南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

05 Divide and Conquer

CS216 Algorithm Design and Analysis (H)

Instructor: Shan Chen

chens3@sustech.edu.cn

Divide-and-Conquer Paradigm

- **Divide and Conquer:**
  - Divide up problem into several subproblems (of the same kind).
  - Conquer (solve) each subproblem recursively.
  - Combine solutions to subproblems into solution to original problem.
- **Most common usage:**
  - Divide problem of size n into two subproblems of size $n/2$.
  - Conquer (solve) two subproblems recursively.
  - Combine two solutions into solution to original problem.
- **Common time complexity:**
  - Brute force: $\Theta(n^2)$
  - Divide and conquer: $O(n \log n)$ ← $T(n) = 2T(n/2) + O(n)$

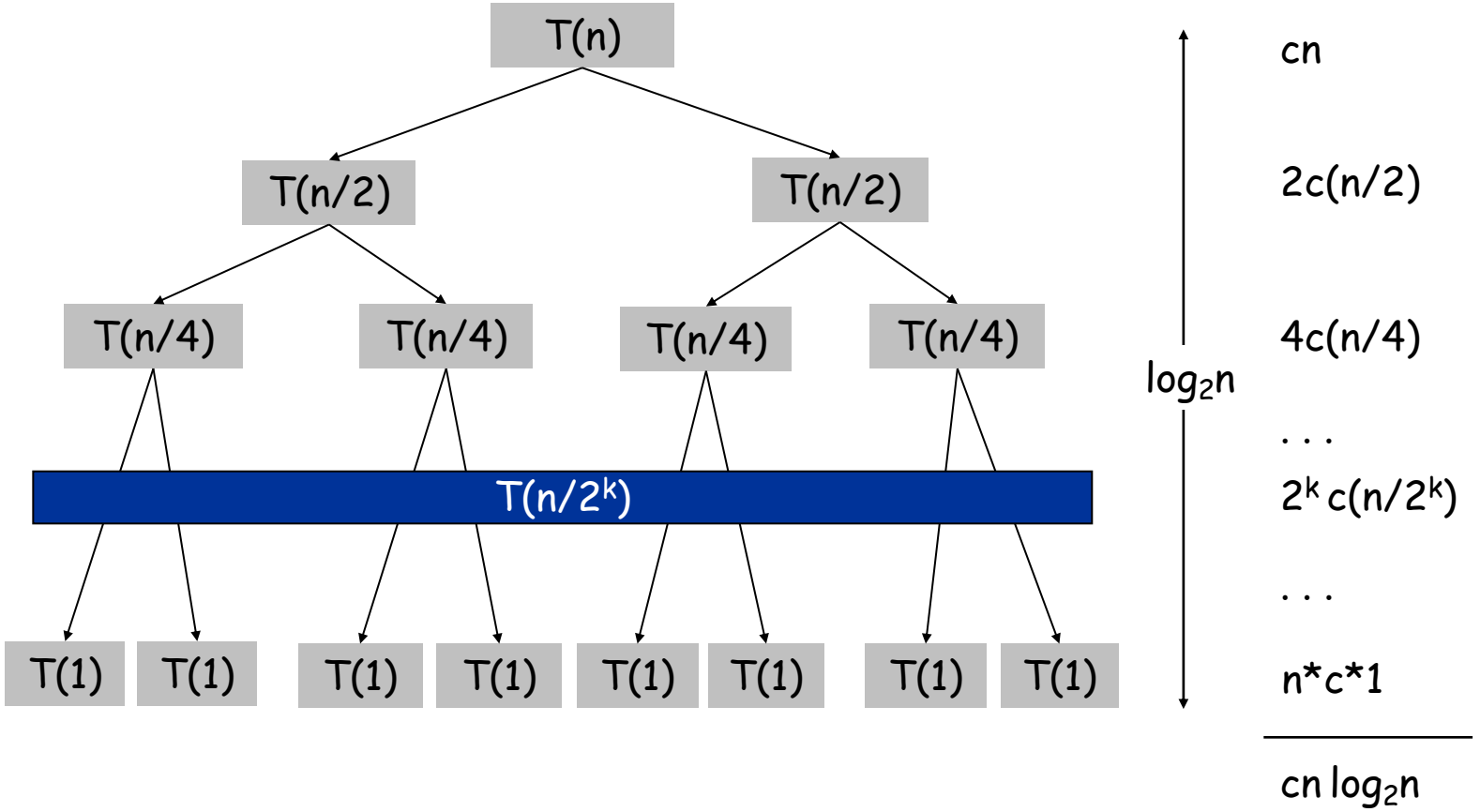
e.g., combine in $O(n)$ time

凡治众如治寡 分数是也

孙子兵法

Divide-and-Conquer Recurrences

- **Example:** $T(n) = 2T(n/2) + O(n)$ with $T(1) = \Theta(1)$



Divide-and-Conquer Recurrences

- **Divide-and-Conquer recurrences.** $T(n) = aT(n/b) + f(n)$ with $T(1) = \Theta(1)$

- **Master Theorem.** Let $a \geq 1$, $b \geq 2$, $c \geq 0$ and suppose $T(n)$ is a function on the non-negative integers that satisfies the recurrence

$$T(n) = aT(n/b) + \Theta(n^c)$$

with $T(1) = \Theta(1)$, where n/b means either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then,

  - Case 1: If $c > \log_b a$, then $T(n) = \Theta(n^c)$.
  - Case 2: If $c = \log_b a$, then $T(n) = \Theta(n^c \log n)$.
  - Case 3: If $c < \log_b a$, then $T(n) = \Theta(n^{\log_b a})$.

1. Counting Inversions

Counting Inversions

- **Match song preferences.**
  - Every person ranks n songs.
  - Goal: Music site consults database to find people with similar tastes.
- **Similarity metric.** Number of inversions between two rankings.
  - My rank: $1, 2, \ldots, n$
  - Your rank: $a_1, a_2, \ldots, a_n$
  - Songs i and j are inverted if $i < j$ but $a_i > a_j$.

Songs

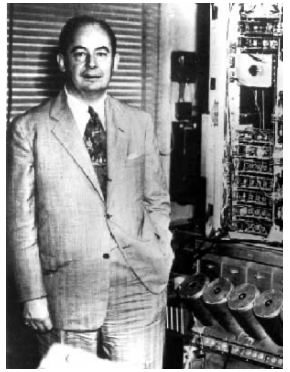
	A	B	C	D	E
Me	1	2	3	4	5
You	1	3	4	2	5

inversions
3-2, 4-2

- **Brute Force.** Check all $\Theta(n^2)$ pairs.
- **Q.** Can we count inversions faster?

Recall: Merge Sort

- **Merge Sort:**
  - Divide array into two halves.
  - Recursively sort each half.
  - Merge two halves to make sorted whole.



Jon von Neumann (1945)

A	L	G	O	R	I	T	H	M	S

A	L	G	O	R

I	T	H	M	S

divide $O(n)$

A	G	L	O	R

H	I	M	S	T

sort $2T(n/2)$

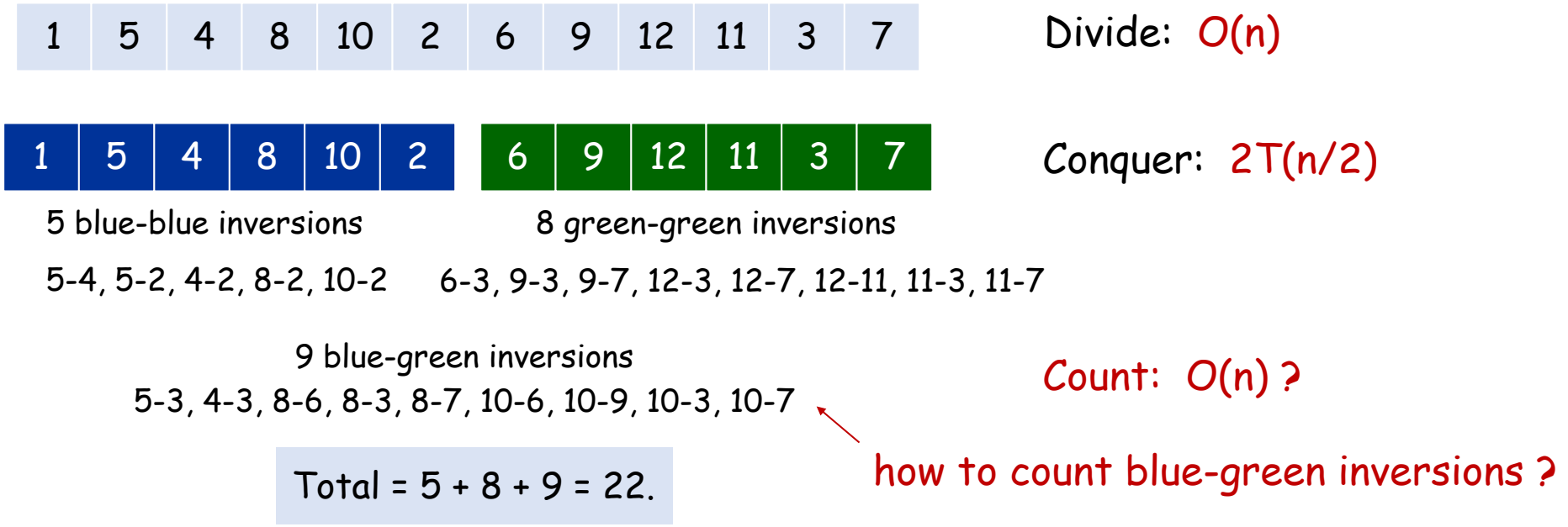
A	G	H	I	L	M	O	R	S	T

merge $O(n)$

total $O(n \log n)$

Counting Inversions: Divide and Conquer

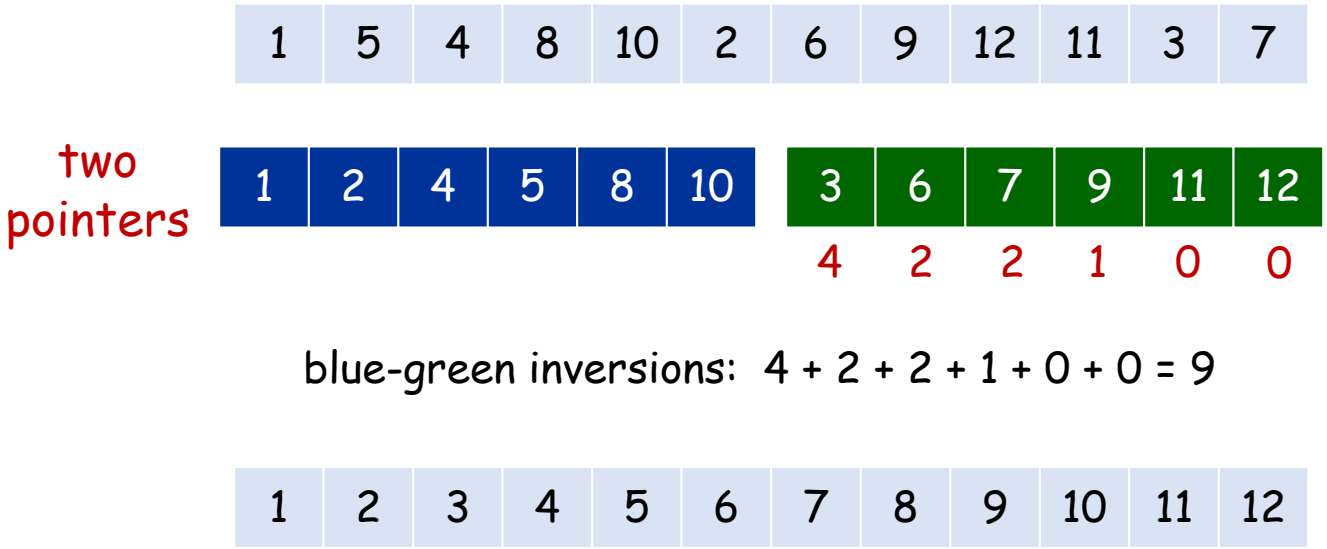
- **Divide and Conquer:**
  - Divide: separate list into two pieces.
  - Conquer: recursively count inversions in each half.
  - Combine: count inversions where a_i and a_j are in different halves and return sum of three quantities.



Counting Inversions: Divide and Conquer

- **Divide and Conquer:**
  - Divide: separate list into two pieces.
  - Conquer: recursively count inversions in each half.
  - Combine: count inversions where a_i and a_j are in different halves and return sum of three quantities.

easy to count if both sorted



Divide: $O(n)$

Conquer: $2T(n/2)$

Count: $O(n)$

Merge-and-Count: $O(n)$

Merge: $O(n)$

Counting Inversions: *O(n log n)* Algorithm

- **Sort-and-Count algorithm:**

```
Sort-and-Count(L) {
   if (list L has one element) return (0, L)

   Divide the list into two halves A and B
      (r_A, A) ← Sort-and-Count(A)
      (r_B, B) ← Sort-and-Count(B)
      (r_AB, L) ← Merge-and-Count(A, B)

   return (r_A + r_B + r_AB, L)
}
```

- **Merge-and-Count sub-algorithm:**
  - Input: *A* and *B* are sorted (by Sort-and-Count).
  - Output: *L* is sorted.

