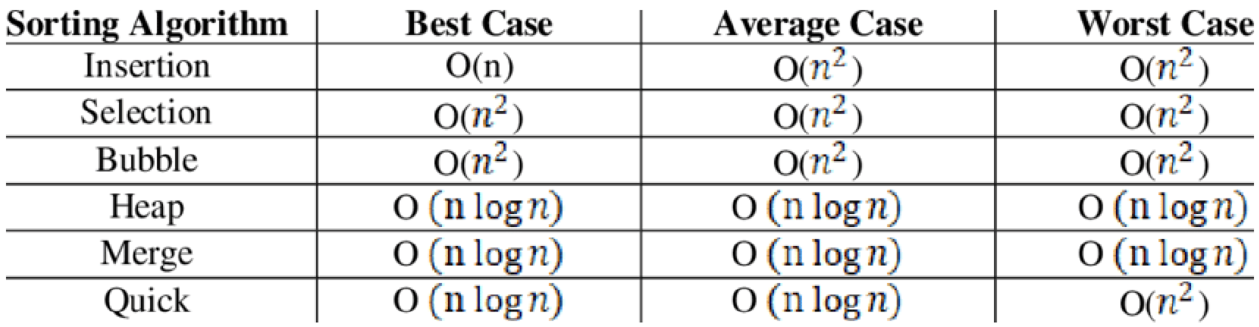
Program for Interview Preparation

Week 4 | Sorting, Hashing & Bit Manipulation

Sorting



Sorting Algorithm	Best Case	Average Case	Worst Case
Insertion	O(n)	$O(n^2)$	$O(n^2)$
Selection	$O(n^2)$	$O(n^2)$	$O(n^2)$
Bubble	$O(n^2)$	$O(n^2)$	$O(n^2)$
Heap	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Merge	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quick	$O(n \log n)$	$O(n \log n)$	$O(n^2)$

Some videos to explain the major sorting algorithms and their corresponding time complexities:

1. Insertion Sort: https://www.youtube.com/watch?v=ROalU379l3U
2. Selection Sort: https://www.youtube.com/watch?v=Ns4TPTC8whw
3. Bubble Sort: https://www.youtube.com/watch?v=IyZQPjUT5B4
4. Heap Sort: https://www.youtube.com/watch?v=Xw2D9aJRBY4
5. Merge Sort: https://www.youtube.com/watch?v=XaqR3G_NVoo
6. Quick Sort: https://www.youtube.com/watch?v=ywWBy6J5gz8

Questions

- https://practice.geeksforgeeks.org/problems/minimum-swaps/1
- https://practice.geeksforgeeks.org/problems/count-triplets-with-sum-smaller-than-x5549/1
- https://www.interviewbit.com/problems/wave-array/
- https://www.interviewbit.com/problems/hotel-bookings-possible/
- https://www.interviewbit.com/problems/noble-integer/

Hashing

Hashing is a technique that is used to uniquely identify a specific object from a group of similar objects. Some examples of how hashing is used in our lives include:

- In universities, each student is assigned a unique roll number that can be used to retrieve information about them.
- In libraries, each book is assigned a unique number that can be used to determine information about the book, such as its exact position in the library or the users it has been issued to etc.

In both these examples the students and books were hashed to a unique number.

Assume that you have an object and you want to assign a key to it to make searching easy. To store the key/value pair, you can use a simple array like a data structure where keys (integers) can be used directly as an index to store values. However, in cases where the keys are large and cannot be used directly as an index, you should use *hashing*.

In hashing, large keys are converted into small keys by using hash functions. The values are then stored in a data structure called hash table. The idea of hashing is to distribute entries (key/value pairs) uniformly across an array. Each element is assigned a key (converted key). By using that key you can access the element in O(1) time. Using the key, the algorithm (hash function) computes an index that suggests where an entry can be found or inserted.

Hashing is implemented in two steps:

1. An element is converted into an integer by using a hash function. This element can be used as an index to store the original element, which falls into the hash table.
2. The element is stored in the hash table where it can be quickly retrieved using hashed key.
   hash = hashfunc(key)
   index = hash % array_size

In this method, the hash is independent of the array size and it is then reduced to an index (a number between 0 and array_size − 1) by using the modulo operator (%).

Hash function

A hash function is any function that can be used to map a data set of an arbitrary size to a data set of a fixed size, which falls into the hash table. The values returned by a hash function are called hash values, hash codes, hash sums, or simply hashes.

To achieve a good hashing mechanism, It is important to have a good hash function with the following basic requirements:

1. Easy to compute: It should be easy to compute and must not become an algorithm in itself.
2. Uniform distribution: It should provide a uniform distribution across the hash table and should not result in clustering.
3. Less collisions: Collisions occur when pairs of elements are mapped to the same hash value. These should be avoided.
   Note: Irrespective of how good a hash function is, collisions are bound to occur. Therefore, to maintain the performance of a hash table, it is important to manage collisions through various collision resolution techniques.

