2025 Tech Olympiad Finals - Algorithm

Shahid Beheshti University

Init to win it

Ali Ghanbari, Zahra Ghalenave

October 28, 2025

Contents

1 Arrays, Strings & Sorting **3**
- 1.1 Longest Common Prefix . . . 3
- 1.2 Maximum Value And Number Of Occurences 3
- 1.3 Maximum Consecutive Ones . . . 3
- 1.4 Majority Element . . . 3
- 1.5 Number Of Distinct Values . . . 3
- 1.6 Single Number . . . 4
- 1.7 Find Duplicates . . . 4
- 1.8 Find Second Largest - Solution 1 . . . 4
- 1.9 Find Second Largest - Solution 2 . . . 4
- 1.10 Group Anagrams . . . 4
- 1.11 Count Binary Substrings . . . 5
- 1.12 Rotate One To Right . . . 5
- 1.13 Minimum Absolute Difference . . . 5
- 1.14 Best Time To Buy And Sell One Stock . . . 5
- 1.15 Increasing Triplet . . . 5

2 Nested Loops & Brute Force Algorithms **6**
- 2.1 Index Of Substring . . . 6
- 2.2 Longest Common Prefix Of Multiple Strings . 6
- 2.3 Repeated Substring Pattern . . . 6
- 2.4 Count Triangles . . . 6
- 2.5 Max Sum Subarray . . . 7
- 2.6 Sum Of Subarray Maximums . . . 7

3 Recursion **7**
- 3.1 Recursive Array Sum . . . 7
- 3.2 Recursive Reverse String . . . 7
- 3.3 Generate Pattern . . . 7
- 3.4 Recursive First Occurence . . . 7
- 3.5 Flatten Multidimensional Array . . . 8

4 Backtracking **8**
- 4.1 Generate Subsets . . . 8
- 4.2 Generate Permutations . . . 8
- 4.3 Generate Parentheses . . . 8
- 4.4 Generate Valley Permutations . . . 9
- 4.5 Word Search . . . 9

5 Stacks **9**

6 Two Pointers & Sliding Window **9**

7 Partial Sums **9**
- 7.1 Max Sum Of 3 Non Overlapping Subarrays . 9

8 Graphs **10**
- 8.1 DFS Find If Path Exists In Graph . . . 10
- 8.2 BFS Min Distance To Every Vertex . . . 10
- 8.3 Shortest Path With Alternating Colors . . . 10
- 8.4 Dijkstra's Algorithm . . . 11
- 8.5 Number Of Islands . . . 11
- 8.6 Word ladder . . . 11

9 Hash Maps **12**

10 Greedy **12**
- 10.1 Maximum Units On Truck . . . 12
- 10.2 Assign Cookies . . . 12
- 10.3 Max Profit Assigning Work . . . 12
- 10.4 Non-overlapping Intervals . . . 13
- 10.5 Meeting Rooms . . . 13

11 Linked Lists **13**
- 11.1 Copy List With Random Pointer . . . 13

12 Algorithmic Fundamentals **14**
- 12.1 Time complexities . . . 14
- 12.2 Math formulas . . . 14
- 12.3 Graph algorithms . . . 14
- 12.4 Data structures . . . 14
- 12.5 String algorithms . . . 14
- 12.6 Dynamic Programming patterns . . . 14
- 12.7 Modular arithmetic . . . 14

13 Essential Math Tables & Constants **14**
- 13.1 Basic Geometry — Areas, Perimeters, Volumes 14
- 13.2 Important Sequences & Small Tables . . . 15
- 13.3 Algebra & Series (basic sums) . . . 16
- 13.4 Basic Number Theory . . . 16
- 13.5 Combinatorics (essentials) . . . 16
- 13.6 Quick constants & reminders . . . 16

14 Graphs & Dynamic Programming **16**
14.1 Graph Traversal 16
14.2 Shortest Paths 16
14.3 Minimum Spanning Tree (MST) 17
14.4 Flows & Matchings 17
14.5 Connectivity & Components 17
14.6 Topological Sort 17
14.7 Classic DP Problems 17
14.8 Tree DP 17
14.9 Important Complexity Table 17

15 Advanced combinatorics & Number Theory **17**
15.1 Advanced Combinatorics 17
15.2 Multiplicative Functions and Transforms . . . 18
15.3 Euler Totient and Carmichael Function . . . 18
15.4 Theorems and Tools 18
15.5 Other Useful Numeric Facts 18

16 String Algorithms, FFT-NTT, Factorization **19**
16.1 String Algorithms 19
16.2 FFT-NTT and Convolution 19
16.3 Integer Factorization 19
16.4 Other Common Algorithms 19
16.5 Pseudo-code Snippets 19

17 Computational Geometry & Miscellaneous **20**
17.1 Vector Geometry 20
17.2 Polygons 20
17.3 Convex Hull 20
17.4 Closest Pair of Points 20
17.5 Line & Circle Intersections 20
17.6 Sweep Line 20
17.7 Voronoi & Delaunay 20
17.8 Miscellaneous Useful Formulas 20
17.9 Coordinate Geometry Tricks 20
17.10 Important Complexity Table 20

1 Arrays, Strings & Sorting

1.1 Longest Common Prefix

```python
def areEqual(strs, index):
  for i in range(1, len(strs)): # O(n)
    if strs[i][index] != strs[0][index]:
      return False
  return True


def longestCommonPrefix(strs: list[str]) -> str:
  # O(n * L) / O(L)
  minLength = len(strs[0])
  for i in range(1, len(strs)): # O(n)
    minLength = min(minLength, len(strs[i]))
  # minLength = min([len(word) for word in strs]) # O(n *
      L)
  longestCommonPrefix = ''
  for i in range(minLength): # O(L)
    if areEqual(strs, i): # O(n)
      longestCommonPrefix += strs[0][i] # O(1)
    else:
      break
  return longestCommonPrefix # O(L)


print(longestCommonPrefix(['flower',
                           'flow',
                           'flood',
                           'flair']))
'''
Input: ['flower',
        'flow',
        'flood',
        'flair']
Output: 'fl'
'''
# Usage: Find the longest common prefix among multiple
    strings
# Useful for: string processing, dictionary/trie problems
    , prefix analysis
```

1.2 Maximum Value And Number Of Occurences

```python
def maxValNumOfOccurrences(nums: list[int]) -> list[int]:
  # O(n) / O(1)
  maxVal = nums[0]
  counter = 0
  for num in nums:
    if num > maxVal:
      maxVal = num
      counter = 1
    elif num == maxVal:
      counter += 1
  return [maxVal, counter]


print(maxValNumOfOccurrences([2, 7, 11, 8, 11, 8, 3, 11])
    )
# Input: [2, 7, 11, 8, 11, 8, 3, 11]
# Output: [11, 3]
# Usage: Find the maximum value in an array and count its
      occurrences
# Useful for: array analysis, frequency counting,
    selection problems
```

1.3 Maximum Consecutive Ones

```python
def findMaxConsecutiveOnes(nums: list[int]) -> int:
  # O(n) / O(1)
  counter = 0
  solution = 0
  for num in nums:
    if num == 1:
      counter += 1
    else:
      counter = 0
    solution = max(solution, counter)
  return counter


print(findMaxConsecutiveOnes([1, 1, 0, 1, 1, 1]))
# Input: [1, 1, 0, 1, 1, 1]
# Output: 3
# Input: [1, 0, 1, 1, 0, 1]
# Output: 2
# Usage: Find the maximum number of consecutive ones in a
      binary array
# Useful for: array analysis, binary sequences, sliding
     window problems
```

1.4 Majority Element

```python
def majorityElement(nums: list[int]) -> int:
  # O(n log n) / O(1)
  counter = 1
  maxCounter = 1
  solution = nums[0]
  nums.sort()
  for i in range(1, len(nums)):
    if nums[i] == nums[i - 1]:
      counter += 1
    else:
      counter = 1
    if counter > maxCounter:
      maxCounter = counter
      solution = nums[i - 1]
  return solution


print(majorityElement([2, 2, 1, 1, 1, 3, 3, 3, 3]))
# Input: [3, 2, 3]
# Output: 3
# Input: [2, 2, 1, 1, 1, 2, 2]
# Output: 2
# Usage: Find the element that appears more than half of
     the array
# Useful for: array analysis, frequency counting, voting
     problems
```

1.5 Number Of Distinct Values

```python
def numOfDistinctValues(nums: list[int]) -> int:
  # O(n log n) / O(1)
  sol = 1
  nums.sort()
  for i in range(1, len(nums)):
```

```python
    if nums[i] != nums[i - 1]:
      sol += 1
  return sol


print(numOfDistinctValues([1, 5, -3, 1, -4, 2, -4, 7, 7])
    )
# Input: [1, 5, -3, 1, -4, 2, -4, 7, 7]
# Output: 6
# Usage: Count the number of distinct elements in an
    array
# Useful for: array analysis, duplicates handling,
    sorting-based problems
```

