# Programming, Data Structures, and Algorithms in Python: Week 4

This document provides a detailed summary of the concepts covered in the fourth week, focusing on efficient O(nlogn) sorting algorithms, new data structures including tuples and dictionaries, and advanced features of Python function definitions.

## 1. Efficient Sorting Algorithms

We move beyond the simple, intuitive, but inefficient O(n2) sorting algorithms like Selection Sort and Insertion Sort to more advanced, recursive strategies that dramatically improve performance on larger datasets.

### Merge Sort

Merge Sort is a classic example of a **divide and conquer** algorithm. This powerful paradigm involves breaking a problem down into smaller, independent subproblems, solving them recursively, and then efficiently combining their solutions to solve the original problem.

* **Strategy:**
  1. **Divide:** The unsorted list is divided into two halves. This is a trivial operation that simply involves calculating a midpoint.
  2. **Conquer:** Each half is sorted recursively by calling Merge Sort on it. The recursion continues, repeatedly halving the lists until they consist of 0 or 1 elements. A list of this size is trivially sorted, providing the **base case** that stops the recursion.
  3. **Combine:** The two now-sorted halves are **merged** back into a single, comprehensively sorted list.
* **The Merge Function:** This is the core of the algorithm's work. It takes two already sorted lists and meticulously combines them into a single sorted list. It works by maintaining a pointer at the beginning of each list and iteratively comparing the elements. The smaller of the two elements is appended to the result list, and the corresponding pointer is advanced. This linear scan continues until one list is exhausted, at which point the remainder of the other list is simply appended. This entire process is linear, taking O(m+n) time for lists of size m and n.
* **Analysis:**
  + The recurrence relation for Merge Sort is T(n)=2T(n/2)+O(n). This means the time to sort n elements is the time to sort two lists of size n/2, plus the linear-time work required to merge them.
  + Solving this recurrence shows that Merge Sort has a reliable worst-case time complexity of O(nlogn). This logarithmic factor comes from the number of times the list can be halved. The significant improvement over O(n2) makes it possible to sort much larger lists (e.g., 100,000+ elements) in a reasonable amount of time.
* **Shortcomings:**
  + **Extra Space:** The primary drawback is that the merge step requires a new list to be created to store the merged result. This means that to sort a list of size n, we need additional space proportional to n, effectively doubling the memory usage.
  + **Recursion Overhead:** While elegant, deep recursion can have a performance cost. Each function call requires saving the current state (local variables) onto a call stack, which consumes memory and processing time.

### Quicksort

Quicksort is another divide and conquer algorithm that aims to improve on Merge Sort's space complexity by cleverly sorting the list **in-place**.

* **Strategy:**
  1. **Choose a Pivot:** Select an element from the list. A common, simple strategy is to pick the first element.
  2. **Partition:** This is the key step. The list is rearranged in-place such that all elements smaller than the pivot are moved to its left, and all elements larger are moved to its right. The pivot itself is placed between these two sub-lists, finding its final sorted position. This is achieved without creating any new lists.
  3. **Conquer:** Recursively apply Quicksort to the sub-list of smaller elements and the sub-list of larger elements.
* **Analysis:**
  + **Best/Average Case:** If the pivot consistently divides the list into roughly equal halves (as the median would), the recurrence is T(n)=2T(n/2)+O(n), leading to an O(nlogn) complexity. Averaged over all possible input permutations, the performance is excellent.
  + **Worst Case:** The performance degrades significantly if the pivot is consistently the smallest or largest element. This occurs, for example, with an already-sorted list when the first element is chosen as the pivot. The list is then partitioned into a subproblem of size n−1 and an empty one. This leads to the recurrence T(n)=T(n−1)+O(n), which results in an O(n2) complexity.
* **Randomization:** To mitigate the risk of the worst-case scenario, practical implementations often choose the pivot randomly from the list. This makes the probability of encountering a worst-case partition vanishingly small, ensuring an expected running time of O(nlogn).
* **Stability:** A sorting algorithm is **stable** if it preserves the original relative order of elements that have equal values. Quicksort, as described, is **not stable** because the partitioning process involves long-distance swaps that can change the relative order of equal elements. Merge Sort and Insertion Sort, by contrast, are stable.

## 2. Tuples and Dictionaries

### Tuples

A tuple is a sequence of values, similar to a list, but with one crucial difference that defines its use.

