Divide and Conquer Strategy

- *Divide-and-Conquer Strategy* for Algorithm Design:
  - **Divide**: If input size is too large to deal with in a straightforward manner, then **divide the problem** into two or more disjoint subproblems that are smaller instances of the same problem.
  - **Conquer**: Conquer subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.
  - **Combine**: Combine the solutions to the subproblems into the solution for the original problem i.e. take the solutions to the subproblems and "merge" these solutions into a complete solution for the original problem.

A General Divide-and-Conquer Algorithm

Step 1: If the problem size is small, then solve this problem directly; otherwise, split original problem into two sub-problems with equal sizes.

Step 2: Recursively solve these two sub-problems by applying this algorithm.

Step 3: Merge solutions of these two sub-problems into a solution of the original problem.

Time Complexity of the General Algorithm

- Time Complexity:

$$T(n)=\begin{cases} 2T(n/2)+S(n)+M(n) & , n \geq c \\ b & , n < c \end{cases}$$

where, S(n): Time for splitting,

M(n): Time for merging,

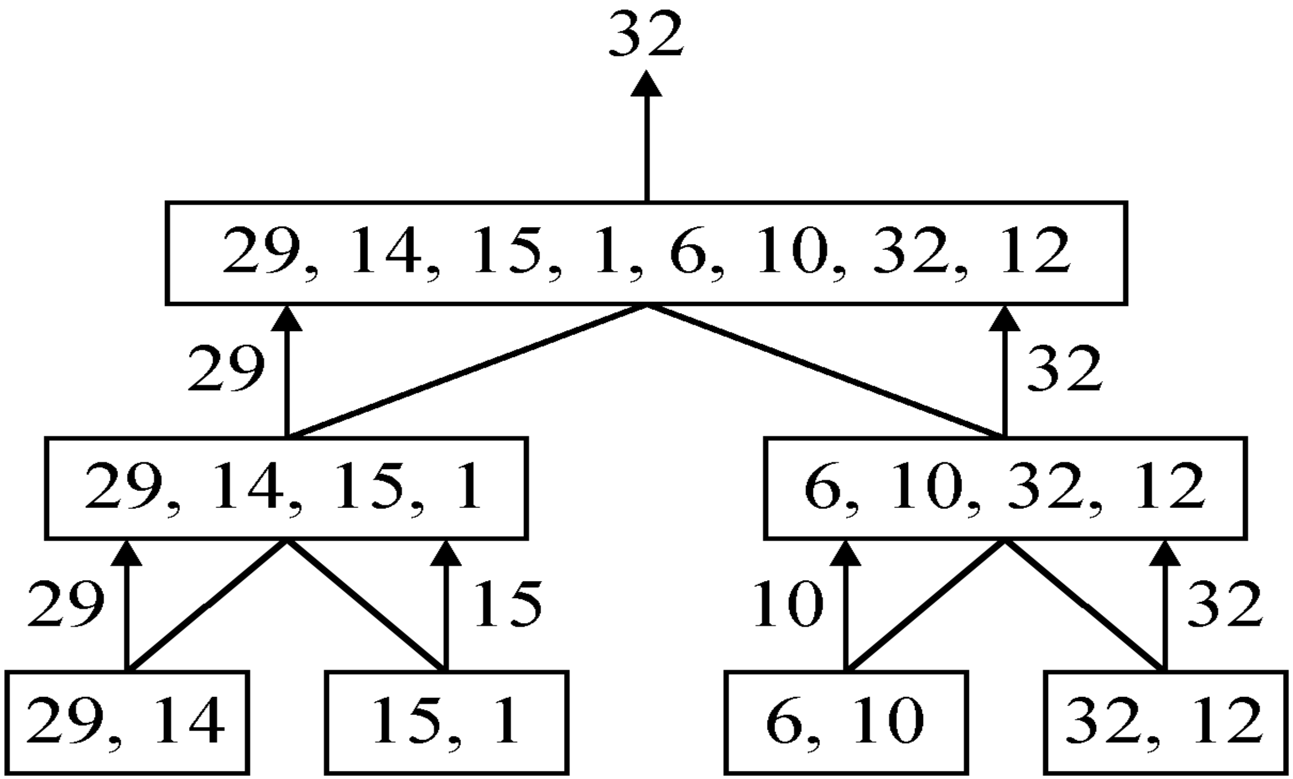
b: a constant,

c: a constant.

- e.g. Binary Search
- e.g. Quick Sort
- e.g. Merge Sort

EXAMPLE: DIVIDE AND CONQUER ALGO.

- Example: Finding the maximum of a set S of n numbers



Time Complexity

- Time Complexity:

$$T(n)=\begin{cases} 2T(n/2)+1 & , n>2 \\ 1 & , n\leq 2 \end{cases}$$

- Calculation of T(n):

Assume $n = 2^k$,

$$\begin{aligned} T(n) &= 2T(n/2)+1 \\ &= 2(2T(n/4)+1)+1 \\ &= 4T(n/4)+2+1 \\ &\;\;\vdots \\ &=2^{k-1}T(2)+2^{k-2}+\ldots+4+2+1 \\ &=2^{k-1}+2^{k-2}+\ldots+4+2+1 \\ &=2^k-1 = n-1 \end{aligned}$$

Recurrence Relations

- Running times of algorithms with **Recursive calls** can be described by using the recurrence relations
- A **Recurrence** is an equation or inequality that describes a function in terms of its value on smaller inputs

$$T(n)=\begin{cases}\text{solving\_trivial\_problem} & \text{if } n=1\\ \text{num\_pieces } T(n/\text{subproblem\_size\_factor})+\text{dividing}+\text{combining} & \text{if } n>1\end{cases}$$

- Example: Merge Sort

$$T(n)=\begin{cases}\Theta(1) & \text{if } n=1\\ 2T(n/2)+\Theta(n) & \text{if } n>1\end{cases}$$

Solving Recurrences

- Iterative (Repeated) Substitution Method
  - Expanding the recurrence by substitution and noticing patterns
- Substitution Method
  - Guessing the solutions
  - Verifying the solution by the mathematical induction
- Recursion-Trees
- Master Theorem Method
  - Templates for different classes of recurrences

Iterative (Repeated) Substitution Method

- Let us find the execution time (running time) of merge sort (let us assume that $n=2^b$, for some b).

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2T(n/2) + n & \text{if } n > 1 \end{cases}$$

$$\begin{aligned}
T(n) &= 2T\left(n/2\right) + n && \text{substitute} \\
&= 2\left(2T\left(n/4\right) + n/2\right) + n && \text{expand} \\
&= 2^2 T(n/4) + 2n && \text{substitute} \\
&= 2^2 (2T(n/8) + n/4) + 2n && \text{expand} \\
&= 2^3 T(n/8) + 3n && \text{observe the pattern} \\
T(n) &= 2^i T(n/2^i) + in \\
&= 2^{\lg n} T(n/n) + n \lg n = n + n \lg n
\end{aligned}$$

Iterative (Repeated) Substitution Method

- The procedure is straightforward:
  - Substitute
  - Expand
  - Substitute
  - Expand
  - …
  - Clearly Observe a Pattern and then Write how our expression looks after the *i*-th substitution
  - Find out what the value of *i* (e.g., lg*n*) should be to get the base case of the recurrence (say *T*(1))
  - Insert the value of *T*(1) and the expression of *i* into our expression

Integer Multiplication

- Algorithm: Multiply two n-bit integers I and J.
  - Divide step: Split I and J into high-order and low-order bits

$$I = I_h 2^{n/2} + I_l$$

$$J = J_h 2^{n/2} + J_l$$

  - We can then define I*J by multiplying the parts and adding:

$$\begin{aligned} I * J &= (I_h 2^{n/2} + I_l) * (J_h 2^{n/2} + J_l) \\ &= I_h J_h 2^n + I_h J_l 2^{n/2} + I_l J_h 2^{n/2} + I_l J_l \end{aligned}$$

  - So, $T(n) = 4T(n/2) + n$, which implies $T(n)$ is $O(n^2)$.
  - But that is no better than the algorithm we've learnt earlier.

