Design and Analysis of Algorithm

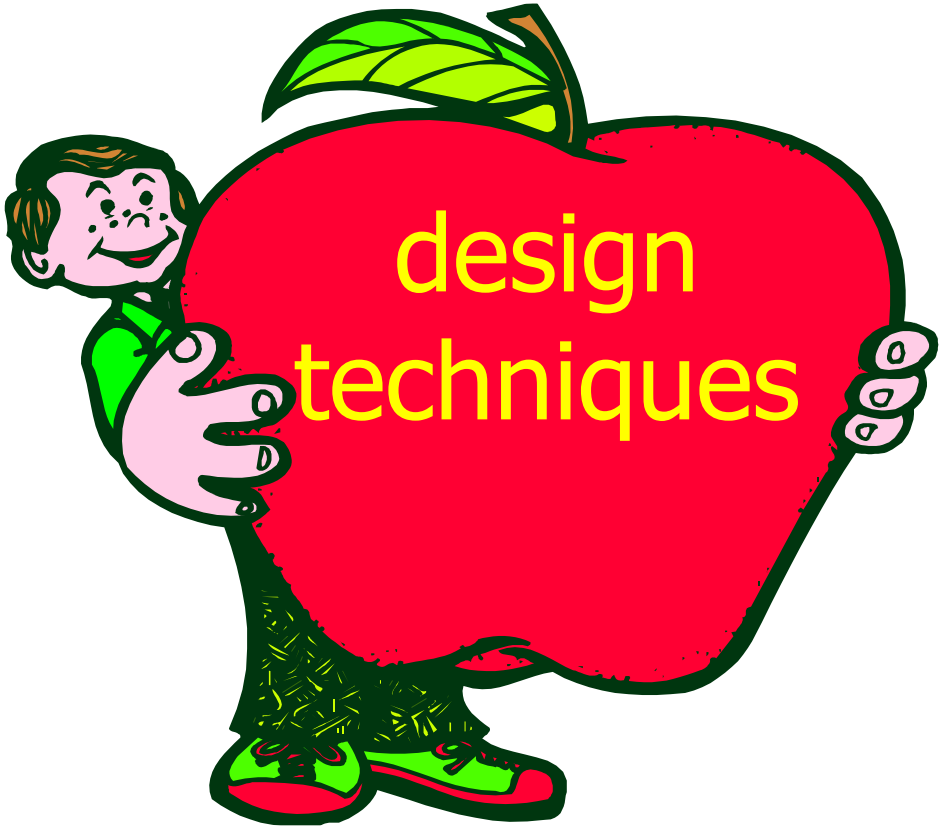
Department of Computer Science
COMSATS Institute of Information Technology, Islamabad

Tanveer Ahmed Siddiqui

Objectives

- To introduce the brute force mind set
  - How to Design Algorithm using Brute Force Approach
- To show a variety of brute-force solutions:
  - Examples of various well known Algorithms
  - Problem-1 Find the value of polynomial
  - Problem-2 Brute-force String Matching
  - Problem-3 Convert a 2D array into 1D array
  - Problem-4 Two closest points in a set of n points
  - **Problem-5 Application**
- To discuss the strengths and weaknesses of a brute force strategy

Designing an algorithm is easy if you know



What is algorithm designing technique?

- An algorithm design techniques (or "strategy" or "paradigm") is a general approach to solve problems algorithmically
- It can be applicable to a variety of problems from different area of computing.
- The design approach depends mainly on the model chosen.

Why do we need to know such techniques?

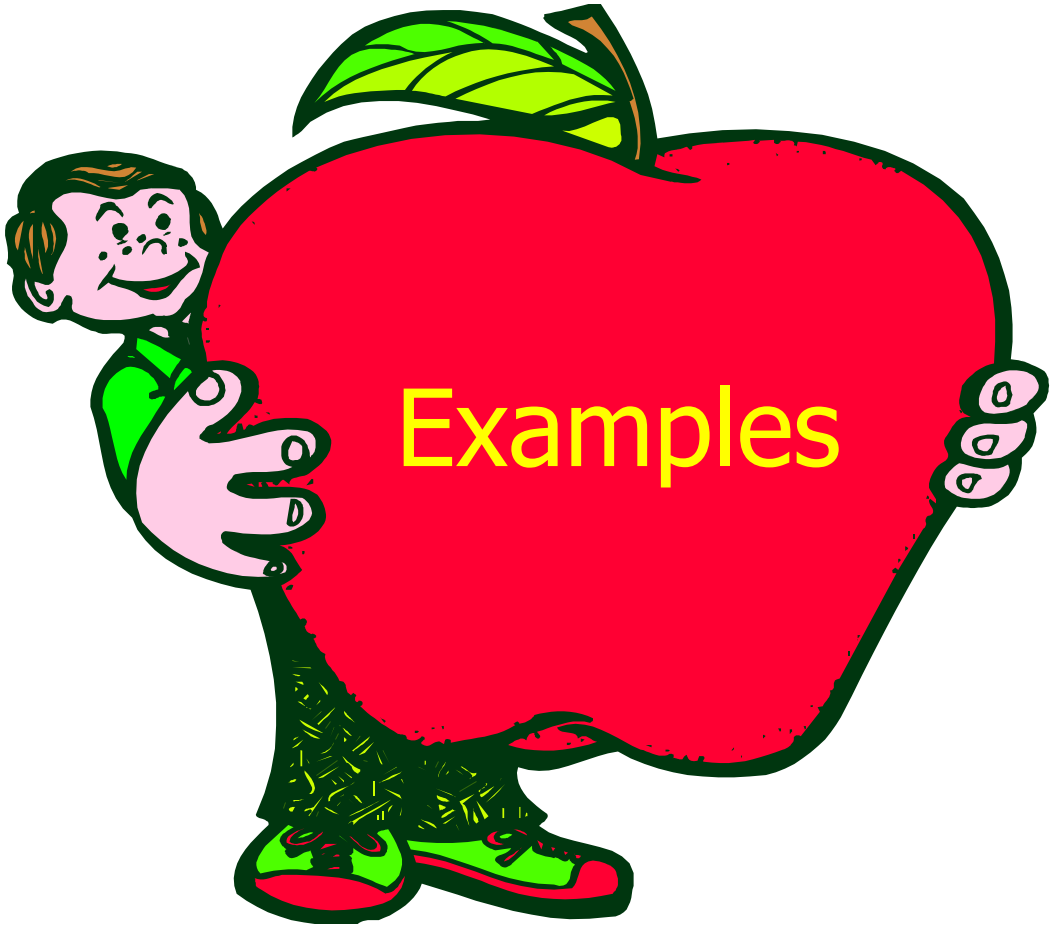
- They provide us **guidance in designing algorithms for new problems**
- They represent a **collection of tools** useful for applications

Which are the most used techniques?

- **Brute force**
- Decrease and conquer
- Divide and conquer
- Greedy technique
- Dynamic programming
- Backtracking

What is brute force?

- "Brute Force" means "Just do it!"
  - Force" comes from using computer power not intellectual power
- It is a straightforward approach to solve a problem in a very easy, yet (usually) inefficient way.
- To solve a problem using the brute force technique, you just need to use:
  - definitions of the concepts involved without making an effort to make the algorithm smart.
  - facts derived from the problem statement


Examples


Example of BF
Algorithm
with Constant
complexity

Exchange Problem

- **Problem: Exchange the values of two variables say x and y**

```
ALGORITHM Swap(x, y)
// Input: Two numbers x, and y
// Output: interchange the value of x and y
   temp ← x
   x  ← y
   y ← temp
```

- **Can you prove its correctness?**

1. Precondition: x = a and y = b
2. temp := x => temp = a
3. x : = y => x = b
4. y := temp => y = a
5. x = b and y = a is the Postcondition

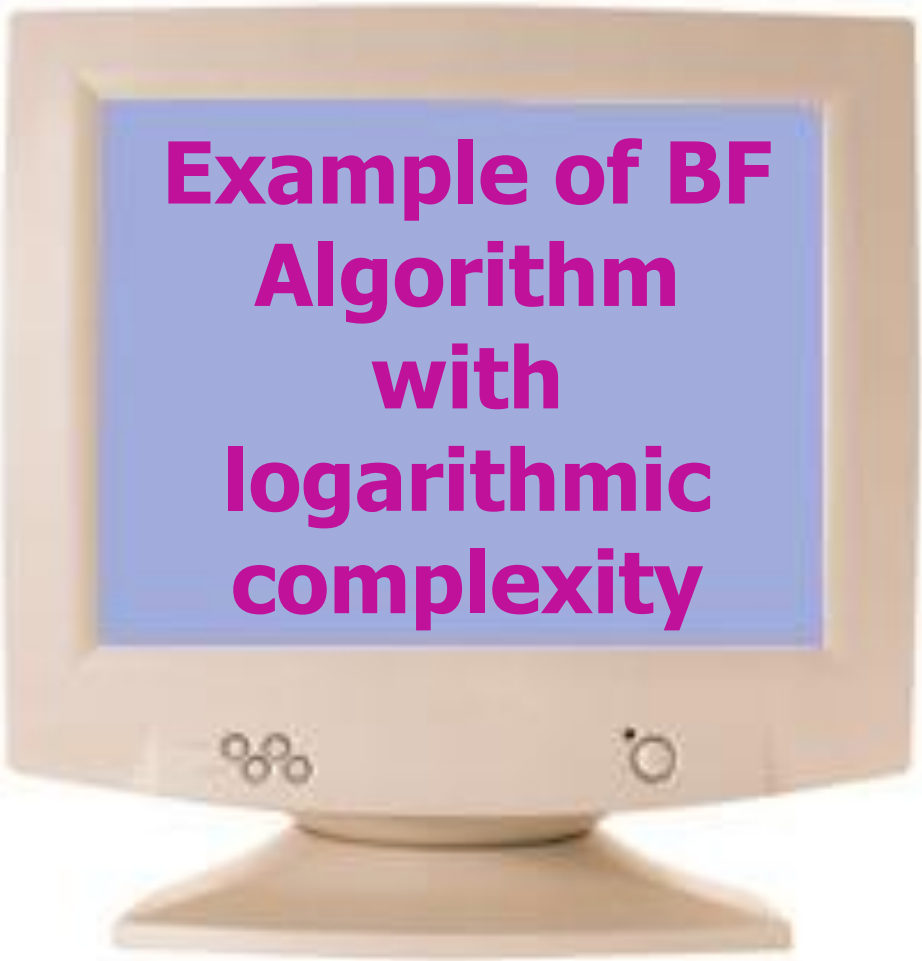
Exchange Problem

- **Problem: Exchange the values of two variables say x and y**

```
ALGORITHM Swap(x, y)
// Input: Two numbers x, and y
// Output: interchange the value of x and y
   temp ← x
   x  ← y
   y ← temp
```

