
Data Structures & Algorithms

# Time and Space Complexity


**B.Bhuvaneswaran, AP (SG) / CSE**

9791519152

bhuvaneswaran@rajalakshmi.edu.in

RAJALAKSHMI ENGINEERING COLLEGE

An AUTONOMOUS Institution

Affiliated to ANNA UNIVERSITY, Chennai

# Time Complexity

- The time complexity of an algorithm is the amount of computer time it needs to run to completion.

# Methods

- Experimental method
- Counter Method
- Table Method

# Experimental Method

- The value of $t_p(n)$ for any given n can be obtained only experimentally.
- The program is typed, compiled, and run on a particular machine.
- The execution time is physically clocked, and $t_p(n)$ obtained.

# Limitations

- Even with this experimental approach, one could face difficulties.
- In a multiuser system, the execution time depends on such factors as system load, the number of other programs running on the computer at the time program P is run, the characteristics of these other programs, and so on.

# Program Steps

- The number of steps any program statement is assigned depends on the kind of statement.
- For example,
  - comments count as zero steps;
  - an assignment statement which does not involve any calls to other algorithms is counted as one step;
  - in an iterative statement such as the for, while, and repeat-until statements, we consider the step counts only for the control part of the statement.

# Number of Steps

| Statement | Step Count |
|---|---|
| Comments and declarative | 0 step |
| Assignment | 1 step |
| Conditional | 1 step |
| Loop condition<br>(for, while - n times) | (n + 1) steps |
| Body of the loop | n steps |

# Methods

- We can determine the number of steps needed by a program to solve a particular problem instance in one of two ways:
  - Counter method
  - Table method

# Counter Method

- We introduce a new variable, count, into the program.
- This is a global variable with initial value 0.
- Statements to increment count by the appropriate amount are introduced into the program.
- This is done so that each time a statement in the original program is executed, count is incremented by the step count of that statement.

# Example

```
void Swap(int a, int b)
{
        int t;
        t = a;
        a = b;
        b = t;
}
```

Ans: 3

# Example

```
void Swap(int a, int b)
{
        int t;
        /* count is global, it is initially zero */
        t = a;
        count = count + 1; /* For assignment */
        a = b;
        count = count + 1; /* For assignment */
        b = t;
        count = count + 1; /* For assignment */
}
```

# Example

```
void Initialize(int a[], int n)
{
        int i;
        for (i = 0; i < n; i++)
        {
                a[i] = 0;
        }
}
```

# Example

```
void Initialize(int a[], int n)
{
        int i;
        /* count is global, it is initially zero */
        for (i = 0; i < n; i++)
        {
                count = count + 1; /* For for */
                a[i] = 0;
                count = count + 1; /* For assignment */
        }
        count = count + 1; /* For last time of for */
}
```

Ans: 2n + 1

# Example

```
int Sum(int a[], int n)
{
        int s, i;
        for (i = 0; i < n; i++)
        {
                s = a + a[i];
        }
        return s;
}
```

# Example

```
int Sum(int a[], int n)
{
        int s, i;
        s = 0;
        count = count + 1; /* count is global, it is initially zero */
        for (i = 0; i < n; i++)
        {
                count = count + 1; /* For for */
                s = a + a[i];
                count = count + 1; /* For assignment */
        }
        count = count + 1; /* For last time of for */
        count = count + 1; /* For the return */
        return s;
}
```

Ans: 2n + 3

# Table Method

- The second method to determine the step count of an algorithm is to build a table in which we list the total number of steps contributed by each statement.
- This figure is often arrived at by first determining the number of steps per execution (s/e) of the statement and the total number of times (i.e., frequency) each statement is executed.
- The s/e of a statement is the amount by which the count changes as a result of the execution of that statement.
- By combining these two quantities, the total contribution of each statement is obtained.
- By adding the contributions of all statements, the step count for the entire algorithm is obtained.

