LeetCode Cheatsheet

Résumé des concepts, astuces et algorithmes pour LeetCode

*Created by **Maël Le Guillouzic**.*
A collection of tricks, tips, and classic problems to tackle coding challenges efficiently.

1. Arithmetic

```
result = 10 + 30   # => 40
result = 40 - 10   # => 30
result = 50 * 5    # => 250
result = 16 / 4    # => 4.0 (Float Division)
result = 16 // 4   # => 4 (Integer Division)
result = 25 % 2    # => 1
result = 5 ** 3    # => 125
```

Note: Use `**` for power, not `^`.
Division `/` gives float, while `//` gives an integer.

---

2. Math Algorithms

Greatest Common Divisor (GCD) and Least Common Multiple (LCM):

```
from math import gcd
lcm = lambda a, b: a * b // gcd(a, b)
```

Perfect Squares:

```
import math
def is_perfect_square(n):
    return int(math.sqrt(n)) ** 2 == n
```

Sieve of Eratosthenes:

```
def sieve(n):
    primes = [True] * (n + 1)
    primes[0] = primes[1] = False
    for i in range(2, int(n**0.5) + 1):
        if primes[i]:
            for j in range(i*i, n+1, i):
                primes[j] = False
    return [x for x in range(n+1) if
       ↪ primes[x]]
```

---

3. Advanced Data Types

Heaps

```
import heapq

myList = [9, 5, 4, 1, 3, 2]
heapq.heapify(myList)  # Turn into Min Heap
print(myList)          # => [1, 3, 2, 5, 9,
   ↪ 4]
heapq.heappush(myList, 10)  # Insert 10
x = heapq.heappop(myList)   # Pop smallest
   ↪ item
print(x)  # => 1
```

Tip: Negate values to use Min Heap as Max Heap.

Stacks and Queues

```
from collections import deque

q = deque([1, 2, 3])
q.append(4)         # Add to right
q.appendleft(0)     # Add to left
print(q)            # => deque([0, 1, 2, 3, 4])
x = q.pop()         # Remove from right
y = q.popleft()     # Remove from left
print(x, y)         # => 4 0
q.rotate(1)         # Rotate right
print(q)            # => deque([3, 1, 2])
```

Manipulations Avancées des Collections

Counting Frequencies:

```
from collections import Counter
freq = Counter("abracadabra")
print(freq)  # {'a': 5, 'b': 2, 'r': 2, 'c':
   ↪ 1, 'd': 1}
```

Grouping Data:

```
from collections import defaultdict
groups = defaultdict(list)
groups['a'].append(1)
print(groups)  # {'a': [1]}
```

Sorting with Custom Keys:

```
arr = [(2, 'b'), (1, 'a'), (3, 'c')]
arr.sort(key=lambda x: x[0])  # Sort by the
   ↪ first element
```

---

4. Strings

Slicing Strings

```
s = 'mybacon'
print(s[2:5])  # => 'bac'
print(s[:2])   # => 'my'
print(s[::5])  # => '11111'
print(s[::-1]) # Reverse string
```

Check Strings

```
s = ’spam’
print(s in ’I saw spamalot!’)  # True
print(s not in ’The Holy Grail!’)  # True
```

4.1. Concatenation

```
s = ’spam’
t = ’egg’
print(s + t)  # => ’spamegg’
```

Formatting

```
name = "John"
print("Hello, %s!" % name)
print("%s is %d years old." % (name, 23))

txt = "My name is {0}, I’m
   ↪ {1}".format("John", 36)
```

String Transformations

```
s = "  hello world  "
s.strip()  # ’hello world’
s.replace(" ", "")  # ’helloworld’
```

Reversing Words

```
s = "hello world"
" ".join(s.split()[::-1])  # ’world hello’
```

Checking Palindromes

```
def is_palindrome(s):
    return s == s[::-1]
```

---

5. Input and Output

Getting User Input

```
name = input("Enter your name: ")
print(f"Hello, {name}!")
```

