DSA Theory Question

Muhammad Qasim

Muhammad Qasim
[COMPANY NAME] [Company address]

Question #1 - Languages

- Python
- Java
- C
- C++
- C#
- Go (Golang)
- Rust
- Swift
- Kotlin
- Ruby
- JavaScript
- TypeScript
- Dart
- Objective-C
- PHP
- R
- Julia
- Perl
- Scala
- Groovy
- F#
- Haskell

- OCaml
- Elixir
- Erlang
- Nim
- Crystal

Question #2 – 1 indexed languages

- Python
- Java
- C
- C++
- C#
- Go (Golang)
- Rust
- Swift
- Kotlin
- Ruby
- JavaScript
- TypeScript
- Dart
- Objective-C
- PHP
- R

- Julia
- Perl
- Scala
- Groovy
- F#
- Haskell
- OCaml
- Elixir
- Erlang
- Nim
- Crystal

Zero - Index Languages

- **C**
- **C++**
- **Java**
- **C#**
- **JavaScript / TypeScript**
- **Go**
- **Rust**
- **Swift**
- **Kotlin**

- **PHP** (for numerically indexed arrays by default)
- **Perl**
- **Ruby**
- **Python**
- **R (technically vectors/lists, but see below note)**
- **Scala**
- **Groovy**
- **D**
- **Julia**
- **Dart**
- **Objective-C**
- **Lua (since 5.0, tables can simulate, but conventionally starts at 1)**
- **Many Assembly languages (x86, ARM, etc.)**

Array information across different languages

🔹 **Language-wise Array Systems**

C

- **Indexing**: Starts at 0.
- **Declaration**: int arr[5];
- **Size**: Fixed at compile time (unless dynamically allocated with malloc).
- **Bounds Checking**: ❌ No (accessing out-of-bounds is undefined behavior).

- **Mutability**: ✅ Mutable.
- **Unique**: Arrays are just pointers to contiguous memory; array name decays to pointer.

C++

- **Indexing**: 0-based.
- **Types**: C-style arrays (int arr[10];) or STL containers (std::array, std::vector).
- **Size**:
    - Fixed (C-style).
    - Dynamic (std::vector grows/shrinks).
- **Bounds Checking**:
    - ❌ No for C-style.
    - ✅ Optional (.at() in std::vector).
- **Unique**: Supports templates → arrays of any type, even user-defined.

Java

- **Indexing**: 0-based.
- **Declaration**: int[] arr = new int[5];
- **Size**: Fixed once created.
- **Bounds Checking**: ✅ Always checked → ArrayIndexOutOfBoundsException.
- **Mutability**: ✅ Elements mutable, but array length fixed.
- **Unique**: Arrays are objects (arr.length). Multi-dimensional arrays supported.

Python

- **Indexing**: 0-based.
- **Array Type**: List (list) is dynamic array; array module exists for typed arrays.

- **Declaration**: arr = [1, 2, 3]
- **Size**: Dynamic, grows automatically.
- **Bounds Checking**: ✅ Raises IndexError.
- **Mutability**: ✅ Lists mutable; tuples immutable.
- **Unique**: Negative indexing (arr[-1] = last element), slicing (arr[1:4]).

JavaScript

- **Indexing**: 0-based.
- **Declaration**: let arr = [1,2,3];
- **Size**: Dynamic, resizes automatically.
- **Bounds Checking**: ❌ Out-of-range index returns undefined.
- **Mutability**: ✅ Elements mutable.
- **Unique**: Technically arrays are special objects; can have "holes" (arr[10] exists but is undefined).

PHP

- **Indexing**: 0-based for numeric arrays, but associative arrays use keys (like maps).
- **Declaration**: $arr = [1, 2, 3];
- **Size**: Dynamic.
- **Bounds Checking**: ❌ Returns null if index not set.
- **Mutability**: ✅ Mutable.
- **Unique**: Unified "array" structure (mix of list + dictionary).

