CPE231RC — Algorithms — 2/2566

COURSE ORIENTATION
RECAP: ANALYSIS OF ALGORITHM EFFICIENCY

Dr. Prapong Prechaprapranwong

Computer Engineering

King Mongkut's University of Technology Thonburi

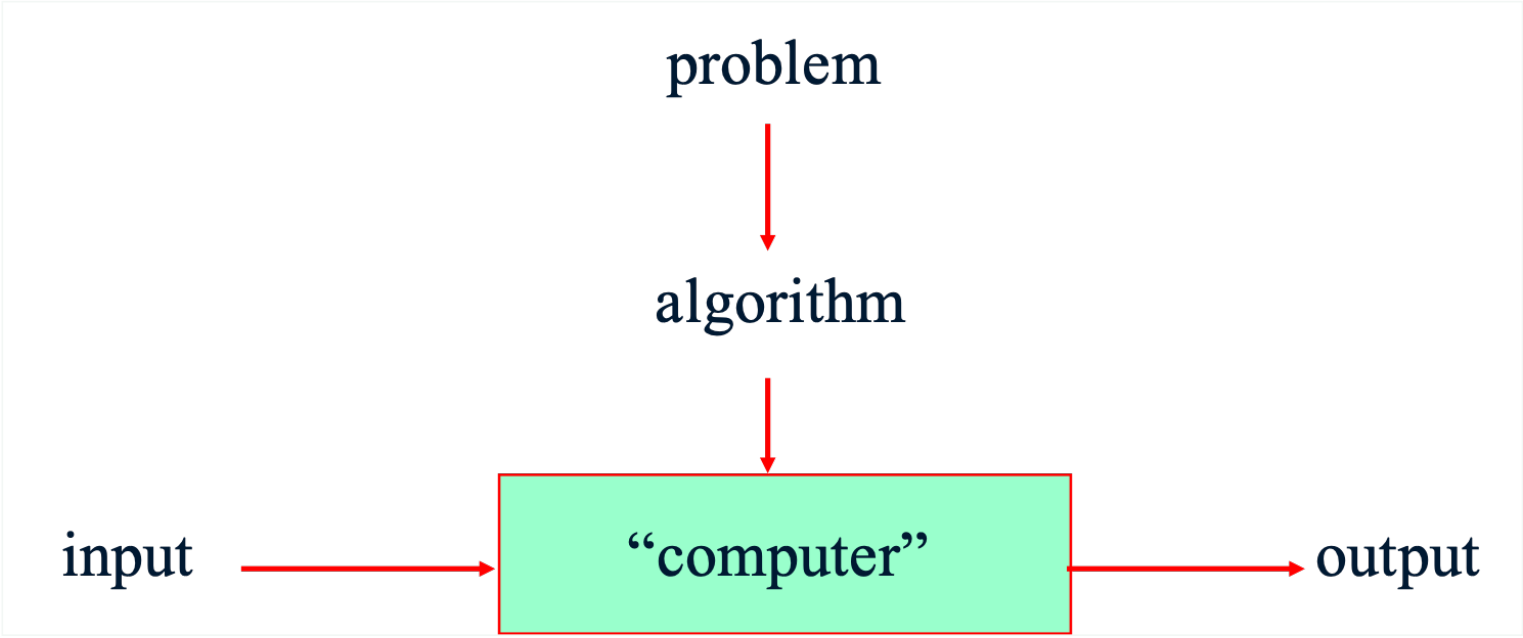
COURSE LEARNING OUTCOMES

- Apply design principles and concepts to algorithm design
- Implement algorithms in a programming language
- Analyze efficiency and correctness of algorithms
- Apply appropriate algorithms to solve problems

WHAT IS AN ALGORITHM

A sequence of unambiguous instructions for solving a well-defined problem,

i.e., obtaining a required output for any legitimate input, in finite time.



PROBLEM VS. INSTANCE OF PROBLEM

- Sorting problem
  - **Input:** A sequence of n orderable keys a_1, a_2, …, a_n
  - **Output:** The permutation (reordering) of the input sequence

- ***Instance*** of sorting problem
  - **Input:** Array of names {Mike, Bob, Sally, Jill, Jan}, or a list of numbers like {154, 245, 568, 324, 654, 324}
  - **Output:** The sorted array or list above.

EXPRESSING ALGORITHM

1.NATURAL LANGUAGE

Algorithm MaxElement

1: Input: A sequence of numbers $a_1, a_2, \cdots, a_n$
2: Output: The largest element in the input sequence
3:
4: **Step 1:** Set the first element as the current maximum
5: **Step 2:** For the rest of the elements, compare the current maximum to the each element. If the element is greater than the current maximum, update the current maximum to the current element.
6: **Step 3:** Take the current maximum as the algorithm output.

EXPRESSING ALGORITHM

2.HIGH-LEVEL, UNAMBIGUOUS LANGUAGE
MIXES OF NATURAL LANGUAGE AND PROGRAMMING LANGUAGE

Algorithm MaxElement

1: Input: A sequence of numbers $a_1, a_2, \cdots, a_n$
2: Output: The largest element in the input sequence
3:
4: $maxval \leftarrow a_1$
5: **for** $i = 2$ to n **do**
6: **if** $maxval < a_i$ **then**
7: $maxval \leftarrow a_i$
8: Return $maxval$ ▷ $maxval$ is the largest element

PSEUDO-CODE CONSTRUCTS (FOR THIS CLASS)

- Header (Name, Input, Output)
- Value assignment, Comparisons
- Alternate path selection
  - if (condition) … else if … else …. end if
- Iterations
  - for (condition) … end for
  - while (condition) … end while
  - do … while (condition)

EXPRESSING ALGORITHM

3.PROGRAMMING LANGUAGE

- Try to write by Python

Algorithm MaxElement

1: Input: A sequence of numbers $a_1, a_2, \cdots, a_n$
2: Output: The largest element in the input sequence
3:
4: $maxval \leftarrow a_1$
5: **for** $i = 2$ to n **do**
6: **if** $maxval < a_i$ **then**
7: $maxval \leftarrow a_i$
8: Return $maxval$ ▷ $maxval$ is the largest element

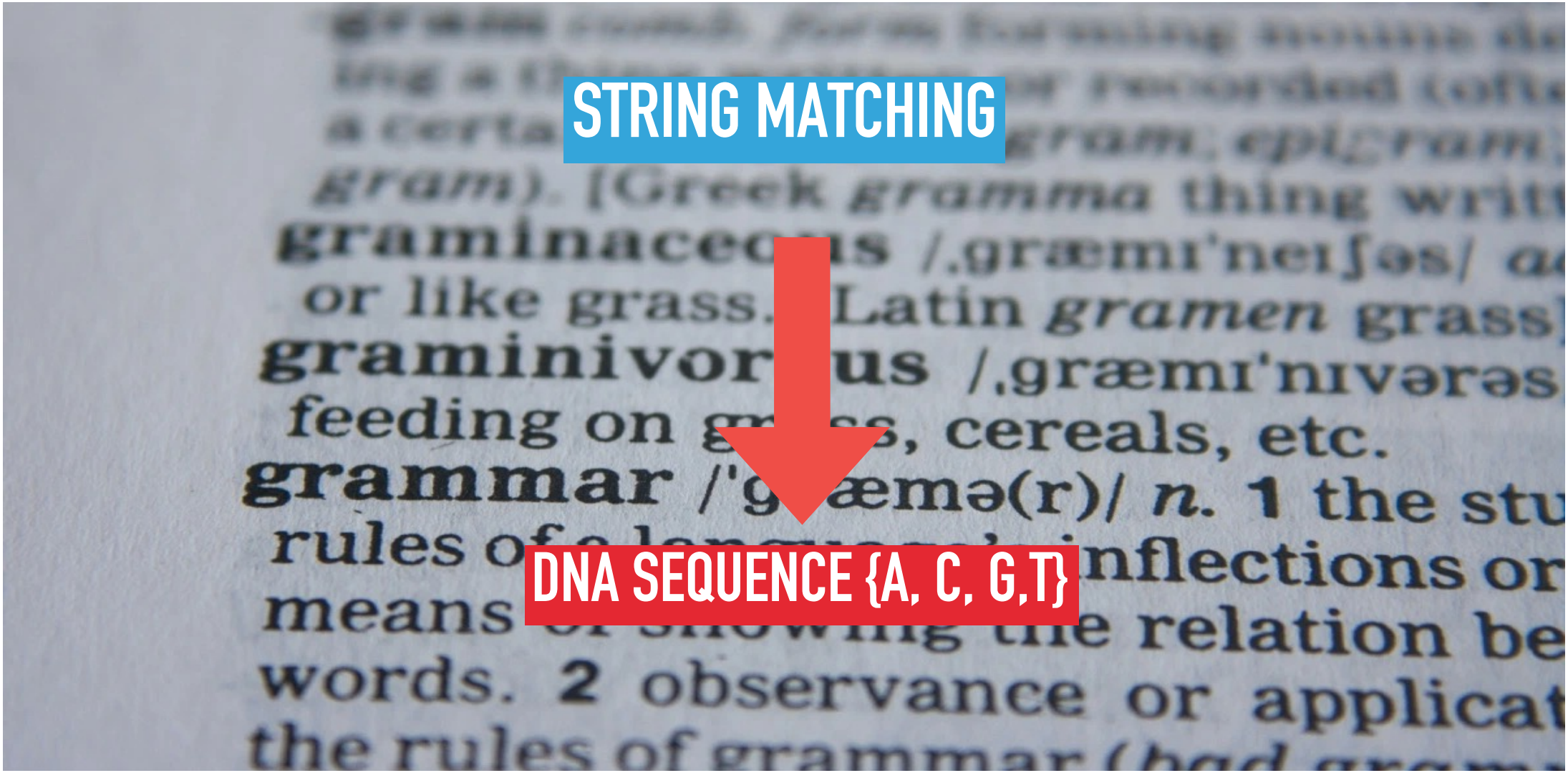
```
def MaxElement(A):
    maxval = A[0]    Python, start the index with 0
    for i in range(1,len(A)):
        if maxval < A[i]:
            maxval = A[i]
    return maxval
```