Joining Strings

```
result = "#".join(["John", "Peter", "Vicky"])
print(result)  # => ’John#Peter#Vicky’
```

Endswith Check

```
s = "Hello, world!"
print(s.endswith("!"))  # => True
```

---

6. Modules and Libraries

From a Module

```
from math import ceil, floor
print(ceil(3.7))  # => 4.0
print(floor(3.7))  # => 3.0
```

Functions and Attributes

```
import math
dir(math)
```

---

7. Working with Files

Reading a File

```
from pathlib import Path

path = Path(’siddhartha.txt’)
contents = path.read_text()
lines = contents.splitlines()
for line in lines:
    print(line)
```

Writing to a File

```
path = Path(’journal.txt’)
msg = "I love programming."
path.write_text(msg)
```

Reading a File Line by Line

```
with open("myfile.txt") as file:
    for line in file:
        print(line)
```

Reading with Line Numbers

```
file = open(’myfile.txt’, ’r’)
for i, line in enumerate(file, start=1):
    print(f"Number {i}: {line}")
```

Writing and Reading Strings

```
contents = {"aa": 12, "bb": 21}
with open("myfile1.txt", "w+") as file:
    file.write(str(contents))

with open("myfile1.txt", "r+") as file:
    contents = file.read()
print(contents)
```

Writing and Reading Objects

```
import json

contents = {"aa": 12, "bb": 21}
with open("myfile2.txt", "w+") as file:
    file.write(json.dumps(contents))

with open("myfile2.txt", "r+") as file:
    contents = json.load(file)
print(contents)
```

Deleting Files and Folders

```
import os

# Delete a file
os.remove("myfile.txt")

# Check and delete a file
if os.path.exists("myfile.txt"):
    os.remove("myfile.txt")
else:
    print("The file does not exist")

# Delete a folder
os.rmdir("myfolder")
```

8. Python One-Liners

Find k Largest/Smallest:

```
import heapq
largest = heapq.nlargest(3, [1, 5, 2, 9, 7])
    ↪ # [9, 7, 5]
smallest = heapq.nsmallest(3, [1, 5, 2, 9,
    ↪ 7]) # [1, 2, 5]
```

Flatten a List:

```
flat_list = [item for sublist in [[1, 2],
    ↪ [3, 4]] for item in sublist]
```

Count Unique Elements:

```
from collections import Counter
Counter("leetcode") # {’l’: 1, ’e’: 3, ’t’:
    ↪ 1, ’c’: 1, ’o’: 1, ’d’: 1}
```

9. Exceptions

Catching an Exception

```
prompt = "How many tickets do you need? "
num_tickets = input(prompt)
try:
    num_tickets = int(num_tickets)
except ValueError:
    print("Please try again.")
else:
    print("Your tickets are printing.")
```

Handling Multiple Exceptions

```
try:
    # Raise an error
    raise IndexError("This is an index
        ↪ error")
except IndexError as e:
    pass  # Handle IndexError
except (TypeError, NameError):
    pass  # Handle multiple exceptions
else:
    print("All good!")  # Runs if no
        ↪ exception is raised
finally:
    print("We can clean up resources here")
```

10. Classes

Defining a Class

```
class Dog:
    def __init__(self, name):
        self.name = name
    def sit(self):
        print(f"{self.name} is sitting.")

my_dog = Dog(’Peso’)
my_dog.sit()
```

Inheritance

```
class SARDog(Dog):
    def search(self):
        print(f"{self.name} is searching.")

my_dog = SARDog(’Willie’)
my_dog.search()
```

11. Python Lists

Defining Lists

```
li1 = []  # Empty list
li2 = [4, 5, 6]  # List with values
li3 = list((1, 2, 3))  # From a tuple
li4 = list(range(1, 11))  # From a range
```

Generating Lists

```
list(filter(lambda x: x % 2 == 1, range(1,
    ↪ 20)))  # Odd numbers
[x**2 for x in range(1, 11) if x % 2 == 1]
    ↪ # Squares of odd numbers
list(filter(lambda x: x > 5, [3, 4, 5, 6,
    ↪ 7]))  # Values > 5
```

