Tricks

by David Oniani

Contents

1 Backtracking

1.1 Generate Permutations

Optimal Solution:

Time Complexity:	$O(m!)$ where m is the length of the array.
Space Complexity:	$O(m!)$ where m is the length of the array.

```python
def permute(nums: list[int]) -> list[list[int]]:
    """Use backtracking to generate all permutations."""

    # Use Python's implicit "booleannes"
    # `[]` => `False` (generally works for empty sequences)
    if not nums:
        return []

    # The array to be filled
    res: list[list[int]] = []

    # Backtrack and fill the results list
    # NOTE: lists in Python work like pass-by-reference in C++
    backtrack(nums, 0, [], res)

    # Return the result
    return res


def backtrack(
    nums: list[int], start: int, cur: list[int], res: list[list[int]]
) -> None:
    """Perform backtracking on the array of numbers."""

    # Base case: we got one full permutation
    if len(cur) == len(nums):
        # Need to copy since
        res.append(cur[:])
        return

    # Recursive case: generate, recurse, and backtrack
    for i in range(start, len(nums)):
        # Append to the current permutation
        cur.append(nums[i])

        # Place the `i`th element at the start position
        # NOTE: Needed to include previous elements (before `start`)
        nums[i], nums[start] = nums[start], nums[i]

        # Recurse
        backtrack(nums, start + 1, cur, res)

        # Backtrack
        nums[start], nums[i] = nums[i], nums[start]
        cur.pop()
```

2 Dynamic Programming

2.1 Largest Area Square

Optimal Solution:

Time Complexity:	$O(m \times n)$ where m is the number of rows and n is the number of columns.
Space Complexity:	$O(m \times n)$ where m is the number of rows and n is the number of columns.

Given an $m \times n$ binary matrix of 0s and 1s, find the area of the largest square containing only 1s.

```python
def largest_area(self, matrix: list[list[int]]) -> int:
    """Return the area of the largest area square containing only 1s."""

    m: int = len(matrix)
    n: int = len(matrix[0])

    dp: list[list[int]] = [[0] * n for _ in range(m)]

    res: int = 0
    for i in range(m):
        for j in range(n):
            if matrix[i][j] == 1:
                dp[i][j] = 1 + min(dp[i - 1][j], dp[i][j - 1], dp[i - 1][j - 1])
                res = max(res, dp[i][j])

    return res ** 2
```

2.2 Levenshtein Distance (Edit Distance)

Optimal Solution:

Time Complexity:	$O(m \times n)$ where m is the number of rows and n is the number of columns.
Space Complexity:	$O(m \times n)$ where m is the number of rows and n is the number of columns.

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:

```python
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] * (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]
```

3 Geometry

3.1 Bresenham's Circle Drawing Algorithm

Optimal Solution:

Time Complexity:	$O(m)$ where m is the radius of the circle.
Space Complexity:	$O(1)$.

Given a function that can draw a dot on the screen, write a function that draws the circle with the given radius.

```python
import matplotlib.pyplot as plt


def bresenham(radius: float) -> None:
    """Draw a circle using Bresenham's Circle Drawing Algorithm."""

    # Initial x coordinate
    x: int = 0

    # Initial y coordinate
    y: int = radius

    # Decision parameter
    d: int = 3 - 2 * radius

    # While y coordinate is not less than the x coordinate
    while x <= y:
        plt.scatter(
            [x, x, -x, -x, y, -y, y, -y],
            [y, -y, y, -y, x, x, -x, -x],
            marker=".",
        )

        if d <= 0:
            d += 4 * x + 6
        else:
            y -= 1
            d += 4 * (x - y) + 10

        x += 1

    plt.title("Bresenham's Circle Drawing Algorithm")
    plt.show()
```

3.2 Draw Circle (Trigonometry)

Optimal Solution:

Time Complexity:	$O(m)$ where m is the number of iterations.
Space Complexity:	$O(1)$.