SORTING PROBLEMS

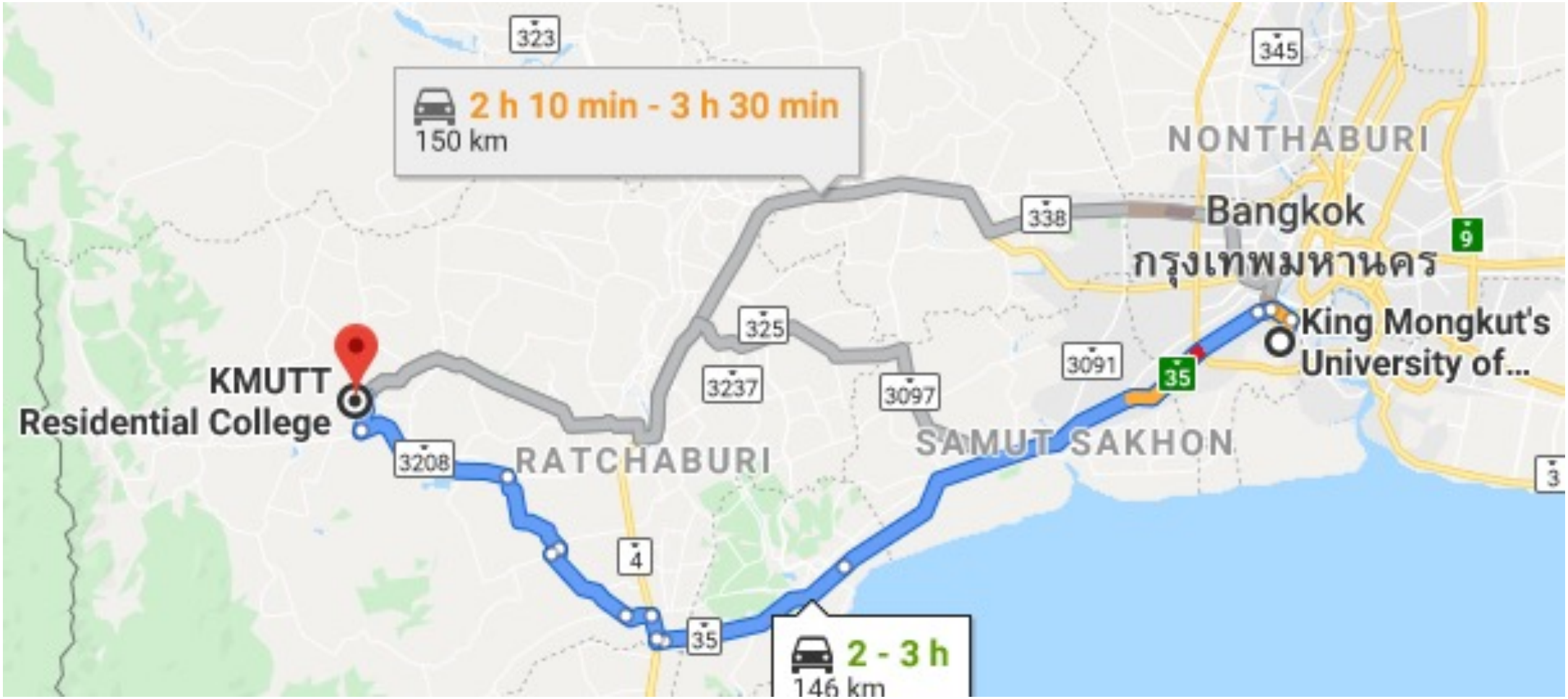
SEARCHING PROBLEMS

STRING PROCESSING

String is a sequence of characters from an alphabet

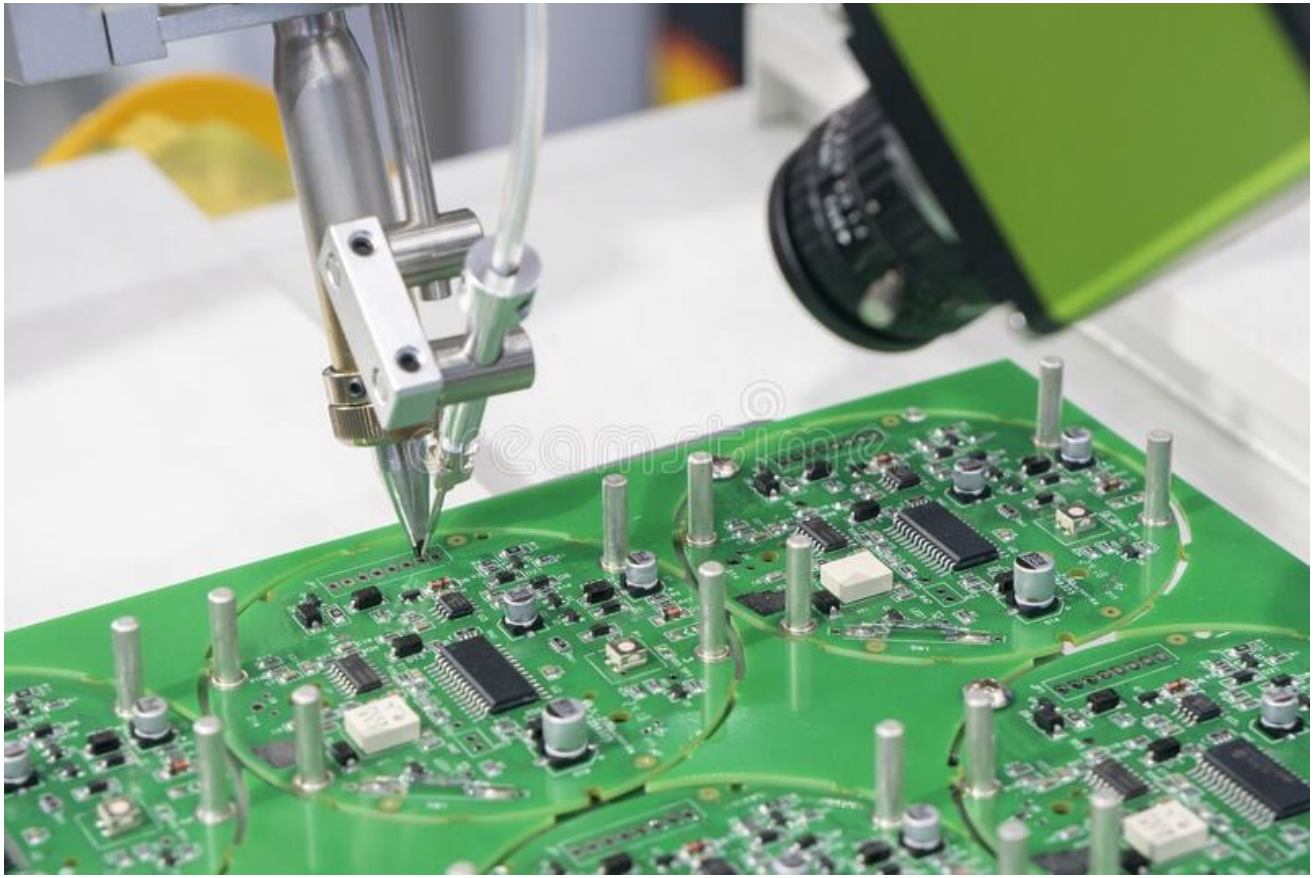


GRAPH PROBLEMS



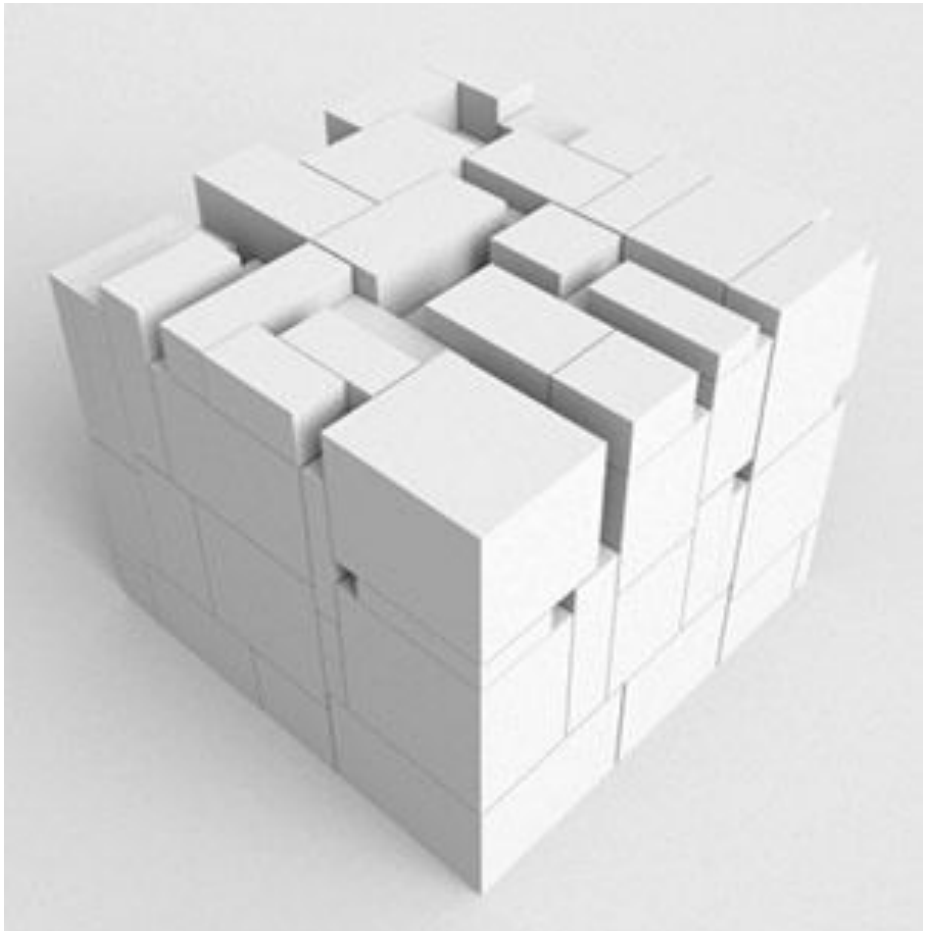
TRAVEL LOCATION PLANNING

GRAPH PROBLEMS IN INDUSTRIAL



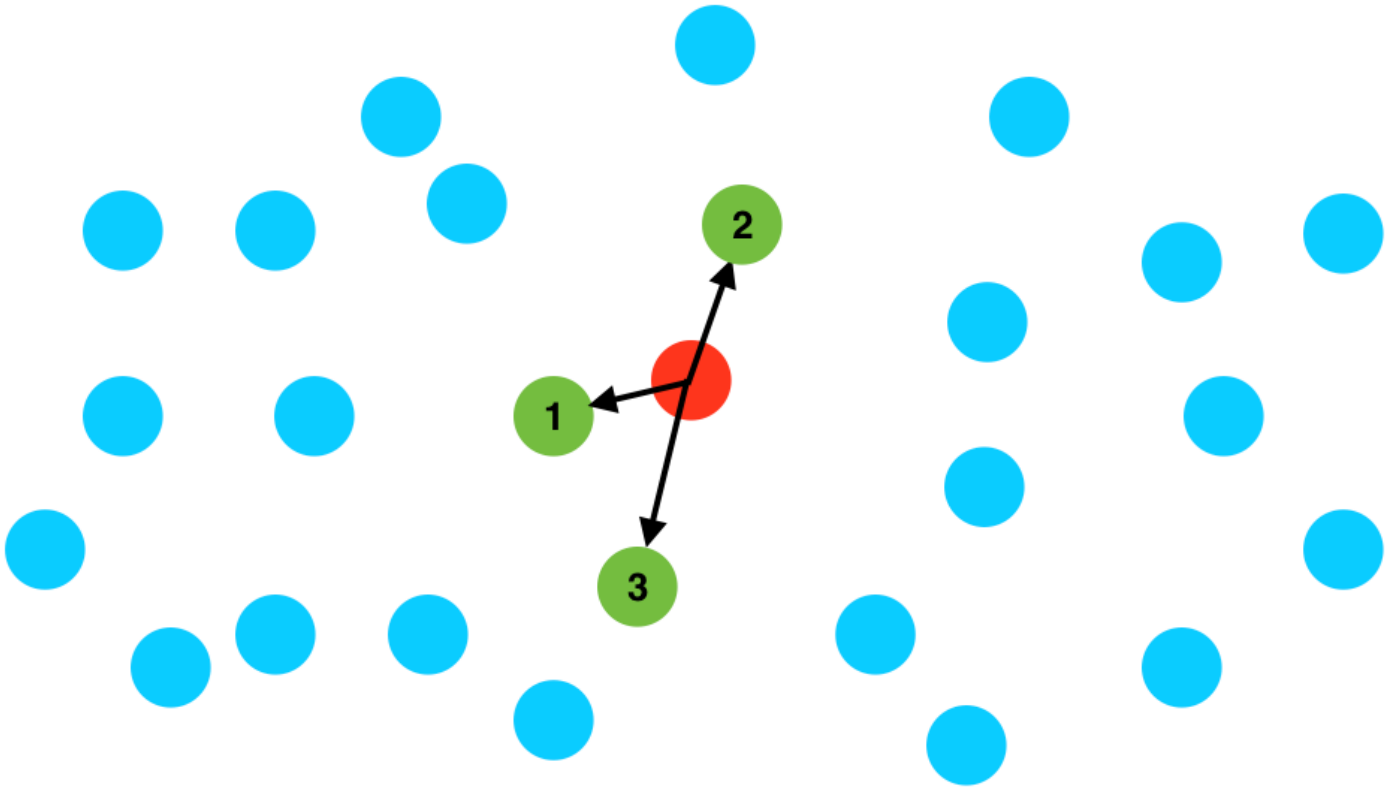
ROBOT TOUR OPTIMIZATION

COMBINATORIAL PROBLEMS IN LOGISTICS



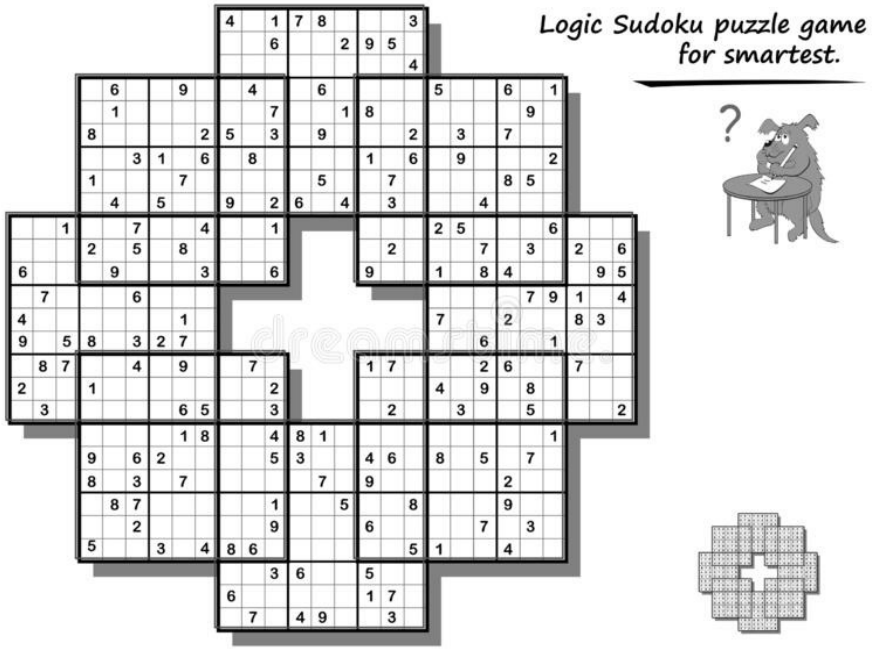
BIN-PACKING PROBLEM

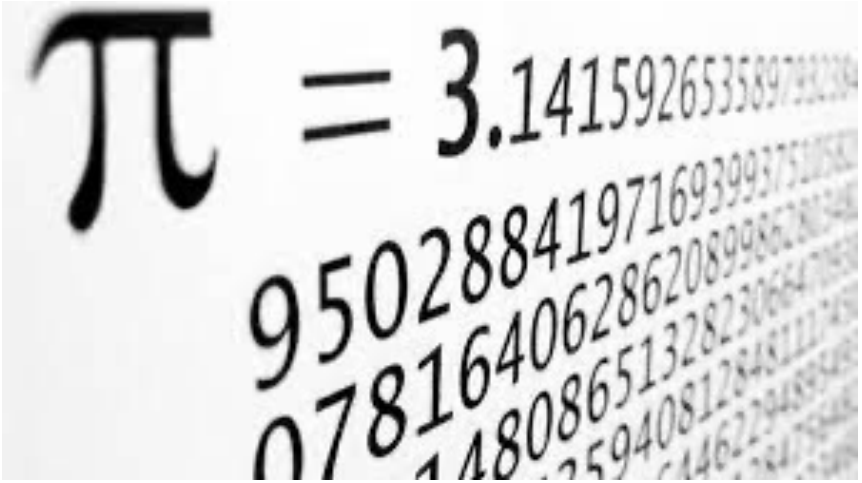
GEOMETRIC PROBLEMS



CLOSET PAIR PROBLEM

NUMERICAL PROBLEMS





RSA Algorithm

Key Generation

Select p,q	p and q both prime; p ≠ q
Calculate $n = p \times q$	
Calculate $\phi(n) = (p-1)(q-1)$	
Select integer e	$\gcd(\phi(n),e) = 1;\ 1 < e < \phi(n)$
Calculate d	$de \bmod \phi(n) = 1$
Public key	$KU = \{e,n\}$
Private key	$KR = \{d,n\}$

Encryption

Plaintext:	$M < n$
Ciphertext:	$C = M^e \pmod n$

Decryption

Plaintext:	C
Ciphertext:	$M = C^d \pmod n$

SUMMARY – IMPORTANT PROBLEM TYPES

- Sorting
- Searching
- String processing
- Graph problems
- Combinatorial problems
- Geometric problems
- Numerical problems

ALGORITHM DESIGN STRATEGIES

- Brute force
