Algorithms Lecture Notes
UC Berkeley

Kelvin Lee

September 10, 2020

These are course notes for UC Berkeley's CS170 Efficient Algorithms and Intractable Problems, instructed by Professor Avishay Tal and Umesh Vazirani.

Contents

§1 Introduction

§1.1 Asymptotic Notation

Let $f, g : \mathbb{R} \to \mathbb{R}$ for the following definitions.

§1.1.1 O Notation

The O notation describes upper bounds for function.

Definition 1.1 (O Notation)
If there exists a constant $c > 0$ and N such that for $x > N, |f(x)| \leq c|g(x)|$, we say that

$$f(x) = \mathcal{O}(g(x)).$$

§1.1.2 Ω Notation

The Ω notation describes lower bounds for functions.

Definition 1.2 (Ω Notation)
If there exists a constant $c > 0$ and N such that for $x > N, |f(x)| \geq c|g(x)|$. This indicates that

$$f(x) = \Omega(g(x))$$

§1.1.3 Θ Notation

The Θ notation describes both upper and lower bounds for functions.

Definition 1.3 (Θ Notation)
If there exist constants $c_1, c_2 > 0$, and N such that $c_1 g(x) \leq f(x) \leq c_2 g(x)$ for $x > N$, we have $f(x) = O(g(x)) = \Omega(g(x))$, which implies

$$f(x) = \Theta(g(x)).$$

Theorem 1.4 (Asymptotic Limit Rules)
If $f(n), g(n) \geq 0$:

- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \implies f(n) = \mathcal{O}(g(n))$.
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$ for some $c > 0 \implies f(n) = \Theta(g(n))$.
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \implies f(n) = \Omega(g(n))$.

§2 Divide and Conquer

Definition 2.1 (Divide and Conquer)
The **divide-and-conquer** strategy solves a problem by:

1. Breaking it into subproblems that are themselves smaller instances of the same type of problem.
2. Recursively solving these subproblems.
3. Appropriately combining their answers.

§2.1 Karatsuba's Algorithm

Suppose we are multiplying two n-bit integers, x and y. Split up x into half, x_L and x_R, so that $x = 2^{n/2}x_{\mathrm{L}} + x_{\mathrm{R}}$.

$$\begin{aligned} xy &= \left(2^{n/2}x_L + x_R\right) + \left(2^{n/2}y_L + y_R\right) \\ &= 2^n\left(x_Ly_L\right) + 2^{n/2}\left(x_Ry_L + x_Ly_R\right) + x_Ry_R \end{aligned}$$

- The additions take linear time, as do the multiplications by powers of 2 (left-shifts).
- It requires 4 multiplications on $n/2$ bit numbers, we get the recurrence relation

$$T(n) = 4T(n/2) + \mathcal{O}(n).$$

However, this can be improved by using 3 multiplications:

- only need x_Ly_L, x_Ry_R, and $\left(x_L + x_R\right)\left(y_L + y_R\right)$ because

$$x_Ly_R + x_Ry_L = \left(x_L + x_R\right)\left(y_L + y_R\right) - x_Ly_L - x_Ry_R.$$

- The improved running time would then be

$$T(n) = 3T(n/2) + \mathcal{O}(n).$$

- The constant factor improvement occurs at every level of the recursion, which dramatically lowers time bound of $O\left(n^{\log_2 3}\right)$.

Algorithm 2.2 Karatsuba's Algorithm

```
function MULT($x, y$)
    $P_1 \leftarrow \text{Mult}(x_L, y_R)$
    $P_2 \leftarrow \text{Mult}(x_R, y_L)$
    $P_3 \leftarrow \text{Mult}(x_L + x_R, y_L + y_R)$
    return $2^nP_1 + 2^{n/2}\left(P_3 - P_1 - P_2\right) + P_3$
```

§2.2 Master Theorem

A divide-and-conquer algorithm might be described by

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

Theorem 2.3 (Master Theorem)
If $T(n) = aT(n/b) + \mathcal{O}\left(n^d\right)$ for $a > 0, b > 1$, and $d \geq 0$, then

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

Proof. Let $T(n) = aT(n/b) + \mathcal{O}\left(n^d\right)$. For simplicity, assume that $T(1) = 1$ and that n is a power of b. From the definition of big-O, we know that there exists constant $c > 0$ such that for sufficiently large $n, T(n) \leq aT(n/b) + cn^d$.

