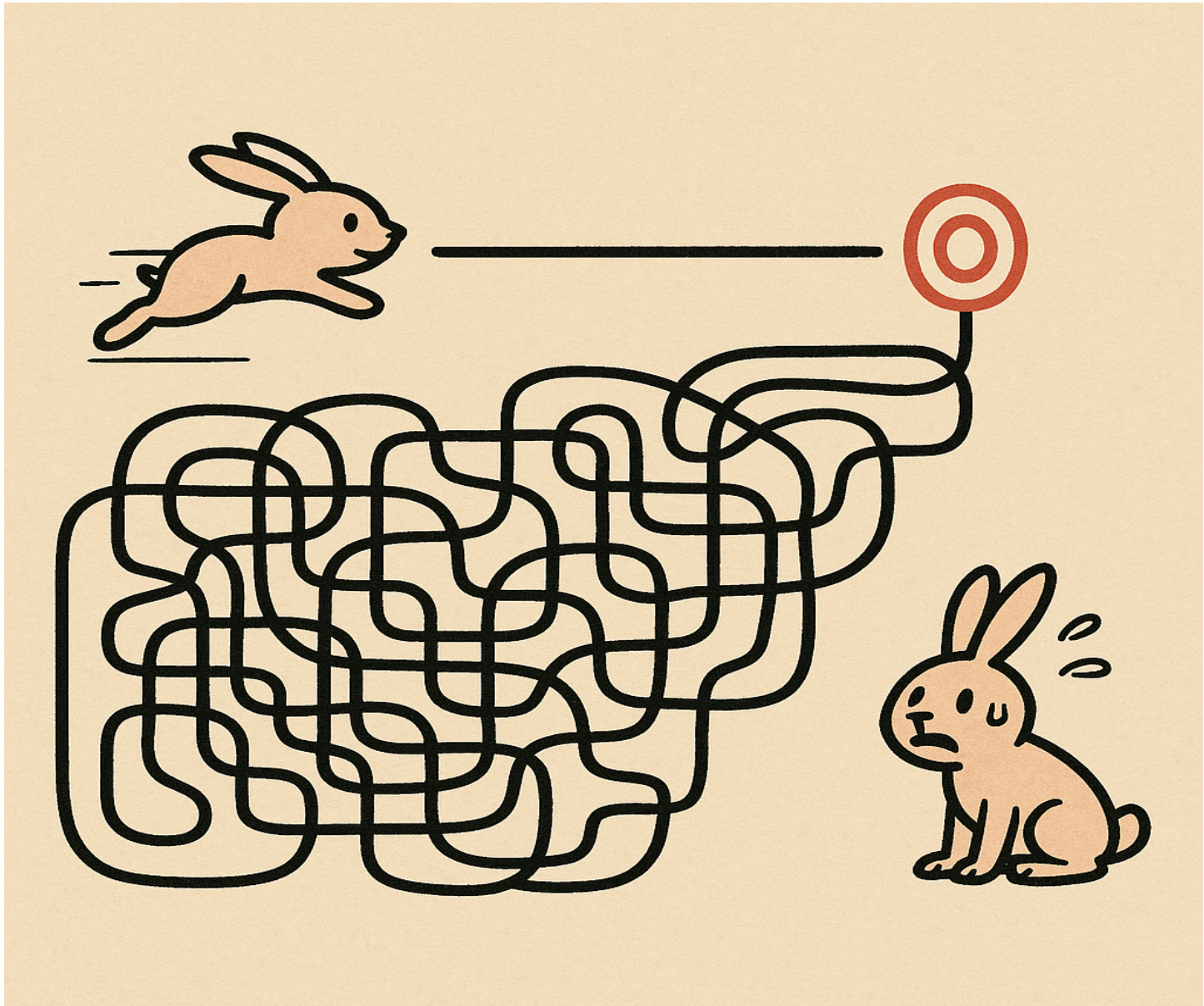
# Programming and Data Structures - I

Lecture 16

**Kripabandhu Ghosh**

CDS, IISER Kolkata

# ANALYSIS OF ALGORITHMS



# Complexity (efficiency) of Algorithms

- When is a program/algorithm better than another?
- What is the notion of the **best** algorithm?

