The Complete Bit Manipulation DSA Handbook

1. Binary Fundamentals & Number System

Binary Number System

Base-2 system using only 0s and 1s, unlike decimal (base-10) system.

Converting Decimal to Binary

```
def decimal_to_binary(n):
    result = ""
    while n > 0:
        result = str(n % 2) + result
        n = n // 2
    return result or "0"
```

Converting Binary to Decimal

```
def binary_to_decimal(binary_str):
    decimal = 0
    power = 0
    for i in range(len(binary_str) - 1, -1, -1):
        if binary_str[i] == '1':
            decimal += (2 ** power)
        power += 1
    return decimal
```

Integer Representation

- **32-bit integer**: Range from -2^{31} to $2^{31}-1$
- **Sign bit**: MSB (leftmost) represents sign (0=positive, 1=negative)
- **Two's complement**: How negative numbers are stored

2. Core Bitwise Operators

AND (&) - Both bits must be 1

```
# Truth table: 1&1=1, 1&0=0, 0&1=0, 0&0=0
a = 5   # 101
b = 3   # 011
c = a & b  # 001 = 1
```

Applications:

- Check if number is even: `n & 1 == 0`
- Clear specific bits
- Extract specific bits

OR (|) - At least one bit must be 1

```
# Truth table: 1|1=1, 1|0=1, 0|1=1, 0|0=0
a = 5   # 101
b = 3   # 011
c = a | b  # 111 = 7
```

Applications:

- Set specific bits
- Combine flags

XOR (^) - Bits must be different

```
# Truth table: 1^1=0, 1^0=1, 0^1=1, 0^0=0
a = 5   # 101
b = 3   # 011
c = a ^ b  # 110 = 6
```

Applications:

- Toggle bits
- Swap numbers without temp variable
- Find unique element

NOT (~) - Flip all bits

```
a = 5    # 00000101
b = ~a   # 11111010 = -6 (in two's complement)
```

Left Shift (<<) - Multiply by 2^k

```
a = 5    # 101
b = a << 2  # 10100 = 20 (5 * 2² = 5 * 4 = 20)
```

Right Shift (>>) - Divide by 2^k

```
a = 20   # 10100
b = a >> 2  # 101 = 5 (20 / 2² = 20 / 4 = 5)
```

3. Essential Bit Manipulation Tricks

3.1 Basic Bit Operations

Check if i-th bit is set

```
def is_bit_set(n, i):
    return (n & (1 << i)) != 0
```

Set i-th bit

```
def set_bit(n, i):
    return n | (1 << i)
```

Clear i-th bit

```
def clear_bit(n, i):
    return n & ~(1 << i)
```

Toggle i-th bit

```
def toggle_bit(n, i):
    return n ^ (1 << i)
```

3.2 Power of 2 Tricks

Check if number is power of 2

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

Get next power of 2

```
def next_power_of_two(n):
    if n <= 1:
        return 1
    n -= 1
    n |= n >> 1
    n |= n >> 2
    n |= n >> 4
    n |= n >> 8
    n |= n >> 16
    return n + 1
```

3.3 Bit Counting Tricks

Count number of set bits (Brian Kernighan's algorithm)

```
def count_set_bits(n):
    count = 0
    while n:
        n &= (n - 1)  # Remove rightmost set bit
        count += 1
    return count
```

Built-in bit counting

```
# Python
bin(n).count('1')

# C++
__builtin_popcount(n)
```

3.4 Bit Isolation Tricks

Get rightmost set bit

```
def rightmost_set_bit(n):
    return n & (-n)
```

Remove rightmost set bit

```
def remove_rightmost_set_bit(n):
    return n & (n - 1)
```

Get rightmost unset bit

```
def rightmost_unset_bit(n):
    return (n + 1) & (~n)
```

3.5 Number Manipulation Tricks

Swap two numbers without temp variable

```
def swap(a, b):
    a ^= b
    b ^= a
    a ^= b
    return a, b
```

Check if two numbers have opposite signs

```
def opposite_signs(a, b):
    return (a ^ b) < 0
```

Toggle between two values (A and B only)

```
def toggle_between(current, A, B):
    return A ^ B ^ current
```

