UNIVERSITI TEKNOLOGI MALAYSIA

DSPD1733: DATA STRUCTURES AND ALGORITHMS

Pusat Pengajian Diploma, SPACE UTM Kuala Lumpur

Jabatan Sains Komputer dan Perkhidmtan

CHAPTER 2:
ALGORITHM ANALYSIS

- Introduction to algorithm, pseudocode
- Characteristic of an algorithm
- Algorithm analysis
- Algorithm complexity analysis

UTM
UNIVERSITI TEKNOLOGI MALAYSIA

01 Algorithms

Why study Algorithms?, Characteristics of an Algorithm,

02 Algorithm Analysis

Priori Analysis, Running Time, Primitive Operations

03 Algorithm Complexity Analysis

Calculate Time and Space Complexity, Analyzing Algorithm Efficiency

Algorithms

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein in Introduction to Algorithms (Second Edition) 2001:

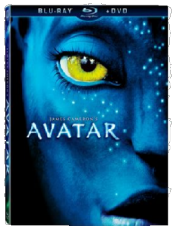
  "an algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values as output."

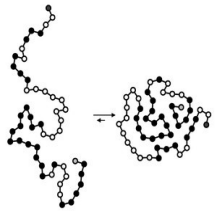
- Algorithm: method for solving a problem.
- Data structure: method to store information.

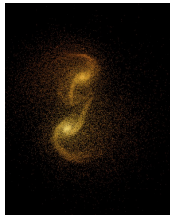
Why study algorithms?

Their impact is broad and far-reaching.

- Internet. Web search, packet routing, distributed file sharing, …
- Biology. Human genome project, protein folding, ...
- Computers. Circuit layout, file system, compilers, ...
- Computer graphics. Movies, video games, virtual reality, ...
- Security. Cell phones, e-commerce, voting machines, …
- Multimedia. MP3, JPG, DivX, HDTV, face recognition, ...
- Social networks. Recommendations, news feeds, advertisements, …
- Physics. N-body simulation, particle collision simulation, ...



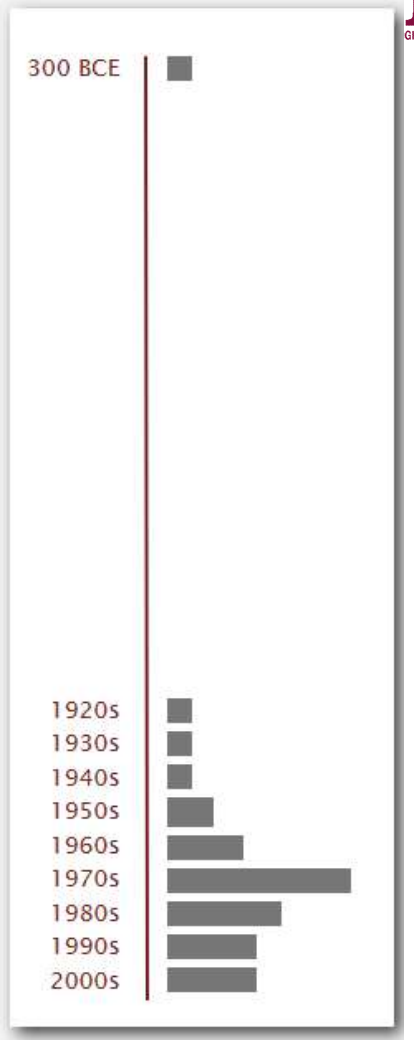




Why study algorithms?

Old roots, new opportunities.

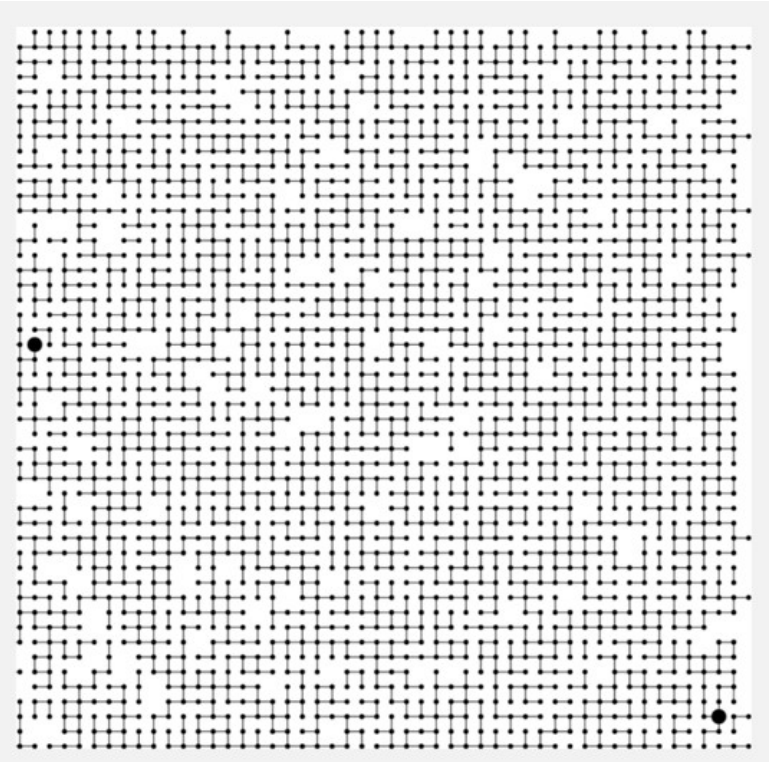
- Formalized by Church and Turing in 1930s.
- Some important algorithms were discovered by undergraduates in a course like this!



Why study algorithms?