- Greedy approach
- Divide/Decrease/Transform and conquer
- Dynamic programming
- Backtracking and branch-and-bound
- etc.

EX: GREATEST COMMON DIVISOR (GCD)

- Find gcd(m,n), the greatest common divisor of two non-negative, not both zero integers, m and n.
  - gcd(60, 24) = ?
  - gcd(60, 0) = ?

"MIDDLE-SCHOOL" PROCEDURE TO GCD(M,N)

- Step 1: Find the prime factorization of m
- Step 2: Find the prime factorization of n
- Step 3: Find all the common prime factors
- Step 4: Compute the product of all the common prime factors and return it as gcd(m,n)

"BRUTE FORCE" PROCEDURE TO GCD(M,N)

- Step 1: Assign the value of min{m,n} to t.
- Step 2: Divide m by t. If the remainder is 0, go to Step 3; Otherwise, go to Step 4
- Step 3: Divide n by t. If the remainder is 0, return t and stop; Otherwise, go to Step 4
- Step 4: Decrease t by 1 and go to Step 2

DOES IT WORK FOR EVERY INPUTS ?

DESCRIPTIONS OF EUCLID'S ALGORITHM

- Step 1: If n = 0, return m and stop; otherwise go to Step 2
- Step 2: Divide m by n and assign the value of the remainder to r
- Step 3: Assign the value of n to m and the value of r to n. Go to Step 1.

```
EUCLID'S ALGORITHM: FOR M ≥ N,