2. Closest Pair of Points

Closest Pair of Points

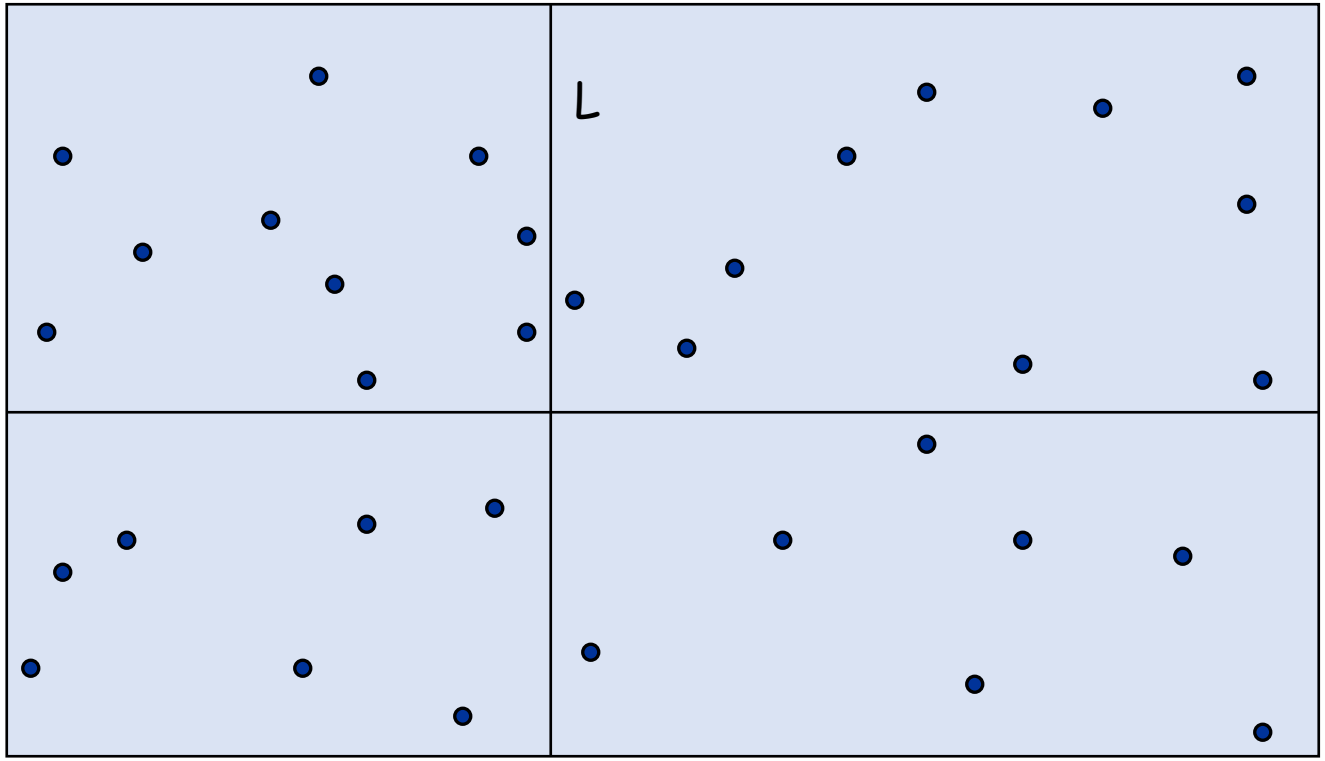
- **Closest pair problem.** Given n points in the plane, find a pair with smallest Euclidean distance between them.
- **Fundamental geometric primitive:**
  - Example applications: graphics, computer vision, geographic information systems, molecular modeling, air traffic control, etc.
  - Special case of nearest neighbor, Euclidean MST, Voronoi diagram, etc.

fast closest pair inspired fast algorithms for these problems

- **Brute Force.** Check all pairs with $\Theta(n^2)$ distance calculations.
- **1-D version.** Easy $O(n \log n)$ algorithm if points are on a line.

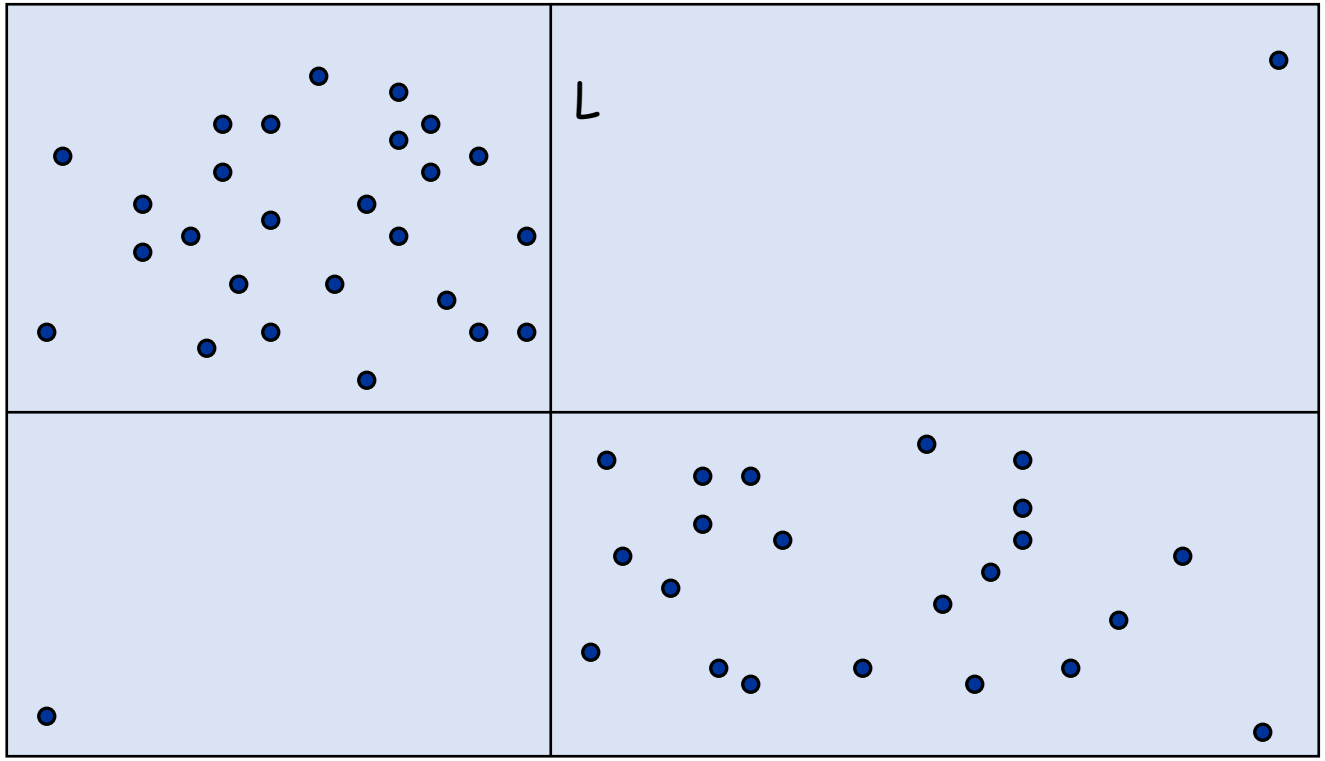
Closest Pair of Points: A First Attempt

- **Divide.** Sub-divide region into *4* quadrants.



Closest Pair of Points: A First Attempt

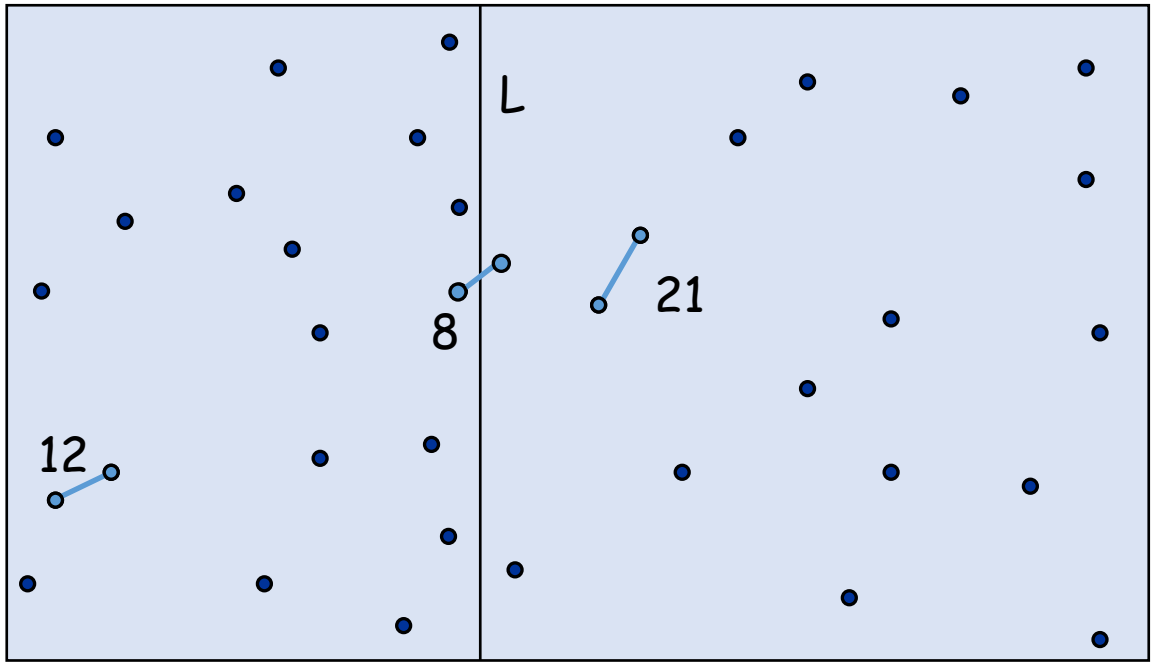
- **Divide.** Sub-divide region into *4* quadrants.
- **Obstacle.** Impossible to ensure *n*/*4* points in each quadrant.



Closest Pair of Points: Divide and Conquer

- **Divide and Conquer:**
  - Divide: draw vertical line *L* so that roughly *n*/2 points lie in each side.
  - Conquer: find closest pair in each side recursively.
  - Combine: find closest pair with one point in each side and return best of *3* solutions.

can we beat $\Theta(n^2)$?



Closest Pair of Points: Divide and Conquer

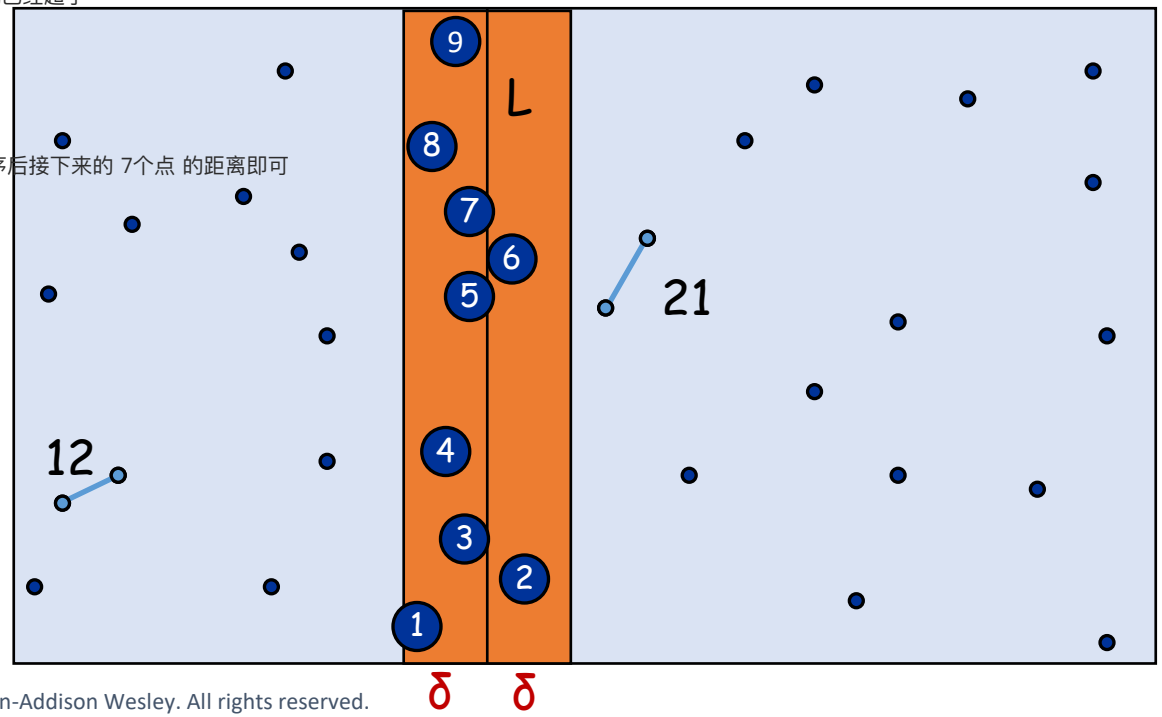
- **Combine step.** Find closest pair with one point in each side.
  - Observation: suffices to consider only points within δ of line L, where δ is the distance of closest pair with both points in one side.
  - Sort points in 2δ-strip by their y-coordinate.
  - Check distances of only those points in the sorted list within 7 positions!

known from recursion

why this works?