Suppose we have a recursive tree with $\log_b n + 1$ level. Consider level j. At level j, there are a^j subproblems. Each of size $\frac{n}{b^j}$, and will take time at most $c\left(\frac{n}{b^j}\right)^d$ to solve (this only considers the work done at level j and does not include the time it takes to solve the subsubproblems). Then the total work done at level j is at most $a^j \cdot c\left(n/b^j\right)^d = cn^d\left(\frac{a}{b^d}\right)^j$, where a is the branching factor and b^d is the shrinkage in the work needed (per subproblem). Summing over all levels, the total running time is at most $cn^d \sum_{j=0}^{\log_b n}\left(\frac{a}{b^d}\right)^j$. Consider each of the three cases:

1. $(a < b^d)$: $\frac{a}{b^d} < 1$, then
$$\sum_{j=0}^{\log_b n}\left(\frac{a}{b^d}\right)^j \leq \sum_{j=0}^{\infty}\left(\frac{a}{b^d}\right)^j = \frac{1}{1-\frac{a}{b^d}} = \frac{b^d}{b^d - a}.$$
Hence, $T(n) = cn^d \cdot \frac{b^d}{b^d-a} = O\left(n^d\right)$. Intuitively, in this case the shrinkage in the work needed per subproblem is more significant, so the work done in the highest level "dominates" the other factors in the running time.

2. $(a = b^d)$: The amount of work done at each level is the same: cn^d. since there are $\log_b n$ levels, $T(n) = (\log_b n + 1)\, cn^d = O\left(n^d \log n\right)$.

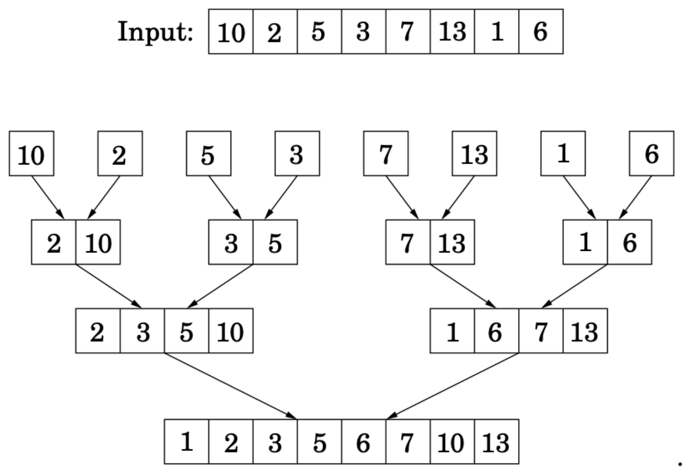
3. $(a > b^d)$: We have
$$\sum_{j=0}^{\log_b n}\left(\frac{a}{b^d}\right)^j = \frac{\left(\frac{a}{b^d}\right)^{\log_b n+1} - 1}{\frac{a}{b^d} - 1}.$$
Since a, b, c, d are constants,
$$T(n) = O\left(n^d \cdot \left(\frac{a}{b^d}\right)^{\log_b n}\right) = O\left(n^d \cdot \frac{a^{\log_b n}}{b^d \log_b n}\right) = O\left(n^d \cdot \frac{n^{\log_b a}}{n^d}\right) = O\left(n^{\log_b a}\right)$$
Intuitively, here the branching factor is more significant, so the total work done at each level increases, and the leaves of the tree "dominate".

□

§2.3 Mergesort

One great example of a divide-and-conquer problem is mergesort: splitting the list into two halves, recursively sort each half, and then merge the two sorted sublists.

Figure 2.4 The sequence of merge operations in `mergesort`.



```
function merge($x[1\dots k], y[1\dots l]$)
if $k = 0$:   return $y[1\dots l]$
if $l = 0$:   return $x[1\dots k]$
if $x[1] \le y[1]$:
  return $x[1] \circ$ merge($x[2\dots k], y[1\dots l]$)
else:
  return $y[1] \circ$ merge($x[1\dots k], y[2\dots l]$)
```

The merging part takes $\mathcal{O}(n)$ time, so the overall runtime is

$$T(n) = 2T(n/2) + \mathcal{O}(n) \implies T(n) = \mathcal{O}(n \log n).$$

§2.4 Inversions Counting

Now let's consider the problem of counting the number of inversions in a list.

Definition 2.4 (Inversion)
Two elements $a[i]$ and $a[j]$ form an inversion if $a[i] > a[j]$ and $i < j$.