Given a function that can draw a dot on the screen, write a function that draws the circle with the given radius.

```python
import numpy as np
import matplotlib.pyplot as plt

def draw_circle(radius: float, step: float) -> None:
    """Draw a circle with a given radius."""

    angle: float = 0.0
    while angle <= 360:
        angle += step
        a: float = np.deg2rad(angle)
        x: float = radius * np.cos(a)
        y: float = radius * np.sin(a)
        plt.plot(x, y, marker=".")

    plt.title("Trigonometry-based Circle Drawing Algorithm")
    plt.show()
```

3.3 π Approximation

Optimal Solution:

Time Complexity:	$O(m)$ where m is the number of iterations.
Space Complexity:	$O(1)$.

Given a function producing random numbers from the uniform distribution of $[0, 1)$, estimate π.

```python
import numpy as np


def pi(iterations: int) -> float:
    """A function for approximating the Pi."""

    circle_area: int = 0
    square_area: int = 0
    for _ in range(iterations):
        # Notice the following:
        #     (1) 0 <= `np.random.uniform()` < 1
        #     (2) 0 <= `2 * np.random.uniform()` < 2
        #     (3) -1 <= `2 * np.random.uniform()`  - 1 < 1
        x: float = 2 * np.random.uniform() - 1
        y: float = 2 * np.random.uniform() - 1
        distance: float = np.sqrt(x ** 2 + y ** 2)
        if distance <= 1:
            circle_area += 1
        square_area += 1

    return circle_area / square_area * 4
```

3.4 Valid Square

Optimal Solution:

Time Complexity: $O(1)$.
Space Complexity: $O(1)$.

Given the coordinates of four points in 2D space p_1, p_2, p_3 and $p4$, check if the given four points can construct a valid square.

```python
import numpy as np

from collections import Counter


# A type to denote a point in two-dimensional space
Point = tuple[float, float]


def valid_square(p1: Point, p2: Point, p3: Point, p4: Point) -> bool:
    """Checks if the given coordinates form a valid square."""

    # All possible edges
    edges: list[tuple[Point, Point]] = [
        (p1, p2),
        (p1, p3),
        (p1, p4),
        (p2, p3),
        (p2, p4),
        (p3, p4),
    ]

    def distance(p1: Point, p2: Point) -> float:
        """Calculate the the Euclidean distance between two points."""

        return np.sqrt((p1[0] - p2[0]) ** 2 + (p1[1] - p2[1]) ** 2)

    # Counts
    distances: Counter[float] = Counter(
        map(lambda edge: distance(edge[0], edge[1]), edges)
    )

    # All edges must be different with only two corresponding values
    if 0 in distances or set(distances.values()) != {2, 4}:
        return False

    return True
```

4 Linked List

4.1 Floyd's Cycle Finding Algorithm (Linked List Cycle Detection)

Optimal Solution:

Time Complexity:	$O(m)$ where m is the number of nodes.
Space Complexity:	$O(m)$ where m is the number of nodes.

```python
from typing import Optional


class ListNode:
    def __init__(self, val: int = 0, next: Optional[ListNode]):
        self._val = val
        self._next = next


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

    # Make sure that the linked list is not empty
    if not head:
        return False

    # Initialize slow and fast pointers
    slow: Optional[ListNode] = head
    fast: Optional[ListNode] = head.next
    while slow != fast:
        # Check only `fast._next` so that `fast._next._next` exists
        if not fast or not fast._next:
            return False

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

    return True
```

4.2 Remove Elements

Optimal Solution:

Time Complexity:	$O(m)$ where m is the length of the list.
Space Complexity:	$O(m)$ where m is the length of the list.

