Coding Interview Notes

Griffin Pinney

Two Pointers

The two pointers strategy is a general approach to solve problems in which we deal with **sorted** arrays (or linked lists) and need to find a set of elements that fulfill certain constraints. The set of elements could be a pair, a triplet or even a subarray. We initialise two pointers 'left' and 'right' which are successively moved throughout the array according to rules determined by the given problem. Common initialisations include:

1. Each pointer at either end of the array, moving inwards.
2. Both pointers somewhere in the middle of the array, moving outwards.
3. Both pointers at the beginning of the array, moving forward at different rates.

The idea is that we can leverage the sorted structure of the array to search more efficiently for a pair, triplet, or subarray with the desired properties. For example, when searching an unsorted array for a pair of elements with a desired sum, one must effectively search all possible pairs, which takes $O(N^2)$ time. If the array is sorted, however, the two-pointer approach would be to first consider the left-most and right-most elements. If their sum is smaller than the target, we know that any other pair involving the left-most element would sum to a value even smaller than the target, so we may narrow our search by ignoring the left-most element; and a symmetric argument holds in the case where the sum of the outermost elements is greater than the target (**this logic can be easily formalised by inducting on the size of the array**). The algorithm is demonstrated in the following code:

```
def search(arr, target_sum):
    left, right = 0, len(arr) - 1
    while(left < right):
      current_sum = arr[left] + arr[right]
      if current_sum == target_sum:
        return [left, right]

      if target_sum > current_sum:
        left += 1  # we need a pair with a bigger sum
      else:
        right -= 1  # we need a pair with a smaller sum
    return [-1, -1]
```

A natural variant of the above problem is to find all unique pairs of entries of a sorted array which sum to a given target. In this case, upon finding such a pair, one must be careful to increment the pointers past any potential duplicates. This is demonstrated in the following algorithm:

```
def searchPair(arr, target_sum):
    solutions = []
    left, right = 0, len(arr) - 1
    while(left < right):
      current_sum = arr[left] + arr[right]
      if current_sum == target_sum:  # found a pair
        solutions.append([arr[left], arr[right]])
        left += 1
        right -= 1
        while left < right and arr[left] == arr[left - 1]:
          left += 1  # skip same element to avoid duplicate pairs
        while left < right and arr[right] == arr[right + 1]:
          right -= 1  # skip same element to avoid duplicate pairs
      elif target_sum > current_sum:
        left += 1  # we need a pair with a bigger sum
      else:
        right -= 1  # we need a pair with a smaller sum
    return solutions
```

An extension of the above class of problems is to find all unique n-tuples of entries in a sorted array which sum to a given target for larger values of n. To solve this problem, we take an inductive approach, choosing a placement of the 'left' pointer and then applying the algorithm for the $(n-1)$-case to the remaining subarray (to the right of the 'left' pointer). In this case, one must also be careful to move past duplicates when choosing the placement of the 'left' pointer. In general, such an algorithm works in $O(N^{n-1})$ time. As such, for $n \geq 3$, it is 'within budget' to sort the array first in $O(N \log N)$ time and then apply the described algorithm, the result of which is still an algorithm with time complexity $O(N^{n-1})$. Such an algorithm is shown below in the $n = 3$ case, utilising a variant of the searchPair function which we do not show the details of.

```
def searchTriplets(arr):
    arr.sort()
    triplets = []
    for i in range(len(arr)):
      if i > 0 and arr[i] == arr[i-1]:  # skip same element to
    avoid duplicate triplets
        continue
      searchPair(arr, -arr[i], i+1, triplets)
    return triplets
```

Fast and Slow Pointers

The fast and slow pointers strategy uses two pointers which move through an array (or linked list) at different speeds, most commonly with one pointer moving twice as fast as the other. This is useful for detecting cycles in arrays or linked lists, since the different speeds of the two pointers guarantee that they will eventually meet in the presence of a cycle.

For use in subsequent algorithms, we introduce a simple node class for linked lists:

```
class Node:
 def __init__(self, value, next=None):
   self.val = value
   self.next = next
```

The most basic application of this strategy is for the detection of a cycle in a linked list, as follows:

```
def hasCycle(head):
    fast, slow = head, head
    while fast and fast.next:
        fast = fast.next.next
        slow = slow.next
        if fast == slow:
            return True
    return False
```

In the presence of a cycle, it is natural to want to know its length. In this case, one may call the following function on a node contained in the cycle (identified using the framework of the previous function):

```
def calculate_cycle_length(head):
    current = head
    cycle_length = 0

    while True:
        current = current.next
        cycle_length += 1
        if current == head:
            break
    return cycle_length
```

In the absence of a cycle, the same framework as the hasCycle function provides a convenient algorithm for finding the middle node of a linked list. This can be used as a subroutine in more complicated algorithms, to check whether a linked list is a palindrome, for example.

```
def findMiddle(head):
    fast, slow = head, head
    while fast and fast.next:
        fast = fast.next.next
        slow = slow.next
    return slow
```

(Note that if there is an even number of nodes, the above algorithm returns the one which comes immediately after the middle.)

The fast and slow pointers paradigm may be applied in any situation where cyclic behaviour is to be detected, regardless of whether a linked list is involved. For example, if we define a number to be *happy* if, after repeatedly replacing it with the sum of the squares of its digits, we ultimately get 1, each number will either be happy or end up in a cycle which doesn't contain 1. We may therefore write a function to determine whether a number is happy as follows:

```
def isHappy(num):
    fast, slow = num, num
    while True:
      fast = find_square_sum(find_square_sum(fast))  # move two
    steps
      slow = find_square_sum(slow)  # move one step
      if fast == slow:  # found a cycle
        break
    return slow == 1  # see if the cycle is stuck on the number '1'
```

Sliding Window

The sliding window strategy is useful for solving problems in which one must examine contiguous subarrays of an array (or linked list). One maintains a 'window' of a certain size which slides throughout the array, and instead of recalculating some data (such as the value of a sum) for each new window from scratch, the algorithm updates the value incrementally.

A typical problem is as follows: given an array of positive integers and a positive integer k, find the maximum sum of any contiguous subarray of size k. A sliding window solution is given below:

```
def findMaxSumSubArray(arr, k):
    max_sum, window_sum = 0, 0
    window_start = 0

    for window_end in range(len(arr)):
        window_sum += arr[window_end]
        if window_end >= k-1:
            max_sum = max(max_sum, window_sum)
            window_sum -= arr[window_start]  # subtract the element
 going out
            window_start += 1  # slide the window ahead
    return max_sum
```

Note how we store the indexes of the beginning and end of the window as variables. Iterating over the window end variable and sliding the window start to 'catch up' when necessary is a common feature of sliding window algorithms.

