

ALGORITHMS

RUNTIME ANALYSIS AND ORDER STATISTICS

DR. MUDASSIR SHABBIR

LUMS UNIVERSITY

JANUARY 21, 2026



Recap: Course & Grading

- Grading in this course is **relative**.
- However:
 - Any number of students can get an **A/F**.
- Your goal: **master the material**, not “beat” others.

Strong understanding \Rightarrow strong performance.



Recap: What Is Runtime Analysis?

- An **algorithm** is a sequence of instructions:
 - Written in **plain English**, bullet points, or pseudocode.
 - Independent of programming language or machine.
- We analyze runtime by:
 - Identifying a **primitive operation**.
 - Counting or **bounding** how many times it executes.
- We care about:

Growth of this count as a function of input size n

- Not the value of the input (e.g., **the number itself in primality**), but the **size of the input**.

This is the foundation of asymptotic runtime analysis.



Don't care about absolute exact time.

Upper Bound: $O(\cdot)$ read Big Oh

$$T(n) \leq f(n)$$

Lower Bound: $\Omega(\cdot)$, read Big Omega

$$T(n) \geq g(n)$$


Tight Bound: $\Theta(\cdot)$, read Big Theta

$$T(n) \geq f(n), \text{ and } T(n) \leq f(n).$$

bubbleSort(A)

```
0. n = len(A)
1. for i in range(n):
2.     for j in range(n-1):
3.         if A[j] > A[j + 1]:
4.             temp = A[j]
5.             A[j] = A[j+1]
6.             A[j+1] = temp
7. return A
```

Input Size : **len(A)**



Upper Bound:

$$T(n) \leq f(n)$$

mystry(p):

1. if $p < 2$:
2. return 1
3. else:
4. return $1 + \text{mystry}(\text{sqrt}(p))$

Bound should be valid for very large n only?

Upper Bound:

$$T(n) \leq f(n) \quad \forall n > n_0$$

mystry(p):

1. if $p < 2$:
2. return 1
3. else:
4. return $1 + \text{mystry}(\text{sqrt}(p))$

Choose a minimum input size n_0 - choose n_0 as big as you like.

Constants are irrelevant!

mystery(p):

1. if $p < 2$:
2. return 1
3. else:
4. return $1 + \text{mystery}(\text{sqrt}(p))$

$T(n)$ is upper-bounded by $f(n)$.

then,

$2 \times T(n)$ is upper-bounded by $f(n)$.

$100 \times T(n)$ is upper-bounded by $f(n)$.

$\forall c > 0, c \times T(n)$ is upper-bounded by $f(n)$.

Constants are irrelevant!

mystry(p):

1. if $p < 2$:
2. return 1
3. else:
4. return $1 + \text{mystry}(\text{sqrt}(p))$

$T(n)$ is upper-bounded by $f(n)$.

then,

$2 \times T(n)$ is upper-bounded by $f(n)$.

$100 \times T(n)$ is upper-bounded by $f(n)$.

$\forall c > 0, c \times T(n)$ is upper-bounded by $f(n)$.

for some $c > 0$, $T(n) \leq c \times f(n)$

Asymptotic Big O:

mystry(p):

1. if $p < 2$:
2. return 1
3. else:
4. return $1 + \text{mystry}(\text{sqrt}(p))$

$$T(n) = O(f(n)).$$

then,

for some $c > 0$, $T(n) \leq c \times f(n)$, for $n > n_0$.

To prove that,

$$T(n) = O(f(n)).$$

find two constants, c , and n_0 , so that,

$$T(n) \leq c \times f(n) \text{ , for } n > n_0.$$

To prove that,

$$T(n) = O(f(n)).$$

find two constants, c , and n_0 , so that,

$$T(n) \leq c \times f(n) \text{ , for } n > n_0.$$

Example 1: Prove that $T(n) = 3n^3 + 2n + 7 = O(n^3)$.



To prove that,

$$T(n) = O(f(n)).$$

find two constants, c , and n_0 , so that,

$$T(n) \leq c \times f(n) \text{ , for } n > n_0.$$

Example 1: Prove that $T(n) = 3n^3 + 2n + 7 = O(n^3)$.

Need to choose n_0 , and c , so that, $3n^3 + 2n + 7 \leq c \times n^3$ when $n > n_0$.

To prove that,

$$T(n) = O(f(n)).$$

find two constants, c , and n_0 , so that,

$$T(n) \leq c \times f(n) \text{ , for } n > n_0.$$

Example 1: Prove that $T(n) = 3n^3 + 2n + 7 = O(n^3)$.