Improved Integer Multiplication Algorithm

- Algorithm: Multiply two n-bit integers I and J.
  - Divide step: Split I and J into high-order and low-order bits

$$I = I_h 2^{n/2} + I_l$$

$$J = J_h 2^{n/2} + J_l$$

  - Observe that there is a different way to multiply parts:

$$\begin{aligned} I * J &= I_h J_h 2^n + [(I_h - I_l)(J_l - J_h) + I_h J_h + I_l J_l] 2^{n/2} + I_l J_l \\ &= I_h J_h 2^n + [(I_h J_l - I_l J_l - I_h J_h + I_l J_h) + I_h J_h + I_l J_l] 2^{n/2} + I_l J_l \\ &= I_h J_h 2^n + (I_h J_l + I_l J_h) 2^{n/2} + I_l J_l \end{aligned}$$

  - So, T(n) = 3T(n/2) + n, which implies T(n) is $O(n^{\log_2 3})$, by the Master Theorem.
  - Thus, T(n) is $O(n^{1.585})$.

Matrix Multiplication

- Let A, B and C be n × n matrices

$$C = AB$$

$$C(i, j) = \sum_{1 \le k \le n} A(i, k)B(k, j)$$

- The straightforward method to perform a matrix multiplication requires $O(n^3)$ time.

Divide-and-Conquer Approach

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

$C_{11} = A_{11} B_{11} + A_{12} B_{21}$
$C_{12} = A_{11} B_{12} + A_{12} B_{22}$
$C_{21} = A_{21} B_{11} + A_{22} B_{21}$
$C_{22} = A_{21} B_{12} + A_{22} B_{22}$

- Time Complexity:

(Number of additions : n^2)

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

$$T(n) = \begin{cases} b & , n \le 2 \\ 8T(n/2)+cn^2 & , n > 2 \end{cases}$$

Strassen’s Matrix Multiplication

- $P = (A_{11} + A_{22})(B_{11} + B_{22})$
  $Q = (A_{21} + A_{22})B_{11}$
  $R = A_{11}(B_{12} - B_{22})$
  $S = A_{22}(B_{21} - B_{11})$
  $T = (A_{11} + A_{12})B_{22}$
  $U = (A_{21} - A_{11})(B_{11} + B_{12})$
  $V = (A_{12} - A_{22})(B_{21} + B_{22})$.
- $C_{11} = P + S - T + V$
  $C_{12} = R + T$
  $C_{21} = Q + S$
  $C_{22} = P + R - Q + U$

Time Complexity

- 7 multiplications and 18 additions or subtractions
- Time Complexity: $T(n) = \begin{cases} b & , n \le 2 \\ 7T(n/2)+an^2 & , n > 2 \end{cases}$

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

$= an^2 + 7(a(n/2)^2 + 7T(n/4))$

$= an^2 + (7/4)an^2 + 7^2T(n/4)$

$= \ldots$

$\vdots$

$= an^2(1 + 7/4 + (7/4)^2+\ldots+(7/4)^{k-1}+7^kT(1))$

$\le cn^2(7/4)^{\log2n}+7^{\log2n}$, c is a constant

$= cn^{\log24+\log27-\log24} +n^{\log27}$

$= O(n^{\log27})$

$\cong O(n^{2.81})$

Computational Geometry Problems

- Computational Geometry is the Branch of Computer Science that studies Algorithms for Solving the Geometric Problems.
- In modern engineering and mathematics, computational geometry has applications such as Computer Graphics, Robotics, VLSI Design, Computer-Aided Design (CAD), Molecular Modeling, Metallurgy, Manufacturing, Textile Layout, Forestry, and Statistics.
- The input to computational geometry problem is typically a description of a set of geometric objects, such as a set of points, line segments, or the vertices of a polygon in counterclockwise order.
- The output is often a response to a query about the objects, such as whether any of lines intersect, or perhaps a new geometric object like Convex Hull (smallest enclosing convex polygon) of a set of points.
- Two geometric problems: The Closest Pair and The Convex Hull.

The Closest Pair Problem

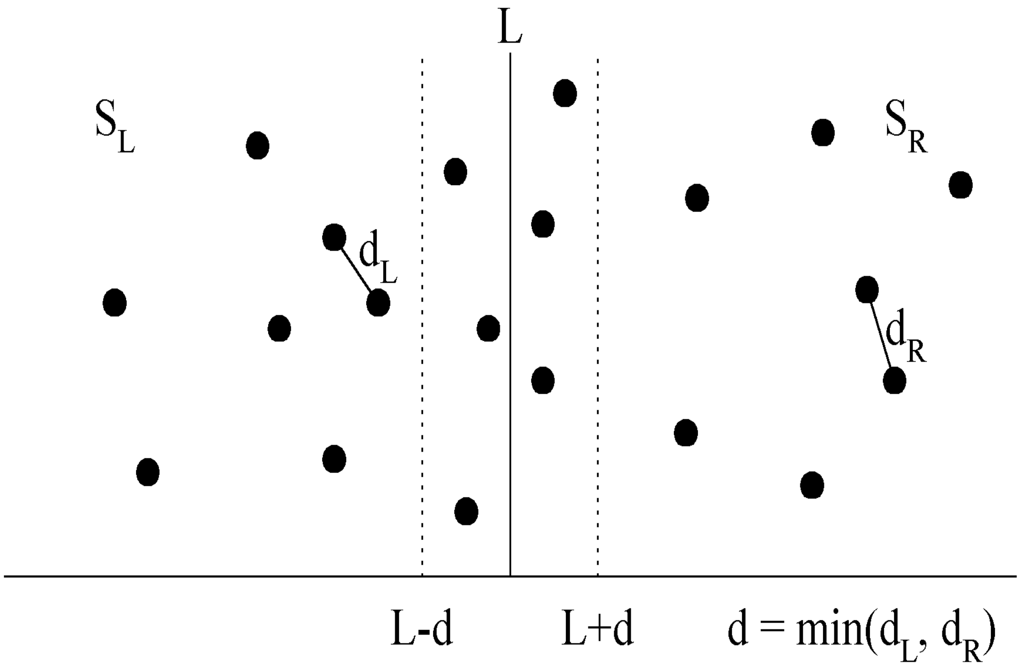
- Given a set S of n points, find a pair of points which are closest together.
- 1-D version