1.6 Single Number

```python
def isSingleNumber(nums, index):
  if index > 0 and nums[index - 1] != nums[index]:
    return False
  if index < len(nums) - 1 and nums[index] != nums[index
      + 1]:
    return False
  return True


def singleNumber(nums: list[int]) -> int:
  # O(n log n)
  if len(nums) == 1:
    return nums[0]
  nums.sort()
  for i in range(0, len(nums)):
    if isSingleNumber(nums, i):
      return nums[i]


print(singleNumber([4, 1, 2, 1, 2]))
# Input: [2, 2, 1]
# Output: 1
# Input: [4, 1, 2, 1, 2]
# Output: 4
# Usage: Find the element that appears exactly once in an
     array
# Useful for: array analysis, duplicates handling,
    sorting-based problems
```

1.7 Find Duplicates

```python
def isDuplicate(nums, index):
  if index > 0 and nums[index] == nums[index - 1]:
    return False
  if index == len(nums) - 1 or nums[index] != nums[index
      + 1]:
    return False
  return True


def findDuplicates(nums: list[int]) -> list[int]:
  # O(n log n) / O(n)
  nums.sort()
  duplicates = []
  for i in range(len(nums)):
    if isDuplicate(nums, i):
      duplicates.append(nums[i])
  return duplicates


print(findDuplicates([1, 5, 1, 2, 3, 5, 4]))
# Input: [2, 3, 1, 1, 4, 3, 2, 1]
# Output: [2, 1, 3]
# Usage: Find all elements that appear more than once in
    an array
# Useful for: array analysis, counting duplicates,
    sorting-based problems
```

1.8 Find Second Largest - Solution 1

```python
def secondLargest(nums: list[int]) -> int:
  # O(n log n) / O(1)
  nums.sort(reverse=True)
  for num in nums:
    if num != nums[0]:
      return num


print(secondLargest([2, 7, 11, 8, 11, 8, 3, 11]))
# Input: [2, 7, 11, 8, 11, 8, 3, 11]
# Output: 8
# Usage: Find the second largest element by sorting the
    array
# Useful for: array analysis, selection problems
```

1.9 Find Second Largest - Solution 2

```python
def secondLargest(nums: list[int]) -> int:
  # O(n) / O(1)
  largest = secondLargest = None
  for num in nums:
    if not largest or num > largest:
      secondLargest = largest
      largest = num
    elif num != largest and (not secondLargest or num >
        secondLargest):
      secondLargest = num
  return secondLargest


print(secondLargest([1000, 100, 100]))
# Input: [2, 7, 11, 8, 11, 8, 3, 11]
# Output: 8
# Input: [1000, 100, 100]
# Output: 100
# Usage: Find the second largest element in an array
# Useful for: array analysis, selection problems, in-
    place operations
```

1.10 Group Anagrams

```python
def groupAnagrams(strings: list[str]) -> list[list[str]]:
  # O(n log n)
  strings.sort(key=lambda word: ''.join(sorted(word)))
  currGroup = [strings[0]]
  groups = []
  for i in range(1, len(strings)):
    if sorted(strings[i]) == sorted(strings[i - 1]):
      currGroup.append(strings[i])
    else:
      groups.append(currGroup)
      currGroup = [strings[i]]
```

```python
    groups.append(currGroup)
  return groups


print(groupAnagrams(['eat', 'tea', 'tan', 'ate', 'nat', 'bat']))
# Input: ['eat', 'tea', 'tan', 'ate', 'nat', 'bat']
# Output: [['bat'], ['nat', 'tan'], ['ate', 'eat', 'tea']]
# Usage: Group strings that are anagrams of each other
# Useful for: string manipulation, hashing, dictionary/trie problems
```

1.11 Count Binary Substrings

```python
def BinarySubstrings(s: str) -> int:
  # O(n) / O(1)
  sol = 0
  len1 = 0
  len2 = 1
  for i in range(1, len(s)):
    if s[i] == s[i - 1]:
      len2 += 1
    else:
      sol += min(len1, len2)
      len1 = len2
      len2 = 1
  sol += min(len1, len2)
  return sol


print(BinarySubstrings('00110011'))
# Input: '00110011'
# Output: 6
# Input: '10101'
# Output: 4
# Usage: Count binary substrings with equal consecutive 0s and 1s
# Useful for: string analysis, pattern counting, binary sequences
```

1.12 Rotate One To Right

```python
def rotate(nums: list[int]) -> None:
  # O(n) / O(1)
  aux_val = nums[-1]
  for i in range(len(nums)-1, 0, -1):
    nums[i] = nums[i - 1]
  nums[0] = aux_val
  return nums


print(rotate([1, 2, 3, 4, 5]))
# Input: [1, 2, 3, 4, 5]
# Output: [5, 1, 2, 3, 4]
# Input: [4, -2, 13, 1]
# Output: [1, 4, -2, 13]
# Usage: Rotate elements of an array by one position to the right
# Useful for: array manipulation, cyclic shifts, in-place operations
```

1.13 Minimum Absolute Difference

```python
def minimumAbsDifference(nums: list[int]) -> list[list[int]]:
  # O(n log n) / O(n^2)
  nums.sort()
  minDiff = nums[1] - nums[0]
  minDiffPairs = []
  for i in range(1, len(nums)):
    curDiff = nums[i] - nums[i - 1]
    if curDiff < minDiff:
      minDiff = curDiff
      minDiffPairs = [[nums[i - 1], nums[i]]]
    elif curDiff == minDiff:
      minDiffPairs.append([nums[i - 1], nums[i]])
  return minDiffPairs


print(minimumAbsDifference([4, 2, 1, 3]))
# Input: [4, 2, 1, 3]
# Output: [[1, 2], [2, 3], [3, 4]]
# Input: [3, 8, -10, 23, 19, -4, -14, 27]
# Output: [[-14, -10], [19, 23], [23, 27]]
# Usage: Find all pairs with minimum absolute difference in a list
# Useful for: array sorting problems, consecutive pair analysis
```

1.14 Best Time To Buy And Sell One Stock

```python
def maxProfit(prices: list[int]) -> int:
  # O(n) / O(1)
  maxProfit = 0
  maxPrice = prices[-1]
  for buyDay in range(len(prices) - 2, -1, -1):
    currMaxProfit = maxPrice - prices[buyDay]
    maxProfit = max(maxProfit, currMaxProfit)
    maxPrice = max(maxPrice, prices[buyDay])
  return maxProfit


print(maxProfit([7, 1, 5, 3, 6, 4]))
# Input: [7, 1, 5, 3, 6, 4]
# Output: 5
# Input: [7, 6, 4, 3, 1]
# Output: 0
# Usage: Find maximum profit from a single buy/sell in stock prices
# Useful for: array analysis, greedy problems, financial algorithms
```

1.15 Increasing Triplet

```python
def increasingTriplet(nums: list[int]) -> bool:
  # O(n) / O(n)
  suffixMax = [0] * len(nums)
  suffixMax[-1] = nums[-1]
  for i in range(len(nums) - 2, -1, -1):
    suffixMax[i] = max(suffixMax[i + 1], nums[i])

  prefixMin = nums[0]
  for j in range(1, len(nums) - 1):
    if prefixMin < nums[j] and suffixMax[j + 1] > nums[j]:
      return True
    prefixMin = min(prefixMin, nums[j])
```

```
  return False


print(increasingTriplet([2, 1, 5, 0, 4, 6]))
# Input: [5, 4, 3, 2, 1]
# Output: False
# Input: [2, 1, 5, 0, 4, 6]
# Output: True
# Usage: Check if an increasing triplet subsequence
    exists in an array
# Useful for: subsequence problems, array analysis,
    greedy patterns
```

2 Nested Loops & Brute Force Algorithms

2.1 Index Of Substring

```
def isSubstring(haystack, needle, start): # O(m)
  for i in range(len(needle)):
    if needle[i] != haystack[start + i]:
      return False
  return True


def indexOf(haystack: str, needle: str) -> int:
  # O(n * m) / O(1)
  n = len(haystack)
  m = len(needle)
  for i in range(n - m + 1): # O(n - m)
    if isSubstring(haystack, needle, i): # O(m)
      return i
  return -1


print(indexOf('hello', 'll'))
# Input: "hello", "ll"
# Output: 2
# Input: "aaaaa", "bba"
# Output: -1
# Usage: Find the first occurrence of a substring in a
    string
# Useful for: string search, naive pattern matching
    problems
```

2.2 Longest Common Prefix Of Multiple Strings

```
def areEqual(strs, index):
  for i in range(1, len(strs)): # O(n)
    if strs[i][index] != strs[0][index]:
      return False
  return True


def longestCommonPrefix(strs: list[str]) -> str:
  # O(n * L) / O(L)
  minLength = len(strs[0])
  for i in range(1, len(strs)): # O(n)
    minLength = min(minLength, len(strs[i]))
  # minLength = min([len(word) for word in strs]) # O(n *
      L)
  longestCommonPrefix = ''
  for i in range(minLength): # O(L)
    if areEqual(strs, i): # O(n)
      longestCommonPrefix += strs[0][i] # O(1)
    else:
      break
  return longestCommonPrefix # O(L)


print(longestCommonPrefix(['flower',
                           'flow',
                           'flood',
                           'flair']))
'''
Input: ['flower',
        'flow',
        'flood',
        'flair']
Output: 'fl'
'''
# Usage: Find the longest common prefix among multiple
    strings
# Useful for: string processing, dictionary/trie problems
```