Need to choose n_0 , and c , so that, $3n^3 + 2n + 7 \leq c \times n^3$ when $n > n_0$.

Let $n_0 = 1$, and $c = 12$.

$$\begin{aligned} 3n^3 + 2n + 7 &\leq 3n^3 + 2n^3 + 7n^3 \\ &\leq 12n^3 \\ &\leq c \times n^3 \end{aligned}$$



To prove that,

$$T(n) = O(f(n)).$$

find two constants, c , and n_0 , so that,

$$T(n) \leq c \times f(n) \text{ , for } n > n_0.$$

Example 2: $T(n) = a_d n^d + a_{d-1} n^{d-1} + a_{d-2} n^{d-2} \dots + a_1 n + a_0 = O(n^d).$



To prove that,

$$T(n) = O(f(n)).$$

find two constants, c , and n_0 , so that,

$$T(n) \leq c \times f(n) \text{ , for } n > n_0.$$

Example 2: $T(n) = a_d n^d + a_{d-1} n^{d-1} + a_{d-2} n^{d-2} \dots + a_1 n + a_0 = O(n^d)$.

Need to choose n_0 , and c , so that,

$$a_d n^d + a_{d-1} n^{d-1} + a_{d-2} n^{d-2} \dots + a_1 n + a_0 \leq c \times n^d \text{ when } n > n_0.$$



To prove that,

$$T(n) = O(f(n)).$$

find two constants, c , and n_0 , so that,

$$T(n) \leq c \times f(n) \text{ , for } n > n_0.$$

Example 2: $T(n) = a_d n^d + a_{d-1} n^{d-1} + a_{d-2} n^{d-2} \dots + a_1 n + a_0 = O(n^d).$

Need to choose n_0 , and c , so that,

$$a_d n^d + a_{d-1} n^{d-1} + a_{d-2} n^{d-2} \dots + a_1 n + a_0 \leq c \times n^d \text{ when } n > n_0.$$

Let $n_0 = 1$, and $c = \sum_{i=0}^d a_i$

$$\begin{aligned} a_d n^d + a_{d-1} n^{d-1} + a_{d-2} n^{d-2} \dots + a_1 n + a_0 &\leq a_d n^d + a_{d-1} n^d + a_{d-2} n^d + \dots + a_1 n^d + a_0 n^d \\ &\leq \sum_{i=0}^d a_i n^d \\ &\leq n^d \times \sum_{i=0}^d a_i \end{aligned}$$



To prove that,

$$T(n) = O(f(n)).$$

find two constants, c , and n_0 , so that,

$$T(n) \leq c \times f(n) \text{ , for } n > n_0.$$

$$T(n) = a_d n^d + a_{d-1} n^{d-1} + a_{d-2} n^{d-2} \dots + a_1 n + a_0 = O(n^d).$$

$$6n^4 + 3n^3 + 3n^2 + 2n + 1 = O(n^4)$$

$$(3n^2 + 4)(2n + 1)$$



To prove that,

$$T(n) = O(f(n)).$$

find two constants, c , and n_0 , so that,

$$T(n) \leq c \times f(n) \text{ , for } n > n_0.$$

$$T(n) = a_d n^d + a_{d-1} n^{d-1} + a_{d-2} n^{d-2} \dots + a_1 n + a_0 = O(n^d).$$

$$6n^4 + 3n^3 + 3n^2 + 2n + 1 = O(n^4)$$

$$(3n^2 + 4)(2n + 1) = O(n^3)$$

$$6n^3 + 3n^2 + 8n + 4$$



To prove that,

$$T(n) = O(f(n)).$$

find two constants, c , and n_0 , so that,

$$T(n) \leq c \times f(n) \text{ , for } n > n_0.$$

$$T(n) = a_d n^d + a_{d-1} n^{d-1} + a_{d-2} n^{d-2} \dots + a_1 n + a_0 = O(n^d).$$

$$6n^4 + 3n^3 + 3n^2 + 2n + 1 = O(n^4)$$

$$(3n^2 + 4)(2n + 1) = O(n^3)$$

$$6n^3 + 3n^2 + 8n + 4 = O(n^4)$$

$$2^n + 2^n \log n + 4n! + 2^{\log^2 n}$$

$$9n^2 + \frac{1}{4}n^2 \log n - 3n + 5\sqrt{n} + 2 = O(n^2 \log n)$$



To prove that,

$$T(n) = O(f(n)).$$

find two constants, c , and n_0 , so that,

$$T(n) \leq c \times f(n) \text{ , for } n > n_0.$$

$$T(n) = a_d n^d + a_{d-1} n^{d-1} + a_{d-2} n^{d-2} \dots + a_1 n + a_0 = O(n^d).$$