Need for a good hash function

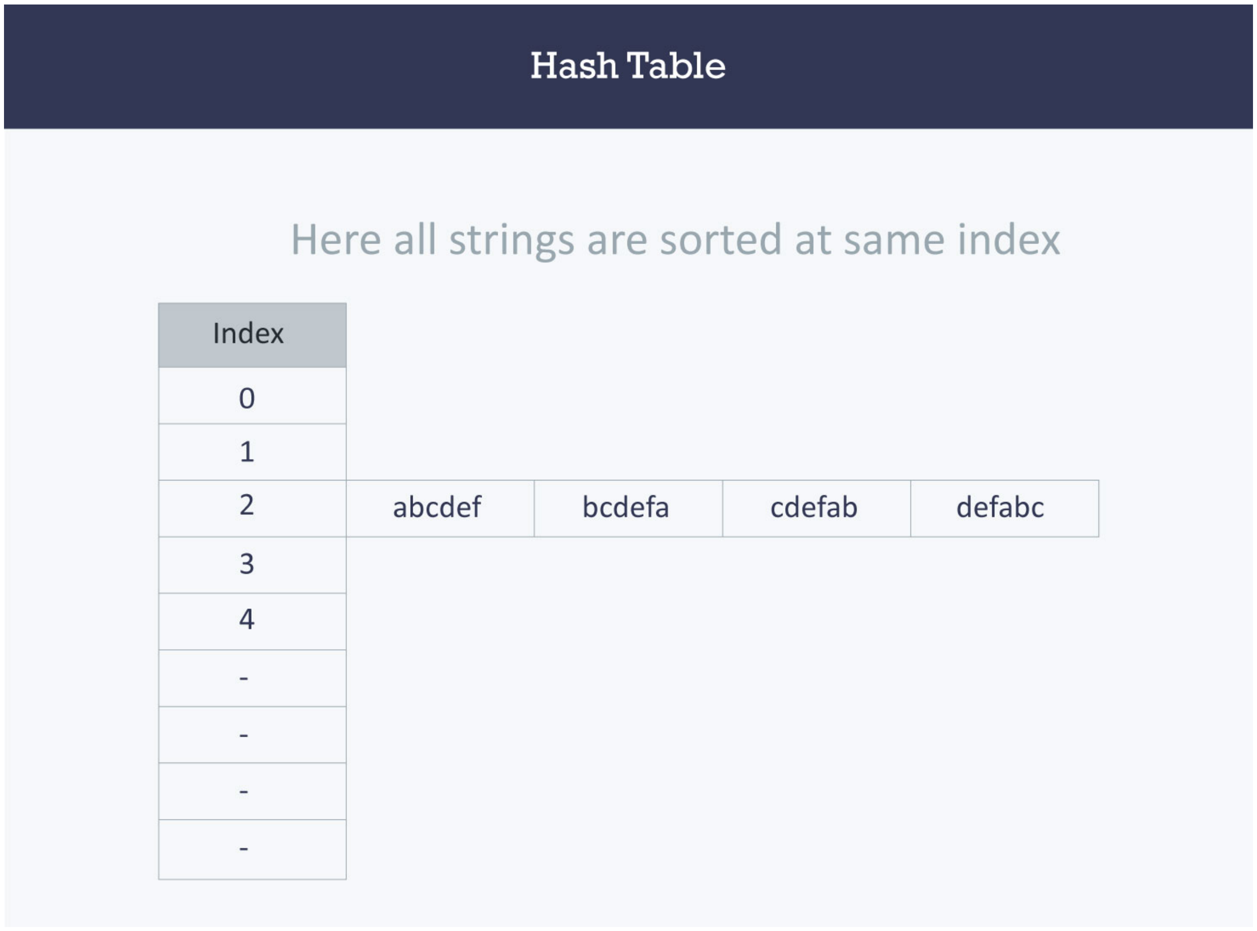
Let us understand the need for a good hash function. Assume that you have to store strings in the hash table by using the hashing technique {“abcdef”, “bcdefa”, “cdefab” , “defabc” }.

To compute the index for storing the strings, use a hash function that states the following:

The index for a specific string will be equal to the sum of the ASCII values of the characters modulo 599.

As 599 is a prime number, it will reduce the possibility of indexing different strings (collisions). It is recommended that you use prime numbers in case of modulo. The ASCII values of a, b, c, d, e, and f are 97, 98, 99, 100, 101, and 102 respectively. Since all the strings contain the same characters with different permutations, the sum will 599.

