Algorithms and Lab (CSE130)

Algorithm Design Techniques

Muhammad Tariq Mahmood

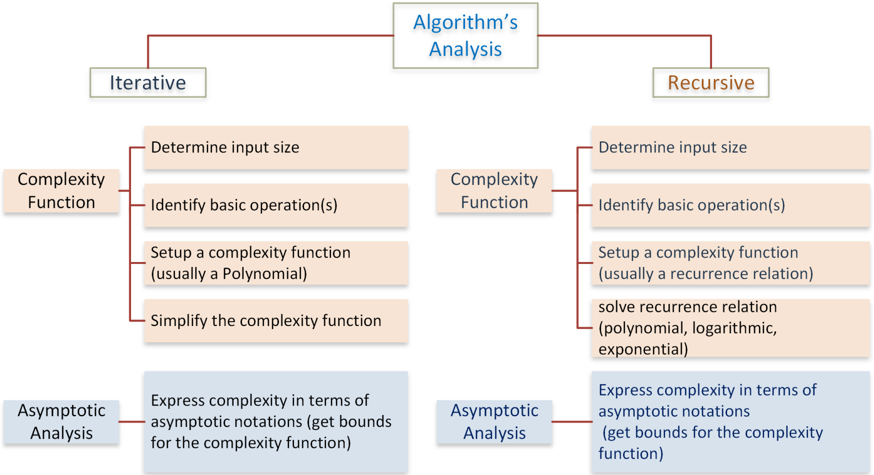
tariq@koreatech.ac.kr
School of Computer Science and Engineering

Note: These notes are prepared from the following resources.

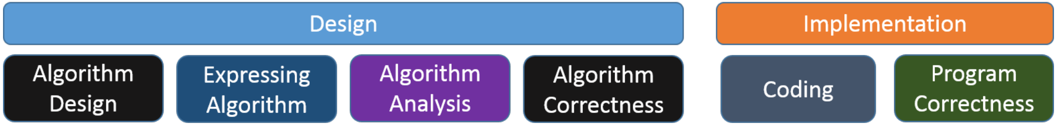
- **(main text)Foundations of Algorithms, by Richard Neapolitan and Kumarss Naimipour**
- **Python Algorithm (파이썬 알고리즘) by Y.K. Choi (2021) (Korean)**
- **Introduction to the Design and Analysis of Algorithms by Anany Levitin**
- **Introduction to Algorithms, by By Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein**
- https://www.geeksforgeeks.org

Last Week : Week03

- A general framework for complexity analysis of iterative and recursive algorithms


Algorithm's Analysis
Iterative
Recursive
Complexity Function
Determine input size
Identify basic operation(s)
Setup a complexity function (usually a Polynomial)
Simplify the complexity function
Asymptotic Analysis
Express complexity in terms of asymptotic notations (get bounds for the complexity function)
Complexity Function
Determine input size
Identify basic operation(s)
Setup a complexity function (usually a recurrence relation)
solve recurrence relation (polynomial, logarithmic, exponential)
Asymptotic Analysis
Express complexity in terms of asymptotic notations (get bounds for the complexity function)

Design and Implementation Phases



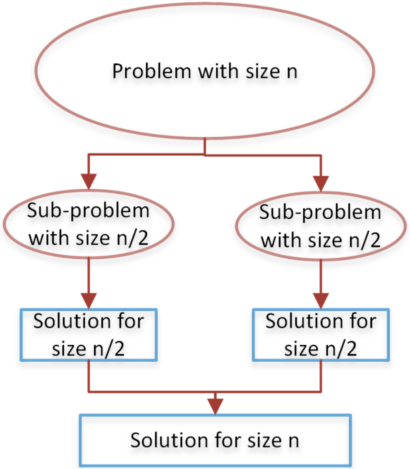
Algorithm design techniques

- There are general approaches to construct efficient solutions to problems. They provide templates suited to solving a broad range of diverse problems.
- Well-known design techniques
  - Brute Force
  - Divide and conquer
  - decrease and conquer
  - transform and conquer
  - Dynamic programming
  - Greedy approach
  - State space search techniques

