

190. Reverse Bits

March 23, 2020 | 14.9K views

Average Rating: 3.43 (21 votes)

Reverse bits of a given 32 bits unsigned integer.

Example 1:

Input: 00000010100101000001111010011100
Output: 001110010111000001010010100000
Explanation: The input binary string 00000010100101000001111010011100 represents the unsigned integer 43261596, so its reverse is 989130216, and the output binary string 001110010111000001010010100000 represents the unsigned integer 989130216.

Example 2:

Input: 11111111111111111111111111111101
Output: 10111111111111111111111111111111
Explanation: The input binary string 11111111111111111111111111111101 represents the unsigned integer 1073741825, so its reverse is 34967133, and the output binary string 10111111111111111111111111111111 represents the unsigned integer 34967133.

Note:

- Note that in some languages such as Java, there is no unsigned integer type. In this case, both input and output will be given as signed integer type and should not affect your implementation, as the internal binary representation of the integer is the same whether it is signed or unsigned.
- In Java, the compiler represents the signed integers using 2's complement notation. Therefore, in Example 2 above the input represents the signed integer -3 and the output represents the signed integer -1073741825.

Follow up:

If this function is called many times, how would you optimize it?

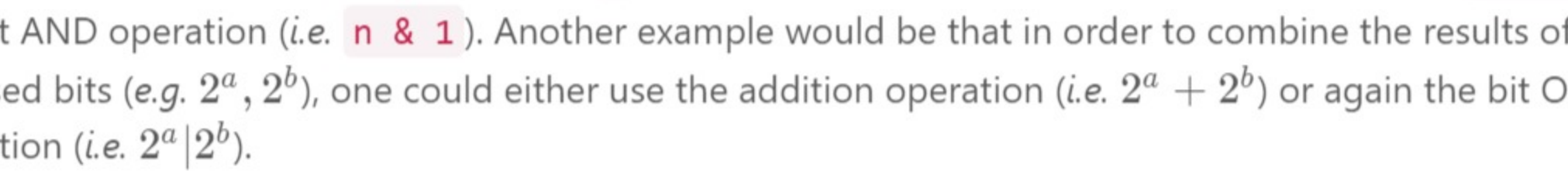
Solution

Approach 1: Bit by Bit

Intuition

Though the question is not difficult, it often serves as a warm-up question to kick off the interview. The point is to test one's basic knowledge on data type and bit operations.

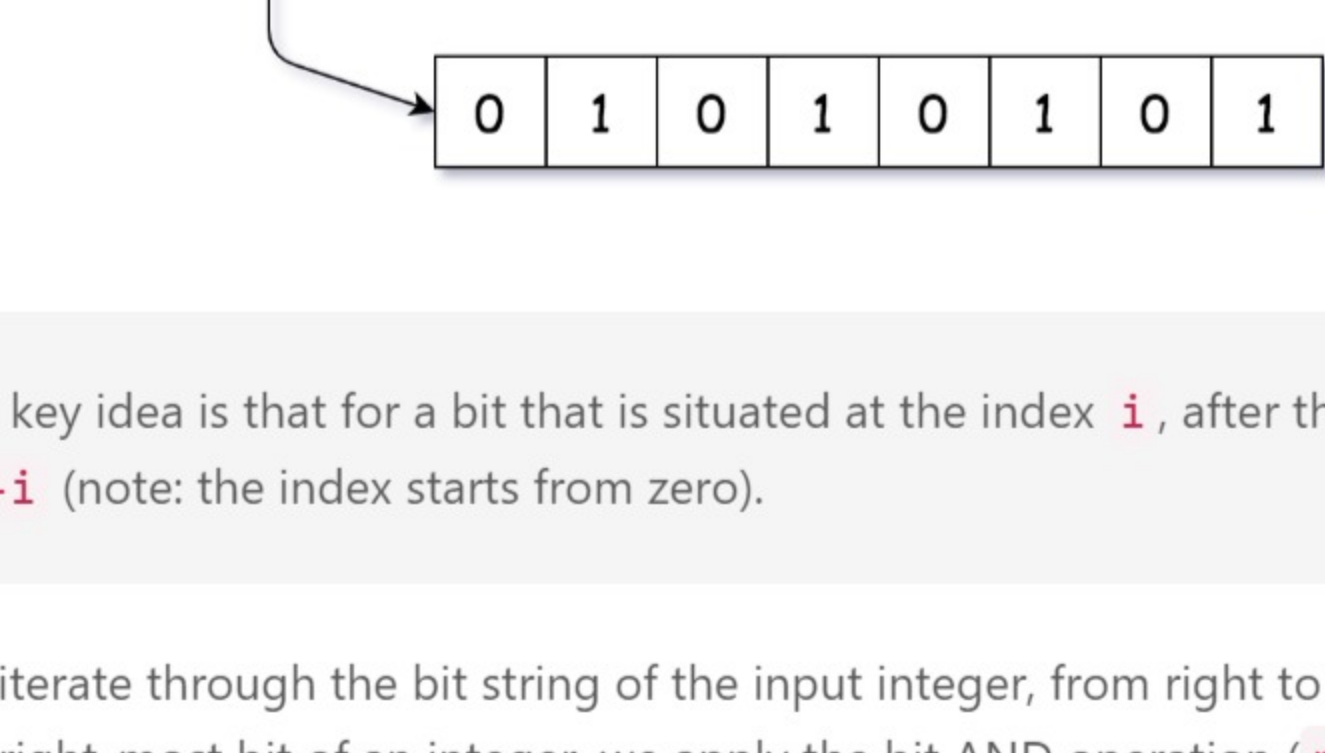
As one of the most intuitive solutions that one could come up during an interview, one could reverse the bits **one by one**.