To solve problems that could not otherwise be addressed.

Ex. Network connectivity.



Why study algorithms?

For intellectual stimulation.

> *" For me, great algorithms are the poetry of computation. Just like verse, they can be terse, allusive, dense, and even mysterious. But once unlocked, they cast a brilliant new light on some aspect of computing. "* — *Francis Sullivan*

FROM THE EDITORS

THE JOY OF ALGORITHMS

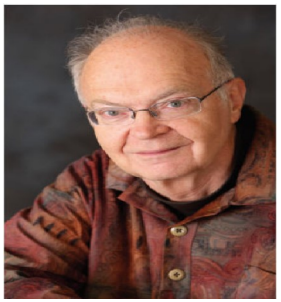
Francis Sullivan, Associate Editor-in-Chief

THE THEME OF THIS FIRST-OF-THE-CENTURY ISSUE OF *COMPUTING IN*

ENOUGH—AND PERHAPS FOOLISH ENOUGH—TO CALL THE 10 EXAMPLES WE'VE SE-

LECTED "THE TOP 10 ALGORITHMS OF THE CENTURY."

> *" An algorithm must be seen to be believed. "* — *Donald Knuth*

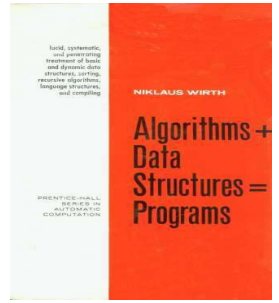


Why study algorithms?

To become a proficient programmer.

> *" I will, in fact, claim that the difference between a bad programmer and a good one is whether he considers his code or his data structures more important. Bad programmers worry about the code. Good programmers worry about data structures and their relationships. "*
>
> *— Linus Torvalds (creator of Linux)*

> *" Algorithms + Data Structures = Programs. "* *— Niklaus Wirth*



Why study algorithms?

They may unlock the secrets of life and of the universe.

Computational models are replacing math models in scientific inquiry.

$$E = mc^2 \qquad F = \frac{Gm_1m_2}{r^2}$$

$$F = ma$$

$$\left[-\frac{\hbar^2}{2m}\nabla^2 + V(r)\right]\Psi(r) = E\,\Psi(r)$$

20th century science
(formula based)

```
for (double t = 0.0; true; t = t + dt)
   for (int i = 0; i < N; i++)
   {
      bodies[i].resetForce();
      for (int j = 0; j < N; j++)
         if (i != j)
            bodies[i].addForce(bodies[j]);
   }
```

21st century science
(algorithm based)

"Algorithms: a common language for nature, human, and computer." — *Avi Wigderson*

Why study algorithms?

For fun and profit.

Characteristics of an Algorithm

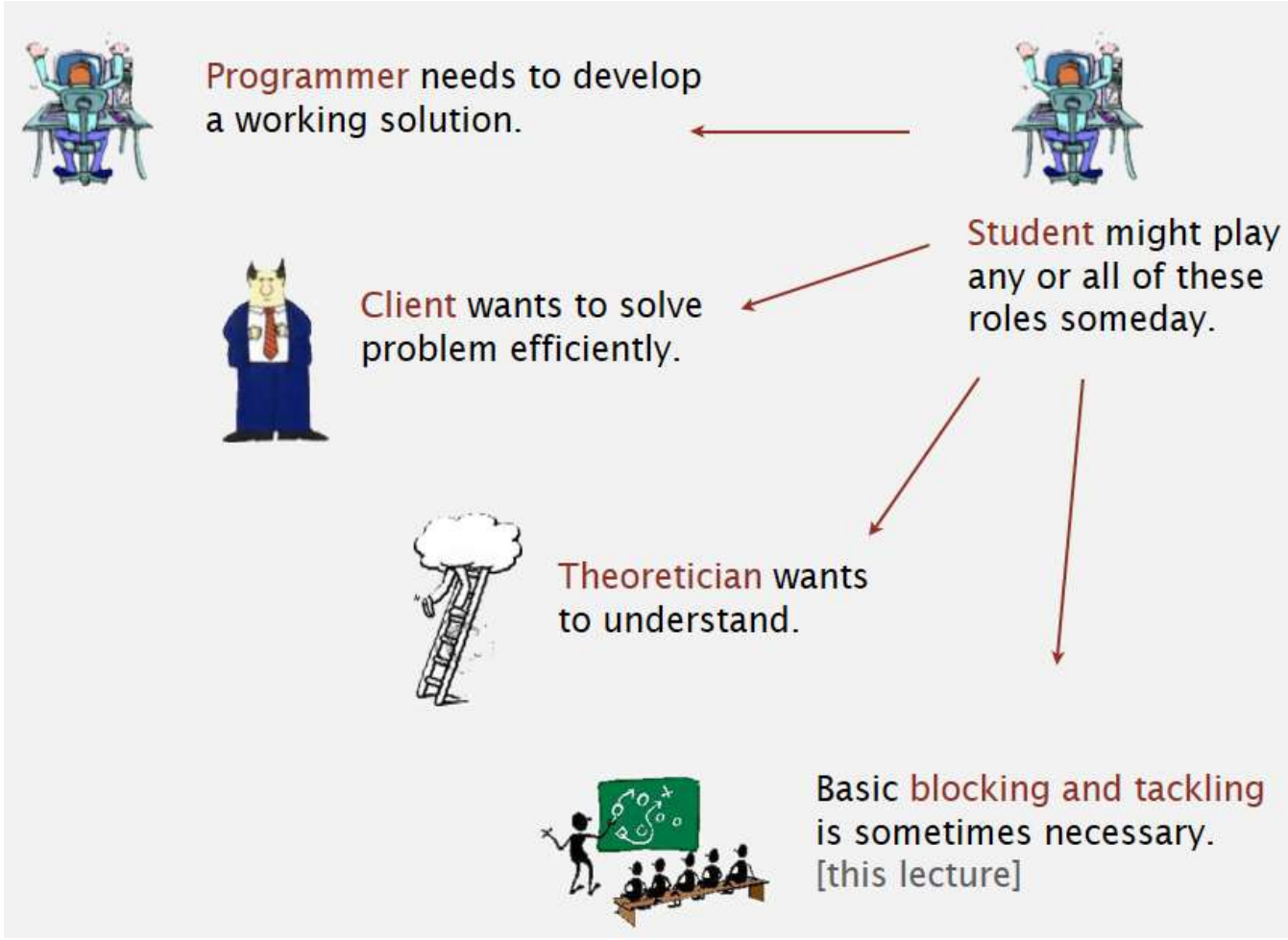
Not all procedures can be called an algorithm.