只有在这个strip里面，才可能更短；超出这个strip，直线距离就已经超了

对于条带内的每个点 pi ， 只需要检查它与y坐标排序后接下来的 7个点 的距离即可



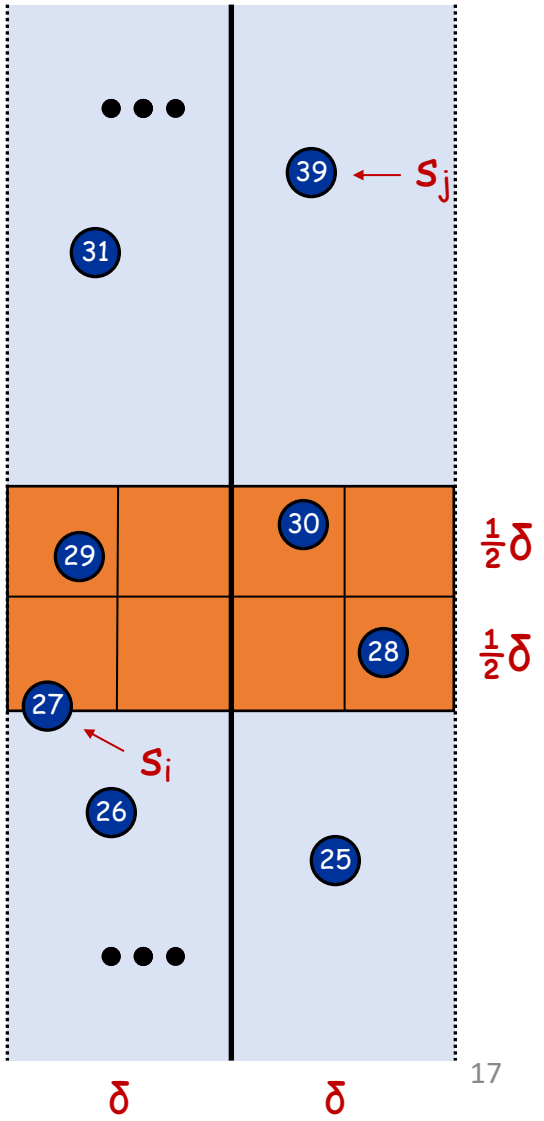
δ = min (12, 21)

Closest Pair of Points: Divide and Conquer

只需要检查它与y坐标排序后接下来的 7个点；在排序数组中的位序大于7，就一定不是

- **Claim.** Let s_i be the point in the 2δ-strip with the i-th smallest y-coordinate. If $|i - j| > 7$, then the distance between s_i and s_j is at least δ.
- **Pf.** (direct proof)
  - Consider the 2δ-by-δ rectangle R in strip whose min y-coordinate is y-coordinate of s_i.
  - Distance between s_i and any point s_j above R is $\geq \delta$.
  - Subdivide R into 8 squares.
  - At most 1 point per square. ← because square diameter < δ
  - At most 7 other points can be in R. ▪

constant can be improved with more refined argument



每个小方格最多1个点：

- 每个小方格的边长是 $\frac{\delta}{2}$，对角线长度是 $\sqrt{(\frac{\delta}{2})^2+(\frac{\delta}{2})^2}=\frac{\delta}{\sqrt{2}}<\delta$。
- 但是，条带内的所有点之间的距离必须 $\geq\delta$（因为 δ 是左右子问题中已知的最小距离）。
- 因此，任何两个点之间的距离小于 δ 是不可能的，所以每个小方格内最多只能有1个点（否则两个点之间的距离会小于 δ，与 δ 的定义矛盾）。

. **矩形 R 最多有8个点**：

- 既然有8个小方格，每个小方格最多1个点，那么矩形 R 内最多有8个点（包括 s_i 本身）。

. **y坐标排序后最多检查7个点**：

- 按y坐标排序后，s_i 是矩形 R 中y坐标最小的点（因为 R 的y坐标范围是 $[y_i, y_i+\delta]$）。
- 矩形 R 内最多有8个点（包括 s_i），所以 s_i 之后最多有7个点（$s_{i+1}, s_{i+2}, \ldots, s_{i+7}$）。
- 如果 $|i-j|>7$，那么 s_j 的y坐标 $y_j>y_i+\delta$（因为 R 最多容纳8个点），因此 s_i 和 s_j 的距离至少是y坐标差 $y_j-y_i>\delta$，必然 $\geq\delta$。

Closest Pair of Points: $O(n \log n)$ Algorithm

- **Divide-and-Conquer algorithm:** [sort by x-axis and y-axis beforehand]

```
Closest-Pair(p1, ..., pn) {
   Compute vertical line L such that half the points
   are on each side of the line and Partition the points.

   δ1 ← Closest-Pair(left half)
   δ2 ← Closest-Pair(right half)
   δ  ← min(δ1, δ2)

   Delete all points further than δ from line L.

   Sort remaining points by y-coordinate.

   Scan points in y-order and compare distance between
   each point and next 7 neighbors. If any of these
   distances is less than δ, update δ.

   return δ
}
```

O(n) use x-sorted list

2T(n / 2)

O(n) use y-sorted list

~~O(n log n)~~ O(n)

可以最开始对x，y各自排序；后面遍历去找

O(n)

Closest Pair of Points: Closing Remarks

- [Rabin 1976] There exists a randomized algorithm that finds the closest pair of points in the plane with expected running time $O(n)$.

- **Remark.** There are ingenious divide-and-conquer algorithms for core geometric problems.
  - 3D or higher dimensions test limits of our ingenuity.

problem	brute	clever
closest pair	$O(n^2)$	$O(n \log n)$
farthest pair	$O(n^2)$	$O(n \log n)$
convex hull	$O(n^2)$	$O(n \log n)$
Delaunay/Voronoi	$O(n^4)$	$O(n \log n)$
Euclidean MST	$O(n^2)$	$O(n \log n)$

running time to solve a 2D problem with n points

Announcement

- **Lab 6 will be released today and the deadline is Apr 8.**

3. Median and Selection

An example of randomized algorithms

[section 13.5 of textbook]

Median and Selection

- **Median and selection.** Given *n* elements from a totally ordered universe, find the median element or in general the *k*-th smallest element.
  - minimum or maximum ($k = 1$ or $k = n$): $O(n)$ compares
  - median: $k = \lfloor (n + 1) / 2 \rfloor$
    - ✓ $O(n \log n)$ compares by sorting
    - ✓ $O(n \log k)$ compares with a binary heap
- **Applications.** Order statistics, find the "top *k*", bottleneck paths, etc.

堆顶是最大的

- **Q.** Can we do it with $O(n)$ compares?
- **A.** Yes! Selection is easier than sorting.

步骤 2：处理前 k 个元素

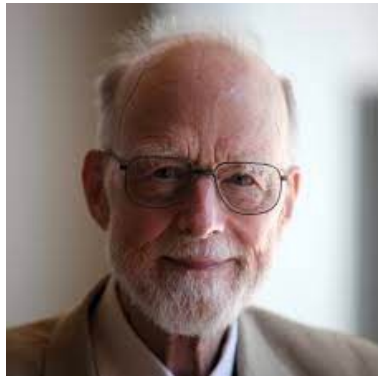
- 取数组的前 k 个元素，构建一个最大堆。
- 构建一个包含 k 个元素的堆需要 $O(k)$ 的时间复杂度（因为我们是从头开始构建堆，而不是插入操作）。

步骤 3：处理剩余的 $n - k$ 个元素

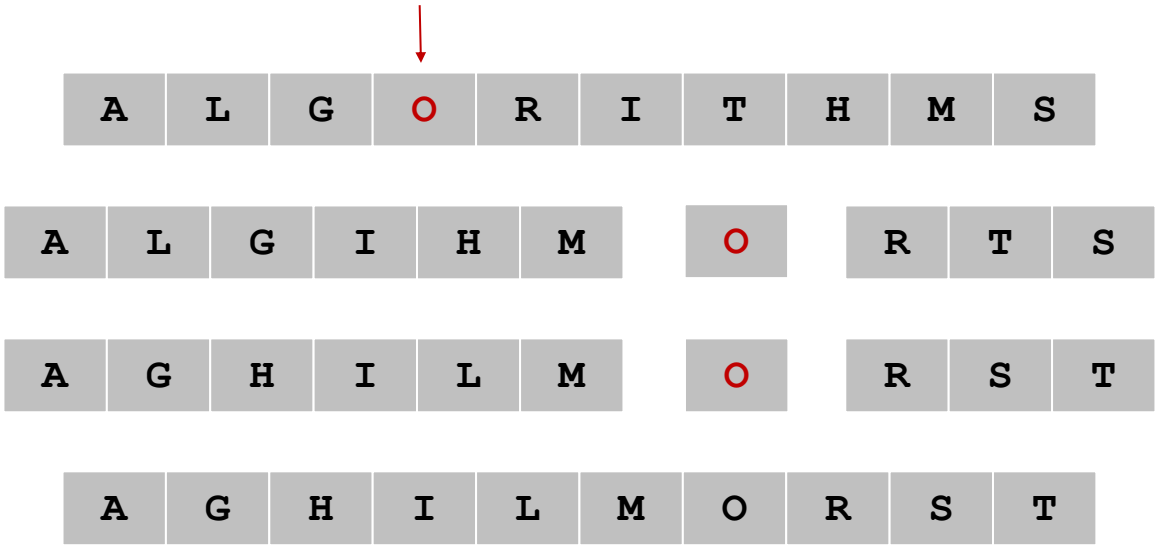
- 对于数组中剩余的 $n - k$ 个元素（从第 $k + 1$ 个元素开始），对每个元素 x 执行以下操作：
  1. 比较 x 和堆顶元素（堆顶是当前堆中最大的元素）。
  2. 如果 x 小于堆顶元素：
     - 删除堆顶元素（最大值），时间复杂度 $O(\log k)$。
     - 将 x 插入堆中，时间复杂度 $O(\log k)$。
  3. 如果 x 大于或等于堆顶元素，直接跳过 x，继续处理下一个元素。

Recall: Randomized Quicksort

- **Randomized quicksort:**
  - Pick a random pivot element *p*.
  - *3*-way partition the array into *L, M,* and *R*.
    - ✓*L*: elements < *p*, *M*: elements = *p*, *R*: elements > *p*.
  - Recursively sort both *L* and *R*.

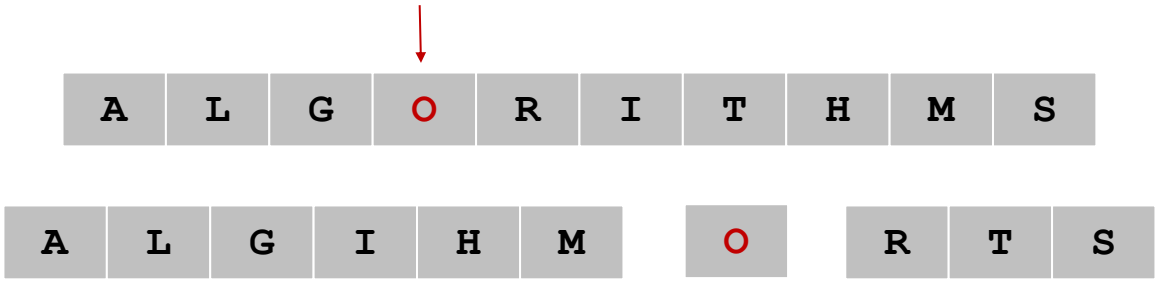


Tony Hoare (1959)



Median and Selection: Divide-and-Conquer

- **Divide and conquer:**
  - Pick a random pivot element *p*.
  - *3*-way partition the array into *L, M,* and *R*.
    - ✓*L*: elements < *p*, *M*: elements = *p*, *R*: elements > *p*.
  - Recursively select in one subarray: the one containing the *k*-th smallest element.



pick random pivot $O(1)$

3-way partition $O(n)$

select $T(|L|)$ or $O(1)$ or $T(|R|)$

Randomized Quickselect

- **Randomized quickselect.** Divide and select.

```
Quick-Select(A, k) { // 1 ≤ k ≤ |A|
   Pick pivot p uniformly at random from A
   Partition the list into two three parts L, M and R

   if (k ≤ |L|)
      return Quick-Select(L, k)
   else if (k > |L| + |M|)
      return Quick-Select(R, k - |L| - |M|)
   else
      return p
}
```

我还是要找第k小的

不够小，就往左边找

- **Q.** What is the expected time complexity of randomized quickselect?
  - Time complexity is measured by the number of compares.

