CS483 Analysis of Algorithms
Lecture 03 – Divide-n-Conquer *

Jyh-Ming Lien

February 06, 2008

*this lecture note is based on *Algorithms* by S. Dasgupta, C.H. Papadimitriou, and U.V. Vazirani and *Introduction to the Design and Analysis of Algorithms* by Anany Levitin.

Today, we will learn...

- In this lecture we will two main topics:
  - Sort and selection
    - Mergesort and quicksort
    - Binary search
    - Closest-pair and convex-hull algorithms
  - Multiplication
    - Multiplication of large integers
    - Matrix multiplication
    - Polynomial multiplication
- We will approach these problems using the divide-and-conquer technique

Introduction

Divide and Conquer

- Divide and conquer was a successful military strategy long before it became an algorithm design strategy
  - **Coalition uses divide-conquer plan in Fallujah**

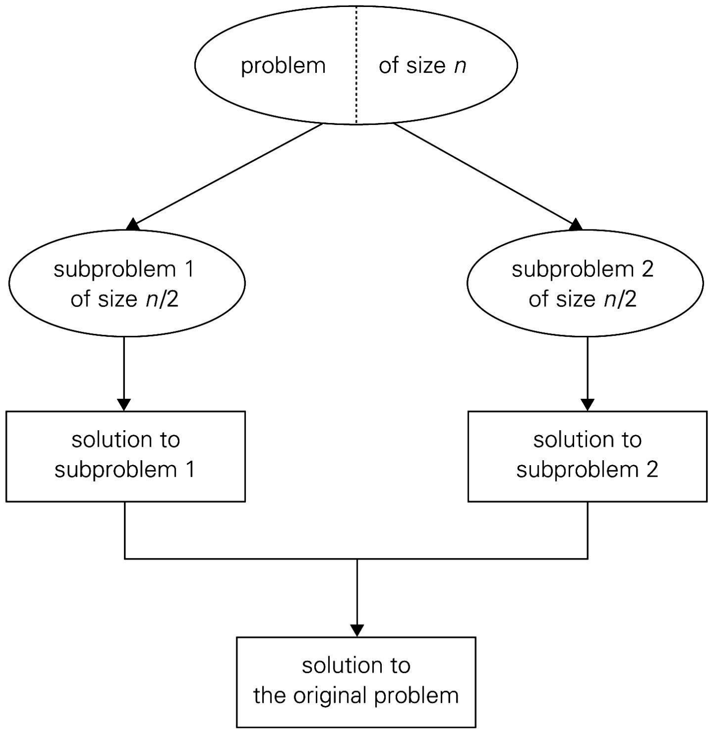
    By Rowan Scarborough and Bill Gertz, THE WASHINGTON TIMES

    Coalition troops are employing a divide-and-conquer strategy in Fallujah, Iraq, capitalizing on months of pinpointed intelligence to seal off terrorist-held neighborhoods and then attack enemy pockets.

- Example: Your CS 483 instructor give you a 50-question assignment today and ask you to turn it in the tomorrow. What should you do?

- The most-well known algorithm design strategy:

1. Divide instance of problem into two or more smaller instances
2. Solve smaller instances recursively
3. Obtain solution to original (larger) instance by combining these solutions



Divide and Conquer Examples

- Example: Given a list $A = \{2, 3, 6, 4, 12, 1, 7\}$, compute $\sum_{i=1}^{7} A_i$

Master Theorem

- If we have a problem of size n and our algorithm divides the problems into b instances, with a of them needing to be solved. Then we can set up our running time $T(n)$ as: $T(n) = aT(n/b) + f(n)$, where $f(n)$ is the time spent on dividing and merging.
- **Master Theorem**: If $f(n) \in \Theta(n^d)$, with $d \geq 0$, then

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

- Examples:

1. $T(n) = 4T(n/2) + n \Rightarrow T(n) =$

2. $T(n) = 4T(n/2) + n^2 \Rightarrow T(n) =$

3. $T(n) = 4T(n/2) + n^3 \Rightarrow T(n) =$

Sort & Select

- Given an array of n numbers, sort the element from small to large.