Contents

Divide-and-Conquer

- The divide-and-conquer approach is a top-down approach
- The divide-and-conquer approach employs this same strategy on a smaller instance of the same problem.
- That is, it divides an instance of a problem into two or more smaller instances (sub-problems).
- The smaller instances are usually instances of the original problem.
- If solutions to the smaller instances can be obtained readily, the solution to the original instance can be obtained by combining these solutions.
- If the smaller instances are still too large to be solved readily, they can be divided into still smaller instances.

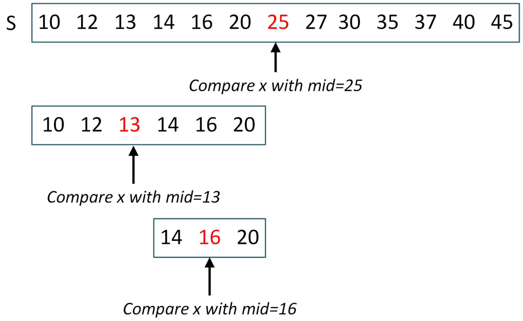


Divide-and-Conquer (cont...)

- Example: Binary Search

```
procedure location($low$, $high$)
    if ($low > high$) then
        return 0
    else
        $mid = \left\lfloor \frac{(low+high)}{2} \right\rfloor$
        if ($x == S[mid]$) then
            return $mid$
        else
            if ($x < S[mid]$) then
                return $location(low, mid - 1)$
            else
                return $location(mid + 1, high)$
            end if
        end if
    end if
end procedure
```

- Steps in binary search algorithm to find x=16 are illustrated below



Divide-and-Conquer (cont...)

- Time Complexity Analysis

  - **Complexity Function:**

$$\begin{cases} T(1) = 1, & n = 1 \\ T(n) = T\left(\frac{n}{2}\right) + 1, & n > 1 \end{cases}$$

  - **Solution:**

$$T(2) = T\left(\frac{2}{2}\right) + 1 \Rightarrow T(1) + 1 = 2$$

$$T(4) = T\left(\frac{4}{2}\right) + 1 \Rightarrow T(2) + 1 = 3$$

$$T(8) = T\left(\frac{8}{2}\right) + 1 \Rightarrow T(4) + 1 = 4$$

$$T(16) = T\left(\frac{16}{2}\right) + 1 \Rightarrow T(8) + 1 = 5$$

It appears

$$T(n) = \lg n + 1$$

  - **Complexity in terms of asymptomatic notations**

    1. upper bound

$$\begin{aligned} &\textit{For } n \geq 2, c_1 = 2, \\ &\lg n + 1 \leq c_1 \lg n \quad \textit{holds} \\ &\textit{hence} \quad \lg n + 1 \in \mathrm{O}(\lg n) \end{aligned}$$

    2. lower bound

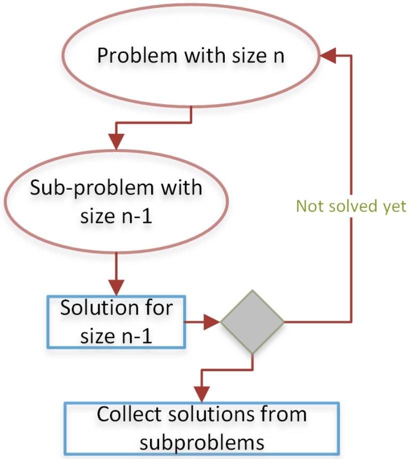
$$\begin{aligned} &\textit{For } n \geq 2, \quad c_2 = 1, \\ &c_2 \lg n \leq \lg n + 1 \quad \textit{holds} \\ &\textit{hence} \quad \lg n + 1 \in \Omega(\lg n) \end{aligned}$$

