Chapter 2 Divide-and-conquer algorithms(분할정보 알고리즘)

1. Break into subproblems(분할)
2. Recursively solve subproblems(재귀)
3. Appropriately combining their answers

2.1 Multiplication

- Number of multiplication decreases from four to three
- When applied recursively, causing significant improvement

- xy = (2^n/2*xL+xR)(2^n/2*yL+yR) = 2^n*xL*yL+2^n/2*(xLyR+xRyL)+xRyR
- Significant operations are the four n/2-bit multiplications
- We can handle by 4 recursive calls
- T(n) = 4T(n/2) + O(n)

Gauss's trick

- xLyR + xRyL = (xL+xR)(yL+yR) - xL*yL - xR*yR
- T(n) = 3T(n/2) + O(n)

The constant factor improvement from 4 to 3 occurs at every level of recursion

--> dramatically lower time bound of O(n^1.59)

- Running time derived by looking at the **algorithm's pattern of recursive calls forming a tree structure**
  - Branching factor(분기계수, 부모가 가질 수 있는 자식 노드의 수): 3 -> each problem recursively produces three smaller ones
  - So, depth=k, subproblems=3^k, size n/2^k
  - Total time spent at depth k = 3^k * O(n/2^k) = (3/2^k) * O(n)
    - k=0 -> O(n)
    - k=log2(n) -> O(3^(log2(n)))
  - Work done increases geometrically from O(n) to O(n^(log2(n)) by a factor of 3/2 per level
  - Overall running time: O(n^(log2(3)) = O(n^1.59)

- If no Gauss's trick -> recursive tree same height but branching factor: 4
  - 4^(log2(n)) = n^2 leaves

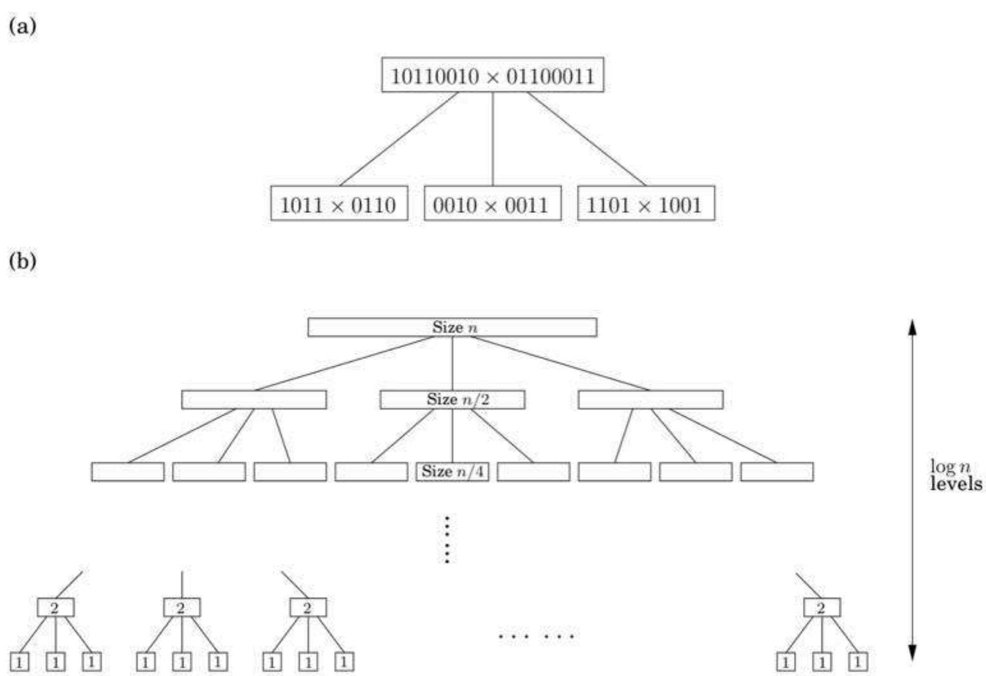
- In divide-and-conquer algorithms, # of subproblems = branching factor of the recursion tree
  - Small changes in coefficients have a big impact on running time

2.2 Recurrence relations

- Generic pattern: Problem of size n -> recursively solve a subproblems of size n/b -> xombine answers in O(n^d) time
- Running time: T(n) = aT(⌈n/b⌉) + O(n^d)

Master theorem

Figure 2.2 Divide-and-conquer integer multiplication. (a) Each problem is divided into three subproblems. (b) The levels of recursion.



