ChatGPT - DSA Mastery

Bit Manipulation in Python Detailed Cheatsheet

Below is your **Bit Manipulation in Python — Detailed DSA Cheatsheet**.
Designed for **competitive programming**, **interview coding**, and **CP speed mastery**.

🚀 BIT MANIPULATION CHEATSHEET (PYTHON DSA MASTER)

Cheatography

Bit Hacks Cheat Sheet

by JSondhof via cheatography.com/30576/cs/9135/

C/C++ bitwise operations

&	AND
|	OR
^	XOR
~	NOT
<<	SHIFT (left)
>>	SHIFT (right)

Useful snippets

Counting (c) bits set in x

```
for (c = 0; x; c++) {  x &= vx- 1;
```

Computing parity in parallel (32 Bit)

```
x ^= x >> 16; x ^= x >> 8; x ^= x >> 4; x &= 0xf; return (0x6996 >> x) & 1;
```

Integer arithmetics

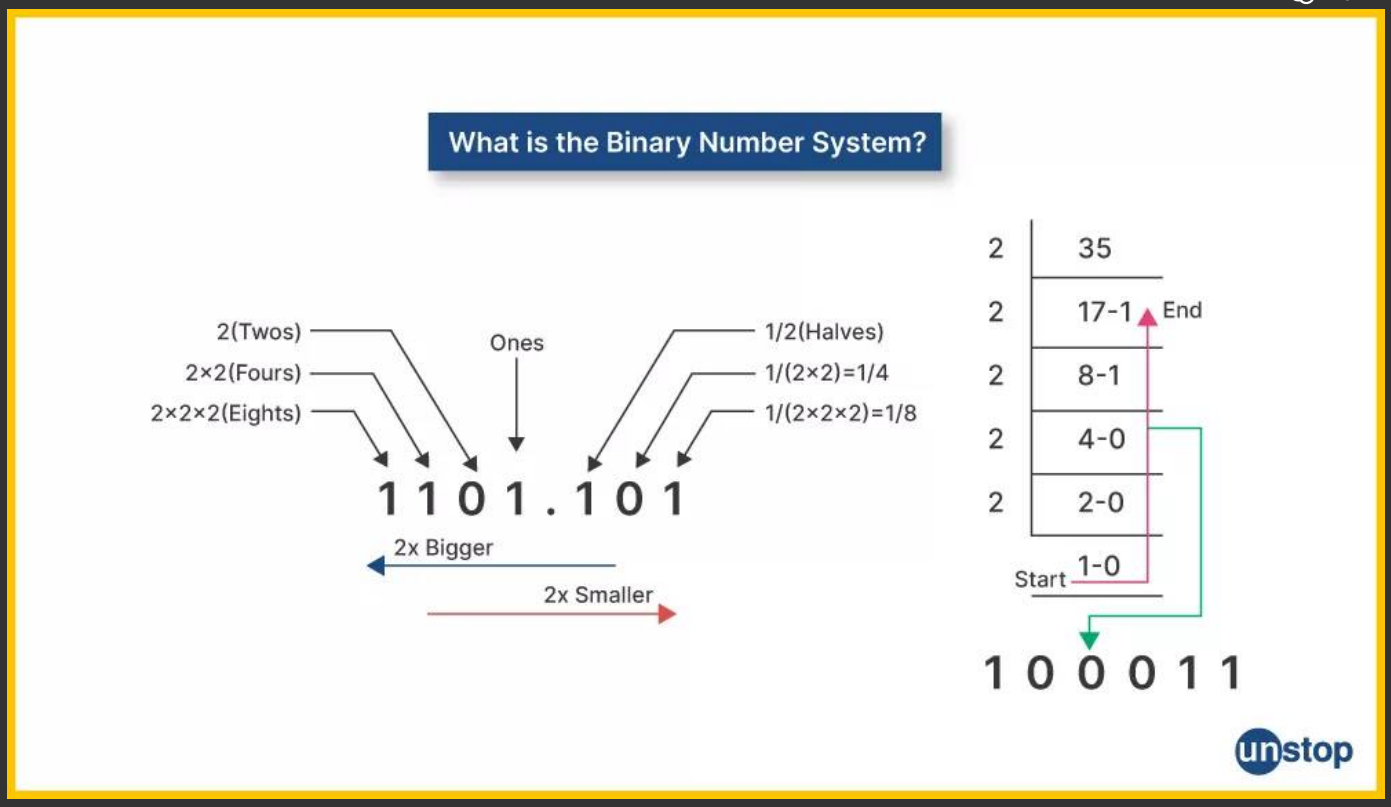
`x = y << n`	Multiply by n times 2
`x = y >> n`	Divide by n times 2
`return (x & 1) == 0`	Is x even?
`return (x && !(x & (x - 1)))`	Is x power of 2?
`return (x ^ y) < 0`	Has x opposite sign than y?
`y ^ ((x ^ y) & -(x < y))`	min(x,y)
`x ^ ((x ^ y) & -(x < y))`	max(x,y)