    3. upper and lower bounds

$$\begin{aligned} &\textit{For } n \geq 2, \quad c_1 = 2, \quad c_2 = 1, \\ &c_2 \lg n \leq \lg n + 1 \leq c_1 \lg n \quad \textit{holds} \\ &\textit{hence} \quad \lg n + 1 \in \Theta(\lg n) \end{aligned}$$

Decrease-and-Conquer

- decrease-and-conquer technique is based on exploiting the relationship between a solution to a given instance of a problem and a solution to its smaller instance.
- There are three major variations of decrease-and-conquer:
  1. decrease by a constant
  2. decrease by a constant factor
  3. variable size decrease
- The decrease-by-a-constant-factor technique suggests reducing a problem instance by the same constant factor on each iteration of the algorithm



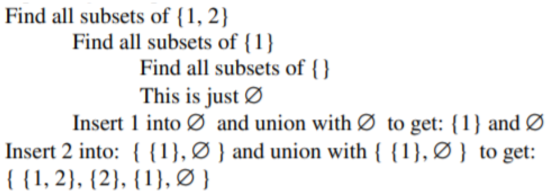
Decrease-and-Conquer (cont...)

Example: Computing power set

- Design a decrease-by-one algorithm for generating the power set of a set of n elements. The power set of a set S is the set of all the subsets of S, including the empty set and S itself.
- Here is a general outline of a recursive algorithm that create list L(n) of all the subsets of $S = \{a_1, a_2, ..., a_n\}$

```
procedure powerSet(L,n)
    if $(n == 0)$ then
        return $L(0) = \phi$
    else
        powerSet(L,n-1)
        L(n-1)=subsets of $\{a_1, a_2, ..., a_{n-1}\}$
        $T = append\ a_n\ to\ each\ in\ L(n-1)$
        return $L(n) = L(n-1)\ union\ T$
    end if
end procedure
```

- For example: Final all subsets of $S = \{1, 2\}$

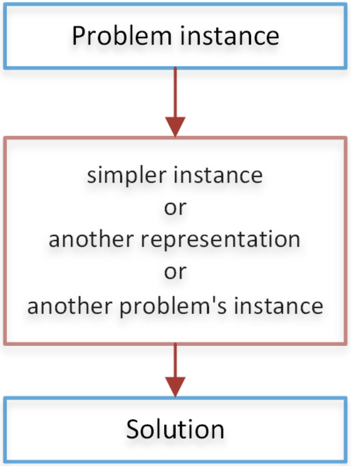


Find all subsets of {1, 2}
    Find all subsets of {1}
        Find all subsets of { }
        This is just ∅
    Insert 1 into ∅ and union with ∅ to get: {1} and ∅
Insert 2 into: { {1}, ∅ } and union with { {1}, ∅ } to get:
{ {1, 2}, {2}, {1}, ∅ }

$$T(n) \in \Theta(n2^n)$$

Transform-and-Conquer

- The transform-and-conquer design technique is based on the idea of transformation.
- There are three major variations of this idea
  1. instance simplification: Transformation to a simpler or more convenient instance of the same problem.
  2. representation change: Transformation to a different representation of the same instance.
  3. problem reduction: Transformation to an instance of a different problem for which an algorithm is already available.
- A variety of growth rates that are representative of typical algorithms are shown.



Transform-and-Conquer (cont...)

Example: finding intersection of two sets problem

- Let $A = \{a_1, a_2, ..., a_n\}$ and $B = \{b_1, b_2, ..., b_n\}$ be two sets of numbers, find $C = A \cap B$. i.e., the set C of all the numbers that are in both A and B.