Master theorem[2] *If* $T(n) = aT(\lceil n/b \rceil) + O(n^d)$ *for some constants* $a > 0$, $b > 1$, *and* $d \geq 0$, *then*

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

Proof

1. Assume n is a power of b(=n^b)
2. Size of the subproblems decreases by a factor of b

    - Reaches the base case after logb(n) levels(hight of the recursion tree)

    Branching factor is a, so kth level of the tree is made up of

    - a^k subproblems
    - each of size n/b^k
3. Total work done at the level = O(n^d) * (a/b^d)^k

- As k goes from - to logb(n), numbers form a geometric series with ratio a/b^d
- Finding sum of such a series in big-O notation is easy and comes down to 3 cases

1. If ratio < 1, then series is decreasing, and sum is O(n^d)
2. If ratio > 1, then seires is increasing and sum is O(n^logb(a))
3. If ratio = 1, all O(logn) terms are equal to O(n^d)

- Ultimate divid-and-conquer algorithm is binary search
- T(n) = T(⌈n/2⌉) + O(1)

- Master theorem plug in -> running time O(logn)

2.3 Mergesort

- In terms of merging the two sorted sublists

```
_function mergesort(a[1..n])
Input: An array of numbers a[1..n]
Output: A sorted version of this array

if n>1:
    return merge(mergesort(a[1..[n/2]]), mergesort(a[[n/2]+1...n]))
else:
    return a_
```

- How to efficiently merge into a single sorted array?
  - x[1..k] and y[1...l] ==> z[1..k+l]
  - First element of z is x[1] or y[1]
  - The rest of z[.] can be constructed recursively

```
_function merge(x[1..k], y[1..l])
if k=0: return y[1..l]
if l=0: return x[1..k]
if x[1] <= y[1]:
    return x[1]∘merge(x[2...k], y[1...l]) #∘ for concatenation
else:
    return y[1]∘merge(x[1...k], y[2...l])_
```

```
* Total running time of O(k+l)
* Merges are linear, and overall time taken by mergesort
    T(n) = 2T(n/2) + O(n) or O(nlogn)

* Merge operation: Singletons are merged into pairs of 2-tuples, merged
 to 4-tuples, and so on
    * mergesort made iterative
        * arrays can be organized in a queue

        _function iterative-mergesort(a[1...n])
        Input: elements a1, a2, ..., an to be sorted_

        _Q = [] (empty queue)
        for i = 1 to n:
            inject(Q, [ai])
        while |Q|>1:
            inject(Q, merge(eject(Q), eject(Q)))
        return eject(Q)_
```

- An nlogn lower bound for sorting

2.4 Medians

- 50th percentile
- Purpose: to summarize a set of numbers by a single, typical value
- Used more than mean because always one of the data values and less sensitive to outliers
- Sotrting takes O(nlogn) time but we don't need to sort all, just need median
- For recursive solutions, easier with a more general version

  SELECTION Input: A list of numbers S: an integer k Output: The kth smallest element of S
  - If k=1, minimum of S is sought
  - If k= [|S|/2], it is the median

A randomized divide-and-conquer algorithm for selection

- Number v, split list s into 3 categories

S : | 2 | 36 | 5 | 21 | 8 | 13 | 11 | 20 | 5 | 4 | 1 |

is split on $v = 5$, the three subarrays generated are

S_L : | 2 | 4 | 1 | S_v : | 5 | 5 | S_R : | 36 | 21 | 8 | 13 | 11 | 20 |

The search can instantly be narrowed down to one of these sublists. If we want, say, the *eighth*-smallest element of S, we know it must be the *third*-smallest element of S_R since $|S_L| + |S_v| = 5$. That is, selection$(S, 8)$ = selection$(S_R, 3)$. More generally, by checking k against the sizes of the subarrays, we can quickly determine which of them holds the desired element:

$$\text{selection}(S,k) = \begin{cases} \text{selection}(S_L,k) & \text{if } k \leq |S_L| \\ v & \text{if } |S_L| < k \leq |S_L| + |S_v| \\ \text{selection}(S_R, k - |S_L| - |S_v|) & \text{if } k > |S_L| + |S_v|. \end{cases}$$

- Three sublists SL, SV, SR can be computed from S in linear time
- Effect of the split: to shrink the number of elements from |S| to at most max{|SL|, |SR|}
- How to choose v:: should be picked quicly, and it shoudl shrink the array substantially
  - Ideally |SL|, |SR| almost equal to 1/2*|S|
  - Then, running time T(n) = T(n/2) + O(n)
    - linear as desired
    - But v needs to be median.. -> Pick v randomly from S