An algorithm should have the following characteristics:

- **Unambiguous** – Algorithm should be clear and unambiguous. Each of its steps (or phases), and their inputs/outputs should be clear and must lead to only one meaning.
- **Input** – An algorithm should have 0 or more well-defined inputs.
- **Output** – An algorithm should have 1 or more well-defined outputs, and should match the desired output.
- **Finiteness** – Algorithms must terminate after a finite number of steps.
- **Feasibility** – Should be feasible with the available resources.
- **Independent** – An algorithm should have step-by-step directions, which should be independent of any programming code.

Roles involves in need of an Algorithms

Cast of characters:



Pseudocode Algorithm

- In this course, mostly will use pseudocode to describe an algorithm.
- Pseudocode is a high-level description of an algorithm.
- More structured than English prose.
- Less detailed than a program.
- Preferred notation for describing algorithms.

Example: find max element of an array

Algorithm ***arrayMax***(***A***, ***n***)
  Input: array ***A*** of ***n*** integers
  Output: maximum element of ***A***

```
currentMax ← A[0]
for i ← 1 to n – 1 do
  if A[i] > currentMax then
    currentMax ← A[i]
return currentMax
```

Pseudocode Details

- Control flow
  - **if** … **then** … [**else** …]
  - **while** … **do** …
  - **repeat** … **until** …
  - **for** … **do** …
  - Indentation replaces braces
- Method declaration

  Algorithm ***method*** (***arg, arg***…)

  Input …

  Output …
- Method call

  var.method (***arg*** [, ***arg***…])
- Return value

  return ***expression***
- Expressions

  Assignment (like = in Java)

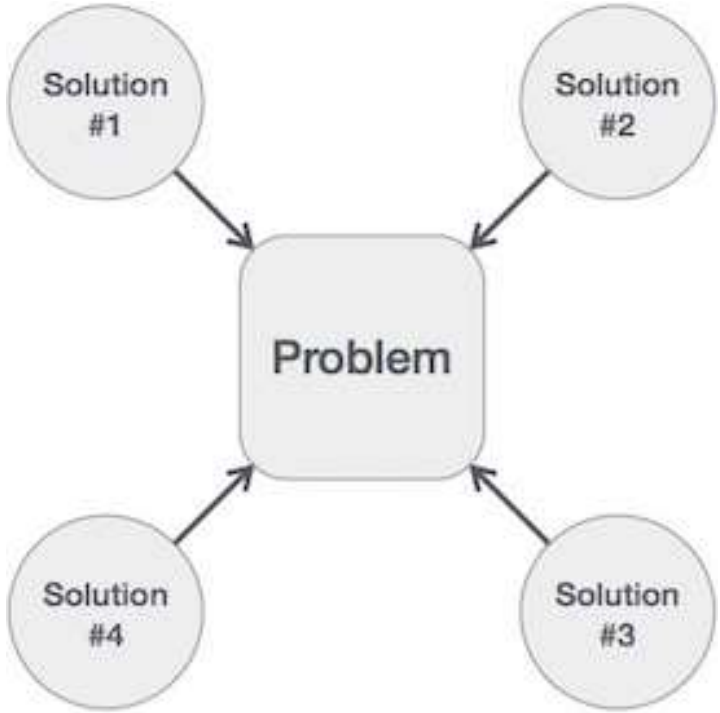
  Equality testing (like == in Java)

Algorithm ***arrayMax***(***A***, ***n***)
  Input: array ***A*** of ***n*** integers
  Output: maximum element of ***A***

```
currentMax ← A[0]
for i ← 1 to n – 1 do
  if A[i] > currentMax then
    currentMax ← A[i]
return currentMax
```

Introduction Analysis of Algorithms

- Design an algorithm to get a solution of a given problem.
- A problem can be solved in more than one ways.
- Many solution algorithms can be derived for a given problem.
- The next step is to analyse those proposed solution algorithms and implement the best suitable solution.



Algorithm Analysis

Algorithm analysis in the context of **data structures** refers to evaluating the efficiency and performance of algorithms used to manipulate and process data stored in various data structures.

The goal is to assess both **time complexity** (how fast an algorithm runs) and **space complexity** (how much memory an algorithm uses).

Understanding these complexities helps in selecting the most efficient algorithm for solving a problem, especially when dealing with large data sets.

Algorithms Analysis (continue)

Efficiency of an algorithm can be analysed at two different stages, before implementation and after implementation.

- **Priori Analysis**

  This is a **theoretical analysis** of an algorithm. Efficiency of an algorithm is measured by assuming that all other factors, for example, processor speed, are constant and have no effect on the implementation.