- **What is its time complexity?**
  - **T(n) = $c_1+c_2+c_3$**
  - **= O(1)**

```
ALGORITHM Swap(x, y)
// Input: Two numbers x, and y
// Output: interchange the value of x and y
                          Cost        No. of Times executed
1.  temp ← x              c1                  1
2.  x  ← y                c2                  1
3.  y ← temp              c3                  1
```


Example of BF
Algorithm
with
logarithmic
complexity

- Problem: Design an algorithm for finding the binary representation of a positive decimal integer.

Representation

Algorithm *Binary*(n)
//The algorithm implements the standard method for finding
//the binary expansion of a positive decimal integer
//Input: A positive decimal integer n
//Output: The list $b_k\ b_{k-1}...b_1\ b_0$ of n's binary digits
$k \leftarrow 0$
while $n \neq 0$
    $b_k \leftarrow n \bmod 2$
    $n \leftarrow \lfloor n/2 \rfloor$
    $k \leftarrow k + 1$

```
ALGORITHM Binary(n)
// Output: # of digits in binary
// representation of n
count <- 1
while n > 1 do
        count <- count+1
        n <- ⌊n/2⌋
return count
```

Input size: n

Basic operation: $\lfloor - \rfloor$

C(n) = ?

Each iteration halves n

Let m be such that $2^m \le n < 2^{m+1}$

Take lg: $m \le \lg(n) < m+1$, so

$m = \lfloor \lg(n) \rfloor$ and $m+1 = \lfloor \lg(n) \rfloor + 1 \in \Theta(\lg(n))$

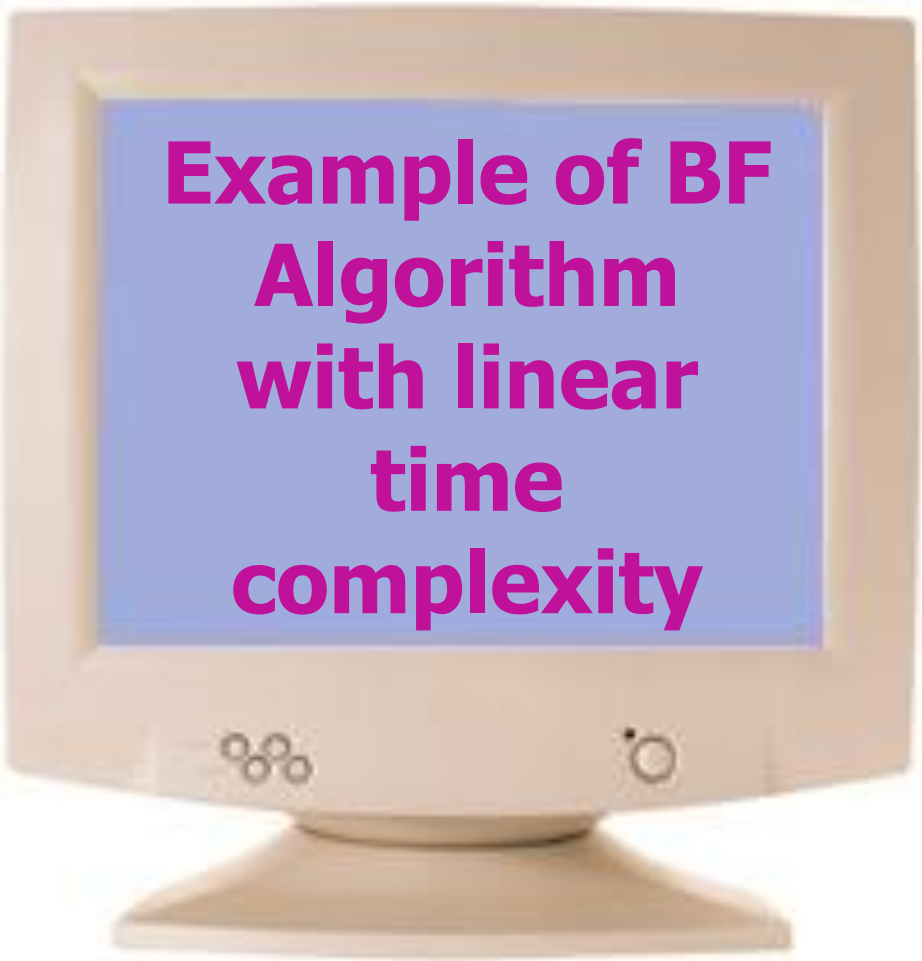
Some Examples

ALGORITHM *Binary*(n)

```
//Input: A positive decimal integer n
//Output: The number of binary digits in n's binary representation
count ← 1
while n > 1 do
    count ← count + 1
    n ← ⌊n/2⌋
return count
```

Basic Operation

The exact formula for the number of times the comparison $n>1$ will be executed is actually $\log_2 n + 1$


Example of BF Algorithm with linear time complexity

Some Examples

ALGORITHM power(x, n)
// Input: Two positive integers
// Output: x raised to the power n i.e. x^n.
answer ← 1 // initialize answer
i ← 1
while(i ≤ n) **do**
  answer ← answer * x
  i ← i + 1
return answer

Basic Operation

$$T(n) = \sum_{i=1}^{n} 1 = O(n)$$

ALGORITHM factorial(n)
// Input: A positive integer
// Output: The factorial of the positive integer n.
factorial ← 1 // initialize answer
i ← 1
while(i ≤ n) **do**
  factorial ← factorial * i
  i ← i + 1
return factorial

Basic Operation

$$T(n) = \sum_{i=1}^{n} 1 = O(n)$$

ALGORITHM multiplication(x , y)
// Input: Two positive integers
// Output: The product of the positive integers x and y.
answer ← 0 // initialize answer
i ← 0
while(i < x) **do**
  answer ← answer + y
  i ← i + 1
return answer

Basic Operation

$$T(n) = \sum_{i=0}^{x-1} 1 = O(x-1) = x$$

ALGORITHM *Binary(n)*
//Input: A positive decimal integer *n*
//Output: The number of binary digits in *n*'s binary representation
count ← 1
while *n* > 1 **do**
  count ← *count* + 1
  n ← ⌊*n*/2⌋

Basic Operation

The exact formula for the number of times the comparison *n*>1 will be executed is actually log2 *n* + 1

Some Examples

ALGORITHM *MaxElement*$(A[0..n-1])$

```
//Determines the value of the largest element in a given array
//Input: An array A[0..n − 1] of real numbers
//Output: The value of the largest element in A
maxval ← A[0]
for i ← 1 to n − 1 do
    if A[i] > maxval          Basic Operation
        maxval ← A[i]
return maxval
```

$$T(n) = \sum_{i=1}^{n-1} 1 = (n-1) = O(n)$$

Some Examples

ALGORITHM *SequentialSearch*$(A[0..n-1], K)$

```
//Searches for a given value in a given array by sequential search
//Input: An array A[0..n − 1] and a search key K
//Output: The index of the first element in A that matches K
//          or −1 if there are no matching elements
i ← 0
while i < n and A[i] ≠ K do        Basic Operation
    i ← i + 1
if i < n return i
else return −1
```

$$T(n) = \sum_{i=1}^{n} 2 = 2n = O(n)$$

$$C_{worst}(n) = n$$

$$C_{best}(n) = 1$$

ALGORITHM *SequentialSearch2*$(A[0..n], K)$

```
//Implements sequential search with a search key as a sentinel
//Input: An array A of n elements and a search key K
//Output: The index of the first element in A[0..n − 1] whose value is
//          equal to K or −1 if no such element is found
A[n] ← K
i ← 0
while A[i] ≠ K do Basic Operation
        i ← i + 1
if i < n return i
else return −1
```

$$T(n) = \sum_{i=1}^{n} 1 = n = O(n)$$

Some Examples

neither the worst-case analysis nor its best-case counterpart yields the necessary information about an algorithm's behavior on a "typical" or "random" input.This is the information that the ***average-case efficiency*** seeks to provide.

To analyze the algorithm's average case efficiency, we must make some assumptions about possible inputs of size n.

The standard assumptions are that

(a) the probability of a successful search is equal to p ($0 \leq p \leq 1$) and

(b) The probability of the first match occurring in the ith position of the list is the same for every i.