Algorithm 0.1: $\textsc{MergeSort}(A[1 \cdots n])$

Sorting: Mergesort

- Merge two sorted arrays, B and C and put the result in A

Algorithm 0.2: $\textsc{Merge}(B[1 \cdots p], C[1 \cdots q], A[1 \cdots p+q])$

Sorting: Mergesort Example

- Example: 24, 11, 91, 10, 22, 32, 22, 3, 7, 99

- Is Mergesort stable?

Analysis of Merge Sort

- $C_{worst}(n)$

Sorting: Quicksort

- Given an array of n numbers, sort the element from small to large.

Algorithm 0.3: QUICKSORT($A[1 \cdots n]$)

- $A[1]$ in the above algorithm is called **pivot**

Sorting: Quicksort Example

- Example: 24, 11, 91, 10, 22, 32, 22, 3, 7, 22

- Is Quicksort stable?

Analysis of Quicksort

- $C_{worst}(n)$

- $C_{best}(n)$

- $C_{avg}(n)$

Why is Quicksort quicker?

- Because quicksort allows very fast "in-place partition"

Algorithm 0.4: $\textsc{Partition}(A[a \cdots b])$

The Selection Problem

- Similar to partition in quicksort!
- Find the k-th smallest element in an array A with n unique elements

Algorithm 0.5: SELECT$(A[1 \cdots n], k)$

- The algorithm above will work well for A with unique elements. How do you change to make it work for more general cases?
- Time complexity:

- Imagine that you are placed in an unknown building and you are given a room number, you need to find your CS 483 instructor. What will you do?
- **Binary Search**:
  - Very efficient algorithm for searching in **sorted array**
    Example: find 70 in {3, 14, 27, 31, 39, 42, 55, 70, 74, 81, 85, 93, 98}
  - Efficient search in even in high dimensional unknown space
    Example:

Binary Search

- Given a sorted array A of n numbers, find a key K in A

Algorithm 0.6: $\textsc{BinarySearch}(A[1 \cdots n], K)$

- Binary search is in fact a bad (degenerate) example of divide-and-conquer

Analysis of Binary Search

- $C_{worst}(n)$

- $C_{best}(n)$

- $C_{avg}(n)$

Closest Pair

- Find the closest distance between points in a given point set

Algorithm 0.7: CP($P[1 \cdots n]$)

comment: P is a set n points

Closest Pair

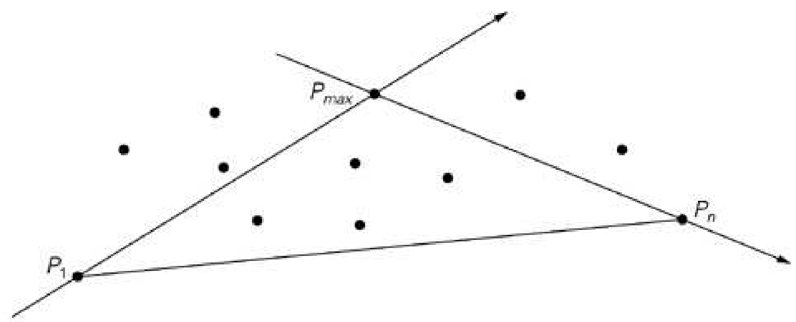
- Find the closest distance between points in a given point set

Algorithm 0.8: COMBINE(c, P, P_1, P_2, d)

- What is the time complexity?

Convex Hull

- Here we consider a divide-and-conquer algorithm called **quickhull**
- Quickhull is similar to quicksort
  why?
- Observations (given a point set P in 2-d):
  - The leftmost and rightmost points in P must be part of the convex hull
  - The furthest point away from any line must be part of the convex hull
  - Points in the triangle formed by any three points in P will **not** be part of the convex hull



- Qhull

Algorithm 0.9: QHULL$(P[1 \cdots n])$

comment: P is a set n points

- Animation: *http://www.cs.princeton.edu/∼ah/alg_anim/version1/QuickHull.html*

Analysis of Quickhull

- Worst case:
- Best case:
- Avg case:

Multiplication

- What is the time complexity of multiplying two integers using the algorithms we learned in elementary schools?
  Example: how do you compute this: 12345×67890?