We are interested in the number of inversions for several reasons:

- it tells us how sorted the list is,
- we can measure how similar two lists are by their number of inversions.

Remark 2.5. The maximum number of inversions in a list of length n is $\binom{n}{2}$ (why?).

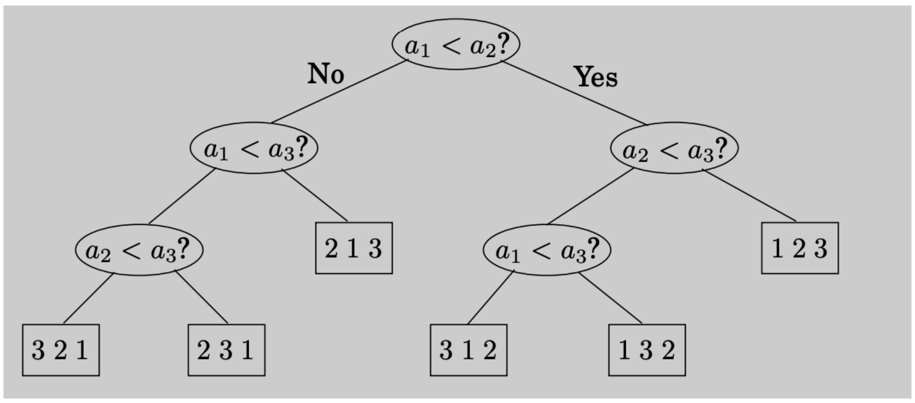
Now the question is how do we solve this by divide and conquer?

As usual, we first split the list into two halves. Then we perform two operations recursively on each half: sorting and counting. Thus, each recursive call should return the sorted half and the number of inversion in that sorted sublist. When merging the two halves, we keep track of the number of skipped elements of the right half when inserting the elements from the left half. We then sum them up to get the total number of inversions. Similar to mergesort, this merging process also takes $\mathcal{O}(n)$ time. Hence, we still have the following runtime:

$$T(n) = 2T(n/2) + \mathcal{O}(n) \implies T(n) = \mathcal{O}(n \log n).$$

Now consider another question: can we do better that $\mathcal{O}(n \log n)$?

- Consider the array $a[1, \ldots, n]$ as an abstraction.
- Imagine a decision-tree-like structure where we compare $a[i], a[j]$ for each $i, j \in 1, \ldots, n$ and $i < j$.



- In order for the algorithm to work, we need each leaf to contain exactly one permutation. Hence, the number of leaves must be at least $n!$.
- The maximum number of comparisons (height of tree) we execute is $\log n!$, which is also equivalent to the worst-case time complexity. Thus, the worst-case runtime of the algorithm would be $\Omega(\log n!)$.
- By Stirling's Approximation, we get

$$\Omega(\log n!) = \Omega\left(\log \left(\frac{n}{e}\right)^n\right) = \Omega(n \log n).$$

Remark 2.6. Recall that the Stirling's formula is

$$n! \approx \sqrt{\pi \left(2n + \frac{1}{3}\right)} \cdot n^n \cdot e^{-n}.$$

§2.5 Order Statistics (Median Finding)

Median finding is important in many situations, including database tasks. A precise statement of the problem is as follows:

Given an array $a[1, ..., n]$ of n numbers and an index k $(1 \leq k \leq n)$, find the kth smallest element of a.

One obvious way to do so is to sort the list using mergesort and then select the ith element. This would take $\mathcal{O}(n \log n)$. Here comes the question again: can we do even better?

Another approach to this problem would be using divide-and-conquer again. Here's how:

- We randomly select an element x from between $i = \frac{n}{4}$ and $i = \frac{3n}{4}$.
- Then we split the list into three categories: elements smaller than x , those equal to x (including duplicates), and those greater than x.
- The search can instantly be narrowed down to one of these sublists and we can check quickly which of these contain the median by checking k against the sizes of these sublists.
- However, the choice of x would determine the efficiency of the algorithm.