ALGORITHM *SequentialSearch*$(A[0..n-1], K)$

```
//Searches for a given value in a given array by sequential search
//Input: An array A[0..n − 1] and a search key K
//Output: The index of the first element in A that matches K
//        or −1 if there are no matching elements
i ← 0
while i < n and A[i] ≠ K do
    i ← i + 1
if i < n return i
else return −1
```

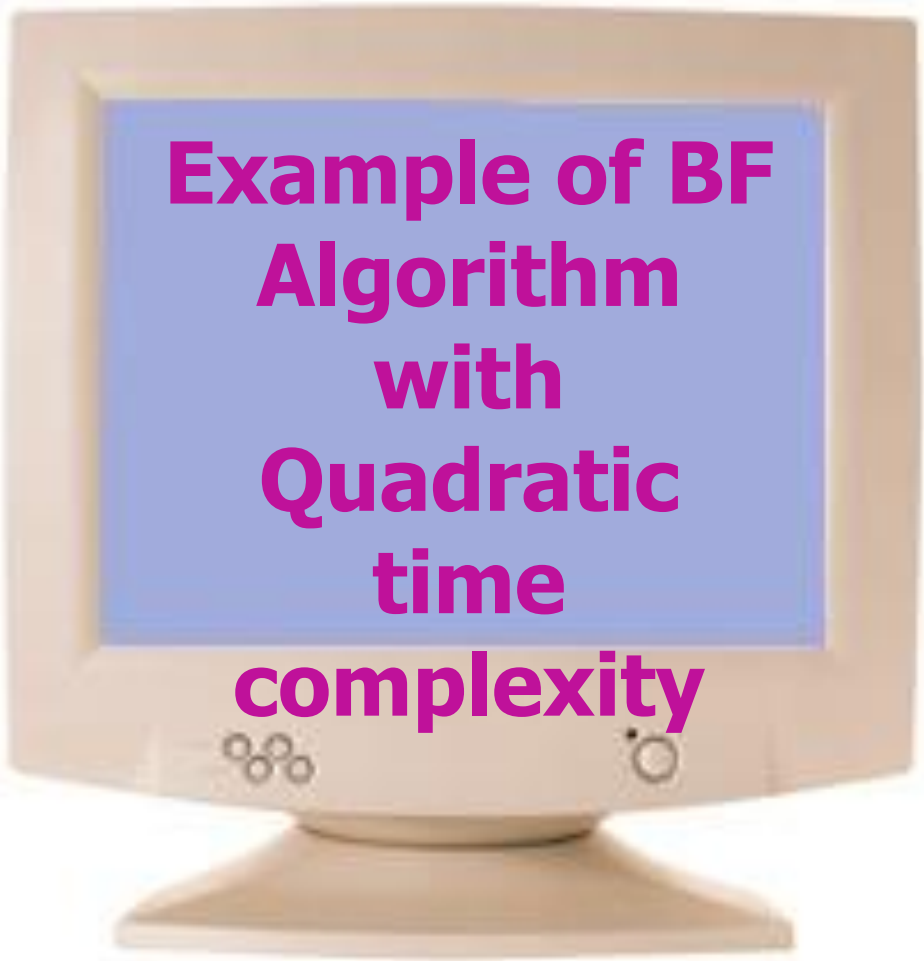
Some Examples

we can find the average number of key comparisons *Cavg(n)* as follows.
In the case of a successful search, the probability of the first match occurring in the *i*th position of the list is *p/n* for every *i*, and the number of comparisons made by the algorithm in such a situation is obviously *i*.
In the case of an unsuccessful search, the number of comparisons will be *n* with the probability of such a search being *(1– p)*. Therefore,

$$C_{avg}(n) = [1 \cdot \frac{p}{n} + 2 \cdot \frac{p}{n} + \cdots + i \cdot \frac{p}{n} + \cdots + n \cdot \frac{p}{n}] + n \cdot (1 - p)$$

$$= \frac{p}{n}[1 + 2 + \cdots + i + \cdots + n] + n(1 - p)$$

$$= \frac{p}{n}\frac{n(n+1)}{2} + n(1 - p) = \frac{p(n+1)}{2} + n(1 - p).$$


Example of BF
Algorithm
with
Quadratic
time
complexity

Some Examples

ALGORITHM *UniqueElements*$(A[0..n-1])$

```
//Determines whether all the elements in a given array are distinct
//Input: An array A[0..n − 1]
//Output: Returns "true" if all the elements in A are distinct
//                and "false" otherwise
for i ← 0 to n − 2 do
    for j ← i + 1 to n − 1 do
        if A[i] = A[j] return false
return true
```

Basic Operation

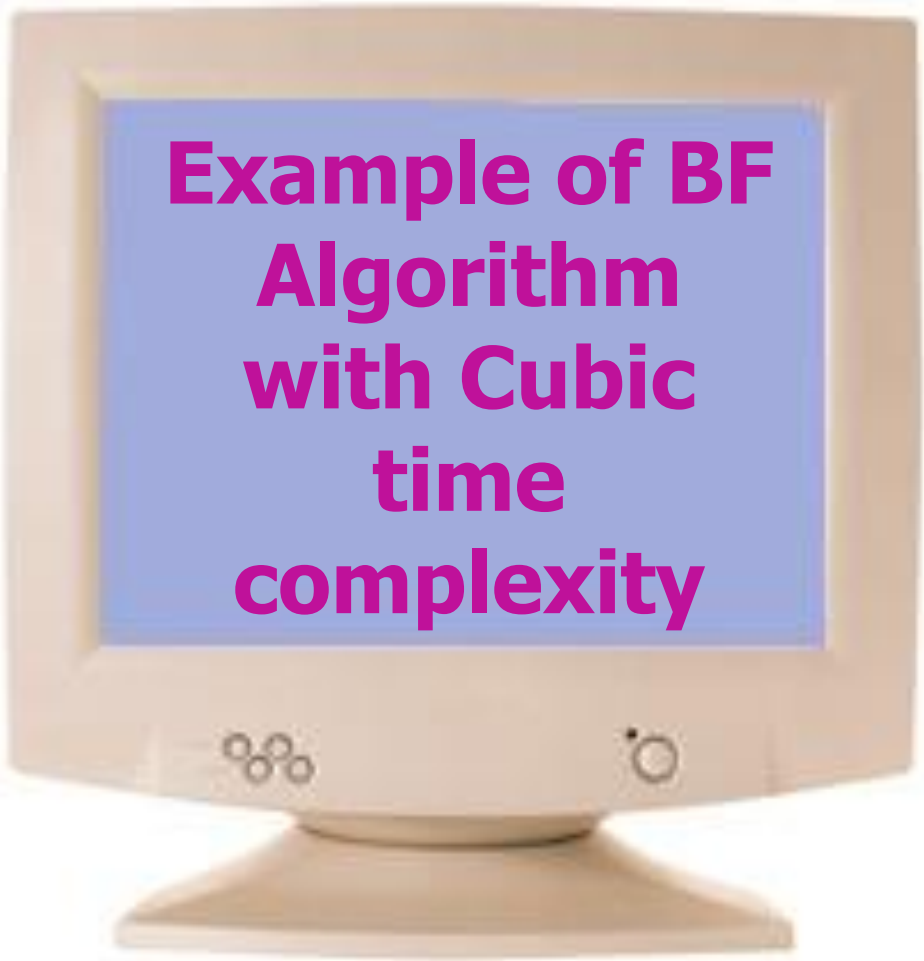
$$C_{worst}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$$
$$= \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1]$$
$$= \sum_{i=0}^{n-2} (n-1-i)$$
$$= (n-1) + (n-2) + \cdots + 1 = \frac{(n-1)n}{2}$$
$$= \Theta(n^2)$$

ALGORITHM *MinDistance*$(A[0..n-1])$

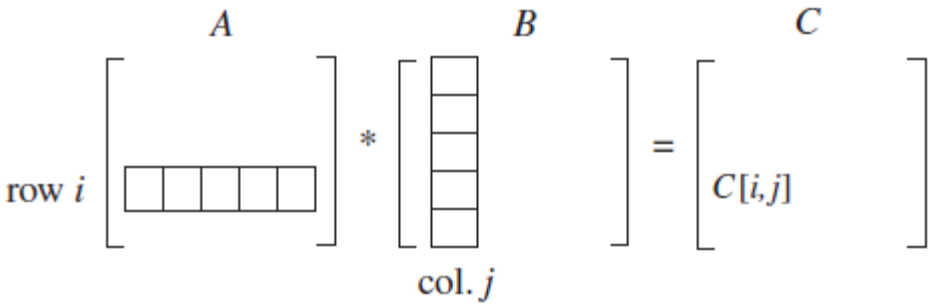
```
//Input: Array A[0..n − 1] of numbers
//Output: Minimum distance between two of its elements
dmin ← ∞
    for i ← 0 to n − 1 do
        for j ← 0 to n − 1 do
            if i ≠ j and |A[i] − A[j]| < dmin
            dmin ← |A[i] − A[j]|
return dmin
```

Basic Operation

$$C_{worst}(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} 1$$
$$= \sum_{i=0}^{n-1} [(n-1)$$
$$= (n-1)^2$$
$$= \Theta(n^2)$$


Example of BF Algorithm with Cubic time complexity

Some Examples



ALGORITHM *MatrixMultiplication*$(A[0..n-1, 0..n-1], B[0..n-1, 0..n-1])$

```
//Multiplies two square matrices of order n by the definition-based algorithm
//Input: Two n × n matrices A and B
//Output: Matrix C = AB
for i ← 0 to n − 1 do
    for j ← 0 to n − 1 do
        C[i, j] ← 0.0
        for k ← 0 to n − 1 do
            C[i, j] ← C[i, j] + A[i, k] ∗ B[k, j]
return C
```