As another example, consider the problem of finding the length of the longest substring of a string with no more than k distinct characters. One keeps a dictionary of character counts within the current window at a cost of $O(k)$ space, and slides the end of the window forward. If at any point the distinct character count in the window exceeds k, one slides the start of the window forward until the window once again contains at most k distinct characters. The code for this algorithm is given below:

```
def findLength(str1, k):
    window_start = 0
    max_length = 0
    char_frequency = {}

    for window_end in range(len(str1)):
        right_char = str1[window_end]
        if right_char not in char_frequency:
            char_frequency[right_char] = 0
        char_frequency[right_char] += 1

        while len(char_frequency) > k:
            left_char = str1[window_start]
            char_frequency[left_char] -= 1
            if char_frequency[left_char] == 0:
                del char_frequency[left_char]
            window_start += 1
        max_length = max(max_length, window_end-window_start + 1)
    return max_length
```

The logic underpinning such an algorithm is sometimes difficult to justify, so we address it in more depth in the case of the above problem. Let $s(n)$ and $e(n)$ denote the indices of the window start and window end respectively at the end of the n^{th} iteration of the outermost loop (noting that $e(n) = n$ by construction). By induction, we can prove that for each n, the substring from $s(n)$ to $e(n)$ is the maximal substring with at most k distinct characters which ends at index $e(n)$, and is therefore detected by the algorithm. Indeed, the $n = 0$ case is clear, and assuming this holds for some n, we note that $s(n)$ forms an increasing sequence, hence $s(n+1) > s(n)$ or $s(n+1) = s(n)$. If $s(n+1) > s(n)$, it must have been the case that the substring from $s(n)$ to $e(n+1)$ contained more than k distinct characters, in which case it is clear from the algorithm that the substring from $s(n+1)$ to $e(n+1)$ is the maximal substring with the desired properties. On the other hand, if $s(n+1) = s(n)$, we assume for contradiction that the maximal substring with at most k distinct characters which ends at index $e(n+1)$ is not the substring from $s(n+1)$ to $e(n+1)$. It is clear from the algorithm that this substring *does* contain at most k distinct characters, so it must be the case that the maximal substring in question starts with some index $i < s(n+1)$. But then, the substring from index i to $e(n)$ has as most k distinct characters and is larger than the substring from $s(n)$ to $e(n)$, contradicting the inductive hypothesis. Thus, we have in all cases that the substring from $s(n+1)$ to $e(n+1)$ is the maximal substring with at most k distinct characters ending at this index, which completes the proof of the validity of the algorithm.

As an example of a similar problem with a slightly different sliding window solution, suppose we wish to write an algorithm to find the longest substring of a string which, after replacing no more than k letters with any letter, can be made to have all the same letters. This is solved as follows:

```
def findLength(str1, k):
    window_start, max_length, max_repeat_letter_count = 0, 0, 0
    frequency_map = {}

    for window_end in range(len(str1)):
        right_char = str1[window_end]
        if right_char not in frequency_map:
            frequency_map[right_char] = 0
        frequency_map[right_char] += 1

        max_repeat_letter_count = max(
            max_repeat_letter_count, frequency_map[right_char])

        if (window_end - window_start + 1 - max_repeat_letter_count
    ) > k:
            left_char = str1[window_start]
            frequency_map[left_char] -= 1
            window_start += 1

        max_length = max(
        max_length, window_end - window_start + 1
    return max_length
```

In this case, the smallest number of letters we would have to replace in

a given substring to make all the letters the same is equal to the number of letters in that substring which are not the same as the substring's most frequent letter. The largest number of occurrences of any single letter in any substring we have considered is therefore a quantity of interest, and we store this in the *max_repeat_letter_count* variable. We can once again prove that this algorithm works by induction, arguing that after the n^{th} iteration of the loop, the algorithm has found the maximal length of a substring ending at (or to the left of) $e(n)$ satisfying the given conditions, and moreover, the window from $s(n)$ to $e(n)$ achieves this length.

As a final demonstration of the sliding window technique, consider the problem of determining whether a given string contains as a substring any permutation of a given pattern. In this case, the sliding window approach is clear: one must maintain a window equal in length to the pattern. The only difficulty lies in keeping track of which of the distinct letters from the pattern have been completely accounted for in this window at any given time. This is elegantly handled with minimal overhead by creating a dictionary with keys given by the distinct characters in the pattern and values originally determined by the occurrences of those characters in the pattern. As one slides the window through the given string, the values of the dictionary are adjusted according to how many of each character appears in the window at any given time. If all occurrences of a given character have been accounted for (as determined by the pattern), that character is condidered 'matched', and we store the number of matches at any given time in a separate variable. If at any point the number of matches equals the number of distinct characters in the pattern (i.e. the length of the dictionary), then we have found a valid permutation.

```
def findPermutation(str1, pattern):
    char_freq = {}
    for char in pattern:
      if char not in char_freq:
        char_freq[char] = 0
      char_freq[char] += 1

    window_start, matched = 0, 0
    for window_end in range(len(str1)):
      right_char = str1[window_end]
      if right_char in char_freq:
        char_freq[right_char] -= 1
        if char_freq[right_char] == 0:
          matched += 1

      if window_end >= len(pattern):
        left_char = str1[window_start]
        if left_char in char_freq:
          if char_freq[left_char] == 0:
            matched -= 1
          char_freq[left_char] += 1
        window_start += 1

      if matched == len(char_freq):
        return True
    return False
```

Merge Intervals

Merge intervals refers to a general class of problems in which one must take unions or intersections of closed intervals of the real line. We model intervals using the following simple class:

```
class Interval:
  def __init__(self, start, end):
    self.start = start
    self.end = end

def print_interval(i):
  print("[" + str(i.start) + ", " + str(i.end) + "]", end='')
```

The classic problem is, given a list of intervals, return a list of disjoint intervals with the same union as the original (i.e. one must *merge* all overlapping intervals). Assuming the intervals are sorted by starting points, it is easy to devise an algorithm which solves this problem in $O(n)$ time. Given that sorting may first be necessary, the best we can do is the following algorithm with time complexity $O(n \log n)$:

```
def merge(intervals):
    if len(intervals) < 2:
      return intervals

    # sort the intervals on the start time
    intervals.sort(key=lambda x: x.start)

    mergedIntervals = []
    start = intervals[0].start
    end = intervals[0].end
    for i in range(1, len(intervals)):
      interval = intervals[i]
      if interval.start <= end:  # overlapping intervals, adjust
    the 'end'
        end = max(interval.end, end)
      else:  # non-overlapping intervals, add the previous interval
     and reset
        mergedIntervals.append(Interval(start, end))
        start = interval.start
        end = interval.end

    # add the last interval
    mergedIntervals.append(Interval(start, end))
    return mergedIntervals
```

Suppose that instead of merging intervals, i.e. taking unions, we wish to take intersections. Clearly, two intervals are disjoint if either of their starts occurs after the other's end. Equivalently, we may detect whether two intervals intersect by checking whether the start of each occurs before the end of the other. In this case, the start of the intersection is then given by the maximum of the two individual start points, and the end of the intersection is given by the minimum of the two individual (as can be verified by thinking about the precise mathematical definition of an interval). We can use these principles to

solve the problem of finding the intersections of two lists of intervals, given that each list is comprised of disjoint intervals sorted by start times, as follows:

```
def intersect(intervals_a, intervals_b):
    result = []
    i, j = 0, 0

    while i < len(intervals_a) and j < len(intervals_b):
      interval_a = intervals_a[i]
      a_start, a_end = interval_a.start, interval_a.end
      interval_b = intervals_b[j]
      b_start, b_end = interval_b.start, interval_b.end

      # check for overlap
      if a_start <= b_end and b_start <= a_end:
        # store the intersection
        result.append(
          Interval(max(a_start, b_start), min(a_end, b_end)))

        # move past the interval which is finishing first
        if a_end <= b_end:
        i += 1
        else:
        j += 1

    return result
```