## Measures

- Time required (Time Complexity)
  - Execution time (not compilation time)
- Space required (Space Complexity)
  - Program size (code length)
  - Data size (variables etc. used)

Time complexity is considered more important

# Machine independent time complexity

- It is difficult to compare algorithms across machines
- Need for a machine-independent measure

## Characteristics of a machine-independent measure

- All the instruction steps (assignment, operation, comparison etc.) will take the same amount of time (e.g. unit time)
- Time required is proportional to the input size
- Time complexity is meaningful for large input size (Asymptotic Comparison)

# Big Oh ($\mathcal{O}$) notation

## Definition

$\mathcal{O}(g(n)) = \{f(n)$ : there exists positive constants $c$ and $n_0$ such that $0 \leq f(n) \leq c^*g(n)$ for all $n \geq n_0\}$

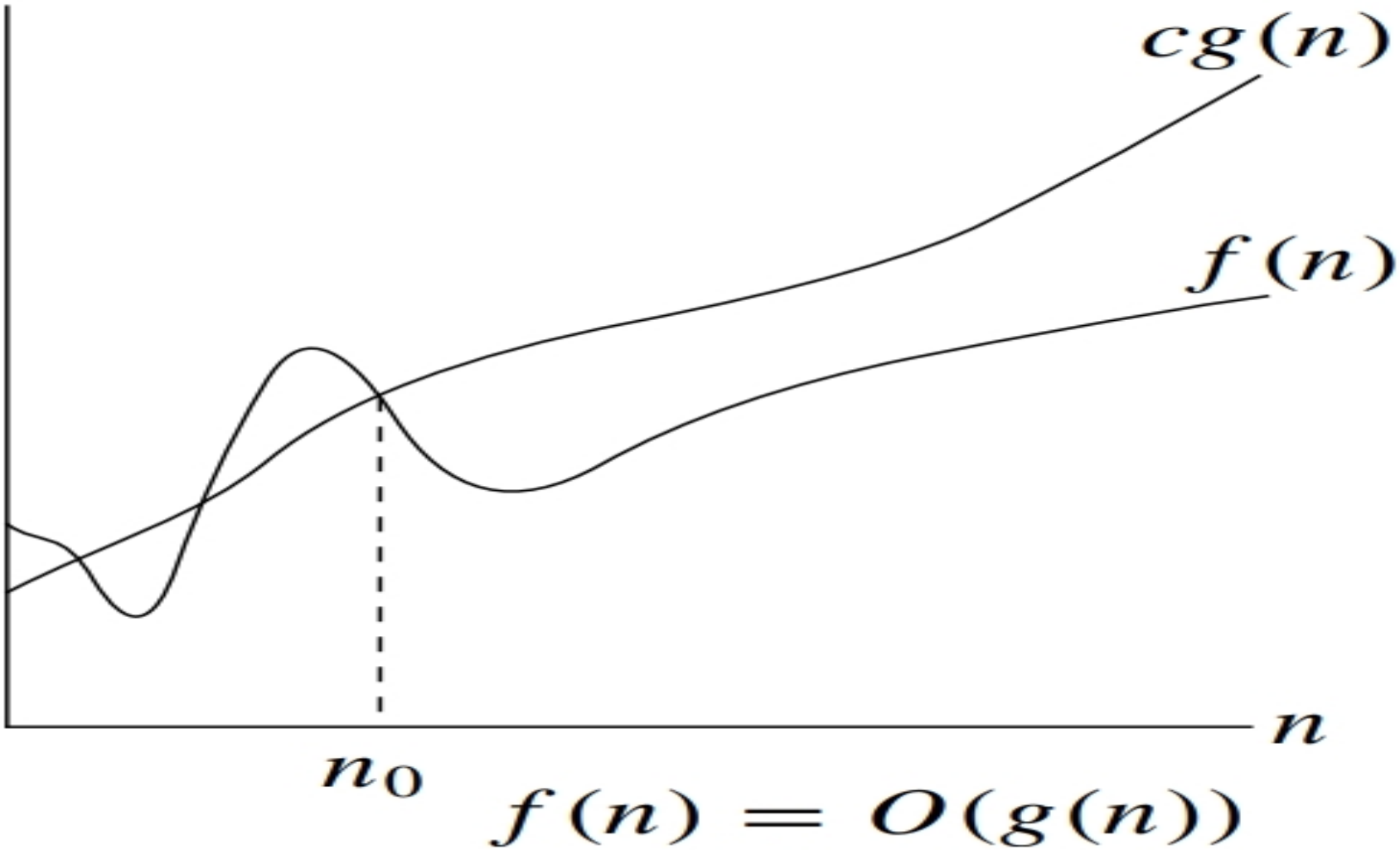
- $f(n) = \mathcal{O}(g(n))$ means $f(n) \in \mathcal{O}(g(n))$
- $\mathcal{O}$ gives an asymptotic upper bound of $f(n)$
- Worst case time complexity