Basic Operation

$$M(n) = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1}\sum_{k=0}^{n-1} 1 = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} n = \sum_{i=0}^{n-1} n^2 = n^3.$$


Class
Exercise

Problem: Find the value of polynomial

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

at a point $x = x_0$

Problem: Find the value of polynomial

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

at a point $x = x_0$

Brute-force algorithm

```
p ← 0.0
for i ← n downto 0 do
    power ← 1
        for j ← 1 to i do          //compute x^i
            power ← power * x
        p ← p + a[i] * power

return p
```

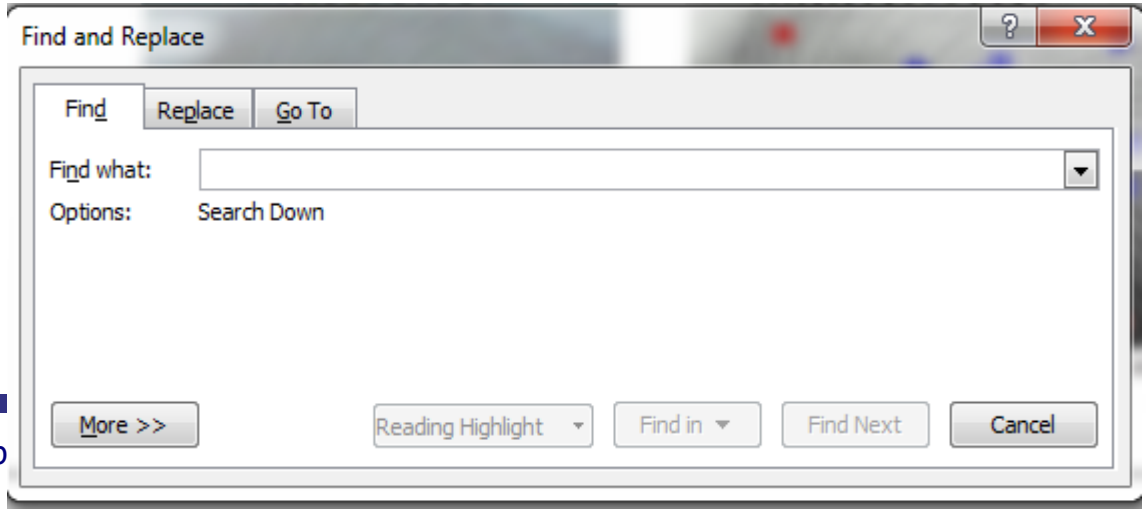
We can do better by evaluating from right to left:

Better brute-force algorithm

```
p ← a[0]
power ← 1
for i ← 1 to n do
        power ← power * x
        p ← p + a[i] * power
 return p
```

Example 2: Searching Sub String in Text

- A string is a sequence of characters from an alphabet.
- Text strings: letters, numbers, and special characters.
- String matching: searching for a given word/pattern in a text.
  - Find/Replace Module in word processor



Pattern Matching Problem

- **Pattern:** a string of m characters to search for
- **Text:** a (longer) string of n characters to search in
- **Problem:** Find a substring in the text that matches the pattern
- Examples
  - Pattern: happy
  - Text: It is never too late to have a happy childhood
  - Pattern: 001011
  - Text: 10010101101001100101111010

Pattern Matching Problem

Brute-force algorithm

Step 1 Align pattern at beginning of text

Step 2 Moving from left to right, compare each character of pattern to the corresponding character in text until

- all characters are found to match (successful search); or
- a mismatch is detected

Step 3 While pattern is not found and the text is not yet exhausted, realign pattern one position to the right and repeat Step 2

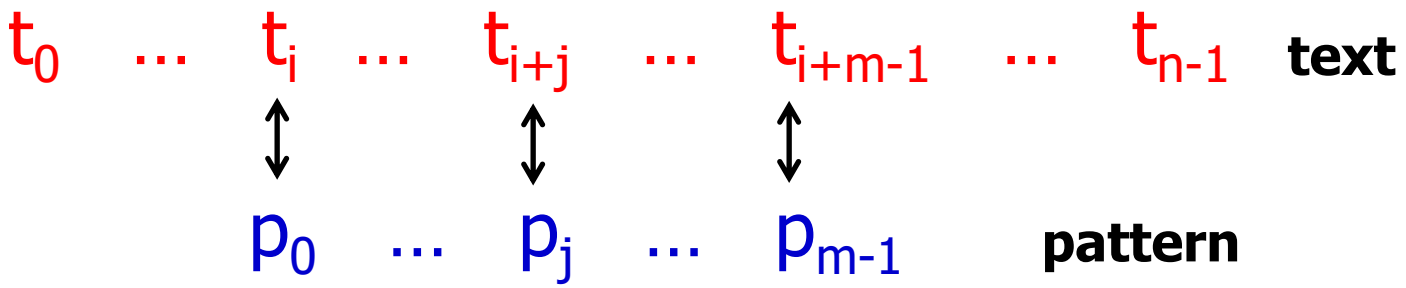
Pattern Matching Problem

Input size: n, m

We shall learn more sophisticated and efficient ones in space-time trade-off chapter!

Pattern Matching Problem

- Given a string of n characters (text) and a string of m (≤ n) characters (pattern), find a substring of the text that matches the pattern
- Text: "nobody noticed him" pattern: "not"



p_0 should be tested with up to $t_?$

Brute-force String Matching

ALGORITHM *BruteForceStringMatch*$(T[0..n-1], P[0..m-1])$

```
//Implements brute-force string matching
//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 or −1 if the search is unsuccessful
for i ← 0 to n − m do
    j ← 0
    while j < m and P[j] = T[i + j] do
        j ← j + 1
    if j = m return i
return −1
```

Brute-force String Matching

```
ALGORITHM BruteForceStringMatching(T[0..n-1], P[0..m-1])
for i <- 0 to n-m do
        j <- 0
        while j < m and P[j] = T[i+j] do
                j <- j+1
        if j = m
                return i
return -1
```

Input size: n, m
Basic op: =

$C_{worst}(n, m) = m(n-m+1) \in O(nm)$

$C_{avg}(n, m) \in \Theta(n)$

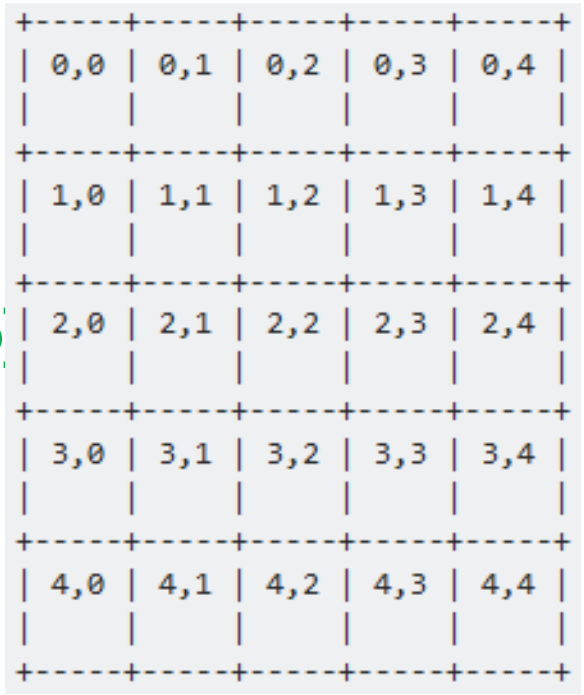
N O B O D Y _ N O T I C E D _ H I M

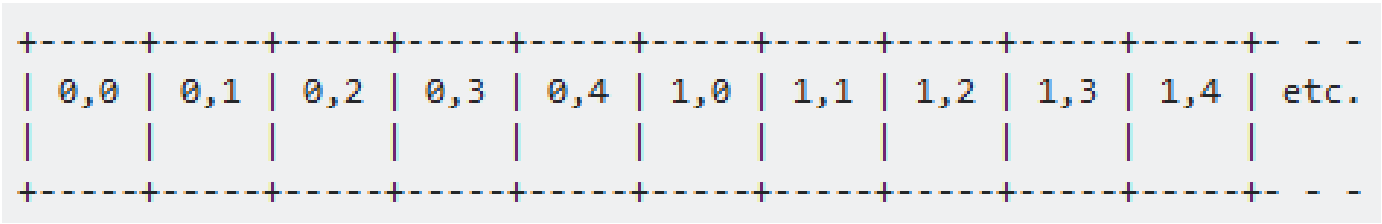
N O B O D Y _ N O T I C E D _ H I M

We shall learn more sophisticated and efficient ones in space-time trade-off chapter!

