CSC3160: Design and Analysis of Algorithms

Week 7: Divide and Conquer

Instructor: Shengyu Zhang

Example 1: Merge sort

Starting example

- Sorting:
  - We have a list of numbers $x_1, \ldots, x_n$.
  - We want to sort them in the increasing order.

An algorithm: merge sort

- Merge sort:
  - Cut it into two halves with equal size.
    - Suppose 2 divides n for simplicity.
  - Suppose the two halves are sorted: Merge them.
    - Use two pointers, one for each half, to scan them, during which course do the appropriate merge.
  - How to sort each of the two halves? Recursively.

Complexity?

- Suppose this algorithm takes $T(n)$ time for an input with n numbers.
- Thus each of the two halves takes $T(n/2)$ time.
- The merging? $O(n)$
  - Scanning n elements, an $O(1)$ time operation needed for each.
- Total amount of time: $T(n) \leq 2T(n/2) + c \cdot n$.

How to solve/bound this recurrence relation?

- $T(n) \leq 2T(n/2) + c \cdot n$

$$\leq 2T(n/4) + c \cdot n/2$$

$$\leq 4T(n/4) + 2c \cdot n$$

$$\leq 2T(n/8) + c \cdot n/4$$

$$\leq 8T(n/8) + 3c \cdot n$$

$$\leq \ \dots$$

$$\leq nT(n/n) + (\log n)c \cdot n$$

$$\leq {\color{red} O(n \log n).}$$

A general method for designing algorithm: Divide and conquer

- Breaking the problem into subproblems
  - that are themselves smaller instances of the same type of problem

- Recursively solving these subproblems

- Appropriately combining their answers

Complexity

- Running time on an input of size n: $T(n)$
- Break problem into a subproblems, each of the same size n/b.
  - In general, a is not necessarily equal to b.
- Time to recursively solve each subproblem: $T(n/b)$
- Time for breaking problem (into subproblems) and combining the answers: $O(n^d)$

Master theorem

- $T(n) = aT(n/b) + O(n^d)$
  - $a > 0$, $b > 1$, and $d \geq 0$ are all constants.
- Then

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

  - Proof in textbook. Not required.
  - But you need to know how to apply it.

- $T(n) = aT(n/b) + O(n^d)$

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

- Merge sort: $T(n) \leq 2T(n/2) + O(n)$.
- $a = b = 2, d = 1$. So $d = \log_b a$.
- By the master theorem: $T(n) = O(n \log n)$.

Example 2: Selection

Selection

- Problem: Given a list of n numbers, find the k-th smallest. $S:$

2	36	5	21	8	13	11	20	5	4	1

- We can sort the list, which needs $O(n \log n)$.
- Can we do better, say, linear time?
- After all, sorting gives a lot more information than we requested.
  - Not always a waste: consider dynamic programming where solutions to subproblems are also produced.

Idea of divide and conquer

- Divide the numbers into 3 parts

  $< v, \qquad = v, \qquad > v$

- Depending on the size of each part, we know which part the k-th element lies in.

- Then search in that part.

- Question: Which v to choose?

Pivot

- Suppose we use a number v in the given list as a pivot.
- As said, we divide the list into three parts.
  - S_L: Those numbers smaller than v
  - S_v: Those numbers equal to v
  - S_R: Those numbers larger than v

S_L :

2	4	1

S_v :

5	5

S_R :

36	21	8	13	11	20

After the partition

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

- The division is simple: just scan the list and put elements into the corresponding part.
  - $O(n)$ time.
- To select the k-th smallest value, it becomes

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

- Complexity?

Divide and conquer

- Note: though there are two subproblems (of sizes $|S_L|$ and $|S_R|$), we need to solve only one of them.
  - Compare: in quicksort, we need to sort both substrings!
- Complexity:

$$T(n) = \max\{T(|S_L|), T(|S_R|)\} + O(n)$$

- A new issue: $|S_L|$ and $|S_R|$ are not determined
  - Depends on the pivot.

- If the pivot is the median:
  - $T(n) = T(n/2) + O(n)$
  - $T(n) = aT(n/b) + O(n^d)$

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

  - Thus finally $T(n) = O(n)$, better than $O(n\log n)$ by sorting.

