# Algorithms

SF Coding Bootcamp

# What is an algorithm?

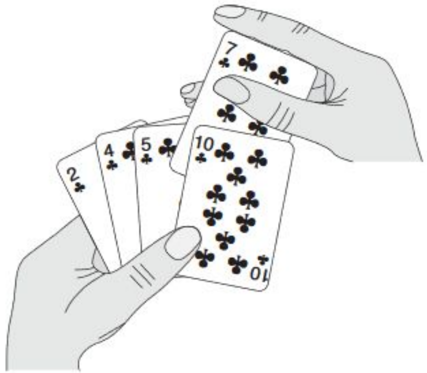
- A *precise* list of instructions to solve a specific problem
  - Bad: If the list is long, break it into two
  - Good: If the list’s length is longer than 100, divide the list into two equal parts

- This list of instructions must be implementable as a computer program
  - Bad: Do this if it’s hot outside

- The algorithm must solve the problem *for any legal input*.
  - Example: If the algorithm finds the shortest path in a map, then it must work for any map and any source and destination.

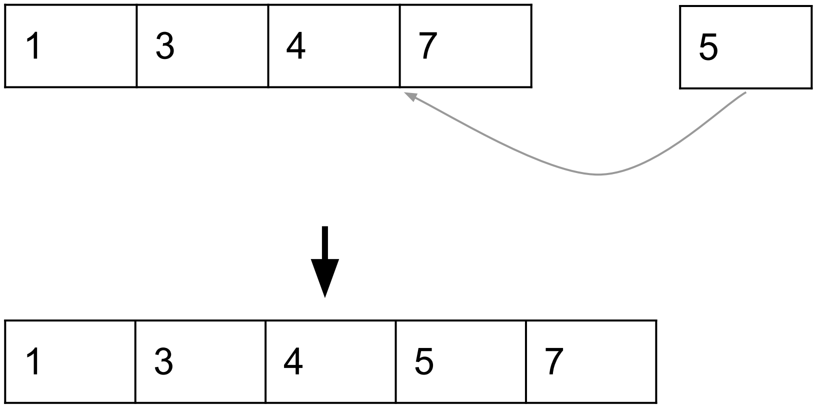
# Sorting via Insertion sort

- Let’s dive right into it: The problem that we will solve is to sort a list of ordered elements in an ascending order.
- We will solve this problem using the **Insertion Sort Algorithm.**

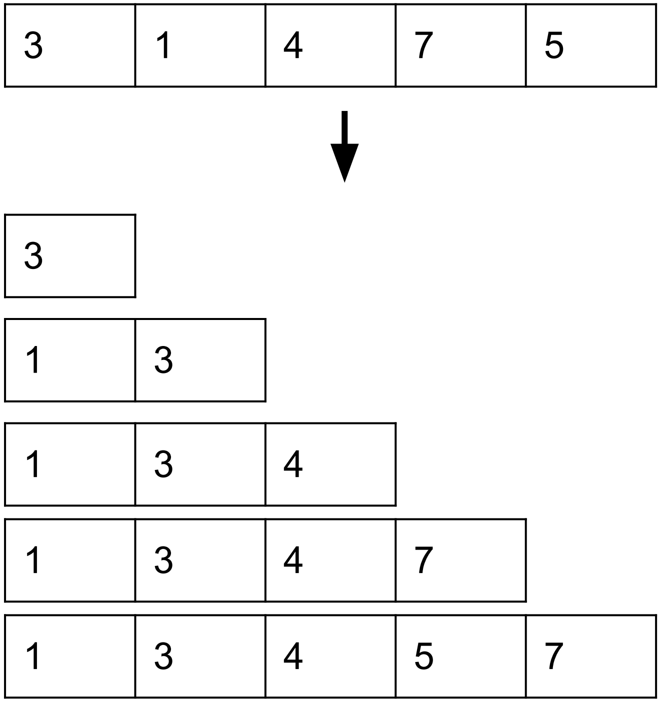


# Insertion sort: Python code

```python
def insert_sorted(sorted_list, element):
    for i, item in enumerate(sorted_list):
        if element <= item:
            sorted_list.insert(i, element)
            break
    else:
        sorted_list.append(element)
```



```python
def insertion_sort(unsorted_list):
    sorted_list = []
    for item in unsorted_list:
        insert_sorted(sorted_list, item)
    return sorted_list
```



# Time complexity

- The most important criteria for performance of an algorithm is its *running time*. This is called the **Time Complexity** of an algorithm.
- The running time is measured the *number of primitive steps* the algorithm needs to finish its work.
    - We can consider operations like assignment, addition, comparison, etc. as primitive steps. Be careful to not assume that calls to built-in functions and methods are primitive operations.
- The running time is expressed as a function of input size.
    - The definition of input size is problem dependent. In case of sorting, the input size is the number of elements that need to be sorted.
- The number of steps can be different for same input size.
    - Usually, we consider the *longest running time* possible as our indicator of running time.