2.3 Repeated Substring Pattern

```
def isSolution(s, length): # O(n) / O(1)
  if len(s) % length:
    return False
  count = int(len(s) / length)
  for index in range(length): # O(length)
    for group in range(1, count): # O(count)
      if s[index] != s[index + group * length]:
        return False
  return True


def repeatedSubstringPattern(s: str) -> bool:
  # O(n^2) / O(1)
  for length in range(1, len(s)): # O(n)
    if isSolution(s, length): # O(n)
      return True
  return False


print(repeatedSubstringPattern('abcabcabcabc'))
# Input: 'abab'
# Output: True
# Input: 'aba'
# Output: False
# Input: 'abcabcabcabc'
# Output: True
# Usage: Check if a string is composed of repeated
    substring(s)
# Useful for: string pattern matching, periodicity
    detection
```

2.4 Count Triangles

```
def isTriangle(num1, num2, num3): # O(1)
  return num1 + num2 > num3 and num1 + num3 > num2 and
      num2 + num3 > num1


def countTriangles(nums: list[int]) -> int:
  # O(n^3) / O(1)
  solution = 0
```

```
    for i in range(len(nums)): # O(n)
      for j in range(i + 1, len(nums)): # O(n)
        for k in range(j+1, len(nums)): # O(n)
          if isTriangle(nums[i], nums[j], nums[k]):
            solution += 1
    return solution


print(countTriangles([3, 5, 10, 7]))
# Input: [3, 5, 10, 7]
# Output: 2
# Explanation: (3, 5, 7), (5, 10, 7)
# Usage: Count number of triplets forming valid triangles
# Useful for: geometry problems, combinatorial
    enumeration
```

2.5 Max Sum Subarray

```
def maxSumSubArray(nums: list[int]) -> int:
  # O(n^2) / O(1)
  greatestSum = nums[0]
  for i in range(len(nums)): # O(n)
    currentSum = 0
    for j in range(i, len(nums)): # O(n)
      currentSum += nums[j] # O(1)
      greatestSum = max(greatestSum, currentSum)
  return greatestSum


print(maxSumSubArray([-2, -5, 6, -2, -3, 1, 5, -6]))
# Input: [-2, -5, 6, -2, -3, 1, 5, -6]
# Output: 7
# Explanation: sum([6, -2, -3, 1, 5]) = 7
# Usage: Brute-force maximum subarray sum
# Useful for: array subproblems, Kadanes algorithm
    comparison
```

2.6 Sum Of Subarray Maximums

```
def computeSum(nums: list[int]) -> int:
  # O(n^2) / O(1)
  totalSum = 0
  for i in range(len(nums)): # O(n)
    curMax = nums[i]
    for j in range(i, len(nums)): # O(n)
      curMax = max(curMax, nums[j])
      totalSum += curMax
  return totalSum


print(computeSum([2, 3, 4, 1]))
# Input: [2, 3, 4, 1]
# Output: 33
# Usage: Brute-force sum of maximums over all subarrays
# Useful for: subarray analysis, enumeration problems
```

3 Recursion

3.1 Recursive Array Sum

```
def sum(nums: list[int]) -> int:
  # O(n^2) / O(n^2)
  if not nums:
    return 0
  return nums[0] + sum(nums[1:]) # O(len) / O(len)


print(sum([1, 2, 3, 4, 5]))
# Input: [1, 2, 3, 4, 5]
# Output: 15
# Usage: Compute the sum of an array using recursion
# Useful for: recursion practice, divide-and-conquer
    illustration
```

3.2 Recursive Reverse String

```
def reverse(s: str) -> str:
  # O(n^2) / O(n^2)
  if not s:
    return ""
  return s[-1] + reverse(s[:-1]) # O(len) / O(len)


print(reverse('abcde'))
# Input: 'abcde'
# Output: 'edcba'
# Usage: Reverse a string using recursion
# Useful for: recursion practice, string manipulation
```

3.3 Generate Pattern

```
def pattern(n: int) -> list[int]:
  # O(n^2) / O(n^2)
  if n == 0:
    return [] # O(1)
  halfPattern = pattern(n - 1) # O(len) / O(len)
  return halfPattern + [n] + halfPattern


print(pattern(4))
# Input: 3
# Output: [1, 2, 1, 3, 1, 2, 1]
# Input: 4
# Output: [1, 2, 1, 3, 1, 2, 1, 4, 1, 2, 1, 3, 1, 2, 1]
# Usage: Generate recursive symmetric patterns
# Useful for: recursion practice, divide-and-conquer
    patterns
```

3.4 Recursive First Occurence

```
def firstOccurence(nums: list[int], value: int) -> int:
  # O(n^2) / O(n^2)
  if not nums:
    return -1
  index = firstOccurence(nums[:-1], value) # O(len)
  if index != -1:
    return index
  if value == nums[-1]:
    return len(nums) - 1
  return -1


print(firstOccurence([1, 3, 5, 7, 9], 11))
# Input: [2, 4, 8, 6, 8, 10], 8
# Output: 2
# Input: [1, 3, 5, 7, 9], 11
# Output: -1
```

```
# Usage: Find the first occurrence of a value in an array
     using recursion
# Useful for: recursion practice, search problems
```

3.5 Flatten Multidimensional Array

```
def flatten(item):
  # O(number of int items * maxDepth)
  # / O(number of int items * maxDepth)
  if type(item) is int: # base case
    return [item]
  flattened_array = []
  for inner_item in item:
    flattened_array += flatten(inner_item)
  return flattened_array


print(flatten([[[1, 2], 3], [[5, [6]], 7], 8]))
# Input: [0, [1, [2]], [[[3]]]]
# Output: [0, 1, 2, 3]
# Usage: Flatten a nested list of integers into a single
    flat list
# Useful for: recursion, tree-like structures, nested
    data processing
```

4 Backtracking

4.1 Generate Subsets

```
def subsets(nums: list[int]) -> list[list[int]]:
  def backtrack(currIdx):
    # T.C: O(2 ^ n * n)
    # S.C: O(2 ^ n * n)
    # T.C 1: O(n) => O(2 ^ n * n) times
    # T.C 2: O(1) => O(2 ^ n * n) times
    # [1 0 1 0 1] => n times the O(1), 1 time the O(n)
    # if we finished generating the current subset
    if currIdx == len(nums): # O(1)
      subsets.append(currSubset[:]) # O(n)
      return

    # if we insert nums[currIdx] into currSubset:
    currSubset.append(nums[currIdx]) # O(1)
    backtrack(currIdx + 1) # O(1)

    # if we don't insert nums[currIdx] into currSubset:
    currSubset.pop() # O(1)
    backtrack(currIdx + 1) # O(1)

  subsets = []
  currSubset = []
  backtrack(0)
  return subsets # O(2 ^ n * n)


print(subsets([1, 2, 3]))
# Input: [1, 2]
# Output: [[], [1], [2], [1, 2]]
```

4.2 Generate Permutations

```
def permute(nums: list[int]) -> list[list[int]]:
  # O(n ^ 2 * n!) / O(n! * n)
  def backtrack():
    # T.C.1: O(n) => O(n * n!)
    # T.C.2: O(n) => O(n ^ 2 * n!)
    if len(currPermutation) == len(nums):
      permutations.append(currPermutation[:]) # O(n)
      return
    # [4, 1, 2, 5, 3] => n times O(n ^ 2) and 1 time O(n)
    #  What number should I place in currPermutation[len(
        currPermutation)]
    availableList = list(availableNumbers) # O(len) => O(
        n)
    for num in availableList: # O(len)
      currPermutation.append(num) # O(1)
      availableNumbers.remove(num) # O(1)
      backtrack() # O(1)
      # backtrack:
      currPermutation.pop() # O(1)
      availableNumbers.add(num) # O(1)

  permutations = []
  currPermutation = []
  availableNumbers = set(nums)
  backtrack()
  return permutations # O(n! * n)


print(permute([1, 2, 3]))
# Input: [1, 2, 3]
# Output: [[1, 2, 3], [1, 3, 2], [2, 3, 1],
#          [2, 1, 3], [3, 1, 2], [3, 2, 1]]
```

4.3 Generate Parentheses

```
def generateParentheses(n: int):
  def backtrack(currSequence, noOfOpened):
    # O(4^n * n^2) / O(4^n * n)
    # n = 2
    #  "((??"
    # noOfOpened = 2
    if len(currSequence) == 2 * n:
      if not noOfOpened:
        validSequences.append(currSequence) # O(n)
      return
    # add '('
    if noOfOpened < 2 * n - len(currSequence):
      backtrack(currSequence + '(', noOfOpened + 1) # O(n
          )

    # add ')'
    if noOfOpened:
      backtrack(currSequence + ')', noOfOpened - 1) # O(n
          )

  validSequences = []
  backtrack("", 0)
  return validSequences


print(generateParentheses(3))
'''
Input: n = 3
Output: ["((()))",
         "(()())",
         "(())()",
         "()(())",
```

```
        ()()()"]
'''
```