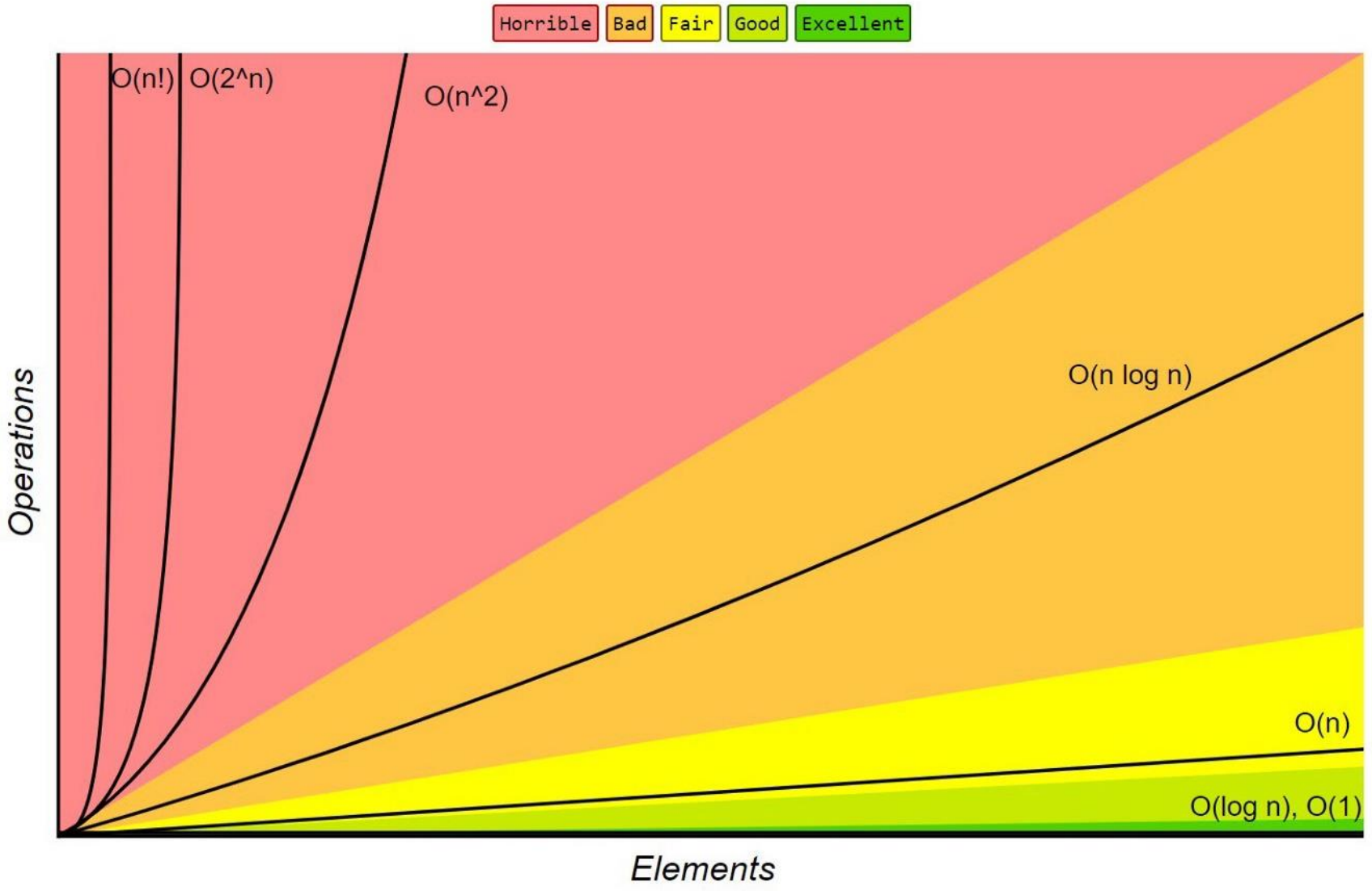
# Step Table

| Line no. | Statement | s/e | frequency | total steps |
|---|---|---|---|---|
| 1 | Algorithm Sum(a, n) | 0 | - | 0 |
| 2 | { | 0 | - | 0 |
| 3 | s := 0.0; | 1 | 1 | 1 |
| 4 | for i := 1 to n do | 1 | n + 1 | n + 1 |
| 5 | s := s + a[i]; | 1 | n | n |
| 6 | return s; | 1 | 1 | 1 |
| 7 | } | 0 | - | 0 |
| | | | Total | 2n + 3 |