List Operations

Append Items:

```
li = []
li.append(1)  # [1]
li.append(4)  # [1, 4]
```

List Slicing

Syntax: `a[start:end:step]`

```
a = [’spam’, ’egg’, ’bacon’, ’ham’]
a[1:4]   # [’egg’, ’bacon’, ’ham’]
a[:3]    # [’spam’, ’egg’, ’bacon’]
a[::-1]  # Reverse list
a[::2]   # Every second item
```

Remove Items:

```
li.pop()  # Removes and returns the last item
del li[0]  # Removes the first item
```

Concatenation:

```
odd = [1, 3, 5]
odd.extend([9, 11])  # [1, 3, 5, 9, 11]
odd + [7, 13]  # [1, 3, 5, 9, 11, 7, 13]
```

Sort & Reverse:

```
li = [3, 1, 4]
li.sort()  # [1, 3, 4]
li.reverse()  # [4, 3, 1]
```

12. Loops and Control Statements

Loop with Index

```
animals = ["dog", "cat", "mouse"]
for i, value in enumerate(animals):
    print(i, value)
```

Break and Continue

```
for num in range(5):
    if num == 3:
        break  # Exit the loop
    if num == 2:
        continue  # Skip this iteration
    print(num)
```

Using zip()

```
words = [’Mon’, ’Tue’, ’Wed’]
nums = [1, 2, 3]
for w, n in zip(words, nums):
    print(f"{n}: {w}")
```

For/Else

```
nums = [60, 70, 30, 110]
for n in nums:
    if n > 100:
        print(f"{n} is bigger than 100")
        break
else:
    print("Not found!")
```

13. Graphs

Graph Construction

```
from collections import defaultdict

edges = [(1, 2), (2, 3), (1, 3)]
graph = defaultdict(list)
for u, v in edges:
    graph[u].append(v)
```

Depth-First Search (DFS)

```
def dfs(graph, node, visited):
    if node not in visited:
        visited.add(node)
        for neighbor in graph[node]:
            dfs(graph, neighbor, visited)
```

Breadth-First Search (BFS)

```
from collections import deque

def bfs(graph, start):
    queue = deque([start])
    visited = set()
    while queue:
        node = queue.popleft()
        if node not in visited:
            visited.add(node)
            queue.extend(graph[node])
```

14. Dynamic Programming

Memoization with lru_cache

```
from functools import lru_cache

@lru_cache(None)
def fib(n):
    if n <= 1:
        return n
    return fib(n-1) + fib(n-2)
```

Knapsack Problem (DP Table)

```
def knapsack(weights, values, capacity):
    dp = [[0] * (capacity + 1) for _ in
        ↪ range(len(weights) + 1)]
    for i in range(1, len(weights) + 1):
        for w in range(capacity + 1):
            if weights[i-1] <= w:
                dp[i][w] = max(dp[i-1][w],
                    ↪ dp[i-1][w-weights[i-1]]
                    ↪ + values[i-1])
            else:
                dp[i][w] = dp[i-1][w]
    return dp[-1][-1]
```

15. Generators

Defining a Generator

```
def double_numbers(iterable):
    for i in iterable:
        yield i + i
```

Generator to List

```
values = (-x for x in [1, 2, 3, 4, 5])
gen_to_list = list(values)

print(gen_to_list)  # => [-1, -2, -3, -4, -5]
```

16. Functions

Positional Arguments

```
def varargs(*args):
    return args

varargs(1, 2, 3)  # => (1, 2, 3)
```

Keyword Arguments

```
def keyword_args(**kwargs):
    return kwargs

keyword_args(big="foot", loch="ness")  # =>
   ↪ {"big": "foot", "loch": "ness"}
```

Returning Multiple Values

```
def swap(x, y):
    return y, x

x, y = swap(1, 2)  # => x = 2, y = 1
```

Anonymous Functions

```
(lambda x: x > 2)(3)  # => True
(lambda x, y: x ** 2 + y ** 2)(2, 1)  # => 5
```