## Example

- $f(n) = 3n^2 + 5n + 7$, $g(n) = n^2$
- $f(n) = \mathcal{O}(g(n)) = \mathcal{O}(n^2)$ $[f(n) \leq c^*g(n)$ for $c = 4$, $n_0 = 7]$[a]

[a] https://docs.google.com/spreadsheets/d/1GKriyllUZMKv-9xgLXqttLZ3dv5YNwk4iQ86APwE-JI/edit?usp=sharing

# Big Oh ($\mathcal{O}$) notation[1]




[1] Introduction to Algorithms, 3rd Edition. Cormen et al.

# Theta ($\Theta$) notation

## Definition

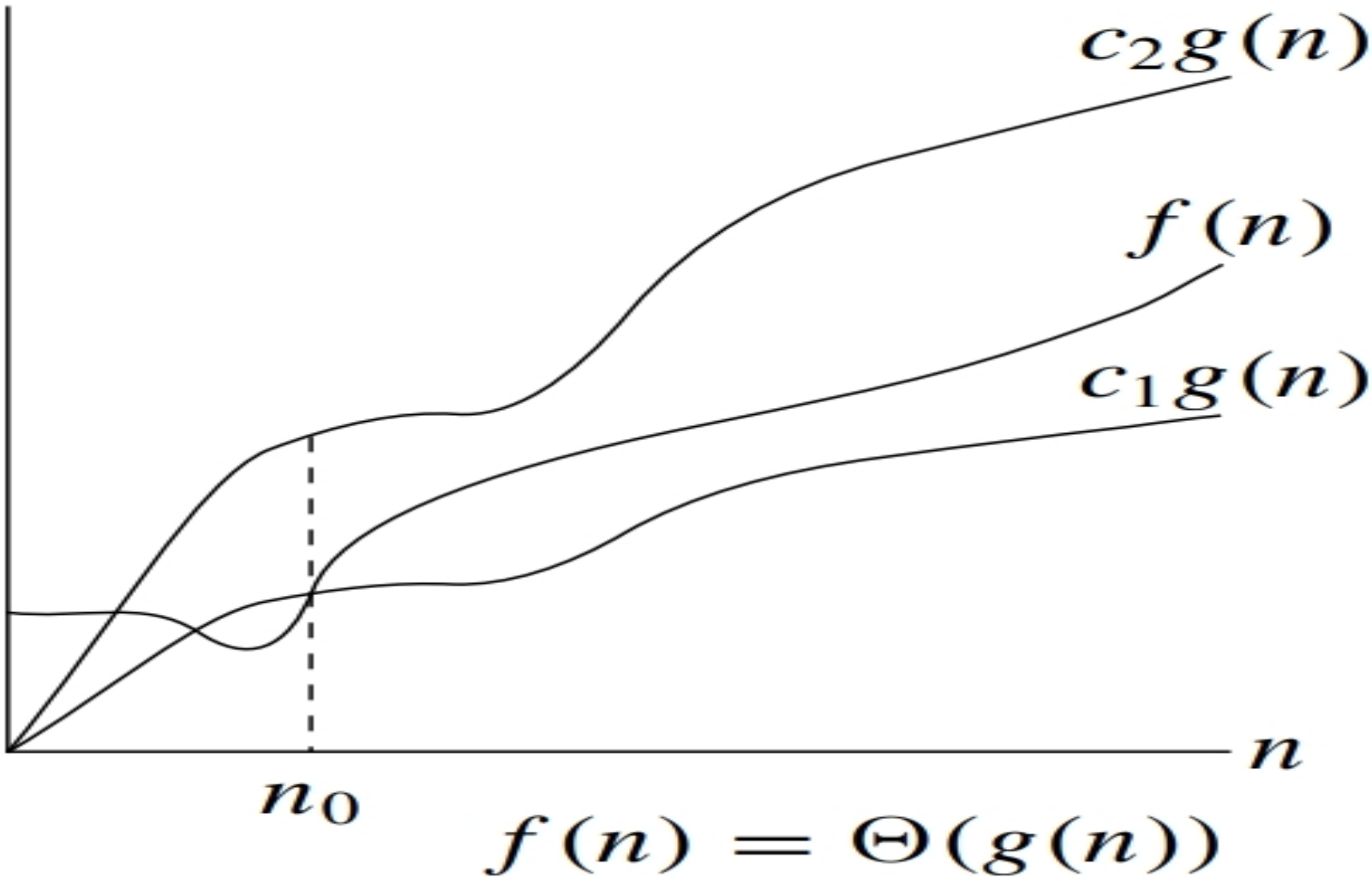
$\Theta(g(n)) = \{f(n)$ : there exists positive constants $c_1$, $c_2$ and $n_0$ such that $0 \leq c_1{*}g(n) \leq f(n) \leq c_2{*}g(n)$ for all $n \geq n_0\}$