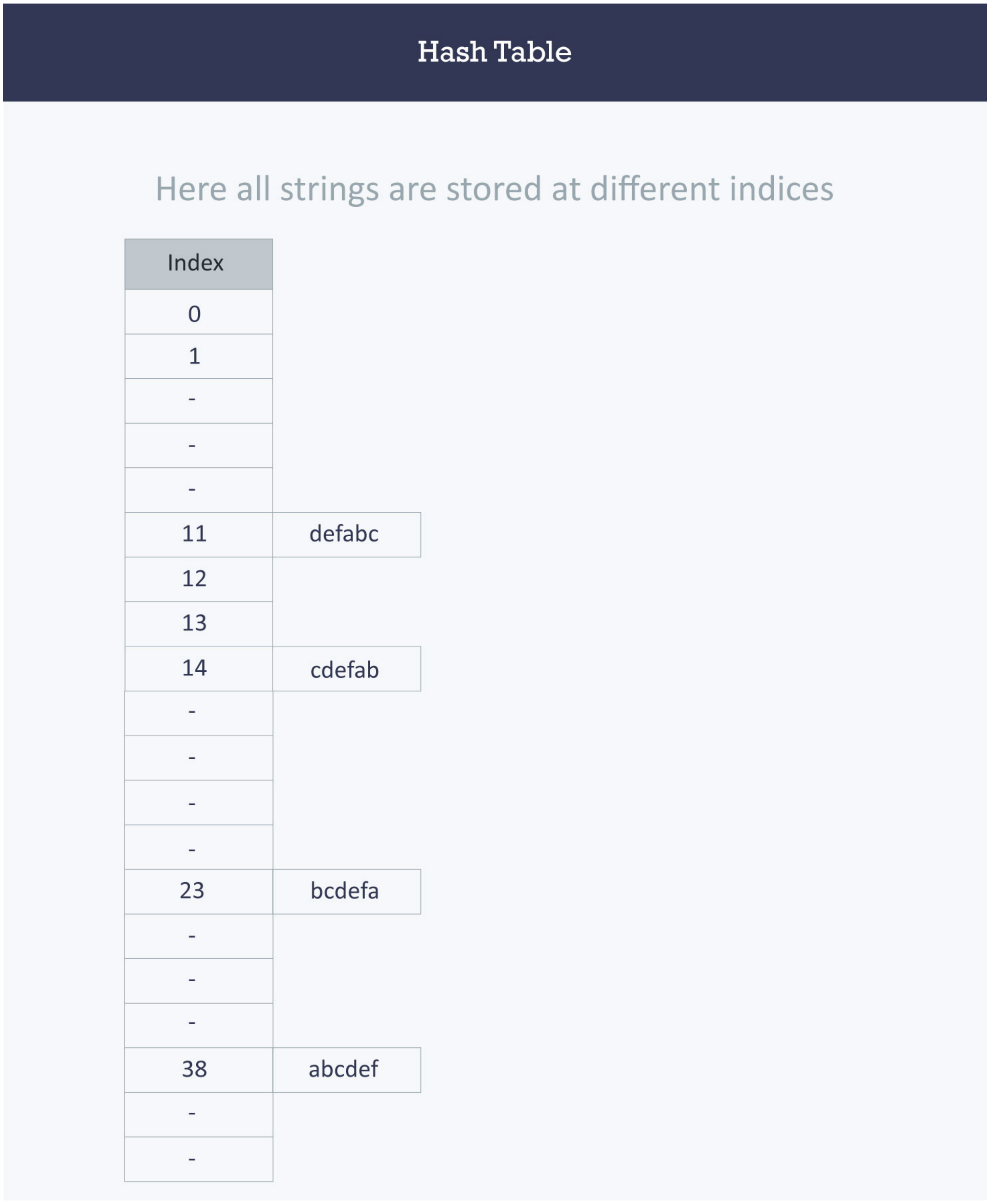
The hash function will compute the same index for all the strings and the strings will be stored in the hash table in the following format. As the index of all the strings is the same, you can create a list on that index and insert all the strings in that list.



Here, it will take O(n) time (where n is the number of strings) to access a specific string. This shows that the hash function is not a good hash function.

Let’s try a different hash function. The index for a specific string will be equal to sum of ASCII values of characters multiplied by their respective order in the string after which it is modulo with 2069 (prime number).

String	Hash function	Index
abcdef	(97*1 + 98*2 + 99*3 + 100*4 + 101*5 + 102*6)%2069	38
bcdefa	(98*1 + 99*2 + 100*3 + 101*4 + 102*5 + 97*6)%2069	23
cdefab	(99*1 + 100*2 + 101*3 + 102*4 + 97*5 + 98*6)%2069	14
defabc	(100*1 + 101*2 + 102*3 + 97*4 + 98*5 + 99*6)%2069	11



Hash Table

Here all strings are stored at different indices

Index	
0	
1	
-	
-	
-	
11	defabc
12	
13	
14	cdefab
-	
-	
-	
-	
23	bcdefa
-	
-	
-	
38	abcdef
-	
-	

Hash table

A hash table is a data structure that is used to store keys/value pairs. It uses a hash function to compute an index into an array in which an element will be inserted or searched. By using a good hash function, hashing can work well. Under reasonable assumptions, the average time required to search for an element in a hash table is O(1).

Let us consider string S. You are required to count the frequency of all the characters in this string.

```
string S = “ababcd”
```

The simplest way to do this is to iterate over all the possible characters and count their frequency one by one. The time complexity of this approach is O(26*N) where N is the size of the string and there are 26 possible characters.

```
 void countFre(string S)
    {
        for(char c = ‘a’;c <= ‘z’;++c)
        {
            int frequency = 0;
            for(int i = 0;i < S.length();++i)
                if(S[i] == c)
                    frequency++;
            cout << c << ‘ ‘ << frequency << endl;
        }
    }
```

Output

```
a 2
b 2
c 1
d 1
e 0
f 0
…
z 0
```

Let us apply hashing to this problem. Take an array frequency of size 26 and hash the 26 characters with indices of the array by using the hash function. Then, iterate over the string and increase the value in the frequency at the corresponding index for each character. The complexity of this approach is O(N) where N is the size of the string.

```
int Frequency[26];

    int hashFunc(char c)
    {
        return (c - ‘a’);
    }

    void countFre(string S)
    {
        for(int i = 0;i < S.length();++i)
        {
            int index = hashFunc(S[i]);
            Frequency[index]++;
        }
        for(int i = 0;i < 26;++i)
            cout << (char)(i+’a’) << ‘ ‘ << Frequency[i] << endl;
    }
```