# Step Table

| Line no. | Statement | s/e | frequency | total steps |
|---|---|---|---|---|
| 1 | Algorithm Add(a, b, c, m, n) | 0 | - | 0 |
| 2 | { | 0 | - | 0 |
| 3 | for i := 1 to m do | 1 | m + 1 | m + 1 |
| 4 | for j := 1 to n do | 1 | m (n + 1) | mn + m |
| 5 | c[i, j] := a[i, j] + b[i, j]; | 1 | mn | mn |
| 6 | } | 0 | - | 0 |
| | | | Total | 2mn+2m+1 |

# Time Complexity Chart



# Commonly used Rate of Growth

| Time Complexity | Name |
|---|---|
| O(1) | Constant |
| log n | Logarithmic |
| n | Linear |
| n log n | Linear Logarithmic |
| $n^2$ | Quadratic |
| $n^3$ | Cubic |
| $c^n$ | Exponential |

# Space Complexity

- The space complexity of an algorithm is the amount of memory it needs to run to completion.

# Space Complexity

- The space needed by each of these algorithms is seen to be the sum of the following components:
  - A fixed part
  - A variable part

# Fixed Part

- A fixed part that is independent of the characteristics (e.g., number, size) of the inputs and outputs.
- This part typically includes the instruction space (i.e., space for the code), space for simple variables and fixed-size component variables (also called aggregate), space for constants, and so on.

# Variable Part

- A variable part that consists of the space needed by component variables whose size is dependent on the particular problem instance being solved, the space needed by referenced variables (to the extent that this depends on instance characteristics), and the recursion stack space (in so far as this space depends on the instance characteristics).

# Space Complexity

- The space requirement S(P) of any algorithm P may therefore be written as

$$S(P) = c + S_p \text{ (instance characteristics)}$$

- where c is a constant.

# Arrays (dynamic array/list)

- Given n = arr.length →
- Add or remove element at the end: O(1) amortized
- Add or remove element from arbitrary index: O(n)
- Access or modify element at arbitrary index: O(1)
- Check if element exists: O(n)
- Two pointers: O(n·k), where k is the work done at each iteration, includes sliding window
- Building a prefix sum: O(n)
- Finding the sum of a subarray given a prefix sum: O(1)

# Strings (immutable)

- Given n = s.length →
- Add or remove character: O(n)
- Access element at arbitrary index: O(1)
- Concatenation between two strings: O(n+m), where m is the length of the other string
- Create substring: O(m), where m is the length of the substring
- Two pointers: O(n·k), where k is the work done at each iteration, includes sliding window
- Building a string from joining an array, stringbuilder, etc.: O(n)

# Linked Lists

- Given n as the number of nodes in the linked list →
- Add or remove element given pointer before add/removal location: O(1)
- Add or remove element given pointer at add/removal location: O(1) if doubly linked
- Add or remove element at arbitrary position without pointer: O(n)
- Access element at arbitrary position without pointer: O(n)
- Check if element exists: O(n)
- Reverse between position i and j: O(j−i)
- Detect a cycle: O(n) using fast-slow pointers or hash map

# Hash table/dictionary

- Given n = dic.length →
- Add or remove key-value pair: O(1)
- Check if key exists: O(1)
- Check if value exists: O(n)
- Access or modify value associated with key: O(1)
- Iterate over all keys, values, or both: O(n)
- Note: the O(1) operations are constant relative to n. In reality, the hashing algorithm might be expensive. For example, if your keys are strings, then it will cost O(m) where m is the length of the string. The operations only take constant time relative to the size of the hash map.

# Set

- Given n = set.length →
- Add or remove element: O(1)
- Check if element exists: O(1)

# Stack

- Stack operations are dependent on their implementation. A stack is only required to support pop and push. If implemented with a dynamic array:
- Given n = stack.length →
- Push element: O(1)
- Pop element: O(1)
- Peek (see element at top of stack): O(1)
- Access or modify element at arbitrary index: O(1)
- Check if element exists: O(n)

# Queue

- Queue operations are dependent on their implementation. A queue is only required to support dequeue and enqueue. If implemented with a doubly linked list:
- Given n = queue.length →
- Enqueue element: O(1)
- Dequeue element: O(1)
- Peek (see element at front of queue): O(1)
- Access or modify element at arbitrary index: O(n)
- Check if element exists: O(n)

# Binary tree problems (DFS/BFS)

- Given n as the number of nodes in the tree →
- Most algorithms will run in O(n·k) time, where k is the work done at each node, usually O(1).
- This is just a general rule and not always the case.
- We are assuming here that BFS is implemented with an efficient queue.