- $f(n) = \Theta(g(n))$ means $f(n) \in \Theta(g(n))$
- $\Theta$ gives an asymptotic tight bound of $f(n)$

## Example

- $f(n) = 3n^2 + 5n + 7$, $g(n) = n^2$
- $f(n) = \Theta(g(n)) = \Theta(n^2)$ [$c_1{*}g(n) \leq f(n) \leq c_2{*}g(n)$ for $c_1 = 1$, $c_2 = 5$ $n_0 = 7$]

# Theta ($\Theta$) notation[2]




[2]Introduction to Algorithms, 3rd Edition. Cormen et al.

# Asymptotic Equivalence

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$$

($c > 0$ is a finite number)

$f$ and $g$ are of the same time complexity

## Example of equivalence

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \frac{3n^2 + 5n + 7}{n^2} = 3$$

# Omega ($\Omega$) notation

## Definition

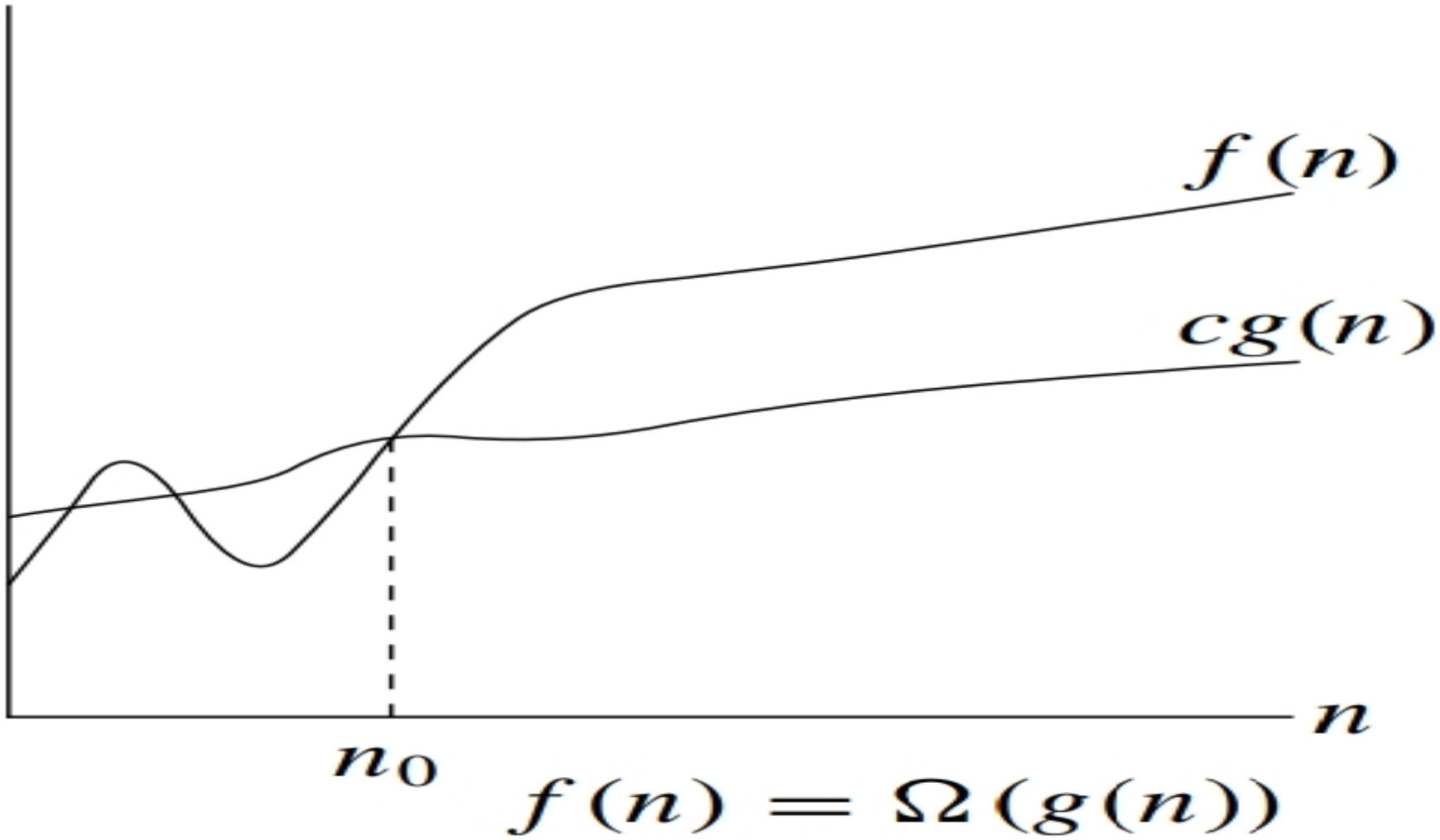
$\Omega(g(n)) = \{f(n)$ : there exists positive constants $c$ and $n_0$ such that $0 \leq c^*g(n) \leq f(n)$ for all $n \geq n_0\}$

- $f(n) = \Omega(g(n))$ means $f(n) \in \Omega(g(n))$
- $\Omega$ gives an asymptotic lower bound of $f(n)$

## Example

- $f(n) = 3n^2 + 5n + 7$, $g(n) = n^2$
- $f(n) = \Omega(g(n)) = \Omega(n^2)$ $[c^*g(n) \leq f(n)$ for $c = 1$, $n_0 = 1]$