  Solved by sorting

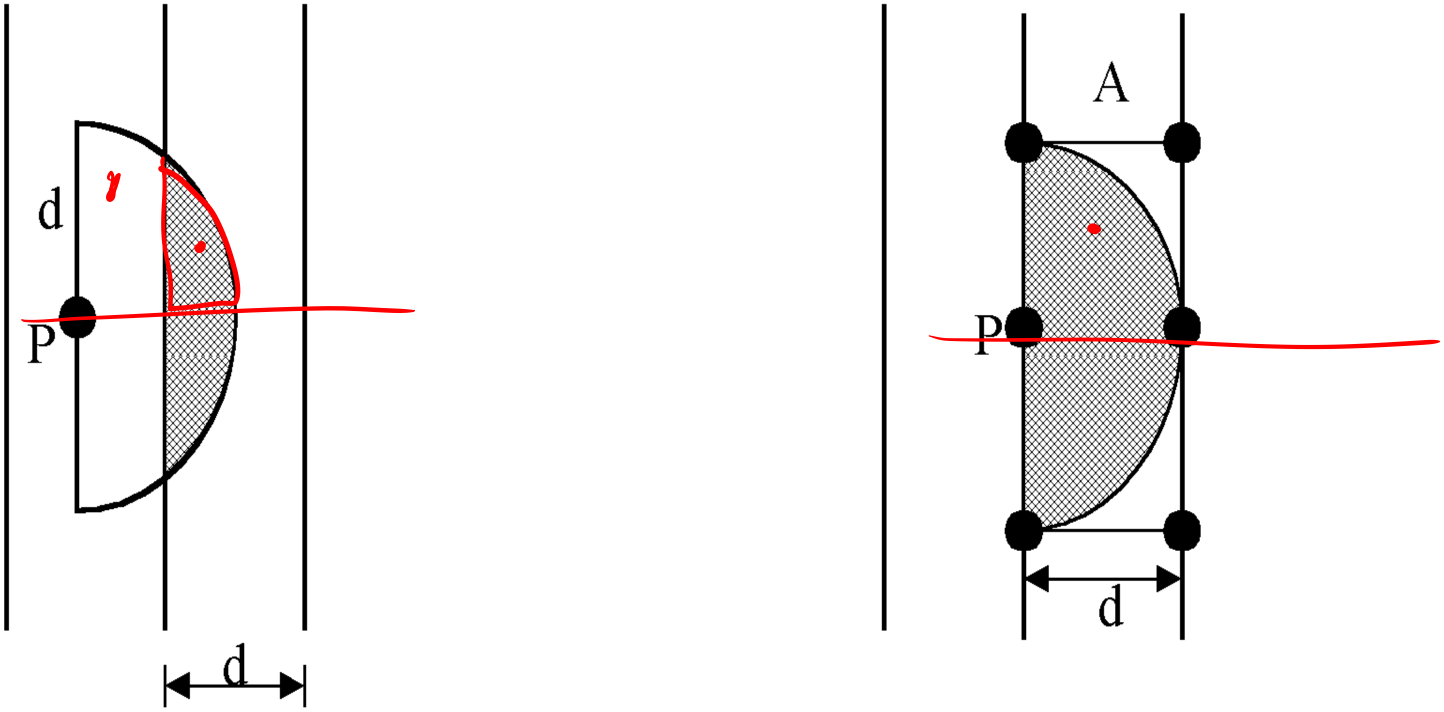
  Time complexity :

  O(n log n)

- 2-D version



- at most 6 points in area A:



The Algorithm:

- Input: A set of n planar points.
- Output: The distance between two closest points.

Step 1: Sort points in S according to y-values and x-values.

Step 2: If S contains only two points, return ∞ as a distance.

Step 3: Find a median line L perpendicular to the X-axis to divide S into two subsets, with equal sizes, S_L and S_R.

Step 4: Recursively apply Steps 2 and 3 to solve the closest pair problems of S_L and S_R. Let $d_L(d_R)$ denote the distance between the closest pair in S_L (S_R). Let $d = \min(d_L, d_R)$.

Step 5: For a point P in the half-slab bounded by L-d and L, let its y-value by denoted as y_P . For each such P, find all points in the half-slab bounded by L and L+d whose y-value fall within y_P +d and y_P -d. If the distance d′ between P and a point in the other half-slab is less than d, let d=d′ . The final value of d is the answer.

- Time complexity: O(n log n)

Step 1: O(n log n)

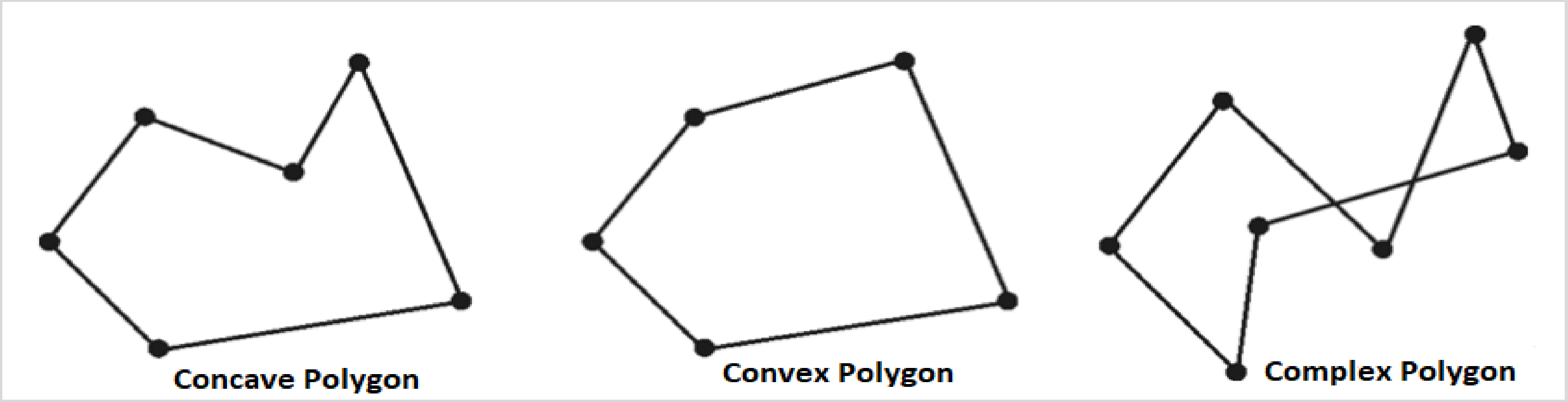
Steps 2~5:

$$T(n)=\begin{cases} 2T(n/2)+O(n)+O(n) & , n > 1 \\ 1 & , n = 1 \end{cases}$$

$\Rightarrow$T(n) = O(n log n)

The Convex Hull Problem

- Convex hull is the smallest region covering given set of planar points. Polygon is called a **Convex Polygon** if the angle between any of its two adjacent edges is always less than 180^0. Otherwise, it is called a **Concave Polygon**. Whereas, the **Complex Polygons** are self-intersecting polygons.



Applications of the Convex Hull

- **Collision Avoidance:** If the convex hull of car avoids collisions with obstructions, so does the car. Because calculating collision-free routes is considerably easier with a convex hull of vehicle, it is frequently used to design paths.
- **Smallest Box:** Convex hull helps to determine the minimum size required to put the object in a box. It is useful to determine the box size for the object. Finding the smallest 3D box enclosing an item is also dependent on the 3D-convex hull.

Applications of the Convex Hull (contd…)

- **Shape Analysis:** Convex hull of object is useful to analyze the shape of object.
- **Pattern Recognition, Image Processing, Mathematics, Statistics, Combinatorial Optimization, Game Theory, Geographic Information Systems, Geometric Modeling, Economics, Phase Diagram Creation, and Static Code Analysis via Abstract Interpretation** are some more practical scenarios where the convex hull plays a key role.
- **Computational-Geometry:** It is also a building component for a variety of other Computational-Geometric Methods, such as the Rotating Calipers Technique for Calculating the Breadth and Diameter of given set of points.

Applications of the Convex Hull (contd…)

- **Finding the Rainfall Area:** Determine total area of rainfall by analyzing all censors those send signal of rainfall using the convex hull.
- **Uses in Image Editing Software i.e. Photoshop**.