4. Bit Manipulation Problem Patterns

Pattern 1: Single Number Problems

Find unique element (all others appear twice)

```
def single_number(nums):
    result = 0
    for num in nums:
        result ^= num
    return result
```

Find unique element (all others appear thrice)

```
def single_number_thrice(nums):
    ones = twos = 0
    for num in nums:
        twos |= ones & num
        ones ^= num
        not_threes = ~(ones & twos)
        ones &= not_threes
        twos &= not_threes
    return ones
```

Pattern 2: Subset Generation

Generate all subsets using bit manipulation

```
def generate_subsets(nums):
    n = len(nums)
    total_subsets = 1 << n  # 2^n
    result = []

    for mask in range(total_subsets):
        subset = []
        for i in range(n):
            if mask & (1 << i):
                subset.append(nums[i])
        result.append(subset)

    return result
```

Pattern 3: Bit Masking for DP

Traveling Salesman Problem (TSP)

```
def tsp_dp(dist, n):
    # dp[mask][i] = minimum cost to visit all cities in mask ending at city i
    dp = [[float('inf')] * n for _ in range(1 << n)]
    dp[^1][^0] = 0  # Start at city 0

    for mask in range(1 << n):
        for u in range(n):
            if mask & (1 << u) == 0:
                continue
            for v in range(n):
                if mask & (1 << v):
                    continue
                new_mask = mask | (1 << v)
                dp[new_mask][v] = min(dp[new_mask][v], dp[mask][u] + dist[u][v])

    # Return to start city
```

```
    result = float('inf')
    for i in range(1, n):
        result = min(result, dp[(1 << n) - 1][i] + dist[i][^0])

    return result
```

Pattern 4: Maximum XOR Problems

Maximum XOR of two numbers in array

```
class TrieNode:
    def __init__(self):
        self.children = {}

class Trie:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, num):
        node = self.root
        for i in range(31, -1, -1):
            bit = (num >> i) & 1
            if bit not in node.children:
                node.children[bit] = TrieNode()
            node = node.children[bit]

    def max_xor(self, num):
        node = self.root
        max_xor = 0
        for i in range(31, -1, -1):
            bit = (num >> i) & 1
            toggled_bit = 1 - bit
            if toggled_bit in node.children:
                max_xor |= (1 << i)
                node = node.children[toggled_bit]
            else:
                node = node.children[bit]
        return max_xor

def find_maximum_xor(nums):
    trie = Trie()
    max_xor = 0

    for num in nums:
        trie.insert(num)
        max_xor = max(max_xor, trie.max_xor(num))

    return max_xor
```

5. Advanced Bit Manipulation Techniques

5.1 Gray Code Generation

```
def gray_code(n):
    result = []
    for i in range(1 << n):
        # Convert i to Gray code
        gray = i ^ (i >> 1)
        result.append(gray)
    return result
```

5.2 Bit Reversal

```
def reverse_bits(n):
    result = 0
    for i in range(32):
        result = (result << 1) | (n & 1)
        n >>= 1
    return result
```

5.3 Count Trailing Zeros

```
def count_trailing_zeros(n):
    if n == 0:
        return 32
    count = 0
    while (n & 1) == 0:
        n >>= 1
        count += 1
    return count

# Using bit manipulation trick
def count_trailing_zeros_fast(n):
    if n == 0:
        return 32
    return (n & -n).bit_length() - 1
```

5.4 Fast Exponentiation using Bits

```
def fast_power(base, exp):
    result = 1
    while exp > 0:
        if exp & 1:  # If current bit is 1
            result *= base
        base *= base
        exp >>= 1
    return result
```

6. Bit Manipulation in Data Structures

6.1 Bitset Implementation

```
class Bitset:
    def __init__(self, size):
        self.size = size
        self.bits = [^0] * ((size + 31) // 32)  # 32-bit chunks

    def set(self, pos):
        if 0 <= pos < self.size:
            chunk = pos // 32
            bit = pos % 32
            self.bits[chunk] |= (1 << bit)

    def clear(self, pos):
        if 0 <= pos < self.size:
            chunk = pos // 32
            bit = pos % 32
            self.bits[chunk] &= ~(1 << bit)

    def test(self, pos):
        if 0 <= pos < self.size:
            chunk = pos // 32
            bit = pos % 32
            return (self.bits[chunk] & (1 << bit)) != 0
        return False
```