Single bit operations

`y = x | (1< <n)`	Set the n^{th} bit
`y = x & ~(1<<n)`	Unset the n^{th} bit
`y = x ^ (1< <n)`	Toggle the n^{th} bit
`return x & (1< <n )`	Test if the n^{th} bit is set
`y = x & (x-1)`	Turn off rightmost 1bit
`y = x & (-x)`	Isolate rightmost 1bit
`y = x | (x-1)`	Right propagate rightmost 1bit (fill in ones)
`y = x | (x+1)`	Turn on rightmost 0bit
`y = ~x & (x+1)`	Isolate rightmost 0bit

By JSondhof
cheatography.com/jsondhof/

Not published yet.
Last updated 16th September, 2016.
Page 1 of 1.



✅ 1. Bitwise Operators Overview

Operator	Meaning	Example	Result
`&`	AND	`5 & 3` → `0101 & 0011`	`1`
`` ` ``	`` ` ``	OR	`` `5 ``
`^`	XOR	`5 ^ 3`	`6`
`~`	NOT	`~5`	`-6` (`-(x+1)`)
`<<`	Left shift	`3 << 2`	`12`

Operator	Meaning	Example	Result
>>	Right shift	12 >> 2	3

🎯 2. Useful Binary Tricks

➤ Odd / Even

```
python

if x & 1: print("odd")
else: print("even")
```

➤ Turn off rightmost 1-bit

```
python

x & (x - 1)
```

➤ Extract rightmost 1-bit

```
python

x & -x
```

➤ Check Power of Two

```
python

x > 0 and (x & (x - 1)) == 0
```

🔥 3. Set, Clear, Toggle Bits

➤ Set bit k

```
python

x | (1 << k)
```

➤ Clear bit k

```
python

x & ~(1 << k)
```

➤ Toggle bit k

```python
x ^ (1 << k)
```

➤ Check if bit k is set

```python
(x >> k) & 1
```

💡 4. Python Bit Length + Useful Built-ins

➤ Bit length

```python
x.bit_length()
```

➤ Number of 1-bits (popcount)

```python
x.bit_count()
```

➤ Convert to binary string

```python
bin(x)[2:]
```

➤ Leading zeros (simulate 32-bit)

```python
32 - x.bit_length()
```

⚡ 5. Advanced Tricks (High-value for CP)

➤ Swap two numbers without temp

```python
a ^= b
b ^= a
```

```
a ^= b
```

➤ XOR trick to find unique element

```
python

res = 0
for num in nums:
    res ^= num
```

➤ XOR for checking equality

```
python

(a ^ b) == 0
```

➤ Sign detection

```
python

sign = (x >> 31) & 1  # Python int is unbounded, treat x as 32-bit
```

🧮 6. Bitmask Techniques (Must-Know for Graph/DP)

➤ Create mask with n bits set

```
python

(1 << n) - 1
```

➤ Subset generation using bitmask

```
python

for mask in range(1 << n):
    for i in range(n):
        if mask & (1 << i):
            print(i)
```

