# Programming, Data Structures and Algorithms using Python


**Prof. Madhavan Mukund**
Director
Chennai Mathematical Institute

**Mr. Omkar Joshi**
Course Instructor
IITM Online Degree Programme

# Programming, Data Structures and Algorithms using Python

## Summary of weeks 1 to 3

# Content

- Timing our code
- Analysing an algorithm
- Asymptotic notations
- Orders of magnitude
- Calculating complexity
- Searching algorithms
- Selection sort
- Insertion sort
- Merge sort
- Quicksort
- Comparing sorting algorithms
- Lists vs. Arrays
- Implementation of lists in Python
- Implementation of dictionaries in Python

# Timing our code

```
import time
start = time.perf_counter()
…
# Execute some code
…
end = time.perf_counter()
time_elapsed = end – start
```

# Analysing an algorithm

- Two parameters to measure an algorithm
  - Running time (time complexity)
  - Memory requirement (space complexity)
- Running time $T(n)$ is a function of input size $n$
- Upper bound on worst case gives us an overall guarantee on performance

# Asymptotic notations

- Big O notation:

  $f(n)$ is $O(g(n))$ means $g(n)$ is an upper bound for $f(n)$

- $\Omega$ notation:

  $f(n)$ is $\Omega(g(n))$ means $g(n)$ is a lower bound for $f(n)$

- $\Theta$ notation:

  $f(n)$ is $\Theta(g(n))$ means we have found matching upper and lower bounds (optimal solution)

# Orders of magnitude

- Commonly encountered classes of functions
- In each case c is a positive constant and n increases without bound
- The slower-growing functions are listed first

| Notation | Name |
|---|---|
| $O(c)$ | Constant |
| $O(\log\log n)$ | Double logarithmic |
| $O(\log n)$ | Logarithmic |
| $O((\log n)^c), c > 1$ | Polylogarithmic |
| $O(n^c), 0 < c < 1$ | Fractional power |
| $O(n)$ | Linear |
| $O(\text{n} \log n)$ | Loglinear |
| $O(n^2)$ | Quadratic |
| $O(n^c)$ | Polynomial |
| $O(c^n), c > 1$ | Exponential |
| $O(n!)$ | Factorial |

# Calculating complexity

- Depends on the type of algorithm
- Iterative programs
  - Focus on loops
- Recursive programs
  - Write and solve a recurrence relation

# Searching algorithms

## Linear search

- Naïve solution
- Check every element in the list one by one
- Worst case is when element is not present in the list

| Best case | Average case | Worst case |
| --- | --- | --- |
| $O(1)$ | $O(n)$ | $O(n)$ |

## Binary search

- Prerequisite: list must be sorted
- Compare with midpoint element
- Halve the list till interval becomes empty
- Recurrence: $T(n) = T(n / 2) + 1$

| Best case | Average case | Worst case |
| --- | --- | --- |
| $O(1)$ | $O(\log n)$ | $O(\log n)$ |

# Selection sort

- It is an intuitive algorithm
- Repeatedly find the minimum/maximum element and append it to the sorted list
- Swapping elements helps us avoid use of second list
- Number of comparisons: $T(n) = n + (n - 1) + \ldots + 1$

  $= n(n + 1) / 2$
- The time complexity of Selection sort remains the same irrespective of the sequence of elements

# Insertion sort

- It is another intuitive algorithm
- Repeatedly pick an element and insert it into the sorted list
- Number of comparisons: $T(n) = n + (n - 1) + \ldots + 1$

$$= n(n + 1) / 2$$

- The time complexity of Insertion sort varies based on the sequence of elements

# Merge sort

- Divide the list into two halves
- Separately sort the left and right half
- Combine the two sorted lists A and B to get a fully sorted list C
  - If A is empty, copy B into C
  - If B is empty, copy A into C
  - Otherwise, compare first elements of A and B
  - Move the smaller of the two to C
  - Repeat till you exhaust A and B
- Recurrence: $T(n) = 2T(n / 2) + n$

# Quicksort

- Choose a pivot element (typically the first element)
- Partition the list into lower and upper parts with respect to the pivot
- Move the pivot between the lower and upper partition
- Recursively sort the two partitions
- This allows an in-place sort
- Iterative implementation is possible to avoid the cost of recursive calls

# Comparing sorting algorithms

| Parameter | Selection sort | Insertion sort | Merge sort | Quicksort |
| --- | --- | --- | --- | --- |
| **Best case** | $O(n^2)$ | $O(n)$ | $O(\mathrm{n} \log n)$ | $O(\mathrm{n} \log n)$ |
| **Average case** | $O(n^2)$ | $O(n^2)$ | $O(\mathrm{n} \log n)$ | $O(\mathrm{n} \log n)$ |
| **Worst case** | $O(n^2)$ | $O(n^2)$ | $O(\mathrm{n} \log n)$ | $O(n^2)$ |
| **In-place** | Yes | Yes | No | Yes |
| **Stable** | No | Yes | Yes | No |

# Lists vs. Arrays

| Lists | Arrays |
|---|---|
| Flexible length | Fixed size |
| Values are scattered in memory | Allocate a contiguous block of memory |
| Need to follow links to access | Random access |
| Insertion and deletion is easy | Insertion and deletion is expensive |
| Swapping elements takes constant time | Swapping elements takes linear time |

# Implementation of lists in Python

- Python lists are NOT implemented as flexible linked lists
- Underlying interpretation maps the list to an array
  - Assign a fixed block when you create a list
  - Double the size if the list overflows the array
- Keep track of the last position of the list in the array
  - l.append() and l.pop() are constant time, amortized – $O(1)$
  - Insertion/deletion require time $O(n)$
- Effectively, Python lists behave more like arrays than lists

# Implementation of dictionaries in Python

- A dictionary is implemented as a hash table
  - An array plus a hash function
- Creating a good hash function is important (and hard!)
- Need a strategy to deal with collisions
  - Open addressing/closed hashing – probe for free space in the array
  - Open hashing – each slot in the hash table points to a list of key-value pairs