# Omega ($\Omega$) notation[3]




[3] Introduction to Algorithms, 3rd Edition. Cormen et al.

# Example: Linear Search

## Code

```
int linear_search(int *a, int n, int key)
{
   int i;
   for(i = 0 ; i < n ; i = i + 1)
   {
      if(a[i] == key)
      {
         break;
      }
   }
   if(a[i] == key)
   {
      return 1;
   }
   else
   {
      return 0;
   }
}
```

# Linear Search : Time complexity

| Step | Type | Time |
|---|---|---|
| int *a | assignment | 1 |
| int n | assignment | 1 |
| int key | assignment | 1 |
| i = 0 | assignment | 1 |
| i < n | comparison | n + 1 |
| i = i + 1 | increment and assignment | 2*n |
| a[i] == key | comparison | n + 1 |
| break | statement | 1 |
| return 1 | statement | 1 |
| return 0 | statement | 1 |
| TOTAL | ALL | 4*n + 9 |

# Linear Search : Time complexity

| Step | Type | Time |
|---|---|---|
| int *a | assignment | 1 |
| int n | assignment | 1 |
| int key | assignment | 1 |
| i = 0 | assignment | 1 |
| i < n | comparison | n + 1 |
| i = i + 1 | increment and assignment | 2*n |
| a[i] == key | comparison | n + 1 |
| break | statement | 1 |
| return 1 | statement | 1 |
| return 0 | statement | 1 |
| TOTAL | ALL | 4*n + 9 |