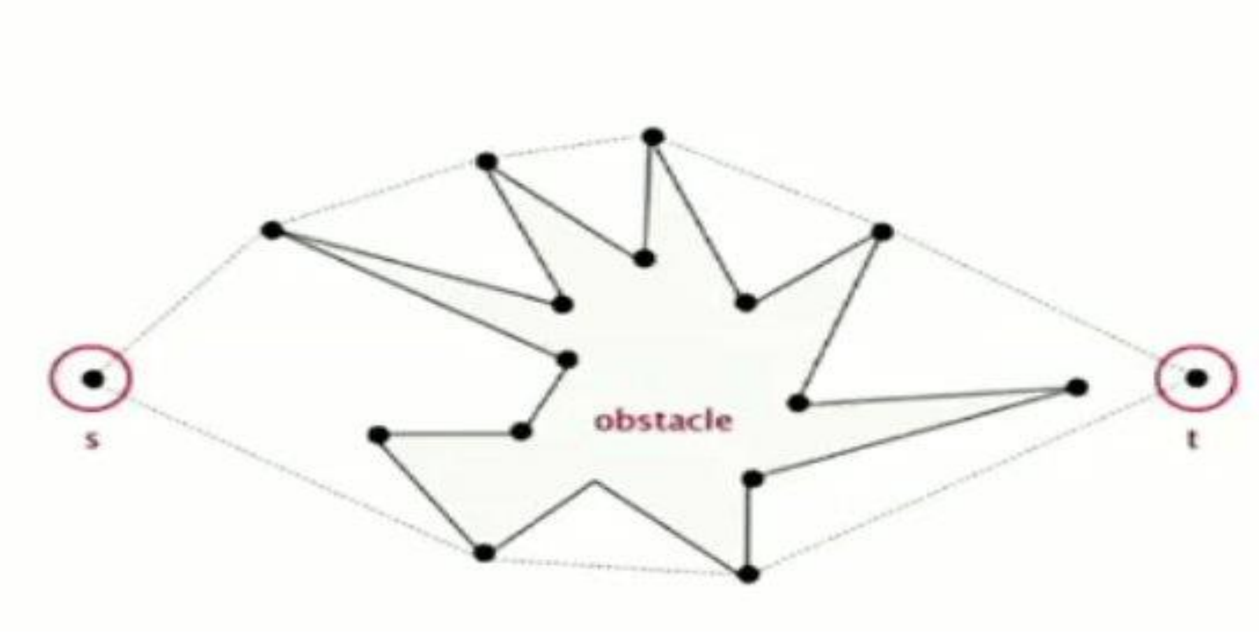
  - Magic-wand Tool

  - Glow & Shadow Effect on Layer. Better explained by applying on non-rectangular image.

  - Make a selection by ctrl+click on Layer.

Applications of the Convex Hull (contd…)

- **Robot Motion Planning:** In order to get from s to t, the shortest path will either be the straight line from s to t (if the obstacle doesn't intersect it) or one of the two polygonal chains of the convex hull.



Applications of the Convex Hull (contd…)

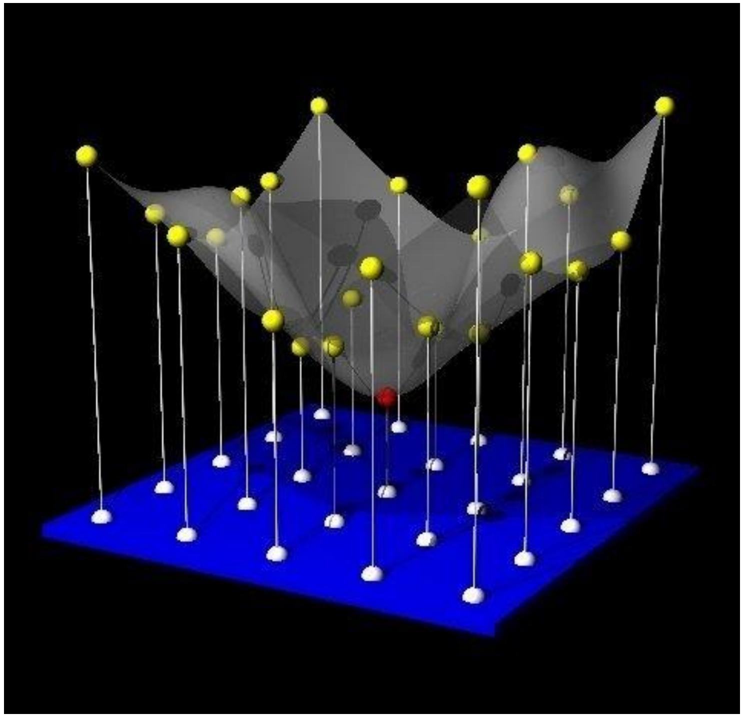
- **Nuclear/Chemical Leak Evacuation:** Imagine a modern city with censors positioned uniformly all over. When a disaster such a chemical leak or nuclear radiation leak, one way to determine the perimeter for immediate evacuation is to construct the convex hull of areas with radiation levels (exceeding a certain threshold).
- **Tracking Disease Epidemic:** Keeping track of the spatial extend of a disease outbreak could be done using the convex hull. A specific example in tracking animal epidemic is given here: Spatial extent of an outbreak in animal epidemics.

Applications of the Convex Hull (contd…)

- **Linear Programming:** The Simplex Algorithm actually finds the optimal point by iterating over the vertices of the convex hull (intersection of constraint half-spaces) constructed from the linear constraints.
- **Building Block in Other Problems:** Computing the diameter of a point set -- the distance between two farthest points. A pre-processing step is to first compute the convex hull and then find the pair of points that are farthest apart. Another is computing convex layers by repeatedly taking the convex hull.

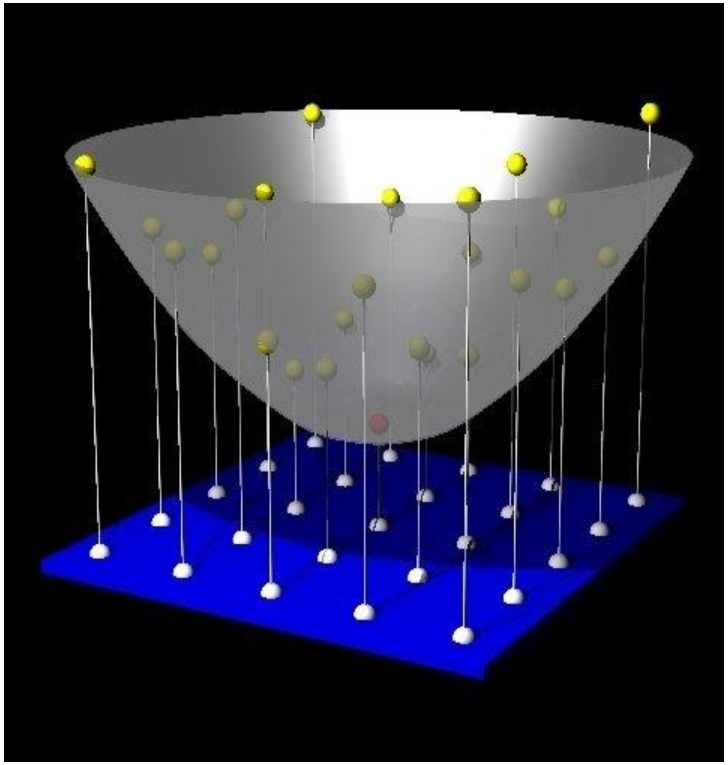
Applications of the Convex Hull (contd…)

- **Differential Geometry**: Construction of **convex relaxations.** Roughly speaking, this is a way to find the 'closest' convex problem to a non-convex problem we attempt to solve. Let us solve a problem whose constraint surface is given below.
- The yellow points represent the possible integer solutions that lie on this surface. This is ugly and has a lot of minima. But, if we look around the true global minima, then we see that we could construct a convex constraint that 'envelopes' this surface:



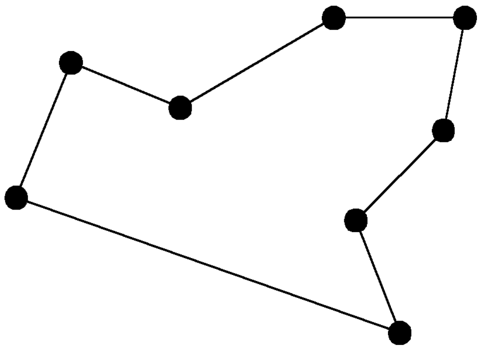
Applications of the Convex Hull (contd…)

- Now one way to construct such a relaxation is to take the convex hull of all the minima (local and global). In this case, we have a convex surface that contains true global minima; hence we've 'relaxed' the problem to a convex setting. This isn't always possible in practical situations.