- Is there a better way of multiplying two intergers than this elementary-school method?
  Carl Friedrich Gauss (1777-1855) discovered that
  $AB = (a10^{\frac{n}{2}} + b)(c10^{\frac{n}{2}} + d) =$

  Example: how do you compute this: 12345×67890?

Carl Friedrich Gauss

Interger multiplication

- Divid-and-conquer interger multiplication

Algorithm 0.10: $\mathrm{M}(A[1 \cdots n], B[1 \cdots n])$

- What is the time complexity?

Matrix multiplication

Strassen's Matrix Multiplication:

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & A_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$= \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix}$$

- ☐ $m_1 = (A_{11} + A_{22})(B_{11} + B_{22})$
- ☐ $m_2 = (A_{21} + A_{22})B_{11}$
- ☐ $m_3 = A_{11}(B_{12} - B_{22})$
- ☐ $m_4 = A_{22}(B_{21} - B_{11})$
- ☐ $m_5 = (A_{11} + A_{12})B_{22}$
- ☐ $m_6 = (A_{21} - A_{11})(B_{11} + B_{12})$
- ☐ $m_7 = (A_{12} - A_{22})(B_{21} + B_{22})$

Matrix multiplication

- What is the time complexity?

- Do you still remember what the time complexity of the brute-force algorithm is?

Polynomial multiplication

- two degree-n polynomials:

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

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

- Multiplication of two degree-n polynomial

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

- The coefficient c_k is:

- A brute force method for computing $C(x)$ will have time complexity=

- Can we do better?

Representing polynomial

- **Fact**: A degree-n polynomial is uniquely defined by any $n+1$ distinct points
- A degree-n polynomial $A(x)$ can be represented by:
  –
  –
- We can convert between these two representations: 1.5cm
- The value representation allows us to develop faster algorithm!
  – We only need $2n+1$ points for $C(x)$
  – It's easy and efficient to generate these $2n+1$ points from $A(x)$ and $B(x)$

Polynomial multiplication

- General idea:

  1. Convert A and B to value representation (Evaluation)
  2. Perform multiplication to obtain C in value representation
  3. Convert C back to coefficient representation (Interpolation)

Coefficient representation

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

$\xrightarrow{\textbf{Multiplication } O(n^2)}$

$$C(x) = c_{2n} x^{2n} + c_{2n-1} x^{2n+1} + \ldots + c_1 x + c_0$$

$\big\downarrow$ **Evaluation** $O(n \log n)$ $\qquad$ $\big\uparrow$ **Interpolation** $O(n \log n)$

$$A(x_0), A(x_1), \ldots, A(x_{2n})$$
$$B(x_0), B(x_1), \ldots, B(x_{2n})$$

$\xrightarrow{\textbf{Multiplication } O(n)}$

$$C(x_0), C(x_1), \ldots, C(x_{2n})$$

$C(x_i) = A(x_i)B(x_i)$

Value representation

Polynomial evaluation

- $f(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$
- Polynomial evaluation: Given x, compute $f(x)$
- Brute force algorithm

Algorithm 0.11: F(x)

- Time complexity of this brute force algorithm?
- Can we do better?

Horner's Rule

- Horner's rule

$$\begin{aligned} f(x) &= a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0 \\ &= (a_n x^{n-1} + a_{n-1} x^{n-2} + \cdots + a_1) x + a_0 \\ &= (\cdots (a_n x + a_{n-1}) x + \cdots) x + a_0 \end{aligned}$$

- Polynomial evaluation using Horner's rule

Algorithm 0.12: F(x)

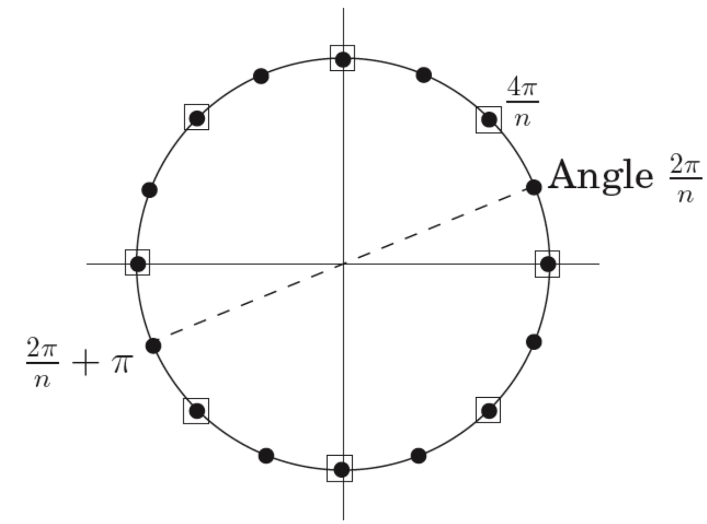
- Time complexity:
- Example: $f(x) = 2x^4 - x^3 + 3x^2 + x - 5$ at $x = 4$