 GCD(M, N) = GCD(N, M MOD N)
 REPEAT UNTIL N BECOMES 0 AND TAKE M AS THE OUTPUT
```

PSEUDO-CODE

Algorithm Euclid's algorithm

1: Input: Two non-negative, not both zero, integers m and n ($m \geq n$)
2: Output: A largest integer not greater than $\min(m, n)$ that divides m and n
3:
4: **while** $n \neq 0$ **do**
5: $r \leftarrow m \bmod n$
6: $m \leftarrow n$
7: $n \leftarrow r$
8: return m

BACK TO "MIDDLE-SCHOOL" PROCEDURE TO GCD(M,N)

How can we know which is prime number ?

- Step 1: Find the prime factorization of m
- Step 2: Find the prime factorization of n
- Step 3: Find all the common prime factors
- Step 4: Compute the product of all the common prime factors and return it as gcd(m,n)

PSEUDO-CODE

Algorithm Sieve of Eratosthenes

Input: Integer $n \geq 2$

Output: List of primes less or equal to n

for $p \leftarrow 2$ to n **do**

    $A[p] \leftarrow p$

for $p \leftarrow 2$ to $\lfloor \sqrt{n} \rfloor$ **do**

    if $A[p] \neq 0$ **then** ▷ p `hasn't been previously eliminated from the list`

        $j \leftarrow p \times p$

        while $j \leq n$ **do**

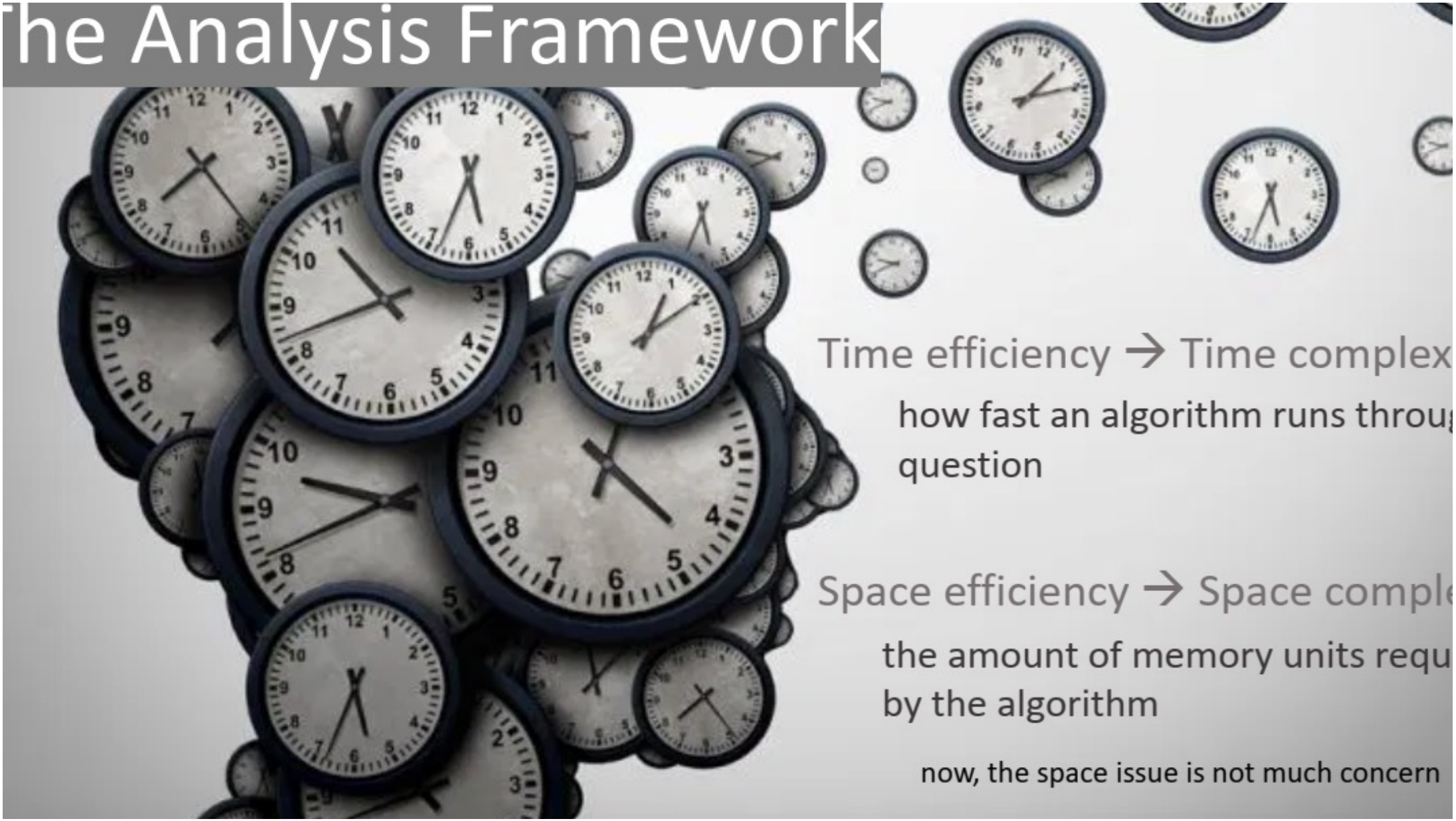
            $A[j] \leftarrow 0$ ▷ `Mark element as eliminated`