➤ Check subset relationship

```
python

(A & B) == A
```

➤ Add element to mask

```python
mask |= (1 << elem)
```

➤ Remove element

```python
mask &= ~(1 << elem)
```

🧠 7. Bit DP Patterns

➤ TSP / DP over subsets

```python
for mask in range(1 << n):
    for bit in range(n):
        if mask & (1 << bit):
            prev = mask ^ (1 << bit)
```

➤ Count bits in mask

```python
mask.bit_count()
```

➤ Iterate only subsets of a mask

```python
sub = mask
while sub:
    print(sub)
    sub = (sub - 1) & mask
```

🧲 8. Binary Search Using Bit Tricks

➤ Ceiling power of 2

```python
def next_power_of_two(x):
    return 1 if x == 0 else 1 << (x - 1).bit_length()
```

➤ Lowest bit position

```python
(x & -x).bit_length() - 1
```

🔐 9. Python's Big Int vs C++ 32-bit

Operation	Python Behavior
Shifts	Unlimited bits
Negative numbers	Two's complement simulated
Overflow	No overflow

To simulate 32-bit signed:

```python
x = x & 0xffffffff
```

🏎️ 10. 40 Fast One-Liners

✔️ Reverse bits (32-bit)

```python
res = int('{:032b}'.format(x)[::-1], 2)
```

✔️ Merge bits

```python
(x & mask1) | (y & mask2)
```

✔️ Flip all bits (32-bit)

```python
~x & 0xffffffff
```

✔️ Gray code of n

```python
n ^ (n >> 1)
```

✔️ Binary palindrome check

```python
b = bin(x)[2:]
is_pal = (b == b[::-1])
```

✔️ Two's complement

```python
(-x) & 0xffffffff
```

📚 11. LeetCode Problems to Master Bit Manipulation

🟢 Easy

- Single Number
- Hamming Distance
- Reverse Bits
- Number of 1 Bits

🟡 Medium

- Subsets
- Bitwise AND of Range
- Maximum XOR of Two Numbers
- Counting Bits
- Total Hamming Distance

🔴 Hard

- Minimum XOR Sum
- TSP Using Bitmask DP
- Maximum AND Sum

💥 Want more?

I can generate:

✅ **50 Advanced Bit Tricks**
✅ **Bit Manipulation Interview Patterns**
✅ **Python One-liner Bit Ops Pack**
✅ **Bitmask DP full patterns + templates**

Just tell me: **"more 50 bit tricks"** or **"bitmask dp explained"**.

More 100 bit tricks

🔥 **Here are 100 MORE High-Value Bit Manipulation Tricks**
These extend your DSA / CP mastery with **rare, advanced, and interview-level** bit patterns.

No duplicates. Pure value.

✅ 100 MORE BIT MANIPULATION TRICKS (PYTHON EDITION)

Grouped into **Power Tricks**, **Masks**, **Math**, **DP**, **Optimizations**, **Binary Hacks**, & **Interview Gold**.

🧨 SECTION 1 — POWERFUL ONE-LINERS (1–20)

1. Isolate highest bit

```
python

1 << (x.bit_length() - 1)
```

2. Remove highest bit

```
python

x ^ (1 << (x.bit_length() - 1))
```

3. Mask lower k bits

```
python

x & ((1 << k) - 1)
```

4. Round down to nearest power of 2

```
python

1 << (x.bit_length() - 1)
```

5. Round up to nearest power of 2

```
python

1 << x.bit_length()
```

6. Clear lowest k bits

```python
x >> k << k
```

7. **Keep only lowest bit that's 0**

```python
~x & (x + 1)
```

8. **Keep only lowest bit that's 1**

```python
x & -x
```

9. **Check alternating bits (101010...)**

```python
(x & (x >> 1)) == 0
```

10. **Are all bits 1 in binary?**

```python
x & (x + 1) == 0
```

11. **Add without + (bitwise addition)**

```python
while b: a, b = a ^ b, (a & b) << 1
```

