* Given an integer value n, find the (191) "Hamming Weight" of the number n.

Cognizant | Wipro | HCL | Infosys

10 – 1010 → 2

11 → 1011 → No of 1's / No of set bits / No of 1 bits

5 – 0101 → 2

0 – 0000 → 0

int hammingWeight (int n) {

}

n = 11 → 1011
& 1 0001
0001 →

Car C++

count = 0
++ 1
++ 2
++ (3)

TCQs

1011 >> 1   101 >> 1   10 >> 1
& 001 → 001   01 → 00   1 → 1

1 >> 1 → 0 stop

1. Perform an & operation with 1
2. If true increment the count
3. Shift to the right by 1 bit
4. Stop if the value of n becomes 0
5. Return the count.

Reverse Integer: →

int n = 123
o/p = 321

123 % 10 = 3 → digit

n/10 = 123/10 = 12

12 % 10 = 2 → digit

12/10 = 1 % 10 = 1 → digit

1/10 = 0 stop (TCQ)

ans = 0
= ans×10 + digit
= 0×10 + 3
= 3

ans = ans×10 + digit
= 3×10 + 2
= 32

32×10 + digit
320 + 1
= 321

Armstrong Number: →

9474 = 4 digits

$= 9^4 + 4^4 + 7^4 + 4^4 = 9474$

(153) 3 digits

$1^3 + 5^3 + 3^3$

1 + 125 + 27 = 153

% 10   /10

Single non-repeating element

1, 4, 5, 2, 1, (3), 4, 2, 5

res = 0

res 0000 0
0001 1
0001 1
0100 4
0101 5
0101 5
0000 0
0010 2
0010 2
0001 1
0011 3

0011 3
0011 3
X
0000 0
0100 4
0100 4
0010 2
0110 6
0101 5
0011 3 = (3)

res = 3

Dry Run!

Two unique (non-repeating) elements: →

arr = { 1, 2, 3, 4, 1, 2 }

(XOR) = 3 ^ 4 = 0011 ^ 0100 = 0111 = 7.

→ n1 = (3) 0
→ n2 = (4) 0

n1 XOR { 1x, 3, 1x } | 1 | 1 | 1 |
n2 XOR { 2x, 4, 2x } 0 0 0

Right most set bit: →

rsb = XORALL & –XORALL = (3) x (4)
= 7 &

–7 0111
1's 1000
+1 0001
1001

= 0111 & 1001 → 0111 & 1001 = rsb 0001

if (arr[i] & rsb) == (1)
(0) n1 ^= arr[i]   1 ^ 3 ^ 1 = (3)
else
(0) n2 ^= arr[i]   2 ^ 4 ^ 2 = (4)

* Happy Number:

19 → $1^2 + 9^2$
1 + 81
= (82)

n=82 $8^2 + 2^2$
= 64+4
= (68)

n=68 = $6^2 + 8^2$
= 36 + 64
= 100
= $1^2 + 0^2 + 0^2$ ← (1)

Sad Number / Unhappy No

(1) (4)

$4^2 = 16$
$1^2 + 6^2$
= 37
$3^2 + 7^2$
= 9 + 49
= 58
= $5^2 + 8^2$
= 25 + 64
= 89
64 + 81
= 145

$1^2 + 4^2 + 5^2$
1 + 16 + 25
42
16 + 4
= 20
4 + 0
= (4)

Important Algorithm for Placements: [O(n)]

Kadane's Algorithm: →

array = { 5, –8, 1, 2, –1, 4 } (indices i 0 1 2 3 4 5)

Maximum Subarray Sum: →

```
int cmax = arr[0]; 5
int gmax = arr[0]; 5
for( int i=1; i<n; i++)
    cmax = max(arr[i], cmax + arr[i]);
    gmax = max (cmax, gmax);
}
return gmax;
```

a arr[i]	b cmax + arr[i]	max a,b cmax	max cmax gmax
–8	5 + (–8) = –3	–3	5
1	1 + (–3) = –2	1	5
2	1 + 2 = 3	3	5
–1	–1 + 3 = 2	2	5
4	4 + 2 = 6	6	(6)

Arrays: → Jagged Array

swap ←
shift ←
SC O(1) ←
SC O(n) ←

Searching

* Linear Search → O(n)
* Binary Search → log(n)
* Recursive Binary Search → log(n)
* Jump Search → (√n)
* Interpolation Search
  ↳ [pos → index]

{{ ___ }, { }, { }}

* Bubble Sort n^2
* Selection Sort n^2
* Insertion Sort n^2
* Quick Sort n log n
* Merge Sort n log n
* Shell Sort → (AIML)
* Heap Sort → n log n
* NCA:
  - O(n+max) (1) Count Sort
  - O(n+max) (4) Radix Sort