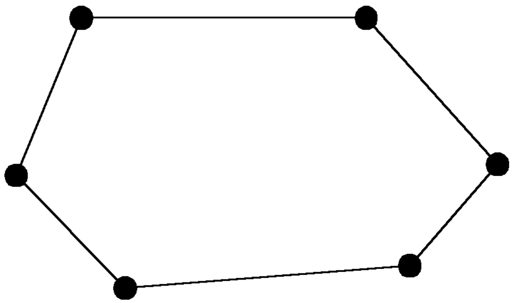


The Convex Hull Problem

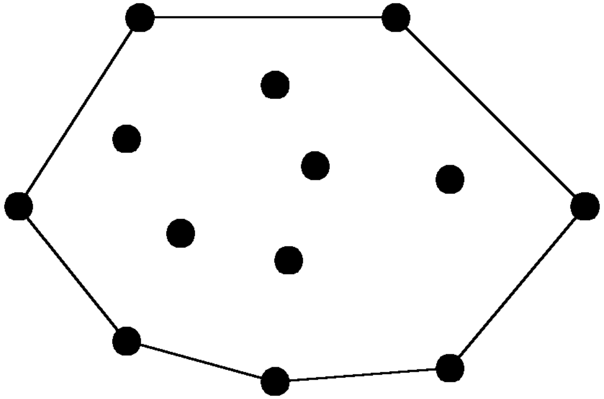
Concave Polygon:



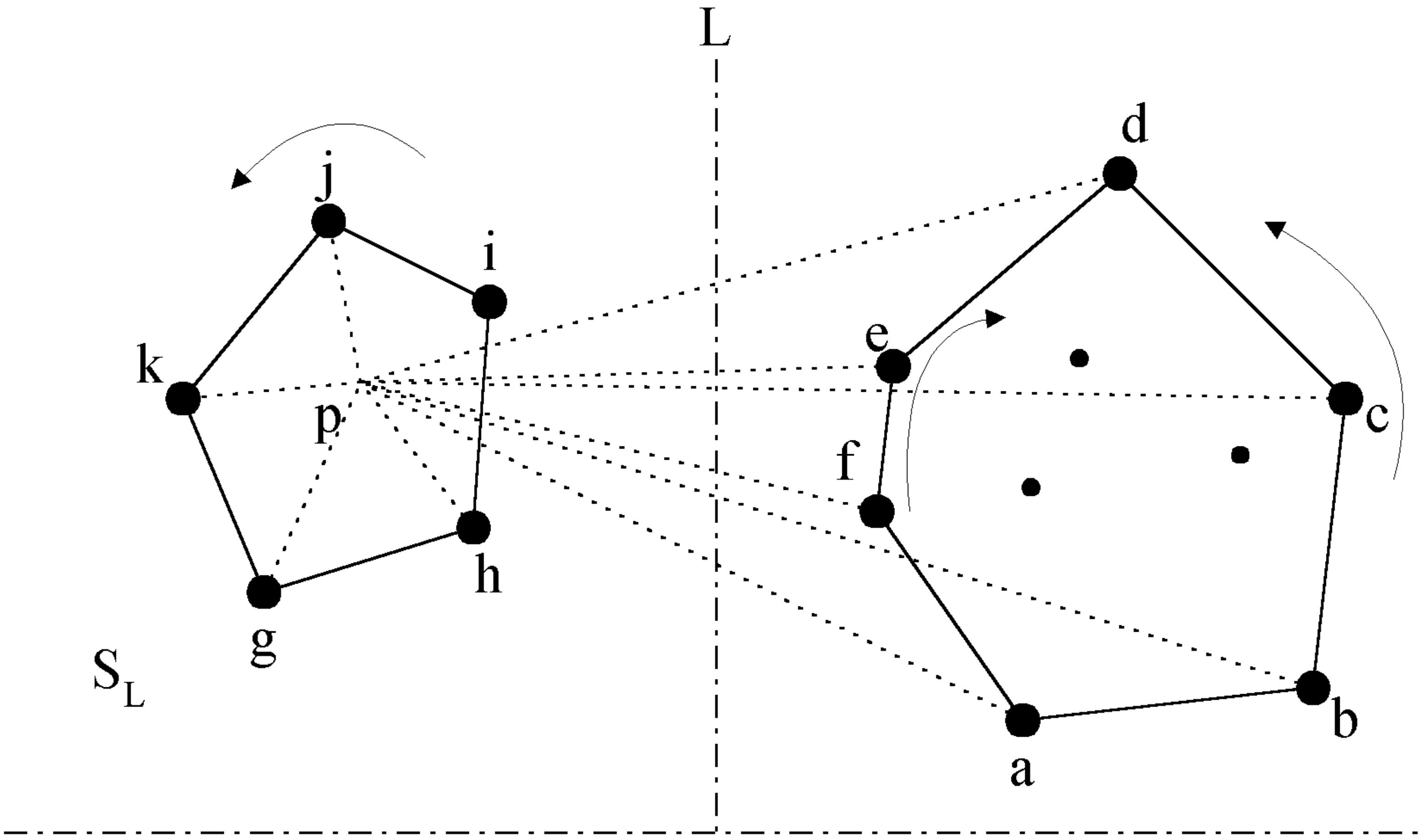
Convex Polygon:



- The convex hull of a set of planar points is the smallest convex polygon containing all of the points.



- **The divide-and-conquer strategy to solve the Convex-Hull Problem:**



- **The Merging Procedure:**

1. Select an interior point p.
2. There are 3 sequences of points which have increasing polar angles with respect to p.

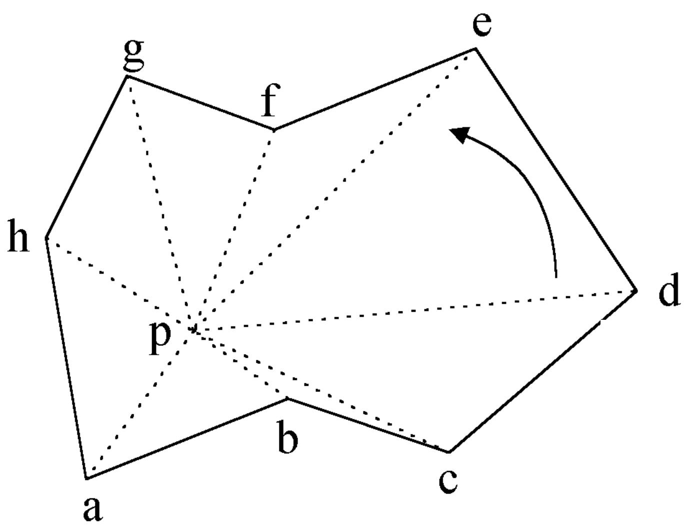
   (1) g, h, i, j, k

   (2) a, b, c, d

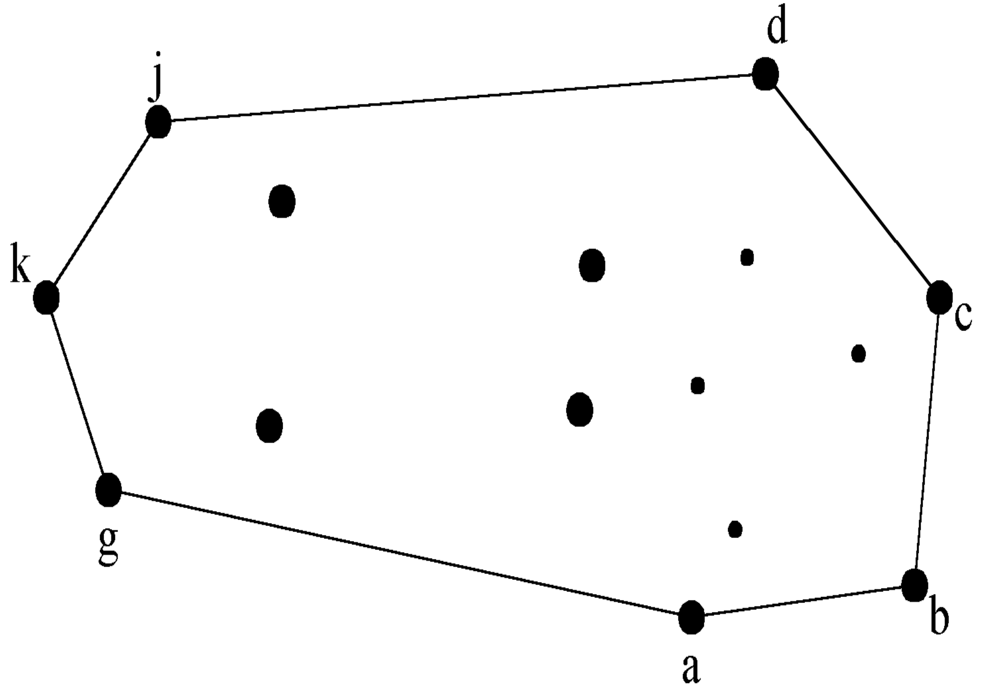
   (3) f, e
3. Merge these 3 sequences into 1 sequence:

   g, h, a, b, f, c, e, d, i, j, k.
4. Apply Graham Scan to examine the points one by one and eliminate the points which cause Reflexive Angles.

- Example: Points b and f need to be deleted.



Final Result:



Divide-and-Conquer for Convex Hull

- Input : A set S of planar points
- Output : A convex hull for S

Step 1: If S contains no more than five points, use exhaustive searching to find the convex hull and return.

Step 2: Find a median line perpendicular to the X-axis which divides S into S_L and S_R ; S_L lies to the left of S_R .

Step 3: Recursively construct the convex hulls for S_L and S_R, which are denoted by Hull(S_L) and Hull(S_R), respectively.

Step 4: Apply the merging procedure for merging the Hull(S_L) and Hull(S_R) together to form a convex hull.

- **Time Complexity**:

$T(n) = 2T(n/2) + O(n) = O(n \log n)$

Polynomial Multiplication Problem

- Polynomial:

$$p(x) = 5 + 2x + 8x^2 + 3x^3 + 4x^4$$

- In general,

$$p(x) = \sum_{i=0}^{n-1} a_i x^i$$

or

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

Polynomial Evaluation

- **Horner's Rule:**
  - Given coefficients (a_0,a_1,a_2,…,a_{n-1}), defining polynomial

$$p(x)=\sum_{i=0}^{n-1} a_i x^i$$

  - Given x, we can evaluate p(x) in O(n) time using the equation

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

- **Eval**(A,x): [Where A=(a_0,a_1,a_2,…,a_{n-1})]
  - If n=1, then return a_0
  - Else,
    - Let A'=(a_1,a_2,…,a_{n-1}) [assume this can be done in constant time]
    - return a_0+x***Eval**(A',x)

Polynomial Multiplication

- Given coefficients (a_0,a_1,a_2,…,a_{n-1}) and (b_0,b_1,b_2,…,b_{n-1}) defining two polynomials, p() and q(), and number x, compute p(x)q(x).

- Horner's rule doesn't help, since

$$p(x)q(x) = \sum_{i=0}^{n-1} c_i x^i$$

where

$$\boxed{c_i = \sum_{j=0}^{i} a_j b_{i-j}}$$

⇨ Convolution of a_i and b_j

- A straightforward evaluation using DFT would take O(n^2) time. However, the "magical" FFT will do it very fast in O(n log n) time.

Polynomial Interpolation and Polynomial Multiplication

- Given a set of n points in the plane with distinct x-coordinates, there is **exactly one** (n-1)-degree polynomial going through all these points.
- Alternate approach to computing p(x)q(x):
  - Calculate p() on 2n x-values, $x_0,x_1,\ldots,x_{2n-1}$.
  - Calculate q() on the same 2n x values.
  - Find the (2n-1)-degree polynomial that goes through the points $\{(x_0,p(x_0)q(x_0)), (x_1,p(x_1)q(x_1)), \ldots, (x_{2n-1},p(x_{2n-1})q(x_{2n-1}))\}$.
- Unfortunately, a straightforward evaluation would still take $O(n^2)$ time, as we need to apply an O(n)-time Horner's Rule evaluation to 2n different points.
- The "magical" FFT algorithm will do it efficiently in O(n log n) time, by picking 2n points that are easy to evaluate…

Primitive Roots of Unity

- A number ω is a ***primitive n-th root of unity***, for n>1, if $\omega^n = 1$ then
  - The numbers 1, ω, ω^2, …, ω^{n-1} are all distinct

Example 1:

- Z^*_{11}:

x	x^2	x^3	x^4	x^5	x^6	x^7	x^8	x^9	x^10
1	1	1	1	1	1	1	1	1	1
2	4	8	5	10	9	7	3	6	1
3	9	5	4	1	3	9	5	4	1
4	5	9	3	1	4	5	9	3	1
5	3	4	9	1	5	3	4	9	1
6	3	7	9	10	5	8	4	2	1
7	5	2	3	10	4	6	9	8	1
8	9	6	4	10	3	2	5	7	1
9	4	3	5	1	9	4	3	5	1
10	1	10	1	10	1	10	1	10	1

- 2, 6, 7, 8 are 10-th roots of unity in Z^*_{11}
- $2^2=4$, $6^2=3$, $7^2=5$, $8^2=9$ are 5-th roots of unity in Z^*_{11}
- $2^{-1}=6$, $3^{-1}=4$, $4^{-1}=3$, $5^{-1}=9$, $6^{-1}=2$, $7^{-1}=8$, $8^{-1}=7$, $9^{-1}=5$

Example 2: The complex number $e^{2\pi \mathbf{i}/n}$ is a primitive n-th root of unity, where, $i = \sqrt{-1}$

Properties of Primitive Roots of Unity

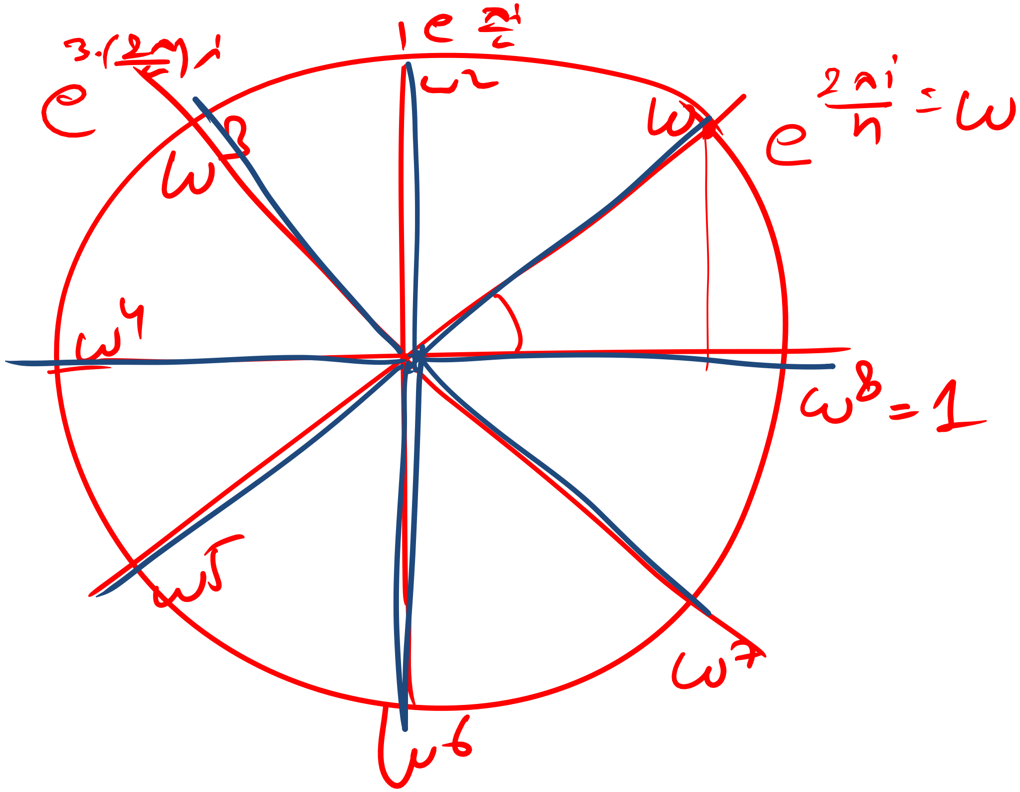
- **Inverse Property:** If ω is a primitive root of unity, then $\omega^{-1}=\omega^{n-1}$
  - Proof: $\omega\omega^{n-1}=\omega^{n}=1$
- **Cancellation Property:** For non-zero -n<k<n, $\sum_{j=0}^{n-1}\omega^{kj}=0$
  - Proof:

$$\sum_{j=0}^{n-1}\omega^{kj}=\frac{(\omega^{k})^{n}-1}{\omega^{k}-1}=\frac{(\omega^{n})^{k}-1}{\omega^{k}-1}=\frac{(1)^{k}-1}{\omega^{k}-1}=\frac{1-1}{\omega^{k}-1}=0$$

- **Reduction Property:** If ω is a primitive (2n)-th root of unity, then ω^2 is a primitive n-th root of unity.
  - Proof: If $1,\omega,\omega^2,\ldots,\omega^{2n-1}$ are all distinct, so are $1,\omega^2,(\omega^2)^2,\ldots,(\omega^2)^{n-1}$
- **Reflective Property:** If n is even, then $\omega^{n/2} = -1$.
  - Proof: By the cancellation property, for k=n/2:

$$0=\sum_{j=0}^{n-1}\omega^{(n/2)j}=\omega^{0}+\omega^{n/2}+\omega^{0}+\omega^{n/2}+\cdots+\omega^{0}+\omega^{n/2}=(n/2)(1+\omega^{n/2})$$

  - Corollary: $\omega^{k+n/2}= -\omega^{k}$.



$n = 8$

$$\omega = e^{\frac{2\pi i}{n}}$$

$$= \cos\frac{2\pi}{n} + i\sin\frac{2\pi}{n}$$

$$p(x) = P_{even}(x^2) + x P_{odd}(x^2)$$

$$p(x_1) = P_e(x_1^2) + x_1 P_o(x_1^2)$$

$$p(x_{2n}) = P_e(x_{2n}^2) + x_{2n} P_o(x_{2n}^2)$$

$x_1 = \omega$

$x_2 = \omega^2$

$x_3 = \omega^3$

$\vdots$

$x_n = \omega^n$

$\vdots$

x_{2n} $\omega_{2n}^{2n} = 1$

P_e is getting evaluated at

$\omega^2, \omega^4, \omega^6$

$x_{n+1}^2 =$ " , "

$$x_n^2 = (\omega^n)^2 = \omega^{2n}$$

$p(\omega)$ $\quad p(x_1) = y_1$

$p(\omega^2)$ $\quad p(x_2) = y_2$

$p(\omega^3)$ $\quad \vdots$

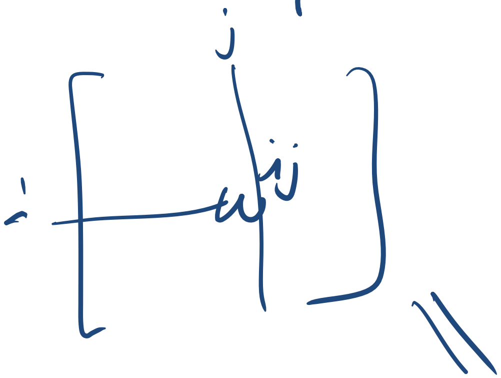
$p(x_n) = y_n$

$$p(x) = a_0 + a_1x^1 + a_2x^2 + \dots a_{n-1}x^{n-1}$$

$$a_0 + a_1x_1 + a_2x_1^2 + a_3x_1^3 + \dots + a_{n-1}x_1^{n-1} = y_1$$

$$\vdots$$

$$a_0 + a_1x_n + a_2x_n^2 + a_3x_n^3 + \dots \; a_{n-1}x_n^{n-1} = y_n$$



$$\begin{bmatrix} 1 & x_1 & x_1^2 & x_1^3 & & x_1^{n-1} \\ 1 & x_2 & x_2^2 & x_2^3 & \cdot & x_2^{n-1} \\ & & & & & \\ & & F & & & \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \\ \\ y_{n-1} \end{bmatrix}$$

$$Fa = y$$

$$a = F^{-1}y$$

The Discrete Fourier Transform

- Given coefficients $(a_0,a_1,a_2,\ldots,a_{n-1})$ for an (n-1)-degree polynomial p(x)
- The **Discrete Fourier Transform** is to evaluate p at the values
  - $1,\omega,\omega^2,\ldots,\omega^{n-1}$
  - We produce $(y_0,y_1,y_2,\ldots,y_{n-1})$, where $y_j=p(\omega^j)$
  - That is,

$$y_j = \underline{\sum_{i=0}^{n-1} a_i \omega^{ij}}$$

$$\begin{array}{cc} & j \\ i & \begin{bmatrix} \omega^{ij} \end{bmatrix} \end{array} \begin{bmatrix} a_0 \\ a_1 \\ \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \\ \\ \end{bmatrix}$$

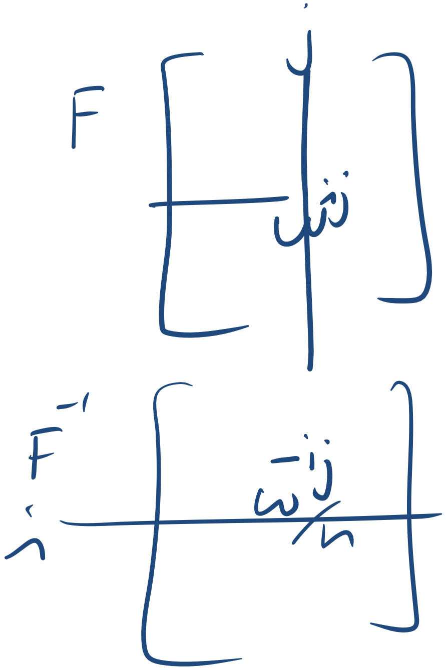
  - Matrix form: **y=Fa,** where **F**$[i,j]=\omega^{ij}$.

- The **Inverse Discrete Fourier Transform** recovers the coefficients of an (n-1)-degree polynomial given its values at $1,\omega,\omega^2,\ldots,\omega^{n-1}$
  - Matrix form: $\mathbf{a}=\mathbf{F}^{-1}\mathbf{y}$, where $\mathbf{F}^{-1}[i,j]=\omega^{-ij}/n$.