            $j \leftarrow j + p$

SIEVE OF ERATOSTHENES

2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25

The Analysis Framework



Time efficiency → Time complex

how fast an algorithm runs throug
question

Space efficiency → Space comple

the amount of memory units requ
by the algorithm

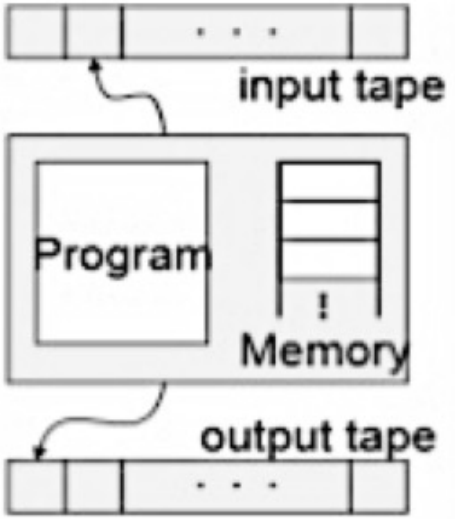
now, the space issue is not much concern

Measuring Running Time

Analyzed Factors depend on

- number of processors
- Read/Write speed (cache, disk)
- 32bit or 64 bit-Architecture
- …
- …
- number of Input

order of growth

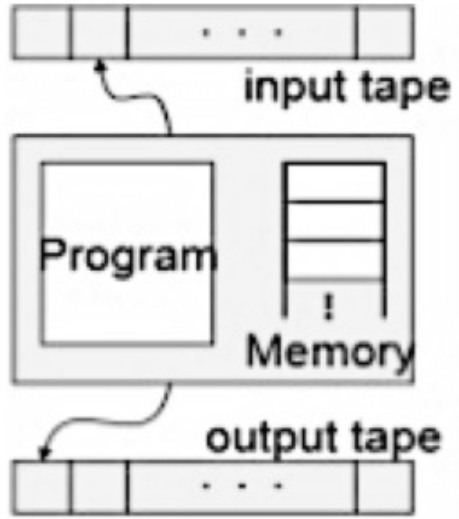


RAM model

- single processor
- sequential execution
- 1 unit time for arithmetical and logical operation
- 1 unit time for assignment and return

Measuring Running Time

RAM Model of Computation



Machine-independent algorithm design depends upon a hypothetical computer called the *Random Access Machine* (RAM). Under this model of computation, simply run time is calculated by

- Each ``simple'' operation (+, *, -, =, if, call) takes exactly 1 time step.
- Loops and subroutines are *not* considered simple operations. Instead, they are the composition of many single-step operations. The time it takes to run through a loop or execute a subprogram depends upon **the number of loop iterations** or the specific nature of the subprogram.
- Each memory access takes exactly one time step, and we have as much memory as we need. no matter an item is in cache or on the disk

run time of an algorithm = No. of steps it takes on a given problem instance

Example: Determine the running time of the algorithm that finds the summation for all elements in the sequence

Algorithm Sum_all_Element

1. Input: A sequence of numbers $a_1, a_2, \dots, a_n$
2. Output: The summation of all elements in the input sequence
3.

	cost	No. of iteration
4. sum ←0	1	1
5. **for** i = 1 to n **do**	2	n+1
6. sum ← sum + a_i	2	n
7. return sum	1	1

total = 1 + 2(n+1) + 2n +1 = 4n + 4

Algorithm running time [ ] input's size

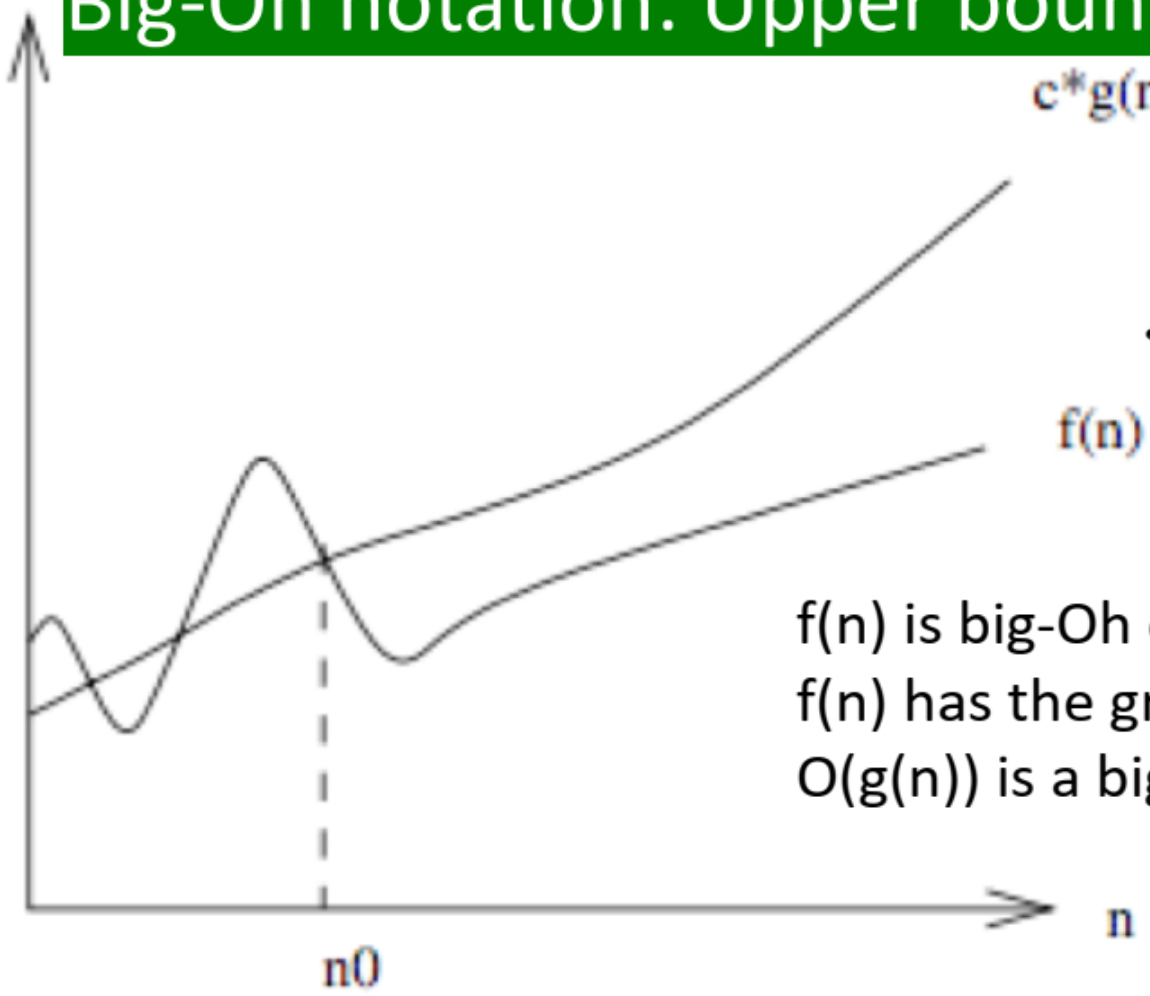
Asymptotic Notation

- Big-Oh notation (O) – Upper bound
- Big-Omega notation (Ω) – Lower bound
- Big-Theta notation (Θ) – Average bound

comparing and ranking growth orders of algorithms solving the same problem.

Big-Oh notation: Upper bound

$$f(n) \in O(g(n))$$



$$f(n) \leq c \cdot g(n),\ \forall n \geq n_0$$

f(n) is big-Oh of g(n)
f(n) has the growth order bounded above by g(n)
O(g(n)) is a big-Oh estimate of f(n)

Example: f (n) = 4 n + 4

$4n + 4 \leq 10n \quad \text{for } n \geq 1$

f(n) c g(n)

$$f(n) \leq c \cdot g(n), \ \forall n \geq n_0$$

∴ f(n) = O(n)

$4n + 4 \leq 4n + 4n \quad \text{for } n \geq 1$

8n

f(n) c g(n)

$$f(n) \leq c \cdot g(n), \ \forall n \geq n_0$$

∴ f(n) = O(n)