# Analyze: insert_sorted()

```
def insert_sorted(sorted_list, element):
    for i, item in enumerate(sorted_list):
        if element <= item:
            sorted_list.insert(i, element)
            break
    else:
        sorted_list.append(element)
```

`sorted_list` has **k** elements.

We loop k times. In each iteration:

We do 1 comparison

insert() is called at most once. insert() can take upto k steps by itself.

append() is called at most once.
append() can be assumed to take 1 step.

Comparison steps: Upto k
Insert: At most once, takes upto k steps
Append: At most once, takes upto 1 steps
Total steps = Comparison steps + max(Insert steps, Append steps) = k + max(k, 1) = max(2 * k, 1)

# Analyze: insertion_sort()

```
def insertion_sort(unsorted_list):
    sorted_list = []

    for item in unsorted_list:

        insert_sorted(sorted_list, item)

    return sorted_list
```

`unsorted_list` has **n** elements.

We loop n times. In each iteration:

insert_sorted() takes 2 * len(sorted_list) steps. len(sorted_list) increases from 0 to n through the course of this loop.

1st iteration: 2 * 0 steps (actually 1 step)
2nd iteration: 2 * 1 steps
...
Kth iteration: 2 * k - 1 steps
…
Nth iteration: 2 * n - 1 steps

Total = 1 + 2 * (1 + 2 + 3 + … + n - 1) = 1 + n * (n - 1) steps

# Big O notation

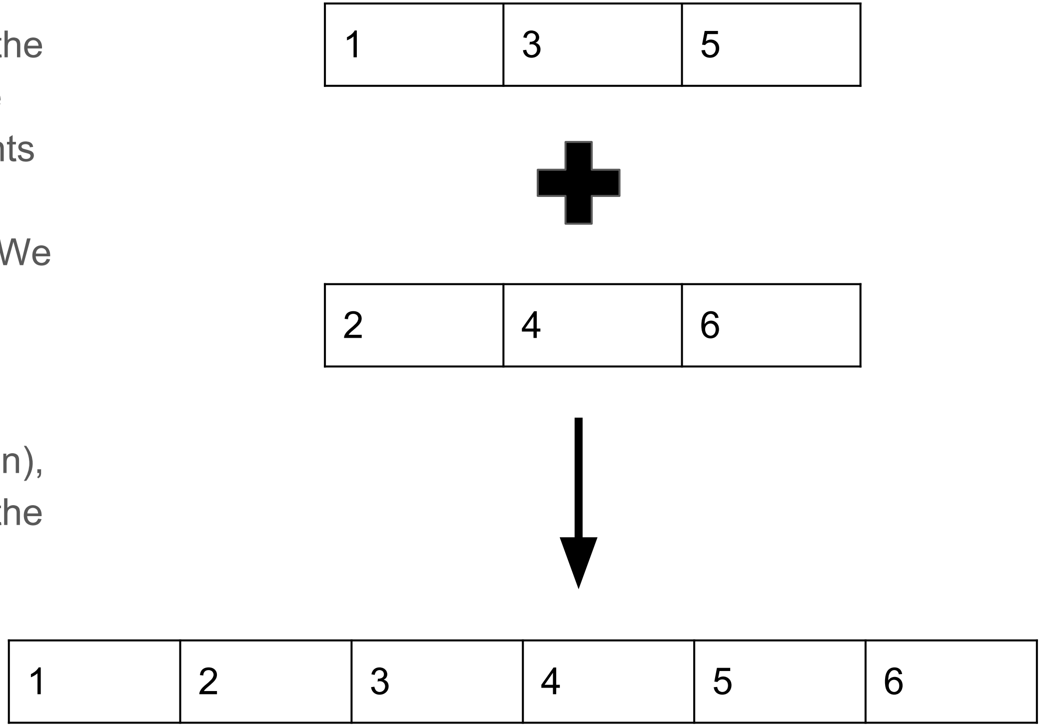
- In the last slide, we found that our implementation of insertion sort takes upto 1 + n * (n-1) steps.
- Usually we are not concerned with the exact number of steps. The most important consideration is how the running time grows as input size grows.
- We represent the worst case time in **O(f(n))** notation, where n is the input size, and f() is a function.
  - We say an algorithm has a time complexity of O(f(n)), if the algorithm always takes less than **C * f(n)** steps, for some constant C.
- In case of insertion sort: Running time = n^2 - n + 1. We can write this as O(n^2)
- Very Very important to understand this notation. It’s used by everyone talking about algorithms.