4.4 Generate Valley Permutations

```python
def generateValleyPermutations(n: int):
  def backtrack(increasing):
    # O(n! * n^2) / O(n * n!)
    if len(currPermutation) == n:
      permutations.append(currPermutation[:])
      return
    availableNumbersList = list(availableNumbers)
    for num in availableNumbersList:
      originalIncreasing = increasing
      if currPermutation:
        if num < currPermutation[-1] and increasing:
          continue
        if num > currPermutation[-1]:
          increasing = True
      currPermutation.append(num)
      availableNumbers.remove(num)
      backtrack(increasing)
      currPermutation.pop()
      availableNumbers.add(num)
      increasing = originalIncreasing

  permutations = []
  currPermutation = []
  availableNumbers = set(range(1, n + 1))
  backtrack(False)
  return permutations


print(generateValleyPermutations(4))
# Input: 4
# Output: [[1, 2, 3, 4], [2, 1, 3, 4],
#         [3, 1, 2, 4], [3, 2, 1, 4],
#         [4, 1, 2, 3], [4, 2, 1, 3],
#         [4, 3, 2, 1], [4, 3, 1, 2]]
```

4.5 Word Search

```python
def exist(board, word: str) -> bool:
  # O(n * m * 3^k)
  def isValid(cell):
    return cell[0] >= 0 and cell[0] < len(board) and \
        cell[1] >= 0 and cell[1] < len(board[0])

  def backtrack():
    if len(currPath) == len(word):
      return True
    nextWordChar = word[len(currPath)]
    currCell = currPath[-1]
    directions = [[-1, 0], [0, 1], [1, 0], [0, -1]]
    for direction in directions:
      nextCell = [currCell[0] + direction[0], currCell[1]
            + direction[1]]
      if isValid(nextCell) and \
            board[nextCell[0]][nextCell[1]] ==
                nextWordChar:
        currPath.append(nextCell)
        originalChar = board[nextCell[0]][nextCell[1]]
        board[nextCell[0]][nextCell[1]] = '#'
        if backtrack():
          return True
        currPath.pop()
        board[nextCell[0]][nextCell[1]] = originalChar
    return False

  for row in range(len(board)):
    for col in range(len(board[0])):
      if board[row][col] == word[0]:
        currPath = [[row, col]]
        originalChar = board[row][col]
        board[row][col] = '#'
        if backtrack():
          return True
        board[row][col] = originalChar
  return False


print(exist([['A', 'C', 'E', 'E'],
            ['C', 'E', 'D', 'E'],
            ['S', 'C', 'A', 'D'],
            ['A', 'D', 'D', 'E']], "ACEC"))
'''
Input: board = [['A', 'C', 'E', 'E'],
                ['C', 'E', 'D', 'E'],
                ['S', 'C', 'A', 'D'],
                ['A', 'D', 'D', 'E']],
      word = 'ABCCED'
Output: true
'''
```

5 Stacks

6 Two Pointers & Sliding Window

7 Partial Sums

7.1 Max Sum Of 3 Non Overlapping Subarrays

```python
def maxSumOf3SubArrays(nums: list[int]):
  # O(n) / O(n)
  n = len(nums)
  leftMaxSum = [0] * n # / O(n)
  leftMaxSum[1] = nums[0]
  maxSum = nums[0]
  for i in range(2, n): # O(n)
    maxSum = nums[i - 1] + max(maxSum, 0)
    leftMaxSum[i] = max(leftMaxSum[i - 1], maxSum)

  rightMaxSum = [0] * n # / O(n)
  rightMaxSum[-1] = nums[-1]
  maxSum = nums[-1]
  for i in range(n - 2, -1, -1): # O(n)
    maxSum = nums[i] + max(maxSum, 0)
    rightMaxSum[i] = max(rightMaxSum[i + 1], maxSum)

  partialSums = [0] # / O(n)
  for i in range(n): # O(n)
    partialSums.append(partialSums[i] + nums[i])

  maxSum = float('-inf')
  maxDiff = float('-inf')
  for right2 in range(1, len(nums) - 1): # O(n)
```

```
    maxDiff = max(maxDiff, leftMaxSum[right2] -
        partialSums[right2])
    maxSum = max(maxSum, partialSums[right2 + 1] +
            rightMaxSum[right2 + 1] + maxDiff)

  return maxSum


print(maxSumOf3SubArrays([2, 3, -8, 7, -2, 9, -9, 7, -2,
    4]))
# Input: [2, 3, -8, 7, -2, 9, -9, 7, -2, 4]
# Output: 28
# Usage: Max sum of 3 non-overlapping subarrays using
    prefix + DP
# Useful for: array DP, max subarray problems, interview-
    style challenges
```

8 Graphs

8.1 DFS Find If Path Exists In Graph

```
from collections import defaultdict


def validPath(n: int, edges: list[list[int]], source: int
    , destination: int):
  # O(n + m) / O(n + m)
  def dfs(node):
    visited.add(node) # Total: O(n)
    for adj_node in graph[node]: # Total: O(m)
      if adj_node not in visited:
        dfs(adj_node)

  graph = defaultdict(list[int]) # / O(m)
  visited = set() # / O(n)
  for edge in edges: # O(m)
    graph[edge[0]].append(edge[1])
    graph[edge[1]].append(edge[0])

  dfs(source)
  return destination in visited


print(validPath(6, [[0, 1], [0, 2], [2, 3], [3, 5],
                    [5, 4], [4, 3]], 0, 5))
'''
Input: n = 6
edges = [[0, 1], [0, 2], [2, 3], [3, 5], [5, 4], [4, 3]]
source = 0,
destination = 5
Output: True
'''
# Usage: Check if there is a valid path between two nodes
     in an undirected graph
# Useful for: connectivity problems, graph traversal
```

8.2 BFS Min Distance To Every Vertex

```
from collections import defaultdict, deque


def findMinDistances(n: int, edges: list[list[int]],
    source: int):
  # O(n + m) / O(n + m)
  graph = defaultdict(list[int]) # / O(m)
  for edge in edges: # O(m)
    graph[edge[0]].append(edge[1])
    # graph[edge[1]].append(edge[0])

  queue = deque([source])
  minDist = [-1] * n # / O(n)
  minDist[source] = 0

  while queue: # Total: O(n + m)
    node = queue.popleft() # Total: O(n)
    for adj_node in graph[node]: # Total: O(m)
      if minDist[adj_node] == -1:
        minDist[adj_node] = minDist[node] + 1
        queue.append(adj_node)

  return minDist


print(findMinDistances(8, [[0, 1], [0, 2], [0, 3],
                           [2, 1], [3, 4], [4, 2],
                           [4, 6], [4, 5], [5, 6],
                           [6, 7]], 0))
'''
Input: n = 8
edges = [[0, 1], [0, 2], [0, 3], [2, 1], [3, 4],
         [4, 2], [4, 6], [4, 5], [5, 6], [6, 7]]
source = 0,
Output: [0, 1, 1, 1, 2, 3, 3, 4]
'''
# Usage: Find minimum distances from a source node in an
    unweighted graph
# Useful for: shortest path problems, BFS traversal,
    graph analysis
```

8.3 Shortest Path With Alternating Colors

```
from collections import defaultdict, deque


def getAnswer(dist1, dist2):
  if dist1 == -1:
    return dist2
  if dist2 == -1:
    return dist1
  return min(dist1, dist2)


def shortestAlternatingPaths(n: int, redEdges: list[list[
    int]], blueEdges: list[list[int]], source: int):
  # O(n + m) / O(n + m)
  graph = defaultdict(list) # / O(n + m)
  for edge in redEdges:
    graph[edge[0]].append([edge[1], 0])
    # graph[edge[1]].append([edge[0], 0])
  for edge in blueEdges:
    graph[edge[0]].append([edge[1], 1])
    # graph[edge[1]].append([edge[0], 1])

  queue = deque([[source, 0], [source, 1]])
  minDist = [[-1, -1] for _ in range(n)] # O(n)
  minDist[source][0] = minDist[source][1] = 0

  while queue: # Total: O(n + m)
```

```python
    [node, last_color] = queue.popleft()
    for [adj_node, edge_color] in graph[node]: # Total: O
        (m)
      if edge_color != last_color and minDist[adj_node][
          edge_color] == -1:
        minDist[adj_node][edge_color] = minDist[node][
            last_color] + 1
        queue.append([adj_node, edge_color])

  answer = []
  for node in range(n): # O(n)
    answer.append(getAnswer(minDist[node][0], minDist[
        node][1]))

  return answer


print(shortestAlternatingPaths(7, [[0, 1], [1, 3], [2,
    3], [3, 5], [4, 5]],
                               [[0, 2], [2, 6], [2, 4], [3,
                                    4]], 0))
'''
Input: n = 7
redEdges = [[0, 1], [1, 3], [2, 3], [3, 5], [4, 5]]
blueEdges = [[0, 2], [2, 6], [2, 4], [3, 4]]
source = 0,
Output: [0, 1, 1, 2, 3, 4, -1]
'''
# Usage: Find shortest paths in a graph with alternating
    edge colors
# Useful for: BFS traversal, graph problems with edge
    constraints, shortest path analysis
```