## Time complexity

$$T(n) = 4*n + 9 = \mathcal{O}(n)$$

# Binary search: Algorithm

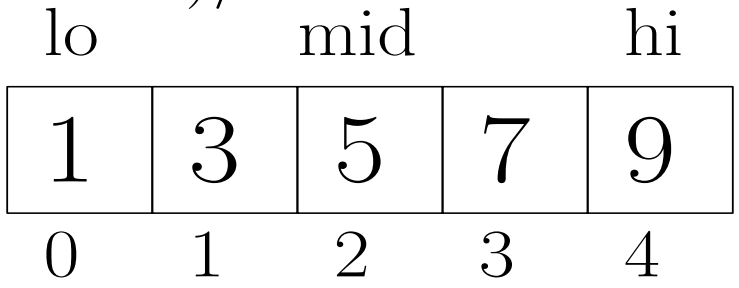
## Code

```
int bin_search(int *a, int n, int key)
{
  int mid, hi, lo;
  lo = 0;
  hi = n - 1;
  mid = (lo + hi)/2;
  while(lo <= hi)
  {
    if(a[mid] == key)
    {
      break;
    }
    else
    {
      if(key < a[mid])
      {
        hi = mid -1;
      }
      else
      {
        lo = mid + 1;
      }
    }
    mid = (lo + hi)/2;
  }
  if(a[mid] == key)
  {
    return 1;
  }
  else
  {
    return 0;
  }
}
```

# Binary search: illustration

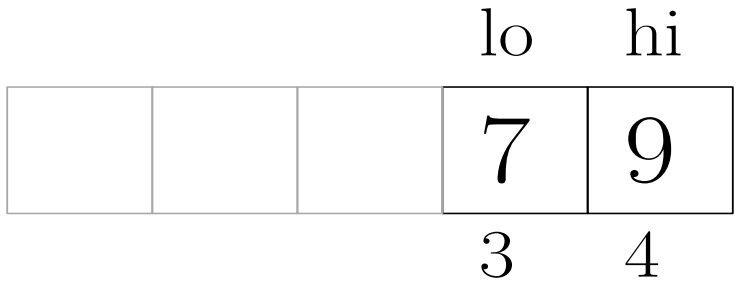
$\text{mid} = (\text{lo} + \text{hi})/2 = 2$

key = 9



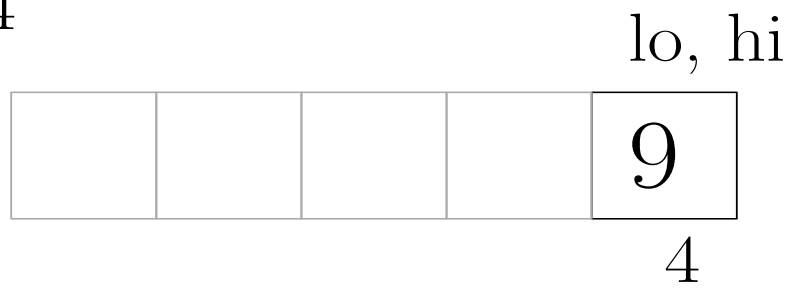

A[mid] = key ? NO
key > A[mid] ? YES
lo = mid + 1

mid = 3




A[mid] = key ? NO
key > A[mid] ? YES
lo = mid + 1

mid = 4




A[mid] = key ? YES

SUCCESS!