17. Tricks

Bit Manipulation

```
x & y  # AND
x | y  # OR
x ^ y  # XOR
x << 1 # Left shift
x >> 1 # Right shift
~x     # NOT
```

Check if a number is a power of 2:

```
def is_power_of_two(n):
    return n > 0 and (n & (n - 1)) == 0
```

Sliding Window (Trick)

```
def max_subarray_sum(nums, k):
    max_sum = curr_sum = sum(nums[:k])
    for i in range(k, len(nums)):
        curr_sum += nums[i] - nums[i - k]
        max_sum = max(max_sum, curr_sum)
    return max_sum
```

Performance Optimizations

Convert to set for fast lookup:

```
nums = [1, 2, 3, 4, 5]
target_set = set(nums)
print(10 in target_set)  # O(1) lookup
```

Pre-compute prefix sums:

```
prefix_sum = [0]
for num in nums:
    prefix_sum.append(prefix_sum[-1] + num)
```

Binary Search for Optimal Value

```
def min_capacity(weights, days):
    l, r = max(weights), sum(weights)
    while l < r:
        mid = (l + r) // 2
        if can_ship(weights, days, mid):
            r = mid
        else:
            l = mid + 1
    return l
```

Union-Find (DSU)

```
class UnionFind:
    def __init__(self, size):
        self.parent = list(range(size))
        self.rank = [1] * size

    def find(self, x):
        if x != self.parent[x]:
            self.parent[x] =
               ↪ self.find(self.parent[x])
               ↪ # Path compression
        return self.parent[x]

    def union(self, x, y):
        root_x = self.find(x)
        root_y = self.find(y)
        if root_x != root_y:
            if self.rank[root_x] >
               ↪ self.rank[root_y]:
                self.parent[root_y] = root_x
            elif self.rank[root_x] <
               ↪ self.rank[root_y]:
                self.parent[root_x] = root_y
            else:
                self.parent[root_y] = root_x
                self.rank[root_x] += 1
```

18. Structures de Données Spéciales

OrderedDict

Maintient l'ordre d'insertion dans un dictionnaire.

```
from collections import OrderedDict

od = OrderedDict()
od['a'] = 1
od['b'] = 2
print(od)  # => {'a': 1, 'b': 2}
```

Custom Heaps

Utiliser des tas avec priorité personnalisée, par exemple avec des tuples.

```
import heapq

heap = []
heapq.heappush(heap, (2, "task1"))
heapq.heappush(heap, (1, "task2"))
print(heapq.heappop(heap))  # => (1, "task2")
```

---

19. Techniques Algorithmiques

Backtracking Avancé: Sudoku Solver

Remplir un tableau Sudoku en utilisant le backtracking.

```
def solve_sudoku(board):
    def is_valid(num, row, col):
        for i in range(9):
            if board[row][i] == num or
                ↪ board[i][col] == num or
                ↪ board[row//3*3 +
                ↪ i//3][col//3*3 + i%3] ==
                ↪ num:
                return False
        return True

    for row in range(9):
        for col in range(9):
            if board[row][col] == '.':
                for num in map(str, range(1,
                    ↪ 10)):
                    if is_valid(num, row,
                        ↪ col):
                        board[row][col] = num
                        if
                            ↪ solve_sudoku(board):
                            return True
                        board[row][col] = '.'
                return False
    return True
```

Dynamic Programming Avancé: LIS

Trouver la Longest Increasing Subsequence (LIS) d'un tableau donné.

```
from bisect import bisect_left

def length_of_lis(nums):
    dp = []
    for num in nums:
        idx = bisect_left(dp, num)
        if idx == len(dp):
            dp.append(num)
        else:
            dp[idx] = num
    return len(dp)
```

---

20. Outils Python

Itertools

Générer des permutations, combinaisons et produits cartésiens.

```
from itertools import permutations,
    ↪ combinations, product

perms = list(permutations([1, 2, 3]))
combs = list(combinations([1, 2, 3], 2))
prod = list(product([1, 2], repeat=2))
```