8.4 Dijkstra's Algorithm

```python
from collections import defaultdict
import heapq # MAX HEAP!!!


def findMinDistances(n: int, edges: list[list[int]],
    source: int):
  # O((m + n) log m) / O(m + n)
  graph = defaultdict(list) # / O(m)
  for edge in edges: # O(m)
    graph[edge[0]].append([edge[1], edge[2]])
    # graph[edge[1]].append([edge[0], edge[2]])

  min_heap = []
  heapq.heappush(min_heap, [0, source])
  minDist = [-1] * n # / O(n)
  minDist[source] = 0

  while min_heap: # Total: O((m + n) log m)
    [distance, node] = heapq.heappop(min_heap)
    distance *= -1
    if distance != minDist[node]:
      continue
    for [adj_node, weight] in graph[node]: # Total: O(m)
      currDist = minDist[node] + weight
      if minDist[adj_node] == -1 or minDist[adj_node] >
          currDist:
        minDist[adj_node] = currDist
        heapq.heappush(min_heap, [-currDist, adj_node])

  return minDist


print(findMinDistances(7, [[0, 1, 6], [0, 2, 2], [2, 1,
    3],
                           [1, 4, 2], [2, 3, 1], [3, 1, 1],
                           [3, 4, 2], [4, 5, 1], [4, 6, 3]],
                                0))
'''
Input: n = 7
edges = [[0, 1, 6], [0, 2, 2], [2, 1, 3], [1, 4, 2], [2,
    3, 1],
         [3, 1, 1], [3, 4, 2], [4, 5, 1], [4, 6,
              3]]
source = 0,
Output: [0, 4, 2, 3, 5, 6, 8]
'''
# Usage: Find shortest paths from a source node in a
    weighted graph (Dijkstra)
# Useful for: graph traversal, shortest path problems,
    network analysis
```

8.5 Number Of Islands

```python
def findMinDistances(n: int, m: int, grid: list[list[int
    ]]):
  # O(n * m) / O(n * m)
  def dfs(i, j):
    visited[i][j] = True
    for x, y in [[i + 1, j], [i, j + 1],
                 [i - 1, j], [i, j - 1]]:
      if x >= 0 and x < n and y >= 0 and y < m and \
          grid[x][y] == 1 and not visited[x][y]:
        dfs(x, y)

  visited = [[False] * m for _ in range(n)] # / O(n * m)
  numOfIslands = 0

  for i in range(n):
    for j in range(m):
      if grid[i][j] and not visited[i][j]:
        numOfIslands += 1
        dfs(i, j) # Total: O(n * m)

  return numOfIslands


print(findMinDistances(4, 5, [[1, 1, 0, 0, 0],
                              [1, 1, 0, 0, 0],
                              [0, 0, 1, 0, 0],
                              [0, 0, 0, 1, 1]]))
'''
Input: n = 4, m = 5
grid = [[1, 1, 0, 0, 0],
        [1, 1, 0, 0, 0],
        [0, 0, 1, 0, 0],
        [0, 0, 0, 1, 1]]
Output: 3
'''
# Usage: Count number of connected islands (1s) in a 2D
    grid using DFS
# Useful for: graph traversal on grids, flood fill,
    connected components
```

8.6 Word ladder

```python
from collections import deque


def getStarWord(word, i):
  return word[:i] + "*" + word[i + 1:]


def ladderLength(words: list[str], beginWord: str,
    endWord: str) -> int:
  # O(w * L^2) / O(w * L)
  graph = {} # / O(w * L)
  for word in words: # O(w)
    for i in range(len(word)): # O(L)
      starWord = getStarWord(word, i) # O(L)
      graph[starWord] = graph.get(starWord, []) + [word]

  queue = deque() # / O(w)
  minDist = {}
  queue.append(beginWord)
  minDist[beginWord] = 1

  while queue: # O(w)
    word = queue.popleft()
    if endWord == word:
      break
    for i in range(len(word)): # O(L)
      starWord = getStarWord(word, i) # O(L)
      for nextWord in graph.get(starWord, []): # Total: O
          (w * L)
        if nextWord not in minDist:
          minDist[nextWord] = minDist[word] + 1
          queue.append(nextWord)
  return minDist.get(endWord, 0)


print(ladderLength(['hit', 'hot', 'dot', 'lot',
                    'dog', 'log', 'cog'], 'hit', 'cog'))
'''
Input: words = ['hit','hot','dot','lot','dog','log','cog'
    ]
begin = 'hit'
end = 'cog'
Output = 5
'''
```

9 Hash Maps

10 Greedy

10.1 Maximum Units On Truck

```python
def maximumUnits(boxTypes: list[list[int]], truckSize:
    int) -> int:
  # O(n log n) / O(1)
  noOfUnits = 0
  boxTypes.sort(key=lambda boxType: -boxType[1]) # O(n
      log n)
  for boxType in boxTypes: # O(n)
    boxesToTake = min(boxType[0], truckSize)
    noOfUnits += boxesToTake * boxType[1]
    truckSize -= boxesToTake
    if truckSize == 0:
      break
  return noOfUnits


print(maximumUnits([[1, 3], [2, 2], [3, 1]], 4))
'''
Input: boxTypes = [[1, 3], [2, 2], [3, 1]]
truckSize = 4
Output: 8
Input: boxTypes = [[5, 10], [2, 5], [4, 7], [3, 9]]
truckSize = 10
Output: 91
'''
# Usage: Maximize units loaded into truck by greedy sort
# Useful for: greedy knapsack-like problems
```

10.2 Assign Cookies

```python
def findContentChildren(g: list[list], s: list[int]) ->
    int:
  # O(n log n) + O(m log m)
  g.sort() # O(n log n)
  s.sort() # O(m log m)
  i, j = 0, 0
  solution = 0
  while i < len(g) and j < len(s): # O(n + m)
    if g[i] > s[j]:
      j += 1
    else:
      solution += 1
      i += 1
      j += 1
  return solution


print(findContentChildren([1, 2, 3], [1, 1]))
'''
Input: g = [1, 2, 3]
s = [1, 1]
Output: 1
Input: g = [2, 3, 5, 7]
s = [1, 2, 5, 6, 6]
Output: 3
'''
# Usage: Assign cookies to maximize content children (
    greedy)
# Useful for: greedy matching, resource allocation
```

10.3 Max Profit Assigning Work

```python
def maxProfitAssignment(difficulty: list[int], profit:
    list[int], worker: list[int]):
  #  O(n log n + m log m) / O(m)
  jobs = []
  for i in range(len(profit)): # O(m)
    jobs.append([difficulty[i], profit[i]])
  jobs.sort() # O(m log m)
  worker.sort() # O(n log n)
  maxTotalProfit = 0
  currJob = 0
  maxProfitForWorker = 0
  for worker in worker:
    while currJob < len(jobs) and jobs[currJob][0] <=
        worker:
```

```
      maxProfitForWorker = max(maxProfitForWorker, jobs[
          currJob][1])
      currJob += 1
    maxTotalProfit += maxProfitForWorker
  return maxTotalProfit


print(maxProfitAssignment([2, 4, 6, 8, 10],
      [10, 20, 30, 40, 50], [4, 5, 6, 7]))
'''
Input: difficulty = [2, 4, 6, 8, 10]
profit = [10, 20, 30, 40, 50]
worker = [4, 5, 6, 7]
Output: 100
'''
# Usage: Assign workers to jobs maximizing profit (greedy
      + sort)
# Useful for: job assignment, scheduling, resource
    optimization
```

10.4 Non-overlapping Intervals

```
def nonOverlappingIntervals(intervals: list[list[int]])
    -> int:
  # O(n log n) / O(1)
  intervals.sort(key=lambda interval: interval[1]) # O(n
      log n)
  result, lastTaken = 1, 0
  for i in range(1, len(intervals)): # O(n)
    if intervals[i][0] >= intervals[lastTaken][1]:
      result += 1
      lastTaken = i
  return result


print(nonOverlappingIntervals([[1, 3], [2, 4],
                               [3, 5], [3, 8],
                               [7, 9], [9, 12],
                               [6, 12]]))
'''
Input: [[1, 3], [2, 4], [3, 5], [3, 8],
        [7, 9], [9, 12], [6, 12]]
Output: 4
'''
# Usage: Find max non-overlapping intervals (greedy by
    end time)
# Useful for: interval scheduling, activity selection
```

10.5 Meeting Rooms

```
def meetingRooms(intervals: list[list[int]]) -> int:
  # O(n log n) / O(n)
  events = [] # / O(n)
  for start, end in intervals: # O(n)
    events.append([start, 1])
    events.append([end, -1])

  events.sort() # O(n log n)

  answer, noOfMeetings = 0, 0
  for time, type in events: # O(n)
    noOfMeetings += type
    answer = max(answer, noOfMeetings)

  return answer


print(meetingRooms([[1, 5], [2, 7], [0, 9], [5, 8], [7,
    11]]))
# Input: [[1, 5], [2, 7], [0, 9], [5, 8], [7, 11]]
# Output: 3
# Usage: Find minimum meeting rooms needed (max overlaps)
# Useful for: interval overlap, resource allocation
```