Output

```
a 2
b 2
c 1
d 1
e 0
f 0
…
z 0
```

Questions

- https://practice.geeksforgeeks.org/problems/smallest-window-in-a-string-containing-all-the-characters-of-another-string-1587115621/1
- https://practice.geeksforgeeks.org/problems/zero-sum-subarrays1825/1
- https://www.interviewbit.com/problems/anagrams/
- https://www.interviewbit.com/problems/an-increment-problem/
- https://www.interviewbit.com/problems/longest-substring-without-repeat/
- https://leetcode.com/problems/lru-cache/

Bit Manipulation

Working on bytes, or data types comprising of bytes like ints, floats, doubles or even data structures which stores large amount of bytes is normal for a programmer. In some cases, a programmer needs to go beyond this - that is to say that in a deeper level where the importance of bits is realized.

Operations with bits are used in Data compression (data is compressed by converting it from one representation to another, to reduce the space) ,Exclusive-Or Encryption (an algorithm to encrypt the data for safety issues). In order to encode, decode or compress files we have to extract the data at bit level. Bitwise Operations are faster and closer to the system and sometimes optimize the program to a good level.

We all know that 1 byte comprises of 8 bits and any integer or character can be represented using bits in computers, which we call its binary form(contains only 1 or 0) or in its base 2 form.

Example:

1) 14 = $\{1110\}_2$

= $1 * 2^3 + 1 * 2^2 + 1 * 2^1 + 0 * 2^0$

= 14.

2) 20 = $\{10100\}_2$

= $1 * 2^4 + 0 * 2^3 + 1 * 2^2 + 0 * 2^1 + 0 * 2^0$

= 20.

For characters, we use ASCII representation, which are in the form of integers which again can be represented using bits as explained above.

Bitwise Operators:

There are different bitwise operations used in the bit manipulation. These bit operations operate on the individual bits of the bit patterns. Bit operations are fast and can be used in optimizing time complexity. Some common bit operators are:

NOT (~): Bitwise NOT is an unary operator that flips the bits of the number i.e., if the ith bit is 0, it will change it to 1 and vice versa. Bitwise NOT is nothing but simply the one’s complement of a number. Lets take an example.

N = 5 = $(101)_2$

~N = ~5 = ~$(101)_2$ = $(010)_2$ = 2

AND (&): Bitwise AND is a binary operator that operates on two equal-length bit patterns. If both bits in the compared position of the bit patterns are 1, the bit in the resulting bit pattern is 1, otherwise 0.

A = 5 = $(101)_2$, B = 3 = $(011)_2$ A & B = $(101)_2$ & $(011)_2$= $(001)_2$ = 1

OR (|): Bitwise OR is also a binary operator that operates on two equal-length bit patterns, similar to bitwise AND. If both bits in the compared position of the bit patterns are 0, the bit in the resulting bit pattern is 0, otherwise 1.

A = 5 = $(101)_2$, B = 3 = $(011)_2$

A | B = $(101)_2$ | $(011)_2$ = $(111)_2$ = 7

XOR (^): Bitwise XOR also takes two equal-length bit patterns. If both bits in the compared position of the bit patterns are 0 or 1, the bit in the resulting bit pattern is 0, otherwise 1.

A = 5 = $(101)_2$, B = 3 = $(011)_2$

A ^ B = $(101)_2$ ^ $(011)_2$ = $(110)_2$ = 6

Left Shift (<<): Left shift operator is a binary operator which shift the some number of bits, in the given bit pattern, to the left and appends 0 at the end. Left shift is equivalent to multiplying the bit pattern with 2^k (if we are shifting k bits).

1 << 1 = 2 = 2^1

1 << 2 = 4 = 2^2 1 << 3 = 8 = 2^3

1 << 4 = 16 = 2^4

…

1 << n = 2^n

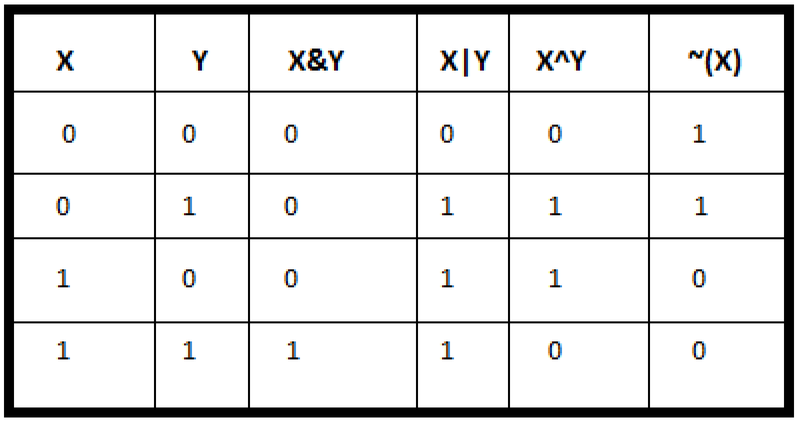
Right Shift (>>): Right shift operator is a binary operator which shift the some number of bits, in the given bit pattern, to the right and append 1 at the end. Right shift is equivalent to dividing the bit pattern with 2^k (if we are shifting k bits).