Efficiency analysis

- Worst scenario: Θ(n^2), best scenario of splitting perfectly in half: O(n)
- Both very unlikely to happen
- Call v good if within 25th-75th percentile

Lemma *On average a fair coin needs to be tossed two times before a "heads" is seen.*

Proof. Let E be the expected number of tosses before a heads is seen. We certainly need at least one toss, and if it's heads, we're done. If it's tails (which occurs with probability 1/2), we need to repeat. Hence $E = 1 + \frac{1}{2}E$, which works out to $E = 2$. ∎

Therefore, after two split operations on average, the array will shrink to at most three-fourths of its size. Letting $T(n)$ be the *expected* running time on an array of size n, we get

$$T(n) \leq T(3n/4) + O(n).$$

Time taken on an array of size n
<= (time taken on an array of size 3n/4) + (time to reduce array size to <+ 3n/4)

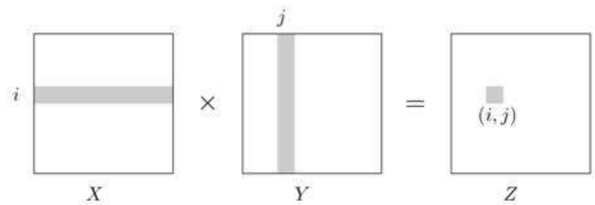
- Conclusion: T(n) = O(n): on any input, algorithm returns the correct answer after a linear number of steps on the average

2.5 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 ith row of X with the jth column of Y:



- XY != YX
- O(n^3) algorithm: n^2 entries to be computed, and each takes O(n) times

- Introduction of matrix multiplication
  - More efficient
  - Based on divide-and-conquer
  - Particularly easy to break into subproblems as performed blockwise
  - To compute size-n product XY, recursively compute eight size-n/2 products and then do a few O(n^2) - time additions
  - Total running time: T(n) = 8T(n/2) + O(n^2)
    - Comes out as O(n^3) :(

- But efficiency improved with clever algebra
  - New running time: T(n) = 7T(n/2) + O(n^2)
  - By master theorem: O(n^log2(7) is almost equal to O(n^2.81))

2.6 The fast Fourier transformation

- So far: how divide-and-conquer gives fast algorithms for multiplying integers and matrices
- Next target: polynomials

and matrices; our next target is *polynomials*. The product of two degree-d polynomials is a polynomial of degree $2d$, for example:

$$(1 + 2x + 3x^2) \cdot (2 + x + 4x^2) = 2 + 5x + 12x^2 + 11x^3 + 12x^4.$$

More generally, if $A(x) = a_0 + a_1 x + \cdots + a_d x^d$ and $B(x) = b_0 + b_1 x + \cdots + b_d x^d$, their product $C(x) = A(x) \cdot B(x) = c_0 + c_1 x + \cdots + c_{2d} x^{2d}$ has coefficients

$$c_k \;=\; a_0 b_k + a_1 b_{k-1} + \cdots + a_k b_0 \;=\; \sum_{i=0}^{k} a_i b_{k-i}$$

- Computing ck takes O(k) steps
- Finding all 2d + 1 coefficients would have required Θ(d^2) time

- How to muptiply polynomials faster than this?

2.6.1 An alternative representation of polynomials

- Important property of polynomials:
  A degree-d polynomial is uniquely characterized by its values at any d+1 distinct points
- Fix any distinct points x0, ..., xd.
- Specify a degree-d polynomial A(x) = a0+a1*x*+*a2*x^2+...+ad*x^d by

  1. Its coefficients a0, a1, ..., ad
  2. The values A(x0), A(x1), ... A(xd) (More attractive for polynomial)

- Product C(x) has degree 2d. Determined by its value at any 2d+1
- And its value at any given point z is easy enough to figure out, just A(z) times B(z)
- Thus, polynomial multiplication takes linear time in the value representation

- Problem: expect the input polynomials and thier product to be specified by coefficients

Interpolation

- So, need to first translate from coefficients to values(evaluating th epolynomial at the chosen points)
- Then multiply in the value representation
- Finally translate back to coefficients

Polynomial multiplication

Input: Coefficients of two polynomials, A(x) and B(x), of degree d
Output: Their product C = A*B

Selection
Pick some points x0, x1, ...xn-1, where n>= 2d+!