As a more challenging example, given a list of intervals, consider the problem of finding the maximum number of intervals which intersect (non-trivially) at a single point. If the intervals represent the start and end times of different meetings, then this problem is equivalent to finding the minimum number of rooms required to hold the meetings. A natural approach is to keep a running count of how many rooms are needed at each point in time, and iterate from left to right through the intervals' start and endpoints, incrementing the room count with each interval start and decrementing the room count with each interval end. This would require us to store and sort two copies of the original interval list (one sorted by start points and one sorted by endpoints), thus demanding $O(n)$ space and $O(n \log n)$ time. We present here a different approach which, despite having the same time and space complexity, requires less space overall, and provides an opportunity for us to work with the priority queue data structure. The approach is to iterate through the intervals in order of starting times, at each step storing the current interval in the priority queue, ordered by soonest ending times first. Each time we consider a new interval, all previous intervals which have already ended are removed from the front of the queue, after which the queue stores only those 'meetings' which are currently taking place. The minimum number of meeting rooms required is therefore given by the maximum length of the priority queue at the end of each step of this process.

To code this solution, we begin by importing the *heapq* library, which allows us to model a priority queue with a *min heap* data structure. A min heap stores the smallest element first (at index 0), so we must also furnish the Interval class with a 'less than' operation so that order can be determined. Our solution makes use of the *heappop* and *heappush* functions from the heapq library, each

of which run in $O(\log n)$ time.

```
import heapq

setattr(Interval, "__lt__",
        lambda self, other: self.end < other.end)

def findMinimumMeetingRooms(meetings):
    if not meetings:
        return 0

    # sort the meetings by start times
    meetings.sort(key=lambda x: x.start)

    minRooms = 0
    minHeap = []
    for meeting in meetings:
        # remove all meetings that have ended
        while minHeap and meeting.start >= minHeap[0].end:
            heapq.heappop(minHeap)
        # add the current meeting into the minHeap
        heapq.heappush(minHeap, meeting)
        # all current meetings are in the minHeap, so we need rooms
     for all of them
        minRooms = max(minRooms, len(minHeap))
    return minRooms
```

Cyclic Sort

The cyclic sort pattern provides an efficient approach for handling problems which involve an array of numbers in a given range. Specifically, this technique is applicable to arrays of length n whose entries are distinct numbers from 1 to n, or slight modifications of this. The technique is reminiscent of the decomposition of an arbitrary permutation as a composition of disjoint cycles: given an array of n distinct integers from 1 to n, it may be sorted in $O(n)$ time and $O(1)$ space by repeatedly sending the first element to its correct position (by swapping it with whatever was already in that position) until the first element itself is correct; the same process is then carried out iteratively at each position in the array.

```
def sort(nums):
    i = 0
    while i < len(nums):
      j = nums[i] - 1  # Calculate the index where the current
    element should be placed.
      if nums[i] != nums[j]:  # Check if the current element is not
     in its correct position.
        nums[i], nums[j] = nums[j], nums[i]  # Swap the current
    element with the one at its correct position.
      else:
        i += 1  # If the current element is already in its correct
    position, move to the next element.
    return nums
```

Cyclic sort problems become more complicated when we do not just have n distinct entries in the range from 1 to n, but a common strategy for such problems is to carry out the cyclic sort as usual, ignoring 'problematic' numbers (those which are out of range or are duplicates of in-range numbers) by skipping past them when they appear. The outcome will be an array in which one of each entry in the appropriate range is correctly placed, and the other entries are placed arbitrarily in the leftover positions. This final configuration can be used to detect which number(s) were missing or which number(s) appear more than once, all in $O(n)$ time. For example, given an unsorted array containing n numbers in the range from 1 to n (some of which may be appear twice, and therefore some of which may be missing), the following algorithm returns a list of all missing numbers.

```
def findNumbers(nums):
    i = 0
    while i < len(nums):
      j = nums[i] - 1
      if nums[i] != nums[j]:
        nums[i], nums[j] = nums[j], nums[i]
      else:
        i += 1
    missingNumbers = []
    for i in range(len(nums)):
      if nums[i] != i + 1:
        missingNumbers.append(i + 1)
    return missingNumbers
```

The majority of cyclic sort problems will have the same framework: one iterates over i in the range from 0 up to the length of the array, each time computing the index j where the current number should be placed. If the number stored at index j is not equal to the current number, then a swap occurs; otherwise we increment i and repeat the process. As another example, given an unsorted array containing n numbers in the range from 1 to n (some of which may appear twice, and therefore some of which may be missing), the following algorithm returns a list of all duplicate numbers. We note that the solution is essentially identical to that of the previous problem.

```
def findNumbers(nums):
    i = 0
    while i < len(nums):
      j = nums[i] - 1
      if nums[i] != nums[j]:
        nums[i], nums[j] = nums[j], nums[i]
      else:
        i += 1
    duplicateNumbers = []
    for i in range(len(nums)):
      if nums[i] != i + 1:
        duplicateNumbers.append(nums[i])
    return missingNumbers
```

As a more complicated example, consider the problem of finding the smallest missing positive number in an unsorted array (with no conditions on the range of the elements). As hinted previously, if we stick to the cyclic sort strategy by simply ignoring problematic numbers, then we obtain a simple solution which runs in $O(n)$ time and $O(1)$ space:

```
def findNumber(nums):
    i, n = 0, len(nums)
    while i < n:
        j = nums[i] - 1
        if 0 <= j and j < n and nums[i] != nums[j]:
            nums[i], nums[j] = nums[j], nums[i]
        else:
            i += 1
    for i in range(n):
        if nums[i] != i + 1:
            return i + 1
    return n + 1
```

In-place Reversal of a Linked List

The purpose of this pattern is self-explanatory from its name, and is useful as a sub-step in more complicated algorithms. The approach is to keep track of two nodes at all times: the previous node and the current node (initialised to *None* and *head* respectively). At each step one updates the current node's head so that it equal to the previous node, and then shifts the previous and current nodes forward (though one must store the next node in a separate variable before any updates take place). This process is repeated as long as the current node is not *None*, and if one wishes to return the new head of the linked list after reversal, one should return the previous node after the loop is complete.