4 >> 1 = 2

6 >> 1 = 3

5 >> 1 = 2

16 >> 4 = 1



X	Y	X&Y	X|Y	X^Y	~(X)
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

Bitwise operators are good for saving space and sometimes to cleverly remove dependencies.

Note: All left and right side taken in this article, are taken with reference to the reader.

Lets discuss some algorithms based on bitwise operations:

1) How to check if a given number is a power of 2 ?

Consider a number N and you need to find if N is a power of 2. Simple solution to this problem is to repeated divide N by 2 if N is even. If we end up with a 1 then N is power of 2, otherwise not. There are a special case also. If N = 0 then it is not a power of 2. Let’s code it.

```
bool isPowerOfTwo(int x)
    {
        if(x == 0)
            return false;
        else
        {
            while(x % 2 == 0) x /= 2;
            return (x == 1);
            }
    }
```

Above function will return true if x is a power of 2, otherwise false.

Time complexity of the above code is O(logN).

The same problem can be solved using bit manipulation. Consider a number x that we need to check for being a power for 2. Now think about the binary representation of (x-1). (x-1) will have all the bits same as x, except for the rightmost 1 in x and all the bits to the right of the rightmost 1.

Let, $x = 4 = (100)_2$

$x - 1 = 3 = (011)_2$

Let, $x = 6 = (110)_2$

$x - 1 = 5 = (101)_2$

It might not seem obvious with these examples, but binary representation of (x-1) can be obtained by simply flipping all the bits to the right of rightmost 1 in x and also including the rightmost 1.

Now think about x & (x-1). x & (x-1) will have all the bits equal to the x except for the rightmost 1 in x.

Let, $x = 4 = (100)_2$

$x - 1 = 3 = (011)_2$

$x \& (x-1) = 4 \& 3 = (100)_2 \& (011)_2 = (000)_2$

Let, $x = 6 = (110)_2$

$x - 1 = 5 = (101)_2$

$x \& (x-1) = 6 \& 5 = (110)_2 \& (101)_2 = (100)_2$

Properties for numbers which are powers of 2, is that they have one and only one bit set in their binary representation. If the number is neither zero nor a power of two, it will have 1 in more than one place. So if x is a power of 2 then x & (x-1) will be 0.

```
 bool isPowerOfTwo(int x)
    {
        // x will check if x == 0 and !(x & (x - 1)) will check if x is a power of 2 or not
        return (x && !(x & (x - 1)));
    }
```

2) Count the number of ones in the binary representation of the given number.

The basic approach to evaluate the binary form of a number is to traverse on it and count the number of ones. But this approach takes $\log_2 N$ of time in every case.

Why $\log_2 N$?

As to get a number in its binary form, we have to divide it by 2, until it gets 0, which will take $\log_2 N$ of time.

With bitwise operations, we can use an algorithm whose running time depends on the number of ones present in the binary form of the given number. This algorithm is much better, as it will reach to logN, only in its worst case.

```
 int count_one (int n)
     {
        while( n )
        {
        n = n&(n-1);
           count++;
        }
        return count;
  }
```

Why this algorithm works ?

As explained in the previous algorithm, the relationship between the bits of x and x-1. So as in x-1, the rightmost 1 and bits right to it are flipped, then by performing x&(x-1), and storing it in x, will reduce x to a number containing number of ones(in its binary form) less than the previous state of x, thus increasing the value of count in each iteration.

Example:

n = 23 = $\{10111\}_2$.

1. Initially, count = 0.

2. Now, n will change to n&(n-1). As n-1 = 22 = $\{10110\}_2$, then n&(n-1) will be $\{10111_2$ & $\{10110\}_2$, which will be $\{10110\}_2$ which is equal to 22. Therefore n will change to 22 and count to 1.

3. As n-1 = 21 = $\{10101\}_2$, then n&(n-1) will be $\{10110\}_2$ & $\{10101\}_2$, which will be $\{10100\}_2$ which is equal to 20. Therefore n will change to 20 and count to 2.

4. As n-1 = 19 = $\{10011\}_2$, then n&(n-1) will be $\{10100\}_2$ & $\{10011\}_2$, which will be $\{10000\}_2$ which is equal to 16. Therefore n will change to 16 and count to 3.

5. As n-1 = 15 = $\{01111\}_2$, then n&(n-1) will be $\{10000\}_2$ & $\{01111\}_2$, which will be $\{00000\}_2$ which is equal to 0. Therefore n will change to 0 and count to 4.

6. As n = 0, the the loop will terminate and gives the result as 4.

Complexity: O(K), where K is the number of ones present in the binary form of the given number.

3) Check if the i^{th} bit is set in the binary form of the given number.

To check if the i^{th} bit is set or not (1 or not), we can use AND operator. How?

Let's say we have a number N, and to check whether it's i^{th} bit is set or not, we can AND it with the number 2^i . The binary form of 2^i contains only i^{th} bit as set (or 1), else every bit is 0 there. When we will AND it with N, and if the i^{th} bit of N is set, then it will return a non zero number (2^i to be specific), else 0 will be returned.