Remove all nodes of the given value from the linked list.

```python
from typing import Optional


class ListNode:
    def __init__(self, val: int = 0, next: Optional[ListNode]):
        self._val = val
        self._next = next


def remove_elements(head: ListNode, val: int) -> Optional[ListNode]:
    """Remove elements from the linked list."""

    # Dummy node is needed as the first node can also be equal to `val`
    dummy: ListNode = ListNode(0)
    dummy._next: ListNode = head

    pre: ListNode = dummy
    cur: Optional[ListNode] = head
    while cur:
        # If the given value is encountered, we unlink the element
        if cur._val == val:
            pre._next = cur._next
        # Otherwise, we move forward
        else:
            pre = pre._next

        # The current point always moves forward
        cur = cur._next

    # Return `dummy._next` as first node is a throwaway node
    return dummy._next
```

4.3 Reverse Linked List

Optimal Solution:

Time Complexity:	$O(m)$ where m is the number of nodes.
Space Complexity:	$O(m)$ where m is the number of nodes.

```python
from typing import Optional


class ListNode:
    def __init__(self, val: int = 0, next: Optional[ListNode]):
        self._val = val
        self._next = next


def reverse(head: ListNode) -> Optional[ListNode]:
    """Reverse a linked list."""

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

        # Rearrange pointers
        cur._next: Optional[ListNode] = pre
        pre = cur

        # Move forward
        cur = tmp

    return pre
```

5 Problem Dependent

5.1 Balanced Parentheses

Optimal Solution:

Time Complexity:	$O(m)$ where m is the length of the string.
Space Complexity:	$O(n)$ where n is the number parentheses symbols.

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:

```python
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:

```python
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
```

5.2 Boyer-Moore Majority Vote Algorithm – Majority Element

Optimal Solution:

Time Complexity:	$O(m)$ where m is the length of the array.
Space Complexity:	$O(1)$.

Given an array of integers, find the majority element in the array. The majority element is the element that appears more than $floor(\frac{n}{2})$ times. Assume that the majority element exists in the array.

```python
def majority_element(nums: list[int]) -> int:
    """Find the majority element in the array."""

    candidate: int = 0
    count: int = 0
    for num in nums:
        # Candidate gets changed iff `count == 0`
        candidate = candidate if count else num
        # Increment/decrement based on whether candidate equals current
        count += 1 if candidate == num else -1

    return candidate
```

5.3 Flood Fill – Number of Islands

Optimal Solution:

Time Complexity:	$O(m \times n)$ where m is the number of rows and n is the number of columns.
Space Complexity:	$O(m \times n)$ where m is the number of rows and n is the number of columns.

```python
def islands(grid: list[list[str]]) -> int:
    """Counts the number of islands of 1s."""

    m: int = len(grid)
    n: int = len(grid[0])

    islands: int = 0
    for i in range(m):
        for j in range(n):
            if grid[i][j] == "1":
                self.fill(grid, i, j)
                islands += 1

    return islands


def flood_fill(grid: list[list[int]], i: int, j: int) -> bool:
    """Implements the Flood Fill algorithm."""

    # Row index out of bounds
    if not (0 <= i <= len(grid) - 1):
        return False

    # Column index out of bounds
    if not (0 <= j <= len(grid[0]) - 1):
        return False

    # If the current symbol is not 1, no need to continue
    if grid[i][j] != "1":
        return False

    # Mark with a sentinel symbol
    grid[i][j] = "$"

    # Recurse in four directions
    flood_fill(grid, i - 1, j)
    flood_fill(grid, i + 1, j)
    flood_fill(grid, i, j - 1)
    flood_fill(grid, i, j + 1)
```

5.4 Kadane's Algorithm – Maximum Contiguous Subarray

Optimal Solution:

Time Complexity:	$O(m)$ where m is the length of the array.
Space Complexity:	$O(1)$

An implemmentation of the Kadane's algorithm.

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

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

This algorithm is a trivial example of dynamic programming.

5.5 Palindromic Substrings - Expand Around Center

Optimal Solution:

Time Complexity:	$O(m^2)$ where m is the length of the string.
Space Complexity:	$O(1)$.

Count the number of palindromic substrings in this given string.