Example-3

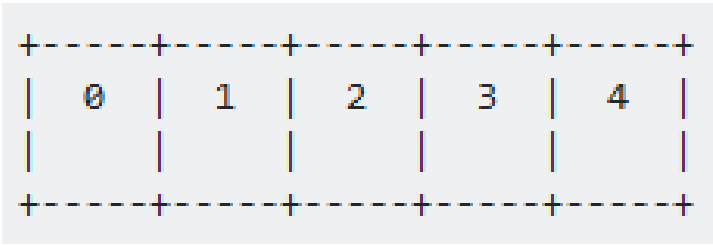
- Problem: Design an algorithm convert a 2D array into 1D array
- Solution:
- **What is input**
  - An 2D-array A[i][j]. e. g. int [5][5
- **What should be the output**
  - You could picture the conversion
  - to the corresponding 1-D array
  - like this:



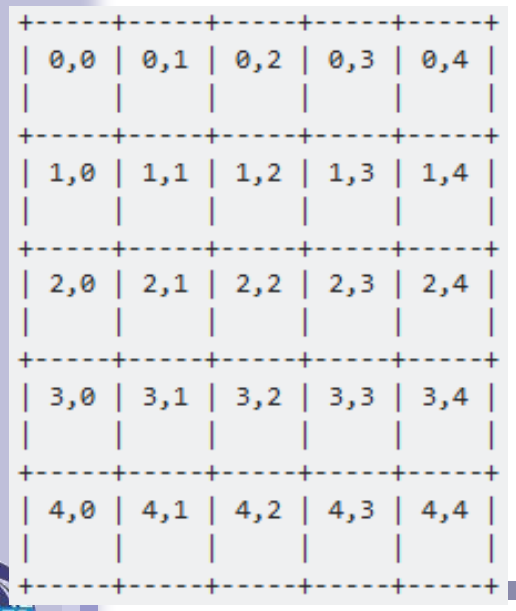


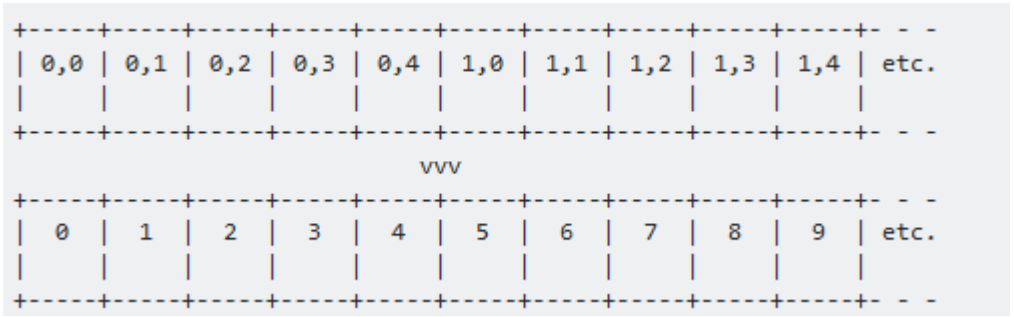
Example-3

- **Which designing technique?**
  - Brute-Force
- **Logic/Idea**
  - A 1-D array looks like this:int [5] :



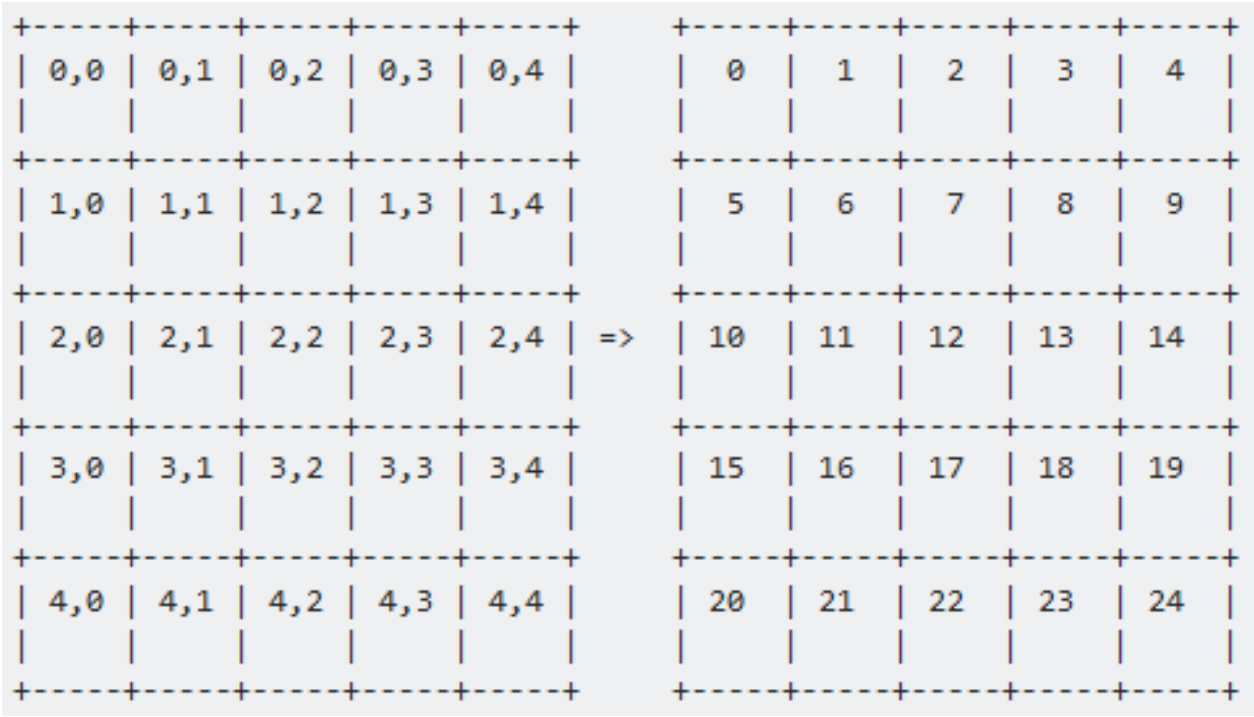
  - You could picture the conversion to the corresponding 1-D array like this





Example-3

- **Logic/Idea**
  - But an alternative way of thinking about it is to picture the original array, but re-labelled - like this



2-D array index [i][j] => 1-D array index [i*5 + j]

Example-3

Example 4: Closest- Pair Problem

- **Problem:** Find the two closest points in a set of n points
- Points can be:
  - airplanes (most probable collision candidates),
  - database records
  - DNA sequences, etc.
- Cluster analysis: pick two points, if they are close enough they are in the same cluster, pick another point, and

  so on…

Example 4: Closest- Pair Problem

Find the two closest points in a set of *n* points (in the two-dimensional Cartesian plane).

Brute-force algorithm

Compute the distance between every pair of distinct points and return the indexes of the points for which the distance is the smallest.

Example 4: Closest- Pair Problem

- For simplicity we consider 2-D case
- Euclidean distance, $d(p_i, p_j) =$

$$\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

- Brute-force: compute distance between each pair of disjoint points and find a pair with the smallest distance
- $d(p_i, p_j) = d(p_j, p_i)$, so we consider only

Example 4: Closest- Pair Problem

ALGORITHM *BruteForceClosestPoints*(P)

//Input: A list P of n $(n \geq 2)$ points $P_1 = (x_1, y_1), \ldots, P_n = (x_n, y_n)$
//Output: Indices *index*1 and *index*2 of the closest pair of points
$dmin \leftarrow \infty$
for $i \leftarrow 1$ **to** $n - 1$ **do**
    for $j \leftarrow i + 1$ **to** n **do**
        $d \leftarrow sqrt((x_i - x_j)^2 + (y_i - y_j)^2)$ //*sqrt* is the square root function
        if $d < dmin$
            $dmin \leftarrow d$; $index1 \leftarrow i$; $index2 \leftarrow j$
return *index*1, *index*2

YOUR TURN

Counting Similarity

- Consider the following three different arrays. Which one is most similar to

3	9	13	21	95

- What is the measure for **counting similarity**?