As easy as it sounds, the above intuition could lead to quite some variants of implementation. For instance, to retrieve the *right-most* bit in an integer `n`, one could either apply the modulo operation (i.e. `n % 2`) or the bit AND operation (i.e. `n & 1`). Another example would be that in order to combine the results of reversed bits (e.g. $2^0, 2^1, \dots$), one could either use the addition operation (i.e. $2^a + 2^b$) or again the bit OR operation (i.e. $2^a | 2^b$).

Algorithm

Here we show an example of implementation based on the above intuition.



- We iterate through the bit string of the input integer, from right to left (i.e. `n = n >> 1`). To retrieve the *right-most* bit of an integer, we apply the bit AND operation (`n & 1`).
- For each bit, we reverse it to the correct position (i.e. `(n & 1) << power`). Then we accumulate this reversed bit to the final result.
- When there is no more bits of one left (i.e. `n == 0`), we terminate the iteration.

```
C++ Python Go
1 class Solution {
2 public:
3     uint32_t reverseBits(uint32_t n) {
4         uint32_t ret = 0, power = 31;
5         while (n != 0) {
6             ret += (n & 1) << power;
7             n = n >> 1;
8             power -= 1;
9         }
10        return ret;
11    }
12};
```

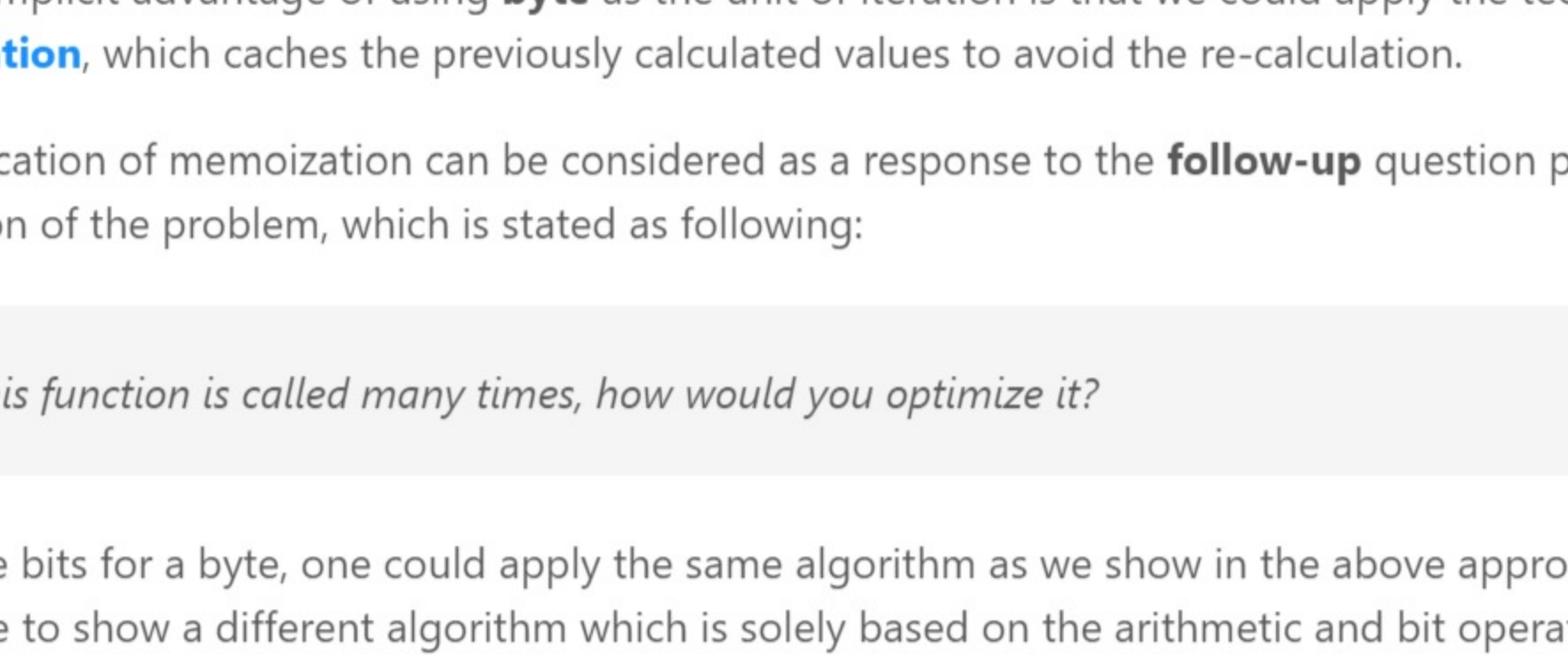
Complexity

- Time Complexity: $\mathcal{O}(1)$. Though we have a loop in the algorithm, the number of iteration is fixed regardless the input, since the integer is of fixed-size (32-bits) in our problem.
- Space Complexity: $\mathcal{O}(1)$, since the consumption of memory is constant regardless the input.

Approach 2: Byte by Byte with Memoization

Intuition

Someone might argue it might be more efficient to reverse the bits, **per byte**, which is an unit of 8 bits. Though it is not necessarily true in our case, since the input is of fixed-size 32-bit integer, it could become more advantageous when dealing with the input of long bit stream.



Another implicit advantage of using **byte** as the unit of iteration is that we could apply the technique of **memoization**, which caches the previously calculated values to avoid the re-calculation.

The application of memoization can be considered as a response to the **follow-up** question posed in the description of the problem, which is stated as following:

If this function is called many times, how would you optimize it?

To reverse bits for a byte, one could apply the same algorithm as we show in the above approach. Here we would like to show a different algorithm which is solely based on the arithmetic and bit operations without resorting to any loop statement, as following:

```
def reverseByte(byte):
    return (byte * 0x0202020202 & 0x010884422010) % 1023
```

The algorithm is documented as "reverse the bits in a byte with 3 operations" on the online book called **Bit Twiddling Hacks** by Sean Eron Anderson, where one can find more details.

Algorithm

- We iterate over the bytes of an integer. To retrieve the right-most byte in an integer, again we apply the bit AND operation (i.e. `n & 0xff`) with the bit mask of `11111111`.
- For each byte, first we reverse the bits within the byte, via a function called `reverseByte(byte)`. Then we shift the reversed bits to their final positions.
- With the function `reverseByte(byte)`, we apply the technique of memoization, which caches the result of the function and returns the result directly for the future invocations of the same input.

Note that, one could opt for a smaller unit rather than byte, e.g. a unit of 4 bits, which would require a bit more calculation in exchange of less space for cache. It goes without saying that, the technique of memoization is a trade-off between the space and the computation.

```
C++ Python Go Python3
1 import functools
2
3 class Solution:
4     @param n, an integer
5     @return an integer
6     def reverseBits(self, n):
7         ret, power = 0, 24
8         while n:
9             ret += self.reverseByte(n & 0xff) << power
10            n = n >> 8
11            power -= 8
12        return ret
13
14 # memoization with decorator
15 @functools.lru_cache(maxsize=256)
16 def reverseByte(self, byte):
17     return (byte * 0x0202020202 & 0x010884422010) % 1023
```

Complexity

- Time Complexity: $\mathcal{O}(1)$. Though we have a loop in the algorithm, the number of iteration is fixed regardless the input, since the integer is of fixed-size (32-bits) in our problem.
- Space Complexity: $\mathcal{O}(1)$. Again, though we used a cache to keep the results of reversed bytes, the total number of items in the cache is bounded to $2^8 = 256$.