11 Linked Lists

11.1 Copy List With Random Pointer

```
# Definition for singly-linked list.
class ListNode:
  def __init__(self, x, next=None, random=None):
    self.val = x
    self.next = next
    self.random = random

def copyRandomList(head: ListNode) -> ListNode:
  # O(n) / O(1)
  if not head:
    return None

  # Step 1: Insert copy node right after each original
      node
  node = head
  while node:
    nextNode = node.next
    copyNode = ListNode(node.val)
    node.next = copyNode
    copyNode.next = nextNode
    node = nextNode

  # Step 2: Assign random pointers for the copied nodes
  node = head
  while node:
    copyNode = node.next
    if node.random:
      copyNode.random = node.random.next
    node = copyNode.next

  # Step 3: Separate the original list and the copied
      list
  newHead = head.next
  node = head
  while node:
    copyNode = node.next
    nextNode = copyNode.next
    if nextNode:
      copyNode.next = nextNode.next
    node.next = nextNode
    node = nextNode

  return newHead


# Usage: Deep copy linked list with next & random
    pointers
# Useful for: linked list cloning, O(1) extra space trick
# Example usage:
# Create nodes
```

```python
a = ListNode(7)
b = ListNode(13)
c = ListNode(11)
d = ListNode(10)
e = ListNode(1)
# Link with next pointers
a.next = b
b.next = c
c.next = d
d.next = e
# Link with random pointers
a.random = None
b.random = a
c.random = e
d.random = c
e.random = a
# Copy the list
copied_head = copyRandomList(a)
# Print copied list: show value and random value if
    exists
node = copied_head
while node:
  rand_val = node.random.val if node.random else None
  print(f"Node({node.val}), Random -> {rand_val}")
  node = node.next
```

12 Algorithmic Fundamentals

12.1 Time complexities

$O(1)$	constant time
$O(\log n)$	binary search, gcd, exponentiation
$O(n)$	linear scan, BFS, DFS
$O(n \log n)$	merge sort, quick sort, heap sort
$O(n^2)$	DP on substrings, Floyd–Warshall
$O(n^3)$	matrix multiplication (naive)

12.2 Math formulas

$$\gcd(a,b) = \begin{cases} b & a \bmod b = 0 \\ \gcd(b, a \bmod b) & \text{otherwise} \end{cases}$$

Fast exponentiation:

$$a^b \bmod m = \begin{cases} 1 & b = 0 \\ (a^{b/2})^2 \bmod m & b \text{ even} \\ a \cdot (a^{b-1} \bmod m) & b \text{ odd} \end{cases}$$

Sieve of Eratosthenes: find primes up to n in $O(n \log\log n)$.

12.3 Graph algorithms

Breadth First Search (BFS): $O(V+E)$.

Depth First Search (DFS): $O(V+E)$.

Dijkstra (using priority queue): $O((V+E)\log V)$.

Floyd–Warshall (all pairs shortest path): $O(V^3)$.

Kruskal (Minimum Spanning Tree with DSU): $O(E \log E)$.

12.4 Data structures

Segment Tree: range query + update in $O(\log n)$.

Fenwick Tree (BIT): prefix sums and updates in $O(\log n)$.

Disjoint Set Union (Union–Find): almost $O(1)$ per operation with path compression.

12.5 String algorithms

KMP algorithm: prefix-function in $O(n)$.

Z-function: compute in $O(n)$.

Hashing: polynomial rolling hash, $O(n)$ preprocessing.

12.6 Dynamic Programming patterns

1D DP: $dp[i] = \min(dp[i-1] + cost)$.

Knapsack: $dp[i][w] = \max(dp[i-1][w], dp[i-1][w-w_i] + v_i)$.

Longest Common Subsequence: $dp[i][j]$ for prefixes.

12.7 Modular arithmetic

$$(a+b) \bmod m = ((a \bmod m) + (b \bmod m)) \bmod m$$

$$(a \cdot b) \bmod m = ((a \bmod m) \cdot (b \bmod m)) \bmod m$$

Modular inverse (if m prime):

$$a^{-1} \equiv a^{m-2} \pmod m$$

13 Essential Math Tables & Constants

13.1 Basic Geometry — Areas, Perimeters, Volumes

Rectangle:

Area: $A = lw$ Perimeter: $P = 2(l+w)$

Square:

Area: $A = a^2$ Perimeter: $P = 4a$

Triangle:

Area: $A = \frac{1}{2}bh$ Perimeter: $P = a+b+c$

Heron: $s = \frac{a+b+c}{2}$, $A = \sqrt{s(s-a)(s-b)(s-c)}$

Equilateral triangle:

Area: $A = \frac{\sqrt{3}}{4}a^2$

Circle:

Area: $A = \pi r^2$ Circumference: $C = 2\pi r$ Diameter:

$d = 2r$

Ellipse:
Area: $A = \pi ab$ (semi-axes a, b)

Sphere:
Surface: $S = 4\pi r^2$ Volume: $V = \frac{4}{3}\pi r^3$

Cylinder:
Volume: $V = \pi r^2 h$ Surface (incl. bases): $S = 2\pi r(h + r)$

Cone:
Volume: $V = \frac{1}{3}\pi r^2 h$

Regular polygon (n sides, side length a):

$$\text{Area: } A = \frac{na^2}{4\tan(\pi/n)}$$

Distance (2D):

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Shoelace formula (polygon area):

$$A = \frac{1}{2}\left|\sum_{i=0}^{n-1}(x_i y_{i+1} - x_{i+1} y_i)\right|, \quad (x_n, y_n) = (x_0, y_0)$$

13.2 Important Sequences & Small Tables

Powers of 2:

$2^0 = 1$	$2^5 = 32$	$2^{10} = 1024$
$2^1 = 2$	$2^6 = 64$	$2^{11} = 2048$
$2^2 = 4$	$2^7 = 128$	$2^{12} = 4096$
$2^3 = 8$	$2^8 = 256$	$2^{13} = 8192$
$2^4 = 16$	$2^9 = 512$	$2^{14} = 16384$
$2^{15} = 32768$	$2^{16} = 65536$	$2^{20} = 1048576$

Powers of 3:

$3^0 = 1$	$3^5 = 243$	$3^{10} = 59049$
$3^1 = 3$	$3^6 = 729$	$3^{11} = 177147$
$3^2 = 9$	$3^7 = 2187$	$3^{12} = 531441$
$3^3 = 27$	$3^8 = 6561$	
$3^4 = 81$	$3^9 = 19683$	

Powers of 5:

$5^0 = 1$	$5^5 = 3125$	$5^{10} = 9765625$
$5^1 = 5$	$5^6 = 15625$	
$5^2 = 25$	$5^7 = 78125$	
$5^3 = 125$	$5^8 = 390625$	
$5^4 = 625$	$5^9 = 1953125$	

Factorials:

$0! = 1$	$5! = 120$	$10! = 3628800$
$1! = 1$	$6! = 720$	$11! = 39916800$
$2! = 2$	$7! = 5040$	$12! = 479001600$
$3! = 6$	$8! = 40320$	$13! = 6227020800$
$4! = 24$	$9! = 362880$	$14! = 87178291200$
		$15! = 1307674368000$

Fibonacci numbers (F_0 start):

$F_0 = 0$	$F_3 = 2$	$F_6 = 8$
$F_1 = 1$	$F_4 = 3$	$F_7 = 13$
$F_2 = 1$	$F_5 = 5$	$F_8 = 21$
	$F_9 = 34$	$F_{10} = 55$

Catalan numbers:

$C_0 = 1$	$C_3 = 5$	$C_6 = 132$
$C_1 = 1$	$C_4 = 14$	$C_7 = 429$
$C_2 = 2$	$C_5 = 42$	

Small primes:

2	19	47	79
3	23	53	83
5	29	59	89
7	31	61	97
11	37	67	
13	41	71	
17	43	73	

Number of primes:

30:	10
60:	17
100:	25
1000:	168
10000:	1229
100000:	9592
1000000:	78498
10000000:	664579

Central Binomial Coefficients $C(2n, n)$:

1:	2
2:	6
3:	20
4:	70
5:	252
6:	924
7:	3432
8:	12870
9:	48620
10:	184756
11:	705432
12:	2704156
13:	10400600
14:	40116600
15:	155117520

Numbers with Most Divisors:

$\le 10^2$: 60 with 12 divisors
$\le 10^3$: 840 with 32 divisors
$\le 10^4$: 7560 with 64 divisors
$\le 10^5$: 83160 with 128 divisors
$\le 10^6$: 720720 with 240 divisors
$\le 10^7$: 8648640 with 448 divisors
$\le 10^8$: 73513440 with 768 divisors
$\le 10^9$: 735134400 with 1344 divisors
$\le 10^{10}$: 6983776800 with 2304 divisors
$\le 10^{11}$: 97772875200 with 4032 divisors
$\le 10^{12}$: 963761198400 with 6720 divisors
$\le 10^{13}$: 9316358251200 with 10752 divisors
$\le 10^{14}$: 97821761637600 with 17280 divisors
$\le 10^{15}$: 866421317361600 with 26880 divisors
$\le 10^{16}$: 8086598962041600 with 41472 divisors
$\le 10^{17}$: 74801040398884800 with 64512 divisors
$\le 10^{18}$: 897612484786617600 with 103680 divisors

13.3 Algebra & Series (basic sums)

Sum of first n integers:

$$\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$$

Sum of squares:

$$\sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6}$$

Sum of cubes:

$$\sum_{k=1}^{n} k^3 = \left(\frac{n(n+1)}{2}\right)^2$$

Arithmetic progression:
$S_n = \frac{n}{2}(a_1 + a_n)$

Geometric progression:
$S_n = a\frac{1-r^n}{1-r}$ (if $r \neq 1$)

Finite geometric (special):
$1 + r + r^2 + \cdots + r^{n-1} = \frac{1-r^n}{1-r}$

13.4 Basic Number Theory

GCD / LCM:

$\gcd(a, b)$ by Euclid (iterative) $\quad \operatorname{lcm}(a, b) = \dfrac{|ab|}{\gcd(a, b)}$

Extended Euclid: Solve $ax + by = \gcd(a, b)$ for x, y

Modular exponentiation:
Compute $a^b \bmod m$ in $O(\log b)$ by binary exponentiation

Modular inverse:
If $\gcd(a, m) = 1$, inverse exists. If m prime: $a^{-1} \equiv a^{m-2} \pmod{m}$ (Fermat)

Euler's totient:
$\varphi(n) = n \prod_{p|n} \left(1 - \frac{1}{p}\right)$

Legendre formula (power of p in $n!$):
$\nu_p(n!) = \sum_{i\ge1} \left\lfloor \frac{n}{p^i} \right\rfloor$

Wilson's theorem:
p prime $\iff (p-1)! \equiv -1 \pmod{p}$

13.5 Combinatorics (essentials)

Factorial/Permutations:

$$P(n, k) = \frac{n!}{(n-k)!}$$

Binomial coefficient:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

Identities: $\sum_{k=0}^{n} \binom{n}{k} = 2^n$ $\quad$ Pascal: $\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$

Stars and bars:
Number of solutions nonnegative: $x_1 + \cdots + x_k = n \Rightarrow \binom{n+k-1}{k-1}$

Inclusion–Exclusion (2 sets):
$|A \cup B| = |A| + |B| - |A \cap B|$

13.6 Quick constants & reminders

Golden ratio: $\varphi = \dfrac{1+\sqrt{5}}{2} \approx 1.6180339887$

Binet (Fibonacci closed): $F_n = \dfrac{\varphi^n - (1-\varphi)^n}{\sqrt{5}}$

Useful approximations:
Stirling: $n! \sim \sqrt{2\pi n}\left(\dfrac{n}{e}\right)^n$

Hardy–Ramanujan (partition approx):
$p(n) \sim \dfrac{1}{4n\sqrt{3}} \exp\left(\pi\sqrt{\frac{2n}{3}}\right)$

14 Graphs & Dynamic Programming

14.1 Graph Traversal

BFS:
- Finds shortest path in unweighted graphs.
- Complexity $O(n + m)$.

DFS:
- For connectivity, cycle detection, topological sort.
- Complexity $O(n + m)$.

14.2 Shortest Paths

Dijkstra:

- Nonnegative edges.
- $O(m \log n)$ with heap.

Bellman-Ford:

- Handles negative edges.
- $O(nm)$. Detects negative cycles.

Floyd-Warshall:

- All-pairs shortest paths.
- $O(n^3)$.

SPFA:

- Optimization of Bellman-Ford, average faster.

14.3 Minimum Spanning Tree (MST)

Kruskal: Sort edges, union-find. $O(m \log m)$.

Prim: Use PQ, $O(m \log n)$.

14.4 Flows & Matchings

Max Flow:

- Edmonds-Karp $O(nm^2)$.
- Dinic $O(n^2 m)$, often faster.
- Push-Relabel: $O(n^3)$.

Min-Cost Max-Flow:

- Successive shortest path or cycle canceling.

Bipartite Matching:

- Hopcroft–Karp: $O(\sqrt{n}m)$.
- Hungarian Algorithm (Assignment): $O(n^3)$.

14.5 Connectivity & Components

SCC (Kosaraju / Tarjan):

- $O(n+m)$.

Bridges & Articulation points:

- Low-link values with DFS. $O(n+m)$.

2-SAT:

- Build implication graph, SCC. $O(n+m)$.

14.6 Topological Sort

Kahn's algorithm: repeatedly remove nodes indegree=0. $O(n+m)$.

DFS ordering: reverse postorder.

14.7 Classic DP Problems

Knapsack:

- 0/1: $O(nW)$.
- Unbounded: $O(nW)$.
- Optimizations: bitset, divide & conquer.

LIS (Longest Increasing Subsequence):

- $O(n \log n)$ via patience sorting.

Matrix Chain Multiplication:

- $dp[i][j] = \min_k(dp[i][k] + dp[k+1][j] + cost)$.

Edit Distance:

- $dp[i][j] = \min\{dp[i-1][j]+1, dp[i][j-1]+1, dp[i-1][j-1] + (a_i \neq b_j)\}$.
- Complexity $O(nm)$.

Subset Sum / Partition:

- Boolean DP: $dp[i][s] = true$ if subset of first i sums to s.

Bitmask DP (TSP):

- $dp[mask][i]$ = shortest path covering set mask ending at i.
- $O(n2^n)$.

14.8 Tree DP

Rerooting technique:

- Compute DP rooted at one node, then propagate to neighbors.

Examples:

- Count paths, subtree sums, DP on independent sets, etc.

14.9 Important Complexity Table

Algorithm	Complexity
DFS/BFS	$O(n+m)$
Dijkstra (PQ)	$O(m \log n)$
Bellman-Ford	$O(nm)$
Floyd-Warshall	$O(n^3)$
MST (Kruskal/Prim)	$O(m \log n)$
Dinic	$O(n^2 m)$
Hopcroft-Karp	$O(\sqrt{n}m)$
Hungarian	$O(n^3)$
Knapsack (0/1)	$O(nW)$
LIS	$O(n \log n)$
TSP (DP)	$O(n2^n)$

15 Advanced combinatorics & Number Theory

15.1 Advanced Combinatorics

Binomial identities:

$$\sum_{k=0}^{n}\binom{n}{k}=2^n,\quad \sum_{k=0}^{n}(-1)^k\binom{n}{k}=0\ (n\geq 1)$$

Algebraic identities:

$\binom{n}{k}=\binom{n}{n-k}$, Pascal: $\binom{n}{k}=\binom{n-1}{k}+\binom{n-1}{k-1}$

Vandermonde:

$$\sum_k\binom{r}{k}\binom{s}{n-k}=\binom{r+s}{n}$$

Multiset / combinations with repetition:

$\binom{n+k-1}{k-1}$ ways to distribute n identical items into k bins

Stirling numbers (2nd kind):

$S(n,k)$: partitions of n labeled objects into k nonempty unlabeled subsets.
Recurrence: $S(n,k)=kS(n-1,k)+S(n-1,k-1)$

Stirling numbers (1st kind, unsigned):

$c(n,k)$: coefficients in falling factorials. Recurrence: $c(n,k)=c(n-1,k-1)+(n-1)c(n-1,k)$

Bell numbers (partitions):

$B_n=\sum_{k=0}^{n}S(n,k)$, $\quad B_{n+1}=\sum_{k=0}^{n}\binom{n}{k}B_k$

Inclusion–Exclusion (general):

$$\left|\bigcup_{i=1}^{m}A_i\right|=\sum_i|A_i|-\sum_{i<j}|A_i\cap A_j|+\sum_{i<j<k}|A_i\cap A_j\cap A_k|-\cdots$$

Polya / Burnside (counting up to symmetry):

$|X/G|=\frac{1}{|G|}\sum_{g\in G}|\mathrm{Fix}(g)|$

15.2 Multiplicative Functions and Transforms

Multiplicative functions: f is multiplicative if $f(ab)=f(a)f(b)$ whenever $\gcd(a,b)=1$.

Examples: $1(n)=1$, $\mathrm{id}(n)=n$, $\varphi(n)$, $\mu(n)$, $d(n)$ (number of divisors).

Möbius function $\mu(n)$:

$\mu(1)=1$. If n has a squared prime factor, $\mu(n)=0$. Otherwise $\mu(n)=(-1)^k$, where k is the number of distinct primes dividing n.

Möbius inversion:

If $g(n)=\sum_{d|n}f(d)$, then $f(n)=\sum_{d|n}\mu(d)g(n/d)$

Divisor count $d(n)$ and divisor sum $\sigma_k(n)$:

If $n=\prod p_i^{e_i}$, then $d(n)=\prod(e_i+1)$, $\quad\sigma_k(n)=\prod\frac{p_i^{(e_i+1)k}-1}{p_i^k-1}$
Especially $\sigma_1(n)=\sigma(n)$ is the sum of divisors.