6.2 Binary Indexed Tree (Fenwick Tree)

```
class BIT:
    def __init__(self, n):
        self.n = n
        self.tree = [^0] * (n + 1)

    def update(self, i, val):
        while i <= self.n:
            self.tree[i] += val
            i += i & (-i)  # Add last set bit

    def query(self, i):
        total = 0
        while i > 0:
            total += self.tree[i]
            i -= i & (-i)  # Remove last set bit
        return total
```

7. Problem Categories & LeetCode Examples

Easy Level

1. **Number of 1 Bits** (LeetCode 191)
2. **Power of Two** (LeetCode 231)
3. **Power of Four** (LeetCode 342)
4. **Missing Number** (LeetCode 268)
5. **Single Number** (LeetCode 136)

Medium Level

1. **Single Number II** (LeetCode 137)
2. **Single Number III** (LeetCode 260)
3. **Subsets** (LeetCode 78)
4. **Subsets II** (LeetCode 90)
5. **Maximum XOR of Two Numbers** (LeetCode 421)
6. **Bitwise AND of Numbers Range** (LeetCode 201)

Hard Level

1. **N-Queens II** (LeetCode 52) - using bitmask
2. **Shortest Path Visiting All Nodes** (LeetCode 847) - TSP with bitmask
3. **Maximum Students Taking Exam** (LeetCode 1349) - bitmask DP

8. Common Bit Manipulation Formulas & Identities

Mathematical Properties

- `a ^ a = 0` (XOR with self is 0)
- `a ^ 0 = a` (XOR with 0 is unchanged)
- `a & (a-1) = 0` if a is power of 2
- `a | (a-1)` sets all trailing zeros to 1
- `-a = ~a + 1` (two's complement)

Useful Bit Tricks Table

Operation	Code	Description
Check even	`n & 1 == 0`	LSB is 0 for even numbers
Check odd	`n & 1 == 1`	LSB is 1 for odd numbers

Operation	Code	Description
Multiply by 2^k	`n << k`	Left shift by k positions
Divide by 2^k	`n >> k`	Right shift by k positions
Clear rightmost set bit	`n & (n-1)`	Removes last 1-bit
Get rightmost set bit	`n & (-n)`	Isolates last 1-bit
Set i-th bit	`n | (1 << i)`	Makes i-th bit 1
Clear i-th bit	`n & ~(1 << i)`	Makes i-th bit 0
Toggle i-th bit	`n ^ (1 << i)`	Flips i-th bit
Check if power of 2	`n > 0 && (n & (n-1)) == 0`	Only one bit set

9. Optimization Techniques

9.1 Space Optimization using Bits

Instead of using boolean arrays, use bits to save space:

```
# Instead of: visited = [False] * n
# Use: visited_bits = 0

def mark_visited(visited_bits, i):
    return visited_bits | (1 << i)

def is_visited(visited_bits, i):
    return (visited_bits & (1 << i)) != 0
```

9.2 Fast Operations

- **Modulo by power of 2**: `n % (2^k) = n & ((1 << k) - 1)`
- **Check divisibility by power of 2**: `n & ((1 << k) - 1) == 0`
- **Absolute value**: `(n ^ (n >> 31)) - (n >> 31)` (for 32-bit)

10. Debugging & Common Pitfalls

Common Mistakes

1. **Integer overflow** when shifting
2. **Sign extension** in right shifts of negative numbers
3. **Off-by-one errors** in bit positions (0-indexed vs 1-indexed)
4. **Mixing signed and unsigned** operations

Debugging Tips

```
def debug_bits(n, name="number"):
    print(f"{name}: {n} = {bin(n)} = {hex(n)}")

def print_bit_operations(a, b):
    print(f"a = {a:08b}")
    print(f"b = {b:08b}")
    print(f"a & b = {(a & b):08b}")
    print(f"a | b = {(a | b):08b}")
    print(f"a ^ b = {(a ^ b):08b}")
```