We determine x to be good if it lies within the 25th to 75th percentile of the list because then we ensure that the sublists have size at most $\frac{3n}{4}$ so that the array shrinks substantially. So x has $1/2$ chance of being good, which implies after two split operations on average, the array will shrink to at most $3/4$ of its size. Let $T(n)$ be the expected runtime. Then we have

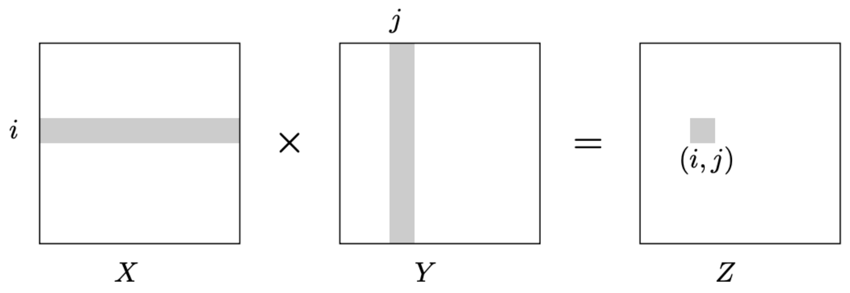
$$T(n) \leq T(3n/4) + \mathcal{O}(n) \implies T(n) = \mathcal{O}(n).$$

§2.6 Matrix Multiplication

The product of two $n \times n$ matrices X and Y is a third $n \times n$ matrix $Z = XY$, with (i, j) th entry

$$Z_{ij} = \sum_{k=1}^{n} X_{ik} Y_{kj}$$

To make it more visual, Z_{ij} is the dot product of the i th row of X with the j th column of Y :



The naive way indicates an $O\left(n^3\right)$ algorithm: there are n^2 entries to be computed, and each takes $O(n)$ time. What if we multiply using divide and conquer?

We can split the matrix blockwise. For example we can carve X into four $n/2 \times n/2$ blocks, and also Y :

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

Then their product can be expressed in terms of these blocks.

$$XY = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} AE+BG & AF+BH \\ CE+DG & CF+DH \end{bmatrix}$$

One divide and conquer approach is to compute the size-n product XY by recursively computing eight size-$n/22$ products $AE, BG, AF, BH, CE, DG, CF, DH$, and then do a few $O\left(n^2\right)$ time additions. The total runtime is given by the following:

$$T(n) = 8T(n/2) + O\left(n^2\right) \implies T(n) = \mathcal{O}(n^3).$$

It is still $O\left(n^3\right)$, the same as for the default algorithm. However, it turns out XY can be computed from just seven $n/2 \times n/2$ subproblems, here's how:

$$XY = \begin{bmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{bmatrix}$$

where

$$\begin{array}{llll} P_1 & = A(F-H) & P_5 = & (A+D)(E+H) \\ P_2 & = (A+B)H & P_6 = & (B-D)(G+H) \\ P_3 & = (C+D)E & P_7 = & (A-C)(E+F) \\ P_4 & = D(G-E) & & \end{array}$$

So our new runtime is now

$$T(n) = 7T(n/2) + O\left(n^2\right) \implies T(n) = O\left(n^{\log_2 7}\right) \approx O\left(n^{2.81}\right).$$

§2.7 Fast Fourier Transform

We see how divide-and-conquer produces fast algorithms for integers and matrices multiplication, but what about polynomials? Let's say we have the following:

$$\begin{aligned} C(x) &= A(x) \cdot B(x) \\ &= (a_0 + a_1 x + \ldots + a_{n-1} x^{n-1})(b_0 + b_1 x + \ldots + b_{n-1} x^{n-1}) \\ &= \sum_{i,j} a_i b_j x^{i+j}. \end{aligned}$$

This multiplication process is equivalent to a convolution of two vectors and requires $\mathcal{O}(n^2)$. How can we do better?

§2.7.1 Representation of polynomials

Let's first recall a quite important fact about polynomials.

Fact 2.7. *A degree-d polynomial is uniquely characterized by its values at any $d+1$ distinct points.*

One intuitive example of this is that any two points determine a line. Hence, any $d+1$ points $x_0, x_1, \ldots, x_d$ would determine a unique degree d polynomial. We can specify a degree-d polynomial $A(x) = a_0 + a_1 x + \ldots + a_d x^d$ with two representations:

1. Its coefficients $a_0, a_1, \ldots, a_d$.
2. The values $A\left(x_0\right), A\left(x_1\right), \ldots, A\left(x_d\right)$.

The second choice would be useful for polynomial multiplication because:

- the product $C(x)$ has degree $2d$, which can be determined by its value at any $2d+1$ points;
- the value at any given point z can be given by $C(z) = A(z) \cdot B(z)$. Thus polynomial multiplication takes linear time in the value representation.