15.3 Euler Totient and Carmichael Function

Euler's totient $\varphi(n)$:

$\varphi(n)=n\prod_{p|n}\left(1-\frac{1}{p}\right)$
If $n=\prod p_i^{e_i}$, then $\varphi(n)=\prod p_i^{e_i-1}(p_i-1)$

Carmichael function $\lambda(n)$: (smallest m such that $a^m\equiv 1 \pmod n$ for all $\gcd(a,n)=1$)
For prime powers:

$$\lambda(p^e)=\begin{cases}\varphi(2^e)=2^{e-2} & \text{if } p=2, e\geq 3,\\ \varphi(p^e)=p^{e-1}(p-1) & \text{if } p \text{ odd prime.}\end{cases}$$

For general n: $\lambda(n)=\mathrm{lcm}(\lambda(p_i^{e_i}))$

15.4 Theorems and Tools

Fermat's little theorem:

$a^p\equiv a \pmod p$, $\quad$ if $\gcd(a,p)=1$ then $a^{p-1}\equiv 1 \pmod p$

Euler's theorem:

$a^{\varphi(n)}\equiv 1 \pmod n$ if $\gcd(a,n)=1$

Multiplicative order:

$\mathrm{ord}_n(a)$ is the smallest k with $a^k\equiv 1 \pmod n$; always divides $\lambda(n)$

Primitive root:

Exists for $n=2,4,p^k,2p^k$ with odd prime p.

Chinese Remainder Theorem (CRT):

If m_i are pairwise coprime, the system $x\equiv a_i \pmod{m_i}$ has a unique solution $\pmod{\prod m_i}$.

Lucas theorem (for $\binom{n}{k} \pmod p$, p prime):

Writing n,k in base p: $n=\sum n_ip^i$, $k=\sum k_ip^i$, then

$$\binom{n}{k}\equiv\prod_i\binom{n_i}{k_i} \pmod p$$

Lifting The Exponent (LTE):

For evaluating $\nu_p(x^n-y^n)$ there are formulas depending on p,x,y (handy in contests).

15.5 Other Useful Numeric Facts

Prime number approximation:

$\pi(x)\sim\frac{x}{\ln x}$ (approximate number of primes $\leq x$)

Trial division and factorization:

For factorization up to 10^{12}, trial division up to $\sqrt{n}$ or optimized sieve-based methods are fine; for larger numbers, Pollard Rho is recommended.

Modular arithmetic (add/mul/div):

Addition/subtraction/multiplication straightforward; division $\Rightarrow$ multiply by modular inverse: $a/b\equiv a\cdot b^{-1} \pmod m$ if inverse exists.

16 String Algorithms, FFT-NTT, Factorization

16.1 String Algorithms

Prefix-function (KMP):

Computes the length of the longest prefix which is also a suffix. Used in KMP for pattern matching.
Complexity: $O(n+m)$.

Z-function:

$Z[i]$ = length of the longest prefix starting at $s[i]$.
Complexity: $O(n)$.

Manacher's algorithm (Palindrome radii):

Computes all palindromic substrings in $O(n)$.

Suffix Array + LCP:

- Construct with radix sort in $O(n \log n)$, or SA-IS in $O(n)$.
- LCP with Kasai in $O(n)$.

Applications: search, longest repeated substring, count distinct substrings.

Aho-Corasick Automaton:

For multiple patterns (dictionary matching). Build: $O(\sum |p|)$, query $O(|text| + occ)$.

Rolling Hash (Polynomial Hash):

$H[i] = \sum_{k=0}^{i-1} s[k] \cdot p^k \pmod{M}$.
Applications: substring comparison, Rabin–Karp.

16.2 FFT-NTT and Convolution

Discrete Fourier Transform (DFT):

$A_k = \sum_{j=0}^{n-1} a_j \omega^{jk}, \quad \omega = e^{2\pi i/n}$.

FFT: Computes DFT in $O(n \log n)$.

Applications: polynomial multiplication, big integer multiplication.

NTT (Number Theoretic Transform):

Analog of FFT over a prime modulus $p = k2^m + 1$. Example: $p = 998244353$, primitive root=3.

Convolution Complexity:

Naive $O(n^2)$, Karatsuba $O(n^{\log_2 3})$, FFT/NTT $O(n \log n)$.

16.3 Integer Factorization

Pollard Rho: Probabilistic factorization algorithm for large integers. Average $O(n^{1/4})$.

Idea: function $f(x) = (x^2 + c) \bmod n$, find cycle with Floyd.

Fermat factorization: Good for numbers close to perfect squares: $n = a^2 - b^2 = (a-b)(a+b)$.

Miller–Rabin primality test:

For odd n, write $n - 1 = 2^s d$. Check $a^d \equiv 1$ or $a^{2^r d} \equiv -1$.
Otherwise composite.
Probabilistic, with fixed witnesses can be deterministic up to 2^{64}.

16.4 Other Common Algorithms

Union-Find (DSU):

Operations: find(x), union(x,y). Optimizations: path compression, union by rank.
Amortized complexity: $O(\alpha(n))$ (inverse Ackermann, almost constant).

Binary Lifting (LCA):

Precompute $up[v][k]$: 2^k-th ancestor.
$O(n \log n)$ build, $O(\log n)$ query.

Segment Tree:

Supports sum, min, lazy propagation. Complexity $O(\log n)$.

Fenwick Tree (BIT):

Supports prefix sums in $O(\log n)$. Lighter memory than segment tree.

16.5 Pseudo-code Snippets

KMP prefix-function:

```
pi[0]=0
for i=1..n-1:
  j=pi[i-1]
  while j>0 and s[i]!=s[j]:
    j=pi[j-1]
  if s[i]==s[j]: j++
  pi[i]=j
```

Z-function:

```
l=0,r=0
for i=1..n-1:
  if i<=r: z[i]=min(r-i+1,z[i-l])
  while i+z[i]<n and s[z[i]]==s[i+z[i]]: z[i]++
  if i+z[i]-1>r: l=i; r=i+z[i]-1
```

Pollard Rho (sketch):

```
def f(x): return (x*x+c)%n
x,y,d=2,2,1
while d==1:
  x=f(x); y=f(f(y))
  d=gcd(abs(x-y),n)
if d!=n: return d
```

17 Computational Geometry & Miscellaneous

17.1 Vector Geometry

- Dot product: $\vec{a} \cdot \vec{b} = |a||b| \cos\theta = a_x b_x + a_y b_y$
- Cross product (2D): $\vec{a} \times \vec{b} = a_x b_y - a_y b_x$
- Orientation test:
  - > 0: counter-clockwise
  - < 0: clockwise
  - $= 0$: collinear
- Distance from point P to line AB: $d = \frac{|(B-A)\times(P-A)|}{|B-A|}$
- Distance from point P to segment AB: project P on AB, check if projection lies in segment, else take min distance to endpoints

17.2 Polygons

- Area (shoelace formula): $A = \frac{1}{2} \left|\sum_{i=1}^{n} (x_i y_{i+1} - x_{i+1} y_i)\right|$
- Convex polygon check: all triples have same orientation
- Point in polygon (ray casting): count ray-edge intersections

17.3 Convex Hull

- Graham scan / Andrew monotone chain: $O(n \log n)$
- Jarvis march (gift wrapping): $O(nh)$, $h =$ hull size

17.4 Closest Pair of Points

- Divide and conquer algorithm: $O(n \log n)$

17.5 Line & Circle Intersections

- Lines $p + ta$ and $q + ub$: solve $p + ta = q + ub$ for (t, u)
- Segment intersection: check orientation + bounding boxes
- Circle: $(x - x_0)^2 + (y - y_0)^2 = r^2$
- Line-circle intersection: solve quadratic
- Circle-circle intersection: check distance between centers

17.6 Sweep Line

- Used for: segment intersection, closest pair, union of rectangles
- Complexity: $O((n+k) \log n)$ with balanced BST, $k =$ intersections

17.7 Voronoi & Delaunay

- Voronoi: partition plane by nearest site
- Delaunay: dual of Voronoi, maximizes minimum angle in triangles

17.8 Miscellaneous Useful Formulas

- Pick's theorem (lattice polygon): $A = I + \frac{B}{2} - 1$
- Euler's formula (planar graphs): $V - E + F = 2$
- Stirling's approximation: $n! \approx \sqrt{2\pi n} \left(\frac{n}{e}\right)^n$

17.9 Coordinate Geometry Tricks

- Rotate (x, y) by θ: $(x', y') = (x \cos\theta - y \sin\theta,\ x \sin\theta + y \cos\theta)$
- Reflect point P over line AB: project P onto AB, then double it

17.10 Important Complexity Table

Algorithm / Operation	Complexity
Convex Hull (Graham/Andrew)	$O(n \log n)$
Convex Hull (Jarvis)	$O(nh)$
Closest Pair	$O(n \log n)$
Sweep Line	$O((n + k) \log n)$