11. Practice Strategy

Step-by-Step Learning Path

1. **Master basic operations** (AND, OR, XOR, NOT, shifts)
2. **Learn fundamental tricks** (power of 2, bit counting, isolation)
3. **Practice subset generation** problems
4. **Study single number** variants
5. **Tackle bitmask DP** problems
6. **Advanced topics** (tries, complex optimizations)

Essential Problems to Practice

- **Beginner**: Power of 2, count bits, single number
- **Intermediate**: All subsets, XOR queries, bitmask DP
- **Advanced**: Maximum XOR with tries, complex bitmask problems

This comprehensive handbook covers all essential bit manipulation concepts, from basic operations to advanced algorithms. Master these patterns and techniques to excel in competitive programming and technical interviews!

⁂

1. https://opg.optica.org/abstract.cfm?URI=oe-28-2-1139
2. https://ieeexplore.ieee.org/document/10141709/
3. http://arxiv.org/pdf/1310.1306.pdf
4. https://arxiv.org/pdf/1511.07275.pdf
5. https://arxiv.org/pdf/2403.06898v1.pdf
6. http://arxiv.org/pdf/2307.15600.pdf
7. https://arxiv.org/pdf/0907.2173.pdf
8. https://arxiv.org/pdf/0711.0261.pdf

9. https://arxiv.org/pdf/2405.15088.pdf
10. https://arxiv.org/pdf/1906.04448.pdf
11. https://ieeexplore.ieee.org/document/10671980/
12. http://arxiv.org/pdf/2310.16165.pdf
13. http://arxiv.org/pdf/2503.05596.pdf
14. https://dev.to/anurag629/the-power-of-bit-manipulation-how-to-solve-problems-efficiently-3p1h
15. https://www.geeksforgeeks.org/competitive-programming/bit-tricks-competitive-programming/
16. https://www.w3schools.com/js/js_bitwise.asp
17. https://cp-algorithms.com/algebra/bit-manipulation.html
18. https://www.youtube.com/watch?v=LGrE0siZ-ZA
19. https://www.geeksforgeeks.org/c/bitwise-operators-in-c-cpp/
20. https://www.hackerearth.com/practice/basic-programming/bit-manipulation/basics-of-bit-manipulation/tutorial/
21. https://leetcode.com/discuss/study-guide/2960396/Bit-Manipulation-Guide-and-Tricks
22. https://www.mdpi.com/2079-9292/13/9/1648
23. https://en.wikipedia.org/wiki/Bitwise_operation
24. https://arminnorouzi.github.io/posts/2023/05/blog-post-15/
25. https://www.geeksforgeeks.org/dsa/bits-manipulation-important-tactics/
26. https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/bitwise-and-shift-operators
27. https://www.geeksforgeeks.org/dsa/bitwise-algorithms/
28. https://graphics.stanford.edu/~seander/bithacks.html
29. https://www.geeksforgeeks.org/dsa/complete-reference-for-bitwise-operators-in-programming-coding/
30. https://www.geeksforgeeks.org/dsa/all-about-bit-manipulation/
31. https://dev.to/stephengade/mastering-bitwise-operations-a-simplified-guide-2031
32. https://www.codechef.com/practice/bit-manipulation
33. https://ieeexplore.ieee.org/document/10958861/
34. https://usaco.guide/silver/intro-bitwise
35. https://www.youtube.com/watch?v=NLKQEOgBAnw
36. https://www.semanticscholar.org/paper/8b1ebf7d49d6a18141f76fd9bdd686db2afc63f8
37. https://www.semanticscholar.org/paper/4c64d2747b823f29359063883694f592e958b24f
38. http://ieeexplore.ieee.org/document/5692243/
39. https://ieeexplore.ieee.org/document/10623806/
40. https://iopscience.iop.org/article/10.1088/1742-6596/1444/1/012039
41. https://ieeexplore.ieee.org/document/9875122/
42. https://ieeexplore.ieee.org/document/9950615/
43. https://ieeexplore.ieee.org/document/9353715/
44. https://dl.acm.org/doi/10.1145/3589335.3641256