Example: $f(n) = 4n + 4$

$4n + 4 \leq 4n^2 + 4n^2$ for $n \geq 1$

$8n^2$

f(n) c $g(n^2)$

$$f(n) \leq c \cdot g(n), \ \forall n \geq n_0$$

$\therefore f(n) = O(n^2)$

$4n + 4 \leq 4n^3 + 4n^3$ for $n \geq 1$

$8n^3$

f(n) c $g(n^3)$

$$f(n) \leq c \cdot g(n), \ \forall n \geq n_0$$

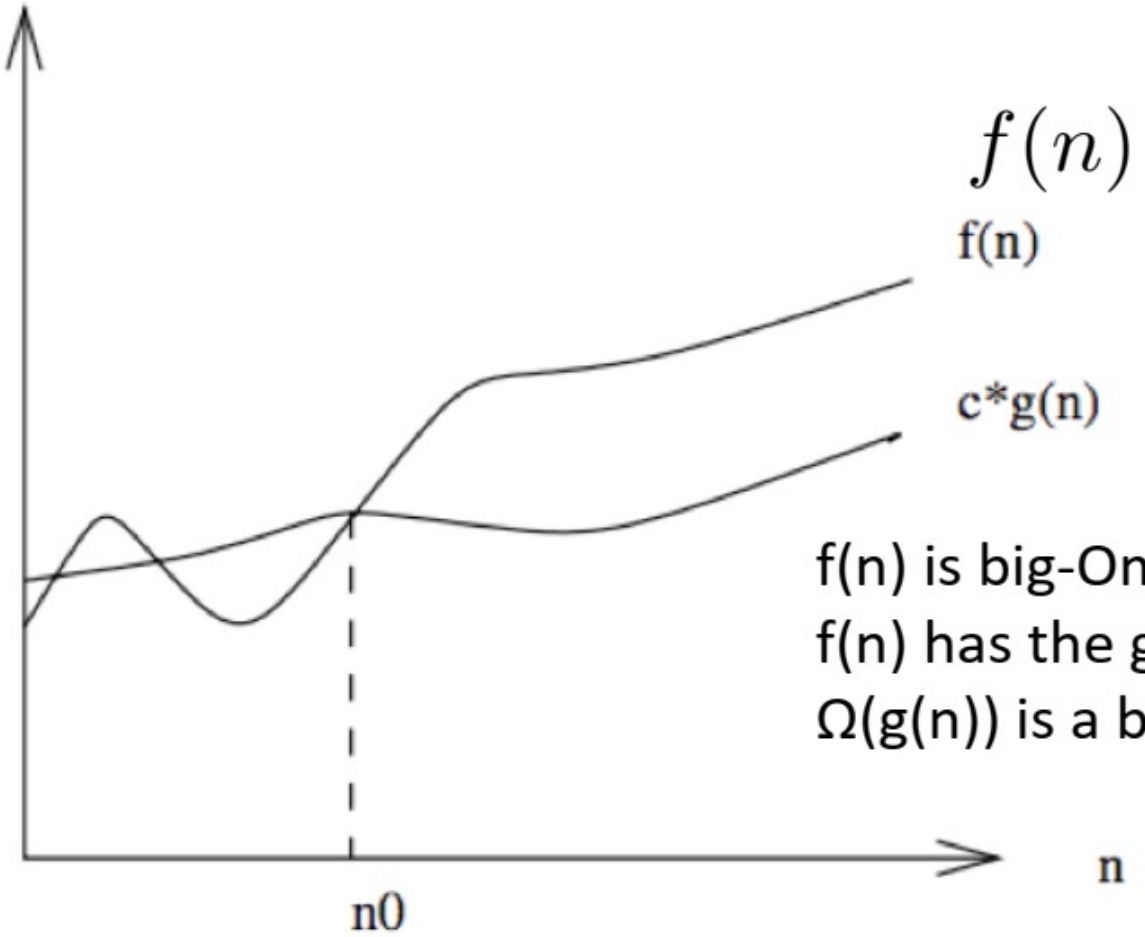
$\therefore f(n) = O(n^3)$

$1 < \log(n) < \sqrt{n} < n < n\log(n) < n^2 < n^3 < \ldots < 2^n < 3^n < n^n$

Big-Omega notation: Lower bound

$$f(n) \in \Omega(g(n))$$

$$f(n) \geq c \cdot g(n), \ \forall n \geq n_0$$



f(n) is big-Omega of g(n)
f(n) has the growth order bounded below by g(n)
Ω(g(n)) is a big-Omega estimate of f(n)

Example: $f(n) = 4n + 4$

$4n + 4 \geq 2n$ for $n \geq 1$

f(n) c g(n)

$$f(n) \geq c \cdot g(n), \ \forall n \geq n_0$$

∴ f(n) = Ω(n)

let's test for log n

∴ f(n) = Ω(log n)

$1 < \log(n) < \sqrt{n} < n < n\log(n) < n^2 < n^3 < \ldots < 2^n < 3^n < n^n$

Determine which one below is true or false.

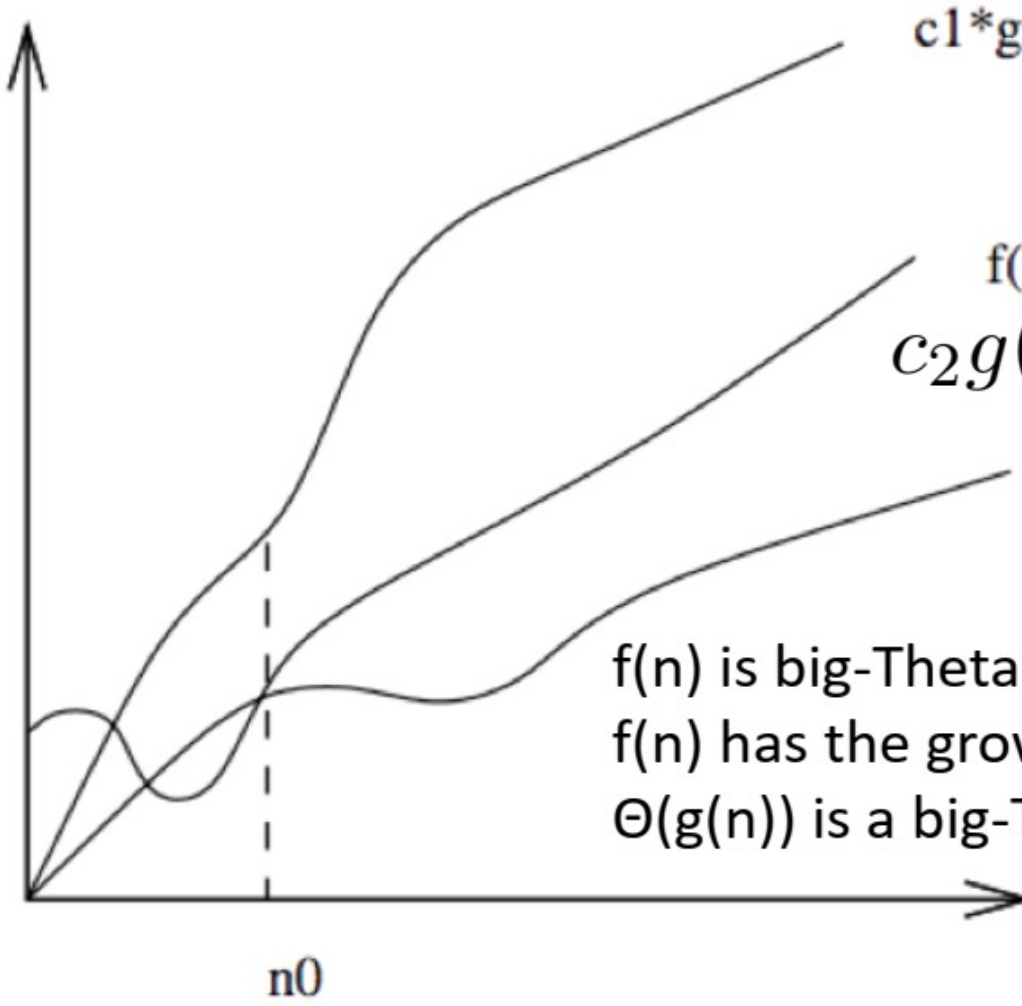
$$3n^2 - 100n + 6 \quad \in \quad \Omega(n^2) \tag{1}$$

$$3n^2 - 100n + 6 \quad \in \quad \Omega(n^3) \tag{2}$$

$$3n^2 - 100n + 6 \quad \in \quad \Omega(n) \tag{3}$$

Big-Theta notation: Average bound

$$f(n) \in \Theta(g(n))$$

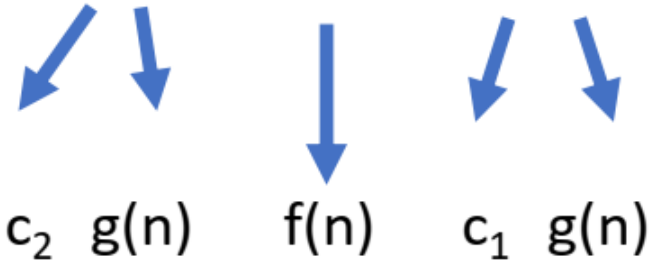


$$c_2 g(n) \leq f(n) \leq c_1 g(n), \ \forall n \geq n_0$$

f(n) is big-Theta of g(n)
f(n) has the growth order bounded above and below by g(n)
Θ(g(n)) is a big-Theta estimate of f(n)