Randomized Quickselect: Time Complexity

正好划分成两等份，和找到最值；两种情况的平均水平

- **Intuition.** Split a length-n array uniformly ⇒ expected larger size ~$3n/4$.
  - $T(n) \le T(3n/4) + n \Rightarrow T(n) \le 4n$

    not rigorous: cannot assume E[T(i)] ≤ T(E[i])

- **Def.** Let $T(n, k)$ be the expected number of compares to select the k-th smallest element in an array of length n. Let $T(n) = \max_k T(n, k)$.

最坏情况下的期望比较次数（对所有可能的k 取最大值）

- **Claim.** $T(n) \le 4n$
- **Pf.** (by strong induction on n)

T(i) ≤ T(n - i) since T(n) is monotonely non-decreasing

主元落在每个位置的可能一样，都是1/n

本次分区比较数量

$$
\begin{aligned}
T(n) &\le n + 1/n \, [\, 2T(n/2) + \ldots + 2T(n-3) + 2T(n-2) + 2T(n-1) \,] \\
&\le n + 1/n \, [\, 8(n/2) + \ldots + 8(n-3) + 8(n-2) + 8(n-1) \,] \\
&\le n + 1/n \, (3n^2) \\
&= 4n
\end{aligned}
$$

inductive hypothesis

Median and Selection: Closing Remarks

- We learned that randomized quickselect runs in $O(n)$ time on average.

- [Blum-Floyd-Pratt-Rivest-Tarjan 1973] There exists a compare-based deterministic selection algorithm whose worst-case running time is $O(n)$.
  - This algorithm is also known as median-of-median
  - Optimized version requires $\leq 5.4305n$ compares.

Median-of-Medians 算法的核心思想：

1. **分组**：将数组分成 $\lceil n/5 \rceil$ 组，每组 5 个元素（最后一组可能不足 5 个）。
2. **找每组中位数**：对每组的 5 个元素排序（可以用插入排序，5 个元素排序是 $O(1)$），找到每组的中位数（第 3 小的元素）。
3. **递归找中位数**：将所有组的中位数（共 $\lceil n/5 \rceil$ 个）组成一个新数组，递归调用 Median-of-Medians 算法，找到这些中位数的中位数（即"中位数的中位数"）。
4. **分区**：用这个"中位数的中位数"作为主元进行分区，得到 L（小于主元）、M（等于主元）和 R（大于主元）。
5. **递归**：根据 k 的值，递归处理 L 或 R，逻辑与 Quickselect 相同。

- **Remark.** In practice, we use randomized selection algorithms since deterministic algorithms have too large constants.
  - However, deterministic algorithms can be used as a fallback for pivot selection.

4. Integer and Matrix Multiplication

Integer Addition and Subtraction

- **Addition.** Given two n-bit integers a and b, compute $a + b$.
- **Subtraction.** Given two n-bit integers a and b, compute $a - b$.
- **Grade-school addition/subtraction algorithm.** $\Theta(n)$ bit operations.

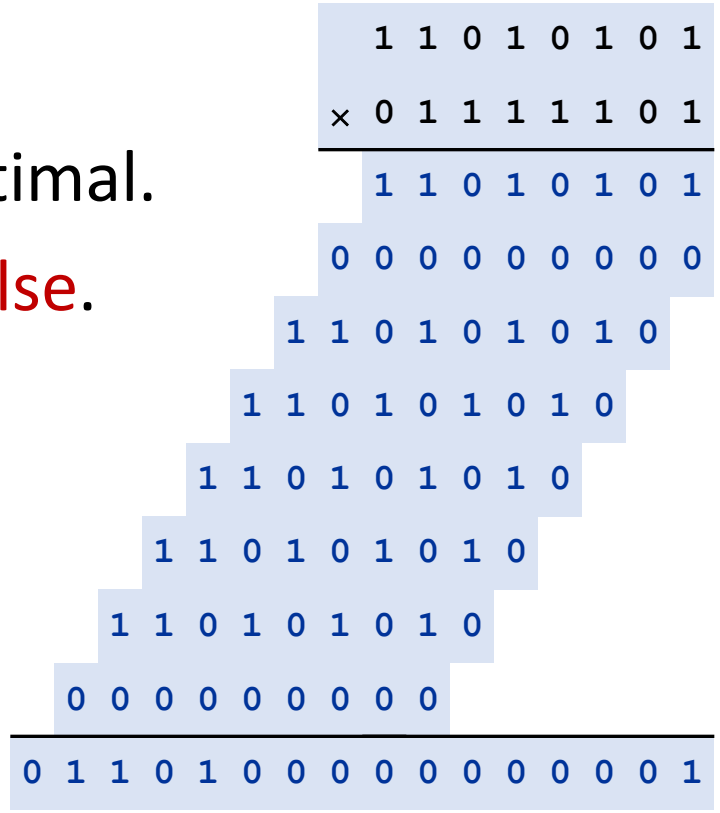
```
  1  1  1  1  1  1  0  1
     1  1  0  1  0  1  0  1
+    0  1  1  1  1  1  0  1
-------------------------
  1  0  1  0  1  0  0  1  0
```

- **Remark.** Grade-school addition and subtraction algorithms are optimal.

Integer Multiplication

- **Multiplication.** Given two n-bit integers a and b, compute $a \times b$.
- **Grade-school multiplication algorithm.** $\Theta(n^2)$ bit operations.

- **Conjecture.** [Kolmogorov 1956]
  Grade-school multiplication algorithm is optimal.
- **Theorem.** [Karatsuba 1960] Conjecture is false.



Integer Multiplication: A First Attempt

- **Divide and conquer:** (multiply two n-bit integers x and y)
  - Divide x and y into low- and high-order bits.
  - Recursively multiply four $n/2$-bit integers: ac, bc, ad, bd
  - Add and shift to obtain result.
- **Example.** $n = 8$, $m = \lceil n/2 \rceil = 4$.

$$x = \underbrace{1000}_{a}\underbrace{1101}_{b} \qquad y = \underbrace{1110}_{c}\underbrace{0001}_{d}$$

$$xy = (2^m a + b)(2^m c + d) = 2^{2m}ac + 2^m(bc + ad) + bd$$

(1) (2) (3) (4)

- **Time complexity.** $\Theta(n^2)$ ⟵ $T(n) = 4T(n/2) + O(n)$

Integer Multiplication: Karatsuba's Trick

- **Divide and conquer:** (multiply two n-bit integers x and y)
  - Divide x and y into low- and high-order bits.
  - Recursively multiply three $n/2$-bit integers: ac, ~~bc, ad,~~ $(a - b)(c - d)$, bd
  - Add and shift to obtain result.
- **Example.** $n = 8$, $m = \lceil n/2 \rceil = 4$.

$$x = \underbrace{1000}_{a}\underbrace{1101}_{b} \qquad y = \underbrace{1110}_{c}\underbrace{0001}_{d}$$

bc + ad = ac + bd - (a - b)(c - d)

$$xy = (2^m a + b)(2^m c + d) = 2^{2m}ac + 2^m(ac + bd - (a - b)(c - d)) + bd$$

(1) (1) (3) (2) (3)

- **Time complexity.** $\Theta(n^{\log_2 3}) = \Theta(n^{1.585})$ ← T(n) = 3T(n/2) + O(n)

Karatsuba's Algorithm

- **Karatsuba's algorithm:**

```
Karatsuba-Multiply(x, y, n) {
   if (n = 1) return x * y
   else
      m ← ⌈ n / 2 ⌉
      a ← ⌊ x / 2^m ⌋; b = x mod 2^m
      c ← ⌊ y / 2^m ⌋; d = y mod 2^m
      e ← Karatsuba-Multiply(a, c, m)
      f ← Karatsuba-Multiply(b, d, m)
      g ← Karatsuba-Multiply(|a − b|, |c − d|, m)
      Flip sign of g if needed

      return 2^(2m) e + 2^m (e + f − g) + f
}
```

O(n)

3T(n / 2)

O(n)

- **Practice.** Use *32/64*-bit word. Faster than grade-school for ≥~*320* bits.

Integer Multiplication: Asymptotic Complexity

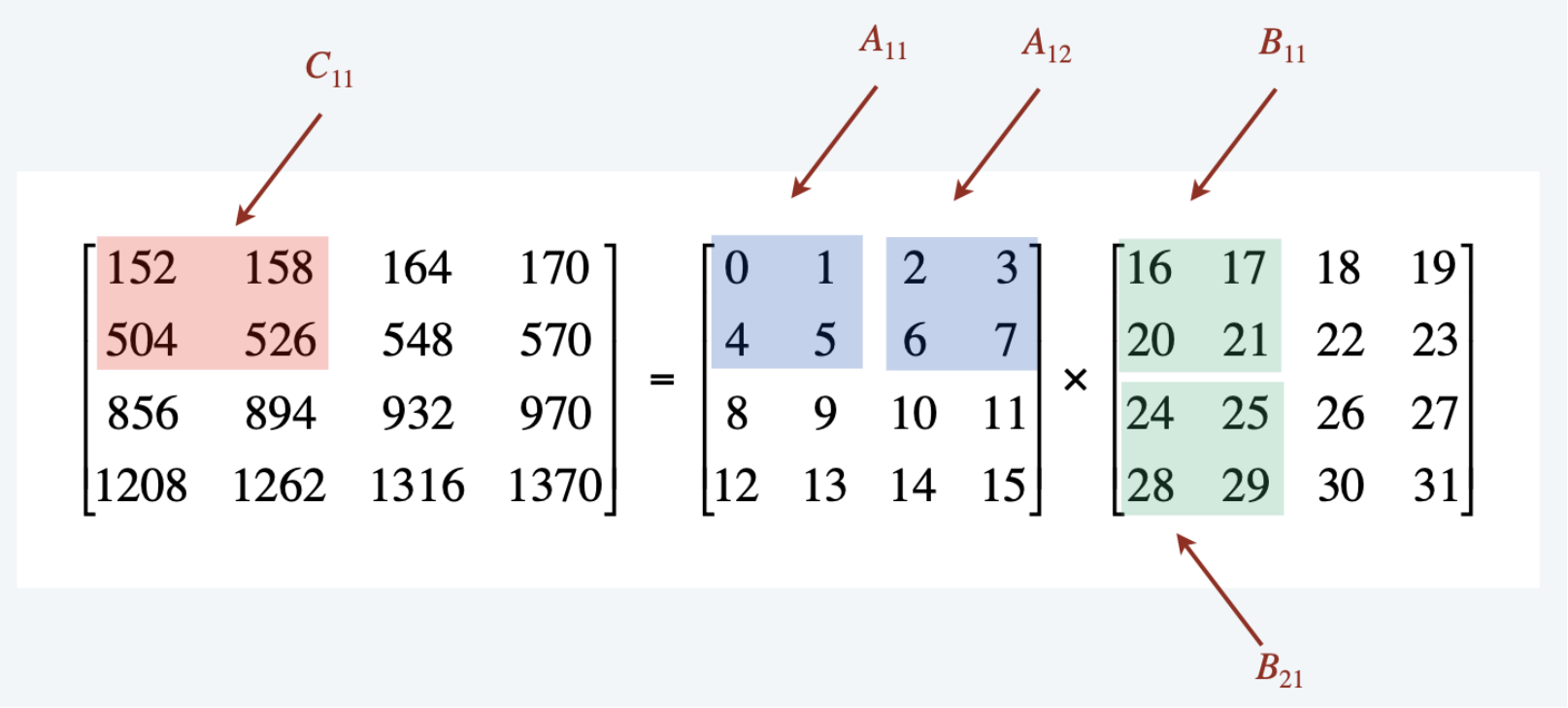
year	algorithm	bit operations
12xx	grade school	$O(n^2)$
1962	Karatsuba–Ofman	$O(n^{1.585})$
1963	Toom-3, Toom-4	$O(n^{1.465})$, $O(n^{1.404})$
1966	Toom-Cook	$O(n^{1+\varepsilon})$
1971	Schönhage–Strassen	$O(n \log n \cdot \log \log n)$
2007	Fürer	$n \log n \, 2^{O(\log^* n)}$
2019	Harvey–van der Hoeven	$O(n \log n)$
		$O(n)$

- **Remark.** GNU Multiple Precision Arithmetic Library (GMP) uses one of first five algorithms depending on n.