- A brute-force algorithm

```
procedure intersection( $A[0..n], B[0..m]$)
    for ($i = 1; i <= n; i++$) do
        for ($j = 1; j <= m; j++$) do
            if ($A[i] == B[j]$) then
                $C.add(A[i])$
            end if
        end for
    end for
    return C
end procedure
```

- Complexity Analysis

$$T(n, m) = \sum_{i=1}^{n} \sum_{j=1}^{m} 1 = nm \in \Theta(nm)$$

- Sort the lists representing sets A and B and output the values common values to the two lists.

- Sorting is done with $\Theta(n.log_2(n)$ algorithm,

- A transform- based (presorting-based) algorithm

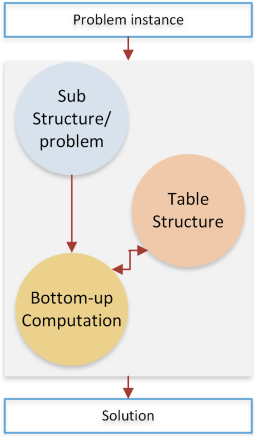
```
procedure intersection( $A[0..n], B[0..m]$)
    A=sort(A)
    B=sort(B)
    while (i < m and j < n) do
        if ($A[i] < B[j]$) then $i += 1$
        else
            if ($A[i] > B[j]$) then $j += 1$
            else
                $C.add(A[i])$ $i += 1$, $j += 1$
            end if
        end if
    end while
    return C
end procedure
```

- Complexity Analysis

$$T(n, m) = \Theta(n.log_2(n) + \Theta(m.log_2(m)) + \Theta(n + m)$$
$$T(n, m) \in \Theta(s.log_2(s), s = max(n, m)$$

Dynamic Programming Approach

- Dynamic programming is a bottom-up approach for solving problems with overlapping subproblems.
- There are basically three elements that characterize a dynamic programming algorithm:
  1. Substructure: Decompose the given problem into smaller subproblems. Express the solution of the original problem in terms of the solution for smaller problems. (Establish a recursive property)
  2. Table Structure: After solving the sub-problems, store the results to the sub problems in a table.
  3. Bottom-up Computation: Using table, combine the solution of smaller subproblems to solve larger subproblems and eventually arrives at a solution to complete problem.



- The word "programming" in the name of this technique stands for "planning" and does not refer to computer programming.

Dynamic Programming (cont...)

Example: Computing Binomial Coefficients

- Binomial Theorem

$$(a+b)^n = \sum_{k=0}^{n} \frac{n!}{k!\,(n-k)} a^k b^{n-k}$$

- Binomial coefficients:

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)}$$

- Another Representation:

$$\binom{n}{k} = \begin{cases} \binom{n-1}{k-1} + \binom{n-1}{k}, & 0 < k < n \\ 1 & k = 0 \ or \ \ k = n \end{cases}$$

$$\begin{aligned} C(n,k) &= n!/(k! * (n-k)!) \\ &= (n-1)! * n/(k! * (n-k)!) \\ &= (n-1)! * n/((k-1)! * k * (n-k-2)! * (n-k-1) * (n-k)) \\ &= (n-1)!/((k-1)! * (n-k-1)!) * (n/k * (n-k)) \\ &= [(n-1)!/((k-1)! * (n-k-1)!)] * (1/(n-k) + 1/k) \\ &= (n-1)!/((k-1)! * (n-k)!) + (n-1)!/(k! * (n-k-1)!) \\ &= C(n-1,k-1) + C(n-1,k) \end{aligned}$$

- Establish a recursive property.

$$B[i][j] = \begin{cases} B[i-1][j-1] + B[i-1][j], & 0 < j < i \\ 1 & , \quad j = 0 \ \ or \ j = i \end{cases}$$

Dynamic Programming (cont...)