As in previous sections, we work with the following simple Node class:

```
class Node:
    def __init__(self, value, next=None):
        self.val = value
        self.next = next
```

The algorithm described above is then as follows:

```
def reverse(head):
    previous, current = None, head
    while current is not None:
        nxt = current.next  # temporarily store the next node
        current.next = previous  # reverse the current node
        # before we move to the next node, point previous to the
    current node
        previous = current
        current = nxt  # move on the next node
    return previous
```

A variant of this problem is to reverse the sub-list of a linked list from nodes p to q, which is accomplished as folows:

```
def reverse(head, p, q):
    curr, prev = head, None
    curr_pos = 1
    while curr_pos < p:
      prev = curr
      curr = curr.next
      curr_pos += 1
    node_p_minus_one = prev
    node_p = curr
    prev = None
    while curr_pos < q+1:
      nxt = curr.next
      curr.next = prev
      prev = curr
      curr = nxt
      curr_pos += 1
    if node_p_minus_one:
      node_p_minus_one.next = prev
    else:
      head = prev
    node_p.next = curr
    return head
```

As can be seen from the algorithm, one first locates the p^{th} node, storing it and its predecessor for later use, and then applies the preceding algorithm up to the q^{th} node. One must then connect the ends of the newly reversed sub-list to the appropriate nodes of the original list, using the previously stored variables to do so, and taking care to handle the case $p = 1$, in which case the first node has no predecessor.

A related but slightly more challenging problem is to reverse every sub-list of length k starting from the head of a linked list. This follows a very similar pattern to the previous solution, with the added complexity that when treating each sub-list of length k, one must store the last node of the previous sub-list and the first node of the current sub-list to ensure the required connections can be made at the end of each reversal. Our solution takes the form of a while loop which runs as long as the current node is not *None*:

```
def reverse(head, k):
    current, previous = head, None
        while current:
          previous_section_last_node = previous
          this_section_first_node = current
          i = 0

          while current and i < k: # reverse the next k nodes
            nxt = current.next
            current.next = previous
            previous = current
            current = nxt
            i += 1

          # connect with the previous part
          if previous_section_last_node:
            previous_section_last_node.next = previous
          else:
            head = previous # ensures we end with the right head;
    only executes on the first iteration of the outer loop

          # connect with the next part
          this_section_first_node.next = current

          # set up the next iteration
          previous = this_section_first_node

        return head
```

This algorithm, as with all others in this section, has time complexity $O(n)$ and space complexity $O(1)$.

Stacks

A stack is a linear data structure that operates on the principle of Last-In, First-Out (LIFO), meaning the most recently added element is the first to be removed. There are four key operations that you can perform on a stack, all in $O(1)$ time:

- Push: This is how we add an element to the stack. The element is always added to the top.
- Pop: This is the removal of an element from the stack. The element is always removed from the top.
- Top (or peek): This operation allows us to see the element on the top of the stack without removing it.
- IsEmpty: This operation checks if the stack is empty.

Stacks are usually implemented in Python using lists, as demonstrated by the following class:

```
class Stack:
    def __init__(self):
        self.stack = []

    def isEmpty(self):
        return self.stack == []

    def push(self, data):
        # Add the given data to the top of the stack (end of the
    list).
        self.stack.append(data)

    def pop(self):
        if self.isEmpty():
            return 'Stack is empty'
        # Remove and return the top element from the stack (the
    last item in the list).
        return self.stack.pop()

    def peek(self):
        if self.isEmpty():
            return 'Stack is empty'
        # Return the top element from the stack (without removing
    it).
        return self.stack[-1]
```

This class is so simple that in practice, when one wishes to use a stack to solve a problem, one uses lists directly without going to the trouble of defining such a class. As an example, given a string of parentheses containing (,), [,], {, and } characters, consider the problem of determining whether the parentheses are balanced (i.e. every opening parenthesis has a corresponding closing parenthesis of the same type in the right order). To solve this problem, we create a stack and iterate through the characters of the string. Each time we encounter an

opening parenthesis, we push it onto the stack, and each time we encounter a closing parenthesis, we hope to pop a matching opening parenthesis from the top of the stack. If at any point this isn't possible, the parentheses could not have been balanced, and we return *False*. If we reach the end of this process and the stack is empty, then the parentheses must have been balanced, and we return *True*. This algorithm is shown below:

```
def isValid(s):
    stack = []
    for c in s:
        if c in ['(', '{', '[']:
            stack.append(c)
        else:
            # If stack is empty and we have a closing parenthesis, the string is not balanced
            if not stack:
                return False
            top = stack.pop()
            if c == ')' and top != '(':
                return False
            if c == '}' and top != '{':
                return False
            if c == ']' and top != '[':
                return False
    # If the stack is empty, all opening parentheses had a corresponding closing match
    return not stack
```

Another simple example is to write an algorithm which, given a positive integer, returns a string containing that integer's binary representation. Since the natural while loop for obtaining the integer's binary digits stores them in the opposite order in which they should be printed, it is natural to use a stack for this problem, as its LIFO nature guarantees that if we pop these digits from the stack one by one and add them to a new string, we will have reversed the order and obtained the correct binary representation:

```
def decimalToBinary(num):
    stack = []
    while num > 0:
        stack.append(num % 2)  # Push the remainder of num divided by 2 onto the stack.
        num //= 2  # Update num by integer division (floor division) by 2.
    result = ''
    while stack:
        result += str(stack.pop())
    return result
```

As a more challenging example, given an array of positive numbers, consider the problem of constructing an array of the same size containing for each position the next greater element to the right of that position in the original array (or -1 for those positions which have no next greater element). To solve this problem efficiently, one creates an array of the appropriate size with each element initialised to -1, and iterates from right to left through the original array,

maintaining a stack which at each step will contain all potential next greater elements for the position under consideration. For each position, we pop elements from the stack until we either find something greater than the current element (in which case we update our solution array) or the stack becomes empty. After the completion of this process, we will have the correct value at the current position in the solution array, and we push the current element of the original array onto the stack before continuing. The code for this algorithm is given below:

```
def nextLargerElement(arr):
    res = [-1] * len(arr)
    s = []
    for i in range(len(arr) - 1, -1, -1):
        while s and s[-1] <= arr[i]:
            s.pop()
        if s:
            res[i] = s[-1]  # If the stack is not empty, set the
    result for the current element to the top element of the stack
        s.append(arr[i])  # Push the current element onto the stack
    return res
```

Since each element of the original array gets added to and removed from the stack at most once, the above algorithm runs in $O(n)$ time (and obviously takes $O(n)$ space). That the algorithm works is intuitively clear but not obviously true; to show it, one proves by induction that after the popping process on each iteration of the outer loop, either the current element has no next greater element and the stack is empty, or the current element's next greater element is on the top of the stack, and each other element of the stack is the next greater element of its successor (the next highest element on the stack).