used in Maple, Mathematica, gcc, cryptography…

Matrix Multiplication

- **Matrix multiplication.** Given n-by-n matrices A and B, compute $C = AB$.
- **Grade-school matrix multiplication.** $\Theta(n^3)$ arithmetic operations.
- **Block matrix multiplication:**



$$C_{11} = A_{11} \times B_{11} + A_{12} \times B_{21}$$

Matrix Multiplication: A First Attempt

- **Divide and conquer:** (multiply two *n*-by-*n* matrices *A* and *B*)
  - Divide: partition *A* and *B* into *n*/2-by-*n*/2 blocks.
  - Conquer: multiply *8* pairs of *n*/2-by-*n*/2 matrices, recursively.
  - Combine: add appropriate products using *4* matrix additions.

n-by-n matrices

$$C = A \times B$$

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

½n-by-½n matrices

8 matrix multiplications (of ½n-by-½n matrices)

$$\begin{aligned} C_{11} &= (A_{11} \times B_{11}) + (A_{12} \times B_{21}) \\ C_{12} &= (A_{11} \times B_{12}) + (A_{12} \times B_{22}) \\ C_{21} &= (A_{21} \times B_{11}) + (A_{22} \times B_{21}) \\ C_{22} &= (A_{21} \times B_{12}) + (A_{22} \times B_{22}) \end{aligned}$$

4 matrix additions (of ½n-by-½n matrices)

$T(n) = 8T(n/2) + O(n^2)$

$T(n) = O(n^3)$

Matrix Multiplication: Strassen's Trick

- **Divide and conquer:** (multiply two *n*-by-*n* matrices *A* and *B*)
  - Divide: partition *A* and *B* into *n*/2-by-*n*/2 blocks.
  - Conquer: multiply *7* pairs of *n*/2-by-*n*/2 matrices, recursively.
  - Combine: *11* matrix additions and *7* matrix subtractions.

scalars

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

$$C_{11} = P_5 + P_4 - P_2 + P_6$$
$$C_{12} = P_1 + P_2$$
$$C_{21} = P_3 + P_4$$
$$C_{22} = P_1 + P_5 - P_3 - P_7$$

$$P_1 \leftarrow A_{11} \times (B_{12} - B_{22})$$
$$P_2 \leftarrow (A_{11} + A_{12}) \times B_{22}$$
$$P_3 \leftarrow (A_{21} + A_{22}) \times B_{11}$$
$$P_4 \leftarrow A_{22} \times (B_{21} - B_{11})$$
$$P_5 \leftarrow (A_{11} + A_{22}) \times (B_{11} + B_{22})$$
$$P_6 \leftarrow (A_{12} - A_{22}) \times (B_{21} + B_{22})$$
$$P_7 \leftarrow (A_{11} - A_{21}) \times (B_{11} + B_{12})$$

$T(n) = 7T(n/2) + O(n^2)$

$T(n) = O(n^{\log_2 7}) = O(n^{2.81})$

Strassen's Algorithm in Practice

- **Implementation issues:**
  - Sparsity.
  - Caching.
  - *n* not a power of *2*.
  - Numerical stability.
  - Non-square matrices.
  - Storage for intermediate submatrices.
  - Crossover to classical algorithm when *n* is "small".
  - Parallelism for multi-core and many-core architectures.
- **Nevertheless, still useful in practice.**
  - Apple reports *8x* speedup when *n* ≈ *2,048*.

Matrix Multiplication: Asymptotic Complexity

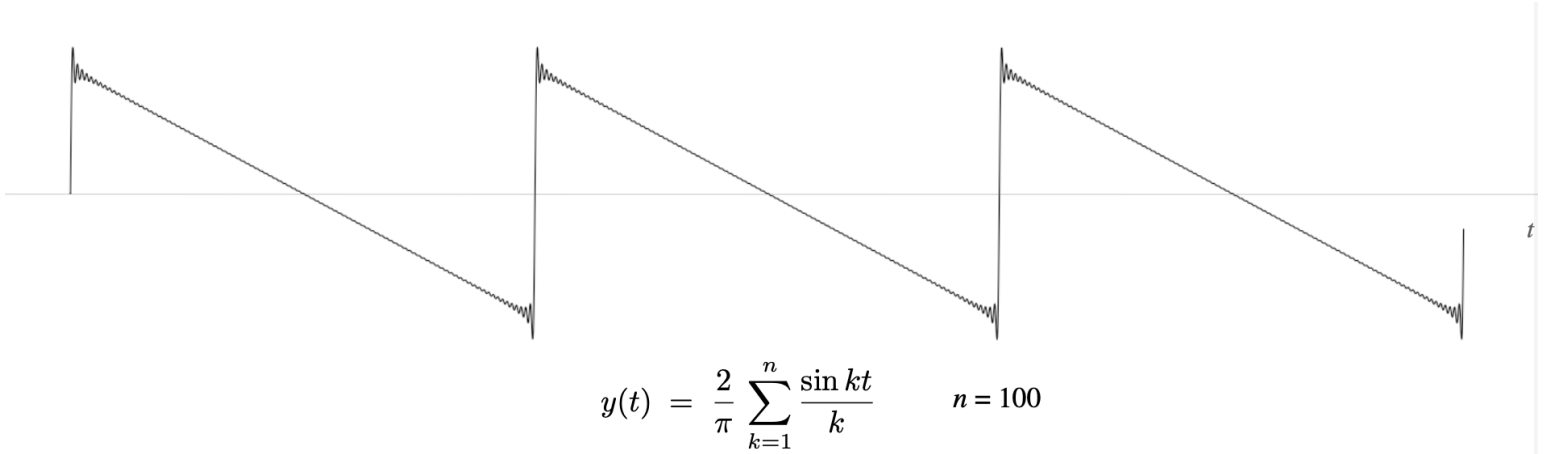
Year	Bound on omega	Authors
1969	2.8074	Strassen[1]
1978	2.796	Pan[10]
1979	2.780	Bini, Capovani [it], Romani[11]
1981	2.522	Schönhage[12]
1981	2.517	Romani[13]
1981	2.496	Coppersmith, Winograd[14]
1986	2.479	Strassen[15]
1990	2.3755	Coppersmith, Winograd[16]
2010	2.3737	Stothers[17]
2012	2.3729	Williams[18][19]
2014	2.3728639	Le Gall[20]
2020	2.3728596	Alman, Williams[21][22]
2022	2.371866	Duan, Wu, Zhou[23]
2024	2.371552	Williams, Xu, Xu, and Zhou[2]

Reference: https://en.wikipedia.org/wiki/Computational_complexity_of_matrix_multiplication

5. Convolution and FFT

Fourier Analysis and Euler's Identity

- **Fourier theorem.** [Fourier, Dirichlet, Riemann] Any (sufficiently smooth) periodic function can be expressed as the sum of a series of sinusoids.



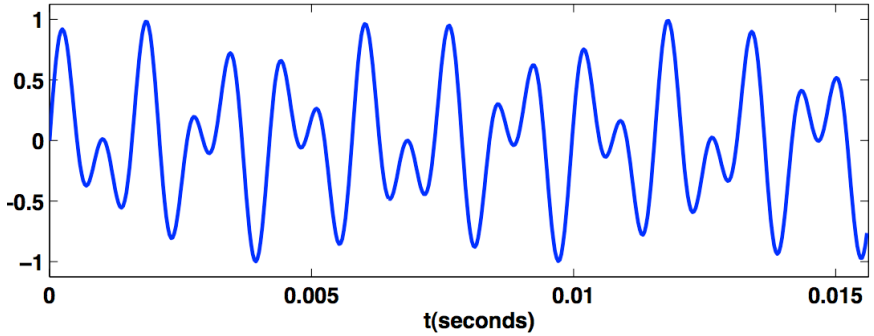
- **Euler's identity.** $e^{ix} = \cos x + i \sin x$
  - Sum of sine and cosines = sum of complex exponentials

Example: Touch Tone

Touch Tone 是一种电话拨号技术，使用双音多频（DTMF）信号来表示按键。DTMF 广泛应用于传统电话系统中，当你按下电话上的按键时，会生成一个由两个特定频率的正弦波叠加而成的信号

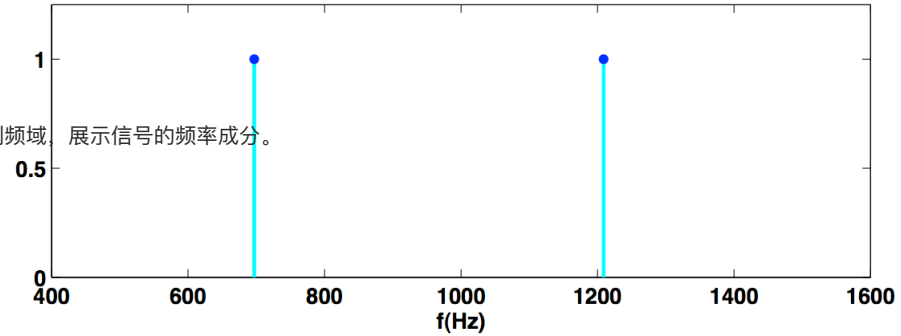
- **Signal for button 1.** $y(t) = \frac{1}{2}\sin(2\pi \cdot 697t) + \frac{1}{2}\sin(2\pi \cdot 1209t)$

- **Time domain:**



- **Frequency domain:**

频域表示是通过傅里叶变换（Fourier Transform）将时域信号 y(t) 转换到频域，展示信号的频率成分。

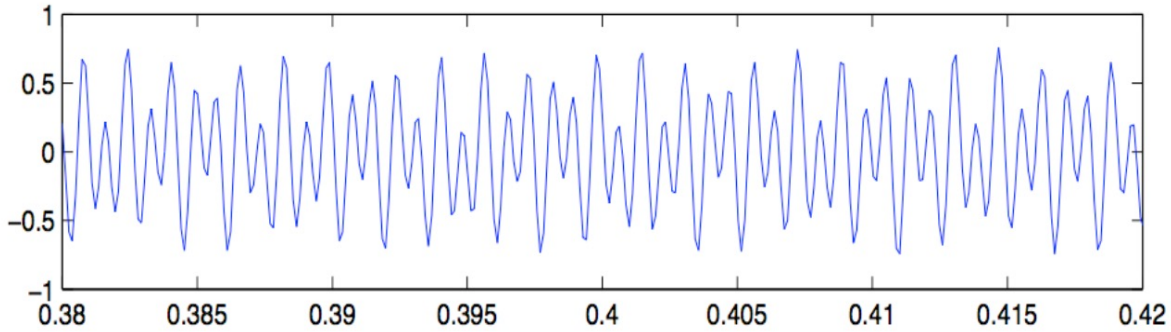


Reference: Cleve Moler, Numerical Computing with MATLAB

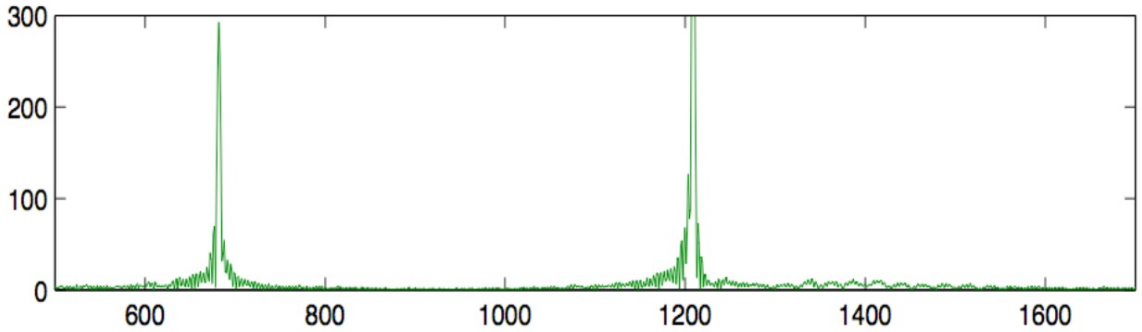
Example: Touch Tone

- **Signal for button 1.** $y(t) = \frac{1}{2}\sin(2\pi \cdot 697t) + \frac{1}{2}\sin(2\pi \cdot 1209t)$

- **Signal:**
  - sample rate: *~8kHz*



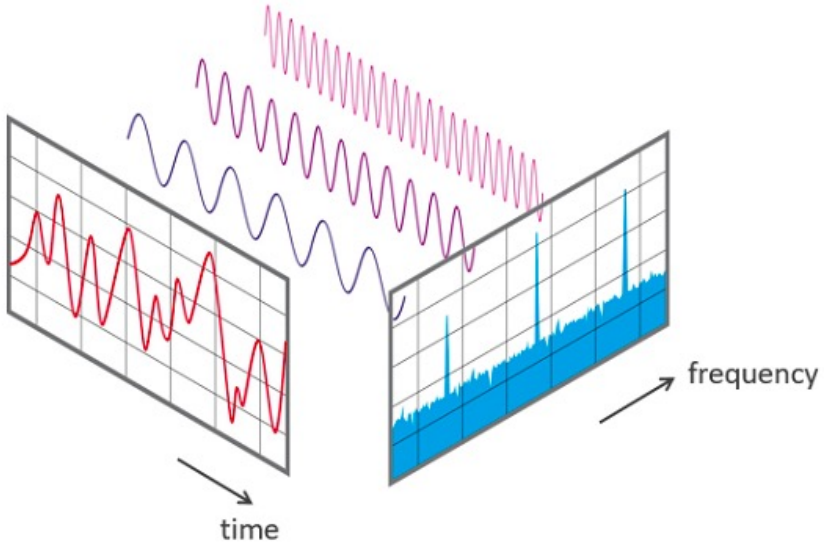
- **Magnitude of discrete Fourier transform:**



Reference: Cleve Moler, Numerical Computing with MATLAB

Fast Fourier Transform (FFT)

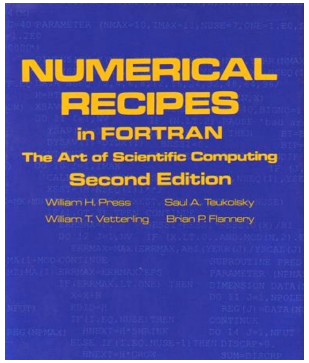
- **FFT.** Fast way to convert between time domain and frequency domain.



we take this viewpoint

- **Alternative viewpoint.** Fast way to multiply and evaluate polynomials.

