# Programming, Data Structures, and Algorithms in Python: Week 6

This document covers the key topics from Week 6, focusing on advanced algorithmic techniques and data structures. These include the backtracking paradigm for solving search problems, the scope of names in Python, and specialized data structures like sets, stacks, queues, and priority queues with heaps.

## 1. Backtracking

Backtracking is a powerful algorithmic technique for solving problems that involve searching through a large, complex set of possibilities to find a solution. It's an essential tool for constraint satisfaction problems, where solutions must adhere to a specific set of rules. Think of it as navigating a maze: you follow a path, and when you hit a dead end, you retrace your steps to the last junction and try a different path.

* **Core Idea:** The process is typically implemented recursively. It explores the "state space" of a problem, which is the set of all possible configurations.
  1. **Build a Solution Path:** Start from an empty or initial state and incrementally extend a candidate solution, making one choice at a time. Each choice moves you deeper into the state space tree.
  2. **Check Constraints:** At each step, verify if the current partial solution is valid according to the problem's rules. This is the "pruning" step: if a path is invalid, you immediately discard it and all other paths that could branch from it, saving significant computation.
  3. **Advance or Backtrack:**
     + If the current path is valid and can be extended, make a recursive call to explore the next step.
     + If the current path hits a "dead end"—it either violates a constraint or cannot be extended to a full solution—the function returns. The calling function then undoes the choice it just made (the "backtrack") and tries the next available option at its level.
  4. **Exhaust Possibilities:** If all options at a certain step have been tried and none have led to a solution, the function returns, signaling failure at that level and triggering a backtrack in the calling function.

### The N-Queens Problem

A classic example of backtracking is the N-Queens problem: *place N chess queens on an N×N chessboard so that no two queens threaten each other*. This means no two queens can be on the same row, column, or diagonal.

* **Strategy:** The problem's constraints simplify the search space. Since no two queens can be in the same row, we can deduce that a solution must have exactly one queen per row. This allows us to structure our search row by row.
  1. Start by attempting to place a queen in the first column of row 0.
  2. For the current row i, iterate through each column c from 0 to N-1 to find a "safe" square (i, c). A square is safe if it is not under attack by any of the queens placed in previous rows (0 to i-1).
  3. If a safe column c is found:
     + Record the placement: a queen is at (i, c).
     + Make a recursive call to place a queen in the next row, i+1.
     + If this recursive call eventually returns True, it means a full solution was found. We can propagate this success by also returning True.
  4. If the recursive call returns False, it signifies that the placement at (i, c) led to a dead end in a subsequent row. **This is the backtrack step**: undo the move by removing the queen from (i, c) (conceptually, freeing up the column and diagonals it occupied) and continue the loop to try the next column in the current row i.
  5. If the loop for row i finishes without finding any column that leads to a solution, it means the board configuration from the previous rows is flawed. The function returns False to trigger a backtrack in the calling function for row i-1.
* **Efficient Board Representation:** Constantly scanning the entire board to check if a square is safe is inefficient (O(N2) per check). A much better approach is to use arrays to track the attack state of columns and diagonals, providing O(1) checking.
  + **Queens:** A single list queens where queens[i] = c stores the column c of the queen in row i.
  + **Attacks:** Boolean or integer arrays can track occupied lines:
    - cols: cols[c] is true if column c is occupied.
    - diag1: For diagonals running top-left to bottom-right, all squares (r, c) share a constant value for r-c.
    - diag2: For diagonals running top-right to bottom-left, all squares (r, c) share a constant value for r+c.  
      By maintaining these arrays, checking if (r, c) is safe becomes a simple lookup: cols[c], diag1[r-c], and diag2[r+c].

## 2. Scope of Names and Global Variables

The **scope** of a name (variable) is the region of the code where it can be accessed directly. Understanding scope is crucial for avoiding bugs related to variable access.

* **Local Scope:** By default, any name that is assigned a value *inside* a function (e.g., x = 10) becomes local to that function. It is created when the function is called and is destroyed when the function returns.
* **Global Scope:** Names defined at the top level of a script or module exist in the global scope and can be accessed from anywhere in the code.

### How Python Looks Up Names

When a name is used inside a function, Python searches for it according to the **LEGB rule** (Local, Enclosing, Global, Built-in):

1. **Local:** It first checks if the name has been assigned a value within the current function.
2. **Enclosing:** If the current function is nested inside another function, Python checks the local scope of the enclosing function(s), moving outwards.
3. **Global:** If not found in any local or enclosing scopes, it checks the module's global scope.
4. **Built-in:** Finally, it checks the set of Python's built-in functions like len(), print(), etc.

A critical subtlety arises when *reading* versus *writing* to a global name from within a function:

* **Reading:** If a global name is only read, Python's lookup will find it in the global scope without issue.
* **Writing (Immutable types):** If you assign a value to a name that also exists globally (e.g., count = 1), and the type is **immutable** (like an integer, string, or tuple), Python assumes you are creating a *new local variable* with that name. This local variable "shadows" the global one, and any attempt to read the variable before this local assignment will result in an UnboundLocalError. This is a safety feature to prevent accidental modification of global state.
* **Writing (Mutable types):** If you modify a global name of a **mutable** type (like a list or dictionary) using its methods (e.g., my\_list.append(x) or my\_dict[key] = val), you are not performing an assignment to the *name* itself. You are accessing the global object (via reference) and changing its internal contents, which is permitted and will affect the object globally.