- Algorithm

```
procedure bc( $n$, $k$)
    integer $i, j$
    integer $B[0..n][0..k]$
    for ($i = 0; i <= n; i++$) do
        for ($j = 0; j <= min(i, k); j++$) do
            if ($j == 0 \mid\mid j == i$) then
                $B[i][j] = 1$
            else
                $B[i][j] = B[i-1][j-1] + B[i-1][j]$
            end if
        end for
    end for
end procedure
```

	0	1	2	3	4	j	k
0	1						
1	1	1					
2	1	2	1				
3	1	3	3	1			
4	1	4	6	4	1		
i	$B[i][j] = \begin{cases} B[i-1][j-1] + B[i-1][j], & 0 < j < i \\ 1, & j = 0 \text{ or } j = i \end{cases}$						
n							

- Time complexity function : $T(n,k) = T_1(n,k) + T_2(n,k) \in \Theta(nk)$

$T_1(n,k) = \sum_{i=1}^{k} \sum_{j=1}^{i} 1 = \sum_{i=1}^{k} i = \frac{k(k+1)}{2}, (i \le k)$

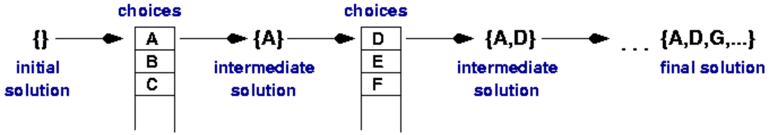
$T_2(n,k) = \sum_{i=k+1}^{n+1} \sum_{j=1}^{k+1} 1 = (n-k+1)(k+1), (i > k)$

Greedy Approach

Greedy Approach

- A **greedy algorithm** arrives at a solution by making a sequence of choices, each of which simply looks the best at the moment.

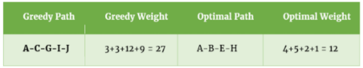
- The choice is: 1) feasible : it has to satisfy the problem's constraints, 2) locally optimal : it has to be the best local choice among all feasible choices available on that step, 3) irrevocable: once made, it cannot be changed on subsequent steps of the algorithm



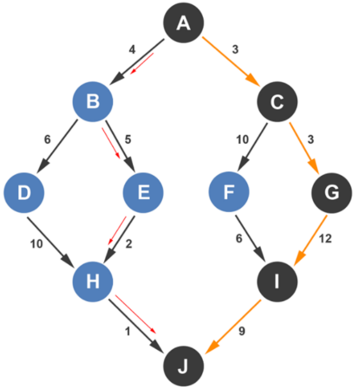
- Each iteration consists of the following components
  1. A selection procedure chooses the next item to add to the set. The selection is performed according to a greedy criterion that satisfies some locally optimal consideration at the time.
  2. A feasibility check determines if the new set is feasible by checking whether it is possible to complete this set in such a way as to give a solution to the instance.
  3. A solution check determines whether the new set constitutes a solution to the instance.

Greedy Approach (cont...)

- Let's look at an example of the greedy algorithm in action. We have a directed graph with weighted edges.
- If we're trying to make our way from A to J, the greedy algorithm will examine all the paths that are immediately connected to A, which are edges to B and C.
- The weight of edge A-C is smaller than A-B, so the algorithm chooses A-C.
- The greedy algorithm continues and chooses immediate lower weighted edges.
- Some algorithms that utilize the greedy approach are Kruskal's algorithm, Prim's algorithm, and Dijkstra's algorithm.



Greedy Path	Greedy Weight	Optimal Path	Optimal Weight
A-C-G-I-J	3+3+12+9 = 27	A-B-E-H	4+5+2+1 = 12

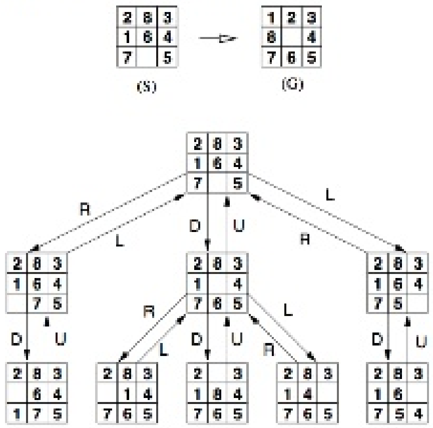


State Space Search

- A problem can be represented by a State Space Tree where nodes represent states.
  1. Set of states
  2. Operators and cost
  3. Start state
  4. Goal state

- The solution/goal state) can be found by searching the State Space Tree.