* **Syntax:** Defined using round brackets (). Example: point = (3.5, 4.8).
* **Immutability:** Tuples are **immutable**. Once a tuple is created, its contents cannot be changed. You cannot reassign an item, e.g., point[0] = 1.0 will raise a TypeError. This guarantees that a tuple's value remains constant.
* **Use Cases:** Tuples are ideal for representing fixed collections of items, like coordinates, dates, or database records, where the data is conceptually a single, unchangeable unit. Their immutability is essential because it allows them to be used as keys in dictionaries.

### Dictionaries (dict)

A dictionary is a highly flexible data structure that stores **key-value pairs**. It generalizes the concept of a list by allowing keys to be almost any immutable type, not just sequential integer indices.

* **Syntax:** Defined using curly braces {}. Example: scores = {'Dhawan': 84, 'Kohli': 200}.
* **Keys and Values:** Each entry consists of a unique key that maps to a value, providing a way to associate related pieces of information.
* **Mutability:** Dictionaries are **mutable**. You can dynamically add new key-value pairs, update the values of existing keys, or delete entries.
  + scores['Pujara'] = 16 adds a new entry or updates the value if 'Pujara' already exists.
* **Immutable Keys:** Keys must be of an immutable type (e.g., int, str, tuple). A list or another dict cannot be used as a key because their values can change, which would make the key unreliable.
* **No inherent order:** Traditionally, the key-value pairs in a dictionary are not stored in any predictable order. This is a trade-off for highly efficient (O(1) on average) lookups. While recent Python versions (3.7+) preserve insertion order, it's best practice not to rely on this for compatibility and conceptual clarity.

### Working with Dictionaries

* **d.keys():** Returns a view object displaying a list of the dictionary's keys.
* **d.values():** Returns a view object displaying a list of the values.
* **d.items():** Returns a view object displaying a list of key-value tuple pairs. This is very useful for iterating over both keys and values simultaneously.
* **k in d:** Checks for the presence of a key k, returning True or False.
* **Iteration:** The standard way to iterate is over the keys, but iterating over items is often more convenient:  
  for player, score in scores.items():  
   print(player, score)

## 3. Advanced Function Definitions

Python offers several features to make function definitions more flexible and powerful.

* **Passing Arguments by Name:** You can pass arguments by specifying the parameter name (keyword arguments), which allows you to supply them in any order. power(n=5, x=4) is equivalent to power(4, 5).
* **Default Arguments:** You can provide default values for parameters directly in the function definition. These parameters become optional when the function is called.  
  def connect(host, port, timeout=30):  
   # ...  
  + connect('server.com', 8080) is interpreted as connect('server.com', 8080, 30).
  + Default arguments are resolved by position and must come after all non-default arguments in the function signature.
* **Functions as Values:** In Python, functions are "first-class objects." This means they can be treated like any other value: assigned to names, passed as arguments into other functions, and returned as the result from functions.
  + g = f makes g another name for the function f.
  + This is incredibly useful for creating generic, customizable functions. For example, a sort function can accept a custom comparison function to define a non-standard sort order.

## 4. Operating on Lists: map, filter, and List Comprehensions

These are powerful, functional-style tools for transforming and selecting data from lists concisely.

* **map(function, list):** Applies a function to every element of a list. In Python 3, it returns a map object, which is an iterator. This is a form of "lazy evaluation"—it doesn't compute the results until they are requested, making it very memory-efficient.
* **filter(property\_function, list):** Applies a property\_function (which must return True or False) to every element. It returns a filter object (also an iterator) containing only the elements for which the function returned True.
* **List Comprehension:** This is a more readable and idiomatic "Pythonic" way to achieve the functionality of map and filter. It provides a concise, declarative syntax for creating a new list based on an existing one.
  + **Syntax:** [expression for item in list if condition]
  + **Example:** To get the squares of even numbers from 0 to 99:  
    [x\*x for x in range(100) if x % 2 == 0]
  + **Components:**
    1. An expression (x\*x) to apply to each item that passes the filter (this is the map part).
    2. A for clause to generate items from a source list (the generator).
    3. An optional if clause to select which items to process (this is the filter part).

List comprehensions can also have multiple for clauses for nested loops, making them highly effective for tasks like initializing multi-dimensional lists or generating combinations of values.