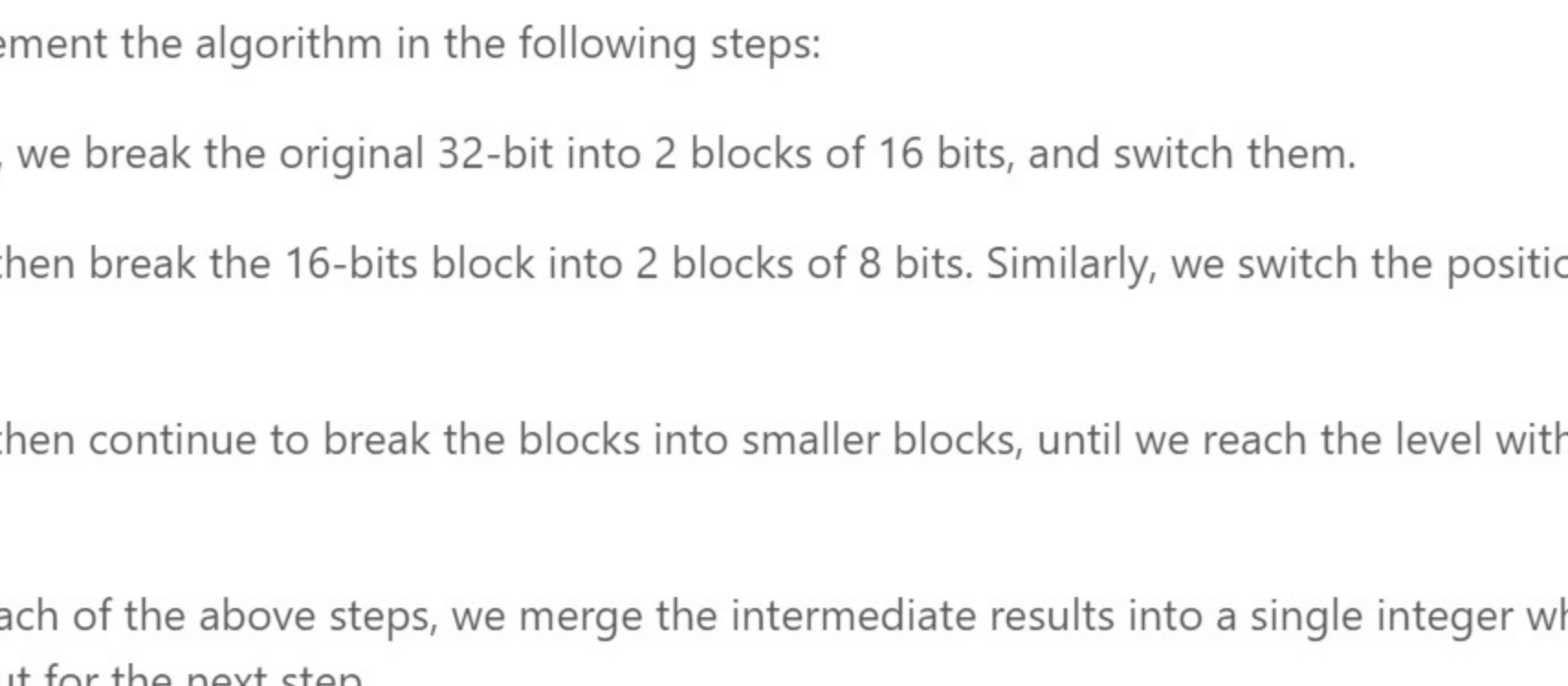
Approach 3: Mask and Shift

Intuition

We have shown in Approach #2 an example on how to reverse the bits in a byte without resorting to the loop statement. During the interview, one might be asked to reverse the entire 32 bits without using loop. Here we propose one solution that utilizes only the bit operations.

The idea can be considered as a strategy of **divide and conquer**, where we divide the original 32-bits into blocks with fewer bits via **bit masking**, then we reverse each block via **bit shifting**, and at the end we merge the result of each block to obtain the final result.

In the following graph, we demonstrate how to reverse two bits with the above-mentioned idea. As one can see, the idea could be applied to **blocks** of bits.