A $n \log n$ time polynomial evaluation

- Basic idea: How we select x_i affects the run time.
- Example: If we pick $\pm x_0, \pm x_1, \ldots, \pm x_{n/2-1}$, then $A(x_i)$ and $A(-x_i)$ have many overlap
  - $x^5 + 2x^4 + 3x^3 + 4x^2 + 5x + 6 =$
  - $A(x) =$
  - When evaluate x_i, $A(x_i) =$
  - When evaluate $-x_i$, $A(-x_i) =$
- What we need is x_i such that

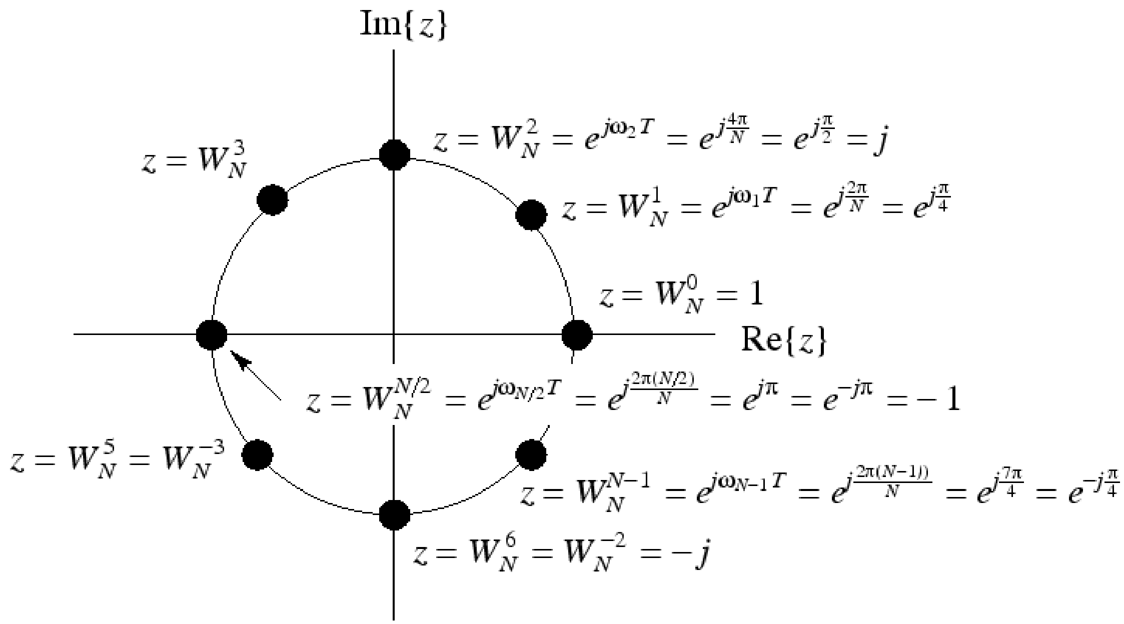
n-th roots of unity

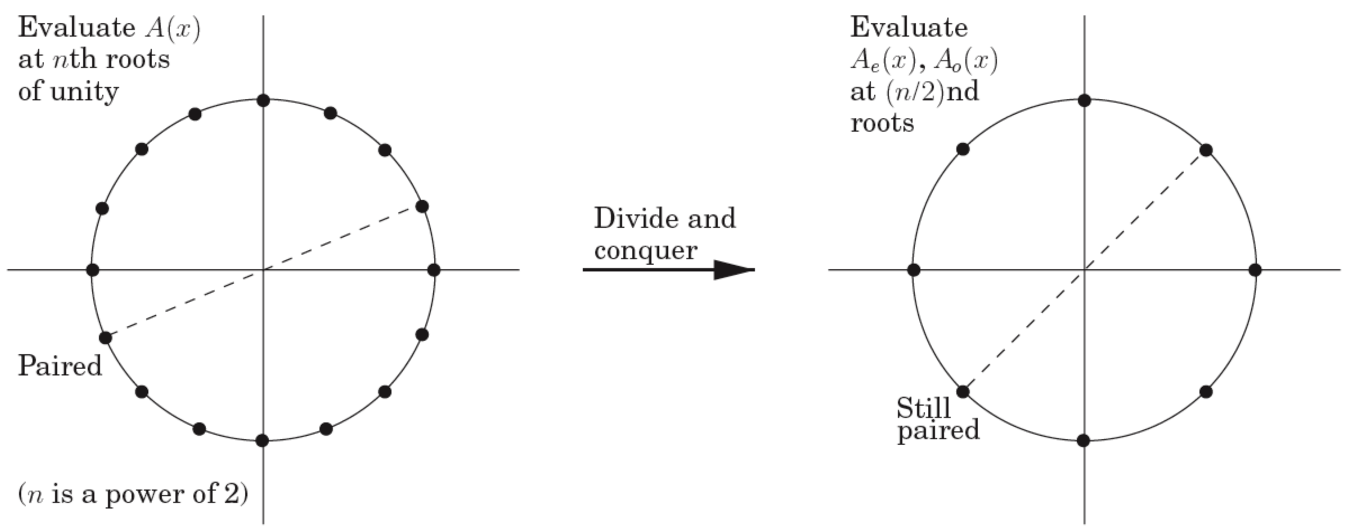
- **Idea**: Use n-th roots of unity: $z^n = 1$ as our x_i
- Background:
  - Complex number $z = r(\cos(\theta) + i\sin(\theta))$
    - Usually denoted as $re^{i\theta}$ or (r, θ)