# Binary search tree

- Given n as the number of nodes in the tree →
- Add or remove element: O(n) worst case, O(logn) average case
- Check if element exists: O(n) worst case, O(logn) average case
- The average case is when the tree is well balanced - each depth is close to full. The worst case is when the tree is just a straight line.

# Heap/Priority Queue

- Given n = heap.length and talking about min heaps →
- Add an element: O(logn)
- Delete the minimum element: O(logn)
- Find the minimum element: O(1)
- Check if element exists: O(n)

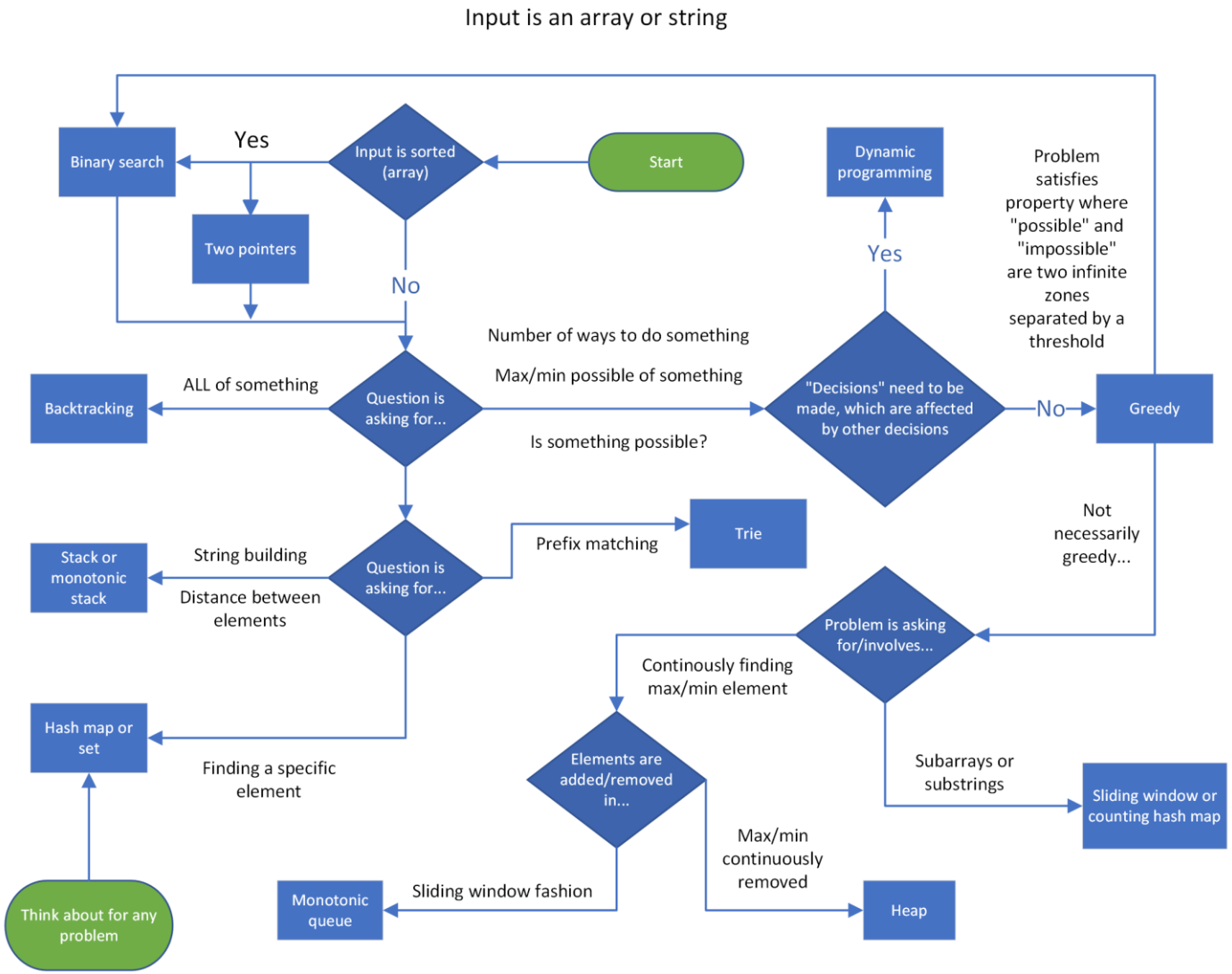
# Binary search

- Binary search runs in O(logn) in the worst case, where n is the size of your initial search space.