### The global Keyword

To explicitly tell a function that an assignment should modify a global immutable variable instead of creating a new local one, you must declare the name with the global keyword.

count = 0  
  
def increment\_counter():  
 # Without the next line, this would create a new local 'count'  
 # and raise an UnboundLocalError.  
 global count  
 count = count + 1  
  
increment\_counter()  
print(count) # Output: 1

## 3. Generating Permutations

A permutation is an arrangement of items in a specific order. Generating permutations is a common subproblem in backtracking. A key task is to find the **next lexicographical permutation**—the very next arrangement in dictionary order.

* **Algorithm to find the next permutation of a sequence p:**
  1. **Find the pivot:** Scan the sequence from right to left to find the first element p[k] that is smaller than the element to its right, p[k+1]. This element p[k] is our "pivot". The suffix starting from p[k+1] is in descending order and is the longest such suffix.
  2. **Check for the last permutation:** If no such pivot k is found, the entire sequence is in descending order, meaning it's the last possible permutation (e.g., [3, 2, 1]).
  3. **Find the successor:** Scan the descending suffix (from p[k+1] onwards) from right to left to find the first element p[m] that is larger than the pivot p[k]. This is the smallest element in the suffix that is still larger than the pivot.
  4. **Swap:** Swap the pivot p[k] with its successor p[m].
  5. **Reverse the suffix:** Reverse the entire suffix starting *after* the original pivot position k. Since this suffix was in descending order, reversing it puts it into ascending order, making it as small as possible. This ensures the new arrangement is the immediate next permutation.

## 4. Specialized Data Structures

### Sets

A set is an unordered collection of **unique** elements. Internally, sets are implemented using hash tables, making them highly optimized for membership testing (in), insertion, and deletion, with an average time complexity of O(1) for these operations.

* **Syntax:** {1, 2, 3}. An empty set must be created with set(), as {} creates an empty dictionary.
* **Use Cases:** Use a set when the order of elements doesn't matter, but you need fast membership checking and want to ensure there are no duplicates.
* **Operations:** Sets support standard mathematical operations efficiently:
  + **Union (|):** All elements from both sets.
  + **Intersection (&):** Elements present in both sets.
  + **Difference (-):** Elements in the first set but not in the second.
  + **Symmetric Difference (^):** Elements in either set, but not in both.

### Stacks (LIFO)

A stack is a Last-In, First-Out (LIFO) data structure. The last element added is the first one to be removed.

* **Operations:** push (add to top), pop (remove from top).
* **Implementation with Python Lists:** A standard list works perfectly as a stack.
  + push is list.append().
  + pop is list.pop().  
    Both operations are efficient, taking amortized O(1) time.
* **Applications:** Stacks are implicitly used by programming languages to manage function calls (the "call stack"). They are also essential for algorithms like Depth-First Search (DFS) and for parsing mathematical or programming expressions (e.g., checking for balanced parentheses).

### Queues (FIFO)

A queue is a First-In, First-Out (FIFO) data structure. The first element added is the first one to be removed.

* **Operations:** enqueue (add to rear), dequeue (remove from front).
* **Implementation:** Using a standard Python list for a queue is **inefficient**. While adding to the rear with list.append() is fast, removing from the front with list.pop(0) is slow (O(n)) because it requires shifting all subsequent elements. The correct tool is the collections.deque object, which is a doubly-linked list providing fast O(1) appends and pops from both ends.
* **Applications:** Queues are fundamental for algorithms that explore layer by layer, such as Breadth-First Search (BFS), and for managing tasks in scheduling systems.

## 5. Priority Queues and Heaps

A priority queue is an abstract data structure where each element has an associated "priority." When removing an element, the one with the highest priority is returned, regardless of when it was added.

### Heaps

A heap is a specialized tree-based data structure that provides an efficient implementation of a priority queue. It balances the work of insertion and deletion effectively.

* **Heap Properties (Max-Heap):**
  1. **Shape Property:** It is a **complete binary tree**. This means all levels are fully filled, except possibly the last, which is filled strictly from left to right. This regular structure allows a heap to be stored perfectly in a simple list or array, where for an element at index i, its children are at 2\*i + 1 and 2\*i + 2, and its parent is at (i-1) // 2.
  2. **Heap Property:** The value of each node is greater than or equal to the value of its children. This property guarantees that the maximum element in the heap is always at the root (index 0).
* **Operations (**O(logn)**):**
  + **insert:** Add the new element to the first available spot at the bottom of the tree (the end of the list) to maintain the shape property. Then, "bubble it up" (or "sift-up") by repeatedly swapping it with its parent as long as it is larger than its parent, until the heap property is restored. This takes at most O(logn) swaps.
  + **delete-max:** The maximum element is at the root. To remove it, swap it with the last element in the heap and remove the last element. The new root is likely out of place, so "bubble it down" (or "sift-down") by repeatedly swapping it with its largest child until the heap property is restored. This also takes at most O(logn) swaps.
* **Heap Sort:** Heaps provide an elegant and efficient in-place sorting algorithm with a reliable O(nlogn) time complexity.
  1. **Build Heap (heapify):** Convert the initial unsorted list into a max-heap. A clever bottom-up approach can achieve this in O(n) time.
  2. **Sort:** Repeatedly perform the following n-1 times:
     + Swap the root element (the current max) with the last element of the current heap.
     + Reduce the considered size of the heap by one (effectively placing the max element at the end of the sorted portion of the list).
     + Run "bubble-down" on the new root to restore the heap property for the smaller heap.