Example: $f(n) = 4n + 4$

$2n \le 4n + 4 \le 10n$ for $n \ge 1$



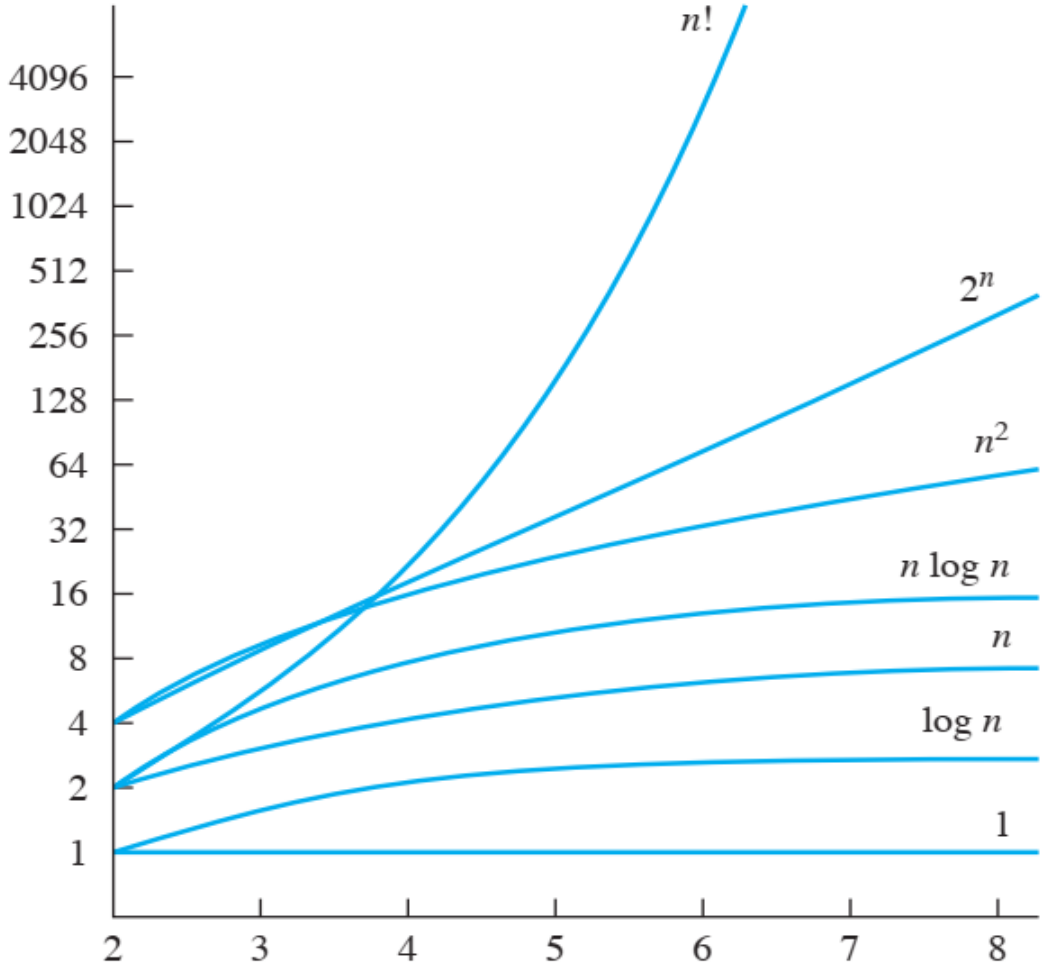
$$c_2 g(n) \le f(n) \le c_1 g(n), \ \forall n \ge n_0$$

$\therefore f(n) = \Theta(n)$

$1 < \log(n) < \sqrt{n} < n < n\log(n) < n^2 < n^3 < \ldots < 2^n < 3^n < n^n$

The growth order of Algorithm

- The growth order is the dominant term in the running time, **with any multiplicative constant removed.**
- The growth order is an indicator of **the algorithm's efficiency** used to rank and compare algorithms.



Summary

- **Big-Oh** is upper bound → execution time is not more than $O(g(n))$
- **Big-Omega** is lower bound → execution time is at least $\Omega(g(n))$
- **Big-Theta** is the most preferred expression for algorithm efficiency
- $O(g(n))$ may equal to $\Omega(g(n))$ and may equal to $\Theta(g(n))$
- **Big-Oh** and **Big-Omega** may have more than one function but normally we use the closest function
- Sometime $\Theta(g(n))$ is not exists
  - e.g.: $f(n) = n!$

- **Basic Asymptotic Efficiency Class**

Class g(n)	Name
1	Constant
log n	Logarithmic
n	Linear
n log n	Linearithmic / Superlinear
$\sqrt{n}$	Sublinear polynomial
n^b	Polynomial
b^n	Exponential
n!	Factorial

Example: Determine the running time of the algorithm that finds the summation for all element in the sequence

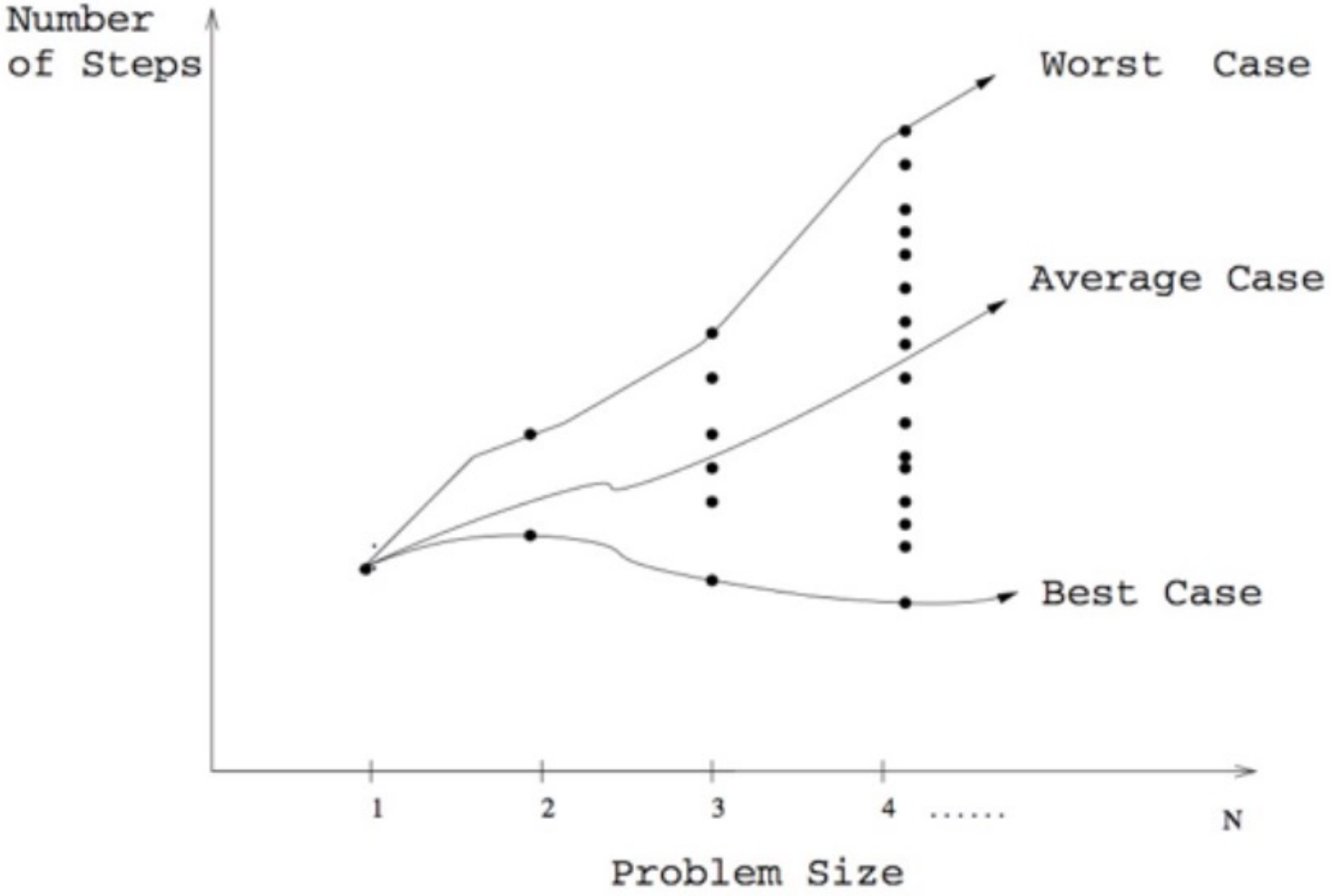
Algorithm UniqueElement

1. Input: A sequence of numbers $a_1, a_2, \dots, a_n$
2. Output: Return "true" if all elements are distinct and "false" otherwise

3.	Cost	No. of Iteration
4. **for** i = 1 to n -1 **do**	2	?
5. **for** j = i+1 to n **do**	2	?
6. **if** a_i == a_j **then**	1	?
7. **return** false	1	1
8. **return** true	1	1

Algorithm running time dose not depend on input's size

Best/ Average/ Worst-case Efficiencies



Example: Determine the running time of the algorithm that finds the summation for all element in the sequence