# Faster Sorting: Merge Sort

- Now that we have discussed the Insertion sort algorithm and understood that it's time complexity is O(n^2), we can ask the question: Is there a sorting algorithm that can run faster?
- The answer is yes. In fact there are tens of such algorithms.
- We are going to look at one of them - it's called Merge sort.
- Merge sort uses an approach called **Divide and Conquer**. The idea is to divide the problem into smaller sub-parts, solve the problem for those sub-parts, and then combine the results.

# Merge Sort: Merging sorted lists

- The essential element of merge sort is the ability to merge two sorted lists into one sorted list which contains all the elements of the two lists.
- The code for this is left as an exercise. We are going to call this function **`merge_sorted()`.**
- I leave it as another exercise for you to prove that it's time complexity is O(m + n), where **`m`** and **`n`** are the sizes of each of the lists to be merged.



# Merge Sort

## Divide

| | | | | | |
|---|---|---|---|---|---|
| 2 | 5 | 1 | 7 | 3 | 4 |

| | | |
|---|---|---|
| 2 | 5 | 1 |

| | | |
|---|---|---|
| 7 | 3 | 4 |

## Conquer sub-parts (recursively)

| | | |
|---|---|---|
| 2 | 5 | 1 |

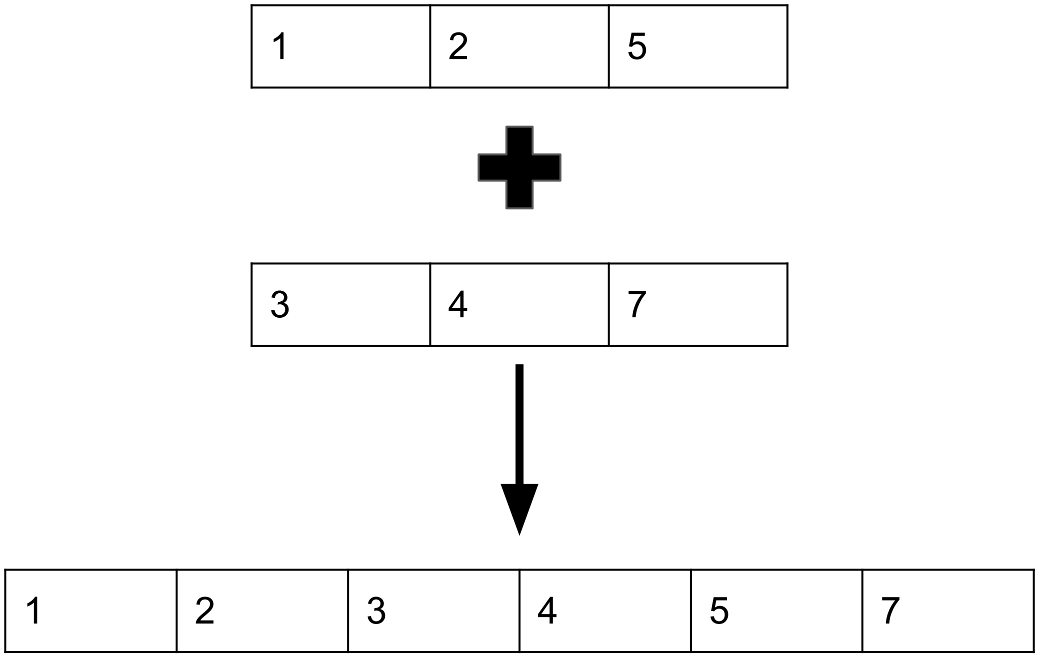
| | | |
|---|---|---|
| 1 | 2 | 5 |

| | | |
|---|---|---|
| 7 | 3 | 4 |

| | | |
|---|---|---|
| 3 | 4 | 7 |

# Merge Sort (contd)

**Merge sub-parts**



# Merge Sort Code

```
def merge_sort(alist):
    if len(alist) <= 1:
        return alist

    mid = len(alist) // 2
    left_sorted = merge_sort(alist[:mid])
    right_sorted = merge_sort(alist[mid:])

    return merge_sorted(left_sorted,  right_sorted)
```

To analyze merge sort, we have to try a new approach. We will denote the Time Complexity function of Merge Sort as **T(n)**.

The actual computation is done on the next slide. It is important to remember that `merge_sorted()` function itself has a time complexity of $O(m + n)$.

# Merge Sort Analysis

```
def merge_sort(alist):

    if len(alist) <= 1:
        return alist

    mid = len(alist) // 2
    left_sorted = merge_sort(alist[:mid])
    right_sorted = merge_sort(alist[mid:])


    return merge_sorted(left_sorted,  right_sorted)
```

`alist` has **n** elements.