- Different search algorithms are applied to explore the state space
  - Depth first search
  - Breadth first search
  - heuristic search
  - Backtracking
  - Branch and Bound



Brute Force

Brute Force

- It is a straightforward approach to solving a problem, usually directly based on the problem statement and definitions of the concepts involved.
- A brute force algorithm simply tries all possibilities until a satisfactory solution is found
- The brute force is applicable to a very wide variety of problems
- It is usually provide algorithm with high computational complexity but it should not be overlooked due to the following reasons.
  - It is the only general approach which can tackle almost all problems
  - For some important problems including sorting, searching, matrix multiplication, and string matching, the brute-force approach yields reasonable algorithms
  - If only a few instances of a problem need to be solved then a brute-force algorithm can solve those instances with acceptable speed.

Largest zero-sum submatrix problem

Largest zero-sum sub-matrix problem :

- Given a 2D matrix, find the largest rectangular sub-matrix whose sum is 0. for example consider the following N x M input matrix. Example:
- input $= \begin{bmatrix} 9 & 7 & 16 & 5 \\ 1 & -6 & -7 & 3 \\ 1 & 8 & 7 & 9 \\ 7 & -2 & 0 & 10 \end{bmatrix}$
- output $= \begin{bmatrix} -6 & -7 \\ 8 & 7 \\ -2 & 0 \end{bmatrix}$
- The brute force solution for this problem is to check every possible rectangle in given 2D array.

- Pseudo-code to compute Integral Image

```
1: procedure computeIntegralMatrix($M[][]$)
2:     $S[n+1][m+1]$
3:     for $((i = 0; i <= n; i++))$ do
4:         for $(j = 0; j <= m; j++)$ do
5:             $S[i][j] = S[i-1][j] + S[i][j-1]$
6:             $-S[i-1][j-1] + M[i-1][j-1]$
7:         end for
8:     end for
9:     return $S$
10: end procedure
```

$$T(n) = \sum_{i=1}^{n+1} \sum_{j=1}^{m+1} 1 = (n+1)(m+1) \in \Theta(nm)$$

Largest zero-sum submatrix problem (cont...)

- A brute Force algorithm

```
procedure findMaxZroSumSubmatrix($M[1, ..n, 1, ..., m]$)
    $S[][] = computeIntegralMatrix(M)$
    $maxMS, rowStart = rowEnd = colStart = colEnd = 0$
    for $(r1 = 0; r1 < n; r1 + +)$ do
        for $(r2 = 0; r2 < m; r2 + +)$ do
            for $(c1 = 0; c1 < n; c1 + +)$ do
                for $(c2 = 0; c2 < m; c2 + +)$ do
                    $ssum = S[r2 + 1][c1 + 1] - S[r2 + 1][c2] - S[r1][c1 + 1] + S[r1][c2]$
                    MS= r2-r1 * c2-c1
                    if ssum == 0 and MS > maxMS then
                        maxMS=MS, rowStart = r1, rowEnd = r2, colStart = c1, colEnd = c2
                    end if
                end for
            end for
        end for
    end for
    return M[rowStart:colStart][rowEnd:colEnd]
end procedure
```

$$T(n, m) = \Theta(nm) + \sum_{r1=1}^{n} \sum_{r2=1}^{m} \sum_{c1=1}^{n} \sum_{c2=1}^{m} 1 = \Theta(nm) + n^2m^2 = \Theta(nm) + \Theta(n^2m^2) \in \Theta(n^2m^2)$$