Lambda Functions

Utiliser des fonctions lambda avec map et filter.

```
nums = [1, 2, 3, 4]
squares = list(map(lambda x: x ** 2, nums))
evens = list(filter(lambda x: x % 2 == 0,
    ↪ nums))
```

Problèmes Classiques

Sliding Window Problems:

Minimum Window Substring: *Trouver la plus petite sous-chaîne dans une chaîne donnée qui contient tous les caractères d'une autre chaîne, y compris les répétitions.*

```
from collections import Counter

def min_window(s, t):
    count_t = Counter(t)
    window = Counter()
    l = 0
    res = ""
    for r in range(len(s)):
        window[s[r]] += 1
        while all(window[c] >= count_t[c] for c in count_t):
            if not res or (r - l + 1) < len(res):
                res = s[l:r+1]
            window[s[l]] -= 1
            l += 1
    return res
```

Graph Problems:

DFS (Depth-First Search): *Parcourir un graphe en profondeur en explorant autant que possible chaque branche avant de revenir en arrière. Utilisé pour détecter des cycles, trouver des composants connectés, ou résoudre des labyrinthes.*

```
def dfs(graph, node, visited):
    if node not in visited:
        visited.add(node)
        for neighbor in graph[node]:
            dfs(graph, neighbor, visited)
```

BFS (Breadth-First Search): *Parcourir un graphe niveau par niveau, en explorant tous les voisins d'un nœud avant de passer au suivant. Idéal pour trouver les plus courts chemins dans des graphes non pondérés.*

```
from collections import deque

def bfs(graph, start):
    queue = deque([start])
    visited = set()
    while queue:
        node = queue.popleft()
        if node not in visited:
            visited.add(node)
            queue.extend(graph[node])
```

Binary Search Problems:

Binary Search classique: *Trouver la position d'un élément cible dans un tableau trié en divisant l'espace de recherche par deux à chaque étape. Complexité en $O(\log n)$.*

```
def binary_search(nums, target):
    l, r = 0, len(nums) - 1
    while l <= r:
        mid = (l + r) // 2
        if nums[mid] == target:
            return mid
        elif nums[mid] < target:
            l = mid + 1
        else:
            r = mid - 1
    return -1
```

Dynamic Programming Problems:

Knapsack Problem: *Trouver la meilleure combinaison d'objets, chacun ayant un poids et une valeur, pour maximiser la valeur totale tout en respectant une contrainte de capacité. Problème typique résolu par programmation dynamique.*

```
def knapsack(weights, values, capacity):
    dp = [[0] * (capacity + 1) for _ in range(len(weights) + 1)]
    for i in range(1, len(weights) + 1):
        for w in range(capacity + 1):
            if weights[i-1] <= w:
                dp[i][w] = max(dp[i-1][w], dp[i-1][w-weights[i-1]] + values[i-1])
            else:
                dp[i][w] = dp[i-1][w]
    return dp[-1][-1]
```

Two Sum: *Trouver deux nombres dans un tableau qui s'additionnent pour donner une cible. Utilise une approche avec dictionnaire pour un temps de recherche en $O(n)$.*

```
def two_sum(nums, target):
    seen = {}
    for i, num in enumerate(nums):
        diff = target - num
        if diff in seen:
            return [seen[diff], i]
        seen[num] = i
```

Longest Substring Without Repeating Characters: *Trouver la plus longue sous-chaîne sans caractères répétés en utilisant une fenêtre glissante.*

```
def length_of_longest_substring(s):
    char_set = set()
    l = 0
    max_length = 0
    for r in range(len(s)):
        while s[r] in char_set:
            char_set.remove(s[l])
            l += 1
        char_set.add(s[r])
        max_length = max(max_length, r - l + 1)
    return max_length
```

Merge Intervals: *Fusionner des intervalles qui se chevauchent dans une liste triée.*

```
def merge_intervals(intervals):
    intervals.sort(key=lambda x: x[0])
    merged = []
    for interval in intervals:
        if not merged or merged[-1][1] < interval[0]:
            merged.append(interval)
        else:
            merged[-1][1] = max(merged[-1][1], interval[1])
    return merged
```