Correctness of the Inverse DFT (IDFT)

- The DFT and inverse DFT really are inverse operations
- Proof: Let **A=F^{-1}F**. We want to show that **A=I**, where

$$A[i,j]=\frac{1}{n}\sum_{k=0}^{n-1}\omega^{-ki}\omega^{kj}$$



- If i=j, then

$$A[i,i]=\frac{1}{n}\sum_{k=0}^{n-1}\omega^{-ki}\omega^{ki}=\frac{1}{n}\sum_{k=0}^{n-1}\omega^{0}=\frac{1}{n}n=1$$

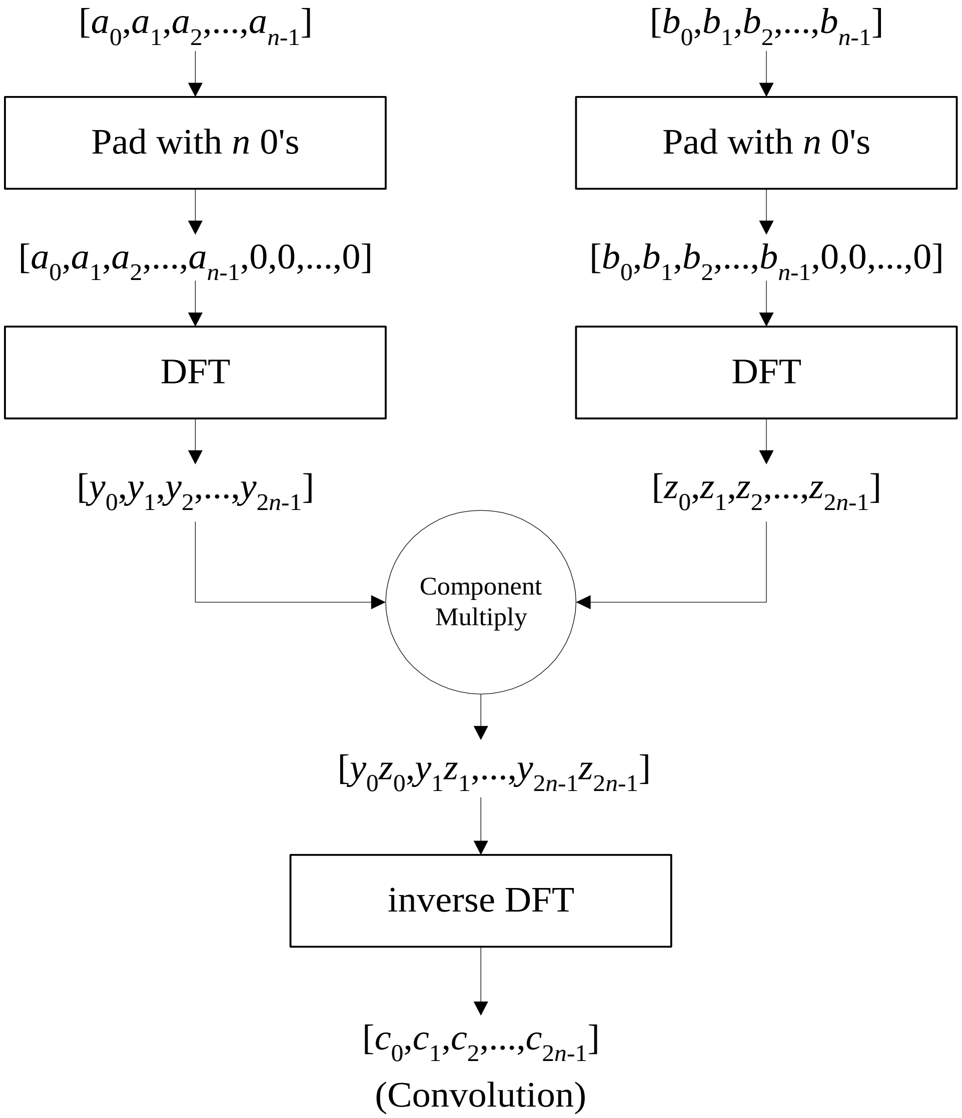
- If i and j are different, then

$$A[i,j]=\frac{1}{n}\sum_{k=0}^{n-1}\omega^{(j-i)k}=0 \qquad \text{(by Cancellation Property)}$$

Convolution

- The DFT and its Inverse (IDFT) can be used to multiply any two given polynomials.

- We can get the coefficients of product polynomial quickly if we can compute the DFT (and its Inverse (IDFT)) quickly…



$[a_0,a_1,a_2,\ldots,a_{n-1}]$
$[b_0,b_1,b_2,\ldots,b_{n-1}]$
Pad with n 0's
Pad with n 0's
$[a_0,a_1,a_2,\ldots,a_{n-1},0,0,\ldots,0]$
$[b_0,b_1,b_2,\ldots,b_{n-1},0,0,\ldots,0]$
DFT
DFT
$[y_0,y_1,y_2,\ldots,y_{2n-1}]$
$[z_0,z_1,z_2,\ldots,z_{2n-1}]$
Component Multiply
$[y_0z_0,y_1z_1,\ldots,y_{2n-1}z_{2n-1}]$
inverse DFT
$[c_0,c_1,c_2,\ldots,c_{2n-1}]$
(Convolution)

The Fast Fourier Transform (FFT)

- The FFT is an efficient algorithm for computing the DFT
- The FFT is based on the divide-and-conquer paradigm:
  - If n is even, we can divide a polynomial

$$p(x) = a_0 + a_1x + a_2x^2 + \cdots + a_{n-1}x^{n-1} \quad 1, \omega, \omega^2 \ldots \omega^{2n}$$

into two polynomials

$$\begin{aligned} p^{\text{even}}(x) &= a_0 + a_2x + a_4x^2 + \cdots + a_{n-2}x^{n/2-1} \\ p^{\text{odd}}(x) &= a_1 + a_3x + a_5x^2 + \cdots + a_{n-1}x^{n/2-1} \end{aligned}$$

$1, \underline{\omega}, \omega^2, \omega^3\ \omega^{n/2} \ldots \omega^{2n}$

~~$1, \omega^2, \omega^4\ \omega^{n/2} \ldots \omega^{2n}$~~

$$p(x) = p^{\text{even}}(x^2) + xp^{\text{odd}}(x^2).$$

$$\cancel{\omega}\, p(\omega) = \underline{p^e(\omega^2)} + \underline{\omega p^o(\omega^2)}$$

$$\boxed{\begin{aligned} & p(\omega^{n/2+1}) \\ & = \underline{p^e(\omega^2)} + \cancel{\omega^{n/2+1}}\, p^o(\omega^2) \\ & \qquad\quad -\omega \end{aligned}}$$

The FFT Algorithm

$$T(n) = 2T(\tfrac{n}{2}) + O(n)$$

```
Algorithm FFT($\mathbf{a}, \omega$):
    Input: An $n$-length coefficient vector $\mathbf{a} = [a_0, a_1, \ldots, a_{n-1}]$ and a primitive $n$th
        root of unity $\omega$, where $n$ is a power of 2
    Output: A vector $\mathbf{y}$ of values of the polynomial for $\mathbf{a}$ at the $n$th roots of unity
  if $n = 1$ then
      return $\mathbf{y} = \mathbf{a}$.
  $x \leftarrow \omega^0$        {$x$ will store powers of $\omega$, so initially $x = 1$.}
  {Divide Step, which separates even and odd indices}
  $\mathbf{a}^{\text{even}} \leftarrow [a_0, a_2, a_4, \ldots, a_{n-2}]$
  $\mathbf{a}^{\text{odd}} \leftarrow [a_1, a_3, a_5, \ldots, a_{n-1}]$
  {Recursive Calls, with $\omega^2$ as $(n/2)$th root of unity, by the reduction property}
  $\boxed{\mathbf{y}^{\text{even}}} \leftarrow \text{FFT}(\mathbf{a}^{\text{even}}, \omega^2)$
  $\mathbf{y}^{\text{odd}} \leftarrow \text{FFT}(\mathbf{a}^{\text{odd}}, \omega^2)$
  {Combine Step, using $x = \omega^i$}
  for $i \leftarrow 0$ to $n/2 - 1$ do
      $y_i \leftarrow y_i^{\text{even}} + x \cdot y_i^{\text{odd}}$
      $y_{i+n/2} \leftarrow y_i^{\text{even}} - x \cdot y_i^{\text{odd}}$        {Uses reflective property}
      $x \leftarrow x \cdot \omega$
  return $\mathbf{y}$
```

$\omega^2, \omega^4, \omega^6, \omega^8 \;.\;.\; \omega^{2n}$

$y = Fa$

$y_1 = P(\omega)$

$y_2 = P(\omega^2)$

$y_3 = P(\omega^3)$

$\vdots$

$y_n = p(\omega^n)$

The Running Time of FFT is O(n log n). Running Time of Inverse FFT is same.