- **Posterior Analysis**

  This is an **empirical analysis** of an algorithm. The selected algorithm is implemented using programming language. This is then executed on target computer machine. In this analysis, actual statistics like running time and space required, are collected.

Priori Analysis

Algorithm analysis deals with the execution or **running time** of various operations involved.

The algorithm's efficiency is a function of the number of elements to be processed.

f(n) = efficiency

Theoretical Analysis of Running Time

Primitive Operations

Counting primitive operations

Running Time

The **running time** of an operation can be defined as the **number of computer instructions executed per operation**.

Running time is measured as a function of n, the size of the input (in bytes assuming a reasonable encoding).

In the RAM model of computation. All “reasonable” operations take “1” unit of time. (e.g. +, *, -, /, array access, pointer following, writing a value, one byte of I/O...)

The running time of an algorithm typically grows with the input size (analyze running time as a function of input size).

Even on inputs of the same size, running time can be very different. Example: algorithm that finds the first prime number in an array by scanning it left to right

The idea is to analyze running time in the situation of

best case

worst case

average case

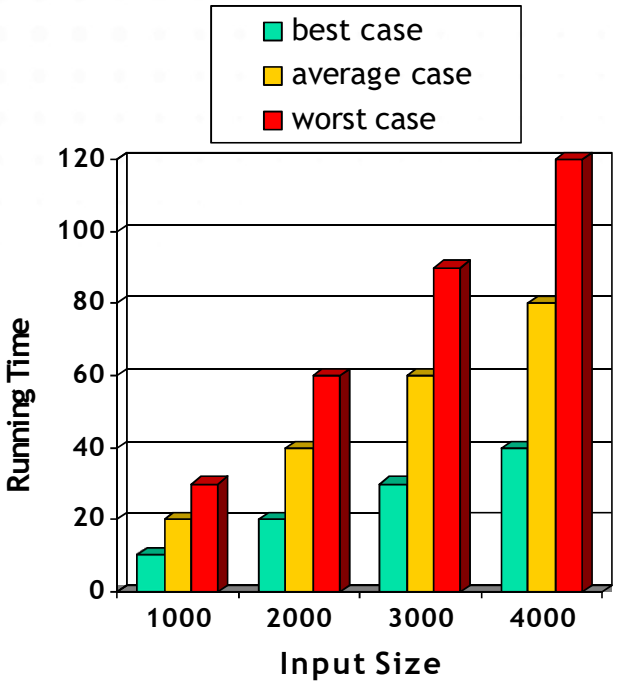
Running Time Evaluation

Best case running time is usually useless.

Average case time is very useful but often difficult to determine.

We focus on the **Worst case** running time because:

- Easier to analyse.
- Crucial to applications such as games, finance and robotic.



Analysis of Algorithms and Asymptotic Notation

Time required by an algorithm falls under three types:

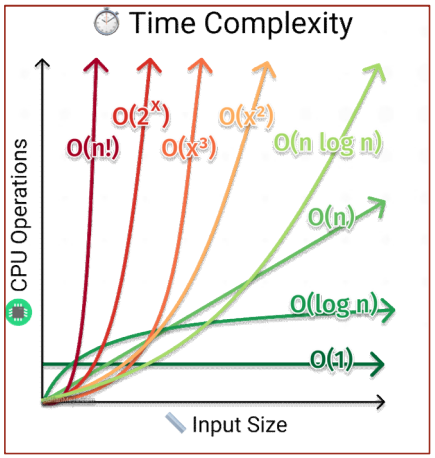
- **Best Case** − Minimum time required for program execution.
- **Average Case** − Average time required for program execution.
- **Worst Case** − Maximum time required for program execution.

An **Asymptotic Notations** are the expressions that are used to represent the complexity of an algorithm.

Types of Data Structure Asymptotic Notation

- Big-O Notation (O) – specifically describes **worst case** scenario.
- Omega Notation (Ω) – specifically describes **best case** scenario
- Theta Notation (θ) – represents the **average case** complexity of an algorithm.

Time Complexity



Time complexity measures how the execution time of an algorithm increases as the size of the input grows. The size of the input is typically denoted by **n**, where **n** could represent the number of elements in a data structure (like an array, list, tree, etc.).

Big O Notation (O): The most common way to express time complexity is using Big O notation. It describes the **upper bound** of an algorithm's growth rate, focusing on the worst-case scenario.

For example:

- **O(1)**: Constant time complexity, meaning the algorithm's execution time does not depend on the size of the input. A simple example is accessing an element in an array by index.
- **O(log n)**: Logarithmic time complexity, common in binary search algorithms and balanced binary search trees (e.g., AVL trees).
- **O(n)**: Linear time complexity, meaning the time taken grows proportionally with the input size. For example, searching through an unsorted list.
- **O(n log n)**: Log-linear time complexity, typical of efficient sorting algorithms like merge sort or quicksort.
- **O(n^2)**: Quadratic time complexity, often found in algorithms with nested loops (e.g., bubble sort).

The **worst-case** and **best-case** scenarios are used to represent time complexities, but it's often the **average-case** complexity that's most practical for real-world use cases.

Primitive Operations

- For theoretical analysis, **primitive** or **basic** operations will be counted for **running time**, which are simple computations performed by an algorithm.
- Basic operations are:
  - Identifiable in pseudocode
  - Largely independent from the programming language
  - The exact definition is not important
  - Assumed to take a constant amount of time in the RAM model

- Examples of primitive operations
  - Evaluating an expression
  - Assigning a value to a variable
  - Indexing into an array
  - Calling a method
  - Returning from a method

x^2+e^y

$cnt \leftarrow cnt+1$

A[5]

mySort(A,n)

return(cnt)

Example 1: Counting Primitive Operation

By inspecting the pseudocode, the maximum number of primitive operations executed can be determined by an algorithm.