Ruby

- **Indexing**: 0-based.
- **Declaration**: arr = [1, 2, 3]

- **Size**: Dynamic.
- **Bounds Checking**: ❌ Out-of-range returns nil.
- **Mutability**: ✅ Mutable.
- **Unique**: Negative indexing supported, slicing flexible.

R

- **Indexing**: 1-based.
- **Declaration:** arr <- c(1,2,3)
- **Size**: Dynamic.
- **Bounds Checking**: ✅ Out-of-range returns NA.
- **Mutability**: ✅ Mutable.
- **Unique**: Vectorized operations → apply operations to whole arrays at once.

MATLAB

- **Indexing**: 1-based.
- **Declaration:** A = [1 2 3; 4 5 6]
- **Size**: Dynamic.
- **Bounds Checking**: ✅ Errors if out-of-bounds.
- **Mutability**: ✅ Mutable.
- **Unique**: Arrays are the **fundamental type** → everything is a matrix/array.

Fortran

- **Indexing**: 1-based by default (but can redefine).
- **Declaration:** REAL :: A(5)
- **Size**: Fixed or allocatable (dynamic).

- **Bounds Checking**: Optional (depends on compiler flags).
- **Unique**: Arrays are first-class, powerful slicing and whole-array operations.

Lua

- **Indexing**: 1-based by convention (though technically any index is allowed).
- **Declaration**: arr = {10,20,30}
- **Size**: Dynamic.
- **Bounds Checking**: ❌ Out-of-range returns nil.
- **Mutability**: ✅ Mutable.
- **Unique**: Arrays are just tables → flexible, associative + numeric.

Julia

- **Indexing**: 1-based by default (but can support 0 or custom ranges).
- **Declaration**: arr = [1,2,3]
- **Size**: Dynamic.
- **Bounds Checking**: ✅ Always checked (can be disabled for performance).
- **Mutability**: ✅ Mutable.
- **Unique**: Supports arbitrary indexing ranges (OffsetArrays).

Haskell

- **Indexing**: 0-based.
- **Declaration**: arr = [1,2,3]
- **Size**: Fixed (immutable lists), but arrays (Data.Array) exist.
- **Bounds Checking**: ✅ Raises exception.
- **Mutability**: ❌ Immutable by default (mutable arrays in special monads).

- **Unique**: Laziness and immutability dominate usage.

SQL (array-like types)

- **Indexing**: 1-based in PostgreSQL arrays.
- **Declaration**: ARRAY[1,2,3]
- **Size**: Fixed at creation.
- **Bounds Checking**: ✅ Error if out-of-range.
- **Mutability**: ❌ Immutable once defined.
- **Unique**: Designed for databases, not general computation.

📊 Summary of Index Bases

- **0-based**: C, C++, Java, C#, Python, JavaScript, Swift, Kotlin, Rust, Go, Ruby.
- **1-based**: R, MATLAB, Fortran (default), Julia (default), Lua (default), SQL arrays.
- **Flexible**: Pascal, Ada, Julia (with packages), Fortran (modern versions).

Question #3 - datatypes size

General Datatypes and Their Sizes

Datatype	Typical Size
Boolean	1 bit (stored as 1 byte in most systems)
Character (char)	1 byte
Wide Character	2 bytes
Short Integer	2 bytes
Integer	2–4 bytes
Long Integer	4 bytes
Long Long Integer	8 bytes
Float (Single Precision)	4 bytes
Double (Double Precision)	8 bytes
Long Double	10–16 bytes

⚖️ Factors Affecting Data Type Size

1. **Language specification**
   - Some languages fix sizes (e.g., Java, C#).
   - Others leave it flexible (C, C++).
2. **Compiler implementation**
   - Compilers may choose different representations for efficiency.
3. **CPU architecture**
   - Word size (16/32/64-bit) affects integer/pointer sizes.
4. **Operating System & ABI**
   - Defines calling conventions, memory alignment, padding.
5. **Encoding Standard**
   - Characters/strings depend on whether ASCII, UTF-8, UTF-16, UTF-32.
6. **Memory alignment / padding**
   - To improve CPU access speed, types may occupy extra unused bytes.