45. https://link.springer.com/10.1007/s10845-020-01590-1
46. https://arxiv.org/pdf/2401.14963.pdf
47. https://arxiv.org/html/2309.06223v3
48. http://arxiv.org/pdf/2401.05753.pdf
49. https://arxiv.org/html/2405.03587v1
50. https://royalsocietypublishing.org/doi/pdf/10.1098/rspa.2017.0117
51. http://arxiv.org/pdf/2401.05859.pdf
52. https://algocademy.com/blog/approaching-bit-manipulation-problems-a-comprehensive-guide-for-coding-interviews/
53. https://github.com/Devinterview-io/bit-manipulation-interview-questions
54. https://www.geeksforgeeks.org/competitive-programming/bit-manipulation-for-competitive-programming/
55. https://ieeexplore.ieee.org/document/11093491/
56. https://www.geeksforgeeks.org/dsa/top-problems-on-bit-manipulation-for-interviews/
57. https://www.youtube.com/watch?v=JcHiYWeLJdE
58. https://www.geeksforgeeks.org/interview-prep/commonly-asked-data-structure-interview-questions-on-bit-manipulation/
59. https://utkarsh1504.github.io/DSA-Java/bitwise-operator/
60. https://leetcode.com/discuss/post/3695233/all-types-of-patterns-for-bits-manipulat-qezp/
61. https://www.interviewbit.com/courses/programming/bit-manipulation/
62. https://www.reddit.com/r/leetcode/comments/tcyyxt/how_often_is_bit_manipulation_question_asked/
63. https://devinterview.io/blog/bit-manipulation-interview-questions/
64. https://www.codeintuition.io/learning-paths/algorithms/bit-manipulation
65. https://leetcode.com/problem-list/bit-manipulation/
66. http://ieeexplore.ieee.org/document/7586576/
67. https://www.youtube.com/watch?v=dhwPOn84DGg
68. https://ijsrem.com/download/a-review-paper-of-least-significant-bit-lsb-based-steganography-with-encryption/
69. https://ieeexplore.ieee.org/document/11022943/
70. https://peninsula-press.ae/Journals/index.php/SHIFRA/article/view/100
71. https://www.semanticscholar.org/paper/e9f8d67c276124dd4880a02a9e0193bd9ba4ad69
72. https://iopscience.iop.org/article/10.1149/10701.8311ecst
73. https://www.semanticscholar.org/paper/a74175a6350183a570baafe287bce4714ca40be6
74. https://www.semanticscholar.org/paper/c496a1767704d28b759e5af1a1e49b6b3a83ade9
75. http://ieeexplore.ieee.org/document/6409081/
76. http://helix.dnares.in/wp-content/uploads/2019/10/5269-5274.pdf
77. http://www.tandfonline.com/doi/abs/10.1080/10637190310001633664
78. https://arxiv.org/pdf/2402.15924.pdf
79. http://arxiv.org/pdf/2407.01549.pdf

80. https://arxiv.org/abs/1905.06845
81. https://arxiv.org/html/2404.12011v1
82. https://arxiv.org/pdf/2005.07336.pdf
83. https://pmc.ncbi.nlm.nih.gov/articles/PMC10912654/
84. http://arxiv.org/pdf/2409.05227.pdf
85. https://arxiv.org/pdf/1103.0801.pdf
86. https://en.wikipedia.org/wiki/Trie
87. https://www.geeksforgeeks.org/dsa/backtracking-to-find-all-subsets/
88. http://ejournal.uin-suka.ac.id/saintek/ijid/article/view/1952
89. https://www.geeksforgeeks.org/dsa/introduction-to-trie-data-structure-and-algorithm-tutorials/
90. https://www.geeksforgeeks.org/dsa/trie-insert-and-search/
91. https://codeanddebug.in/blog/generate-all-subsets-using-bit-manipulation-leetcode-78/
92. https://blog.bitsrc.io/advanced-data-structures-and-algorithms-tries-47db931e20e
93. https://www.geeksforgeeks.org/dsa/find-distinct-subsets-given-set/
94. https://takeuforward.org/trie/bit-prerequisites-for-trie-problems
95. https://www.topcoder.com/thrive/articles/print-all-subset-for-set-backtracking-and-bitmasking-approach
96. https://www.interviewcake.com/concept/java/trie
97. https://takeuforward.org/bit-manipulation/power-set-bit-manipulation