# Binary search: loop time requirement

## Notations

- $n$: Length of the array
- $k$: Number of times the loop runs

## Deduction

- $\frac{n}{2^k} = 1$ (Until the search space reduces to a single element)

# Binary search: loop time requirement

## Notations

- $n$: Length of the array
- $k$: Number of times the loop runs

## Deduction

- $\frac{n}{2^k} = 1$ (Until the search space reduces to a single element)
- $k = \lceil log_2 n \rceil$

## Example

- $n = 5$
- $k = \lceil log_2 5 \rceil = \lceil 2.321928 \rceil = 3$

# Binary Search : Time complexity

| Step | Type | Time |
|---|---|---|
| int *a | assignment | 1 |
| int n | assignment | 1 |
| int key | assignment | 1 |
| lo = 0 | assignment | 1 |
| hi = n - 1 | assignment | 1 |
| mid = (lo + hi)/2 | addition, division, assignment | 1 |
| lo <= hi | comparison | k + 1 |
| a[mid] == key | comparison | k + 1 |
| break | statement | 1 |
| key < a[mid] | comparison | k |
| hi = mid - 1 | assignment | k |
| lo = mid + 1 | assignment | k |
| mid = (lo + hi)/2 | addition, division, assignment | k |
| return 1 | statement | 1 |
| return 0 | statement | 1 |
| TOTAL | ALL | 6*k + 11 |

# Binary Search : Time complexity

| Step | Type | Time |
|---|---|---|
| int *a | assignment | 1 |
| int n | assignment | 1 |
| int key | assignment | 1 |
| lo = 0 | assignment | 1 |
| hi = n - 1 | assignment | 1 |
| mid = (lo + hi)/2 | addition, division, assignment | 1 |
| lo <= hi | comparison | k + 1 |
| a[mid] == key | comparison | k + 1 |
| break | statement | 1 |
| key < a[mid] | comparison | k |
| hi = mid - 1 | assignment | k |
| lo = mid + 1 | assignment | k |
| mid = (lo + hi)/2 | addition, division, assignment | k |
| return 1 | statement | 1 |
| return 0 | statement | 1 |
| TOTAL | ALL | 6*k + 11 |