- If the pivot is at one end (say, the smallest)
  - $T(n) = T(n-1) + O(n)$
  - What's the complexity?
  - Complexity: $O(n^2)$

- The similarity to quicksort tells us: a random pivot performs well
  - It's away from either end by cn with const. prob.
- To be more precise, it's in $(n/4, 3n/4)$ with probability $1/2$.
- And in this case, the recursion becomes
  - $T(n) = T(3n/4) + O(n)$

- $T(n) = T(3n/4) + O(n)$
- Recall: $T(n) = aT(n/b) + O(n^d)$

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

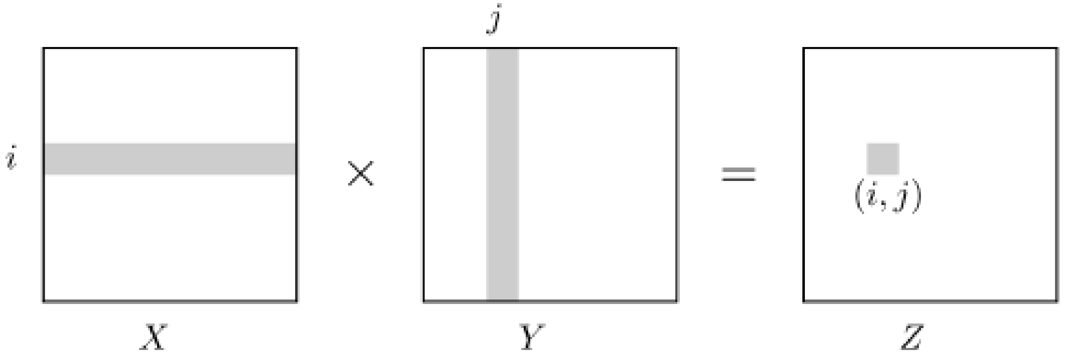
- So $T(n) = O(n)$

- Thus we can use the following simple strategy:
  - Pick a random pivot,
  - **do** the recursion
- Each random pivot falls in $\left(\frac{n}{4}, \frac{3n}{4}\right)$ w/ prob. ½.
- $\mathbf{E}\left[\text{number of trials to get a pivot in}\left(\frac{n}{4}, \frac{3n}{4}\right)\right] = 2$.
- It is enough to get $\log_{4/3} n$ good pivots to make the problem size to drop to 1.
- Thus $\mathbf{E}[\text{running time}] = 2 \cdot O(n) = O(n)$.

Example 3: Matrix multiplication

Matrix multiplication

- Recall: the product of two $n \times n$ matrices is another $n \times n$ matrix.
- Question: how fast can we multiply two matrices?
- Recall: $z_{ij} = \sum_{k=1,\ldots,n} x_{ik} y_{kj}$
  - z_{ij}: the entry (i, j) in the matrix Z. Similar for x_{ik}, y_{kj}



- This takes $O(n^3)$ multiplications (of numbers).
- For a long time, people thought this was the best possible.
- Until Straussen came up with the following.

- If we break the matrix into blocks

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

- Then the product is just block multiplication

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

- 8 matrix multiplications of dimension $n/2$
- Plus $O(n^2)$ additions.

- Thus the recurrence is
  - $T(n) = 8T(n/2) + O(n^2)$
  - $T(n) = aT(n/b) + O(n^d)$

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

- This gives exactly the same $O(n^3)$, not interesting.

- However, Straussen observed that we can actually use only 7 (instead of 8) multiplications of matrices with dimension $n/2$.

God knows how he came up with it.

- And here is 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

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

- Thus the recurrence becomes
  - $T(n) = 7T(n/2) + O(n^2)$
  - $T(n) = aT(n/b) + O(n^d)$

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

- And solving this gives
  - $T(n) = O\left(n^{\log_2 7}\right) = O(n^{2.81\ldots})$.

- Best in theory? $O(n^{2.37\ldots})$.
- Conjecture: $O(n^2)$!
- In practice? People still use the $O(n^3)$ algorithm most of the time. (For example, in matlab.)
  - It's simple and robust.