$$6n^4 + 3n^3 + 3n^2 + 2n + 1 = O(n^4)$$

$$(3n^2 + 4)(2n + 1) = O(n^3)$$

$$6n^3 + 3n^2 + 8n + 4 = O(n^4)$$

$$9n^2 + \frac{1}{4}n^2 \log n - 3n + 5\sqrt{n} + 2 = O(n^2 \log n)$$

$$2^n + 2^n \log n + 4n! + 2^{\log^2 n} = O(n!)$$

$$126. + 2\frac{1}{n} + \frac{1}{n^2}$$



To prove that,

$$T(n) = O(f(n)).$$

find two constants, c , and n_0 , so that,

$$T(n) \leq c \times f(n) \text{ , for } n > n_0.$$

$$T(n) = a_d n^d + a_{d-1} n^{d-1} + a_{d-2} n^{d-2} \dots + a_1 n + a_0 = O(n^d).$$

$$6n^4 + 3n^3 + 3n^2 + 2n + 1 = O(n^4)$$

$$(3n^2 + 4)(2n + 1) = O(n^3)$$

$$6n^3 + 3n^2 + 8n + 4 = O(n^4)$$

$$9n^2 + \frac{1}{4}n^2 \log n - 3n + 5\sqrt{n} + 2 = O(n^2 \log n)$$

$$2^n + 2^n \log n + 4n! + 2^{\log^2 n} = O(n!)$$

$$126. + 2\frac{1}{n} + \frac{1}{n^2} = O(1)$$

$$2\frac{1}{n} + \frac{1}{n^2}$$



To prove that,

$$T(n) = O(f(n)).$$

find two constants, c , and n_0 , so that,

$$T(n) \leq c \times f(n) \text{ , for } n > n_0.$$

$$T(n) = a_d n^d + a_{d-1} n^{d-1} + a_{d-2} n^{d-2} \dots + a_1 n + a_0 = O(n^d).$$

$$\begin{aligned} 26 &< \log^* n < \log \log n < \log n < \log^2 n \\ &< \log^k n < n^\epsilon < \sqrt{n} < n < n^2 < n^k \\ &< 1.01^n < 2^n < n! < n^n \end{aligned}$$

$$6n^4 + 3n^3 + 3n^2 + 2n + 1 = O(n^4)$$

$$(3n^2 + 4)(2n + 1) = O(n^3)$$

$$6n^3 + 3n^2 + 8n + 4 = O(n^4)$$

$$9n^2 + \frac{1}{4}n^2 \log n - 3n + 5\sqrt{n} + 2 = O(n^2 \log n)$$

$$2^n + 2^n \log n + 4n! + 2^{\log^2 n} = O(n!)$$

$$126. + 2\frac{1}{n} + \frac{1}{n^2} = O(1)$$

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



To prove that,

$$T(n) = O(f(n)).$$

find two constants, c , and n_0 , so that,

$$T(n) \leq c \times f(n) \text{ , for } n > n_0.$$

To prove that,

$$T(n) = O(f(n)).$$

find two constants, c , and n_0 , so that,

$$T(n) \leq c \times f(n) \text{ , for } n > n_0.$$

To prove that,

$$T(n) = \Omega(f(n)).$$

find two constants, c , and n_0 , so that,

$$T(n) \leq c \times f(n) \text{ , for } n > n_0.$$

To prove that,

$$T(n) = O(f(n)).$$

find two constants, c , and n_0 , so that,

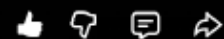
$$T(n) \leq c \times f(n) \text{ , for } n > n_0.$$

To prove that,

$$T(n) = \Omega(f(n)).$$

find two constants, c , and n_0 , so that,

$$T(n) \geq c \times f(n) \text{ , for } n > n_0.$$



$f(n) = \Theta(g(n))$ informally means:

f grows no faster than g

f grows at least as fast as g

f equals g

f is smaller than g

$f(n) = \Theta(g(n))$ informally means:

f grows no faster than g

0%

f grows at least as fast as g

0%

f equals g

0%

f is smaller than g

0%

$f(n) = \Theta(g(n))$ informally means:

f grows no faster than g

0%

f grows at least as fast as g

0%

f equals g

0%

f is smaller than g

0%

Asymptotic Notation: Why?