Algorithm UniqueElement

1. Input: A sequence of numbers $a_1, a_2, \dots, a_n$
2. Output: Return "true" if all elements are distinct and "false" otherwise
3.
4. **for** i = 1 to n -1 **do**
5. **for** j = i+1 to n **do**
6. **if** $a_i == a_j$ **then**
7. **return** false
8. **return** true

Best Case	Worst Case
$a_1 = a_2$	all elements are
f(n) = 6	unique

Algorithm running time dose not depend on input's size

Linear Search

2	23	45	12	16	76	34	87	56	39

Key = 12

Key =25

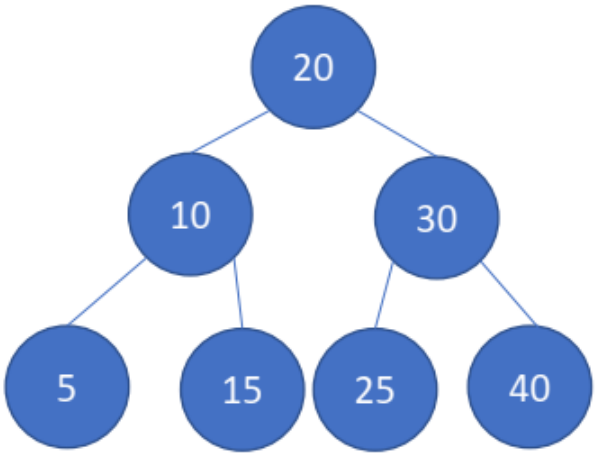
Best Case → found at the first order → O(n)

Worst Case → found at the last order/ not found → O(n)

Average Case → $\frac{all\ possible\ case\ time}{no.of\ case}$

$$= \frac{1+2+\cdots n}{n} = \frac{\frac{n(n+1)}{2}}{n} = \frac{n+1}{2}$$

Binary Search Tree



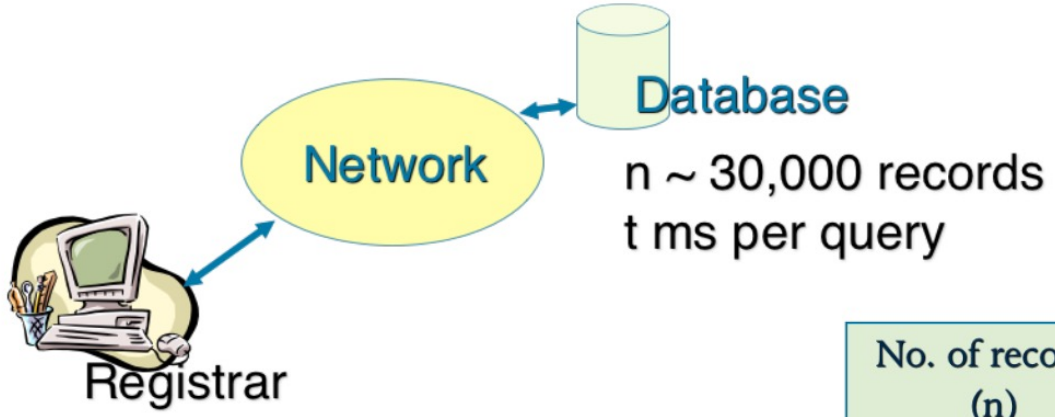
Best case → found at root → O(1)

Worst case → found at leaf → height of Tree

General Plan for Analyzing Non-Recursive Algorithms

Step	Ex: UniqueElement()
Decide on parameters indicating an input's size.	n
Identify algorithm's basic operations	for, if $a_i == a_j$ and return
Check if basic operations depend on additional properties other than the input size.	if $a_i == a_j$
Set up the sum expressing # executions of basic operations.	Best case = f(n) Worst case = f(n)
Evaluate the sum to closed-form (or establish growth order)	Best case = O(g(n)) Worst case = O(g(n))

How does time-complexity grow the time?



How long to wait on average?

n	$f(n)$	$\lg n$	n	$n \lg n$	n^2	2^n	$n!$
10		0.003 μs	0.01 μs	0.033 μs	0.1 μs	1 μs	3.63 ms
20		0.004 μs	0.02 μs	0.086 μs	0.4 μs	1 ms	77.1 years
30		0.005 μs	0.03 μs	0.147 μs	0.9 μs	1 sec	8.4×10^{15} yrs
40		0.005 μs	0.04 μs	0.213 μs	1.6 μs	18.3 min	
50		0.006 μs	0.05 μs	0.282 μs	2.5 μs	13 days	
100		0.007 μs	0.1 μs	0.644 μs	10 μs	4×10^{13} yrs	
1,000		0.010 μs	1.00 μs	9.966 μs	1 ms		
10,000		0.013 μs	10 μs	130 μs	100 ms		
100,000		0.017 μs	0.10 ms	1.67 ms	10 sec		
1,000,000		0.020 μs	1 ms	19.93 ms	16.7 min		
10,000,000		0.023 μs	0.01 sec	0.23 sec	1.16 days		
100,000,000		0.027 μs	0.10 sec	2.66 sec	115.7 days		
1,000,000,000		0.030 μs	1 sec	29.90 sec	31.7 years		

No. of records (n)	Waiting times (10 msec per search)	
	Linear search (n)	Binary search ($\log_2 n$)
n = 1000	10 s	0.1 s
n = 10000	100 s	0.1329 s
n = 30000	300	0.1487 s

No. of records (n)	Waiting times (1 msec per search)	
	Linear search (n)	Binary search ($\log_2 n$)
n = 1000	1 s	0.01 s
n = 10000	10 s	0.01329 s
n = 30000	30 s	0.01487 s

Running Time Analysis of Recursive Algorithms

Algorithm Factorial(n)

1: Input: A non-negative integer n
2: Output: The value of $n!$
3:
4: **if** $n = 0$ **then**
5: return 1
6: **else**
7: return Factorial$(n-1)*n$

$$F(n) = \begin{cases} F(n-1) \cdot n & \text{for } n > 0 \\ 1 & \text{for } n = 0 \end{cases}$$

Method of Backward Substitutions

M(n), # basic operations (multiplication), has the recurrence relation:

$$M(n) = \begin{cases} M(n-1)+1 & \text{for } n > 0 \\ 0 & \text{for } n = 0 \end{cases}$$

$$F(n) = \begin{cases} F(n-1) \cdot n & \text{for } n > 0 \\ 1 & \text{for } n = 0 \end{cases}$$

$$M(n) = M(n-1) + 1 \qquad \text{substitute } M(n-1) = M(n-2) + 1$$
$$= [M(n-2) + 1] + 1 = M(n-2) + 2 \qquad \text{substitute } M(n-2) = M(n-3) + 1$$
$$= [M(n-3) + 1] + 2 = M(n-3) + 3.$$

Therefore,

$$M(n) = M(n-1) + 1 = \cdots = M(n-i) + i = \cdots = M(n-n) + n = n.$$

Example: Determine the running time of the algorithm that finds the Fibonacci sequence

Algorithm Fibonacci

1. Input: A numbers *n*
2. Output: Return the Fibonacci series value of *n*
3.
4. **if** n ≤ 1
5. **return** n
6. **else**
7. **return** Fibonacci(n-1) + Fibonacci(n-2)

$T(n) = T(n-1) + T(n-2) + C$

$T(0) = T(1) = 1$

assume that $T(n-1) \simeq T(n-2)$

$T(n) = 2T(n-2) + C$

$= 2[2T(n-4)+C] + C$

$= 4T(n-4) + 3C$

$= 8T(n-6) + 7C$

$= 16T(n-8) + 15C$

$T(n) = 2^k T(n-2K) + (2^k - 1) C$

given $k = n/2$

$\Omega(2^{n/2})$

Lower bound

$T(n) = 2^{n/2} T(0) + (2^{n/2} - 1)C$

$= 2^{n/2} (1+C) - C$

$T(n) = T(n-1) + T(n-2) + C$

$T(0) = T(1) = 1$

assume that $T(n-2) \simeq T(n-1)$

$$
\begin{aligned}
T(n) &= 2T(n-1) + C \\
&= 2[2T(n-2)+C] + C \\
&= 4T(n-2) + 3C \\
&= 8T(n-3) + 7C \\
&= 16T(n-4) + 15C
\end{aligned}
$$

$T(n) = 2^k T(n-K) + (2^k - 1) C$

given $k = n$

$O(2^n)$

Upper bound

$$
\begin{aligned}
T(n) &= 2^n T(0) + (2^n - 1)C \\
&= 2^n (1+C) - C
\end{aligned}
$$

General Plan for Analyzing Recursive Algorithms

- Decide on parameters indicating an input's size.
- Identify algorithm's basic operations
- Check if basic operations depend on additional properties other than the input size. If so, worst-case, average-case, best-case efficiencies have to be derived separately.
- Set up a recurrence relation with appropriate initial condition expressing # basic operations executed.
- Solve the recurrence to closed-form (or establish growth order)