Using Left shift operator, we can write 2^i as 1 << i . Therefore:

```
bool check (int N)
  {
      if( N & (1 << i) )
          return true;
      else
          return false;
  }
```

Example:

Let's say N = 20 = $\{10100\}_2$. Now let's check if it's 2nd bit is set or not(starting from 0). For that, we have to AND it with 2^2 = 1<<2 = $\{100\}_2$.

{10100} & {100} = {100} = 2^2 = 4(non-zero number), which means it's 2nd bit is set.

4) How to generate all the possible subsets of a set ?

A big advantage of bit manipulation is that it can help to iterate over all the subsets of an N-element set. As we all know there are 2^N possible subsets of any given set with N elements. What if we represent each element in a subset with a bit. A bit can be either 0 or 1, thus we can use this to denote whether the corresponding element belongs to this given subset or not. So each bit pattern will represent a subset.

Consider a set A of 3 elements.

A = {a, b, c}

Now, we need 3 bits, one bit for each element. 1 represent that the corresponding element is present in the subset, whereas 0 represent the corresponding element is not in the subset. Let's write all the possible combination of these 3 bits.

0 = $(000)_2$ = {}

1 = $(001)_2$ = {c}

2 = $(010)_2$ = {b}

3 = $(011)_2$ = {b, c}

4 = $(100)_2$ = {a}

5 = $(101)_2$ = {a, c}

6 = $(110)_2$ = {a, b}

7 = $(111)_2$ = {a, b, c}

```
possibleSubsets(A, N):
    for i = 0 to 2^N:
        for j = 0 to N:
            if jth bit is set in i:
                print A[j]
        print '\n'
```

```
 void possibleSubsets(char A[], int N)
  {
      for(int i = 0;i < (1 << N); ++i)
      {
        for(int j = 0;j < N;++j)
            if(i & (1 << j))
                cout << A[j] << ' ';
        cout << endl;
    }
}
```

5) Find the largest power of 2 (most significant bit in binary form), which is less than or equal to the given number N.

Idea: Change all the bits which are at the right side of the most significant digit, to 1.

Property: As we know that when all the bits of a number N are 1, then N must be equal to the 2^i -1 , where i is the number of bits in N.

Example:

Let's say binary form of a N is $\{1111\}_2$ which is equal to 15.

15 = 2^4-1, where 4 is the number of bits in N.

This property can be used to find the largest power of 2 less than or equal to N. How?

If we somehow, change all the bits which are at right side of the most significant bit of N to 1, then the number will become x + (x-1) = 2 * x -1 , where x is the required answer.

Example:

Let's say N = 21 = {10101}, here most significant bit is the 4th one. (counting from 0th digit) and so the answer should be 16.

So lets change all the right side bits of the most significant bit to 1. Now the number changes to

{11111} = 31 = 2 * 16 -1 = Y (let's say).

Now the required answer is (Y+1)>>1 or (Y+1)/2.

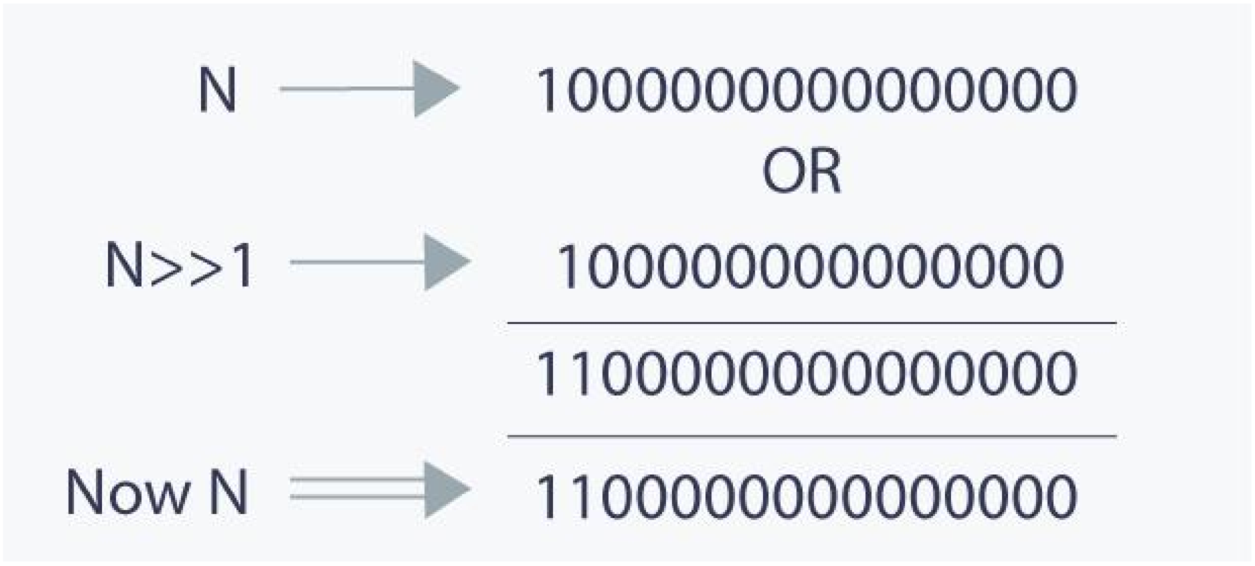
Now the question arises here is how can we change all right side bits of most significant bit to 1?

