - \mathbf{P}_7 \end{array}\right)$$

Matrix multiplication: Strassen algorithm

The complexity of Strassen algorithm can be expressed as

$$T(n) = 7 \cdot T\left(\frac{n}{2}\right) + O(n).$$

Thus, $T(n) = O\left(n^{\log_2 7}\right) = O(n^{2.8074})$.