Evaluation
Compute A(x0), A(x1), ... A(xn-1) and B(x0), B(x1), ..., B(xn-1)

Multiplication
Compute C(xk) = A(xk)*B(Xk) for all k=0, ..., n-1

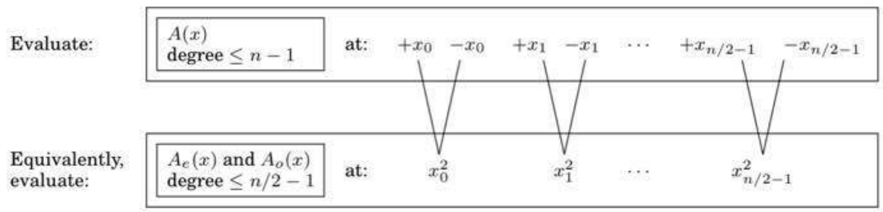
Interpolation
Recover C(x) = c0+c1*x+...+c2d*x^2d

- Equivalence of the two polynomial representations -> approach is correct, but how efficient?
- How about evaluation?

- Baseline for n points: Θ(n^2)
- Faster Fourier transform(FFT) does it in just O(nlogn) times

2.6.2 Evaluation by divide-and-conquer

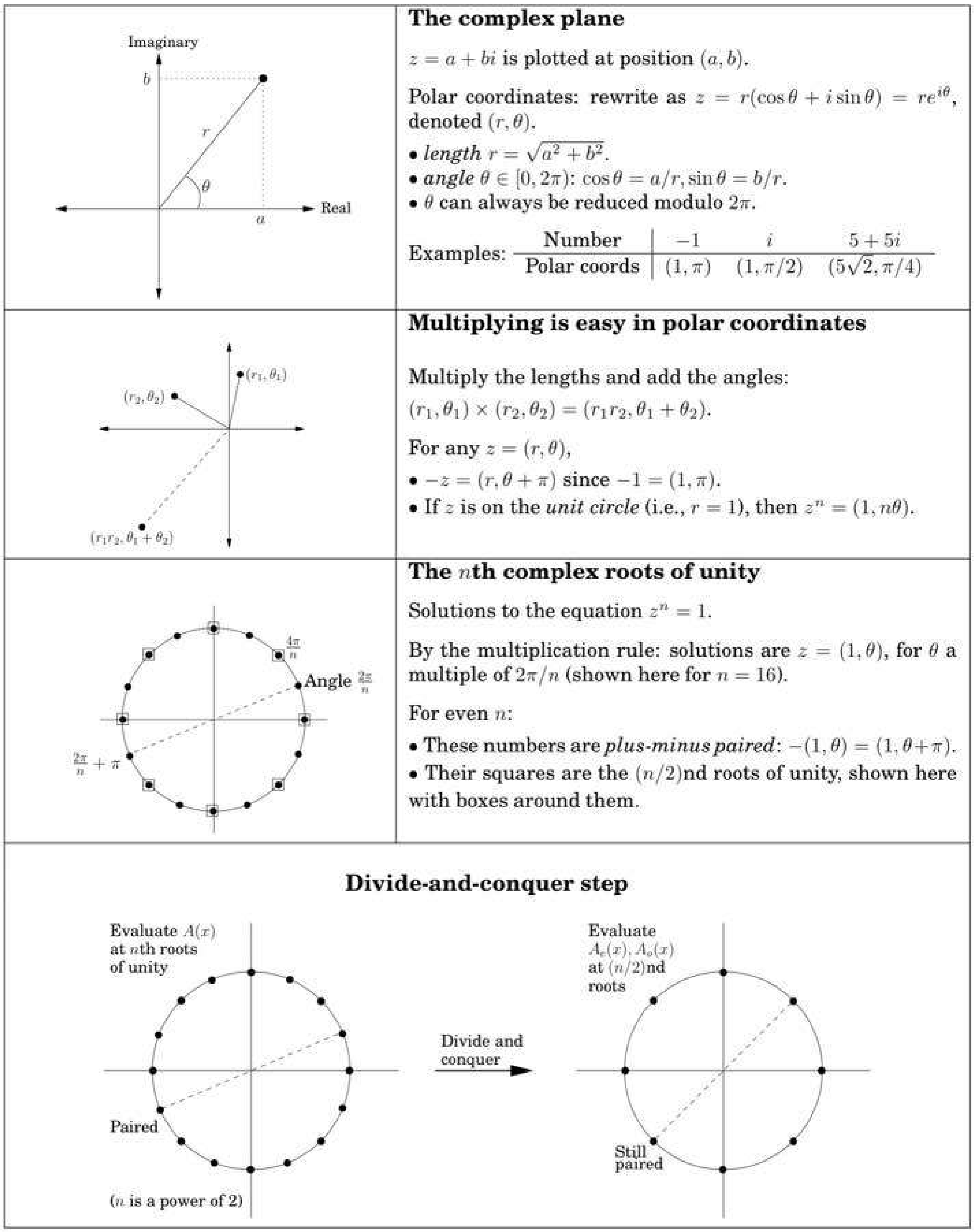
- Idea for how to pick the n points at which to evaluate a polynomial A(x) of degree <= n-1
- If choose them to be positive-negative paiers, +-x0, +-x1, ..., xn/2-1
- Evaluating A(x) at n paired points +-x0, ..., +- xn/2-1 redueces to evaluating Ae(x) and Ao(x)