As a final example, consider the task of sorting a stack using only stack operations (push and pop) and an additional temporary stack. To solve this problem, one must appreciate that the imposed restrictions are sufficiently strict that a time-efficient algorithm is not possible, and one should not hope for an algorithm with a time complexity any better than $O(n^2)$. Having set our expectations, the main idea of the algorithm is easily found: one moves elements from the original stack to the temporary one, maintaining order on the temporary stack by using the original as extra storage space for any elements which would violate the order:

```
def sortStack(stack):
    tempStack = []
    while stack:
        next_element = stack.pop()
        while tempStack and tempStack[-1] > next_element:
            stack.append(tempStack.pop())
        tempStack.append(next_element)
    return tempStack
```

Monotonic Stacks

A monotonic stack describes a stack in which we enforce all elements to maintain a specific order. Note that **this is not a new data structure**; to maintain order, we do not devise a new pushing operation to insert elements mid-way through the stack (if we did, the structure would no longer deserve to be called a stack). Instead, if we wish to push a new element to the stack, we simply pop elements from the top until pushing would no longer violate the order. Thus, monotonic stacks are more of a problem-solving strategy than a new data structure, and we have already used a similar strategy in the previous section for finding the next greatest element for each element in an array. More generally, the monotonic stack strategy is useful for problems requiring any kind of sequential comparison of the elements in an array, with the next greatest element (NGE) and next smaller element (NSE) problems being particularly notable examples. Indeed as our first example, given an array of distinct integers, we once again consider the problem of constructing an array of the same size containing for each position the next greater element to the right of that position in the original array (or -1 for those positions which have no next greater element). Although we solved an almost identical problem in the previous section with similar techniques, we take a slightly different approach here for the sake of furthering our exposure to the monotonic stack strategy. This time, the idea is to maintain a monotonically decreasing stack to which elements are added from left to right from the original array. It is easy to see by induction that the elements remaining on the stack at each step are those for which a next greater element hasn't yet been found, and when an element gets popped from the stack, it has been caused to do so by its own next greater element. Thus, when an element is popped from the stack, we pair it with its next greatest element in a dictionary, so the only elements not accounted for in the keys of the dictionary at the end of this process are those which have no next greater element. These are easily handled using the 'get' method for dictionaries, as shown in the following code:

```
def nextGreaterElement(nums):
    stack, solutionDict = [], {}
    for num in nums:
        while stack and stack[-1] < num:
            solutionDict[stack.pop()] = num
        stack.append(num)
    return [solutionDict.get(num, -1) for num in nums]
```

We note that this algorithm has time complexity $O(n)$ (as is common with monotonic stack problems), the reason being that each node is pushed to and popped from the stack at most once each. Clearly, the algorithm also requires $O(n)$ space.

As a second example, given the head of a singly linked list, consider the problem of modifying the list so that any node which has a node with a greater value to its right gets removed (as usual, the function we write to solve this problem should return the head of the modified list). There are two key obser-

vations here, the first being that we can have a stack store not just the values of the nodes, but the nodes themselves; the second is that the process of creating a monotonically decreasing stack from the nodes of the linked list will, as described previously, terminate with a stack containing all those nodes satisfying the required condition (as can be proven by induction). With these two observations in mind, a solution presents itself without too much difficulty:

```
def removeNodes(head):
    stack = []
    current = head
    while current:
        while stack and stack[-1].val < current.val:
            stack.pop()
        if stack:
            stack[-1].next = current
        stack.append(current)
        current = current.next
    return stack[0]
```

We notice a key pattern in monotonic stack problems emerging: we iterate over the elements of whatever data structure is under consideration, and on each iteration, we pop elements from the stack until the desired order is preserved, before finally adding the current element to the stack.

As a final, more challenging example, as well as an example of a monotonically increasing stack being useful, we consider the following problem: given a non-negative integer represented as a string, and given an integer k, find the minimum number which can be formed from the starting integer by deleting k digits. Our strategy is to place the digits from left to right in a monotonically increasing stack, as this will remove all instances of a digit being followed by a smaller digit (in which case removing that digit would result in a smaller number). The full details are as follows:

```
def removeKdigits(num, k):
    stack = []
    for digit in num:
        while k > 0 and stack and stack[-1] > digit:
            stack.pop()
            k -= 1
        stack.append(digit)

    # Truncate the remaining k digits (if there are any)
    stack = stack[:-k] if k > 0 else stack

    # Remove any leading zeros
    return "".join(stack).lstrip("0") or "0"
```

We note that the *lstrip* method used above is for removing leading characters from a string.

Hash Maps

A hash map is a data structure based on the concept of a *hash function*, which allows us to store a collection of key-value pairs with constant-time insertion, access, and deletion operations. There are some theoretical considerations behind the operation of such a data structure, but it suffices to know that hash maps are implemented in Python via the familiar *dict* type. Though there are many use cases for hash maps, they are particularly useful in problems in which a key goal is to count the total number of occurrences of certain objects. A simple example is the problem of determining the largest-length palindrome that can be constructed using some of the characters of a given string (in some order). Since we are free to select and reorder whichever characters we want, the only important information is the frequency of each character, and this problem has a simple solution as follows:

```
from collections import defaultdict

def longestPalindrome(s):
    # Get character frequencies
    freq_map = defaultdict(int)
    for ch in s:
        freq_map[ch] += 1

    length = 0
    oddFound = False

    # Calculate the palindrome length
    for freq in freq_map.values():
        if freq % 2 == 0:
            length += freq
        else:
            length += freq - 1
            oddFound = True

    # Add the central character if any odd frequency was found
    if oddFound:
        length += 1

    return length
```

We note that the particular type of hash map utilised in the above example was a *defaultdict* from the *collections* module, which conveniently gives all keys a default value of 0. The same functionality could have been achieved using standard Python dictionaries by writing either of the following:

```
freq_map = {}
for ch in s:
    if not ch in freq_map:
        freq_map[ch] = 0
    freq_map[ch] += 1
```

```
freq_map = {}
for ch in s:
    freq_map[ch] = freq_map.get(ch, 0) + 1
```

Tree Breadth First Search

Tree breadth-first search (BFS) traversal is a technique for exploring the nodes of a binary tree level by level, starting from the root; for this reason, it is ideal for solving problems that require a level-wise approach. The main idea is to process the nodes using a queue. We begin by inserting the root node into the queue, then repeatedly process nodes from the front of the queue. After each node is processed, we add its children to the back of the queue, ensuring that all nodes at the current level are processed before processing begins for the next level.

We model tree nodes using the following simple class, to be used throughout this section:

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
```

As a preliminary example, given the root of a binary tree, consider the problem of computing the sum of the values of all nodes in the tree:

```
from queue import Queue

def sumOfNodes(root):
    if root is None:
        return 0

    # Initialize a queue to store nodes for BFS traversal
    queue = Queue()
    queue.put(root)
    sum = 0

    # Perform BFS traversal
    while not queue.empty():
        currentNode = queue.get()
        sum += currentNode.val
        if currentNode.left:
            queue.put(currentNode.left)
        if currentNode.right:
            queue.put(currentNode.right)

    return sum
```

We note that in the above solution, we utilised the *Queue* data structure from the *queue* module. It is also possible to use a double-ended queue, i.e. a *deque*, which we can import from the *collections* module, as long as we make sure to push to and pop from the correct ends of the deque. Since the syntax for deque operations is more consistent with that for lists, and therefore more intuitive, we will use this data structure for the remainder of our BFS demonstrations.

As a second example, given a binary tree, consider the problem of populating a list to represent its level-by-level traversal. The list should store the values of all nodes at each level from left to right in separate sub-lists. To acheive

this, we require a way to delineate successive levels of the tree. We do so by recording the length of the queue at the beginning of each iteration of the main loop, which can inductively be seen to represent the number of nodes in the corresponding level. We then process the nodes in that level within a further nested loop:

```
from collections import deque

def traverse(root):
    result = []
    if root is None:
      return result

    queue = deque()
    queue.append(root)
    while queue:
      levelSize = len(queue)
      currentLevel = []
      for _ in range(levelSize):
        currentNode = queue.popleft()
        currentLevel.append(currentNode.val)
        if currentNode.left:
          queue.append(currentNode.left)
        if currentNode.right:
          queue.append(currentNode.right)
      result.append(currentLevel)

    return result
```

The above method of keeping track of the number of nodes in each level is very common in BFS problems when each level must be treated differently. As a final example, we consider the problem of finding the minimum depth of a binary tree, i.e. the number of nodes along a path of minimal length from root to leaf. Since this problem is essentially asking for the highest level of a leaf node, it makes sense to distinguish between levels as in the previous example:

```
from collections import deque

def find_maximum_depth(root):
    if root is None:
      return 0

    queue = deque()
    queue.append(root)
    maximumTreeDepth = 0
    while queue:
      maximumTreeDepth += 1
      levelSize = len(queue)
      for _ in range(levelSize):
        currentNode = queue.popleft()
        if currentNode.left:
          queue.append(currentNode.left)
        if currentNode.right:
          queue.append(currentNode.right)

    return maximumTreeDepth
```

Tree Depth First Search

Tree depth-first search (DFS) traversal is a technique for exploring the nodes of a binary tree. Using a recursive approach, DFS explores each branch of the tree as far as possible before backtracking, and is therefore useful in scenarios in which one must find paths from root to leaf satisfying certain properties. Generally, one must treat the empty tree and leaf nodes as a base case, and in other cases the function is recursively called on the root's left and right children in such a way that the desired goal will be accomplished, assuming inductively that the function carries out the required task on the sub-trees having these nodes as roots. A simple example is the problem of determining whether a given tree (whose nodes have integer values) has a path from root to leaf with a given sum. If the tree is empty, the answer must be no, and if the root node has no children, the answer is clear; in all other cases, we recursively check whether the left and right sub-trees have paths from root to leaf which sum to the originally desired value minus the value of the original root. The code is as follows:

```
def hasPath(root, sum):
    if not root:
      return False

    if not root.left and not root.right:
      return root.val == sum

    return hasPath(root.left, sum - root.val) or hasPath(root.right
    , sum - root.val)
```

When solving problems involving DFS, it is often the case that one must define a recursive helper function to carry out the desired task, in which case one's main solution function is a simple call to the recursive helper. This is especially the case when the problem requires certain states (such as incomplete sequences of node values) to be kept track of, and a collection of the successful complete states to saved. For example, consider the variant of the previous problem in which we not only wish to determine whether a given sum can be made, but we also wish to return a list of all those paths from root to leaf which achieve the desired sum. We initialise an empty list of all such paths, and call a recursive helper function which will add examples to this list once they are found. In more detail, the helper function is given a starting node, a required sum, a current path (to be interpreted as the sequence of nodes visited **before** reaching the current node), and a running list of found examples, and will add to this list **copies** of all appropriate extensions of the current path to leaf nodes via the starting node. Importantly, at the end of the function call, the current path argument should be in the same state as when it was originally passed into the function (though it may be temporarily modified during function's execution). If we devise a helper function for which this behaviour can be proven by induction (finding appropriate extensions and leaving the current path unchanged), then having our main function call the helper function with the appropriate initialisations will solve the problem. Such a solution is given

below:

```
def findPaths(root, required_sum):
    allPaths, currentPath = [], []
    recursiveFindPaths(root, required_sum, currentPath, allPaths)
    return allPaths

def recursiveFindPaths(currentNode, required_sum, currentPath,
    allPaths):
    if not currentNode:
      return

    currentPath.append(currentNode.val)

    if not currentNode.left and not currentNode.right:
      if currentNode.val == required_sum:
        allPaths.append(currentPath.copy()) # Must create a copy!
    else:
      recursiveFindPaths(currentNode.left, required_sum -
    currentNode.val, currentPath, allPaths)
      recursiveFindPaths(currentNode.right, required_sum -
    currentNode.val, currentPath, allPaths)

    del currentPath[-1]
    return
```

This pattern of using a helper function to keep track of partially created top-down examples – and appropriately adding data from the current node before recursively calling the helper function – is common in DFS problems. As another example, consider a binary tree with all nodes having digit values (0-9). Each path from root to leaf represents a number, and the problem is to write a function to compute the sum of these numbers. We write a recursive helper function which, given a number representing a partial string of digits up to some starting node, will compute the sum of all extensions of this string from the starting node to leaves. Calling this helper function on the root of the tree with a starting number of 0 (because no previous nodes have been visited) will give the desired sum:

```
def findSumOfPathNumbers(root):
    return recursiveFindSum(root, 0)

  def recursiveFindSum(currentNode, startVal):
    if not currentNode:
      return 0

    new_val = 10 * startVal + currentNode.val

    if not currentNode.left and not currentNode.right:
      return new_val

    return recursiveFindSum(currentNode.left, new_val) +
    recursiveFindSum(currentNode.right, new_val)
```

As a similar but slightly more complicated example, consider the problem of counting the number of paths in a tree with a given sum, where paths can start and end at any node (but must travel in the direction from parent to child).

Our approach is to write a recursive function which, given a starting node, a list of previously visited nodes, and a target sum, will count all paths with the appropriate sum which end at or below the starting node (if such paths extend above the starting node, they can only do so using the list of previously visited nodes). For inductive purposes, this function should also leave the given list of previously visited nodes unmodified (though it may be temporarily modified during function's execution). Calling this function with the root as the starting node, the empty list as the list of previously visited nodes, and the given sum as the target sum, will solve the problem:

```
def countPaths(root, S):
    return recursiveCountPaths(root, [], S)

def recursiveCountPaths(currentNode, currentPath, S):
    if not currentNode:
      return 0

    currentPath.append(currentNode.val)
    total, pathSum = 0, 0
    for i in range(len(currentPath) - 1, -1, -1):
      pathSum += currentPath[i]
      if pathSum == S:
        total += 1

    total += recursiveCountPaths(currentNode.left, currentPath, S)
    total += recursiveCountPaths(currentNode.right, currentPath, S)

    del currentPath[-1]
    return total
```

As a slightly different example, we consider the problem of determining the diameter of a binary tree, i.e. the length of the longest path between two leaf nodes. Such a path must change direction somewhere, so we must effectively compute the maximum across all nodes of the length of the longest path from leaf to leaf which turns around at that node. The length of this longest path is always equal to the depth of the sub-tree to the left plus the depth of the sub-tree to the right plus one, so the problem essentially reduces to writing a recursive function for computing tree depths, whilst keeping a running maximum of the longest path from leaf to leaf that has been found:

```
def findDiameter(root):
    treeDiameter = [0]
    depthFinder(root, treeDiameter)
    return treeDiameter[0]

def depthFinder(currentNode, treeDiameter):
    if not currentNode:
      return 0

    depthLeft = depthFinder(currentNode.left, treeDiameter)
    depthRight = depthFinder(currentNode.right, treeDiameter)
    treeDiameter.append(max(treeDiameter.pop(), 1 + depthLeft +
    depthRight))

    return 1 + max(depthLeft, depthRight)
```

As seen above, one way to keep track of the running maximum is to utilise the mutability of Python lists and store the running maximum as the single element of a list which is passed as an additional argument to all recursive functions. A natural alternative is to have the diameter and depth-finding functions be methods in a class which has the tree diameter as an attribute, in which case the running maximum can be updated without the need for an additional argument to be fed to all recursive functions.

As a final example, we consider instead the problem of finding the maximum sum of any path in a binary tree. Again, since we must effectively compute the maximum across all nodes of the largest sum achieved by paths which change direction at that node, and since this largest sum for a given node is found by adding the value of that node to the values of the largest possible sums obtained from top-down paths starting from the node's left and right children, the problem essentially reduces to writing a recursive function for computing the maximum sum of any top-down path from a given node. There is one small subtlety, namely, there is no need to add negative path sums from left and right sub-trees, as these cannot possibly contribute to the maximum sum achieved by paths which turn around at the given node. To account for this, we ensure that the recursive function for computing the maximum top-down path sums from each node always returns a non-negative value.

Since this problem once again requires the keeping of a running maximum, we can again either take the approach of exploiting the mutability of Python lists, or introduce a class which has the running maximum as an attribute. We once again demonstrate the former approach:

```
def findMaximumPathSum(root):
    globalMaximumSum = [-float('inf')]
    recursiveFindMaxSum(root, globalMaximumSum)
    return globalMaximumSum[0]

def recursiveFindMaxSum(currentNode, globalMaximumSum):
    if not currentNode:
      return 0

    leftSum  = recursiveFindMaxSum(currentNode.left,
    globalMaximumSum)
    rightSum = recursiveFindMaxSum(currentNode.right,
    globalMaximumSum)
    globalMaximumSum.append(max(globalMaximumSum.pop(),
                            currentNode.val + leftSum + rightSum))

    return max(currentNode.val + max(leftSum, rightSum), 0)
```

Graphs

A graph is a data structure that consists of a set of vertices (nodes) and edges connecting those vertices. For the purposes of coding problems, graphs are often represented using **adjacency lists**: a list, for each vertex, of all those other vertices it is connected to. This information is often collected in a dictionary whose keys are the vertices are the graph, and whose values are the associated adjacency lists. As an abstract datatype, the following operations can be performed on graphs:

- Adding a new vertex
- Removing a vertex
- Adding an edge between two vertices
- Removing an edge between two vertices
- Getting a list of all the vertices
- Checking if two vertices are adjacent
- Getting a count of the total vertices in the graph
- Getting a count of the total edges in the graph
- Getting a list of the edges
- Getting the neighbors of a given vertex

It is easy to construct a class with a dictionary attribute for the adjacency list which can perform each of the above operations. However, in practice, for coding problems, graphs are simply represented as a list of lists of two integers, each sub-list specifying a pair of vertices between which there is an edge.

Much like trees, there are two main techniques for traversing the nodes of a graph: breadth-first search (BFS) and depth-first search (DFS). BFS can be implemented using a queue, and explores all nodes level by level, moving outwards to visit all the nodes at the same distance from the starting node before moving on to nodes at the next distance; DFS can be implemented using recursion or a stack, and explores all nodes by systematically visiting as far as possible along each branch before backtracking. The main difference in the implementation of these traversal methods for graphs compared to trees is that since graphs may have loops, one must keep track of which vertices have been visited, to avoid visiting the same vertex multiple times. This is usually accomplished by maintaining an array of Booleans of the same size as the number of vertices in the graph, representing whether each vertex has been visited. Since both traversal methods visit each vertex and each edge a bounded number of times (and constant-time operations are performed there), both techniques have a time complexity of $O(V + E)$, where V and E denote the number of vertices

and edges in the graph respectively. The space complexity is dominated by the array of visited vertices, and both techniques therefore require $O(V)$ space.

As a first example of a problem involving graphs, given an undirected graph represented as a list of lists of two integers, and given start and end nodes, write a function which returns *True* if a path exists from the start node to the end node, and returns *False* otherwise. After converting the list of lists into an adjacency list (in the form of a defaultdict), we solve this problem by calling a DFS helper function:

```
from collections import defaultdict

def validPath(n, edges, start, end):
    graph = defaultdict(list)
    for u, v in edges:
        graph[u].append(v)
        graph[v].append(u)

    visited = [False] * n
    return recursiveValidPath(graph, start, end, visited)

def recursiveValidPath(graph, start, end, visited):
    if start == end:
        return True

    visited[start] = True

    for v in graph[start]:
        if not visited[v] and \
        recursiveValidPath(graph, v, end, visited):
            return True

    return False
```

As a second and final example, consider n cities, some of which are connected. If a group of cities is connected directly or indirectly, they form a *province*. Given an $n \times n$ matrix whose entry in position (i, j) is 1 if cities i and j are connected and is 0 otherwise, we write a function to return the number of provinces. The idea is to use a DFS traversal to mark all those cities that can be reached from a given city. Each time we have to do this for a city that hasn't yet been visited, we have found a new province:

```
def findProvinces(isConnected):
    provinces = 0
    visited = [False] * len(isConnected)
    for city in range(len(isConnected)):
        if not visited[city]:
            provinces += 1
            recursiveFindProvinces(city, isConnected, visited)
    return provinces

def recursiveFindProvinces(city, isConnected, visited):
    visited[city] = True
    for i in range(len(isConnected)):
        if isConnected[city][i] and not visited[i]:
            recursiveFindProvinces(i, isConnected, visited)
```

Matrix Traversal

This pattern regards the efficient traversal of a matrix using either DFS or BFS, for the purposes of achieving a certain goal such as detecting isolated clusters of ones among zeros. Indeed, as our first example, we consider a matrix containing only ones (land) and zeros (water), and we wish to write a function to count the number of islands. The approach is very similar to that of the counting provinces problem from the previous section: each time we encounter a new piece of land, we increment a running total of the number of islands found, and we call a recursive function to turn the entire landmass into water so that it doesn't get double counted.