Fast Fourier Transform (FFT)

Multiplication of polynomials

- Many applications need to multiply two polynomials.
  - e.g. signal processing.
- A degree-n polynomial is

$$A(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

  - The degree is at most n. (The degree is exactly n if $a_n \neq 0$.)
- The summation of two polynomials is easy

$$A(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

$$+)\ B(x) = b_n x^n + b_{n-1} x^{n-1} + \cdots + b_1 x + b_0$$

$$A(x) + B(x) = (a_n + b_n) x^n + \cdots + (a_1 + b_1) x + (a_0 + b_0).$$

  - which still has degree at most n.

Multiplication

- Multiplication of two polynomials: a different story.

$$A(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$
$$\times)\ B(x) = b_n x^n + b_{n-1} x^{n-1} + \cdots + b_1 x + b_0$$

$$A(x)B(x) = c_{2n} x^{2n} + c_{2n-1} x^{2n-1} + \cdots + c_1 x + c_0$$

  - where $c_k = a_0 b_k + a_1 b_{k-1} + \cdots + a_k b_0$ (convolution)
- Try an example: $(x^3 + 2x^2 + 4x + 5)(3x^3 + x + 8)$
- The multiplication can have degree $2n$.
- If we directly use this formula to multiply two polynomials, it takes $O(n^2)$ time.

Better? YES!

- By FFT: We can do it in time $O(n \log n)$.
  - FFT: Fast Fourier Transform

What determines/represents a polynomial?

- Let's switch to degree-$(n-1)$ for later simplicity.
- We already used coefficients $(a_{n-1}, a_{n-2}, \dots, a_0)$ to represent a polynomial.
  - Coefficient representation
- Other ways?
- Yes: By giving values on n (distinct) points.
  - Point-value representation.
- [Fact] A degree-$(n-1)$ polynomial is uniquely determined by values on n distinct points.

Reason

- We have n coefficients to determine
  - $(a_{n-1}, a_{n-2}, \dots, a_0)$
- We know values on n distinct points.
  - $(x_0, y_0), \dots, (x_{n-1}, y_{n-1})$
- How to get the coefficients? Just solve the system of equations:

$$\begin{gathered} a_{n-1}x_0^{n-1} + \cdots + a_1 x_0 + a_0 = y_0 \\ a_{n-1}x_1^{n-1} + \cdots + a_1 x_1 + a_0 = y_1 \\ \vdots \\ a_{n-1}x_{n-1}^{n-1} + \cdots + a_1 x_{n-1} + a_0 = y_{n-1} \end{gathered}$$

- In matrix form:

unknowns

$$\begin{bmatrix} x_0^{n-1} & \cdots & x_0 & 1 \\ x_1^{n-1} & \cdots & x_1 & 1 \\ \vdots & \vdots & \vdots & \vdots \\ x_{n-1}^{n-1} & \cdots & x_{n-1} & 1 \end{bmatrix} \begin{bmatrix} a_{n-1} \\ a_{n-2} \\ \vdots \\ a_0 \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{bmatrix}$$

- The matrix is *Vandermonde* matrix, which is invertible. Thus it has a unique solution for the coefficients $(a_{n-1}, a_{n-2}, \dots, a_0)$.