Algorithm

We can implement the algorithm in the following steps:

- 1). First, we break the original 32-bit into 2 blocks of 16 bits, and switch them.
- 2). We then break the 16-bits block into 2 blocks of 8 bits. Similarly, we switch the position of the 8-bits blocks.
- 3). We then continue to break the blocks into smaller blocks, until we reach the level with the block of 1 bit.
- 4). At each of the above steps, we merge the intermediate results into a single integer which serves as the input for the next step.

The credit of this solution goes to @two1er and @bhc3n for their [post and comment](#) in the discussion forum.

```
C++ Python Go
1 class Solution {
2 public:
3     uint32_t reverseBits(uint32_t n) {
4         n = (n >> 16) | (n << 16);
5         n = ((n & 0xffff0000) >> 8) | ((n & 0x0000ffff) << 8);
6         n = ((n & 0xff00ff00) >> 4) | ((n & 0x00ff00ff) << 4);
7         n = ((n & 0xcccccccc) >> 2) | ((n & 0x33333333) << 2);
8         n = ((n & 0xaaaaaaaa) >> 1) | ((n & 0x55555555) << 1);
9         return n;
10    }
11};
```

Complexity

- Time Complexity: $\mathcal{O}(1)$, no loop is used in the algorithm.
- Space Complexity: $\mathcal{O}(1)$. Actually, we did not even create any new variable in the function.

Rate this article: ★★★★★

Previous Next

Comments: 16

Sort By

Type comment here... (Markdown is supported)

Preview Post

jinsiang ★25 May 27, 2020 1:12 PM
It is a good question, and I don't think it is a easy level problem, although the code is short.

munkhbat ★22 May 22, 2020 10:59 PM
java version.

```
public class Solution {
    public int reverseBits(int n) {
        int ans = 0;
    }
}
```

10 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207 208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223 224 225 226 227 228 229 230 231 232 233 234 235 236 237 238 239 240 241 242 243 244 245 246 247 248 249 250 251 252 253 254 255 256 257 258 259 260 261 262 263 264 265 266 267 268 269 270 271 272 273 274 275 276 277 278 279 280 281 282 283 284 285 286 287 288 289 290 291 292 293 294 295 296 297 298 299 300 301 302 303 304 305 306 307 308 309 310 311 312 313 314 315 316 317 318 319 320 321 322 323 324 325 326 327 328 329 330 331 332 333 334 335 336 337 338 339 340 341 342 343 344 345 346 347 348 349 350 351 352 353 354 355 356 357 358 359 360 361 362 363 364 365 366 367 368 369 370 371 372 373 374 375 376 377 378 379 380 381 382 383 384 385 386 387 388 389 390 391 392 393 394 395 396 397 398 399 400 401 402 403 404 405 406 407 408 409 410 411 412 413 414 415 416 417 418 419 420 421 422 423 424 425 426 427 428 429 430 431 432 433 434 435 436 437 438 439 440 441 442 443 444 445 446 447 448 449 450 451 452 453 454 455 456 457 458 459 460 461 462 463 464 465 466 467 468 469 470 471 472 473 474 475 476 477 478 479 480 481 482 483 484 485 486 487 488 489 490 491 492 493 494 495 496 497 498 499 500 501 502 503 504 505 506 507 508 509 510 511 512 513 514 515 516 517 518 519 520 521 522 523 524 525 526 527 528 529 530 531 532 533 534 535 536 537 538 539 540 541 542 543 544 545 546 547 548 549 550 551 552 553 554 555 556 557 558 559 560 561 562 563 564 565 566 567 568 569 570 571 572 573 574 575 576 577 578 579 580 581 582 583 584 585 586 587 588 589 590 591 592 593 594 595 596 597 598 599 600 601 602 603 604 605 606 607 608 609 610 611 612 613 614 615 616 617 618 619 620 621 622 623 624 625 626 627 628 629 630 631 632 633 634 635 636 637 638 639 640 641 642 643 644 645 646 647 648 649 650 651 652 653 654 655 656 657 658 659 660 661 662 663 664 665 666 667 668 669 670 671 672 673 674 675 676 677 678 679 680 681 682 683 684 685 686 687 688 689 690 691 692 693 694 695 696 697 698 699 700 701 702 703 704 705 706 707 708 709 710 711 712 713 714 715 716 717 718 719 720 721 722 723 724 725 726 727 728 729 730 731 732 733 734 735 736 737 738 739 740 741 742 743 744 745 746 747 748 749 750 751 752 753 754 755 756 757 758 759 760 761 762 763 764 765 766 767 768 769 770 771 772 773 774 775 776 777 778 779 780 781 782 783 784 785 786 787 788 789 790 791 792 793 794 795 796 797 798 799 800 801 802 803 804 805 806 807 808 809 810 811 812 813 814 815 816 817 818 819 820 821 822 823 824 825 826 827 828 829 830 831 832 833 834 835 836 837 838 839 840 841 842 843 844 845 846 847 848 849 850 851 852 853 854 855 856 857 858 859 860 861 862 863 864 865 866 867 868 869 870 871 872 873 874 875 876 877 878 879 880 881 882 883 884 885 886 887 888 889 890 891 892 893 894 895 896 897 898 899 900 901 902 903 904 905 906 907 908 909 910 911 912 913 914 915 916 917 918 919 920 921 922 923 924 925 926 927 928 929 930 931 932 933 934 935 936 937 938 939 940 941 942 943 944 945 946 947 948 949 950 951 952 953 954 955 956 957 958 959 960 961 962 963 964 965 966 967 968 969 970 971 972 973 974 975 976 977 978 979 980 981 982 983 984 985 986 987 988 989 990 991 992 993 994 995 996 997 998 999 1000