FFT 是快速进行多项式乘法和求值的工具



" If you speed up any nontrivial algorithm by a factor of a million or so the world will beat a path towards finding useful applications for it. "

Fast Fourier Transform: Applications

- **Applications:**
  - Optics, acoustics, quantum physics, telecommunications, radar, control systems, signal processing, speech recognition, data compression, image processing, seismology, mass spectrometry, ...
  - Digital media. [DVD, JPEG, MP3, H.264]
  - Medical diagnostics. [MRI, CT, PET scans, ultrasound]
  - Numerical solutions to Poisson's equation.
  - Integer and polynomial multiplication.
  - Shor's quantum factoring algorithm.
  - ....

> The FFT is one of the truly great computational developments of this [20th] century. It has changed the face of science and engineering so much that it is not an exaggeration to say that life as we know it would be very different without the FFT. --- Charles van Loan

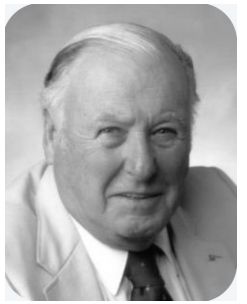
Fast Fourier Transform: Brief History

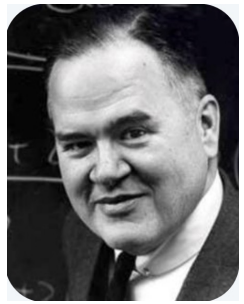
- [Gauss 1805, 1866] Analyzed periodic motion of asteroid Ceres.
- [Runge-König 1924] Laid theoretical groundwork.
- [Danielson-Lanczos 1942] Efficient algorithm, *x*-ray crystallography.
- [Cooley-Tukey 1965] Detect nuclear tests in Soviet Union and track submarines. Rediscovered and popularized FFT.

An Algorithm for the Machine Calculation of Complex Fourier Series

By James W. Cooley and John W. Tukey

An efficient method for the calculation of the interactions of a 2^m factorial experiment was introduced by Yates and is widely known by his name. The generalization to 3^m was given by Box et al. [1]. Good [2] generalized these methods and gave elegant algorithms for which one class of applications is the calculation of Fourier series. In their full generality, Good's methods are applicable to certain problems in which one must multiply an N-vector by an $N \times N$ matrix which can be factored into m sparse matrices, where m is proportional to $\log N$. This results in a procedure requiring a number of operations proportional to $N \log N$ rather than N^2.





- **Note.** Importance not fully realized until emergence of digital computers.

Polynomials: Coefficient Representation

- **Univariate polynomial.** [coefficient representation]

$$A(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1} \qquad B(x) = b_0 + b_1 x + b_2 x^2 + \cdots + b_{n-1} x^{n-1}$$

- **Addition.** *O(n)* arithmetic operations.

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

- **Evaluation.** *O(n)* using Horner's method.

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

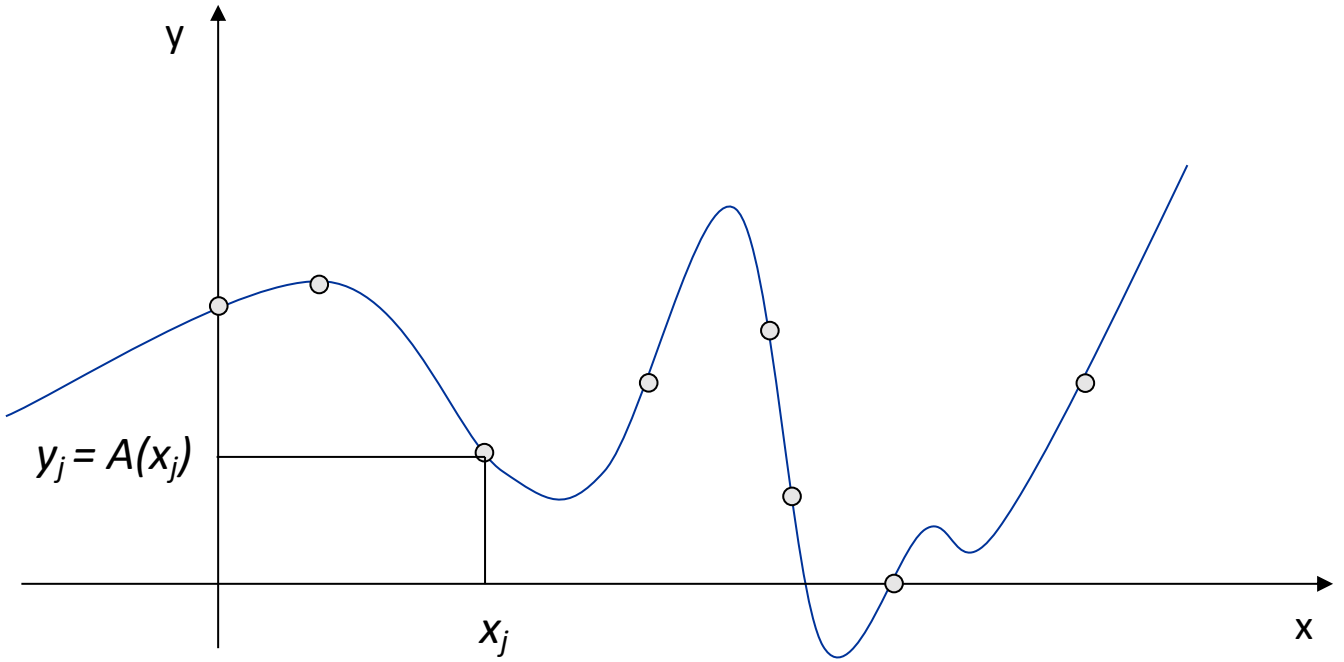
- **Multiplication** (linear convolution 线性卷积): *O(n²)* using brute force.

$$A(x) \times B(x) = \sum_{i=0}^{2n-2} c_i x^i, \text{ where } c_i = \sum_{j=0}^{i} a_j b_{i-j}$$

Polynomials: Point-Value Representation

- **Fundamental theorem of algebra.** [Gauss, PhD thesis] A degree n polynomial with complex coefficients has n complex roots.
- **Corollary.** A degree $n - 1$ polynomial $A(x)$ is uniquely specified by its evaluation at n distinct values of x.

次数为n–1的多项式，在n个不同的x处知道其函数值，那么这个多项式是确定的



Polynomials: Point-Value Representation

- **Polynomial.** [point-value representation] 就是这n个点能唯一表示多项式

$$A(x): (x_0, y_0), \ldots, (x_{n-1}, y_{n-1}) \qquad B(x): (x_0, z_0), \ldots, (x_{n-1}, z_{n-1})$$

- **Addition.** *O(n)* arithmetic operations.

$$A(x) + B(x): (x_0, y_0 + z_0), \ldots, (x_{n-1}, y_{n-1} + z_{n-1})$$

- **Multiplication.** *O(n)*, but represent *A(x)* and *B(x)* using *2n* points. 实际上可以2n−1个点，但多一个有冗余度

$$A(x) \times B(x): (x_0, y_0 \times z_0), \ldots, (x_{2n-1}, y_{2n-1} \times z_{2n-1})$$

- **Evaluation.** *O(n²)* using Lagrange's method. 直接带入x计算乘出来，不需要卷积计算

求值

公式作用：从一组点中构造出一个n−1次的多项式，使得 A(xk)=yk

这是一个平滑的多项式，并且有能够在n个点值确定；说明这个就是原来的多项式

$$A(x) = \sum_{k=0}^{n-1} y_k \frac{\prod_{j \neq k} (x - x_j)}{\prod_{j \neq k} (x_k - x_j)}$$

← not used in FFT

Converting between Two Representations

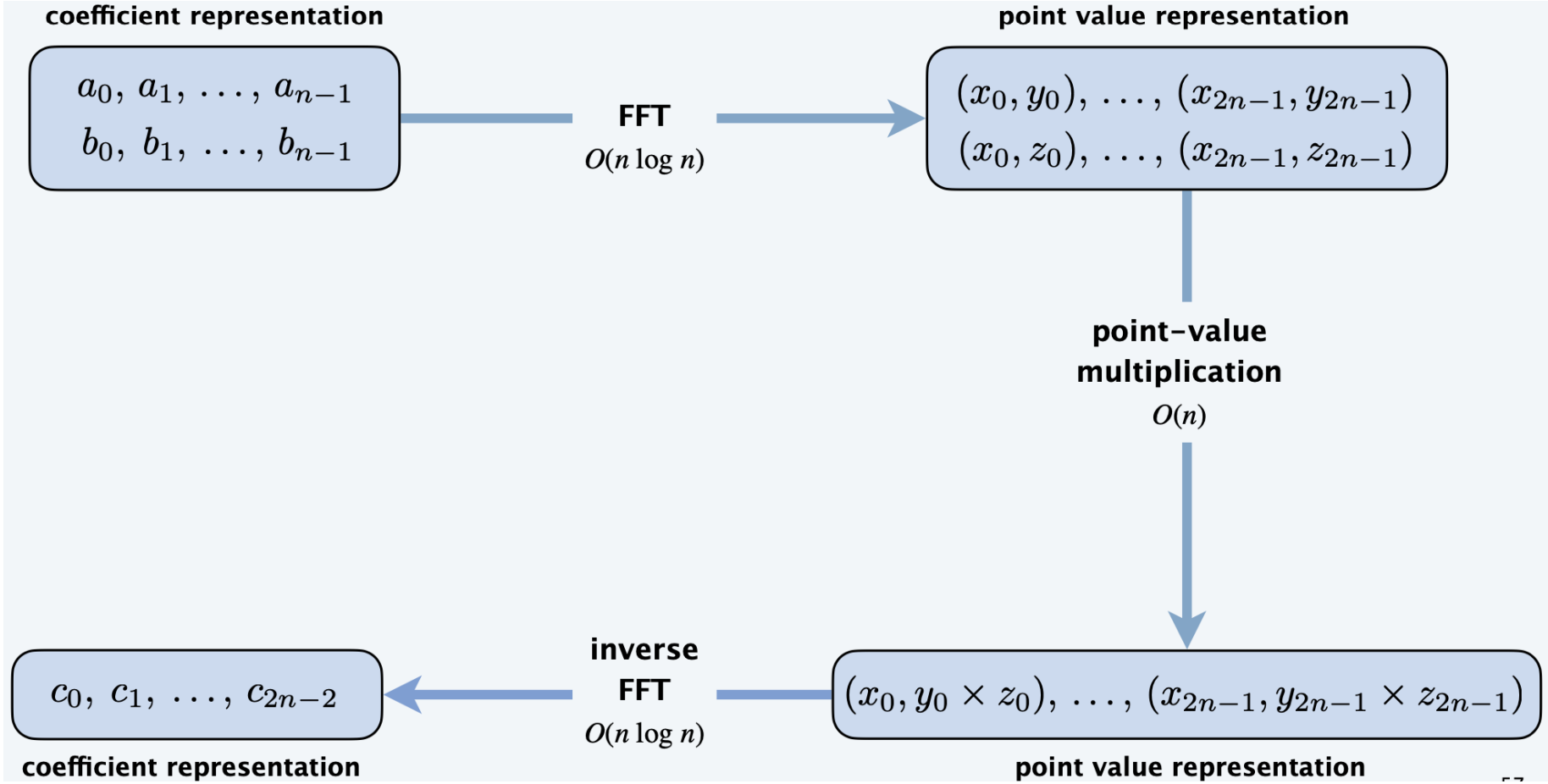
- **Tradeoff.** Either fast evaluation or fast multiplication. We want both!