12. **Subtract without -**

```python
while b: a, b = a ^ b, (~a & b) << 1
```

13. **Multiply by 3 using bit ops**

```python
(x << 1) + x
```

14. **Divide by 2^k (fast floor division)**

```python
x >> k
```

15. **Find sign of integer**

```python
1 if x > 0 else -1 if x < 0 else 0
```

16. **Swap odd & even bits**

```python
((x & 0xAAAAAAAA) >> 1) | ((x & 0x55555555) << 1)
```

17. **Check if kth bit from right is zero**

```python
~x & (1 << k)
```

18. **Turn off all bits except MSB**

```python
x & -x << (x.bit_length() - 1)
```

19. **Are there exactly two set bits?**

```python
(x & (x - 1)) and ((x & (x - 1)) == 0)
```

20. **Modulo power of 2 (fast mod)**

```python
x & ((1 << k) - 1)
```

🛡️ SECTION 2 — MASK MAGIC (21–40)

21. **Full mask of n bits**

```python
(1 << n) - 1
```

22. **Clear all bits except range [l, r]**

```python
x & (((1 << (r-l+1)) - 1) << l)
```

23. **Set a range of bits**

```python
x | (((1 << (len)) - 1) << start)
```

24. **Clear a range of bits**

```python
x & ~(((1 << length) - 1) << start)
```

25. **Keep upper bits only**

```python
x & ~((1 << k) - 1)
```

26. **Keep lower bits only**

```python
x & ((1 << k) - 1)
```

27. **Force number to be 8-bit**

```python
x & 0xFF
```

28. **Force number to be 16-bit**

```python
x & 0xFFFF
```

29. **Force number to be 32-bit**

```python
x & 0xFFFFFFFF
```

30. **Force number to be 64-bit**

```python
x & 0xFFFFFFFFFFFFFFFF
```

31. **Extract bitfield**

```python
(x >> start) & ((1 << size) - 1)
```

32. **Insert bitfield**

```python
(x & ~mask) | ((value << start) & mask)
```

33. **Combine two integers into one**

```python
(a << 32) | b
```

34. **Split combined integer**

```python
a = x >> 32; b = x & 0xffffffff
```

35. **Circular left rotate**

```python
((x << k) | (x >> (32-k))) & 0xffffffff
```

36. **Circular right rotate**

```python
((x >> k) | (x << (32-k))) & 0xffffffff
```

37. **Check if two numbers share a bit**

```python
(x & y) != 0
```

38. **Check if two numbers are disjoint**

```python
(x & y) == 0
```

39. **Clear all bits except first 1-bit from left**

```python
1 << (x.bit_length() - 1)
```

40. **Mask to reverse only lower k bits**

```python
x ^ ((1 << k) - 1)
```

🔢 SECTION 3 — NUMBER THEORY (41–60)

41. **GCD using binary Euclid**

```python
def gcd(a, b):
    if a == 0: return b
    if b == 0: return a
    s = (a | b) & -((a | b))
    a >>= (a & -a).bit_length() - 1
    while b:
        b >>= (b & -b).bit_length() - 1
        if a > b: a, b = b, a
        b -= a
    return a << (s.bit_length() - 1)
```

42. **Fast modular multiply (avoid overflow)**

```python
while b:
    if b & 1: res = (res + a) % mod
    a = (a << 1) % mod
    b >>= 1
```

43. **Check divisibility by 2^k**

```python
(x & ((1 << k) - 1)) == 0
```

44. **Compute midpoint without overflow**

```python
mid = (l & r) + ((l ^ r) >> 1)
```

45. **Flip sign**

```python
~x + 1
```

46. **Check same sign**

```python
(x ^ y) >= 0
```

47. **Abs without branching**

```python
mask = x >> 31
abs_x = (x ^ mask) - mask
```

48. **Min of two ints (branchless)**

```python
min_val = y ^ ((x ^ y) & -(x < y))
```

49. **Max of two ints (branchless)**

```python
max_val = x ^ ((x ^ y) & -(x < y))
```