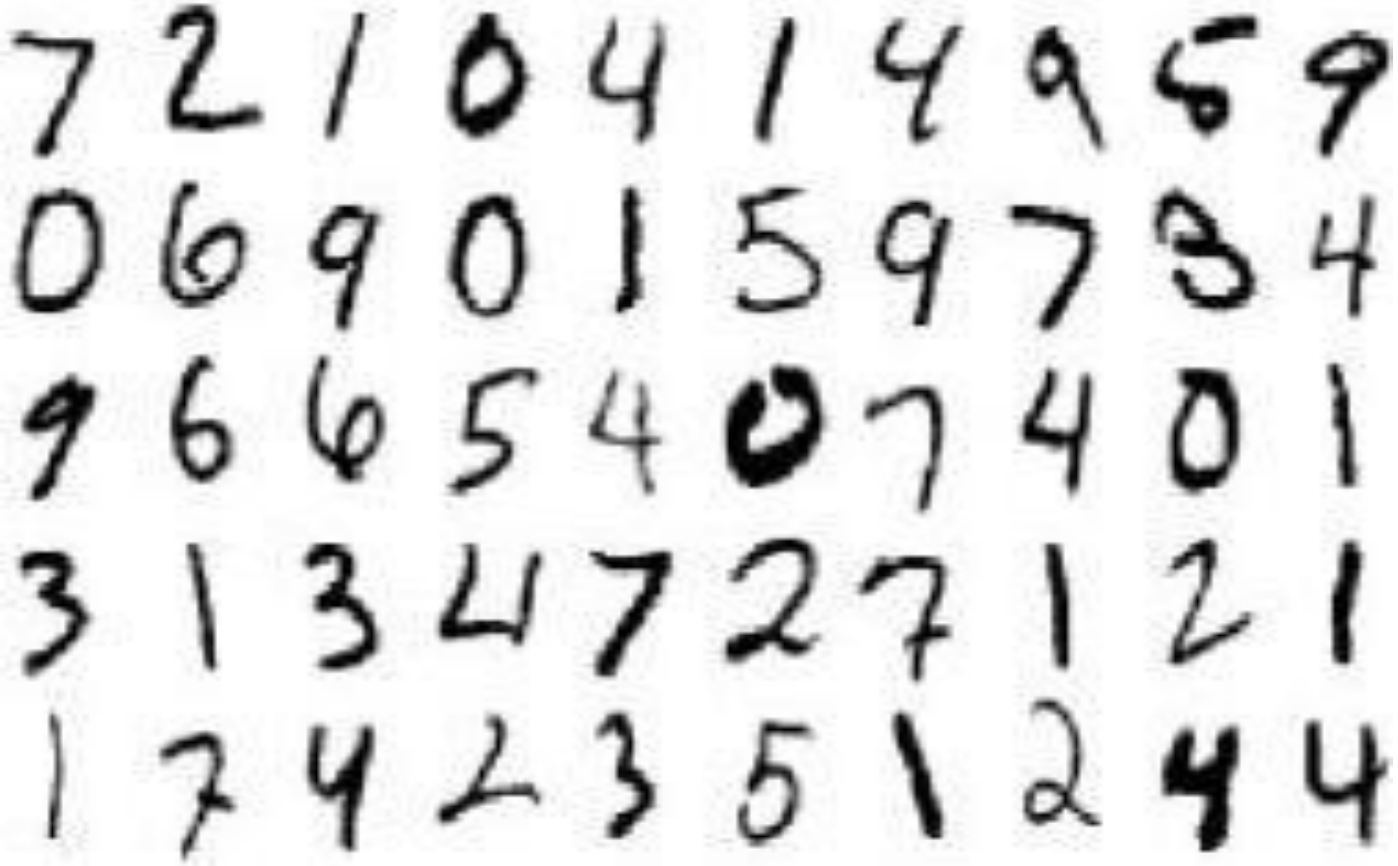
9	3	13	95	21

3	9	21	13	95

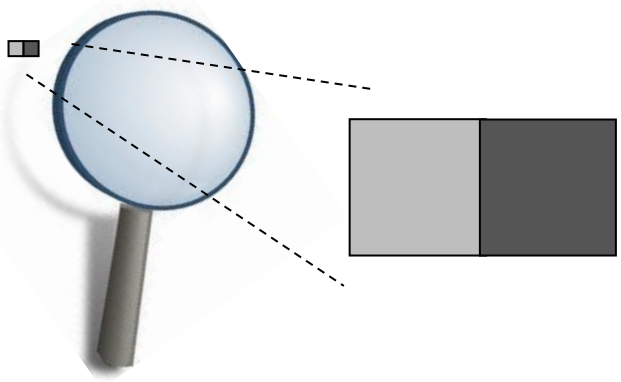
95	21	13	9	3

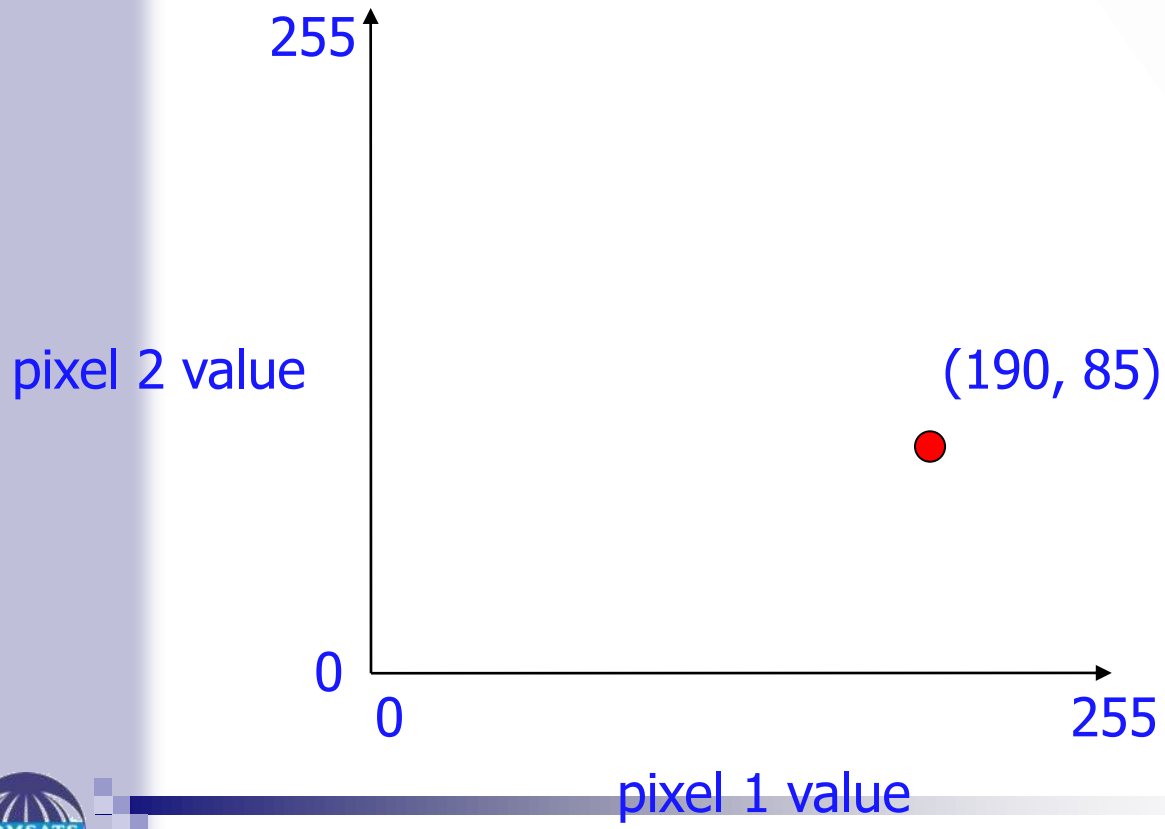
21	95	3	9	13

Now, how is this problem like handwriting recognition?

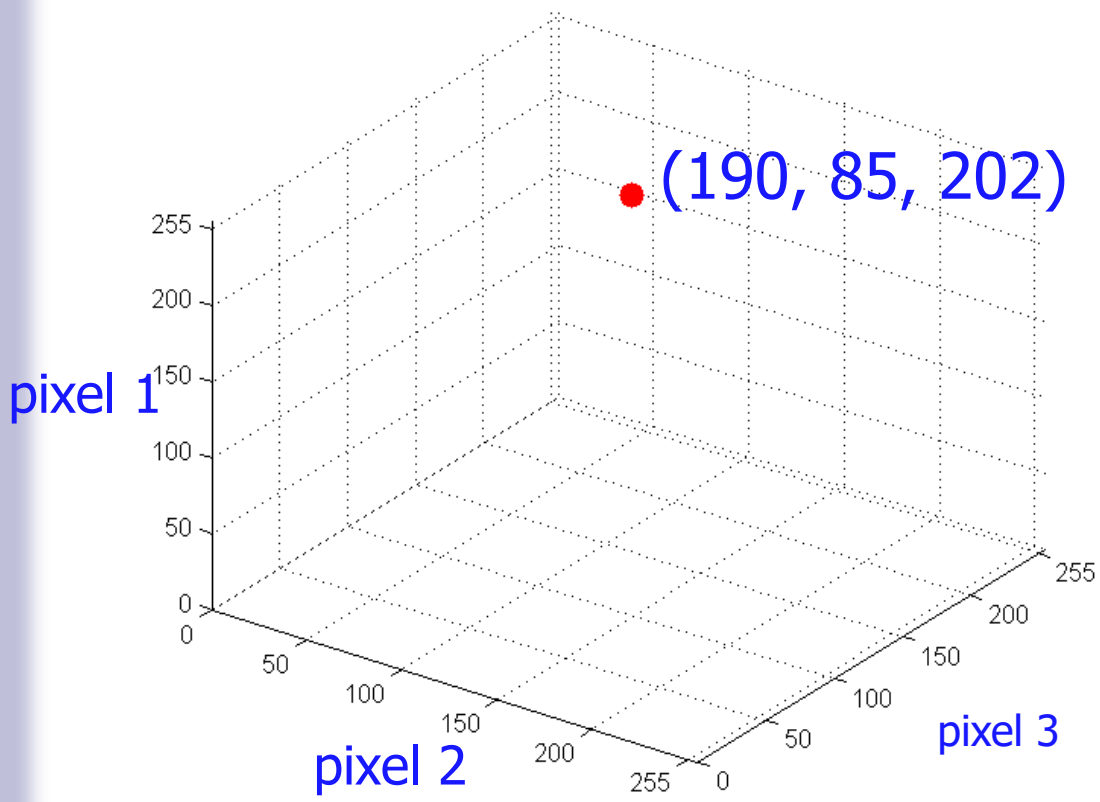


A two-pixel image

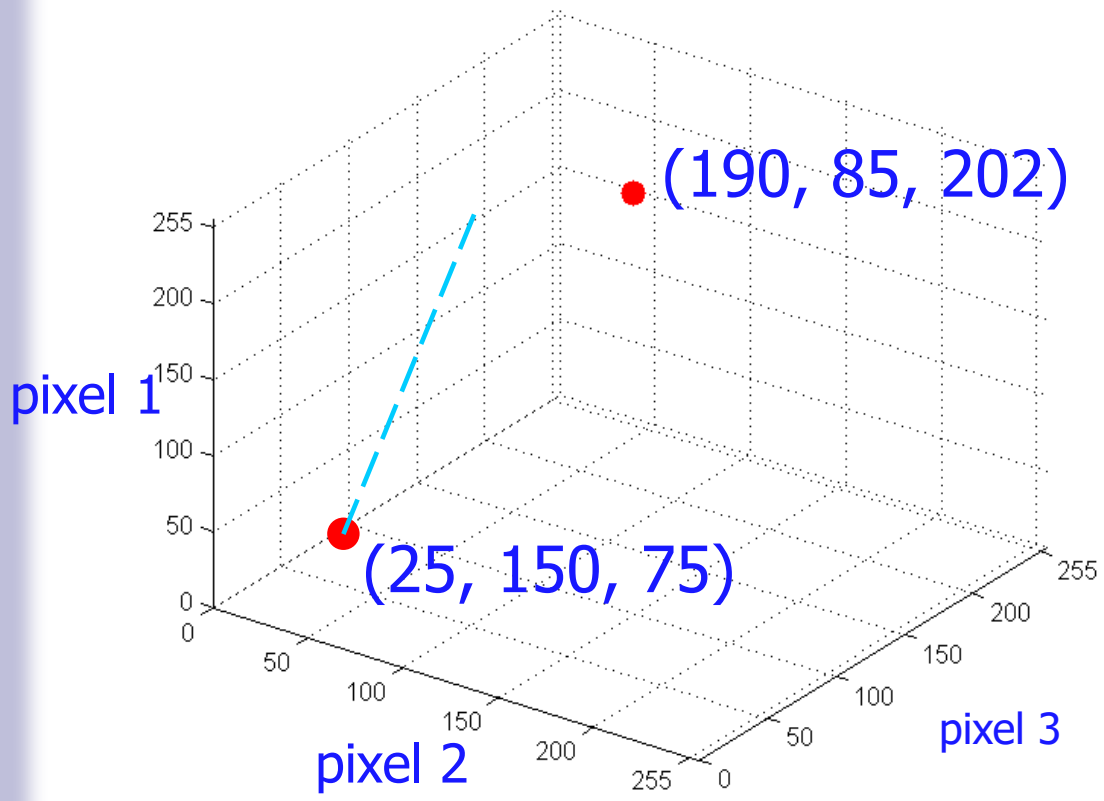




A three-pixel image



This 3-pixel image is represented by a SINGLE point in a 3-D space.



