CS 260
Design and Analysis of Algorithms
2. Divide and Conquer Algorithms

Mikhail Moshkov

Computer, Electrical and Mathematical Sciences & Engineering Division
King Abdullah University of Science and Technology

Divide and Conquer Algorithm

We divide the problem into two or more subproblems (ideally of approximately equal sizes), solve each subproblem recursively, and combine solutions of subproblems into a global solution.

The example of divide and conquer algorithm was considered in the previous section. It is merge sort. To analyze the complexity of this algorithms it was necessary for us to study a recurrence.

Now we describe a way for the work with some kind of recurrences.

Variant of Master Theorem

Theorem 2.1 (Master Theorem). *Let $L(n)$ be a function depending on natural n. Let c be a natural number, $c \geq 2$, a, b, γ be real constants, $a \geq 1, b > 0, \gamma \geq 0$, and for any $n = c^k$, where k is an arbitrary natural number, the following inequality holds:*

$$L(n) \leq aL\left(\frac{n}{c}\right) + bn^{\gamma}.$$

Let for any natural k for any $n \in \{c^k + 1, c^k + 2, \ldots, c^{k+1}\}$ the inequality $L(n) \leq L\left(c^{k+1}\right)$ holds. Then

$$L(n) = \begin{cases} O(n^{\gamma}), & \text{if } \gamma > \log_c a, \\ O(n^{\log_c a}), & \text{if } \gamma < \log_c a, \\ O(n^{\gamma} \log n) & \text{if } \gamma = \log_c a. \end{cases}$$

Variant of Master Theorem

Proof. Let $n = c^k$ where $k \in \{1, 2, 3, \ldots\}$. Using the inequality $L(n) \leq aL\left(\frac{n}{c}\right) + bn^\gamma$ we obtain

$$\begin{aligned}
L(n) &\leq aL\left(\frac{n}{c}\right) + bn^\gamma \leq a\left(aL\left(\frac{n}{c^2}\right) + b\left(\frac{n}{c}\right)^\gamma\right) + bn^\gamma \\
&= bn^\gamma + ab\left(\frac{n}{c}\right)^\gamma + a^2 L\left(\frac{n}{c^2}\right) \\
&\leq bn^\gamma + b\left(\frac{a}{c^\gamma}\right) n^\gamma + a^2\left(aL\left(\frac{n}{c^3}\right) + b\left(\frac{n}{c^2}\right)^\gamma\right) \\
&= bn^\gamma + bn^\gamma\left(\frac{a}{c^\gamma}\right) + bn^\gamma\left(\frac{a}{c^\gamma}\right)^2 + a^3 L\left(\frac{n}{c^3}\right) \leq \ldots \\
&\leq bn^\gamma + bn^\gamma\left(\frac{a}{c^\gamma}\right) + \ldots + bn^\gamma\left(\frac{a}{c^\gamma}\right)^{k-1} + a^k L\left(\frac{n}{c^k}\right).
\end{aligned}$$

Variant of Master Theorem

Let $d = \max(b, L(1))$. Since $\frac{n}{c^k} = 1$, we have

$$\begin{aligned} L(n) &\leq dn^\gamma \left(1 + \frac{a}{c^\gamma} + \left(\frac{a}{c^\gamma}\right)^2 + \ldots + \left(\frac{a}{c^\gamma}\right)^{k-1}\right) + da^k \\ &= dn^\gamma \left(1 + \frac{a}{c^\gamma} + \left(\frac{a}{c^\gamma}\right)^2 + \ldots + \left(\frac{a}{c^\gamma}\right)^{k}\right). \end{aligned}$$

Variant of Master Theorem

We will use the following fact about geometric series. Let α be a real number and $0 < \alpha < 1$. Then, for any n,

$$\sum_{i=0}^{n} \alpha^i = 1 + \alpha + \ldots + \alpha^n = \frac{1 - \alpha^n}{1 - \alpha} < \frac{1}{1 - \alpha}.$$

Variant of Master Theorem

Let us consider three cases.

1. Let $\gamma > \log_c a$. Then $\frac{a}{c^\gamma} < 1$.

In this case $L(n) \leq dn^\gamma const_1 = p_1 n^\gamma$ for some positive constant p_1.