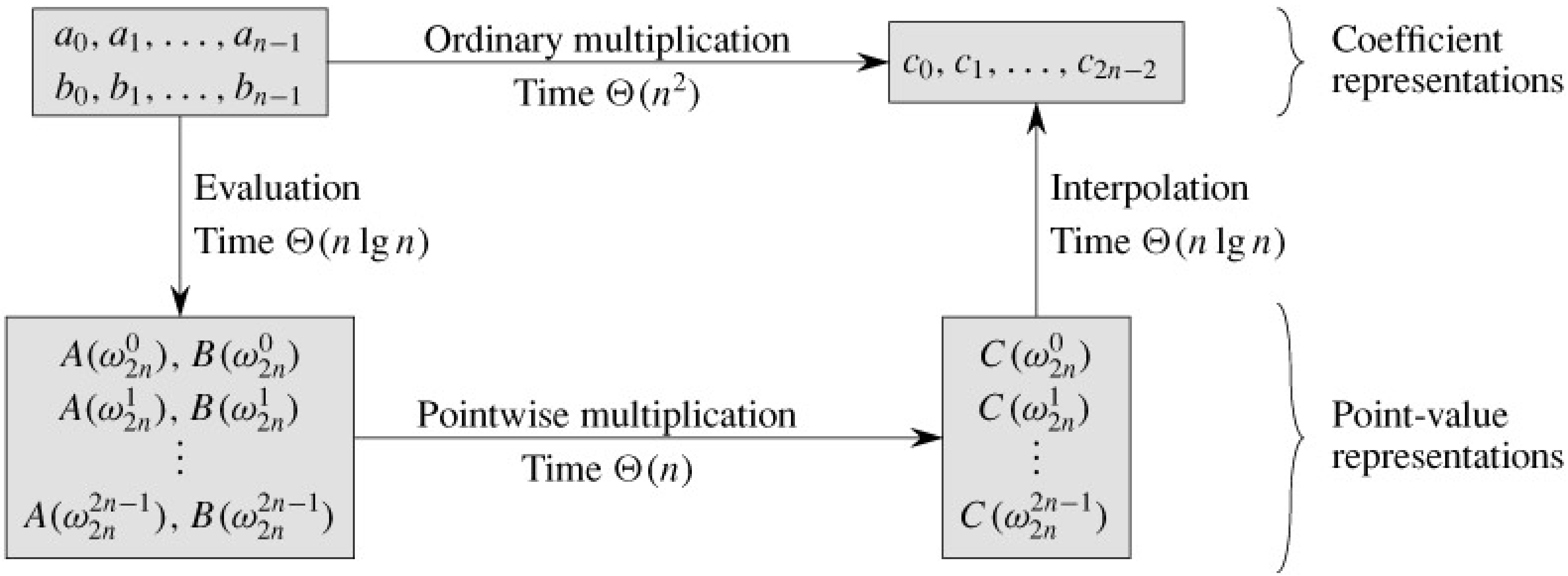
Advantage of point-value representation?

- It makes the multiplication extremely easy:
  - Though $A(x)B(x)$ is hard
  - For any fixed point x_0, $A(x_0)B(x_0)$ is simply multiplication of two numbers.
- So given $\big(x_0, A(x_0)\big), \dots, \big(x_{n-1}, A(x_{n-1})\big)$
  and $\big(x_0, B(x_0)\big), \dots, \big(x_{n-1}, B(x_{n-1})\big)$,
  it's really easy to get
  $\big(x_0, A(x_0)B(x_0)\big), \dots, \big(x_{n-1}, A(x_{n-1})B(x_{n-1})\big)$.
  - $O(n)$ time.
- Thus we have the following interesting idea for polynomial multiplication…

Go to an easy world and come back

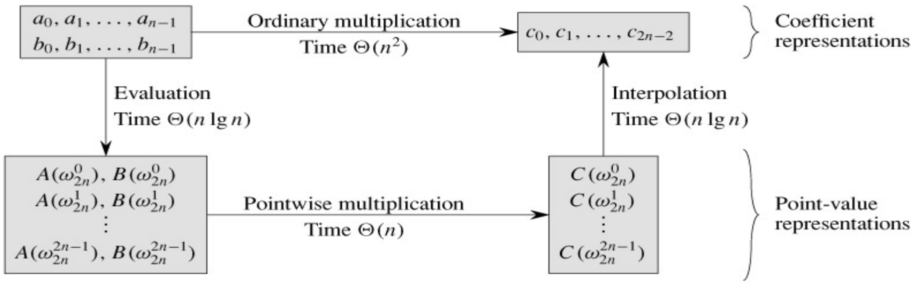
- HK used to have many industries.
- Later found mainland has less expensive labor. So:
  - moved the companies to mainland,
  - did the work there,
  - and then shipped the products back to HK to sell
- This is worth doing if: it's cheaper in mainland, and the traveling/shipping is not expensive either.
  - which turned out to be true.

In our case



- We need to investigate: the cost of traveling.
  - Both way.