- T(n) = 2T(n/2) + O(n), which is O(nlogn) <- what we want!

- But have a problem... +- trick only working at the top level of the recursion
- We can use complex numbers for recursion on next levels z^n = 1

Figure 2.6 The complex roots of unity are ideal for our divide-and-conquer scheme.



The complex plane

$z = a + bi$ is plotted at position (a, b).

Polar coordinates: rewrite as $z = r(\cos\theta + i\sin\theta) = re^{i\theta}$, denoted (r, θ).

- *length* $r = \sqrt{a^2 + b^2}$.
- *angle* $\theta \in [0, 2\pi)$: $\cos\theta = a/r, \sin\theta = b/r$.
- θ can always be reduced modulo 2π.

Examples:

Number	-1	i	$5 + 5i$
Polar coords	$(1, \pi)$	$(1, \pi/2)$	$(5\sqrt{2}, \pi/4)$

Multiplying is easy in polar coordinates

Multiply the lengths and add the angles:
$(r_1, \theta_1) \times (r_2, \theta_2) = (r_1 r_2, \theta_1 + \theta_2)$.

For any $z = (r, \theta)$,

- $-z = (r, \theta + \pi)$ since $-1 = (1, \pi)$.
- If z is on the *unit circle* (i.e., $r = 1$), then $z^n = (1, n\theta)$.

The nth complex roots of unity

Solutions to the equation $z^n = 1$.

By the multiplication rule: solutions are $z = (1, \theta)$, for θ a multiple of $2\pi/n$ (shown here for $n = 16$).

For even n:

- These numbers are *plus-minus paired*: $-(1, \theta) = (1, \theta + \pi)$.
- Their squares are the $(n/2)$nd roots of unity, shown here with boxes around them.

Divide-and-conquer step

- In the figure, thier panel introduces nth roots of unity
  - The complex numbers 1, w, w^2, ..., w^n-1, where w = e^(2*pii*/n)
  - If n is even,

1. The nth roots are plus-minus paiered, w^(n/(2+j) = -w^j)
2. Squaring them produces the (n/2)nd roots of utility

*If we start with numbers for some n that is a power of 2, then we will have the (n/2^k)th roots or unity at succssive levels of recursion

- All thses sets of numbers are plus-minus paierd, and so our divide-and-conquer works perfectly
- Resulting algorithm is the fast Fourier transform

The fast Fourier transform (polynomial formulation)

function FFT(A, w)

```
Input: Coefficient representation of a polynomial A(x) of degree <= n-1,
where n is a power of 2w, an nth root of unity
Output: Value representation A(w0), ..., A(w^(n-1))

if w = 1: return A(1)
express A(x) in the form Ae(x^2) + xAo(x^2)
call FFT(Ae, w^2) to evaluate Ae at even powers of w
call FFT(A0, w^2) to evaluate Ao at even powers of w
for j = 0 to n-1:</p>
    compute A(w^j) = Ae(w^2j) + w^j*A0(w^2j)

return A(w0), ...., A(w^(n-1))
```

2.6.3 Interpolation

- Designed the FFT(a way to move from coefficients to valeus in time) just O(nlogn)
- When the points {xi} are complex nth roots of unity (1, w, w^2, ..., w^(n-1))
  - {values} = FFT({coefficients}, 2)
- Inverse operations, interpolation
  - {coefficients} = 1/n*FFT({values}, w^(-1))
- Interpolation is solved simply and elegently using FFT algorithm but called with w^(-1) in place of w!