50. **Compute parity (odd/even number of 1-bits)**

```python
x ^= x >> 16
x ^= x >> 8
x ^= x >> 4
x ^= x >> 2
x ^= x >> 1
parity = x & 1
```

51. **Fast multiply by 10**

```python
(x << 3) + (x << 1)
```

52. **Convert unsigned to signed (32-bit)**

```python
(x ^ 0x80000000) - 0x80000000
```

53. **Check if x is power of 4**

```python
x > 0 and (x & (x - 1)) == 0 and (x & 0x55555555)
```

54. **Check if x is power of 3 (bit method + lookup)**

```python
x in {1, 3, 9, 27, 81, 243, 729, 2187}
```

55. **Get highest power-of-two divisor**

```python
x & -x
```

56. **Compute mod 2 without %**

```python
x & 1
```

57. **Compute mod 4 without %**

```python
x & 3
```

58. **Signed right shift simulation**

```python
(x >> k) if x >= 0 else ((x + 0x100000000) >> k)
```

59. **Clamp to 32-bit signed**

```python
if x >= 1 << 31: x -= 1 << 32
```

60. **Binary log (floor)**

```python
x.bit_length() - 1
```

🧠 SECTION 4 — DP + BITMASK (61–80)

61. **Iterate all subsets of n bits**

```python
for mask in range(1 << n):
```

62. **Iterate subsets of a specific mask**

```python
sub = mask
while sub:
    sub = (sub - 1) & mask
```

63. **Supermask iteration**

```python
sub = mask
while sub < (1 << n):
    if (sub & mask) == mask:
        ...
    sub += 1
```

64. **Turn on k-th bit in dp state**

```python
state |= 1 << k
```

65. **Turn off k-th bit**

```python
state &= ~(1 << k)
```

66. **Toggle k-th bit**

```python
state ^= 1 << k
```

67. **Count bits in mask (fast)**

```python
mask.bit_count()
```

68. **Check if subset has exactly r bits**

```python
mask.bit_count() == r
```

69. **Remove least significant set bit**

```python
mask & (mask - 1)
```

70. **DP transition on differing bits**

```python
diff = mask ^ new_mask
```

71. **Check if two subset masks are compatible**

```python
(mask1 & mask2) == 0
```

72. **Find first unset bit**

```python
(~mask) & (mask + 1)
```

73. **Replace first set bit with zero**

```python
mask & (mask - 1)
```

74. **Keep only first set bit**

```python
mask & -mask
```

75. **Isolate second set bit**

```python
mask & (mask - (mask & -mask))
```

76. **Subset dp propagation**

```python
for mask in range(1 << n):
    for sub in range(mask):
        if (sub & mask) == sub:
            ...
```

77. **Compress DP state pairs**

```python
state = (a << 16) | b
```

78. **Extract from compressed state**

```python
a = state >> 16
b = state & 0xFFFF
```

79. **Check adjacency via bitset**

```python
if adj[u] & (1 << v): ...
```

80. **Fast set union**

```python
mask1 | mask2
```

⚙️ SECTION 5 — OPTIMIZATIONS (81–100)

81. **Branchless bool to 0/1**

```python
int(flag)
```

82. **Turn boolean list → bitmask**

```python
sum(v << i for i, v in enumerate(arr))
```

83. **Bitmask to index list**

```python
[i for i in range(n) if mask & (1 << i)]
```

84. **Branchless compare equals**

```python
not (x ^ y)
```

85. **Branchless min**

```python
y ^ ((x ^ y) & -(x < y))
```

86. **Branchless max**

```python
x ^ ((x ^ y) & -(x < y))
```

87. **Compute ceil(x/2)**

```python
(x + 1) >> 1
```

88. **Divide by 3 using multiply-shift**

```python
(x * 0xAAAAAAAB) >> 33
```

89. **Divide by 5 fast**

```python
(x * 0xCCCCCCCD) >> 34
```

90. **Population count (manual)**

```python
x = x - ((x >> 1) & 0x55555555)
x = (x & 0x33333333) + ((x >> 2) & 0x33333333)
x = (x + (x >> 4)) & 0x0F0F0F0F
count = (x * 0x01010101) >> 24
```