Traveling



- From coefficient representation to point-value representation:
  - Evaluate two polynomials both on $2n$ points. --- evaluation.
- From point-value representation to coefficient representation:
  - Compute one polynomial $C(x)$ back from $2n$ point values. --- interpolation.
- Both can be done in $O(n \log n)$ time.

Evaluation

- One point?
- $A(x_0) = a_0 + a_1 x_0 + \cdots + a_{n-1} x_0^{n-1}$
- Directly by the above: n^2 multiplications
- Horner's rule:

$$A(x_0) = a_0 + x_0(a_1 + x_0(a_2 + \cdots + x_0(a_{n-2} + x_0 a_{n-1}) \ldots))$$

--- $O(n)$ multiplications.

Number of points

- How many points we need to evaluate on?
- Since we later want to reconstruct the product polynomial $C(x) = A(x)B(x)$, which has degree $2n - 2$.
- So $2n - 1$ (point,value) pairs are enough to recover $C(x)$.
- We'll evaluate $A(x)$ and $B(x)$ on $2n$ points $x_0, \dots, x_{2n-1}$, and get $C(x_i) = A(x_i)B(x_i)$.
  - $2n - 1$ points are enough. We use $2n$ for convenience.
- Then recover $C(x)$ from $\{C(x_i): i = 0, \dots, 2n - 1\}$.

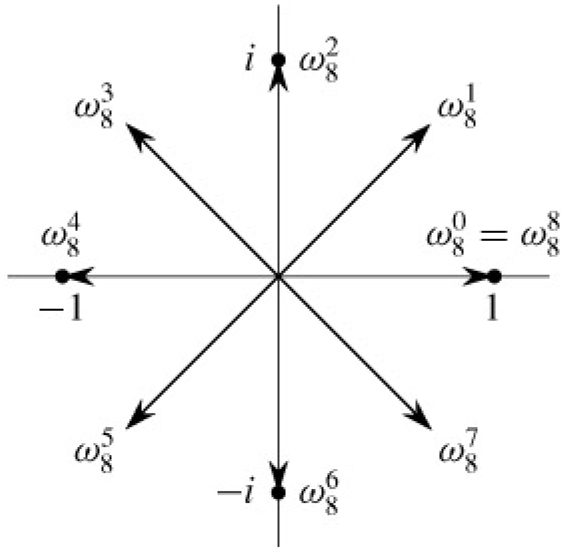
- Now evaluation on one point needs $O(n)$ time, so evaluations on $2n$ points need $O(n^2)$ time.
- Too bad. We want $O(n \log n)$.
- Important fact: cost(evaluations on $2n$ points) can be cheaper than $2n \times$ cost(evaluation on 1 point).
  - If we choose the $2n$ points carefully.
- And it turns out that the $2n$ chosen points also make the (later) interpolation easier.
- This powerful tool is called *Fourier Transform*.

Complex roots of unity

- 1 has a complex root $\omega_m = e^{i\frac{2\pi}{m}}$, satisfing $\omega_m^m = 1$.
- *Discrete Fourier Transform* (DFT):

  $$(a_0, \ldots, a_{m-1}) \to (y_0, \ldots, y_{m-1})$$

  where $y_k = a_0 + a_1\omega_m^k + \cdots + a_{m-1}\omega_m^{k(m-1)}$
  - Try $m = 3$ now.



- Note: y_k evaluates polynomial $A(x) = \sum_{j=0}^{m-1} a_j x^j$ on ω_m^k.
- So our evaluation task is just DFT.

All the essences are here…

- We want to evaluate $A(x)$ and $B(x)$ on ω_{2n}^{0}, ω_{2n}^{1}, …, ω_{2n}^{2n-1}.
- Suppose for simplicity that n is a power of 2.
- For $A(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1}$, define two new polynomials:
  - $A_0(x) = a_0 + a_2 x + a_4 x^2 + \cdots + a_{n-2} x^{n/2-1}$
  - $A_1(x) = a_1 + a_3 x + a_5 x^2 + \cdots + a_{n-1} x^{n/2-1}$
- Then $A(x) = A_0(x^2) + x A_1(x^2)$
  - Divide and Conquer.
- So we can evaluate $A(x)$ by evaluating $A_0(x)$ and $A_1(x)$ on $(\omega_{2n}^{0})^2, (\omega_{2n}^{1})^2, \ldots, (\omega_{2n}^{2n-1})^2$.