Brute-Force String Matching

String Matching Problem

- A string is a sequence of characters from an alphabet. Strings of particular interest are text strings, which comprise letters, numbers, and special characters;
- String matching: given a string of n characters called the text and a string of m characters ($m \leq n$) called the pattern, find a substring of the text that matches the pattern. If the indices for Text $t_0 ... tn - 1$ and Pattern $p_0 ... pm - 1$, then

$$Text = t_0, ..., t_i, ...t_{i+j}, ..., t_{i+m-1}, ..., t_{n-1}$$
$$Pattern = p_0, ..., p_j, ...p_{m-1}$$
$$substring = t_i = p_0, ..., t_{i+j} = p_j, ..., t_{i+m-1} = p_{m-1}$$

- String matching algorithms have greatly influenced computer science and play an essential role in various real-world problems
- Applications of String Matching Algorithms:
  - Plagiarism Detection:
  - Bioinformatics and DNA Sequencing:
  - Digital Forensics:
  - Spelling Checker:
  - Spam filters:
  - Search engines or content search in large databases:
  - Intrusion Detection System:

Brute-Force String Matching (cont...)

- A brute-force algorithm for the string-matching problem
  - Input: An array $T[0..n`1]$ of n characters representing a text and an array $P[0..m`1]$ of m characters representing a pattern
  - Output: The index of the first character in the text that starts a matching substring, otherwise -1
  - Algorithm

```
1: procedure SM(T[0..n-1], P[0..m-1])
2:     for ($i = 0; i <= n - m; i++$) do
3:         j=0
4:         while ($j < m$ and $P[j] == T[i + j]$) do
5:             j=j+1
6:             if $j == m$ then
7:                 return i
8:             end if
9:         end while
10:    end for
11:    return -1
12: end procedure
```

  - Example of brute-force string matching. The pattern's characters that are compared with their text counterparts are in bold type.

```
N O B O D Y _ N O T I C E D _ H I M
N O T
  N O T
    N O T
      N O T
        N O T
          N O T
            N O T
              N O T
```

$$T(n,m) = \sum_{i=0}^{n-m}\sum_{j=0}^{m-1} 1 = (n-m+1)m = nm - m^2 + 1 \in \Theta(nm)$$

Closest-Pair by Brute Force

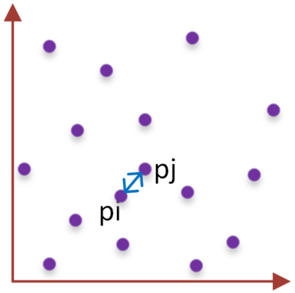
Closest-Pair Problem

- The closest-pair problem calls for finding the two closest points in a set of n points in d-dimensions.

- For simplicity, we consider the two-dimensional case of the closest-pair problem.

- The distance between two points $p_i(x_i, y_i), p_j(x_j, y_j)$ is the standard Euclidean distance.

$$d(p_i, p_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$



- Fundamental problem in many applications as well as a key step in many algorithms.
  - Dynamic minimum spanning trees
  - Straight skeletons and roof design
  - Collision detection applications
  - Hierarchical clustering
  - Robot Motion Planning
  - Traveling salesman heuristics
  - Greedy matching

Closest-Pair by Brute Force (cont...)

- A brute-force algorithm for the closest pair problem
  - Input: A list P of n ($n \geq 2$) points $p_1(x_1, y_1), ..., p_n(x_n, y_n)$
  - Output: The distance d between the closest pair of points
  - The brute-force algorithm:

```
procedure CP( $P[]$ )
    $d = \infty$
    for $(i = 1; i <= n - 1; i++)$ do
        for $(j = i + 1; j <= n; j++)$ do
            $d = min\left\{d, sqrt(x_i - x_j)^2 + (y_i - y_j)^2\right\}$
        end for
    end for
    return $d$
end procedure
```