- We want to describe how an algorithm's runtime **grows** as the input size n becomes **large**.
- **Exact counts** of operations are often messy or machine-dependent.
- **Asymptotic notation** **abstracts away constants** and low-order terms.
- Example:

$$5n + 20 \sim O(n)$$

because for large n , the $5n$ term dominates and constants don't matter.



Big-O Notation (Upper Bound)

Definition: A function $f(n)$ is $O(g(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that:

$$0 \leq f(n) \leq c \cdot g(n) \quad \text{for all } n \geq n_0$$

- Intuition: $f(n)$ grows at most like $g(n)$ for large n .
- Example: $f(n) = 5n + 20$ Then $f(n) = O(n)$ with $c = 6$, $n_0 = 20$.
- Used to express worst-case runtime of algorithms.



Other Notations: Ω and Θ

- **Big-Omega Ω (Lower Bound):** $f(n) = \Omega(g(n))$ if there exist constants $c > 0$, n_0 such that

$$f(n) \geq c \cdot g(n) \quad \text{for all } n \geq n_0$$

Intuition: $f(n)$ grows at least as fast as $g(n)$.

- **Big-Theta Θ (Tight Bound):** $f(n) = \Theta(g(n))$ if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$ Intuition: $f(n)$ grows exactly like $g(n)$ asymptotically.
- Example: $f(n) = 5n + 20$ Then $f(n) = \Theta(n)$



$O(\cdot)$ Big oh notation means

A Best Case Scenario

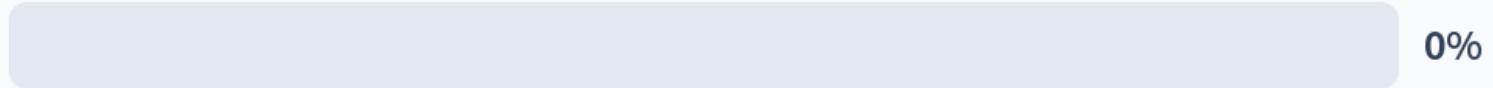
B Worst Case Scenario

C Average Case Scenario

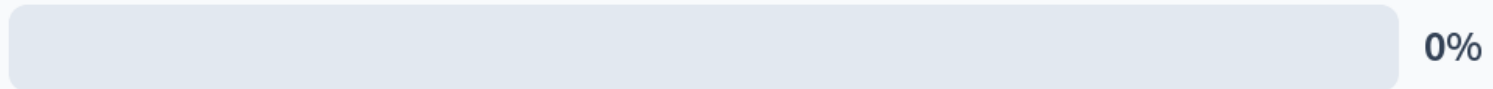
D None of the above

$O(\cdot)$ Big oh notation means

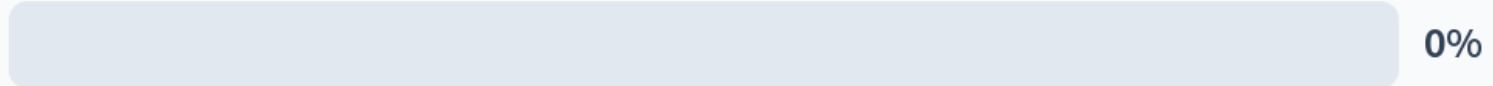
A Best Case Scenario



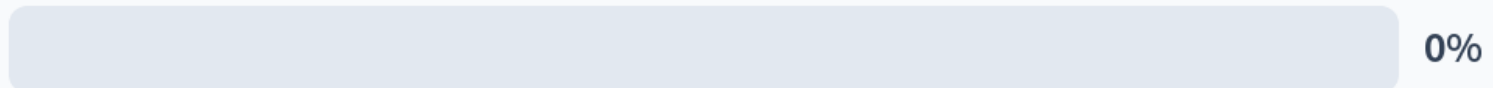
B Worst Case Scenario



C Average Case Scenario

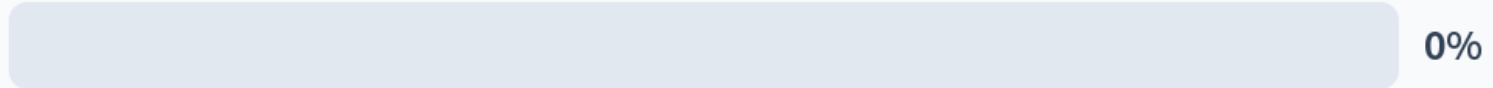


D None of the above

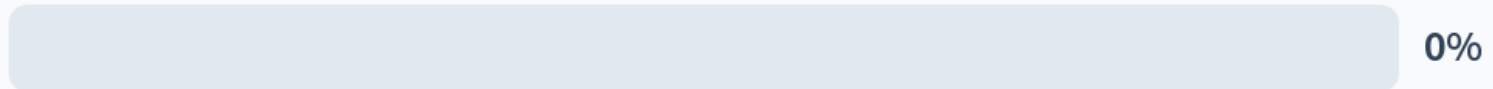


$O(\cdot)$ Big oh notation means

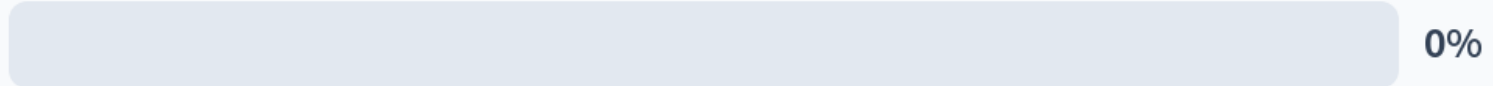
A Best Case Scenario



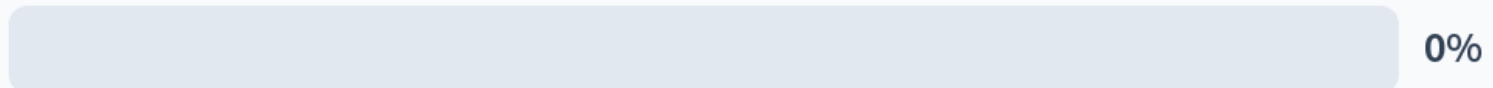
B Worst Case Scenario



C Average Case Scenario



D None of the above



Best Case, Worst Case, Average Case

- When analyzing an algorithm, **input** can affect runtime.
- **Best Case:** Minimum number of steps the algorithm takes for any input of size n .
- **Worst Case:** Maximum number of steps the algorithm takes for any input of size n .
- **Average Case:**



Best Case, Worst Case, Average Case

- When analyzing an algorithm, **input** can affect runtime.
- **Best Case:** Minimum number of steps the algorithm takes for any input of size n .
- **Worst Case:** Maximum number of steps the algorithm takes for any input of size n .
- **Average Case:** Expected number of steps, assuming a probability distribution over all inputs.



Best Case, Worst Case, Average Case

- When analyzing an algorithm, **input** can affect runtime.
- **Best Case:** Minimum number of steps the algorithm takes for any input of size n .
- **Worst Case:** Maximum number of steps the algorithm takes for any input of size n .
- **Average Case:** Expected number of steps, assuming a probability distribution over all inputs.

Helps us understand performance under different scenarios.



Example: Finding Maximum

- Input: List of n numbers
- Algorithm: Scan all elements, updating largest seen so far.
- Best Case:
 - No matter what, must check every element.
 - Comparisons: $n - 1$
- Worst Case:
 - Also $n - 1$ comparisons (same as best case)
- Average Case:
 - Also $n - 1$ comparisons
- **Observation:** For some algorithms, best, worst, and average cases differ; for others (like this one), they are the same.



Example: Random Guess Algorithm

Algorithm: To find the largest number in a list of n numbers:

- ➊ Pick a number at random from the list.
- ➋ Check if it is the largest in the list:
 - Compare it with all other $n - 1$ elements.
- ➌ If it is the largest, return it; otherwise, pick another random number and repeat.

Let's analyze its best, worst, and average case.



Analysis of Random Guess Algorithm

- **Best Case:** First guess is the largest element:

$n - 1$ comparisons

- **Worst Case:** You keep guessing the wrong numbers and finally pick the largest last:

Potentially infinite if random picks repeat!

- **Average Case:** On average, you find the largest after trying about half the elements:

Expected guesses $\approx n$, each guess costs $n - 1$ comparisons

\Rightarrow Average comparisons $\approx (n - 1) \cdot n = O(n^2)$

Observation: This random algorithm is much worse on average than the simple linear scan.



Example

Problem: Given a list of n (distinct) numbers, find the 2nd largest one.

!



Example

Problem: Given a list of n (distinct) numbers, find the 2nd largest one.

Problem: Given a list of n (distinct) numbers, find the k^{th} largest one.

!



Example

Problem: Given a list of n (distinct) numbers, find the 2nd largest one.

Problem: Given a list of n (distinct) numbers, find the k^{th} largest one.

Think of k as, $\log n$, and $\frac{n}{2}$!