A matrix reformulation

- Both vectors of n numbers, and one is a linear transformation of the other

$$\begin{bmatrix} A(x_0) \\ A(x_1) \\ \vdots \\ A(x_{n-1}) \end{bmatrix} = \begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ & & \vdots & & \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix}.$$

- Middle matrix(M) is a Vandermonde matrix
  - If x0, ..., xn-1 are distinct numbers, then M is invertible
- Existance of M^-1 allows to invert the preceding matrix equation so as to express coefficients in terms of values
  - **Briefly, evaluation is multiplication by M, while interpolation is multiplication by M^-1**
- Justifies an assumption that A(x) is uniquely characterized by its values at any n points
- We have an explicit formula that will give us the coefficients of A(x) in this situation

We have an explicit formula that will give us the coefficients of A(x) in this situation

- Vandermodes also quicker to invert than more general matrices(In O(n^2) than O(n^3))
- But still not fast enough for us

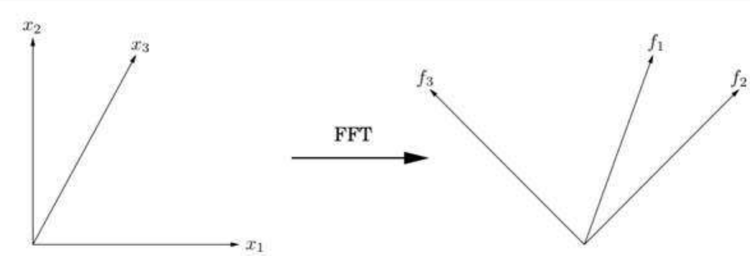
Interpolation resolved

- In linear algebra terms, the FFT multiplies an arbitrary n-dimenstional vector--which we have been calling the coefficient representation--by the n x n matrix

$$M_n(\omega) = \begin{array}{cl} \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2(n-1)} \\ & & \vdots & & \\ 1 & \omega^j & \omega^{2j} & \cdots & \omega^{(n-1)j} \\ & & \vdots & & \\ 1 & \omega^{(n-1)} & \omega^{2(n-1)} & \cdots & \omega^{(n-1)(n-1)} \end{bmatrix} & \begin{array}{l} \longleftarrow \text{ row for } \omega^0 = 1 \\ \longleftarrow \omega \\ \longleftarrow \omega^2 \\ \quad\ \vdots \\ \longleftarrow \omega^j \\ \quad\ \vdots \\ \longleftarrow \omega^{n-1} \end{array} \end{array}$$

- Crucial observation to prove: The columns of M are orthogonal (at right angles) to each other <== **Fourier basis**

Figure 2.8 The FFT takes points in the standard coordinate system, whose axes are shown here as x_1, x_2, x_3, and rotates them into the Fourier basis, whose axes are the columns of $M_n(\omega)$, shown here as f_1, f_2, f_3. For instance, points in direction x_1 get mapped into direction f_1.



- Effect of multiplying a vector by M is to rotate it from the standard basis, with the usual set of axes, into the Fourier basis, which is defined by the columns of M
- The FFT is thus a change of basis, a rigid rotation
- The inverse of M is the opposite rotation, from the Fourier basis back into the standard basis

- Inversion formula Mn(w)^-1 = 1/n*Mn(w^-1)
  - But w^-1 is also an nth root of unity, so interpolation(or multiplication by Mn(w)^-1) is itself just an FFT opertion with w replaced by w^-1

- Details: Take w to be e^(2*pi*i/n) for convenience
- Think of the columns of M as vectors in C^n
- Angle between two vecots in C^n is just a **scaler factor times their inner product**

  uv* = u0v0* + u1v1* + ... + u(n-1)v(n-1)*

- Quantity maximized when vectors lie in the smae direction and is zero when vectors are orthogonal to each other

Fundamental observation

- **Lemma** The columns of matrix M are orthogonal to each other

- **Proof** Take the inner product of any columns j and k of matrix M,

  1 + w^(j-k)+w^2(j-k)+...+w^(n-1)*(j-k)
- Orthogonality property summarized in the single equation

  MM* = nI,

- (MM)*ij is the inner product, which implies M^-1 = (1/n)M*
- Polynomial multiplication a lot easier in the Fourier basis