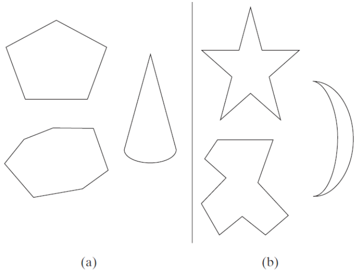
  - Complexity Analysis

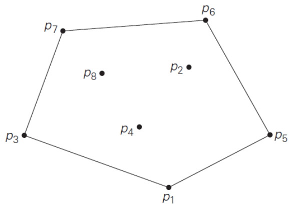
$$\begin{aligned} T(n) &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 1 \\ &= \sum_{i=1}^{n-1}(n-i) = \sum_{i=1}^{n-1} n - \sum_{i=1}^{n-1} i \\ &= n(n-1) - \frac{n(n-1)}{2} \\ &= \frac{n(n-1)}{2} \in \Theta(n^2) \end{aligned}$$

Convex-Hull by Brute Force

Convex-Hull Problem

- Convex: A set of points (finite or infinite) in the plane is called convex if for any two points p and q in the set, the entire line segment with the endpoints at p and q belongs to the set. (a) Convex sets. (b) not convex.
- The convex hull of a set S of points is the smallest convex set containing S. (The "smallest" requirement means that the convex hull of S must be a subset of any convex set containing S.)
- The convex-hull problem is the problem of constructing the convex hull for a given set S of n points.
- Example: The convex hull for the set of eight points is the convex polygon with vertices at p_1, p_5, p_6, p_7, p_3.

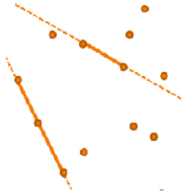




Convex-Hull by Brute Force (cont...)

- Applications of Convex Hull
  - computational geometry problems
  - approximation of object shapes
  - collision detection
  - computer animation applications
  - path planning
  - detecting outliers
  - solving optimization problems

- An extreme point of a convex set is a point of this set that is not a middle point of any line segment with endpoints in the set. For example, the extreme points of a triangle are its three vertices, the extreme points of a circle are all the points of its circumference



- The straight line through two points $(x_1, y_1), (x_2, y_2)$ in the coordinate plane can be defined by the equation

$$ax + bx - c = 0, \quad a = y_2 - y_1, \quad b = x_2 - x_1, \quad c = x_1 y_2 - y_1 x_2$$

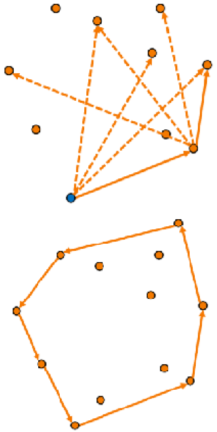
Convex-Hull by Brute Force (cont...)

A brute-force algorithm for the convex hull problem

- Input: A list P of n ($ne2$) points $p_1(x_1, y_1), ..., p_n(x_n, y_n)$
- Output: Set of points (Convex Hull)

```
 1: procedure CH(P[])
 2:     for each point $p_i$ do
 3:         for each point $p_j \neq p_i$ do
 4:             for each point $p_k \neq p_i \neq p_j$ do
 5:                 if $P_k$ is not (left or on) $(p_i, p_j)$
    then
 6:                     $(p_i, p_j)$ is not extreme
 7:                 else
 8:                     $(p_i, p_j)$ include in Convex Hull
 9:                 end if
10:             end for
11:         end for
12:     end for
13: end procedure
```

- Complexity $\Theta(n^3)$



- Better algorithms: Gift Wrapping, Quick Hull, Graham's Algorithm

Summary

1. Algorithm Design Techniques
   - Divide-and-Conquer
   - Decrease-and-Conquer
   - Transform-and-Conquer
   - Dynamic Programming Approach
   - Greedy Approach
   - State Space Search
2. Brute Force Algorithms
   - Largest zero-sum submatrix problem
   - Brute-Force String Matching
   - Closest-Pair by Brute Force
   - Convex-Hull by Brute Force