Opimenov ★24 April 27, 2020 5:05 AM
to get the bit value bit AND has to be used. In your explanation you state that n | 1 will give you the value of the right most bit. It is incorrect. Please correct, so people don't learn the wrong way, as anything OR with 1 will always give you 1.

ShaneTsui ★109 April 17, 2020 9:00 AM
Mask & shift, but simpler.

```
class Solution:
    def reverseBits(self, n: int) -> int:
        mask = 1
```

4 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207 208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223 224 225 226 227 228 229 230 231 232 233 234 235 236 237 238 239 240 241 242 243 244 245 246 247 248 249 250 251 252 253 254 255 256 257 258 259 260 261 262 263 264 265 266 267 268 269 270 271 272 273 274 275 276 277 278 279 280 281 282 283 284 285 286 287 288 289 290 291 292 293 294 295 296 297 298 299 300 301 302 303 304 305 306 307 308 309 310 311 312 313 314 315 316 317 318 319 320 321 322 323 324 325 326 327 328 329 330 331 332 333 334 335 336 337 338 339 340 341 342 343 344 345 346 347 348 349 350 351 352 353 354 355 356 357 358 359 360 361 362 363 364 365 366 367 368 369 370 371 372 373 374 375 376 377 378 379 380 381 382 383 384 385 386 387 388 389 390 391 392 393 394 395 396 397 398 399 400 401 402 403 404 405 406 407 408 409 410 411 412 413 414 415 416 417 418 419 420 421 422 423 424 425 426 427 428 429 430 431 432 433 434 435 436 437 438 439 440 441 442 443 444 445 446 447 448 449 450 451 452 453 454 455 456 457 458 459 460 461 462 463 464 465 466 467 468 469 470 471 472 473 474 475 476 477 478 479 480 481 482 483 484 485 486 487 488 489 490 491 492 493 494 495 496 497 498 499 500 501 502 503 504 505 506 507 508 509 510 511 512 513 514 515 516 517 518 519 520 521 522 523 524 525 526 527 528 529 530 531 532 533 534 535 536 537 538 539 540 541 542 543 544 545 546 547 548 549 550 551 552 553 554 555 556 557 558 559 560 561 562 563 564 565 566 567 568 569 570 571 572 573 574 575 576 577 578 579 580 581 582 583 584 585 586 587 588 589 590 591 592 593 594 595 596 597 598 599 600 601 602 603 604 605 606 607 608 609 610 611 612 613 614 615 616 617 618 619 620 621 622 623 624 625 626 627 628 629 630 631 632 633 634 635 636 637 638 639 640 641 642 643 644 645 646 647 648 649 650 651 652 653 654 655 656 657 658 659 660 661 662 663 664 665 666 667 668 669 670 671 672 673 674 675 676 677 678 679 680 681 682 683 684 685 686 687 688 689 690 691 692 693 694 695 696 697 698 699 700 701 702 703 704 705 706 70