```
def countIslands(matrix):
    totalIslands = 0
    for i in range(len(matrix)):
      for j in range(len(matrix[0])):
        if matrix[i][j] == 1:
          totalIslands += 1
          recursiveCountIslands(matrix, i, j)
    return totalIslands

def recursiveCountIslands(matrix, i, j):
    if i < 0 or i >= len(matrix) or j < 0 or j >= len(matrix[0]):
      return

    if matrix[i][j] == 0:
      return

    matrix[i][j] = 0

    recursiveCountIslands(matrix, i+1, j)
    recursiveCountIslands(matrix, i-1, j)
    recursiveCountIslands(matrix, i, j+1)
    recursiveCountIslands(matrix, i, j-1)
```

The above algorithm has both time and space complexity $O(MN)$, where M and N denote the number of rows and columns in the matrix respectively. This is because we are essentially performing DFS on the graph with MN vertices and $O(MN)$ edges created by connecting adjacent entries of the matrix (recalling that DFS has time and space complexity $O(V+E)$ and $O(V)$ respectively).

As a more complicated example, suppose we are in the same situation as above but with only one island, and consider the problem of writing a function to compute the perimeter of the island. We take a similar DFS approach when we encounter land, but in this case our recursive function should return 1 when it encounters either water or the edge of the matrix, as it is in these cases that the landmass ends, contributing to the perimeter of the island. One caveat is that our recursive function in this case should no longer turn the landmass into water to avoid revisits, as this could lead to an overcount of the total perimeter of the island. Instead, we maintain an auxiliary matrix to keep track of which cells have been visited, which we pass in as an additional argument to our function. The algorithm is as follows:

```
def findIslandPerimeter(matrix):
    visited = [[False for _ in range(len(matrix[0]))] for _ in
    range(len(matrix))]
    for i in range(len(matrix)):
      for j in range(len(matrix[0])):
        if matrix[i][j] == 1:
          return recursiveFindPerimeter(matrix, i, j, visited)
    return 0

def recursiveFindPerimeter(matrix, i, j, visited):
    if i < 0 or i >= len(matrix) or j < 0 or j >= len(matrix[0]):
      return 1

    if matrix[i][j] == 0:
      return 1

    if visited[i][j]:
      return 0

    visited[i][j] = True
    perimeter = 0

    perimeter += recursiveFindPerimeter(matrix, i+1, j, visited)
    perimeter += recursiveFindPerimeter(matrix, i-1, j, visited)
    perimeter += recursiveFindPerimeter(matrix, i, j+1, visited)
    perimeter += recursiveFindPerimeter(matrix, i, j-1, visited)

    return perimeter
```

As a final example, given a matrix of characters, we consider the problem of writing a function to determine whether there exists any cycle consisting of the same character in the matrix, where a cycle is any path through adjacent cells starting and ending at the same cell, but otherwise visiting no cell more than once. We take a similar DFS approach as in the previous two problems, and we once again find it convenient to maintain an auxiliary matrix to keep track of which cells have been visited. For each location in the original matrix, if it has not already been visited, we call a recursive function which returns *True* if that cell is part of a contiguous block of entries with the same character in the original matrix which has a cycle somewhere, and returns *False* otherwise (the starting cell need not be part of the cycle; picture a ring with an extra cell jutting out somewhere). We detect a cycle based on whether we are visiting a cell with the appropriate character which has already been visited (as recorded in our auxiliary matrix), but for this to work, we must ensure that our DFS never doubles back. To do so, we have our recursive function take an additional argument describing the direction in which it has most recently moved (left, right, up, or down), and when we recursively call the function, we do not do so in a direction which would cause the search to double back. When the function is initially called, it has no most recent direction, so we give this parameter a default value of *None* and do not specify it in the original function call. Note that if at any point we leave the grid or encounter a character different from the one currently under consideration, we immediately return *False* as there is no further searching to be done from that point. The full algorithm is as follows:

```
def hasCycle(matrix):
    visited = [[False for _ in range(len(matrix[0]))] for _ in
    range(len(matrix))]
    for i in range(len(matrix)):
      for j in range(len(matrix[0])):
        if not visited[i][j]:
            if recHasCycle(matrix, i, j, matrix[i][j], visited):
                return True
    return False

  def recursiveHasCycle(matrix, i, j, ch, visited, direction=None):
    if i < 0 or i >= len(matrix) or j < 0 or j >= len(matrix[0]):
      return False

    if matrix[i][j] != ch:
      return False

    if visited[i][j]:
      return True

    visited[i][j] = True

    if direction != 'U' \
    and self.recursiveHasCycle(matrix, i+1, j, ch, visited, 'D'):
      return True
    if direction != 'D' \
    and self.recursiveHasCycle(matrix, i-1, j, ch, visited, 'U'):
      return True
    if direction != 'L' \
    and self.recursiveHasCycle(matrix, i, j+1, ch, visited, 'R'):
      return True
    if direction != 'R' \
    and self.recursiveHasCycle(matrix, i, j-1, ch, visited, 'L'):
      return True

    return False
```

Miscellaneous Tricks

Multiple assignment and swapping

One can assign values to multiple variables in a single statement as follows:

```
# Multiple assignment
a, b = 1, 2
```

This provides shortcuts in many contexts, such as initialising the pointers in a two-pointers algorithm. One can use a similar trick to easily swap variables, as follows:

```
# Swapping values
a, b = b, a
print(a)  # Output: 2
print(b)  # Output: 1
```

Checking character types

One can check whether a given character (or the characters in a string) are alphabetic or alphanumeric using the 'isalpha' and 'isalnum' methods respectively. For example:

```
char1 = 'a'
char2 = '1'
char3 = '@'

print(char1.isalpha())  # Output: True
print(char2.isalpha())  # Output: False
print(char3.isalpha())  # Output: False

char1 = 'a'
char2 = '1'
char3 = '@'

print(char1.isalnum())  # Output: True
print(char2.isalnum())  # Output: True
print(char3.isalnum())  # Output: False
```

Splitting a string into a list of substrings

One can use the 'split' method on a string to divide it into a list of substrings separated by a given delimiter. For example:

```
text = "apple:orange:banana"
result = text.split(':')
print(result)  # Output: ['apple', 'orange', 'banana']
```

We note that the string is split at every occurrence of the delimiter, which can result in empty strings if the delimiter occurs multiple times consecutively. If no delimiter is specified, the method splits the string by whitespace characters (spaces, newlines, tabs, etc.), but treats consecutive whitespace characters as a single delimiter. Note that the 'split' method has both time and space complexity $O(n)$.

Joining a list of substrings into a string

One can use the 'join' method on a list of strings to merge them into a single string, separated each time by a delimiter (another string) which is passed as the implicit argument:

```
words = ["apple", "orange", "banana"]
result = ":".join(words)
print(result) # Output: apple:orange:banana
```

As a special (and often useful) case, one can create a string from a list of characters by calling the 'join' method on the empty string:

```
def list_to_string(char_list):
    return ''.join(char_list)
```

Creating a list from a string (and other conversions)

One can create a list from a string by simply calling the constructor which converts an iterable into a list:

```
# String to list
s = "abc"
l = list(s)
print(l)  # Output: ['a', 'b', 'c']
```

One can perform similar conversions between many other pairs of datatypes. For example:

```
# String to integer
string_num = "123"
num = int(string_num)
print(num)  # Output: 123
```

Checking for duplicates

To check whether a list contains duplicates, a nice technique is to convert it into a set and compare the length of the set to the length of the original list:

```
def contains_duplicates(lst):
    return len(lst) != len(set(lst))
```

This method has a time complexity of $O(n)$ on average due to the hashing involved in creating a set.