A three-pixel image

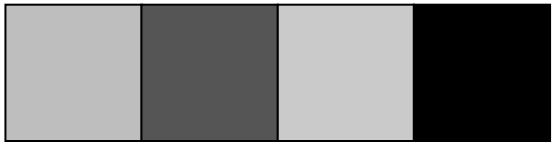
Another 3-pixel image

Straight line distance between them?

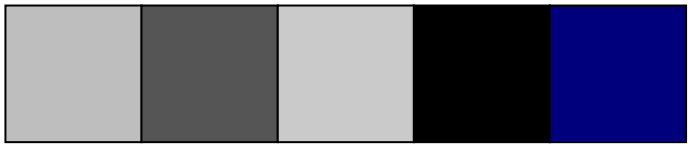
4-dimensional space? 5-d? 6-d?

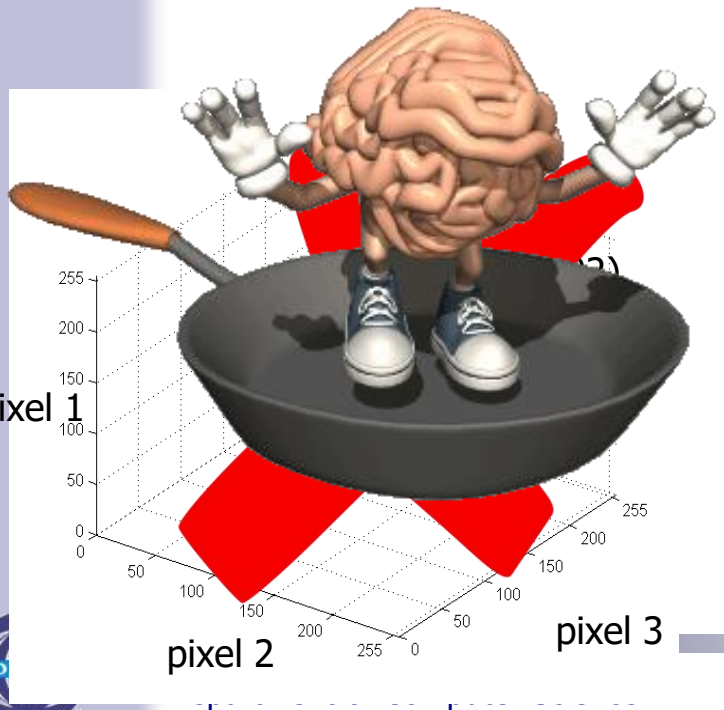
A three-pixel image

A four-pixel image.

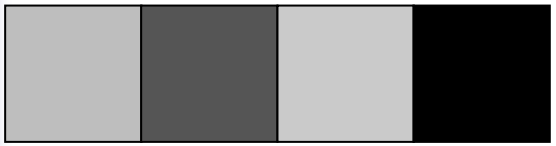


A five-pixel image



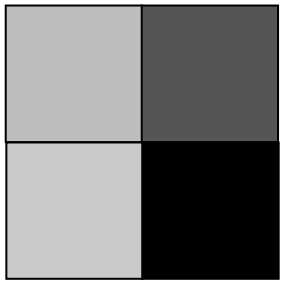


A four-pixel image.

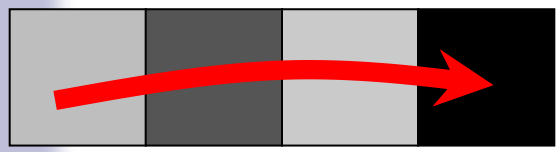


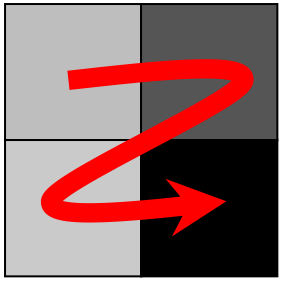
(190, 85, 202, 10)

A different four-pixel image.



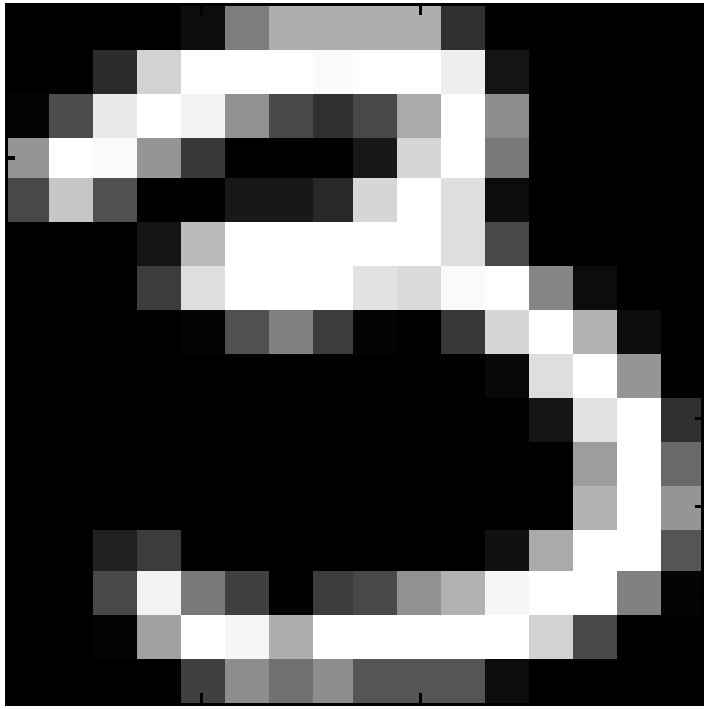
(190, 85, 202, 10)
Same 4-dimensional vector!



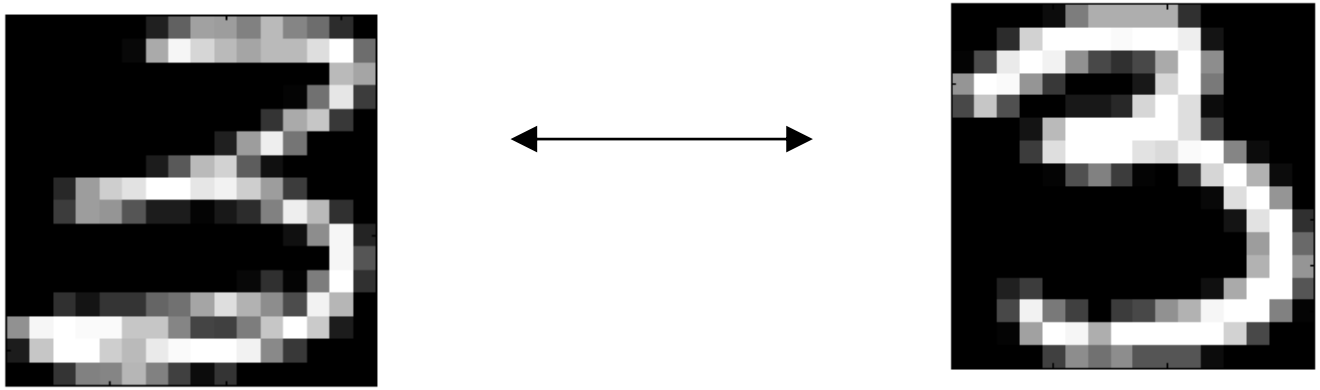


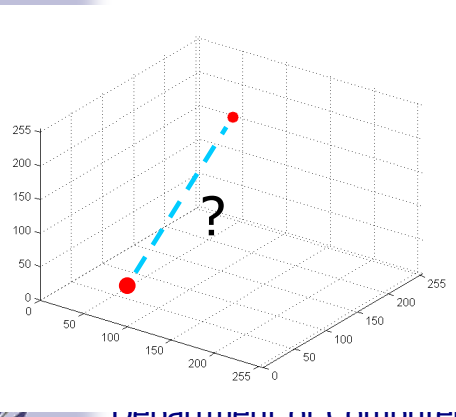
Assuming we read pixels in a systematic manner, we can now represent any image as a single point in a high dimensional space.

16 x 16 pixel image. How many dimensions?