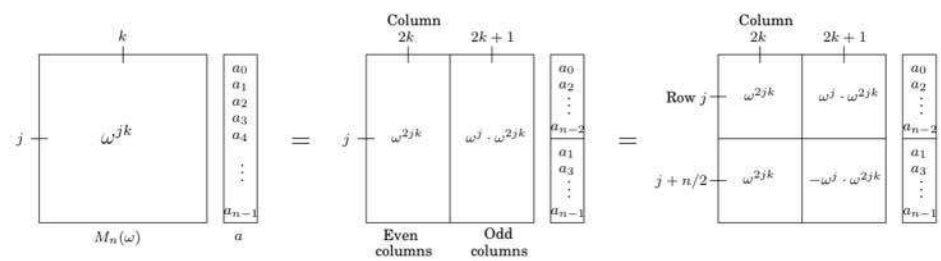
- Therefore,

1. Rotate vectors into the Fourier basis (evaluation)
2. Perform the task(multiplication) while their rotated coutnerparts are value representations

2.6.4 A closer look at the fast Fourier transform

The definitive FFT algorithm

- FFT takes as input a vector a = (a0, ..., an-1) and a complex number w whose powers 1, w, w^2, ..., w^n-1 are the complex nth roots of unity
- Multiplies vector a by the n x n matrix Mn(w), which has (j, k)th entry
- The potential of using divide-and-conquer in matrix-vector multiplication becomes apparent when M's columns are segregated into evens and offs



- 2nd step: Simplified entries in the bottom half of the matrix using w^n/2 = -1 and w^n/2 = 1
- Top left n/2*n/2 submatrix is Mn/2(w^2)
- Top and bottom right submatrices are almost as same as Mn/2(w^2) but with their jth rows multiplied through by w^j and -w^j

*Figure: The fast Fourier transform

```
function FFT(a,w)
Input: An array a = (a0, a1, ..., an-1), for n a power of 2
       A primitive nth root of unity, w
Output: Mn(w)a

if w = 1: return a
(so, s1, ..., sn/3-1) = FFT((a0, a2, ..., an-2), w^2)
(s'o, s'1, ...., s'n/2-1) = FFT((a1, a3, ..., an-1), w^2)
for j = 0 to n/2-1:
    rj = sj + w^js'j
    rj+n/2 = sj - w^js'j
return (r0, r1, ..., rn-1)
```

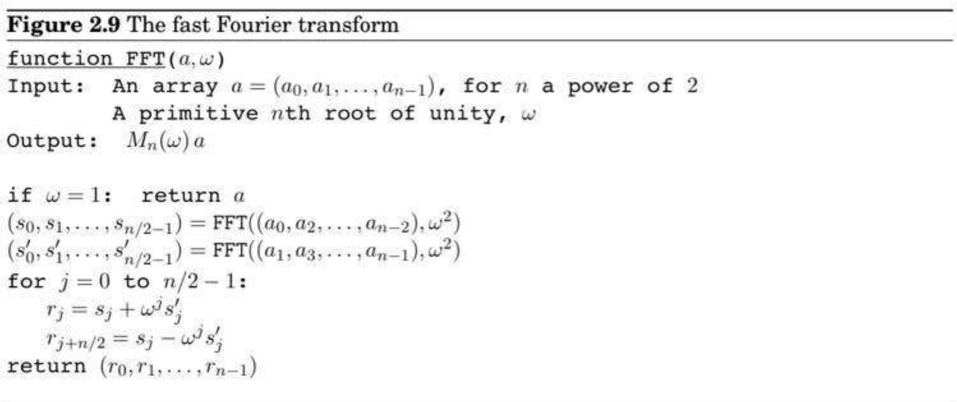
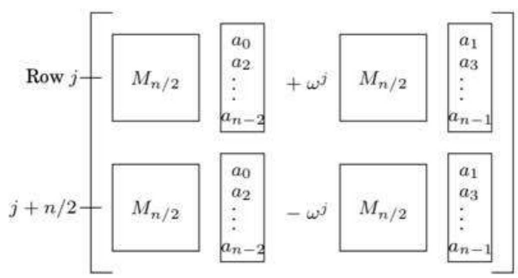


Figure 2.9 The fast Fourier transform

function FFT(a, ω)
Input: An array $a = (a_0, a_1, \ldots, a_{n-1})$, for n a power of 2
A primitive nth root of unity, ω
Output: $M_n(\omega)\, a$

if $\omega = 1$: return a
$(s_0, s_1, \ldots, s_{n/2-1}) = \text{FFT}((a_0, a_2, \ldots, a_{n-2}), \omega^2)$
$(s'_0, s'_1, \ldots, s'_{n/2-1}) = \text{FFT}((a_1, a_3, \ldots, a_{n-1}), \omega^2)$
for $j = 0$ to $n/2 - 1$:
$r_j = s_j + \omega^j s'_j$
$r_{j+n/2} = s_j - \omega^j s'_j$
return $(r_0, r_1, \ldots, r_{n-1})$