Algorithm *arrayMax*(*A*, *n*)	
currentMax $\leftarrow$ ***A***[0]	2
for ***i*** $\leftarrow$ 1 **to** ***n*** $-$ 1 **do**	2+***n***
if ***A***[*i*] $>$ ***currentMax*** **then**	2(***n*** $-$1)
currentMax $\leftarrow$ ***A***[*i*]	2(***n*** $-$1)
{ increment counter ***i*** }	2(***n*** $-$1)
return ***currentMax***	1
Total	7***n*** $-$1

Example 1:Estimating Running Time

- Algorithm *arrayMax* executes $7n-1$ primitive operations in the worst case. Define:
  - a = Time taken by the fastest primitive operation
  - b = Time taken by the slowest primitive operation
- Let $T(n)$ be worst-case time of *arrayMax*. Then

$$a(7n-1) \leq T(n) \leq b(7n-1)$$

- Hence, the running time $T(n)$ is bounded by two linear functions

notes: y = mx + c is a **linear equation**

Discussion 1: Write algorithm to search data in an array

SOLUTION 1:

```
Algorithm LinearSearch(array, target):
    Input:
        array - A list of elements (size n)
        target - The element to be searched
    Output:
        The index of the target element in the array,
        or -1 if the element is not found.

    For i = 0 to length(array) - 1:
        If array[i] == target:
            Return i  // Return the index of the element
    End For

    Return -1  // Target not found
```

Explanation:

Input: An array and a target value to search.

Steps:

- Loop through each element in the array.
- Compare each element with the target.
- If a match is found, return the index of that element.

If the loop completes and no match is found, return -1 to indicate the target is not in the array.

Discussion 1: Write algorithm to search data in an array (continue)

SOLUTION 2:

```
Algorithm BinarySearch(array, target):
    Input:
        array - A sorted list of elements (size n)
        target - The element to be searched
    Output:
        The index of the target element in the array,
        or -1 if the element is not found.

    low = 0
    high = length(array) - 1

    While low <= high:
        mid = (low + high) / 2
        If array[mid] == target:
            Return mid  // Return the index of the element
        Else If array[mid] < target:
            low = mid + 1
        Else:
            high = mid - 1
    End While

    Return -1  // Target not found
```

Explanation:

Input: A sorted array and a target value.

Steps:

Set low to 0 and high to the last index.
Calculate the middle index mid.
Compare the middle element with the target:
- If it's equal, return the index.
- If it's smaller than the target, adjust the low pointer to mid + 1.
- If it's greater than the target, adjust the high pointer to mid - 1.

If no match is found, return -1.

The idea design solution is to:

Initial state: You start with an array of size n.

Iteration 1: You compare the target value with the middle element, reducing the search space to half.

Iteration 2: The search space is halved again.

This process continues, halving the search space at each step.

After about $\log_2(n)$ steps, the search space is reduced to a single element.

Discussion 1: Write algorithm to search data in an array (continue)

SOLUTION 1:

```
Algorithm LinearSearch(array, target):
    Input:
        array - A list of elements (size n)
        target - The element to be searched
    Output:
        The index of the target element in the array,
        or -1 if the element is not found.

    For i = 0 to length(array) - 1:          2+n
        If array[i] == target:               2(n)
            Return i // Return the index of the element   2(n)
    End For

    Return -1 // Target not found            1
```

n

Total = 2+n+2n+2n+2n+1 = 5n+3
f(n)=5n+1
O(n) Linear time complexity

Explanation Time Complexity (Linear Search):

Best Case:
If the target element is at the first position, the search will stop immediately.
- Time complexity: **O(1)** (constant time)

Worst Case:
In the worst case, the target element is either at the end of the array or not in the array at all. The algorithm will need to check every element.
- Time complexity: **O(n)** (where n is the number of elements in the array)

Average Case:
On average, the algorithm will have to check approximately half of the elements in the array.
- Time complexity: **O(n)**

Overall Time Complexity of Linear Search: O(n)

Discussion 1: Write algorithm to search data in an array (continue)

SOLUTION 2:

```
Algorithm BinarySearch(array, target):
    Input:
        array - A sorted list of elements (size n)
        target - The element to be searched
    Output:
        The index of the target element in the array,
        or -1 if the element is not found.

    low = 0                                          1
    high = length(array) - 1                         1
    While low <= high:                               log2(n)
        mid = (low + high) / 2                       log2(n)
        If array[mid] == target:
            Return mid  // Return the index of the element
        Else If array[mid] < target:                 7(log2(n))
            low = mid + 1
        Else:
            high = mid - 1
    End While

    Return -1 // Target not found                    1
```

Total = $1+1+2+\log_2(n)+7\log_2(n)+1 = 5+8\log_2(n)$

$f(n)=8\log_2(n)+1$

O(log(n)) Logarithmic time complexity

Explanation Time Complexity (Binary Search)**:**

Best Case:
If the target element is found in the middle of the array in the first iteration.

- Time complexity: **O(1)** (constant time)

Worst Case:
The array is divided in half each time, and the search continues in one of the halves. This reduces the size of the problem by half in each step. The number of iterations needed is proportional to the logarithm of the number of elements (log n).

- Time complexity: **O(log n)**

Average Case:
The time complexity is similar to the worst case, as the algorithm cuts the search space in half with each step.

- Time complexity: **O(log n)**

Overall Time Complexity of Binary Search: O(log n)

Discussion 1: Write algorithm to search data in an array (continue)

SOLUTION 1:

```
Algorithm LinearSearch(array, target):
    Input:
        array - A list of elements (size n)
        target - The element to be searched
    Output:
        The index of the target element in the array,
        or -1 if the element is not found.

    For i = 0 to length(array) - 1:
        If array[i] == target:
            Return i  // Return the index of the element
    End For

    Return -1  // Target not found
```

SOLUTION 2:

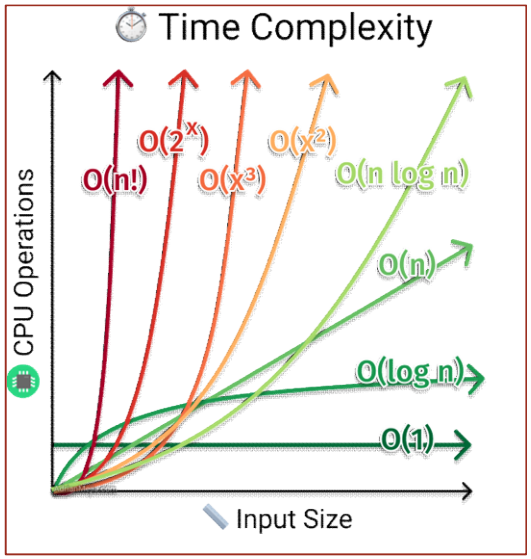
```
Algorithm BinarySearch(array, target):
    Input:
        array - A sorted list of elements (size n)
        target - The element to be searched
    Output:
        The index of the target element in the array,
        or -1 if the element is not found.

    low = 0
    high = length(array) - 1

    While low <= high:
        mid = (low + high) / 2
        If array[mid] == target:
            Return mid  // Return the index of the element
        Else If array[mid] < target:
            low = mid + 1
        Else:
            high = mid - 1
    End While

    Return -1  // Target not found
```

Algorithm	Best Case	Worst Case	Average Case	Time Complexity
Linear Search	O(1)	O(n)	O(n)	**O(n)**
Binary Search	O(1)	O(log n)	O(log n)	**O(log n)**

Summary:

Linear Search: Suitable for unsorted arrays but has a time complexity of **O(n).**

Binary Search: Efficient for sorted arrays with a time complexity of **O(log n).**



Space Complexity

Space complexity is a way to measure how much memory an algorithm needs to run. It tells us the total amount of space or memory an algorithm will use from start to finish.

This includes the space needed for all the variables, data structures, and any extra space used during the execution.

Components of Space Complexity

Fixed Part

This includes the space required for constants, simple variables, fixed-size data structures, and the code itself. It is the memory that does not change regardless of the input size.

Examples:

Space needed for storing constants like numbers or fixed-size arrays.

Memory used by the program code and instructions.

Variable Part

This includes the space required for dynamic memory allocation, recursive stack space, and temporary variables whose size depends on the input size.

Examples:

Space needed for dynamic data structures like **linked lists**, **trees**, or **graphs**, which grow with the input size.

Memory used for the recursion stack during recursive function calls.

Temporary variables and data structures used during the execution of the **algorithm**.

How to Calculate Space Complexity?

Calculating space complexity involves analyzing the memory usage of an algorithm by considering both the fixed and variable parts of memory:

1. Identify the Fixed Part

Determine the memory required for constants, simple variables, and fixed-size data structures.

This part does not change with the size of the input.

2. Identify the Variable Part

Determine the memory required for dynamic data structures, recursion stack space, and temporary variables. This part changes with the size of the input.

3. Sum the Fixed and Variable Parts

Add the memory required for the fixed part and the variable part to get the total space complexity.

Discussion 2: Write algorithm to calculate the sum of all integers from 1 to n

SOLUTION 1:

```
Algorithm: sum_fixed_memory(n)
    Input: n (an integer)
    Output: Sum of integers from 1 to n

    Begin
        sum = n * (n + 1) / 2
        // Calculate the sum using the formula
        Return sum
        // Return the result
    End
```

Solve using Fixed Memory Algorithm (Constant Space Complexity)

This algorithm calculates the sum using a mathematical formula, requiring no extra space besides the input and result.

Explanation:

- The formula n * (n + 1) / 2 directly computes the sum of integers from 1 to n in constant time and space.
- The space complexity is **O(1)** because it uses only a few variables (n, sum).

Discussion 2: Write algorithm to calculate the sum of all integers from 1 to n (continue)

SOLUTION 2:

```
Algorithm: sum_variable_memory(n)
    Input: n (an integer)
    Output: Sum of integers from 1 to n

    Begin
        Create an empty list called numbers
        For i = 1 to n do
            Append i to numbers
            // Add each integer from 1 to n into the list
        End For

        sum = 0
        For each number in numbers do
            sum = sum + number
            // Add each number in the list to sum
        End For

        Return sum              // Return the result
    End
```

Solve using Variable Memory Algorithm (Linear Space Complexity)

This algorithm uses a loop to iterate through all integers from 1 to n, storing them in a list, and then calculates the sum.

Explanation:

- This approach creates a list numbers that stores all integers from 1 to n.
- Then, it sums all the numbers in the list using a loop.
- The space complexity is **O(n)** because it stores n integers in the list.

Discussion 2: Write algorithm to calculate the sum of all integers from 1 to n (continue)

Summary:

- **Fixed Memory Algorithm**:

  Uses the formula $\frac{n(n+1)}{2}$ to calculate the sum in **constant space** (O(1)).

- **Variable Memory Algorithm**:

  Uses a list to store numbers from 1 to n and sums them in **linear space** (O(n)).

Both algorithms achieve the same result, but the fixed memory algorithm is more efficient in terms of space usage.

Common Space Complexities

Space Complexity	Description	Examples
O(1)	Constant space	Finding max in an array
O(n)	Linear space	Storing a list of n elements
O(n^2)	Quadratic space	2D matrix operations
O(log n)	Logarithmic space	Binary search recursion depth
O(n log n)	Linearithmic space	Merge Sort
O(2^n)	Exponential space	Subset sum problem using recursion
O(n!)	Factorial space	Generating all permutations of a string

Space Complexity of Data Structures

Data Structure	Space Complexity	Explanation
Array	O(n)	Space is proportional to the number of elements stored in the array.
Linked List	O(n)	Space is proportional to the number of nodes, each node having data and a pointer.
Stack	O(n)	Space is proportional to the number of elements in the stack.
Queue	O(n)	Space is proportional to the number of elements in the queue.
Hash Table	O(n)	Space is proportional to the number of elements stored, including array and linked lists or other collision handling.
Binary Tree	O(n)	Space is proportional to the number of nodes in the tree.

Data Structure	Space Complexity	Explanation
Binary Search Tree	O(n)	Space is proportional to the number of nodes in the tree.
Balanced Trees (e.g., AVL, Red-Black Tree)	O(n)	Space is proportional to the number of nodes in the tree.
Heap (Binary Heap)	O(n)	Space is proportional to the number of elements in the heap.
Trie	O(n * m)	Space is proportional to the number of words (n) times the average length of the words (m).
Graph (Adjacency Matrix)	O(V^2)	Space is proportional to the square of the number of vertices (V).
Graph (Adjacency List)	O(V + E)	Space is proportional to the number of vertices (V) plus the number of edges (E).

Analyzing Algorithm Efficiency

Algorithm analysis doesn't just focus on raw time or space complexity.

It also takes into account factors like:

Scalability: How does the algorithm perform as the input size increases?

Best, Worst, and Average Case: Which scenario is most relevant for your problem? Many algorithms perform differently based on input conditions.

Amortized Analysis: For some algorithms, individual operations might take a varying amount of time, but over a series of operations, the average time per operation may be more predictable. For instance, in dynamic array resizing, most insertions are O(1), but occasional resizing is O(n). The amortized time per insertion is O(1).

Choosing the Right Data Structure

The choice of data structure can significantly affect algorithm performance.

For example:

A **hash table** might be ideal for searching, inserting, and deleting elements quickly.

A **binary search tree** (BST) or **heap** might be better for maintaining a sorted collection.

Linked lists are often used when you need frequent insertions and deletions but can tolerate slower searches.

By analyzing the time and space complexities, you can make informed decisions about which algorithm to use based on the problem's requirements.

Practical Considerations

When analyzing algorithms in real-world applications, other factors come into play:

Cache efficiency: Accessing memory sequentially is usually faster due to how caches work.

Constant factors: Though Big O focuses on the growth rate, smaller constant factors can have a significant impact on performance.

Parallelism: Some algorithms may be parallelizable, which can reduce time complexity in practice (though this is not always reflected in Big O).

CONCLUSION

- Algorithm analysis in data structures is essential for designing efficient software.
- By understanding the time and space complexities of different algorithms and data structures, you can choose the most suitable approach for solving a problem.
- This involves considering the size of the input, the required performance, and the trade-offs between different algorithms to optimize both time and space.

UTM
UNIVERSITI TEKNOLOGI MALAYSIA

ANOTHER EXTRA NOTE ON HOW TO CALCULATE THE TIME COMPLEXITY

O(1) — Constant Time

Constant time algorithms will always take same amount of time to be executed.

The execution time of these algorithm is independent of the size of the input.

Example of O(1) time is

- accessing a value with an array index. Example: int a = ARR[5];
- Inserting a node in Linked List
- Pushing and Poping on Stack
- Insertion and Removal from Queue
- Finding out the parent or left/right child of a node in a tree stored in Array
- Jumping to Next/Previous element in Doubly Linked List

```
1   i = 1, n =1000,
    b=86
2   Create Stack s
3   s.push(b)
```

O(n) - Linear time complexity

An algorithm has a linear time complexity if the time to execute the algorithm is directly proportional to the input size *n*.

Therefore the time it will take to run the algorithm will increase proportionately as the size of input *n* increases.

Example of O(n) time is

- Traversing an array
- Traversing a linked list
- Linear Search
- Deletion of a specific element in a Linked List (Not sorted)
- Comparing two strings

Example 1 : Linear loop

Assume i is an integer and given n is 1000

How many times the body of the loop is repeated?

Answer: 1000 times

because the efficiency is directly proportional to the numbers of iterations.

It is

f(n) = n, its O(n)

```
n =1000
for ( i=1; i <= n; i++ )
{
    Application code
}
```

Loop 1 : i = 1 = 1

i = i +1

Loop 2 : i = 1+1 = 2

Loop 3 : i = 2+1 = 3

Loop 4 : i = 3+1 = 4

:

Loop k : i = k

Means, i =k where k <= n

k = n

f(n)=n

O(n)

```
1     i = 1, n =1000
2     loop ( i <= n )
        1     Application code
        2     i = i + 1
3     end loop
```

The loop iterates from 1 to n.

The loop increments the variable I by 1 in each iteration.