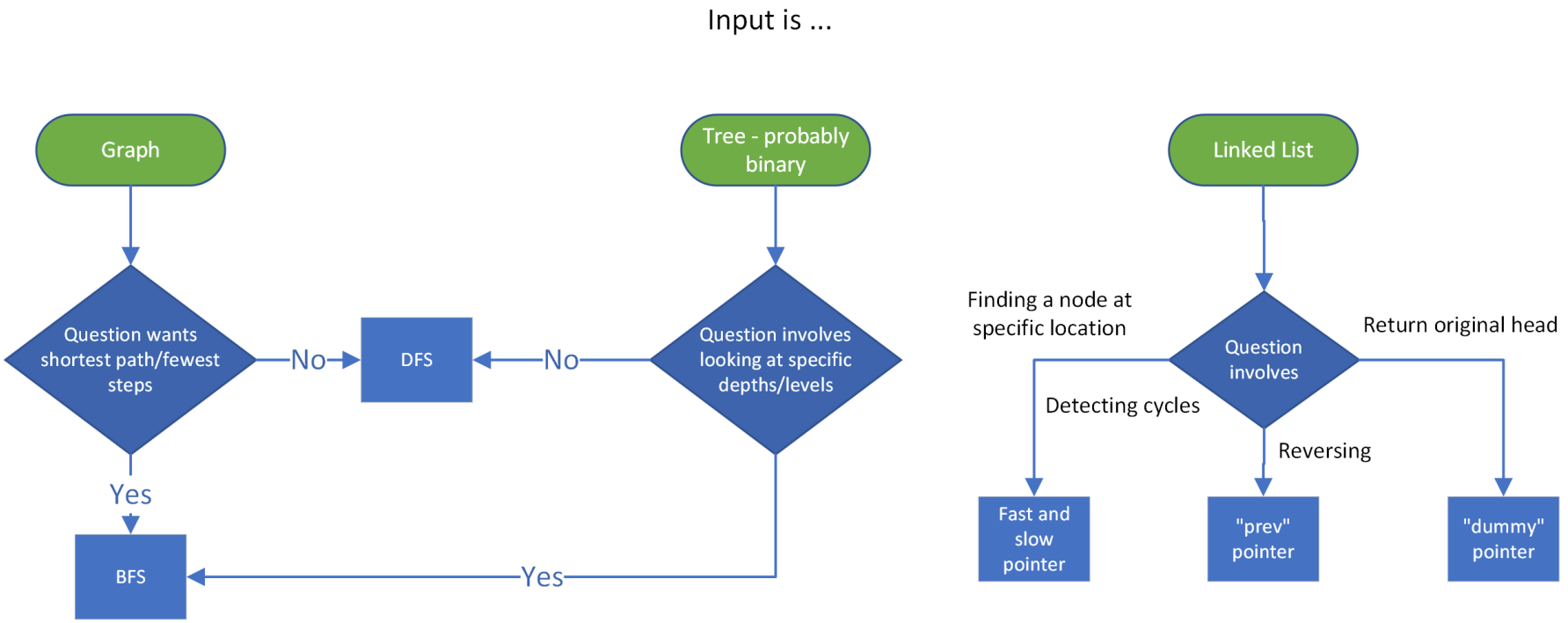
# Miscellaneous

- Sorting: O(n·logn), where n is the size of the data being sorted
- DFS and BFS on a graph: O(n·k+e), where n is the number of nodes, e is the number of edges, if each node is handled in O(1) other than iterating over edges
- DFS and BFS space complexity: typically O(n), but if it's in a graph, might be O(n+e) to store the graph
- Dynamic programming time complexity: O(n·k), where n is the number of states and k is the work done at each state
- Dynamic programming space complexity: O(n), where n is the number of states

# General DS/A flowchart



# General DS/A flowchart



# Queries?

# Thank You…!