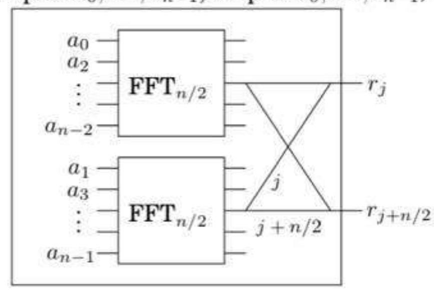


- The product of Mn(w) with vector (a0, ..., an-1), a size-n problem, can be expressed in terms of two size-n/2 problems: the product of Mn/2(w^2) with (a0, a2, ..., an-2) and with (a1, a3, ..., an-1)
- This divide-and-conquer strategy leads to the definitive FFT algorithm of the above FFT(running time is T(n) = 2T(n/2) + O(n) = O(nlogn)

The fast Fourier transform unraveled

- problem of size n -> reduced to two subproblems of size n/2

FFT$_n$ **(input:** $a_0, \ldots, a_{n-1}$**, output:** $r_0, \ldots, r_{n-1}$**)**



- Two outputs depicted are executing the commands from the FFT algorithm via a pattern of wires known as a butterfly

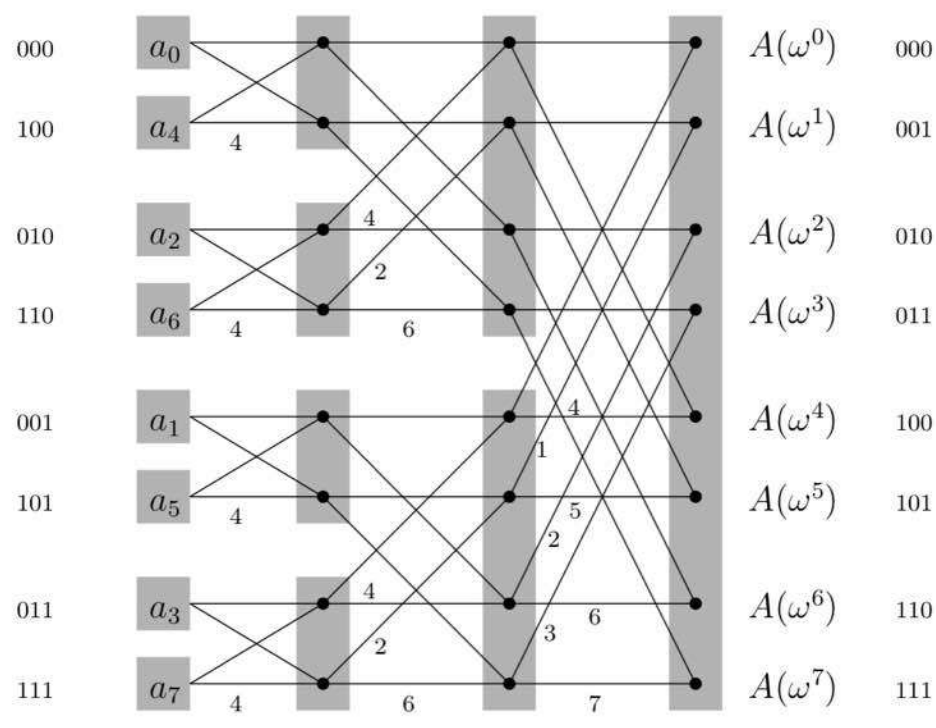
  rj = sj + w^js*'j rj+n/2 = sj - w^*js'j

1. For n inputs there are log2n levels, each with n nodes, for a total of nlogn operations
2. The inputs are arranged in a peculiar order: 0, 4, 2, 6, 1, 5, 3, 7

- Inputs are arranged by increasing last bit of the binary representation of their index

3. There is a unique path between each input aj and each output A(w^k)
4. On the path between aj and A(w^k), the labels add up to jk mod 8
5. Notice FFT circuit is a natural for parallel computation and direct implementation in hardware

Figure 2.10 The fast Fourier transform circuit.



In [ ]: