Why Use NumPy?

Python lists are flexible but **slow** for numerical computing because they:

- Store elements as **pointers** instead of a continuous block of memory.
- Lack **vectorized operations**, relying on loops instead.
- Have significant **overhead** due to dynamic typing.

NumPy's Superpowers:

- **Faster** than Python lists (C-optimized backend)
- **Uses less memory** (efficient storage)
- **Supports vectorized operations** (no explicit loops needed)
- **Has built-in mathematical functions**

NumPy vs. Python Lists – Performance Test

Let's compare Python lists with NumPy arrays using a simple example.

Example 1: Adding Two Lists vs. NumPy Arrays

```
import numpy as np
import time

# Python list
size = 1_000_000
list1 = list(range(size))
list2 = list(range(size))

start = time.time()
result = [x + y for x, y in zip(list1, list2)]
end = time.time()
```

```
print("Python list addition time:", end - start)

# NumPy array
arr1 = np.array(list1)
arr2 = np.array(list2)

start = time.time()
result = arr1 + arr2  # Vectorized operation
end = time.time()
print("NumPy array addition time:", end - start)
```

Key Takeaway: NumPy is significantly **faster** because it performs operations in **C**, avoiding Python loops.

Creating NumPy Arrays

```
import numpy as np

# Creating a 1D NumPy array
arr1 = np.array([1, 2, 3, 4, 5])
print(arr1)

# Creating a 2D NumPy array
arr2 = np.array([[1, 2, 3], [4, 5, 6]])
print(arr2)

# Checking type and shape
print("Type:", type(arr1))
print("Shape:", arr2.shape)
```

◇ **NumPy stores data in a contiguous memory block**, making access faster than lists. ◇ `shape` shows the **dimensions** of an array.

Memory Efficiency – NumPy vs. Lists

Let's check memory consumption.

```
import sys

list_data = list(range(1000))
numpy_data = np.array(list_data)

print("Python list size:", sys.getsizeof(list_data) * len(list_data), "bytes")
print("NumPy array size:", numpy_data.nbytes, "bytes")
```

NumPy arrays use significantly less memory compared to Python lists.

Vectorization – No More Loops!

NumPy avoids loops by applying operations to entire arrays at once using SIMD (Single Instruction, Multiple Data) and other low-level optimizations. SIMD is a CPU-level optimization provided by modern processors.

Example 2: Squaring Elements

```
# Python list (loop-based)
list_squares = [x ** 2 for x in list1]

# NumPy (vectorized)
numpy_squares = arr1 ** 2
```

- NumPy is **cleaner** and **faster**!

Summary

- NumPy is **faster** than Python lists because it is optimized in **C**.

- It consumes **less memory** due to efficient storage.
- It provides **vectorized operations**, removing the need for slow loops.
- Essential for **data science** and **machine learning** workflows.

Exercises for Practice

- Create a NumPy array with values from **10 to 100** and print its shape.
- Compare the time taken to multiply **two Python lists** vs. **two NumPy arrays**.
- Find the **memory size** of a NumPy array with **1 million elements**.

Creating NumPy Arrays

Why NumPy Arrays?

NumPy arrays are the **core** of numerical computing in Python. They are:

- **Faster** than Python lists (C-optimized)
- **Memory-efficient** (store data in a contiguous block)
- **Support vectorized operations that support SIMD** (no slow Python loops)
- **Used in ML, Data Science, and AI**

---

1. Creating NumPy Arrays

From Python Lists:

```
import numpy as np


arr1 = np.array([1, 2, 3, 4, 5])  # 1D array
arr2 = np.array([[1, 2, 3], [4, 5, 6]])  # 2D array
```

```
print(arr1)  # [1 2 3 4 5]
print(arr2)
# [[1 2 3]
#  [4 5 6]]
```

- **Unlike lists, all elements must have the same data type.**

Creating Arrays from Scratch:

```
np.zeros((3, 3))    # 3x3 array of zeros
np.ones((2, 4))     # 2x4 array of ones
np.full((2, 2), 7)  # 2x2 array filled with 7
np.eye(4)           # 4x4 identity matrix
np.arange(1, 10, 2) # [1, 3, 5, 7, 9] (like range)
np.linspace(0, 1, 5) # [0. 0.25 0.5 0.75 1.] (evenly spaced)
```

Key Takeaway: NumPy offers powerful shortcuts to create arrays **without loops!**

---

2. Checking Array Properties

```
arr = np.array([[10, 20, 30], [40, 50, 60]])

print("Shape:", arr.shape)   # (2, 3) → 2 rows, 3 columns
print("Size:", arr.size)     # 6 → total elements
print("Dimensions:", arr.ndim) # 2 → 2D array
print("Data type:", arr.dtype) # int64 (or int32 on Windows)
```

- **NumPy arrays are strongly typed**, meaning all elements share the same data type.

---

3. Changing Data Types

```
arr = np.array([1, 2, 3], dtype=np.float32)  # Explicit type
print(arr.dtype)  # float32

arr_int = arr.astype(np.int32)  # Convert float to int
print(arr_int)  # [1 2 3]
```

- **Efficient memory usage** by choosing the right data type.

---

4. Reshaping and Flattening Arrays

```
arr = np.array([[1, 2, 3], [4, 5, 6]])
print(arr.shape)  # (2, 3)

reshaped = arr.reshape((3, 2))  # Change shape
print(reshaped)
# [[1 2]
#  [3 4]
#  [5 6]]

flattened = arr.flatten()  # Convert 2D → 1D
print(flattened)  # [1 2 3 4 5 6]
```

Indexing and slicing

Lets now learn about indexing and slicing in Numpy

Indexing (Same as Python Lists)

```
arr = np.array([10, 20, 30, 40])
print(arr[0])  # 10
print(arr[-1]) # 40
```

Slicing (Extracting Parts of an Array)

```
arr = np.array([10, 20, 30, 40, 50])

print(arr[1:4])  # [20 30 40] (slice from index 1 to 3)
print(arr[:3])   # [10 20 30] (first 3 elements)
print(arr[::2])  # [10 30 50] (every 2nd element)
```

Slicing returns a view, not a copy! Changes affect the original array.**

This might seem counterintuitive since Python lists create copies when sliced. But in NumPy, slicing returns a view of the original array. Both the sliced array and the original array share the same data in memory, so changes in the slice affect the original array.

Why does this happen?

- Memory Efficiency: Avoids unnecessary copies, making operations faster and saving memory.
- Performance: Enables faster access and manipulation of large datasets without duplicating data.

```
sliced = arr[1:4]
sliced[0] = 999
print(arr)  # [10 999 30 40 50]
```

- Use `.copy()` if you need an independent copy.

---

6. Fancy Indexing & Boolean Masking

Fancy Indexing (Select Multiple Elements)

```
arr = np.array([10, 20, 30, 40, 50])
idx = [0, 2, 4]  # Indices to select
print(arr[idx])  # [10 30 50]
```

Boolean Masking (Filter Data)

```
arr = np.array([10, 20, 30, 40, 50])
mask = arr > 25  # Condition: values greater than 25
print(arr[mask])  # [30 40 50]
```

This is a powerful way to filter large datasets efficiently!

Summary

- **NumPy arrays** are faster, memory-efficient alternatives to lists.
- You can create arrays using `np.array()`, `np.zeros()`, `np.ones()`, etc.
- **Indexing & slicing** allow efficient data manipulation.
- **Reshaping & flattening** change array structures without copying data.
- **Fancy indexing & boolean masking** help filter and access specific data.

Exercises for Practice

- Create a **3×3** array filled with random numbers and print its shape.
- Convert an array of floats `[1.1, 2.2, 3.3]` into integers.
- Use **fancy indexing** to extract **even numbers** from `[1, 2, 3, 4, 5, 6]`.
- Reshape a **1D array of size 9** into **a 3×3 matrix**.

- Use **boolean masking** to filter numbers **greater than 50** in an array.

Multidimensional Indexing and Axis

NumPy allows you to efficiently work with **multidimensional arrays**, where indexing and axis manipulation play a crucial role. Understanding how indexing works across multiple dimensions is essential for data science and machine learning tasks.

---

1. Understanding Axes in NumPy

Each dimension in a NumPy array is called an **axis**. Axes are numbered starting from **0**.

For example:

- **1D array** → 1 axis (axis 0)
- **2D array** → 2 axes (axis 0 = rows, axis 1 = columns)
- **3D array** → 3 axes (axis 0 = depth, axis 1 = rows, axis 2 = columns)

Example: Axes in a 2D Array

```
import numpy as np

arr = np.array([[1, 2, 3],
                [4, 5, 6],
                [7, 8, 9]])

print(arr)
```

Output:

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

- **Axis 0 (rows)** → Operations move **down** the columns.
- **Axis 1 (columns)** → Operations move **across** the rows.

Summing along axes:

```
print(np.sum(arr, axis=0))  # Sum along rows (down each column)
print(np.sum(arr, axis=1))  # Sum along columns (across each row)
```

Output:

```
[12 15 18]  # Column-wise sum
[ 6 15 24]  # Row-wise sum
```

---

2. Indexing in Multidimensional Arrays

You can access elements using **row and column indices**.

```
# Accessing an element
print(arr[1, 2])  # Row index 1, Column index 2 → Output: 6
```

You can also use **slicing** to extract parts of an array:

```
print(arr[0:2, 1:3])  # Extracts first 2 rows and last 2 columns
```

Output:

```
[[2 3]
 [5 6]]
```

3. Indexing in 3D Arrays

For **3D arrays**, the first index refers to the "depth" (sheets of data).

```
arr3D = np.array([[[1, 2, 3], [4, 5, 6]],
                  [[7, 8, 9], [10, 11, 12]]])

# Output of arr3D.shape is → (depth, rows, columns)
print(arr3D.shape)  # Output: (2, 2, 3)
```

Accessing elements in 3D:

```
# First sheet, second row, third column
print(arr3D[0, 1, 2])  # Output: 6

print(arr3D[:, 0, :])   # Get the first row from both sheets
```

4. Practical Example: Selecting Data Along Axes

```
# Get all rows of the first column
first_col = arr[:, 0]
print(first_col)  # Output: [1 4 7]
```

```
# Get the first row from each "sheet" in a 3D array
first_rows = arr3D[:, 0, :]
print(first_rows)
```

Output:

```
[[ 1  2  3]
 [ 7  8  9]]
```

5. Changing Data Along an Axis

```
# Replace all elements in column 1 with 0
arr[:, 1] = 0
print(arr)
```

Output:

```
[[1 0 3]
 [4 0 6]
 [7 0 9]]
```

6. Summary

- Axis 0 = rows (vertical movement), Axis 1 = columns (horizontal movement)
- Indexing works as `arr[row, column]` for 2D arrays and `arr[depth, row, column]` for 3D arrays
- Slicing allows extracting subarrays
- Operations along axes help efficiently manipulate data without loops

Data Types in NumPy

Let's learn about NumPy's **data types** and explore how they affect memory usage and performance in your arrays.

1. Introduction to NumPy Data Types

NumPy arrays are **homogeneous**, meaning that they can only store elements of the same type. This is different from Python lists, which can hold mixed data types. NumPy supports various **data types** (also called **dtypes**), and understanding them is crucial for optimizing memory usage and performance.

Common Data Types in NumPy:

- `int32`, `int64`: Integer types with different bit sizes.
- `float32`, `float64`: Floating-point types with different precision.
- `bool`: Boolean data type.
- `complex64`, `complex128`: Complex number types.
- `object`: For storing objects (e.g., Python objects, strings).

You can check the dtype of a NumPy array using the `.dtype` attribute.

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])
print(arr.dtype)  # Output: int64 (or int32 depending on the system)
```

---

2. Changing Data Types

You can **cast** (convert) the data type of an array using the `.astype()` method. This is useful when you need to change the type for a specific operation or when you want to reduce memory usage.

Example: Changing Data Types

```
arr = np.array([1.5, 2.7, 3.9])
print(arr.dtype)  # Output: float64

arr_int = arr.astype(np.int32)  # Converting float to int
print(arr_int)    # Output: [1 2 3]
print(arr_int.dtype)  # Output: int32
```

Example: Downcasting to Save Memory

```
arr_large = np.array([1000000, 2000000, 3000000], dtype=np.int64)
arr_small = arr_large.astype(np.int32)  # Downcasting to a smaller dtype
```

```
print(arr_small)  # Output: [1000000 2000000 3000000]
print(arr_small.dtype)  # Output: int32
```

---

3. Why Data Types Matter in NumPy

The choice of data type affects: - **Memory Usage**: Smaller data types use less memory. - **Performance**: Operations on smaller data types are faster due to less data being processed. - **Precision**: Choosing the appropriate data type ensures that you don't lose precision (e.g., using `float32` instead of `float64` if you don't need that extra precision).

Example: Memory Usage

```
arr_int64 = np.array([1, 2, 3], dtype=np.int64)
arr_int32 = np.array([1, 2, 3], dtype=np.int32)

print(arr_int64.nbytes)  # Output: 24 bytes (3 elements * 8 bytes each)
print(arr_int32.nbytes)  # Output: 12 bytes (3 elements * 4 bytes each)
```

---

4. String Data Type in NumPy

Although NumPy arrays typically store numerical data, you can also store strings by using the `dtype='str'` or `dtype='U'` (Unicode string) format. However, working with strings in NumPy is **less efficient** than using lists or Python's built-in string types.

Example: String Array

```
arr = np.array(['apple', 'banana', 'cherry'], dtype='U10')  # Unicode string array
print(arr)
```

---

5. Complex Numbers

NumPy also supports **complex numbers**, which consist of a real and imaginary part. You can store complex numbers using `complex64` or `complex128` data types.

Example: Complex Numbers

```
arr = np.array([1 + 2j, 3 + 4j, 5 + 6j], dtype='complex128')
print(arr)
```

---

6. Object Data Type

If you need to store mixed or complex data types (e.g., Python objects), you can use `dtype='object'`. However, this type sacrifices performance, so it should only be used when absolutely necessary.

Example: Object Data Type

```
arr = np.array([{'a': 1}, [1, 2, 3], 'hello'], dtype=object)
print(arr)
```

---

7. Choosing the Right Data Type

Choosing the correct data type is essential for: - **Optimizing memory**: Using the smallest data type that fits your data. - **Improving performance**: Smaller types generally lead to faster operations. - **Ensuring precision**: Avoid truncating or losing important decimal places or values.

Summary:

- **NumPy arrays are homogeneous**, meaning all elements must be of the same type.
- Use `.astype()` to **change data types** and optimize memory and performance.

- The choice of data type affects **memory usage**, **performance**, and **precision**.
- Be mindful of **complex numbers** and **object data types**, which can increase memory usage and reduce performance.

Broadcasting in NumPy

Now, we'll explore how to make your code faster with **vectorization** and **broadcasting** in NumPy. These techniques are key to boosting performance in numerical operations by avoiding slow loops and memory inefficiency.

1. Why Loops Are Slow

In Python, loops are typically slow because: - **Python's interpreter**: Every iteration of the loop requires Python to interpret the loop logic, which is inherently slower than lower-level, compiled code. - **High overhead**: Each loop iteration in Python involves additional overhead for function calls, memory access, and index management.

While Python loops are convenient, they don't take advantage of the **optimized memory and computation** that libraries like **NumPy** provide.

Example: Looping Over Arrays in Python

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])
result = []

# Using a loop to square each element (slow)
for num in arr:
    result.append(num ** 2)

print(result)  # Output: [1, 4, 9, 16, 25]
```

This works, but it's not efficient. Each loop iteration is slow, especially with large datasets.

---

2. Vectorization: Fixing the Loop Problem

Vectorization allows you to perform operations on entire arrays **at once**, instead of iterating over elements one by one. This is made possible by **NumPy's optimized C-based backend** that executes operations in compiled code, which is much faster than Python loops.

Vectorized operations are also **more readable** and compact, making your code easier to maintain.

Example: Vectorized Operation

```
arr = np.array([1, 2, 3, 4, 5])
result = arr ** 2  # Vectorized operation
print(result)  # Output: [1 4 9 16 25]
```

Here, the operation is applied to all elements of the array simultaneously, and it's much faster than looping over the array.

Why is it Faster?

- **Low-level implementation**: NumPy's vectorized operations are implemented in **C** (compiled language), which is much faster than Python loops.
- **Batch processing**: NumPy processes multiple elements in parallel using **SIMD** (Single Instruction, Multiple Data), allowing multiple operations to be done simultaneously.

---

3. Broadcasting: Scaling Arrays Without Extra Memory

Broadcasting is a powerful feature of NumPy that allows you to perform operations on arrays of different shapes without creating copies. It "stretches" smaller arrays across larger arrays in a memory-efficient way, avoiding the overhead of creating multiple copies of data.

Example: Broadcasting with Scalar

Broadcasting is often used when you want to perform an operation on an array and a scalar value (e.g., add a number to all elements of an array).

```
arr = np.array([1, 2, 3, 4, 5])
result = arr + 10  # Broadcasting: 10 is added to all elements
print(result)  # Output: [11 12 13 14 15]
```

Here, the scalar `10` is "broadcast" across the entire array, and no extra memory is used.

---

4. Broadcasting with Arrays of Different Shapes

Broadcasting becomes more powerful when you apply operations on arrays of **different shapes**. NumPy automatically adjusts the shapes of arrays to make them compatible for element-wise operations, without actually copying the data.

Example: Broadcasting with Two Arrays

```
arr1 = np.array([1, 2, 3])
arr2 = np.array([10, 20, 30])

result = arr1 + arr2  # Element-wise addition
print(result)  # Output: [11 22 33]
```

NumPy automatically aligns the two arrays and performs element-wise addition, treating them as if they have the same shape.

Example: Broadcasting a 2D Array and a 1D Array

```
arr1 = np.array([[1, 2, 3], [4, 5, 6]])
arr2 = np.array([1, 2, 3])

result = arr1 + arr2  # Broadcasting arr2 across arr1
print(result)
# Output:
# [[2 4 6]
#  [5 7 9]]
```

In this case, `arr2` is broadcast across the rows of `arr1`, adding `[1, 2, 3]` to each row.

How Broadcasting Works

1. **Dimensions must be compatible**: The size of the trailing dimensions of the arrays must be either the same or one of them must be 1.
2. **Stretching arrays**: If the shapes are compatible, NumPy stretches the smaller array to match the larger one, element-wise, without copying data.

---

5. Hands-on: Applying Broadcasting to Real-World Scenarios

Let's apply broadcasting to a real-world scenario: **scaling data** in machine learning.

Example: Normalizing Data Using Broadcasting

Imagine you have a dataset where each row represents a sample and each column represents a feature. You can **normalize** the data by subtracting the mean of each column and dividing by the standard deviation.

```
# Simulating a dataset (5 samples, 3 features)
data = np.array([[10, 20, 30],
                 [15, 25, 35],
                 [20, 30, 40],
                 [25, 35, 45],
                 [30, 40, 50]])

# Calculating mean and standard deviation for each feature (column)
mean = data.mean(axis=0)
std = data.std(axis=0)

# Normalizing the data using broadcasting
normalized_data = (data - mean) / std

print(normalized_data)
```

In this case, broadcasting allows you to subtract the mean and divide by the standard deviation for each feature without needing loops or creating copies of the data.

Summary:

- **Loops are slow** because Python's interpreter adds overhead, making iteration less efficient.
- **Vectorization** allows you to apply operations to entire arrays at once, greatly improving performance by utilizing NumPy's optimized C backend.
- **Broadcasting** enables operations between arrays of different shapes by automatically stretching the smaller array to match the shape of the larger array, without creating additional copies.
- **Real-world use**: Broadcasting can be used in data science tasks, such as **normalizing datasets**, without sacrificing memory or performance.

Built in Mathematical Functions in NumPy

Here are some common NumPy methods that are frequently used for statistical and mathematical operations:

1. `np.mean()` – Compute the **mean** (average) of an array.

```
np.mean(arr)
```

2. `np.std()` – Compute the **standard deviation** of an array.

```
np.std(arr)
```

3. `np.var()` – Compute the **variance** of an array.

```
np.var(arr)
```

4. `np.min()` – Compute the **minimum** value of an array.

```
np.min(arr)
```

5. `np.max()` – Compute the **maximum** value of an array.

```
np.max(arr)
```

6. `np.sum()` – Compute the **sum** of all elements in an array.

```
np.sum(arr)
```

7. `np.prod()` – Compute the **product** of all elements in an array.

```
np.prod(arr)
```

8. `np.median()` – Compute the **median** of an array.

```
np.median(arr)
```

9. `np.percentile()` – Compute the **percentile** of an array.

```
np.percentile(arr, 50)  # For the 50th percentile (median)
```

10. `np.argmin()` – Return the **index of the minimum** value in an array.

```
np.argmin(arr)
```

11. `np.argmax()` – Return the **index of the maximum** value in an array.

```
np.argmax(arr)
```

12. `np.corrcoef()` – Compute the **correlation coefficient** matrix of two arrays.

```
np.corrcoef(arr1, arr2)
```

13. `np.unique()` – Find the **unique elements** of an array.

```
np.unique(arr)
```

14. `np.diff()` – Compute the **n-th differences** of an array.

```
np.diff(arr)
```

15. `np.cumsum()` – Compute the **cumulative sum** of an array.

```
np.cumsum(arr)
```

16. `np.linspace()` – Create an array with **evenly spaced numbers** over a specified interval.

```
np.linspace(0, 10, 5)  # 5 numbers from 0 to 10
```

17. `np.log()` – Compute the **natural logarithm** of an array.

```
np.log(arr)
```

18. `np.exp()` – Compute the **exponential** of an array.

```
np.exp(arr)
```

These methods are used for performing mathematical and statistical operations with NumPy

Data Structures in Python

Python provides powerful built-in data structures to store and manipulate collections of data efficiently.

---

1. Lists and List Methods

Lists are ordered, mutable (changeable) collections of items.

Creating a List:

```
numbers = [1, 2, 3, 4, 5]
mixed = [10, "hello", 3.14]
```

Common List Methods:

```
my_list = [1, 2, 3]

my_list.append(4)   # [1, 2, 3, 4]
my_list.insert(1, 99)  # [1, 99, 2, 3, 4]
my_list.remove(2)   # [1, 99, 3, 4]
my_list.pop()       # Removes last element -> [1, 99, 3]
my_list.reverse()   # [3, 99, 1]
my_list.sort()      # [1, 3, 99]
```

List Comprehensions (Efficient List Creation)

```
squared = [x**2 for x in range(5)]
print(squared)  # Output: [0, 1, 4, 9, 16]
```

---

2. Tuples and Operations on Tuples

Tuples are ordered but **immutable** collections (cannot be changed after creation).

Creating a Tuple:

```
my_tuple = (10, 20, 30)
single_element = (5,)  # Tuple with one element (comma required)
```

Accessing Tuple Elements:

```
print(my_tuple[1])  # Output: 20
```

Tuple Unpacking:

```
a, b, c = my_tuple
print(a, b, c)  # Output: 10 20 30
```

Common Tuple Methods:

Method	Description	Example	Output
`count(x)`	Returns the number of times `x` appears in the tuple	`(1, 2, 2, 3).count(2)`	`2`
`index(x)`	Returns the index of the first occurrence of `x`	`(10, 20, 30).index(20)`	`1`

```
my_tuple = (1, 2, 2, 3, 4)
print(my_tuple.count(2))  # Output: 2

print(my_tuple.index(3))   # Output: 3
```

Why Use Tuples?

- **Faster than lists** (since they are immutable)
- **Used as dictionary keys** (since they are hashable)
- **Safe from unintended modifications**

3. Sets and Set Methods

Sets are **unordered, unique collections** (no duplicates).

Creating a Set:

```
fruits = {"apple", "banana", "cherry"}
```

Key Set Methods:

```
my_set = {1, 2, 3, 4}

my_set.add(5)         # {1, 2, 3, 4, 5}
my_set.remove(2)      # {1, 3, 4, 5}
my_set.discard(10)    # No error if element not found
my_set.pop()          # Removes random element
```

Set Operations:

```
a = {1, 2, 3}
b = {3, 4, 5}

print(a.union(b))        # {1, 2, 3, 4, 5}
print(a.intersection(b))  # {3}
print(a.difference(b))   # {1, 2}
```

Use Case: Sets are great for eliminating duplicate values.

---

4. Dictionaries and Dictionary Methods

Dictionaries store key-value pairs and allow fast lookups.

Creating a Dictionary:

```
student = {"name": "Alice", "age": 21, "grade": "A"}
```

Accessing & Modifying Values:

```
print(student["name"])  # Output: Alice
student["age"] = 22     # Updating value
student["city"] = "New York"  # Adding new key-value pair
```

Common Dictionary Methods:

```
print(student.keys())    # dict_keys(['name', 'age', 'grade', 'ci
print(student.values())  # dict_values(['Alice', 22, 'A', 'New Yo
print(student.items())   # dict_items([('name', 'Alice'), ('age',

student.pop("age")  # Removes "age" key
student.clear()  # Empties dictionary
```

Dictionary Comprehensions:

```
squares = {x: x**2 for x in range(5)}
print(squares)  # {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

5. When to Use Each Data Structure?

Data Structure	Features	Best For
List	Ordered, Mutable	Storing sequences, dynamic data
Tuple	Ordered, Immutable	Fixed collections, dictionary keys

Data Structure	Features	Best For
Set	Unordered, Unique	Removing duplicates, set operations
Dictionary	Key-Value Pairs	Fast lookups, structured data

URBAN EDGE ___/___/___

Data Structures & Algorithms by CodeWithHarry

This course will get you prepared for placements and will teach you how to create efficient and fast algorithms.

Data structures and algorithms are two different things.

Data Structures: Arrangement of data so that they can be used efficiently in memory (data items)

Algorithms: Sequence of steps on data using efficient data structures to solve a given problem.

Other Terminology

Database :- Collection of information in permanent storage for faster retrieval and updation.

Data warehousing :- Management of huge amount of legacy data for better analysis.

Big data :- Analysis of too large or complex data which cannot be dealt with traditional data processing application.

Data Structures and Algorithms are nothing new. If you have done programming in any language like C you must have used Arrays → A data structure and some sequence of processing steps to solve a problem → Algorithm

URBAN EDGE

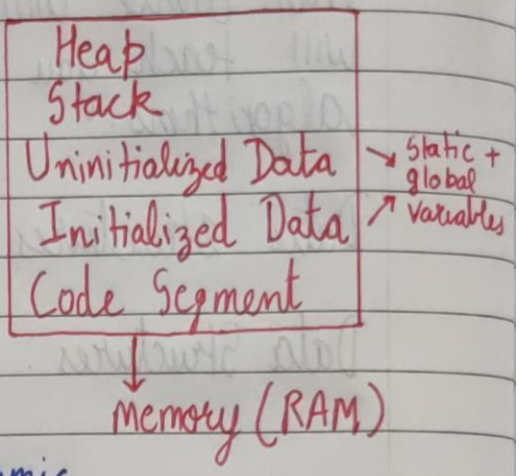
Memory layout of C programs

When the program starts, its Code is copied to the main memory.

Stack holds the memory occupied by the functions.

Heap contains the data which is requested by the program as dynamic memory.

Initialized and uninitialized data segments hold initialized and uninitialized global variables respectively



URBAN EDGE

Time Complexity & Big O notation

This morning I wanted to eat some pizzas; So I asked my brother to get me some from Dominos (3 km far)
He got me the pizza and I was happy only to realize it was too less for 29 friends who came to my house for a surprize visit!

My brother can get 2 pizzas for me on his bike but pizza for 29 friends is too huge of an input for him which he cannot handle.

2 pizzas ⟶  okay! not a big deal!

68 pizzas ⟶  Not possible! in short time

What is Time Complexity?
Time Complexity is the study of efficiency of algorithms.

(i) Time Complexity = How time taken to execute an algorithm grows with the size of the input!

Consider two developers who created an algorithm to sort n numbers. Shubham and Rohan did this independently.

When ran for input size n, following results were recorded:

no of elements (n)	Shubham's Algo	Rohan's Algo
10 elements	90 ms	122 ms
70 elements	110 ms	124 ms
110 elements	180 ms	131 ms
1000 elements	2 s	800 ms

We can see that initially Shubham's algorithm was shining for smaller input but as the number of elements increases Rohan's algorithm looks good!

Quick Quiz: Who's Algorithm is better?

Time Complexity: Sending GTA V to a friend
Let us say you have a friend living 5 kms away from your place. You want to send him a game.
Final exams are over and you want him to get this 60 GB file from you. How will you send it to him?
Note that both of you are using (JIO) 4G with 1 GB/day data limit.

The best way to send him the game is by delivering it to his house.
Copy the game to a Hard disk and send it!

Will yo do the same thing for sending a game like minesweeper which is in kbs of size?
No because you cand send it via internet.

As the file size grows, time taken by online sending increases linearly → $O(n^1)$
As the file size grows, time taken by physical sending remains constant. $O(n^0)$ or $O(1)$

Calculating Order in terms of Input size
In order to calculate the order, most impactful term containing n is taken into account.
↳ Size of input

Let us assume that formula of an algorithm in terms of input size n looks like this:

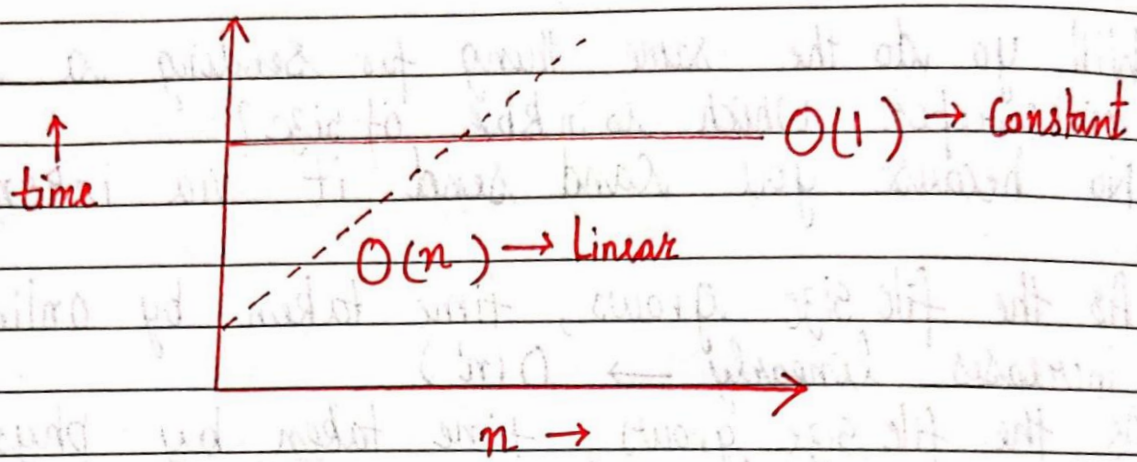
Algo 1 → $k_1 n^2 + k_2 n + 36$ ⇒ $O(n^2)$
($k_1 n^2$: Highest order term; $k_2 n + 36$: can ignore lower order terms)

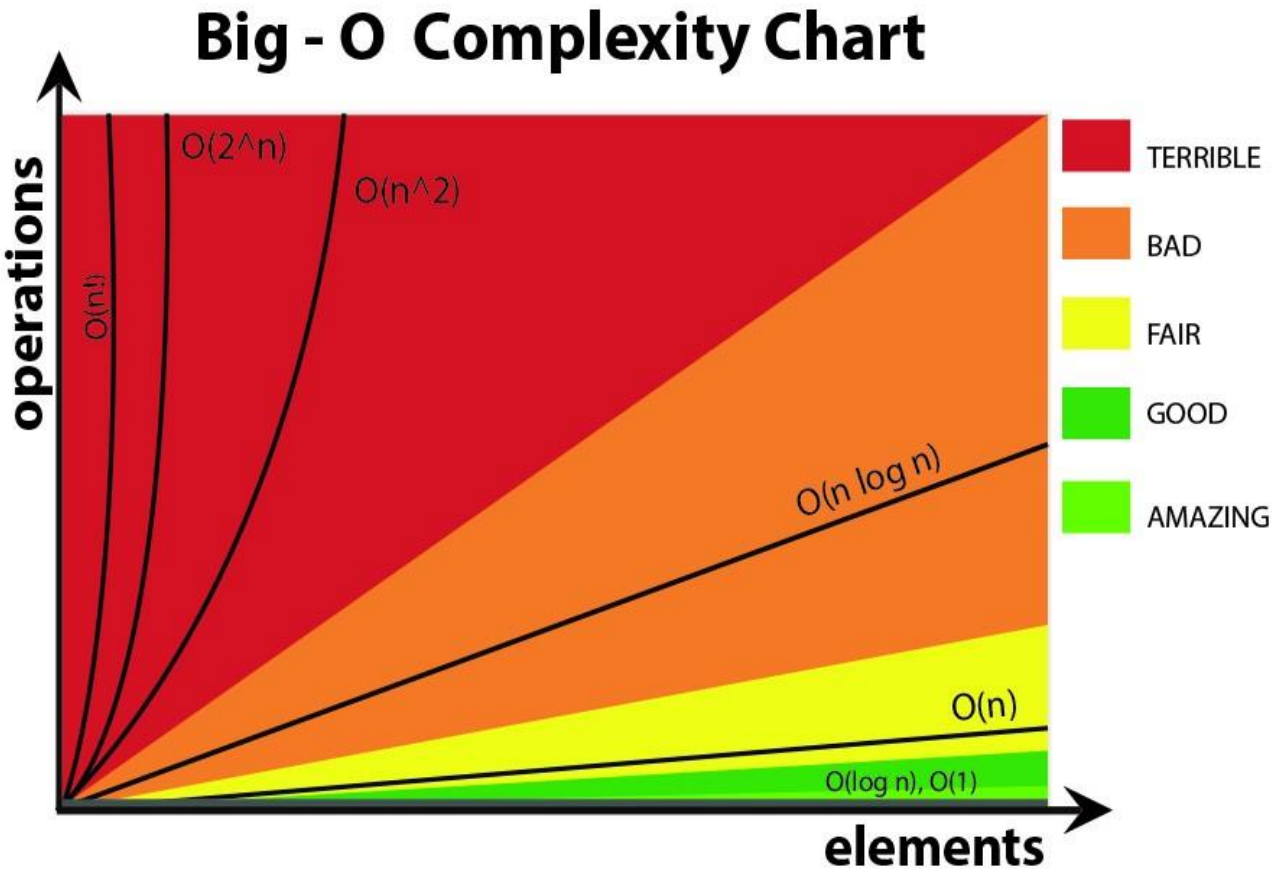
Algo 2 → $k_1 k_2^2 + k_3 k_2 + 8$
⇓
$k_1 k_2^2 n^0 + k_3 k_2 + 8$ ⇒ $O(n^0)$ or $O(1)$

Note that these are the formulas for time taken by them.

Visualising Big O

If we were to plot O(1) and O(n) on a graph, they will look something like this :





Source: https://stackoverflow.com/questions/3255/big-o-how-do-you-calculate-approximate-it

Asymptotic Notations

Asymptotic notations give us an idea about how good a given algorithm is compared to some other algorithm
Let us see the mathematical definition of "order of" now.

Primarily there are three types of widely used asymptotic notations.

1) Big oh notation (O)
2) Big omega notation (Ω)
3) Big theta notation (Θ) ⟶ Widely used one!

Big oh notation

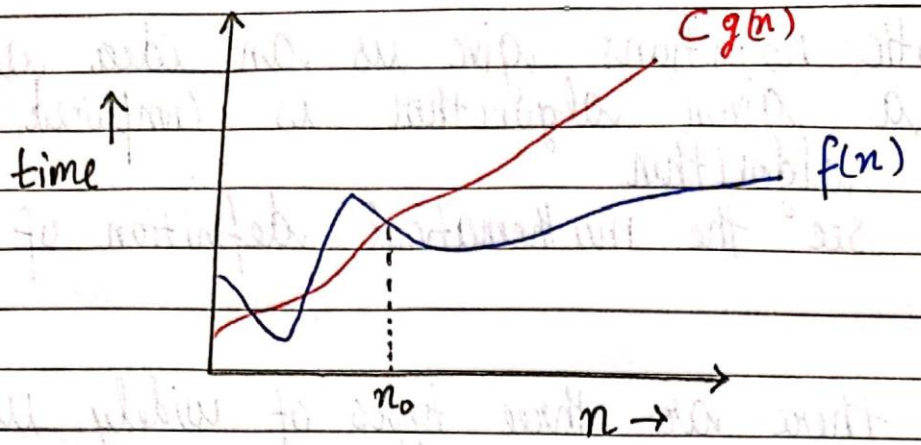
Big oh notation is used to describe asymptotic upper bound
Mathematically, if $f(n)$ describes running time of an algorithm; $f(n)$ is $O(g(n))$ iff there exist positive constants C and n_0 such that

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

⇓
used to give upper bound on a function.

if a function is $O(n)$, it is automatically $O(n^2)$ as well!

Graphic example for Big oh (O)



Big Omega notation

Just Like O notation provides an asymptotic upper bound, Ω notation provides asymptotic lower bound. Let $f(n)$ define running time of an algorithm;

$f(n)$ is said to be $\Omega(g n)$ if there exists positive constants C and n_0 such that

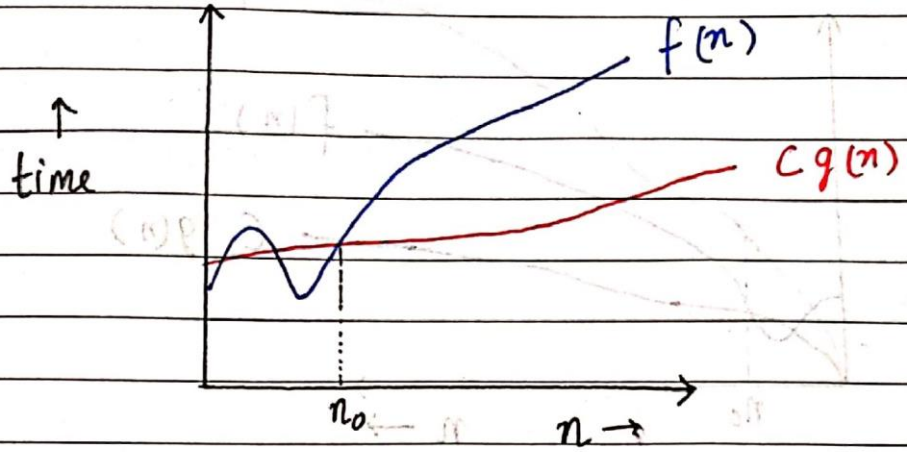
$$0 \leq c\,g(n) \leq f(n) \quad \text{for all } n \geq n_0$$

⇓ used to give lower bound on a function

if a function is $O(n^2)$ it is automatically $O(n)$ as well

URBAN EDGE

Graphic example for Big omega (Ω)



Big theta notation

Let $f(n)$ define running time of an algorithm

$f(n)$ is said to be $\Theta(gn)$ iff $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$

Mathematically,

$$0 \leq f(n) \leq C_1 g(n) \quad \forall\ n \geq n_0$$

$$0 \leq C_2 g(n) \leq f(n) \quad \forall\ n \geq n_0$$

Sufficiently large value of n

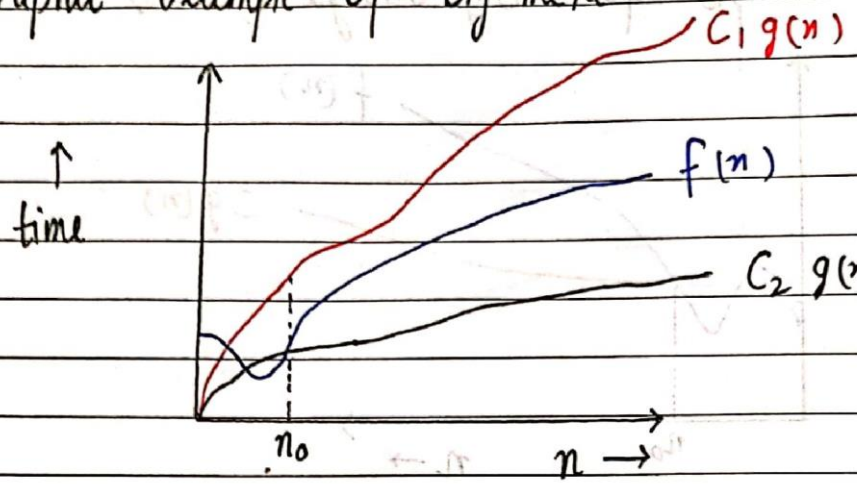
Merging both the equations, we get:

$$0 \leq C_2 g(n) \leq f(n) \leq C_1 g(n) \quad \forall\ n \geq n_0$$

The equation simply means there exist positive constants C_1 and C_2 such that $f(n)$ is sandwiched between $C_2 g(n)$ and $C_1 g(n)$

URBAN EDGE

Graphic example of Big theta



Which one of these to use ?
Since Big theta gives a better picture of runtime for a given algorithm, most of the interviewers expect you to provide an answer in terms of Big theta when they say "Order of".

Quick Quiz :- Prove that n^2+n+1 is $O(n^3)$, $\Omega(n^2)$ and $\Theta(n^2)$ using respective definitions.

Increasing order of common runtimes

$$1 < \log n < n < n\log n < n^2 < n^3 < 2^n < n^n$$

Better ⟵ ⇓ ⟶ Worse

Common runtimes from better to worse ⟶

URBAN EDGE

Best, worst and Expected Case

Sometimes we get lucky in life. Exams cancelled when you were not prepared, surprise test when you were prepared etc. ⟹ Best case

Some times we get unlucky. Questions you never prepared asked in exams, rain during sports period etc. ⇒ Worst case

But overall the life remains balance with the mixture of lucky and unlucky times. ⇒ Expected case.

Analysis of a search algorithm

Consider an array which is sorted in increasing order

1	7	18	28	50	180

We have to search a given number in this array and report whether its present in the array or not.

Algo 1 → Start from first element until an element greater than or equal to the number to be searched is found.

Algo 2 → Check whether the first or the last element is equal to the number. If not find the number between these two elements (center of the array). If the center element is greater than the number to be searched, repeat the process for first half else repeat for second half until the number is found.

Analyzing Algo 1

If we really get lucky, the first element of the array might turn out to be the element we are searching for. Hence we made just one comparison.

$$\text{Best case Complexity} = O(1)$$

If we are really unlucky, the element we are searching for might be the last one.

$$\text{Worst case Complexity} = O(n)$$

For calculating Average case time, we sum the list of all the possible case's runtime and divide it with the total number of cases.

⇓

Sometimes calculation of average case time gets very complicated

Analyzing Algo 2

If we get really lucky, the first element will be the only one which gets compared

$$\text{Best case Complexity} = O(1)$$

If we get unlucky we will have to keep dividing the array into halves until we get a single element (the array gets finished)

Worst case Complexity = $O(\log n)$

What log(n)? What is that

log(n) → Number of times you need to half the array of size n before it gets exhausted

$\log 8 = 3 \implies \frac{8}{2} \to \frac{4}{2} \to \frac{2}{2}$ → Cant break anymore.

$1 + 1 + 1$

$\log 4 = 2 \implies \frac{4}{2} \to \frac{2}{2}$ → Cant break anymore.

$1 + 1$

Log n simply means how many time I need to divide n units such that we cannot divide them (into halves) anymore.

Space Complexity

Time is not the only thing we worry about while analyzing algorithms. Space is equally important.

Creating an array of size n → O(n) space
↳ Size of input

If a function calls itself recursively n times its space complexity is O(n)

URBAN EDGE

Quick Quiz → Calculate Space Complexity of a function which calculates factorial of a given number n.

Why cant we calculate Complexity in seconds?

→ Not everyone's Computer is equally powerful

→ Asymptotic analysis is the measure of how time (runtime) grows with input

Techniques to Calculate Time Complexity

Once we are able to write the runtime in terms of size of the input (n), we can find the time complexity.

For example $T(n) = n^2 \Rightarrow O(n^2)$

$T(n) = \log n \Rightarrow O(\log n)$

Some tricks to calculate complexity

1. Drop the constants : Any thing you might think is $O(3n)$ is $O(n)$
   ↳ Better representation
2. Drop the non dominant terms : Any thing you represent as $O(n^2+n)$ can be written as $O(n^2)$
3. Consider all variables which are provided as input : $O(mn)$ & $O(mnq)$ might exist for some cases!

In most of the cases, we try to represent the runtime in terms of the input which can be more than one in number. For example :

Painting a park of dimension $m \times n \Rightarrow O(mn)$

Time Complexity – Competitive Practice Sheet

1. Fine the time complexity of the func1 function in the program show in program1.c as follows:

```c
#include <stdio.h>

void func1(int array[], int length)
{
    int sum = 0;
    int product = 1;
    for (int i = 0; i < length; i++)
    {
        sum += array[i];
    }

    for (int i = 0; i < length; i++)
    {
        product *= array[i];
    }
}

int main()
{
    int arr[] = {3, 5, 66};
    func1(arr, 3);
    return 0;
}
```

2. Fine the time complexity of the func function in the program from program2.c as follows:

```c
void func(int n)
{
    int sum = 0;
    int product = 1;
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            printf("%d , %d\n", i, j);
        }
    }

}
```

3. Consider the recursive algorithm above, where the random(int n) spends one unit of time to return a random integer which is evenly distributed within the range [0,n][0,n]. If the average processing time is T(n), what is the value of T(6)?

```
int function(int n)
{
    int i;

    if (n <= 0)
    {
        return 0;
    }
    else
    {
        i = random(n - 1);
        printf("this\n");
        return function(i) + function(n - 1 - i);
    }
}
```

4. Which of the following are equivalent to O(N)? Why?
    a) O(N + P), where P < N/9
    b) 0(9N-k)
    c) O(N + 8log N)
    d) O(N + M^2)

5. The following simple code sums the values of all the nodes in a balanced binary search tree. What is its runtime?

```
int sum(Node node)
{
    if (node == NULL)
    {
        return 0;
    }
    return sum(node.left) + node.value + sum(node.right);
}
```

6. Find the complexity of the following code which tests whether a give number is prime or not?

```
int isPrime(int n){
    if (n == 1){
        return 0;
    }

    for (int i = 2; i * i < n; i++) {
        if (n % i == 0)
            return 0;
    }
```

```
    return 1;
}
```

7. What is the time complexity of the following snippet of code?

```
int isPrime(int n){

    for (int i = 2; i * i < 10000; i++) {
        if (n % i == 0)
            return 0;
    }

    return 1;
}
isPrime();
```

URBAN EDGE

Operations on an Array

Following operations are supported by an array.

Traversal
Insertion
Deletion
Search

⇒ There can be many other operations one can perform on arrays as well!
eg: Sorting asc., Sorting desc.

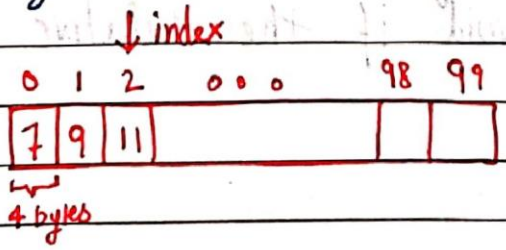
Traversal

Visiting every element of an array once → Traversal

Why traversal? → For use cases like:
→ Storing all elements → using scanf
→ Printing all elements → using printf

An important note about arrays
If we create an array of length 100 using a[100] in C language, we need not use all the elements. It is possible for a program to use just 60 elements out of these 100.
↳ But we cannot go beyond 100 elements.

An array can easily be traversed using a for loop in C language

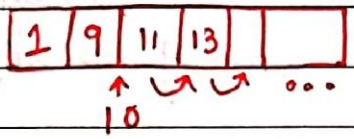


URBAN EDGE

Insertion

An element can be inserted in an array at a specified position.

In order for this operation to be successful, the array should have enough capacity

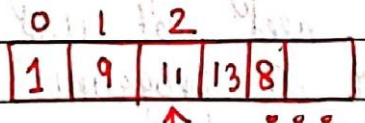


⇒ Elements need to be shifted to maintain relative order.

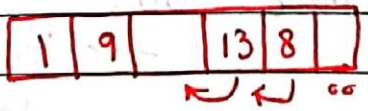
When no position is specified its best to insert the element at the end.

Deletion

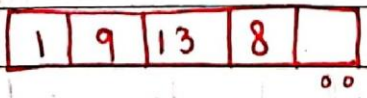
An element at specified position can be deleted creating a void which needs to be fixed by shifting all the elements to the left as follows:



Delete 11 at ind 2



Shift the elements

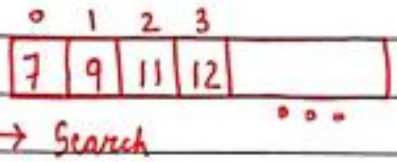


Deletion done!

We can also bring the last element of the array to fill the void if the relative ordering is not important

Searching

Searching can be done by traversing the array until the element to be searched is found



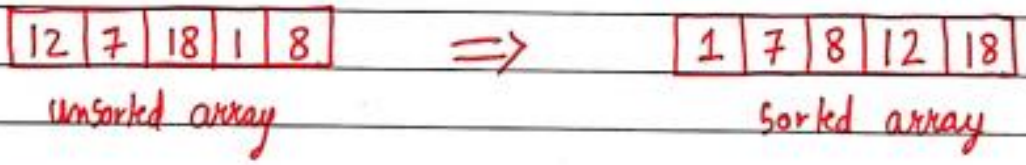
⇓

For sorted array time taken to search is much less than unsorted array!!

Sorting

Sorting means arranging an array in order (asc or desc)

We will see various sorting techniques later in the course.



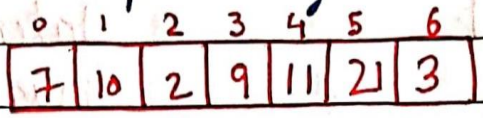
Unsorted array

Sorted array

Linear Vs Binary Search

Linear Search

Searches for an element by visiting all the elements sequentially until the element is found.



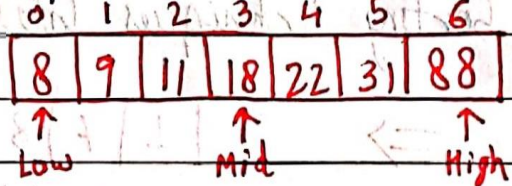
⇒ can be sorted or unsorted

Search '2' → Element found

WC Complexity : O(n)

Binary Search

Searches for an element by breaking the search space into half in a sorted array.



WC Complexity O(log n)

Search 18

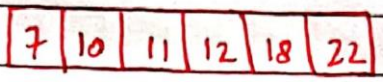
The search continues towards either side of mid based on whether the element to be searched is lesser or greater than mid.

	Linear Search	Binary Search
1.	Works on both sorted and unsorted arrays	Works only on sorted arrays
2.	Equality operations	inequality operations
3.	O(n) WC Complexity	O(log n) WC Complexity

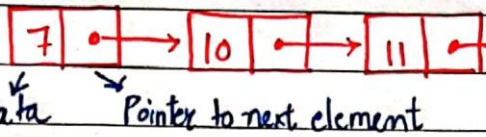
URBAN EDGE

Introduction to Linked Lists

Linked lists are similar to arrays (Linear data structures)



⇒ In Arrays elements are stored in contiguous memory locations



⇒ In linked lists, elements are stored in non contiguous memory locations

Why Linked Lists?

Memory and the capacity of an array remains fixed.
In case of linked lists, we can keep adding and removing elements without any capacity constraints

Drawbacks of Linked lists

→ Extra memory space for pointers is required (for every node 1 pointer is needed)
→ Random access not allowed as elements are not stored in contiguous memory locations.

Implementation

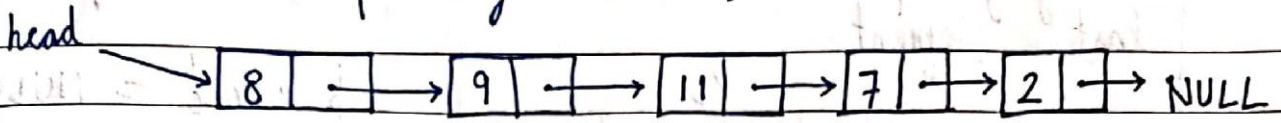
Linked list can be implemented using a structure in C language