Maximum Product Subarray: *Trouver le produit maximal d'un sous-tableau contigu dans un tableau donné.*

```
def max_product(nums):
    curr_max, curr_min, result = nums[0], nums[0], nums[0]
    for i in range(1, len(nums)):
        temp = curr_max
        curr_max = max(nums[i], nums[i] * curr_max, nums[i] * curr_min)
        curr_min = min(nums[i], nums[i] * temp, nums[i] * curr_min)
        result = max(result, curr_max)
    return result
```

Coin Change: *Trouver le nombre minimal de pièces nécessaires pour atteindre une somme cible.*

```
def coin_change(coins, amount):
    dp = [float('inf')] * (amount + 1)
    dp[0] = 0
    for coin in coins:
        for x in range(coin, amount + 1):
            dp[x] = min(dp[x], dp[x - coin] + 1)
    return dp[amount] if dp[amount] != float(’inf’) else -1
```

Trap Rain Water: *Calculer la quantité maximale d’eau retenue entre des barres dans un histogramme.*

```
def trap(height):
    left, right = 0, len(height) - 1
    max_left, max_right = 0, 0
    water = 0
    while left < right:
        if height[left] < height[right]:
            if height[left] >= max_left:
                max_left = height[left]
            else:
                water += max_left - height[left]
            left += 1
        else:
            if height[right] >= max_right:
                max_right = height[right]
            else:
                water += max_right - height[right]
            right -= 1
    return water
```

Subsets: *Générer toutes les sous-ensembles possibles d’un tableau donné.*

```
def subsets(nums):
    result = []
    def backtrack(start, path):
        result.append(path[:])
        for i in range(start, len(nums)):
            path.append(nums[i])
            backtrack(i + 1, path)
            path.pop()
    backtrack(0, [])
    return result
```

Palindrome Partitioning: *Découper une chaîne en sous-chaînes où chaque sous-chaîne est un palindrome.*

```
def partition(s):
    result = []
    def backtrack(start, path):
        if start == len(s):
            result.append(path[:])
            return
        for end in range(start, len(s)):
            if s[start:end+1] == s[start:end+1][::-1]:
                path.append(s[start:end+1])
                backtrack(end + 1, path)
                path.pop()
    backtrack(0, [])
    return result
```

Rotated Sorted Array Search: *Rechercher un élément dans un tableau trié mais pivoté à une position inconnue. Combine recherche binaire et vérification des intervalles.*

```
def search(nums, target):
    left, right = 0, len(nums) - 1
    while left <= right:
        mid = (left + right) // 2
        if nums[mid] == target:
            return mid
        if nums[left] <= nums[mid]:
            if nums[left] <= target < nums[mid]:
                right = mid - 1
            else:
                left = mid + 1
        else:
            if nums[mid] < target <= nums[right]:
                left = mid + 1
            else:
                right = mid - 1
    return -1
```

House Robber: *Maximiser les gains en visitant des maisons sans voler deux maisons consécutives. Résolu par programmation dynamique.*

```
def rob(nums):
    prev, curr = 0, 0
    for num in nums:
        prev, curr = curr, max(curr, prev + num)
    return curr
```

Number of Islands: *Compter le nombre de zones connectées (îles) dans une matrice 2D représentant des terres (1) et de l'eau (0). Résolu par DFS ou BFS.*

```
def num_islands(grid):
    def dfs(x, y):
        if x < 0 or y < 0 or x >= len(grid) or y >= len(grid[0]) or grid[x][y] == ’0’:
            return
        grid[x][y] = ’0’
        dfs(x+1, y)
        dfs(x-1, y)
        dfs(x, y+1)
        dfs(x, y-1)

    count = 0
    for i in range(len(grid)):
        for j in range(len(grid[0])):
            if grid[i][j] == ’1’:
                count += 1
                dfs(i, j)
    return count
```