Let's take the N as 16 bit integer and binary form of N is {1000000000000000}.

Here we have to change all the right side bits to 1.

Initially we will copy that most significant bit to its adjacent right side by:

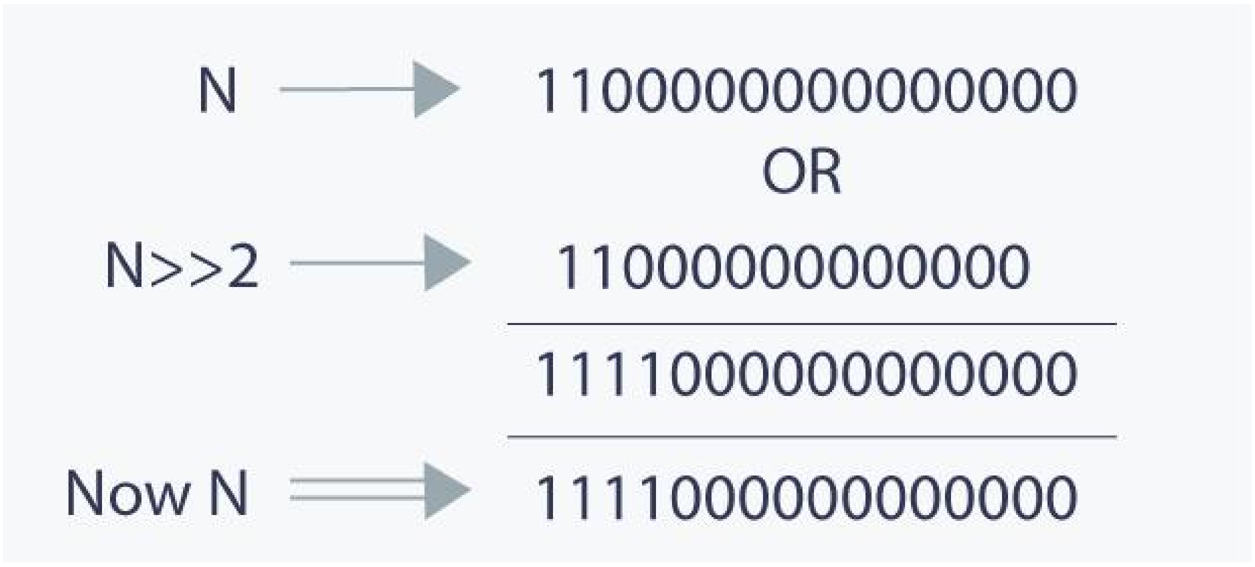
N = N | (N>>1).



As you can see, in above diagram, after performing the operation, rightmost bit has been copied to its adjacent place.

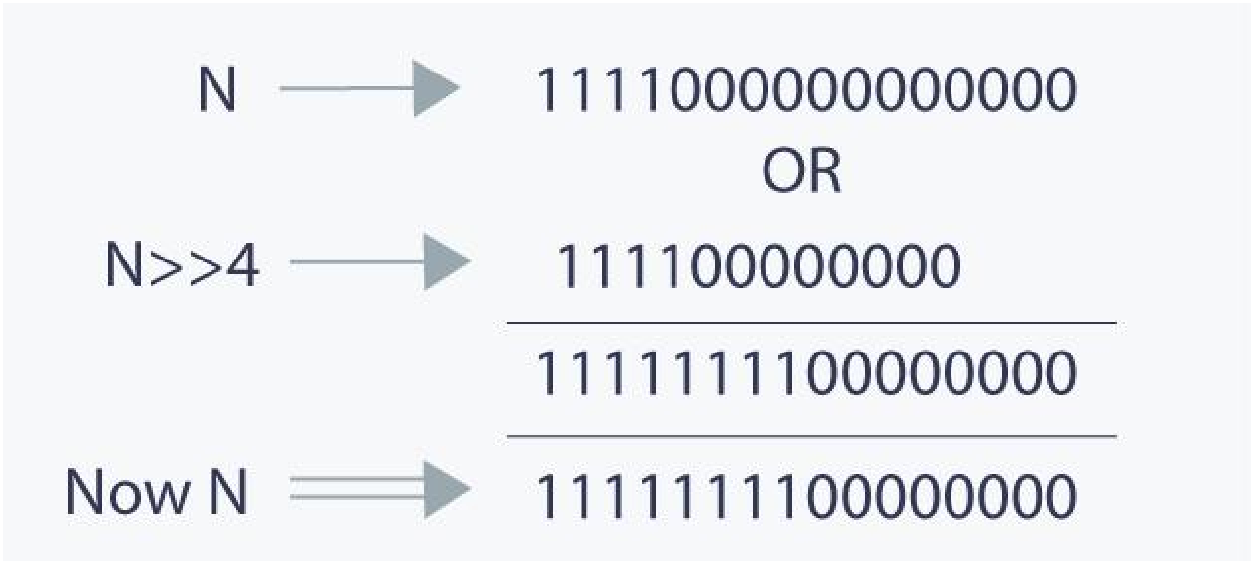
Now we will copy the 2 rightmost set bits to their adjacent right side.

N = N | (N>>2).