```python
def palindromic_substrings(string: str) -> int:
    """Returns the number of palindromic substrings."""

    res: int = 0
    for idx, _ in enumerate(string):
        # For odd-length palindromes, with single-character center
        res += expand(string, idx, idx)
        # For even-length palindromes, with double-character center
        res += expand(string, idx, idx + 1)

    return res

def expand(string: str, i: int, j: int) -> int:
    """Expands around the center."""

    res: int = 0
    while i >= 0 and j <= len(string) - 1:
        # If characters are different, substring is not palindromic
        if string[i] != string[j]:
            break

        # Record the substring
        res += 1

        # Advance the pointers
        i -= 1
        j += 1

    return res
```

5.6 Rotate Array

Optimal Solution:

Time Complexity:	$O(m)$ where m is the length of the array.
Space Complexity:	$O(1)$.

Rotate array a given number of times. In other words, every element of the array will be shifted to the right the given number of times.

```
def rotate(arr: list[int], num: int) -> None:
    """Rotate array `num` times."""

    # Empty sequences evaluate to `False`
    if not arr:
        return arr

    # Notice that every `len(A)` operations, there is a cycle
    num: int = num % len(A)

    # Reverse the entire list
    rev(arr, 0, len(arr) - 1)

    # Reverse the first `num` numbers
    rev(arr, 0, num - 1)

    # Reverse the last `len(A) - num` numbers
    rev(arr, num, len(arr) - 1)


def rev(arr: list[int], i: int, j: int) -> None:
    """Reverse array from index `i` to index `j` inclusive."""

    while i <= j:
        arr[i], arr[j] = arr[j], arr[i]
        i += 1
        j -= 1
```

6 Sorting

6.1 Dutch National Flag Problem

Optimal Solution:

Time Complexity:	$O(m)$ where m is the length of the array.
Space Complexity:	$O(1)$.

Given an array nums with n objects colored red, white, or blue, sort them in-place so that objects of the same color are adjacent, with the colors in the order red, white, and blue. Integers 0, 1, and 2 are used to represent the color red, white, and blue, respectively.

The first approach performs two passes, but the space complexity is constant.

```python
from collections import Counter


def sort_colors(arr: list[int]) -> None:
    """Sort the list in-place."""

    # Counts
    counts: Counter[int] = Counter(arr)

    # Index
    idx: int = 0

    # Store all the 0s in the beginning
    while (counts[0] > 0):
        arr[idx] = 0
        counts[0] -= 1
        idx += 1

    # Then all the 1s
    while (counts[1] > 0):
        arr[idx] = 1
        counts[1] -= 1
        idx += 1

    # Finally, all the 2s
    while (counts[2] > 0):
        arr[idx] = 2
        counts[2] -= 1
        idx += 1
```

The second approach is more efficient in terms of runtime complexity as it only performs a single pass. The space complexity is still constant.

```
        def sort_colors(arr: list[int]) -> None:
            """Sort the list in-place."""

            ptr1: int = 0
            ptr2: int = len(arr) - 1
            cur: int = 0
            while cur <= ptr2:
                # If `cur` at 0, swap and advance both `cur` and `ptr1`
                if arr[cur] == 0:
                    arr[cur], arr[ptr1] = arr[ptr1], arr[cur]
                    cur += 1
                    ptr1 += 1
                # If `cur` at 2, swap and move only `ptr2` as `ptr2` can be 1
                elif arr[cur] == 2:
                    arr[cur], arr[ptr2] = arr[ptr2], arr[cur]
                    ptr2 -= 1
                # If `cur` at 0, just increment `cur`
                else:
                    cur += 1
```

6.2 Merge Sort

Optimal Solution:

Time Complexity:	$O(m \times log(m))$ where m is the length of the array.
Space Complexity:	$O(m)$ where m is the length of the array.

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(nums) <= 1:
        return nums

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

    return merge(left, right)
```

7 Trie

7.1 Trie (A Prefix Tree)

There are many ways to constuct 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 characters 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
```