Representation	Multiply	Evaluate
coefficient	$O(n^2)$ 卷积	$O(n)$
point-value	$O(n)$	$O(n^2)$

- **Goal.** Efficient conversion between two representations ⇒ all ops fast.

Converting between Two Representations

- **Application.** Polynomial multiplication (coefficient representation).



Converting between Two Representations

- **Coefficient ⇒ point-value.** Given a polynomial $a_0 + a_1 x + \ldots + a_{n-1} x^{n-1}$, evaluate it at n distinct points $x_0, \ldots, x_{n-1}$.
  - Running time: $O(n^2)$ via matrix-vector multiplication or n Horner's
- **Point-value ⇒ coefficient.** Given n distinct points $x_0, \ldots, x_{n-1}$ and values $y_0, \ldots, y_{n-1}$, find unique polynomial $a_0 + a_1 x + \ldots + a_{n-1} x^{n-1}$ that has given values at given points.

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{bmatrix}$$

← $O(n^3)$ via Gaussian elimination
$O(n^{2.37})$ via fast matrix multiplication

Vandermonde matrix is invertible iff x_i s are distinct

Coefficient to Point-Value: Intuition

- **Coefficient ⇒ point-value.** Given a polynomial $a_0 + a_1 x + \ldots + a_{n-1} x^{n-1}$, evaluate it at n distinct points $x_0, \ldots, x_{n-1}$. ← we get to choose these points!
- **Divide.** [Cooley-Tukey] Break up polynomial into even and odd degrees.
  - $A(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + a_4 x^4 + a_5 x^5 + a_6 x^6 + a_7 x^7$
  - $A_{even}(x) = a_0 + a_2 x + a_4 x^2 + a_6 x^3$ $\quad A_{odd}(x) = a_1 + a_3 x + a_5 x^2 + a_7 x^3$
  - $A(x) = A_{even}(x^2) + x\, A_{odd}(x^2)$
  - $A(-x) = A_{even}(x^2) - x\, A_{odd}(x^2)$

can also divide into low and high degrees [Sande-Tukey]

- **Intuition.** Choose two points to be ± 1.
  - $A(1) = A_{even}(1) + 1\, A_{odd}(1)$
  - $A(-1) = A_{even}(1) - 1\, A_{odd}(1)$

can evaluate polynomial of degree $n - 1$ at 2 points by evaluating 2 polynomials of degree $\frac{1}{2} n - 1$ at 1 point

Coefficient to Point-Value: Intuition

- **Coefficient ⇒ point-value.** Given a polynomial $a_0 + a_1 x + \ldots + a_{n-1} x^{n-1}$, evaluate it at n distinct points $x_0, \ldots, x_{n-1}$. ← we get to choose these points!
- **Divide.** [Cooley-Tukey] Break up polynomial into even and odd degrees.
  - $A(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + a_4 x^4 + a_5 x^5 + a_6 x^6 + a_7 x^7$
  - $A_{even}(x) = a_0 + a_2 x + a_4 x^2 + a_6 x^3$ $\quad A_{odd}(x) = a_1 + a_3 x + a_5 x^2 + a_7 x^3$
  - $A(x) = A_{even}(x^2) + x\, A_{odd}(x^2)$
  - $A(-x) = A_{even}(x^2) - x\, A_{odd}(x^2)$

can also divide into low and high degrees [Sande-Tukey]

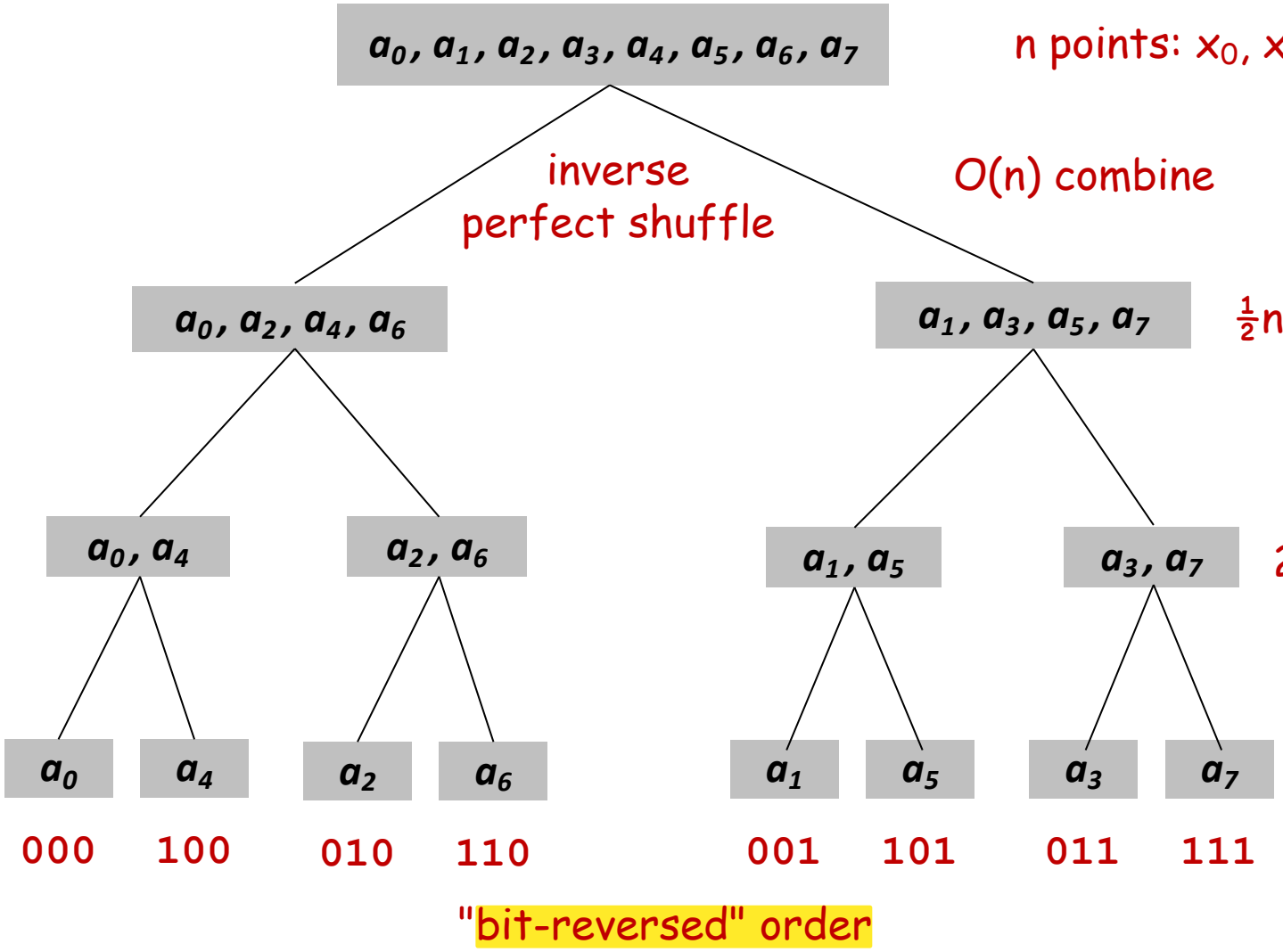
- **Intuition.** Choose four complex points to be $\pm 1, \pm i$.
  - $A(1) = A_{even}(1) + 1\, A_{odd}(1)$
  - $A(-1) = A_{even}(1) - 1\, A_{odd}(1)$
  - $A(i) = A_{even}(-1) + i\, A_{odd}(-1)$
  - $A(-i) = A_{even}(-1) - i\, A_{odd}(-1)$

can evaluate polynomial of degree $n-1$ at 4 points by evaluating 2 polynomials of degree ½ $n-1$ at 2 points

can evaluate polynomial of degree $n-1$ at n points by evaluating 2 polynomials of degree ½ $n-1$ at ***n/2*** points

一个n次多项式，可以通过在4个点上评估两个n/2次多项式来计算；递归使用，时间复杂度是O（nlogn）

Coefficient to Point-Value: Recursion Tree



n points: $x_0, x_1, \ldots, x_{n-1}$

$x_k^2 = x_{k+\frac{1}{2}n}^2$

$\frac{1}{2}n$ points: $x_0^2, x_1^2, \ldots, x_{\frac{1}{2}n-1}^2$

2 points: $x_0^{\frac{1}{2}n}, x_1^{\frac{1}{2}n}$

$(x_0^{\frac{1}{2}n})^2 = (x_1^{\frac{1}{2}n})^2$

1 point: $x_0^n = ?$

$x_k^n = ?$

$x_0^n = \ldots = x_{n-1}^n = 1$

Discrete Fourier Transform (DFT)

- **Coefficient ⇒ point-value.** Given a polynomial $a_0 + a_1 x + \ldots + a_{n-1} x^{n-1}$, evaluate it at n distinct points $x_0, \ldots, x_{n-1}$. ← we get to choose these points!
- **Key idea:** choose $x_k = \omega^k$ where $\omega = e^{2\pi i / n}$ is principal n^{th} root of unity.

$y_k = A(\omega^k)$ →

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

↑ DFT

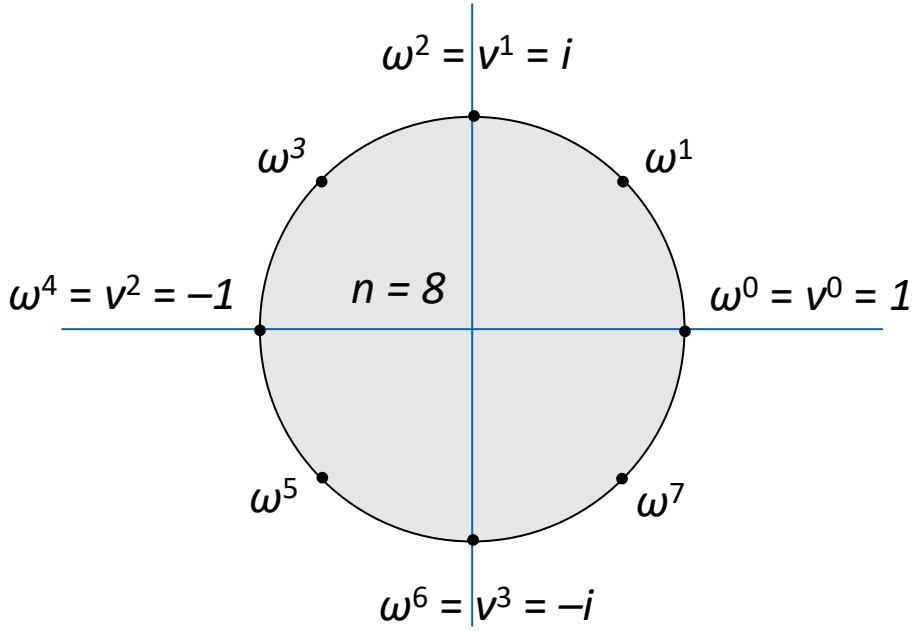
↑ Fourier matrix F_n

Roots of Unity

- **Def.** An n^{th} root of unity is a complex number x such that $x^n = 1$.
- **Fact.** The n^{th} roots of unity are: $\omega^0, \omega^1, \ldots, \omega^{n-1}$ where $\omega = e^{2\pi i / n}$.
- **Pf.** $(\omega^k)^n = (e^{2\pi i k / n})^n = (e^{\pi i})^{2k} = (-1)^{2k} = 1$

common alternative: $\omega = e^{-2\pi i / n}$

- **Fact.** The $\frac{1}{2}n^{th}$ roots of unity are: $\nu^0, \nu^1, \ldots, \nu^{n/2-1}$ where $\nu = \omega^2 = e^{4\pi i / n}$.