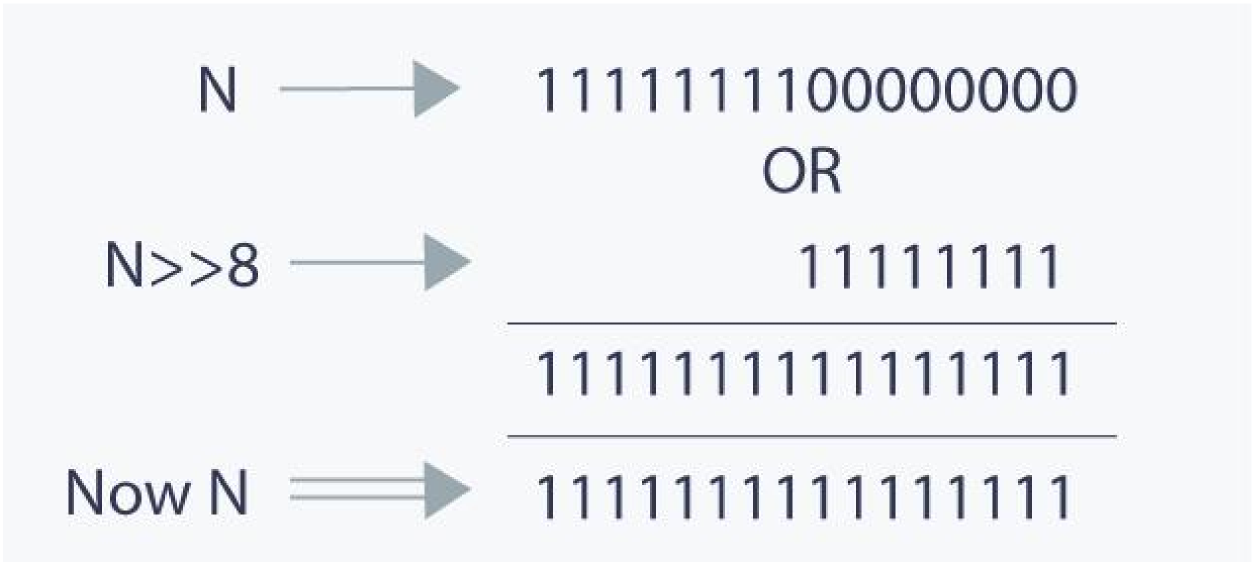
Now we will copy the 4 rightmost set bit to their adjacent right side.

N = N | (N>>4)



Now we will copy these 8 rightmost set bits to their adjacent right side.

N = N| (N>>8)



Now all the right side bits of the most significant set bit has been changed to 1 .This is how we can change right side bits. This explanation is for 16 bit integer, and it can be extended for 32 or 64 bit integer too.

```
long largest_power(long N)
  {
      //changing all right side bits to 1.
      N = N| (N>>1);
      N = N| (N>>2);
      N = N| (N>>4);
      N = N| (N>>8);
```

```
    //as now the number is 2 * x-1, where x is required answer, so adding 1 and dividing it by
     2.
            return (N+1)>>1;

    }
```

Tricks with Bits:

1) x ^ (x & (x-1)) : Returns the rightmost 1 in binary representation of x.

As explained above, (x & (x - 1)) will have all the bits equal to the x except for the rightmost 1 in x. So if we do bitwise XOR of x and (x & (x-1)), it will simply return the rightmost 1. Let's see an example.

x = 10 = $(1010)_2$ ` x & (x-1) = $(1010)_2$ & $(1001)_2$ = $(1000)_2$

x ^ (x & (x-1)) = $(1010)_2$ ^ $(1000)_2$ = $(0010)_2$

2) x & (-x) : Returns the rightmost 1 in binary representation of x

(-x) is the two's complement of x. (-x) will be equal to one's complement of x plus 1.

Therefore (-x) will have all the bits flipped that are on the left of the rightmost 1 in x. So x & (-x) will return rightmost 1.

x = 10 = $(1010)_2$

(-x) = -10 = $(0110)_2$

x & (-x) = $(1010)_2$ & $(0110)_2$ = $(0010)_2$

3) x | (1 << n) : Returns the number x with the nth bit set.

(1 << n) will return a number with only nth bit set. So if we OR it with x it will set the nth bit of x.

x = 10 = $(1010)_2$ n = 2

1 << n = $(0100)_2$

x | (1 << n) = $(1010)_2$ | $(0100)_2$ = $(1110)_2$

Applications of bit operations:

1) They are widely used in areas of graphics, specially XOR(Exclusive OR) operations.

2) They are widely used in embedded systems, in situations, where we need to set/clear/toggle just one single bit of a specific register without modifying the other contents. We can do OR/AND/XOR operations with the appropriate mask for the bit position.

3) Data structure like an n-bit map can be used to allocate an n-size resource pool to represent the current status.

4) Bits are used in networking, framing the packets of numerous bits which is sent to another system generally through any type of serial interface.

Questions

- https://practice.geeksforgeeks.org/problems/finding-the-numbers0215/1
- https://www.interviewbit.com/problems/min-xor-value/
- https://www.interviewbit.com/problems/single-number-ii/
- https://leetcode.com/problems/maximum-product-of-word-lengths/
- https://practice.geeksforgeeks.org/problems/power-set4302/1