```
Struct Node {
    int data;
    Struct Node * next;
};
```

⇒ Self refencing structure

Deletion in a Linked List

Consider the following Linked List

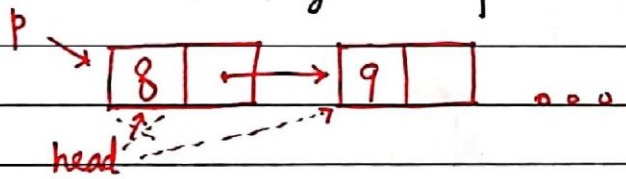


Deletion can be done for the following cases:

1. Deleting the first Node
2. Deleting the node at an index
3. Deleting the last Node
4. Deleting the first node with a given value.

The deletion just like insertion is done by rewiring the pointer connections, the only caveat being: we need to free the memory of the deleted node using free().

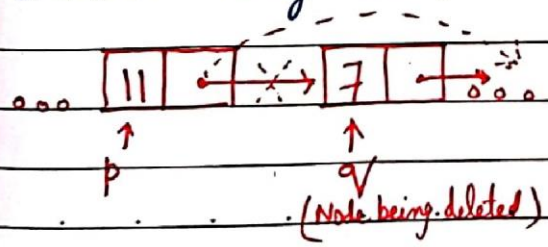
Case 1: Deleting the first node




```
Struct Node * p = head
head = head → next;
free(p);
```

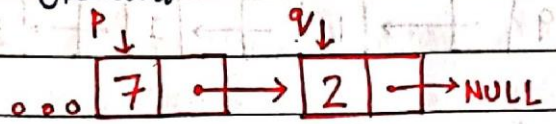
Case 2: Deleting the node at an index

for deleting a given node, we first bring a temporary pointer p before element to be deleted and q on the element being deleted



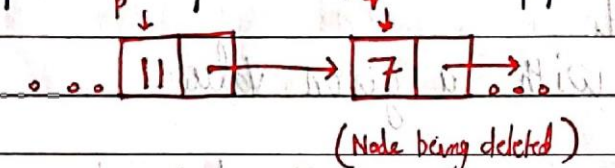

```
p → next = q → next;
free(q);
```

Case 3: Deleting the last Node
Last node can be deleted just like Case 2 by bringing p on second last element and q on last element.



p→next = NULL
free(q) → To free the memory!

Case 4: Delete the first node with a given value
This can be done exactly like Case 2 by bringing pointers p & q to appropriate positions

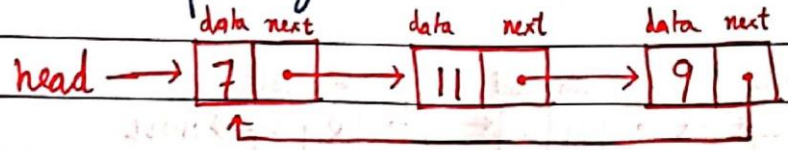


p→next = q→next;
free(q);

URBAN EDGE

Circular Linked List

A circular linked list is a linked list where the last element points to the first element (head) hence forming a circular chain



Operations on a circular linked List
operations on a circular linked lists can be performed exactly like a singly linked list.

Visit www.codewithharry.com for practice sets / code / more

URBAN EDGE

Doubly Linked List

In a doubly linked list, each node contains a data part along with the two addresses, one for the previous node and the other one for the next node

Implementation

A doubly linked list can be implemented in C language as follows:

```
struct Node {
    int data;
    struct Node * next;
    struct Node * prev;
};
```

Operations on a Doubly Linked List

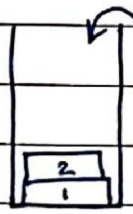
The insertion and deletion on a Doubly linked list can be performed by rewiring pointer connections just like we saw in a Singly linked list.

The difference here lies in the fact that we need to adjust two pointers (prev & next) instead of one (next) in the case of a Doubly linked list.

URBAN EDGE

Introduction to Stack Data Structure

Stack is a linear data structure. Operations on stack are performed in LIFO (Last in first out) order.



⇒ Item 2 which entered the basket last will be the first one to come out

LIFO (Last in first out)

Applications of Stack

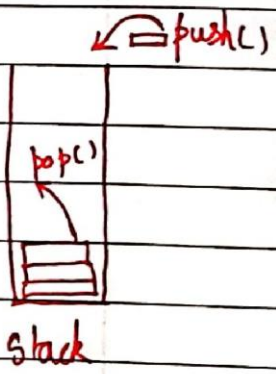
1. Used in function calls
2. Infix to postfix conversion (and other similar conversions)
3. Parenthesis matching & more...

Stack ADT

In order to create a stack we need a pointer to the topmost element along with other elements which are stored inside the stack.

Some of the operations of stack ADT are:

1. push() → push an element into the stack
2. pop() → remove the topmost element from the stack
3. peek(index) → Value at a given position is returned
4. isEmpty() / isFull() → Determine whether the stack is empty or full.



Implementation

A stack is a collection of elements with certain operations following LIFO (Last in First out) dicipline.

A Stack can be implemented using an array or a linked list