    - $(r_1, \theta_1) \times (r_2, \theta_2) = (r_1 r_2, \theta_1 + \theta_2)$
  - Let $\omega_n = \cos(\frac{2\pi}{n}) + i\sin(\frac{2\pi}{n}) = e^{2\pi i/n}$ be a complex n-th root of unity
  - Other roots include: $\omega_n^2, \omega_n^3, \ldots, \omega_n^{n-1}, \omega_n^n$
  - Properties:
    - $\omega_n^j = -\omega_n^{j+n/2}$
    - Therefore, $(\omega_n^j)^2 = (-\omega_n^{j+n/2})^2$
    - Moreover, $(\omega_n^j)^2 = \omega_{n/2}^j$
    - $\sum_{i=1}^n \omega_n^i = \frac{1-\omega_n^n}{1-\omega_n} = 0$



n-th roots of unity

- Examples $n = 8$:





- FFT

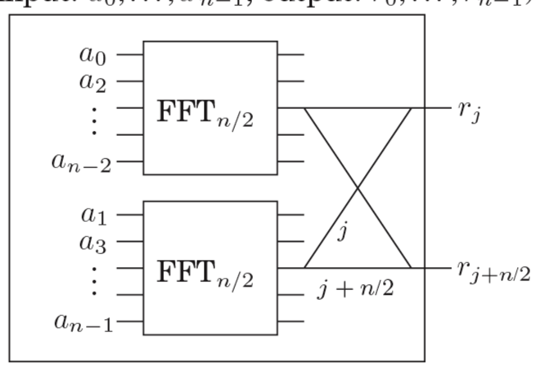
Algorithm 0.13: $\text{FFT}((a_0, a_1, a_2, \ldots, a_{n-1}), \omega)$