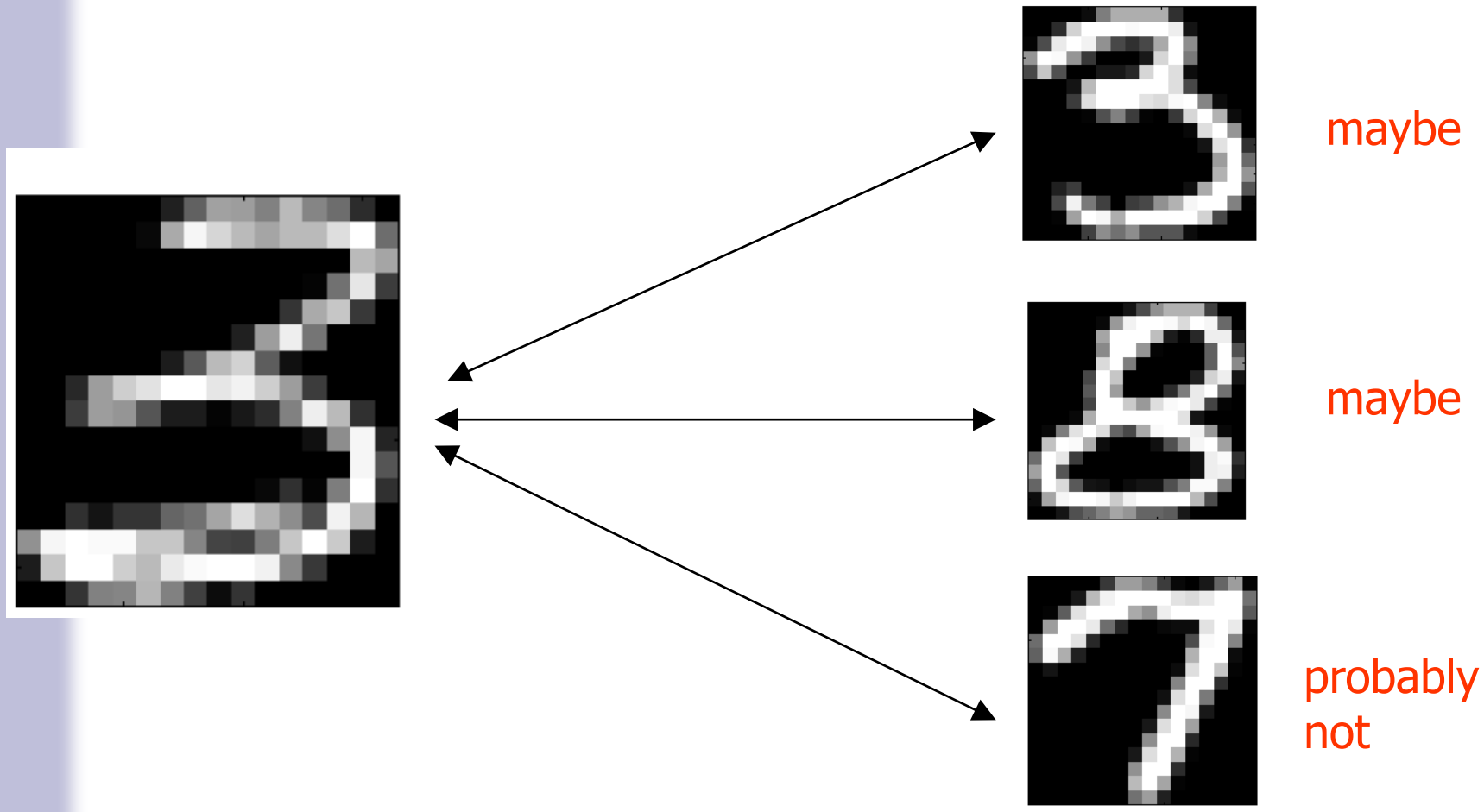


We can measure distance in 256 dimensional space.





$$\text{distance}(x, x') = \sqrt{\sum_{i=1}^{i=256} (x_i - x'_i)^2}$$


maybe
maybe
probably
not

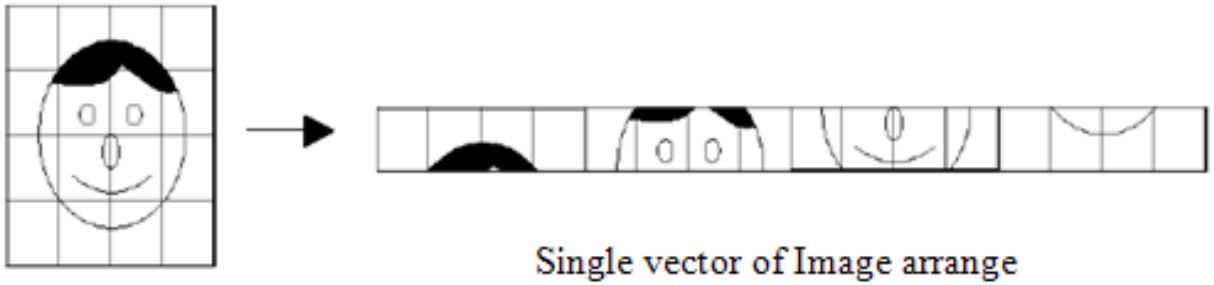
Identity of face image/Face Recognition

- **Problem:** Design an algorithm find the identity of a given face image according to their memory. The memory of a face recognizer is generally simulated by a training set. Your training set consists of the features extracted from known face images of different persons.
- **What is input?**

Known face images **unknown image**

Identity of face image/Face Recognition

- Thus, the task of the face recognizer is to find the most similar feature vector among the training set to the feature vector of a given test image.
- **Idea(Brute Force):** Convert of facial image (the pixel values) into single vector per image. Following figure illustrate this.

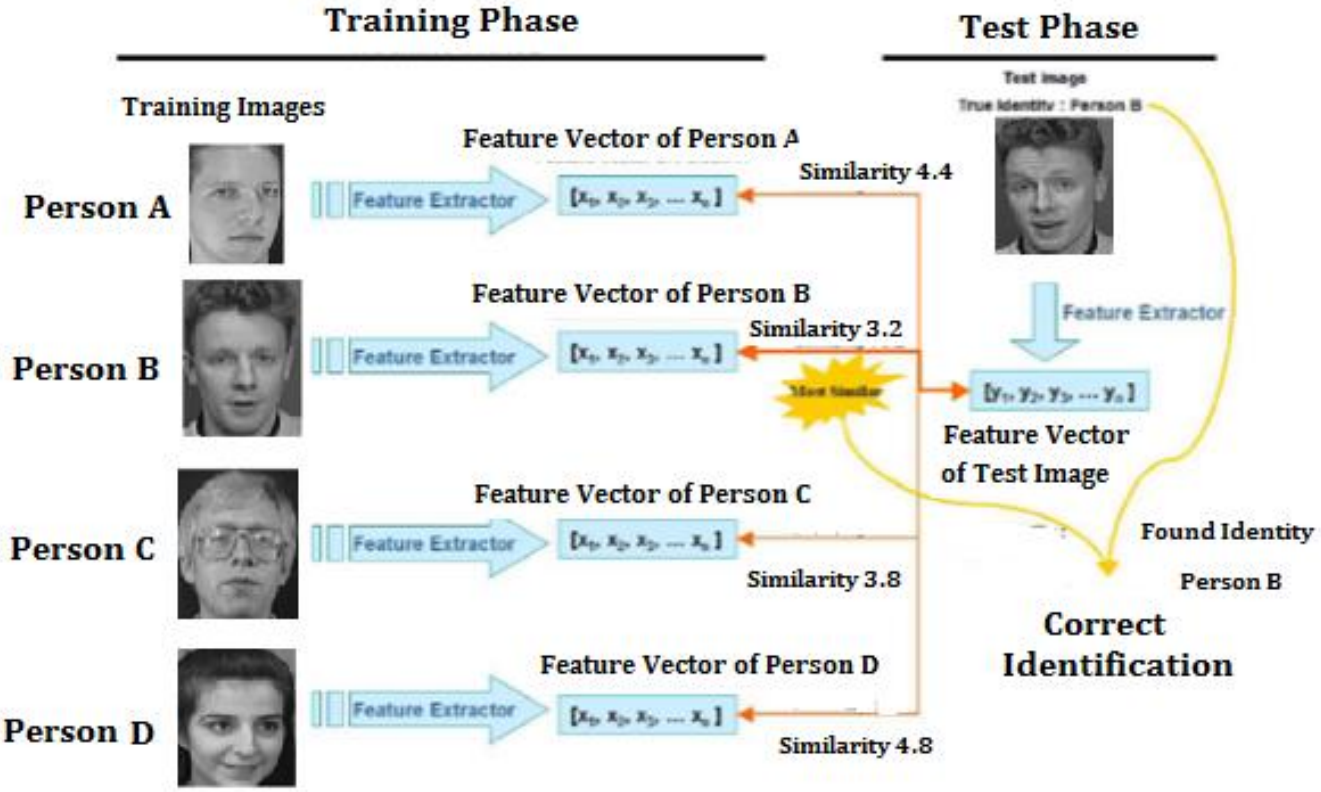


Original Image

Single vector of Image arrange in lexicographical order

- Find the most similar feature vector among the training set to the feature vector of a given test image using Euclidean Distance formula.

Identity of face image/Face Recognition



Generic Face recognition recognition system

CONCLUSION

Conclusion

- **Strengths**
  - wide applicability
  - simplicity
  - yields reasonable algorithms for some important problems
    (e.g., matrix multiplication, sorting, searching, string matching)

- **Weaknesses**
  - rarely yields efficient algorithms
  - some brute-force algorithms are unacceptably slow
  - not as constructive as some other design techniques

- Design of Cruder versions providing a ladder