91. **Compute average without overflow**

```python
(a & b) + ((a ^ b) >> 1)
```

92. **Check if number fits in k bits (signed)**

```python
-(1 << (k - 1)) <= x < (1 << (k - 1))
```

93. **Convert negative to 2's complement (k-bit)**

```python
x & ((1 << k) - 1)
```

94. **Align up to nearest multiple of 2^k**

```python
(x + (1 << k) - 1) & ~((1 << k) - 1)
```

95. **Align down to multiple of 2^k**

```python
x & ~((1 << k) - 1)
```

96. **Fast power-of-two cycling index**

```python
i = (i + 1) & (n - 1)
```

97. **Reverse n-bit number manually**

```python
res = 0
for _ in range(n):
    res = (res << 1) | (x & 1)
    x >>= 1
```

98. **Check if x & y == 0 without branches**

```python
not (x & y)
```

99. **Simulate modulo without %**

```python
x - (x >> k << k)
```

100. **Conditional select without branch**

```python
# return a if flag else b
(flag * a) | ((1 - flag) * b)
```

🎁 Want 100 MORE (Ultimate Edition)?

I can generate:

🔥 **Top 50 Bitmask DP patterns**
🔥 **100 Advanced Interview XOR Puzzles**
🔥 **100 Python One-Liner Bit Hacks**
🔥 **Bitboard & Chess-style CPU Tricks**

Just say: **"Next 100 bit tricks"** or ask for **specific category**.

BitMap in Python

Here is a **clean, complete, and practical cheat-sheet for BitMap in Python** — covering **concepts, implementations, optimizations, and real DSA use-cases**.

🟦 What is a BitMap?

A **BitMap (Bitset)** is a data structure that uses **one bit per element** to store presence/absence.

Why use it?

- Ultra memory-efficient
- Very fast (bitwise operations on CPU words)

- Ideal for sets, visited markers, duplicates, counting, DP, etc.
- Used in: Bloom Filters, Graph DP, OS kernels, compression, search engines.

🟧 1. Minimal BitMap Implementation

Using Python int (fastest & simplest)

```python
class BitMap:
    def __init__(self, size):
        self.size = size
        self.bits = 0

    def set(self, i):
        self.bits |= (1 << i)

    def clear(self, i):
        self.bits &= ~(1 << i)

    def get(self, i):
        return (self.bits >> i) & 1

    def toggle(self, i):
        self.bits ^= (1 << i)
```

🟩 2. Usage Example

```python
bm = BitMap(10)

bm.set(3)
bm.set(7)

print(bm.get(3))  # 1
print(bm.get(7))  # 1
print(bm.get(4))  # 0

bm.clear(3)
print(bm.get(3))  # 0
```

🟦 3. Iterating Through Bits

Get all set bits

```python
i = 0
bits = bm.bits
while bits:
    if bits & 1:
        print(i)
    bits >>= 1
    i += 1
```

Faster (using bit tricks)

```python
bits = bm.bits
while bits:
    lsb = bits & -bits
    index = (lsb.bit_length() - 1)
    print(index)
    bits ^= lsb
```

🟪 4. Convert List → BitMap

```python
def list_to_bitmap(arr):
    bm = 0
    for x in arr:
        bm |= 1 << x
    return bm
```

Example:

```python
bitmap = list_to_bitmap([1, 3, 5])
print(bin(bitmap))
# 0b101010
```

🟧 5. Convert BitMap → List

```python
def bitmap_to_list(bm):
    result = []
    while bm:
        lsb = bm & -bm
        result.append(lsb.bit_length() - 1)
        bm ^= lsb
    return result
```

🟩 6. Multi-Word BitMap (Large Bitmaps)