However, we expect the input polynomials and the product to be specified by coefficients. So we need a translation from coefficients to values. In particular, we evaluate the polynomial at the chosen points, then multiply in the value representation, and finally translate back to coefficients, this process is known as **interpolation**.

$$\begin{tikzcd} \boxed{\begin{array}{l}\textbf{Coefficient representation}\\ a_0, a_1, \ldots, a_d\end{array}} \arrow[r, shift left, "\textbf{Evaluation}"] & \boxed{\begin{array}{l}\textbf{Value representation}\\ A(x_0), A(x_1), \ldots, A(x_d)\end{array}} \arrow[l, shift left, "\textbf{Interpolation}"] \end{tikzcd}$$

However, this approach actually takes quadratic time. In fact, we can actually do better by cleverly choosing the n points.

§2.7.2 Evaluation by divide-and-conquer

We choose them to be positive-negative pairs, that is,

$$\pm x_0, \pm x_1, \ldots, \pm x_{n/2-1}$$

then there will be overlaps in computations required for each $A\left(x_i\right)$ and $A\left(-x_i\right)$ because of the even powers terms. Let's explore deeper, we split $A(x)$ into its odd and even powers, for instance

$$3 + 4x + 6x^2 + 2x^3 + x^4 + 10x^5 = \left(3 + 6x^2 + x^4\right) + x\left(4 + 2x^2 + 10x^4\right)$$

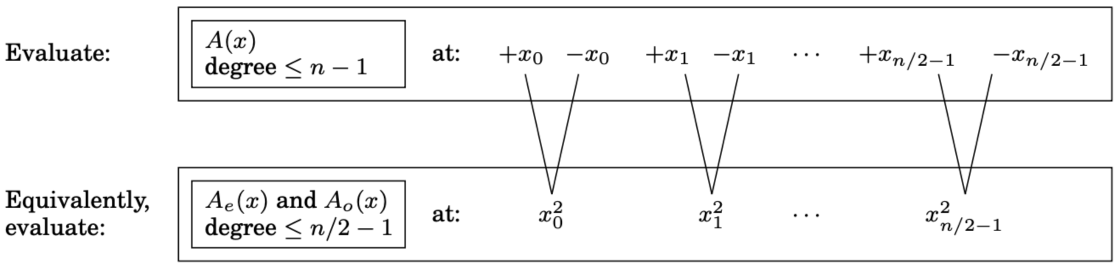
which can be written as

$$A(x) = A_e\left(x^2\right) + xA_o\left(x^2\right)$$

where $A_e(\cdot)$, with the even-numbered coefficients, and $A_o(\cdot)$, with the odd-numbered coefficients, are polynomials of degree $\leq n/2 - 1$ (assume for convenience that n is even). Given points $\pm x_i$, the calculations needed for $A\left(x_i\right)$ can be recycled toward computing $A\left(-x_i\right)$:

$$\begin{aligned} A\left(x_i\right) &= A_e\left(x_i^2\right) + x_i A_o\left(x_i^2\right) \\ A\left(-x_i\right) &= A_e\left(x_i^2\right) - x_i A_o\left(x_i^2\right) \end{aligned}$$

In other words, evaluating $A(x)$ at n paired points $\pm x_0, \ldots, \pm x_{n/2-1}$ boils down to evaluating $A_e(x)$ and $A_o(x)$ (each have half the degree of $A(x)$) at just $n/2$ points, $x_0^2, \ldots, x_{n/2-1}^2$.

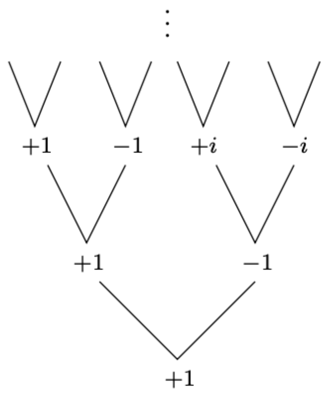

Evaluate:
$A(x)$ degree $\leq n-1$
at: $+x_0$ $-x_0$ $+x_1$ $-x_1$ $\cdots$ $+x_{n/2-1}$ $-x_{n/2-1}$
Equivalently, evaluate:
$A_e(x)$ and $A_o(x)$ degree $\leq n/2-1$
at: x_0^2 x_1^2 $\cdots$ $x_{n/2-1}^2$