Variant of Master Theorem

2. Let $\gamma < \log_c a$. Then $\frac{a}{c^\gamma} > 1$ and

$$L(n) \leq dn^\gamma \left(\frac{a}{c^\gamma}\right)^k \left(1 + \frac{c^\gamma}{a} + \left(\frac{c^\gamma}{a}\right)^2 + \ldots + \left(\frac{c^\gamma}{a}\right)^k\right).$$

Since $n = c^k$, we have $L(n) \leq da^k const_2 = p_2 a^k$ for some positive constant p_2. Therefore

$$L(n) \leq p_2 a^k = p_2 a^{\log_c n} = p_2 n^{\log_c a}.$$

Variant of Master Theorem

3. Let $\gamma = \log_c a$. Then $\frac{a}{c^\gamma} = 1$ and

$$L(n) \leq dn^\gamma (k+1) = dn^\gamma (1 + \log_c n) \leq 2dn^\gamma \log_c n$$

for $n \geq c$.

Let us denote $p_3 = 2d$. Then $L(n) \leq p_3 n^\gamma \log_c n$ for $n \geq c$.

Variant of Master Theorem

Let now n be an arbitrary natural number and $n > c$. Then there exists a natural k such that $c^k < n \leq c^{k+1}$.

Let us consider three cases taking into account that the inequality $L(n) \leq L\left(c^{k+1}\right)$ holds.

Variant of Master Theorem

1. Let $\gamma > \log_c a$. Then

$$L(n) \leq L(c^{k+1}) \leq p_1 \left(c^{k+1}\right)^{\gamma} = p_1 c^{\gamma} \left(c^{k}\right)^{\gamma} \leq p_1 c^{\gamma} n^{\gamma}.$$

Thus, $L(n) = O(n^{\gamma})$.

Variant of Master Theorem

2. Let $\gamma < \log_c a$. Then

$$L(n) \leq L(c^{k+1}) \leq p_2 \left(c^{k+1}\right)^{\log_c a} = p_2 c^{\log_c a} \left(c^k\right)^{\log_c a} \leq p_2 a n^{\log_c a}.$$

Thus, $L(n) = O(n^{\log_c a})$.

Variant of Master Theorem

3. Let $\gamma = \log_c a$. Then

$$L(n) \leq L(c^{k+1}) \leq p_3 c^{(k+1)\gamma} \log_c(c^{(k+1)})$$
$$\leq p_3 c^{\gamma} \left(c^k\right)^{\gamma} (k+1) \leq p_3 c^{\gamma} n^{\gamma} (1 + \log_c n) \leq 2 p_3 c^{\gamma} n^{\gamma} \log_c n.$$

Thus, $L(n) = O(n^{\gamma} \log n)$. □

Variant of Master Theorem

Remark 2.2. *If in Theorem 2.1 instead of the inequality* $L(n) \leq aL\left(\frac{n}{c}\right) + bn^{\gamma}$ *we have the inequality* $L(n) \leq aL\left(\frac{n}{c}\right) + O(n^{\gamma})$ *the statement of the theorem will be also true.*

Let $f(n) = O(n^{\gamma})$. It means that there exist positive constants c_0 and n_0 such that $f(n) \leq c_0 n^{\gamma}$ for $n \geq n_0$.

Since $n^{\gamma} > 0$ for any n, there exists a constant $B > 0$ such that $f(n) \leq Bc_0 n^{\gamma}$ for any n. Set $b = Bc_0$. Then $f(n) \leq bn^{\gamma}$ for any n.

Variant of Master Theorem

Example 2.3. We proved for time complexity of merge sort that $L_{mer}(n) \leq 2L_{mer}\left(\frac{n}{2}\right) + n$ for any $n = 2^k$, $k = 1, 2, 3, \ldots$.

So $a = c = 2$, $b = 1$ and $\gamma = 1$. We have $\gamma = \log_c a$. We know (see proof of Theorem 1.13) that $L_{mer}(n)$ is a nondecreasing function. By Theorem 2.1 (Master Theorem),

$$L_{mer}(n) = O(n \log n).$$

Integer Multiplication (Karatsuba, 1960, published in 1962)

We will work with n-bit integers. The operations of addition and substraction take linear time $O(n)$. The traditional multiplication technique requires $O(n^2)$ time. We will try to decrease time complexity of multiplication.

We will use the following notation to represent n-bit integers:

$$(x_0, x_1, \ldots, x_{n-1})_2 = x_0 2^{n-1} + x_1 2^{n-2} + \ldots + x_{n-1} 2^{n-n}.$$

Integer Multiplication

First, we will show that the multiplication of two $(n+1)$-bit integers can be reduced to the multiplication of two n-bit integers plus some additional operations in $O(n)$ time.

Let $X_1 = (x_0, x_1, \ldots, x_n)_2$ and $Y_1 = (y_0, y_1, \ldots, y_n)_2$ be two $(n+1)$-bit integers. Let us denote $X = (x_1, \ldots, x_n)_2$ and $Y = (y_1, \ldots, y_n)_2$. Then $X_1 = x_0 2^n + X$, $Y_1 = y_0 2^n + Y$ and

$$X_1 Y_1 = x_0 y_0 2^{2n} + (x_0 Y + y_0 X) 2^n + XY.$$

If we compute XY then to compute $X_1 Y_1$ it is enough to make additional operations that require $O(n)$ time.

Integer Multiplication

Let us assume that $n = 2^k$ for some natural k. Let we have two n-bit integers X and Y. We split X and Y into their left and right halves, which are $\frac{n}{2}$ bits long: $X = 2^{\frac{n}{2}} X_L + X_R$ and $Y = 2^{\frac{n}{2}} Y_L + Y_R$. Then

$$XY = \left(2^{\frac{n}{2}} X_L + X_R\right)\left(2^{\frac{n}{2}} Y_L + Y_R\right)$$
$$= 2^n X_L Y_L + 2^{\frac{n}{2}}(X_L Y_R + X_R Y_L) + X_R Y_R.$$

To compute $X_L Y_R + X_R Y_L$ we will use the following formula:

$$X_L Y_R + X_R Y_L = (X_L + X_R)(Y_L + Y_R) - X_L Y_L - X_R Y_R.$$

Integer Multiplication

To compute XY it is enough to make two multiplications of $\frac{n}{2}$-bit integers, and one multiplication of $\left(\frac{n}{2}+1\right)$-bit integers.

Also we should make a constant number of additions, substractions and multiplications by 2^t (digit shifts) for $t \leq n$ (each of these operations requires $O(n)$ time).

We know that the multiplication of two $\left(\frac{n}{2}+1\right)$-bit integers can be reduced to the multiplication of two $\frac{n}{2}$-bit integers plus additional operations that require $O(n)$ time.

Integer Multiplication

If $\frac{n}{2} > 1$ then we can continue this process recursively. If $\frac{n}{2} = 1$, then we apply the usual algorithm for multiplication of integers.

Let $L(n)$ be the time required for multiplication of two n-bit integers by the considered algorithm, and $n = 2^k$ for some natural k. Then we have

$$L(n) \leq 3L\left(\frac{n}{2}\right) + O(n).$$

Integer Multiplication

Let $2^k < n < 2^{k+1}$ for some natural k. We add $0 \ldots 0$ at the beginning of considered n-bit integers and obtain 2^{k+1}-bit integers. We apply our algorithm to these integers.

The result of multiplication will be the same as for the initial integers. So we have $L(n) \leq L\left(2^{k+1}\right)$.

Using Theorem 2.1 (Master Theorem) and Remark 2.2 we obtain

$$L(n) = O(n^{\log_2 3}).$$

Note that $\log_2 3 = 1.585\ldots$, and therefore $L(n) = O(n^{1.59})$.

Integer Multiplication

During the study of Karatsuba algorithm, one step means one logical operation with two bits.

This result was improved by Schöhage and Strassen in 1971. They obtained the bound $O(n \cdot \log n \cdot \log \log n)$.

Fürer's algorithm (2007) has $O(n \cdot \log n \cdot 2^{O(\log^* n)})$ time complexity.

Integer Multiplication

Example 2.4. Multiplication of two integers $X = 11112222$ and $Y = 33334444$.

We have $X = 10^4 X_L + X_R = 10^4 \times 1111 + 2222$,
$Y = 10^4 Y_L + Y_R = 10^4 \times 3333 + 4444$,

$$\begin{aligned} XY &= (10^4 X_L + X_R)(10^4 Y_L + Y_R) \\ &= 10^8 X_L Y_L + 10^4 (X_L Y_R + X_R Y_L) + X_R Y_R \\ &= 10^8 \times 1111 \times 3333 + 10^4 (1111 \times 4444 + 2222 \times 3333) \\ &\quad + 2222 \times 4444, \end{aligned}$$

and

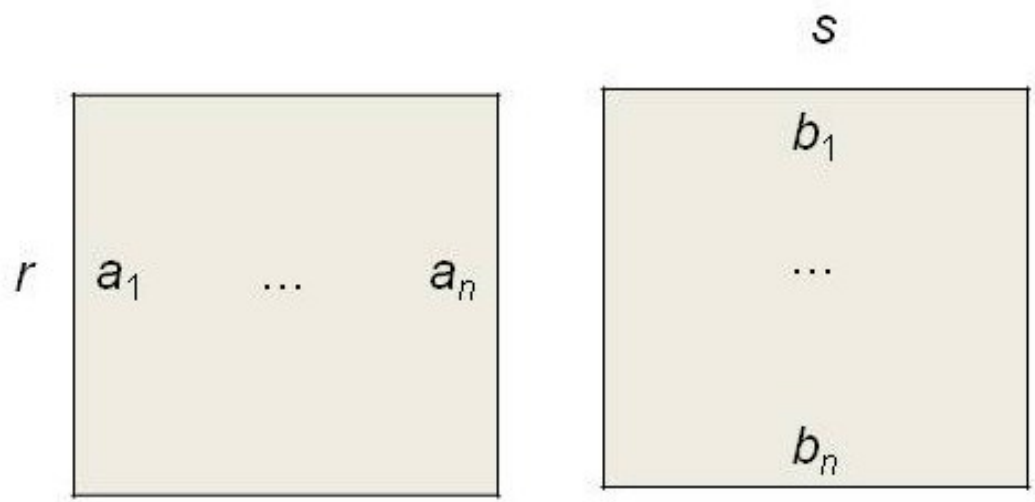
$$\begin{aligned} X_L Y_R + X_R Y_L &= (X_L + X_R)(Y_L + Y_R) - X_L Y_L - X_R Y_R \\ &= (1111 + 2222)(3333 + 4444) - 1111 \times 3333 \\ &\quad - 2222 \times 4444. \end{aligned}$$

Matrix Multiplication (Strassen, published in 1969)

In this section, we will study so-called arithmetic complexity of problem of multiplication of two matrices, where one step is an operation of addition, substraction, multiplication or division.

Matrix Multiplication

Let us consider the problem of multiplication of two $n \times n$ matrices $A = [a_{ij}]$ and $B = [b_{kl}]$. Let $A \cdot B = C = [c_{rs}]$. Then, by definition, $c_{rs} = \sum_{p=1}^{n} a_{rp} b_{ps}$.



$$c_{rs} = a_1 b_1 + \ldots + a_n b_n$$

Usual algorithm requires n^3 multiplications and $n^2(n-1)$ additions. So the complexity of usual algorithm is $O(n^3)$.

Matrix Multiplication

Theorem 2.5. *There exists an algorithm for multiplication of two $n \times n$ matrices with the number of arithmetical operations $O(n^{\log_2 7})$.*

Matrix Multiplication

Proof. Let $n = 2^k$ where k is a natural number, and A, B be matrices $2^k \times 2^k$.

Let us cut each of matrices A, B and C into four $2^{k-1} \times 2^{k-1}$ sub-matrices:

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

Matrix Multiplication

It is known that in this case we have

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21}, \quad C_{12} = A_{11}B_{12} + A_{12}B_{22}, \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21}, \quad C_{22} = A_{21}B_{12} + A_{22}B_{22}. \end{aligned}$$

Thus, the computation of C can be reduced to 8 multiplications of $\frac{n}{2} \times \frac{n}{2}$ matrices and 4 additions of $\frac{n}{2} \times \frac{n}{2}$ matrices.

The idea of Strassen consists of the use of only 7 multiplications.

Matrix Multiplication

Let us consider the following 7 products:

$$\begin{aligned} D_1 &= (A_{11} + A_{22})(B_{11} + B_{22}), \quad D_5 = (A_{11} + A_{12})B_{22}, \\ D_2 &= (-A_{11} + A_{21})(B_{11} + B_{12}), \quad D_6 = A_{22}(-B_{11} + B_{21}), \\ D_3 &= (A_{12} - A_{22})(B_{21} + B_{22}), \quad D_7 = (A_{21} + A_{22})B_{11}, \\ D_4 &= A_{11}(B_{12} - B_{22}). \end{aligned}$$

One can show that

$$\begin{aligned} C_{11} &= D_1 + D_3 - D_5 + D_6, \quad C_{12} = D_4 + D_5, \\ C_{21} &= D_6 + D_7, \quad C_{22} = D_1 + D_2 + D_4 - D_7. \end{aligned}$$

Matrix Multiplication

Thus, the problem of multiplication of two $n \times n$ matrices can be reduced to 7 multiplications and a fixed number of additions and subtractions of $\frac{n}{2} \times \frac{n}{2}$ matrices.

If $\frac{n}{2} > 1$ then we can continue this process recursively. If $\frac{n}{2} = 1$ then for multiplication of two 1×1 matrices it is enough to make one multiplication of elements. Since one addition or substraction of two $n \times n$ matrices requires $O(n^2)$ arithmetical operations, we obtain for $n = 2^k$, $k = 1, 2, \ldots$, that

$$L(n) \leq 7L\left(\frac{n}{2}\right) + O(n^2),$$

where $L(n)$ is the number of arithmetical operations required for multiplication of two $n \times n$ matrices by the considered algorithm.

Matrix Multiplication

Let $2^k < n < 2^{k+1}$ for some natural k. In this case we extend matrices A and B up to $2^{k+1} \times 2^{k+1}$ matrices A' and B' which contain A and B in the left upper corners. All other elements in A' and B' are equal to 0. One can show that $A' \cdot B' = C'$ where C' contains $C = A \cdot B$ in the left upper corner.

From here it follows that $L(n) \leq L\left(2^{k+1}\right)$. Using Theorem 2.1 (Master Theorem) and Remark 2.2 we obtain

$$L(n) = O(n^{\log_2 7}).$$

□

Matrix Multiplication

Note that $\log_2 7 = 2.807355\ldots$

Improvements:

1990, Coppersmith and Winograd, $O(n^{2.375477})$

2010, Stothers, $O(n^{2.373})$

2011, Williams, $O(n^{2.3728642})$

2014, Le Gall, $O(n^{2.3728639})$

Matrix Multiplication

Example 2.6. Multiplication of two matrices

$$A = \begin{bmatrix} a_{11}=1 & a_{12}=2 \\ a_{21}=3 & a_{22}=4 \end{bmatrix} \quad B = \begin{bmatrix} b_{11}=5 & b_{12}=6 \\ b_{21}=7 & b_{22}=8 \end{bmatrix}$$

$$C = A \times B = \begin{bmatrix} c_{11}=19 & c_{12}=22 \\ c_{21}=43 & c_{22}=50 \end{bmatrix}$$

$$\begin{aligned} c_{11} &= a_{11}b_{11} + a_{12}b_{21} = 1 \times 5 + 2 \times 7 = 19 \\ c_{12} &= a_{11}b_{12} + a_{12}b_{22} = 1 \times 6 + 2 \times 8 = 22 \\ c_{21} &= a_{21}b_{11} + a_{22}b_{21} = 3 \times 5 + 4 \times 7 = 43 \\ c_{22} &= a_{21}b_{12} + a_{22}b_{22} = 3 \times 6 + 4 \times 8 = 50 \end{aligned}$$

Matrix Multiplication

Example 2.6 (continuation).

$$A = \begin{bmatrix} a_{11} = 1 & a_{12} = 2 \\ a_{21} = 3 & a_{22} = 4 \end{bmatrix} \quad B = \begin{bmatrix} b_{11} = 5 & b_{12} = 6 \\ b_{21} = 7 & b_{22} = 8 \end{bmatrix}$$

$$\begin{aligned}
d_1 &= (a_{11} + a_{22})(b_{11} + b_{22}) = (1 + 4)(5 + 8) = 65 \\
d_2 &= (-a_{11} + a_{21})(b_{11} + b_{12}) = (-1 + 3)(5 + 6) = 22 \\
d_3 &= (a_{12} - a_{22})(b_{21} + b_{22}) = (2 - 4)(7 + 8) = -30 \\
d_4 &= a_{11}(b_{12} - b_{22}) = 1(6 - 8) = -2 \\
d_5 &= (a_{11} + a_{12})b_{22} = (1 + 2)8 = 24 \\
d_6 &= a_{22}(-b_{11} + b_{21}) = 4(-5 + 7) = 8 \\
d_7 &= (a_{21} + a_{22})b_{11} = (3 + 4)5 = 35
\end{aligned}$$

Matrix Multiplication

Example 2.6 (continuation).

$$d_1 = 65, d_2 = 22, d_3 = -30, d_4 = -2, d_5 = 24, d_6 = 8, d_7 = 35$$

$$\begin{aligned} c_{11} &= d_1 + d_3 - d_5 + d_6 = 65 - 30 - 24 + 8 = 19 \\ c_{12} &= d_4 + d_5 = -2 + 24 = 22 \\ c_{21} &= d_6 + d_7 = 8 + 35 = 43 \\ c_{22} &= d_1 + d_2 + d_4 - d_7 = 65 + 22 - 2 - 35 = 50 \end{aligned}$$

$$C = A \times B = \begin{bmatrix} c_{11} = 19 & c_{12} = 22 \\ c_{21} = 43 & c_{22} = 50 \end{bmatrix}$$

Fast Fourier Transform

Let we have two polynomials $A(x) = a_0 + a_1 x + \ldots + a_{n-1} x^{n-1}$ and $B(x) = b_0 + b_1 x + \ldots + b_{n-1} x^{n-1}$ of degree $n-1$. Our aim is to compute their product $C(x) = A(x) \cdot B(x)$, $C(x) = c_0 + c_1 x + \ldots + c_{2n-2} x^{2n-2}$. The usual way requires $O(n^2)$ operations of addition and multiplication.

We consider *Fast Fourier Transform* (FFT) – an algorithm which computes $C(x)$ using $O(n \log n)$ operations. This algorithm is based on the following fundamental fact about polynomials: any polynomial of degree d can be reconstructed from its values on any set of $d+1$ or more points.

Since $C(x)$ has degree at most $2n-2$, it can be reconstructed from its values on $2n$ points.

Fast Fourier Transform

FFT contains three steps:

(i) First we choose $2n$ values $x_1, x_2, \ldots, x_{2n}$ and evaluate $A(x_j)$ and $B(x_j)$ for each of $j = 1, \ldots, 2n$.

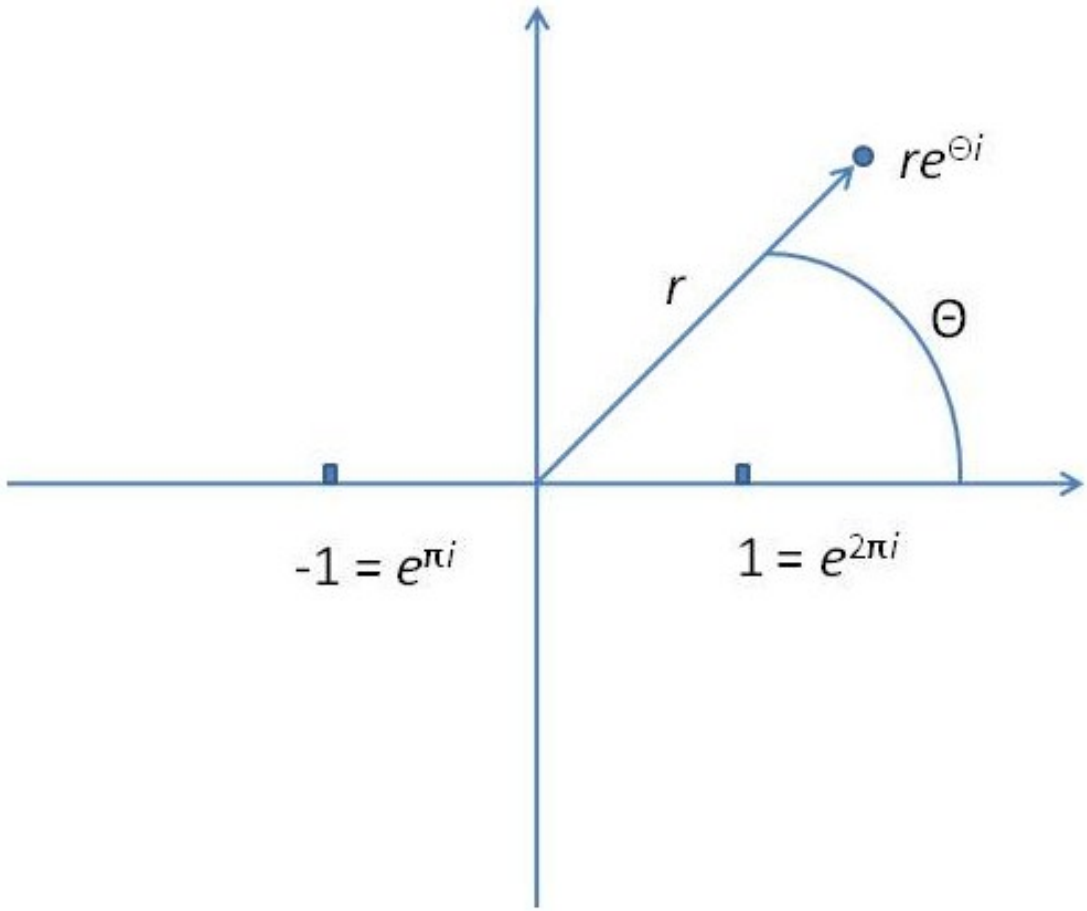
(ii) We can now compute $C(x_j) = A(x_j) \cdot B(x_j)$ for each j.

(iii) Finally, we recover C from values $C(x_1), \ldots, C(x_{2n})$.

The step (ii) requires $O(n)$ operations of multiplication. Now we consider the step (i).

Fast Fourier Transform

We will work with complex numbers, which can be considered as points in the "complex plane" with axes representing their real an imaginary parts. We can write a complex number using polar coordinates with respect to this plane as $re^{\Theta i}$. For example, $e^{\pi i} = -1$ and $e^{2\pi i} = 1$.



Fast Fourier Transform

Let k be a natural number. Then the equation $x^k = 1$ has k distinct complex roots

$$\omega_{j,k} = e^{\frac{2\pi ji}{k}},$$

$j = 0, 1, \ldots, k-1$. We will say about these numbers as about kth *roots of unity*.

In the capacity of $x_1, \ldots, x_{2n}$ we will use the $(2n)$th roots of unity $\omega_{0,2n}, \ldots, \omega_{2n-1,2n}$.

Fast Fourier Transform

We now describe an algorithm for the step (i).

Let n be power of 2. We represent $A(x)$ in the following way: $A(x) = A_{even}(x^2) + xA_{odd}(x^2)$ where

$$\begin{aligned} A_{even}(x) &= a_0 + a_2 x + a_4 x^2 + \ldots + a_{n-2} x^{\frac{n-2}{2}}, \\ A_{odd}(x) &= a_1 + a_3 x + a_5 x^2 + \ldots + a_{n-1} x^{\frac{n-2}{2}}. \end{aligned}$$

Fast Fourier Transform

If we know values of A_{even} and A_{odd} on $(\omega_{j,2n})^2$, $j = 0, \ldots, 2n-1$ then it will be enough $4n$ operations of addition and multiplication to evaluate $A(\omega_{j,2n})$, $j = 0, \ldots, 2n-1$.

Note that for $j = 0, \ldots, 2n-1$, $(\omega_{j,2n})^2 = e^{\frac{2\pi ji}{n}} = e^{\frac{2\pi li}{n}}$ where $l \equiv j \mod n$. It means that we should evaluate A_{even} and A_{odd} on $n$ht roots of unity (instead of squares of $(2n)$th roots of unity).

Fast Fourier Transform

Let $L(n)$ be the number of operations of multiplication and addition required to evaluate a polynomial A of degree $n-1$ on $(2n)$th roots of unity by the considered algorithm. Then $L(n) \leq 2L\left(\frac{n}{2}\right) + 4n$ if n is a power of 2.

Let now $2^k < n < 2^{k+1}$ for some natural k. In this case we represent the polynomial $A(x) = a_0 + a_1 x + \ldots + a_{n-1}x^{n-1}$ in the form $A(x) = a_0 + a_1 x + \ldots + a_{n-1}x^{n-1} + 0x^n + \ldots + 0x^{2^{k+1}-1}$ and apply to it the considered algorithm. So we have $L(n) \leq L\left(2^{k+1}\right)$.

Using Theorem 2.1 (Master Theorem) we obtain $L(n) = O(n \log n)$. Thus the step (i) requires $2L(n) = O(n \log n)$ operations of multiplication and addition.

Fast Fourier Transform

Let us consider now the step (iii): we should recover $C(x) = \sum_{s=0}^{2n-1} c_s x^s$ from values $d_0 = C(\omega_{0,2n}), \ldots, d_{2n-1} = C(\omega_{2n-1,2n})$.

Let us define a new polynomial $D(x) = \sum_{s=0}^{2n-1} d_s x^s$. We now consider the value of $D(x)$ at $(2n)$th roots of unity:

$$\begin{aligned} D(\omega_{j,2n}) &= \sum_{s=0}^{2n-1} C(\omega_{s,2n})\omega_{j,2n}^s = \sum_{s=0}^{2n-1} \left(\sum_{t=0}^{2n-1} c_t \omega_{s,2n}^t \right) \omega_{j,2n}^s \\ &= \sum_{t=0}^{2n-1} c_t \left(\sum_{s=0}^{2n-1} \omega_{s,2n}^t \omega_{j,2n}^s \right). \end{aligned}$$

Fast Fourier Transform

We know that $\omega_{s,2n} = \left(e^{\frac{2\pi i}{2n}}\right)^s$. We extend this notation even to the case when $s \geq 2n$.

Then

$$D(\omega_{j,2n}) = \sum_{t=0}^{2n-1} c_t \left(\sum_{s=0}^{2n-1} e^{\frac{2\pi i(st+js)}{2n}} \right) = \sum_{t=0}^{2n-1} c_t \left(\sum_{s=0}^{2n-1} \omega_{t+j,2n}^s \right).$$

Fast Fourier Transform

Let us consider the sum $\sum_{s=0}^{2n-1} \omega_{t+j,2n}^{s}$. We denote $\omega = \omega_{t+j,2n}$. Let $l \in \{0, 1, \ldots, 2n-1\}$ and $l \equiv t + j \mod 2n$.

If $l = 0$, then $\omega = 1$ and $\sum_{s=0}^{2n-1} \omega^s = 2n$. Let $\omega \neq 1$. It is clear that $x^{2n} - 1 = (x-1)\left(\sum_{s=0}^{2n-1} x^s\right)$. Since $\omega^{2n} - 1 = 0$ and $\omega - 1 \neq 0$, we have $\sum_{s=0}^{2n-1} \omega^s = 0$.

Therefore $D(\omega_{j,2n}) = 2nc_{2n-j}$ and $c_{2n-j} = \frac{D(\omega_{j,2n})}{2n}$.

Fast Fourier Transform

So, if we evaluate D at the $(2n)$th roots of unity $\omega_{2n,2n} = \omega_{0,2n}, \omega_{2n-1,2n}, \omega_{2n-2,2n}, \ldots, \omega_{1,2n}$ we will obtain values $c_0 2n, c_1 2n, \ldots, c_{2n-1} 2n$.

We can do all evaluations in $O(n \log n)$ operations of multiplication and addition using divide-and-conquer approach developed for the step (i). To find $c_0, \ldots, c_{2n-1}$ we need $O(n)$ operations of division.

Thus FFT requires $O(n \log n)$ operations of addition, multiplication and division.