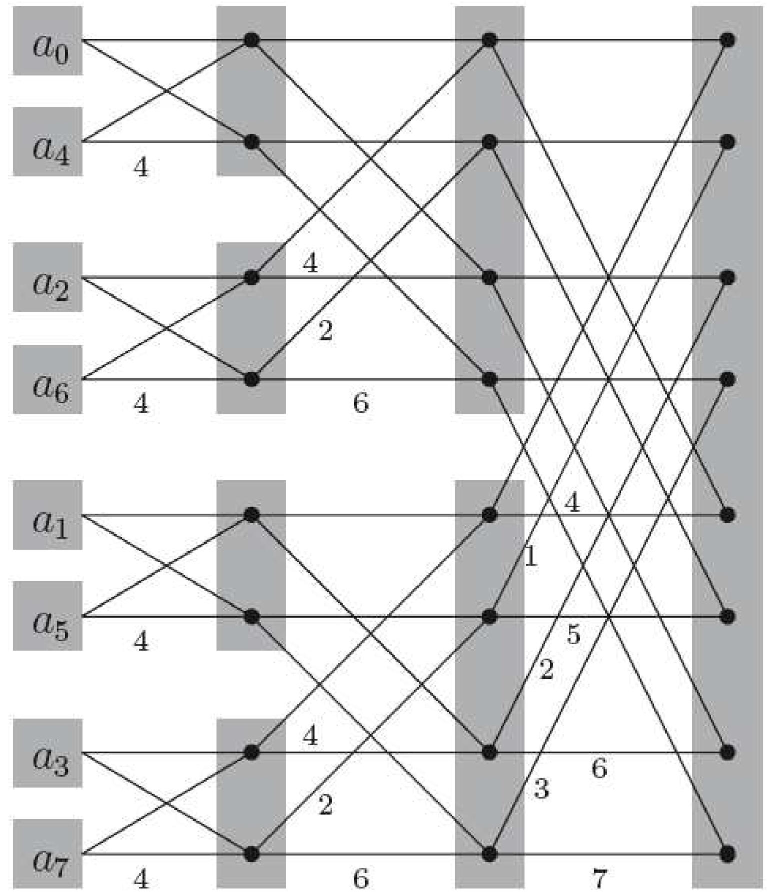
- Time complexity?

A $n \log n$ time polynomial evaluation

- Hardware implementation

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





A $n \log n$ time polynomial interpolation

- Convert the values $C(x_i)$ back to coefficients: $\{c_i\}$=FFT($C(x_i), \omega^{-}1$)
- Here is why

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

- $M_n(\omega) =$

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

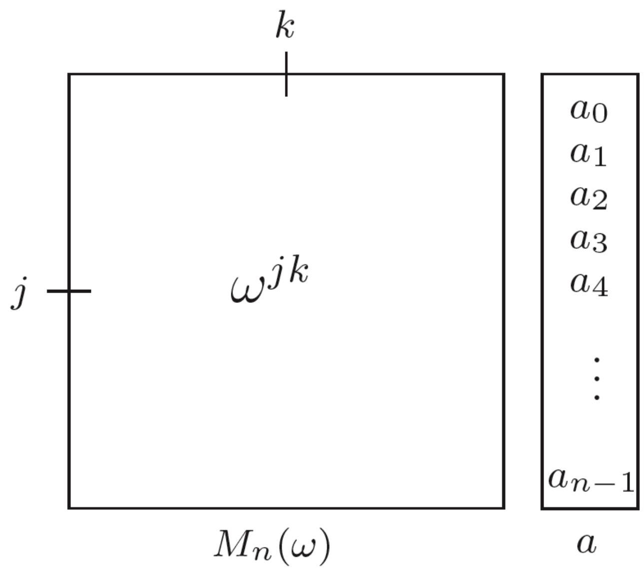
- Entry (j, k) of M_n is ω^{jk}

A $n \log n$ time polynomial interpolation

- $M_n(\omega)$ is invertible, i.e., column j and column k are orthogonal
  - *proof*:

- Inversion formula $M_n(\omega)^{-1} = \frac{1}{n} M_n(\omega^{-1})$
  - *proof*:

A closer look



Conclusion

Summary

- Summary
  - Sort and select
    - Mergesort and quicksort [1]
    - Binary search
    - Closest-pair and convex-hull algorithms
  - Multiplication
    - Multiplication of large integers - from Gauss
    - Matrix multiplication
    - Polynomial multiplication - FFT[1] (Also from Gauss)
- Divide-n-conquer strategy
  - Advantages of
    - Make problems easier
    - Easy parallelization
  - Disadvantages of Divide-n-conquer strategy
    - Recursion can be slow
    - Subproblems may overlap

[1] Named one of 10 best algorithms in last century