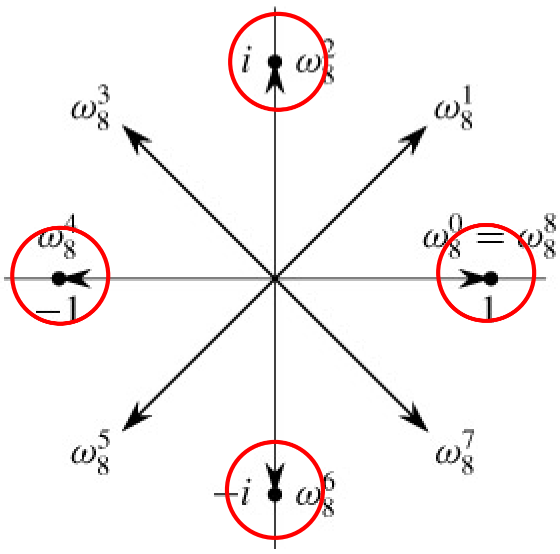
Distinct points

- Note:

$(\omega_{2n}^{0})^2, (\omega_{2n}^{1})^2, \dots, (\omega_{2n}^{2n-1})^2$

are not all distinct.

- Only n values, each repeating twice!



- So we evaluate only n points (instead of $2n$).
- Recursion: $T(2n) = 2T(n) + O(n)$
  - $O(n)$: To compute $A(\omega_{2n}^{2i}) = A_0(\omega_{2n}^{2i}) + xA_1(\omega_{2n}^{2i})$ for $i = 0, 1, \ldots, 2n - 1$.
- Rewriting recursion: $T(k) = 2T(k/2) + O(k)$
  - $k = 2n$.
- Applying master theorem: $T(k) = O(k \log k)$.
- Since $n = k/2$, the cost of evaluating $A(x)$ is $T(n) = O(n \log n)$, as claimed.

Interpolation

- How about interpolation?
  - i.e., to get the coefficients by point values.
- Almost the same process!

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n & \omega_n^2 & \omega_n^3 & \cdots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \omega_n^6 & \cdots & \omega_n^{2(n-1)} \\ 1 & \omega_n^3 & \omega_n^6 & \omega_n^9 & \cdots & \omega_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \omega_n^{3(n-1)} & \cdots & \omega_n^{(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{pmatrix}$$

- $y = Fa \Leftrightarrow a = F^{-1}y$.

- DFT matrix: $F = \left[\omega_n^{jk}\right]_{jk}$.
- What's the inverse of the matrix F?

- Pretty much the same matrix

replace ω_n^{jk} with ω_n^{-jk}.

  - …and then divide the whole matrix by n for normalization.

- Why?
- You can directly check by multiplying the two matrices and get I (the identity matrix).
- e.g. $\frac{1}{3}\begin{bmatrix} 1 & 1 & 1 \\ 1 & \omega_3 & \omega_3^2 \\ 1 & \omega_3^2 & \omega_3 \end{bmatrix}\begin{bmatrix} 1 & 1 & 1 \\ 1 & \omega_3^{-1} & \omega_3^{-2} \\ 1 & \omega_3^{-2} & \omega_3^{-1} \end{bmatrix} = I$

Or

$$F = \frac{1}{\sqrt{3}} \begin{bmatrix} 1 & 1 & 1 \\ 1 & \omega_3 & \omega_3^2 \\ 1 & \omega_3^2 & \omega_3 \end{bmatrix}$$

- DFT matrix is unitary.
- Namely $F^{-1} = (F^T)^*$
  - T: transpose. $*$: complex conjugate
- DFT is symmetric: $F^T = F$.
- So taking complex conjugate ($\omega_n^{jk} \to \omega_n^{-jk}$) gives inverse.

Summary

- Divide and conquer is a general method to design algorithms.
- Master theorem to compute the complexity.
- Several examples.
  - Merge sort
  - Selection
  - Matrix multiplication
  - FFT