Therefore, the time complexity of the code is linear time complexity, O(n), where n value is the value of the input parameter n

O(n) - Linear time complexity (cont.)

```
1       i = 1, n =1000
2       loop ( i <= n )
        1       Application code
        2       i = i + 2
3       end loop
```

```
n =1000
for (  i=1;  i <= n;  i=i+2 ) {
      Application code
}
```

Example 2 : Linear loop

Assume i is an integer and given n is 1000

How many times the body of the loop is repeated?

Answer: 500 times

Loop 1 : i=1 = 1 <= (1 *2)-1

i = i +2

Loop 2 : i=1+2 = 3 <= (2 *2)-1

Loop 3 : i=3+2 = 5 <= (3 *2)-1

Loop 4 : i=5+2 = 7 <= (4 *2)-1

:

Loop k: i= (k *2)-1

Means, i =(k*2)-1 where (k*2)-1 <= n

=> k*2-1 = n

=> 2k = n+1

=> k = (n+1)/2

=> $k = \frac{n}{2} + \frac{1}{2}$

$k = \frac{n}{2} + \frac{1}{2}$

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

$O(n) = \frac{n}{2}$

Loop executes n/2 times, so take the least upper bound n. Its O(n)

The loop iterates from 1 to n with a step size 2. which means the loop will run n/2 times.

Therefore, the time complexity of the code is linear time complexity, O(n/2),which simplifies to O(n) as the constant factor is ignored in Big O notation

O(log n) - Logarithmic time complexity

An algorithm has logarithmic time complexity if the time it takes to run the algorithm is proportional to the logarithm of the input size *n*.

Example of O(log n) time is

- Binary Search
- Finding largest/smallest number in a binary search tree
- Certain Divide and Conquer Algorithms based on Linear functionality
- Calculating Fibonacci Numbers - Best Method The basic premise here is NOT using the complete data, and reducing the problem size with every iteration

Example 3 : multiply loops

How many times the body of the loop is repeated?

Answer: 9 times

Loop 1: i= 1 $= 2^{(1-1)}$

Loop 2: i= 2 $= 2^{(2-1)}$

Loop 3: i= 4 $= 2^{(3-1)}$

Loop 4: i= 8 $= 2^{(4-1)}$

:

Loop k: i= 2^{k-1}

Means, i= 2^{k-1} where 2^{k-1}<= n

=> 2^{k-1} = n

=> k-1 = $log_2 n$

=> k = $log_2 n$+1

f(n) = $log_2 n$+1

O(n)= $log\ n$

Loop executes $log_2 n$ times, Its O($log\ n$)

```
1       int i, n=1000
2       for (i = 1;  i<=n;  i=i*2)
          1       Application code
3       end for
```

```
n =1000
for (  i=1;  i <= n;  i=i*2 ) {
      Application code
}
```

The loop iterates from 1 to n by doubling the value of i in each iteration.

The doubling behavior indicates a logarithmic time complexity.

Therefore, the time complexity of the code is logarithmic time complexity, O(log n)

O(n^2) - Quadratic time complexity

An algorithm has quadratic time complexity if the time to execution it is proportional to the square of the input size.

Example of O(n^2) time is

- Bubble Sort
- Insertion Sort
- Selection Sort
- Traversing a simple 2D array

Example 4 : Quadratic Loops

Assume i is an integer and given n is 10

How many times the body of the loop is repeated?

Answer:

outer loop iteration: 10 times

Total iteration in nested loop: 100 times

The nested loop iterates from 1 to n. Therefore, the time complexity of the code is quadratic time complexity, O(n^2) because the number of iterations is proportional to the square of the input size.

Inner loop Iteration:

Loop 1: i = 1+0 = 1

Loop 2: i = 1+1 = 2

Loop 3: i = 2+1 = 3

Loop 4: i = 3+1 = 4

:

Loop k: i = k

i = k <= n

=> k = n

Iteration in loops that increment determined by

O(n)

Outer loop Iteration:

Loop 1: i = 1+0 = 1

Loop 2: i = 1+1 = 2

Loop 3: i = 2+1 = 3

Loop 4: i = 3+1 = 4

:

Loop k: i = k

i = k <= n

=> k = n

Iteration in loops that increment determined by

O(n)

Refer to code on the right

inner loops iteration : n times each loop

outer loop iteration : n times

Total iteration in nested lopp : n * n

because the efficiency is directly proportional to the numbers of iterations. It is

$$f(n) = n^2$$

$$O(n) = n^2$$

```
1       i = 1, n=10
2       loop ( i <= n )
    1       j = 1
    2       loop ( j <= n )
                1       Application code
                2       j = j + 1
    3       end loop
    4       i = i +1
3       end loop
```

Introduction Analysis of Algorithms (cont.)

- Algorithm's efficiency is a function of the number of elements to be processed.

  f(n) = efficiency

  f(n) = n

- We measure as a function of n, and ignore low order terms.

$5n^3 + n - 6$ **becomes** n^3

$8n \log n - 60n$ **becomes** $n \log n$

$2^n + 3n^4$ **becomes** 2^n

UTM
UNIVERSITI TEKNOLOGI MALAYSIA

EXERCISE

UNIVERSITI TEKNOLOGI MALAYSIA

SPACE UTM KL

Exercise 1: Calculate Time Complexity

What is the time complexity for the following program?

```
void fun(int n){
   int i, j, k, count=0;
   for (i=n/2; i<=n; i++)
      for (j=1; j<=n; j=j*2)
         for (k=1; k<=n; k++)
               count++;
}
```

Exercise 2: Calculate Time Complexity

What is the time complexity for the following program?

```
void fun(int n){
   int i, j, k, count=0;
   for (i=0; i<=n; i++)
      for (j=1; j<=n; j=j*4)
         for (k=0; k<=j; k=k+2)
               count++;
}
```