
Tricks

by David Oniani



Contents

1	Balanced Parentheses	3
2	Kadane's Algorithm - Maximum Contiguous Subarray	4
3	Levenshtein Distance - Edit Distance	5
4	Merge Sort	6
5	Floyd's Cycle Finding Algorithm - Linked List Cycle Detection	7
6	Reverse Linked List	8
7	Trie - A Prefix Tree	9

1 Balanced Parentheses

The parentheses in a string are balanced if and only if the following two conditions are met:

1. The number of "(" and ")" are equal
2. Scanning through the string from left to right and counting how many "(" and ")" there are so far, there should never be a time where there are more ")" than "(". We denote the balance of the string as `balance = count("(") - count(")")`

If we only allow for "()" pair of parentheses, then the following will check if the parenthesis are balanced:

```
def is_balanced(string: str) -> bool:
    """Check if a given string has balanced parentheses."""

    balance: int = 0
    for char in string:
        if char == "(":
            balance += 1
        elif char == ")":
            balance -= 1

        if balance < 0:
            return False

    return balance == 0
```

If we allow for more than one pair of parentheses, the following stack-based implementation generalizes well:

```
def is_balanced(string: str) -> bool:
    """Check if a given string has balanced parentheses."""

    stack: list[str] = []
    closing: dict[str, str] = {")": "(", "}": "{", "]": "["}
    opening: set[str] = set(closing.values())
    for char in string:
        if char in opening:
            stack.append(char)
        elif char in closing:
            if not stack or closing[char] != stack.pop():
                return False

    return not stack
```

2 Kadane's Algorithm - Maximum Contiguous Subarray

An implementation of the Kadane's algorithm.

```
def max_subarray_sum(arr: list[float]) -> float:
    """Return the maximum subarray sum."""

    cur: float = 0
    res: float = float("-inf")
    for num in arr:
        cur = max(cur + num, num)
        res = max(res, cur)
    return res
```

This algorithm is a trivial example of dynamic programming.

3 Levenshtein Distance - Edit Distance

Given two strings `word1` and `word2`, we need to find the optimal/minimum number of operations required to convert `word1` to `word2`. We are permitted the following four operations:

1. Insert a character
2. Delete a character
3. Replace a character
4. Keep a character

We take the bottom-up dynamic programming approach:

```
def levenshtein_distance(word1: str, word2: str) -> int:
    """Computes the Levenshtein's distance between two words."""

    # Get the lengths
    m: int = len(word1)
    n: int = len(word2)

    # Prepopulate the DP matrix
    dp: list[list[int]] = [
        [0 for _ in range(n + 1)] for _ in range(m + 1)
    ]

    # Initialize the DP matrix with base cases
    for i in range(1, m + 1):
        dp[i][0] += i
    for j in range(1, n + 1):
        dp[0][j] += j

    # Fill the DP Matrix
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            left: int = dp[i][j - 1]
            top: int = dp[i - 1][j]
            top_left: int = dp[i - 1][j - 1]

            # If characters are equal, just copy the diagonal value
            if word1[i - 1] == word2[j - 1]:
                dp[i][j] = top_left
            # Otherwise, take the minimum of three adjacent cells and
            # increment the result by one
            else:
                dp[i][j] = min(left, top, top_left) + 1

    # Return the optimal solution
    return dp[m][n]
```

4 Merge Sort

Merge sort is one of the simplest sorting algorithms that scales very well. It relies on two operations - merge and sort.

```
def merge(arr1: list[int], arr2: list[int]) -> list[int]:
    """Merge two sorted lists."""

    res: list[int] = []

    i: int = 0
    j: int = 0
    while i <= len(arr1) - 1 and j <= len(arr2) - 1:
        if arr1[i] <= arr2[j]:
            res.append(arr1[i])
            i += 1
        else:
            res.append(arr2[j])
            j += 1

    while i <= len(arr1) - 1:
        res.append(arr1[i])
        i += 1

    while j <= len(arr2) - 1:
        res.append(arr2[j])
        j += 1

    return res
```

We now need to implement the sort function.

```
def sort(arr: list[int]) -> list[int]:
    """Performs a merge sort."""

    if len(arr) <= 1:
        return arr

    mid: int = len(arr) // 2
    left: list[int] = sort(arr[:mid])
    right: list[int] = sort(arr[mid:])

    return merge(left, right)
```

5 Floyd's Cycle Finding Algorithm - Linked List Cycle Detection

```
from typing import Union

class Node:
    def __init__(self, val: int = 0, nxt: Node = Union[None, Node]):
        self._val = val
        self._next = nxt

def floyd_cycle(head: Node) -> bool:
    """An implementation of Floyd's Cycle Finding Algorithm."""

    if not head:
        return False

    # Initialize slow and fast pointers
    slow: Node = head
    fast: Node = head
    while slow != fast:
        # No need to check if `fast._next._next`
        # We check only `fast._next` so that `fast = fast._next._next`
        # does not result in an error
        if not fast or not fast._next:
            return False

        slow = slow._next
        fast = fast._next._next

    return True
```

6 Reverse Linked List

```
from typing import Union

class Node:
    def __init__(self, val: int = 0, next: Node = Union[None, Node]):
        self._val = val
        self._next = next

def reverse(head: Node) -> bool:
    """Reverse a linked list."""

    cur: Node = head
    pre: Node = Union[None, Node]
    while cur:
        # Variable `tmp` is needed for moving forward in the loop
        tmp: Node = cur._next

        # Rearrange pointers
        cur.next: Union[None, Node] = pre
        pre = cur

        # Move forward
        cur = tmp

    return pre
```


7 Trie - A Prefix Tree

There are many ways to construct a trie. We will use a hash map based approach. We first create a Node class.

```
class TrieNode:
    def __init__(self) -> None:
        """Initialize the `TrieNode` class."""

        self._children: dict[str, TrieNode] = {}
        self._isendofword: bool = False
```

We can now design the trie data structure. The initializer will only contain a root of the trie.

```
class Trie:
    def __init__(self) -> None:
        """Initialize the `Trie` class."""

        self._root = TrieNode()
```

Now, the first method to implement is the insertion operation. If we cannot insert a word into a trie, we are not going to be able to do much with it. The algorithm is very straightforward. One iterates over all characters in the word to be inserted and checks if the character is in the children of the current node. If it is, follow the path of that node. If not, create a new node at that position and once again, follow that road.

```
def insert(self, word: str) -> None:
    """Initialize the `Trie` class."""

    ptr: Node = self._root
    for char in word:
        if char not in ptr._children:
            ptr._children[char] = TrieNode()
        ptr = ptr._children[char]

    ptr._isendofword = True
```

One of the primary features of the trie is to be able to search for a word. The implementation of this algorithm is shown below.

```
def search(self, word: str) -> bool:
    """Check if a trie contains a given word."""

    ptr: Node = self._root
    for char in word:
        if char not in ptr._children:
            return False
        ptr = ptr._children[char]

    return ptr._isendofword
```

Similarly, searching a prefix is also very important. It looks almost exactly the same as the code for searching a word.

```
def prefix_search(self, word: str) -> bool:
    """Check if a trie contains a given prefix."""

    ptr: Node = self._root
    for char in word:
        if char not in ptr._children:
            return False
        ptr = ptr._children[char]

    return True
```