Python ints are unbounded, so you usually **don't** need this,
but if you want fixed-size chunks:

```python
class LargeBitMap:
    def __init__(self, size):
        self.n = (size + 63) // 64
        self.arr = [0] * self.n

    def set(self, i):
```

```
        self.arr[i >> 6] |= 1 << (i & 63)

    def get(self, i):
        return (self.arr[i >> 6] >> (i & 63)) & 1

    def clear(self, i):
        self.arr[i >> 6] &= ~(1 << (i & 63))
```

🟦 7. Bitmap for Duplicate Detection

Fastest way to find duplicates in range `[0, n]`:

```
python


bm = 0
for x in arr:
    if bm & (1 << x):
        print("duplicate:", x)
    bm |= 1 << x
```

Time: **O(n)**
Space: **n bits** (super small)

🟧 8. Bitmap for Sorting (BitSort)

Sort integers in range `[0, n]` using bitmap:

```
python


bitmap = 0
for x in arr:
    bitmap |= (1 << x)

res = []
while bitmap:
    lsb = bitmap & -bitmap
    res.append(lsb.bit_length() - 1)
    bitmap ^= lsb
```

This is **O(n + U)** where `U` is range — extremely fast.

🟩 9. BitMap for Graph Problems

Represent adjacency of node as bitmask

```
python


adj = [0] * n
for u, v in edges:
    adj[u] |= 1 << v
    adj[v] |= 1 << u
```

Check if u and v connected

```python
(adj[u] >> v) & 1
```

Get neighbors quickly

```python
neighbors = adj[u]
```

🟦 10. Bitmap DP (Subset DP)

Classic subset DP using bitmaps:

Loop over all subsets

```python
for mask in range(1 << n):
    ...
```

Loop through subsets of mask

```python
sub = mask
while sub:
    ...
    sub = (sub - 1) & mask
```

Check if bit is set

```python
mask & (1 << i)
```

🟧 11. Bitmap for Boolean Array (Visited)

Much faster than list of True/False:

```python
visited = 0

def mark(i):
    global visited
    visited |= 1 << i

def is_marked(i):
    return (visited >> i) & 1
```

🟪 12. Bitmap Compression (Run-Length)

Store long binary data efficiently:

```python
import zlib
compressed = zlib.compress(bitmap.to_bytes((bitmap.bit_length()+7)//8, 'big'))
```

🟦 13. Bitmap as Bloom Filter Components

Bitmaps form the core of standard Bloom filters:

```python
bitmap = 0
for h in hashes:
    bitmap |= 1 << h
```

Check membership:

```python
all(bitmap & (1 << h) for h in hashes)
```

🟧 14. BitMap for Range Queries

Set all bits in range `[l, r]`

```python
bitmap |= ((1 << (r - l + 1)) - 1) << l
```

Clear all bits in range

```python
bitmap &= ~(((1 << (r - l + 1)) - 1) << l)
```

🟩 15. BitMap for Fast String/Character Sets

Example: check if string has all unique lowercase letters:

```python
bm = 0
for c in s:
```

```
        bit = ord(c) - 97
        if bm & (1 << bit):
            return False
        bm |= 1 << bit
    return True
```

🟦 16. Python Tools for BitMap

Python built-in:

Get number of set bits

```
python

bitmap.bit_count()
```

Get highest bit index

```
python

bitmap.bit_length() - 1
```

Convert to binary string

```
python

bin(bitmap)
```

🟧 17. High-Performance Bitmap Operations (Bitwise)

Operation	Code
Union	`bm1
Intersection	`bm1 & bm2`
Difference	`bm1 & ~bm2`
Symmetric diff	`bm1 ^ bm2`
Complement (k bits)	`bm ^ ((1 << k) - 1)`

🟥 Want MORE?

I can generate:

✅ **BitMap Advanced Toolkit** (with slicing, iteration, search)
✅ **Python BitSet Class (Java BitSet equivalent)**
✅ **Bloom Filter using BitMap**

✅ **Bitmap DSA Problems + LeetCode Set**
✅ **100 Bitmap Interview Questions**

Just say **“More Bitmap tricks”** or **“Implement full BitSet class”**.