Fast Fourier Transform (FFT)

- **Goal.** Evaluate a degree $n - 1$ polynomial $A(x) = a_0 + \ldots + a_{n-1}\, x^{n-1}$ at its n^{th} roots of unity: $\omega^0, \omega^1, \ldots, \omega^{n-1}$.
- **Divide.** Break up polynomial into even and odd powers.
  - $A_{even}(x) = a_0 + a_2 x + a_4 x^2 + \ldots + a_{n-2}\, x^{n/2-1}$
  - $A_{odd}(x) = a_1 + a_3 x + a_5 x^2 + \ldots + a_{n-1}\, x^{n/2-1}$
  - $A(x) = A_{even}(x^2) + x\, A_{odd}(x^2)$
  - $A(-x) = A_{even}(x^2) - x\, A_{odd}(x^2)$
- **Conquer.** Evaluate $A_{even}(x)$, $A_{odd}(x)$ at $\frac{1}{2}n^{\text{th}}$ roots of unity: $v^0, v^1, \ldots, v^{n/2-1}$
- **Combine.** ($v = \omega^2$ and $\omega^{n/2} = -1$)
  - $A(\omega^k) = A_{even}(v^k) + \omega^k\, A_{odd}(v^k), \quad 0 \le k < n/2$
  - $A(\omega^{k+n/2}) = A_{even}(v^k) - \omega^k\, A_{odd}(v^k), \quad 0 \le k < n/2$

Fast Fourier Transform (FFT): Algorithm

- **FFT algorithm:**

```
FFT(n, a_0, a_1, …, a_{n-1}) {
    if (n = 1) return a_0
    (e_0, e_1, …, e_{n/2-1}) = FFT(n/2, a_0, a_2, a_4, …, a_{n-2})
    (d_0, d_1, …, d_{n/2-1}) = FFT(n/2, a_1, a_3, a_5, …, a_{n-1})

    for k = 0 to n/2 - 1 {
        ω^k = e^{2πik/n}
        y_k     = e_k + ω^k d_k
        y_{k+n/2} = e_k - ω^k d_k
    }

    return (y_0, y_1, …, y_{n-1})
}
```

$2T(n / 2)$

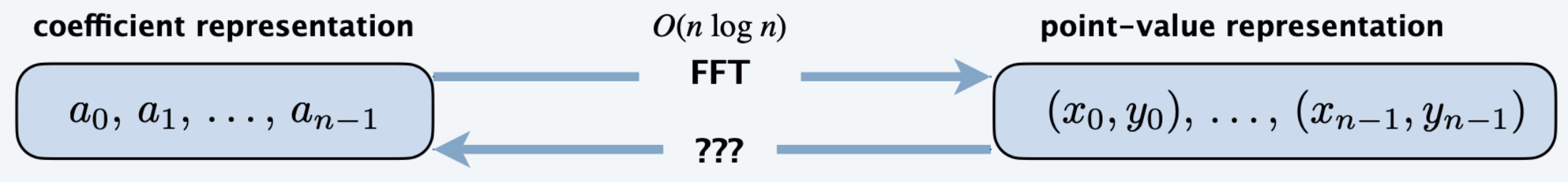
$O(n)$

FFT Summary

- **Theorem.** The FFT algorithm evaluates a degree $n - 1$ polynomial at each of the n^{th} roots of unity in $O(n \log n)$ operations.

assume n is a power of 2

- **Pf.** $T(n) = 2T(n/2) + O(n) \Rightarrow T(n) = O(n \log n)$
- **Q.** What about space complexity?
- **A.** $O(n)$



Point-Value to Coefficient: Inverse DFT

- **Point-value ⇒ coefficient.** Given n distinct points $x_0, \ldots, x_{n-1}$ and values $y_0, \ldots, y_{n-1}$, find unique polynomial $a_0 + a_1 x + \ldots + a_{n-1} x^{n-1}$ that has given values at given points.

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

inverse DFT

Fourier matrix inverse F_n^{-1}

Inverse Discrete Fourier Transform

- **Claim.** Inverse of Fourier matrix F_n is given by following formula:

$$G_n = \frac{1}{n} \begin{bmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega^{-1} & \omega^{-2} & \omega^{-3} & \cdots & \omega^{-(n-1)} \\ 1 & \omega^{-2} & \omega^{-4} & \omega^{-6} & \cdots & \omega^{-2(n-1)} \\ 1 & \omega^{-3} & \omega^{-6} & \omega^{-9} & \cdots & \omega^{-3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{-(n-1)} & \omega^{-2(n-1)} & \omega^{-3(n-1)} & \cdots & \omega^{-(n-1)(n-1)} \end{bmatrix}$$

- **Consequence.** To compute the inverse FFT, apply the same algorithm but use $\omega^{-1} = e^{-2\pi i / n}$ as principal n^{th} root of unity (and divide the result by n).

Inverse FFT: Proof of Correctness

- **Claim.** F_n and G_n are inverses.
- **Pf.**

$$(F_n G_n)_{kk'} = \frac{1}{n}\sum_{j=0}^{n-1} \omega^{kj}\omega^{-jk'} = \frac{1}{n}\sum_{j=0}^{n-1} \omega^{(k-k')j} = \begin{cases} 1 & \text{if } k = k' \\ 0 & \text{otherwise} \end{cases}$$

summation lemma (below)

- **Summation lemma.** Let ω be a principal n^{th} root of unity. Then

$$\sum_{j=0}^{n-1} \omega^{kj} = \begin{cases} n & \text{if } k \equiv 0 \pmod{n} \\ 0 & \text{otherwise} \end{cases}$$

- **Pf.** (direct proof)
  - If k is a multiple of n then $\omega^k = 1$, so the series sums to n.
  - Each n^{th} root of unity ω^k is a root of $x^n - 1 = (x - 1)(1 + x + x^2 + \ldots + x^{n-1})$.
  - If $\omega^k \neq 1$, then $1 + \omega^k + \omega^{k(2)} + \ldots + \omega^{k(n-1)} = 0$, so the series sums to 0. ■

Inverse FFT: Algorithm

- **Inverse FFT algorithm:**

```
Inverse-FFT(n, y0, y1, …, yn-1) {
    if (n == 1) return a0
    (e0, e1, …, en/2-1) = Inverse-FFT(n/2, y0, y2, y4, …, yn-2)
    (d0, d1, …, dn/2-1) = Inverse-FFT(n/2, y1, y3, y5, …, yn-1)

    for k = 0 to n/2 - 1 {
        ω^k = e^(-2πik/n)
        ak     = ek + ω^k dk
        ak+n/2 = ek - ω^k dk
    }
    return (a0, a1, …, an-1)
}
Output: Inverse-FFT(n, y0, y1, …, yn-1) / n
```

$2T(n / 2)$

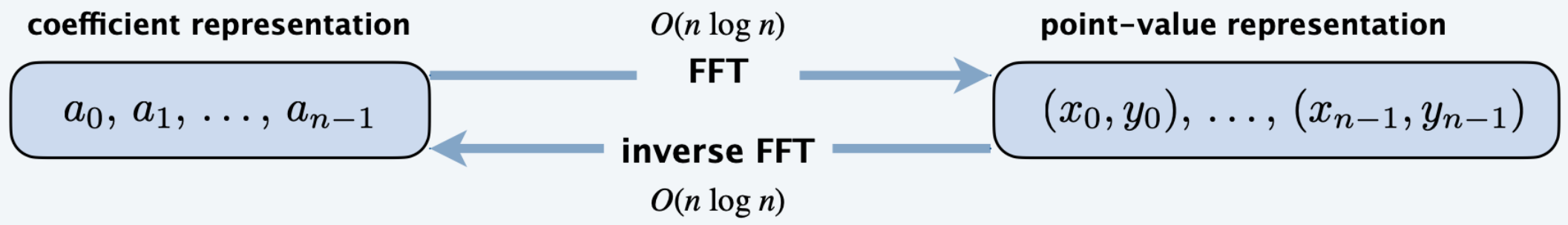
$O(n)$

Inverse FFT Summary

- **Theorem.** The inverse FFT algorithm interpolates a degree $n - 1$ polynomial at each of the n^{th} roots of unity in $O(n \log n)$ operations.

assume n is a power of 2

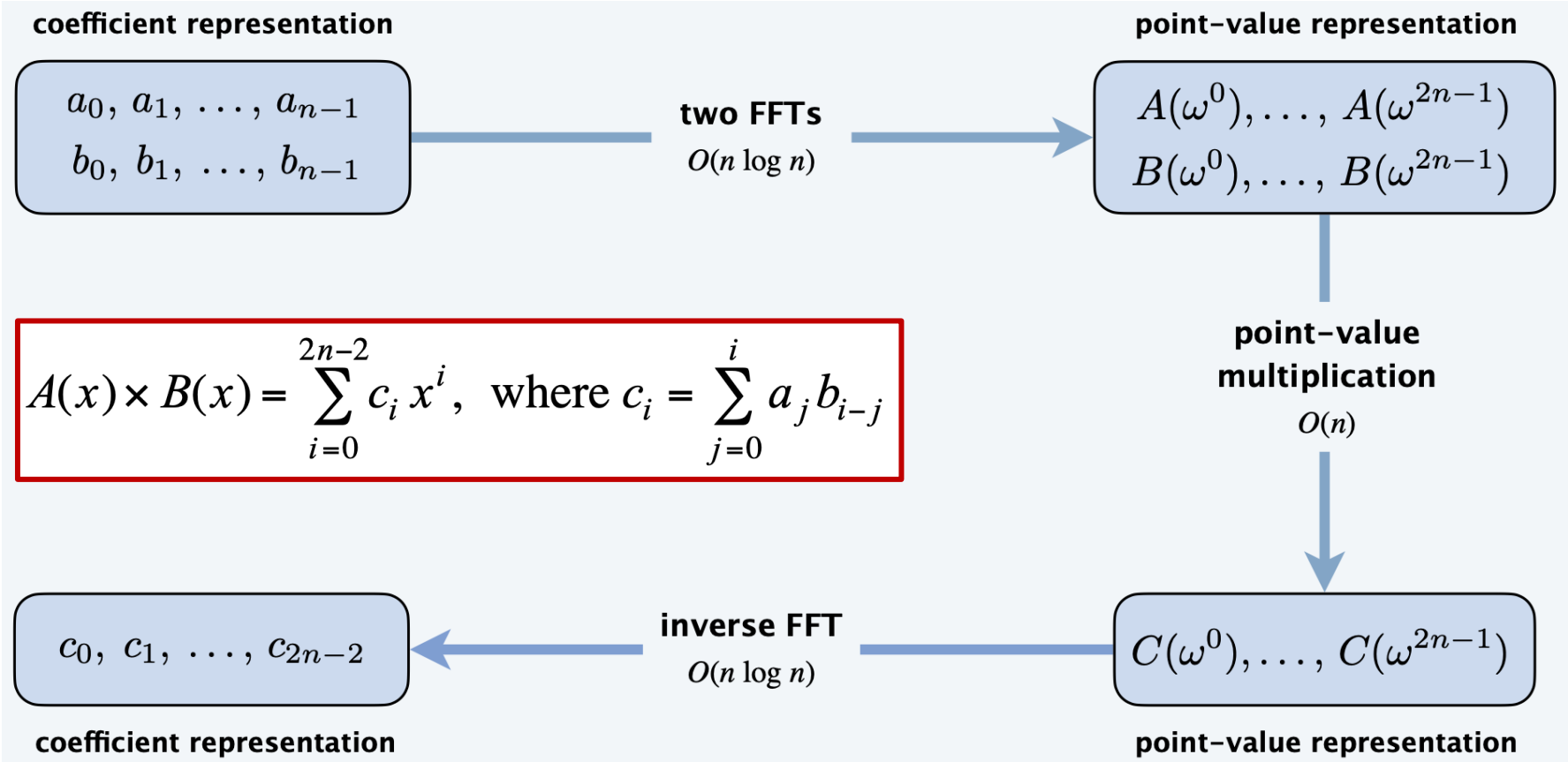
- **FFT + Inverse FFT.** Can convert between coefficient and point-value representations in $O(n \log n)$ arithmetic operations.



Fast Polynomial Multiplication

- **Theorem.** Can multiply two degree $n - 1$ polynomials in $O(n \log n)$ steps.
- **Pf.**

pad 0 items to make n a power of 2



Integer Multiplication Revisited

- **Integer multiplication.** Given two n bit integers $a = a_{n-1} \ldots a_1 a_0$ and $b = b_{n-1} \ldots b_1 b_0$, compute their product $c = ab$.
- **Convolution algorithm:**
  - Form two polynomials $A(x)$, $B(x)$.
  - Note that $a = A(2)$, $b = B(2)$.
  - Compute $C(x) = A(x)\,B(x)$ and evaluate $C(2) = ab$.
  - Running time: $O(n \log n)$ floating-point operations ($O(n (\log n)^3)$ bit operations).

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

$$B(x) = b_0 + b_1 x + b_2 x^2 + \cdots + b_{n-1} x^{n-1}$$

- **Theory.** [Schönhage-Strassen 1971]
  - $O(n \log^2 n)$ bit operations over complex numbers (with $O(\log n)$ bit precision)
  - $O(n \log n \log \log n)$ bit operations over ring of integers (mod a Fermat number)
- **Practice.** GNU Multiple Precision Arithmetic Library (GMP) switches to FFT-based algorithms when n is large (≥ 5~$10K$)

FFT in Practice

- **FFT in the West (FFTW):** [Frigo-Johnson]
  - Optimized C library.
  - Features: DFT, DCT, real, complex, any size, any dimension.
  - Won 1999 Wilkinson Prize for Numerical Software.
  - Portable, competitive with vendor-tuned code.
- **Implementation details:**
  - Core algorithm is an in-place, nonrecursive version of Cooley–Tukey.
  - Instead of executing a fixed algorithm, it evaluates the hardware and uses a special-purpose compiler to generate an optimized algorithm catered to the "shape" of the problem.
  - Runs in $O(n \log n)$ time, even when n is prime.
  - Multidimensional FFTs.
  - Parallelism.

Reference: http://www.fftw.org

Announcement

- **Assignment 3 has been released and the deadline is April 22.**