The original problem of size n is now divided into two subproblems of size $n/2$, followed by linear-time arithmetics. We get the following runtime:

$$T(n) = 2T(n/2) + O(n) \implies T(n) = \mathcal{O}(n \log n).$$

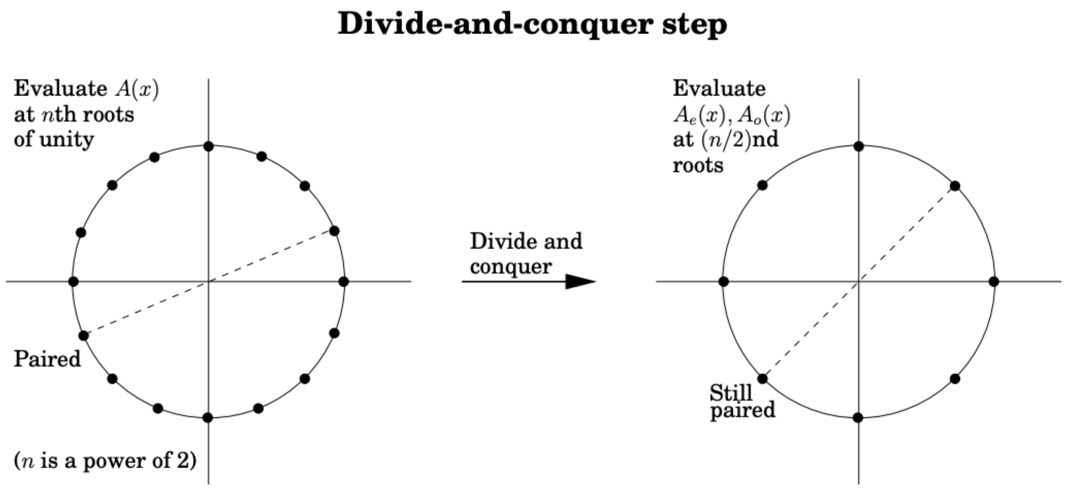
However, we now encounter a problem: The plus-minus trick only works at the top level of the recursion. To recurse at the next level, we need the $n/2$ points $x_0^2, x_1^2, \ldots, x_{n/2-1}^2$ to be plus-minus pairs. Hence, complex numbers come in handy now.

To figure out the choices of complex numbers, we can "reverse engineer" the process. Let's say we have a 1 at the very bottom of the recursion. Then the level above it must consist of its square roots, $\pm\sqrt{1} = \pm 1$, and we continue in this fashion.



We will finally reach the desired n points. It turns out that these n points are in fact the nth roots of unity, i.e., solutions to $z^n = 1$, which are $1, \omega_n, \omega_n^2, \ldots, \omega_n^{n-1}$ where $\omega_n = e^{2\pi i/n}$. If n is even,

1. The nth roots are plus-minus paired, $\omega^{n/2+j} = -\omega^j$.
2. $\omega_n^2 = \omega_{n/2}$.


Divide-and-conquer step
Evaluate $A(x)$ at nth roots of unity
Paired
(n is a power of 2)
Divide and conquer
Evaluate $A_e(x), A_o(x)$ at $(n/2)$nd roots
Still paired

Thus, if we start with these numbers for some n that is a power of 2, then at successive levels of recursion we will have the $\left(n/2^k\right)$th roots of unity, for $k = 0, 1, 2, 3, \ldots$. All these sets of numbers are plus-minus paired and so the divide-and-conquer approach would work perfectly, which results in the **fast Fourier transform**.

Figure 2.7 The fast Fourier transform (polynomial formulation)

```
function FFT($A,\omega$)
Input:  Coefficient representation of a polynomial $A(x)$
        of degree $\leq n-1$, where $n$ is a power of 2
        $\omega$, an $n$th root of unity
Output:  Value representation $A(\omega^0),\ldots,A(\omega^{n-1})$

if $\omega = 1$:  return $A(1)$
express $A(x)$ in the form $A_e(x^2) + xA_o(x^2)$
call FFT($A_e,\omega^2$) to evaluate $A_e$ at even powers of $\omega$
call FFT($A_o,\omega^2$) to evaluate $A_o$ at even powers of $\omega$
for $j = 0$ to $n-1$:
   compute $A(\omega^j) = A_e(\omega^{2j}) + \omega^j A_o(\omega^{2j})$

return $A(\omega^0),\ldots,A(\omega^{n-1})$
```

§2.7.3 Interpolation