**Rounding up to next power of 2.**

Source. <http://graphics.stanford.edu/~seander/bithacks.html#RoundUpPowerOf2>

The naïve method, is the find the highest bit set, then the bit higher (only that) to that would be the rounded two powers. E.g., if number is 5, i.e., 101, highest bit set is 3rd bit. Now, if we set the fourth bit to 1 and no other bit, we get 1000, which is 8 (that is, rounded to next power of 2).

Smart solution given by the link above, is to set all the bits from the highest bit set till first bit to 1, then add 1 to them—which makes them rounded to next 2 power number. For example, binary 111 + 1 is 1000 which is 8. I.e., 7’s next 2 power number is 8. How to achieve this faster? By recursively setting first highest bit, then first two highest bits, then 4 highest bits etc. Example, let’s say we have a number with highest bit set like this: 0001XXXXXX. Here X means it can be zero or one (doesn’t matter). For peace sake, let’s consider that number as 0001010001.

Now, first iterations

0001XXXXXX  
OR 00001XXXXX (Right shift by 1)  
= 00011XXXXX

Second Iteration

00011XXXXX  
OR 0000011XXX (Right shift by 2)  
= 0001111XXX

Third Iteration

0001111XXX  
OR 0000000111 (Right shift by 4)  
= 0001111111

Note how 0001010001 is transformed to 0001111111. All we need to do now is to add 1 to it. This makes the number 0010000000 which is the next 2 powers of the given number. This algorithm works by using the property that any number, 1 less than two powers has all its bit set. This algorithm works in *log(bits)*. I.e., number of bits used to represent the number. Code which does this is below.

unsigned int v; // compute the next highest power of 2 of 32-bit v

v--;

v |= v >> 1;

v |= v >> 2;

v |= v >> 4;

v |= v >> 8;

v |= v >> 16;

v++;

**Second solution:** This also runs ins **log(bits)** time. Find the highest bit set using kind of binary search. Divide the number into two halvs, check if the first half is zero (meaning highest bit is the second half), if yes then goto second half, else recurse on the first half. When there are only two bits left, we’re at the base case.

int round\_two\_powers(int num) {

int divisor = 16;

int bit\_count = 0;

*/\* Doing binary search here \*/*

while (divisor >= 1) {

*/\* Check if the first part of number is greater than zero.*

*\* If yes, this means the highest set bit is in the first half \*/*

if (num >> divisor) {

*/\* as highest set bit is in the second half, we would need divisor number of*

*\* bit shifts to get to first half \*/*

bit\_count += divisor;

*/\* We need to recurse in the first half again, thus reduce the num to only*

*\* first half \*/*

num >>= divisor;

} else {

*/\* second half. As first half is zero, the highest bit should be in*

*\* the second half\*/*

}

*/\* We keep halving the division of bits. I.e., we divide the number into 16 bit two chunks.*

*\* Then we divide it into two 8-bit chunks the, two 4-bits chunks till two 1-bits cunks*

*\*/*

divisor /= 2;

}

*/\* bit\_count will tell the exact highest bit set. We need one bit more to that \*/*

return 1 << (bit\_count + 1);

}

**An application of Stack.**

You are given two arrays **(without duplicates)** nums1 and nums2 where nums1’s elements are subset of nums2. Find all the next greater numbers for nums1's elements in the corresponding places of nums2.

The Next Greater Number of a number **x** in nums1 is the first greater number to its right in nums2. If it does not exist, output -1 for this number.

**Example 1:**

**Input:** **nums1** = [4,1,2], **nums2** = [1,3,4,2].

**Output:** [-1,3,-1]

**Explanation:**

For number 4 in the first array, you cannot find the next greater number for it in the second array, so output -1.

For number 1 in the first array, the next greater number for it in the second array is 3.

For number 2 in the first array, there is no next greater number for it in the second array, so output -1.

**Example 2:**

**Input:** **nums1** = [2,4], **nums2** = [1,2,3,4].

**Output:** [3,-1]

**Explanation:**

For number 2 in the first array, the next greater number for it in the second array is 3.

For number 4 in the first array, there is no next greater number for it in the second array, so output -1.

**Note:**

1. All elements in nums1 and nums2 are unique.
2. The length of both nums1 and nums2 would not exceed 1000.

Solution: Very naïve solution would be, for every number in *nums1*, find it in *nums2*, then iterate to find the next greater number. This would be an O(n^2) solution.

Smart Solution: Use stacks. For a given array like this [20, 5, 3, 1, 6, 7], for numbers 5,3,1 the next highest number would be 6. All we need to do is store the number in decreasing order in the stack and when a great element appears, pop elements from the stack till stack’s top is greater than the current element. For e.g., 20 will be on top of the stack when we hit 6 and pop the number 1,3,5 in that order. When we’re done iterating the array, whichever elements left in the stack would not have a greater element. In the above example, 20 and 7 will not have greater elements and thus, they’ll have -1 as answer.

Since nums1 is a subset, we may get query for any number. Thus, we should be quickly able to find solution to the queried number. Thus, we can use an unordered\_map to store the every numbers next greater number. Thus, queries can be done quickly using maps.

**Find if any permutation of a given string is a palindrome.**

Solution: Definitely we cannot permute the string and check if it is a palindrome. That’s an O(n!) or O(nn) solution.  
We can utilize the properties of the string and palindrome to solve it fast:

* Every character must appear even number of times, or
* There must be only one character which is of odd occurrence.

Thus, we can use a bitset of 256 bits to toggle the bit of the corresponding character (ascii value). The number of bits set at the end of all the characters should be 1 or 0. Anything more than that means more than one character appears odd number of times and thus it cannot be a palindrome.

**Alternative version of Binary Search doing finding lower bound index.**

// This function is an alternate version of binary serach which returns

// the lower bound index of 'num' in the array. I.e., 'num' is supposed to

// be at the returned location. 'num' will be <= to the element present at

// the returned index.

int LowerBoundIndex(uint64 \*arr, uint64 num, bool\* equal, int N) {

int step = N - 1, start = 0;

while(step > 0) {

step /= 2;

int mid = start + step;

if (arr[mid] == num) {

\*equal = true;

return mid;

} else if (arr[mid] > num) {

// do nothing

} else {

start = mid + 1;

}

}

\*equal = false;

return start;

}