We are doing one comparison here.

We are calling merge_sort() recursively on two lists that are half the size. Thus, these steps should take 2 * T(n/2) steps.

This step called merge_sorted() function, which takes O(n/2 + n/2) steps.

Total Steps: $T(n) = 1 + 2 * T(n/2) + C * n$

# Merge Sort Analysis (contd)

- We have arrived at a formula to calculate the time complexity of merge sort (this is simplified): $T(n) = 2 * T(n/2) + C * n$.
- This kind of an equation is called a **recurrence relation**.
- The answer to this equation turns out to be: $T(n) = O(n * \log_2 n)$.
- As shown in a table below, this is much faster than an algorithm which has the time complexity of $O(n ^ 2)$.

| **Input Size** | **N ^ 2** | **N * $\log_2$ N** |
|---|---|---|
| 100 | 10,000 | 664 |
| 10,000 | 100,000,000 | 132,877 |
| 1,000,000 | 1,000,000,000,000 | 19,931,568 |

# Bounds on sorting algorithms

- Considering that we were able to improve the time complexity of sorting algorithms from O(n^2) to $O(n * \log_2 n)$, it's fair to ask if we can keep going.
- The answer is no: A *general purpose* sorting algorithm must take $C * n * \log_2 n$ steps in the worst case. This has been proven.
- There are other sorting algorithms (like bucket sort) which can perform better for certain kinds of inputs, and in general, the research on sorting continues to try and find better algorithms for special use cases.

# Binary Search

- We are going to look at another algorithm which is very popular.
- The problem is as follows: Check if a number k is present in a sorted list of numbers. If it is present, return the position at which it is present.
- Note that the list in which we are searching is sorted. If that were not so, there is no way to optimize the solution to better than the trivial solution.
    - What is the time complexity of the trivial solution?

# Binary Search solution

Find if 5 is present in the list:

| | | | | | |
|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 7 |

Check if 5 is in first half or second half. Since 3 < 5, 5 is in the second half. The problem becomes:

Find if 5 is present in the list:

| | | |
|---|---|---|
| 4 | 5 | 7 |

Check if 5 is in first half or second half. Since 4 < 5, 5 is in the second half. The problem becomes:

Find if 5 is present in the list:

| | |
|---|---|
| 5 | 7 |

Then we find 5 in the next check.

# Binary Search code

```
def binary_search(k, alist, left=0, right=-1):
    if not alist:
        return -1

    if right < left:
        right = len(alist)

    if (right - left) == 1:
        if alist[left] == k:
            return left
        else:
            return -1

    mid = (left + right) // 2
    if k < alist[mid]:
        return binary_search(k, alist, left, mid)
    else:
        return binary_search(k, alist, mid, right)
```

As you can see, we check the middle element, and based on that comparison the binary_search() function recursively with only a subpart of the list.

In the analysis, we will again assume that the time complexity of binary search is **T(n)**.

# Binary Search analysis

```
def binary_search(k, alist, left=0, right=-1):
    if not alist:
        return -1

    if right < left:
        right = len(alist)

    if (right - left) == 1:
        if alist[left] == k:
            return left
        else:
            return -1

    mid = (left + right) // 2
    if k < alist[mid]:
        return binary_search(k, alist, left, mid)
    else:
        return binary_search(k, alist, mid, right)
```

`alist` has **n** elements.

We are doing one comparison here.

We are doing one more comparison here.

We are doing another comparison here.

We have an assignment step and another comparison step.

The recursive call costs T(n/2) steps.

Total Steps: $T(n) = T(n/2) + 5$

# Binary Search analysis (contd)

- From the previous slide: we get the recurrence relation: $T(n) = T(n/2) + C$, where C is a constant.
- The answer to this recurrence relation is $T(n) = C * \log_2 n$. Thus, binary search has the time complexity of $O(\log_2 n)$.
- This is much faster than the trivial search algorithm which has the time complexity of $O(n)$.

# Faster searching - using Python dictionary

- We just saw that if a list is sorted, it helps us perform a search inside it much faster ($O(\log_2 n)$ vs $O(n)$).
- Is there any other way we can store our data which helps us perform search even faster? As it happens, the answer is yes, and we already know about it.
- A data structure called Hash Table is designed to perform a search in $O(1)$ (i.e., constant time).
- But this is not the worst case time compexity - it is the average case time. Still, they perform very well in practice and are widely used.
- Python's dictionary data type uses a Hash Table in its implementation. That means, you can store all your data elements in a Python dictionary, and a membership check will only take $O(1)$ in the average case.