## Time complexity

$$T(n) = 6*k + 11 = 6*\lceil log_2 n \rceil + 11 = \mathcal{O}(log_2 n)$$

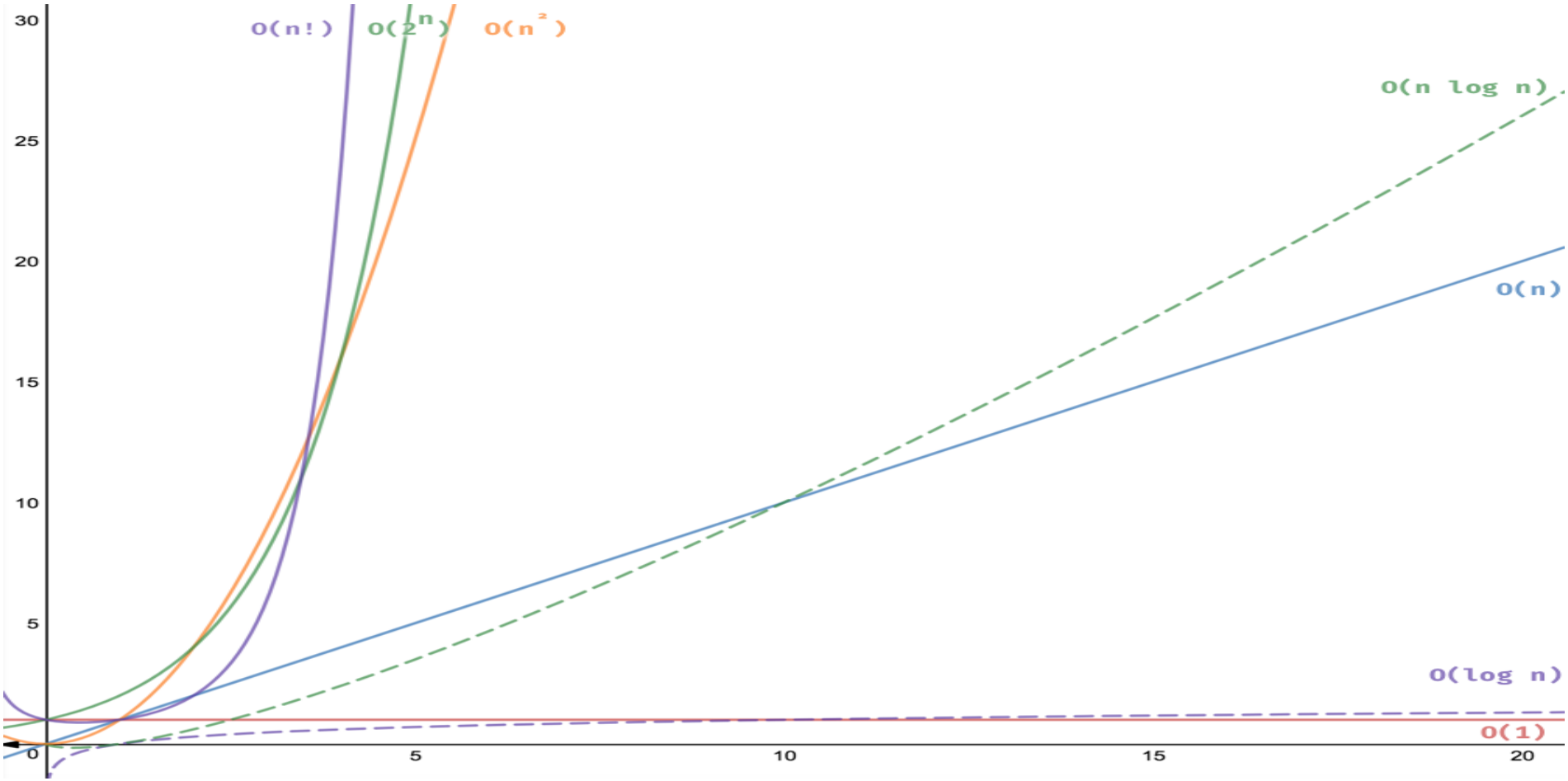
# Important takeaways

- Worst case complexity is usually considered
- The most dominating $n$ term in a running time (execution time) expression $f(n)$ for large $n$ leads to $\mathcal{O}$ bound for $f(n)$
  - $f(n) = 3*n + 5$ ($n$ is the most dominating term) $\Rightarrow f(n) = \mathcal{O}(n)$
  - $f(n) = 5*n^3 + 6*n^2 + 7$ ($n^3$ is the most dominating term) $\Rightarrow f(n) = \mathcal{O}(n^3)$
  - $f(n) = 4*n^2*log_2 n + 9*n + 8*\sqrt{n}$ - 2 ($n^2*log_2 n$ is the most dominating term) $\Rightarrow f(n) = \mathcal{O}(n^2 * log_2 n)$
- Tight (or precise) upper bound is considered
  - $f(n) = 3*n + 5$
  - $f(n) = \mathcal{O}(n)$ (tight/precise upper bound)
  - $f(n) = \mathcal{O}(n^2)$, $f(n) = \mathcal{O}(n^2)$ etc. give loose/imprecise upper bounds
  - It is correct to say that the compexity of *Binary Search* is $\mathcal{O}(n)$ or $\mathcal{O}(n^2)$, but these are loose/imprecise upper bounds (rarely useful)
- Two algorithms may be asymptotically equivalent even they actually have slightly different time requirements
  - $f(n) = 3*n + 5$ and $g(n) = 25*n + 67$ are both $\mathcal{O}(n)$

# Example complexities

| Type | Name | Example |
|---|---|---|
| $\mathcal{O}(1)$ | Constant time | Accessing array element, Push and Pop, Enqueue and Dequeue, best case of search in hashing etc. |
| $\mathcal{O}(n)$ | Linear time | Linear Search, Max/min in an array |
| $\mathcal{O}(n^2)$ | Quadratic time | Bubble sort, Selection sort worst case of quick sort etc. |
| $\mathcal{O}(n^c)$ $(c > 1)$ | Polynomial time | Double or more nesting in loop |
| $\mathcal{O}(log\ n)$ | Logarithmic time | Binary search |
| $\mathcal{O}(nlog\ n)$ | Linearithmic time | Mergesort, best/average case of Quick sort |
| $\mathcal{O}(2^n)$ | Exponential time | Powerset checking, moves in Tower of Hanoi |
| $\mathcal{O}(n!)$ | Factorial time | No of permutations |

# Complexities: graphs[4]




[4] https://tinyurl.com/graphcomplexity


THANK YOU
THANK YOU
THANK
YOU
THANK YOU
HINTS