CHAPTER

21

MISCELLANEOUS CONCEPTS

21.1 Introduction

In this chapter we will cover the topics which are useful for interviews and exams.

21.2 Hacks on Bitwise Programming

In C and $C++$ we can work with bits effectively. First let us see the definitions of each bit operation and then move onto different techniques for solving the problems. Basically, there are six operators that C and $C++$ support for bit manipulation:

Symbol	Operation
&	Bitwise AND
|	Bitwise OR
^	Bitwise Exclusive-OR
<<	Bitwise left shift
>>	Bitwise right shift
~	Bitwise complement

21.2.1 Bitwise AND

The bitwise AND tests two binary numbers and returns bit values of 1 for positions where both numbers had a one, and bit values of 0 where both numbers did not have one:

```
  01001011
& 00010101
----------
  00000001
```

21.2.2 Bitwise OR

The bitwise OR tests two binary numbers and returns bit values of 1 for positions where either bit or both bits are one, the result of 0 only happens when both bits are 0:

```
  01001011
| 00010101
----------
  01011111
```

21.2.3 Bitwise Exclusive-OR

The bitwise Exclusive-OR tests two binary numbers and returns bit values of 1 for positions where both bits are different; if they are the same then the result is 0:

```
  01001011
^ 00010101
----------
  01011110
```

21.2.4 Bitwise Left Shift

The bitwise left shift moves all bits in the number to the left and fills vacated bit positions with 0.

```
  01001011
<< 2
--------
  00101100
```

21.2.5 Bitwise Right Shift

The bitwise right shift moves all bits in the number to the right.

```
  01001011
>> 2
--------
  ??010010
```

Note the use of ? for the fill bits. Where the left shift filled the vacated positions with 0, a right shift will do the same only when the value is unsigned. If the value is signed then a right shift will fill the vacated bit positions with the sign bit or 0, whichever one is implementation-defined. So the best option is to never right shift signed values.

21.2.6 Bitwise Complement

The bitwise complement inverts the bits in a single binary number.

```
  01001011
~
---------
  10110100
```

21.2.7 Checking Whether K-th Bit is Set or Not

Let us assume that the given number is n. Then for checking the K^{th} bit we can use the expression: $n \,\&\, (1 \ll K - 1)$. If the expression is true then we can say the K^{th} bit is set (that means, set to 1).

Example:

n = 01001011 and $K = 4$

$1 \ll K - 1$	00001000
$n \,\&\, (1 \ll K - 1)$	00001000

21.2.8 Setting K-th Bit

For a given number n, to set the K^{th} bit we can use the expression: $n \mid 1 \ll (K - 1)$

Example:

n = 01001011 and $K = 3$

$1 \ll K - 1$	00000100
$n \mid (1 \ll K - 1)$	01001111

21.2.9 Clearing K-th Bit

To clear K^{th} bit of a given number n, we can use the expression: $n \,\&\, \sim(1 \ll K - 1)$

Example:

n = 01001011 and $K = 4$

$$\begin{array}{rl} 1 \ll K-1 & 00001000 \\ \sim(1 \ll K-1) & 11110111 \\ n \,\&\, \sim(1 \ll K-1) & 01000011 \end{array}$$

21.2.10 Toggling K-th Bit

For a given number n, for toggling the K^{th} bit we can use the expression: $n \,\hat{}\, (1 \ll K-1)$

Example:

$$\begin{array}{rl} n = 01001011 \text{ and } K = 3 & \\ 1 \ll K-1 & 00000100 \\ n \,\hat{}\, (1 \ll K-1) & 01001111 \end{array}$$

21.2.11 Toggling Rightmost One Bit

For a given number n, for toggling rightmost one bit we can use the expression: $n \,\&\, n-1$

Example:

$$\begin{array}{rl} n = & 01001011 \\ n-1 & 01001010 \\ n \,\&\, n-1 & 01001010 \end{array}$$

21.2.12 Isolating Rightmost One Bit

For a given number n, for isolating rightmost one bit we can use the expression: $n \,\&\, -n$

Example:

$$\begin{array}{rl} n = & 01001011 \\ -n & 10110101 \\ n \,\&\, -n & 00000001 \end{array}$$

Note: For computing $-n$, use two's complement representation. That means, toggle all bits and add 1.

21.2.13 Isolating Rightmost Zero Bit

For a given number n, for isolating rightmost zero bit we can use the expression: $\sim n \,\&\, n+1$

Example:

$$\begin{array}{rl} n = & 01001011 \\ \sim n & 10110100 \\ n+1 & 01001100 \\ \sim n \,\&\, n+1 & 00000100 \end{array}$$

21.2.14 Checking Whether Number is Power of 2 or Not

Given number n, to check whether the number is in 2^n form for not, we can use the expression: $if(n \,\&\, n-1 == 0)$

Example:

$$\begin{array}{rl} n = & 01001011 \\ n-1 & 01001010 \\ n \,\&\, n-1 & 01001010 \\ if(n \,\&\, n-1 == 0) & 0 \end{array}$$

21.2.15 Multiplying Number by Power of 2

For a given number n, to multiply the number with 2^K we can use the expression: $n \ll K$

Example:

$$\begin{array}{rl} n = 00001011 \text{ and } K = 2 & \\ n \ll K & 00101100 \end{array}$$

21.2.16 Dividing Number by Power of 2

For a given number n, to divide the number with 2^K we can use the expression: $n \gg K$

Example:

$$\begin{array}{rl} n = 00001011 \text{ and } K = 2 & \\ n \gg K & 00010010 \end{array}$$

21.2.17 Finding Modulo of a Given Number

For a given number n, to find the %8 we can use the expression: $n \& 0x7$. Similarly, to find %32, use the expression: $n \& 0x1F$

Note: Similarly, we can find modulo value of any number.

21.2.18 Reversing the Binary Number

For a given number n, to reverse the bits (reverse (mirror) of binary number) we can use the following code snippet:

```
def reverseNumber(n):
    nReverse = n
    s = n.bit_length()
    while(n):
            nReverse <<= 1
            nReverse |= (n & 1)
            s -= 1
            n >>= 1
    nReverse <<= s
    return nReverse
n = 4
print n, reverseNumber(n)
```

Time Complexity: This requires one iteration per bit and the number of iterations depends on the size of the number.

21.2.19 Counting Number of One's in Number

For a given number n, to count the number of 1's in its binary representation we can use any of the following methods.

Method1: Process bit by bit

```
def numberOfOnes(n):
    count=0
    while(n):
        count += n & 1
        n >>= 1
    print count
```

Time Complexity: This approach requires one iteration per bit and the number of iterations depends on system.

Method2: Using modulo approach

```
def numberOfOnes2(n):
    count=0
    while(n):
        if(n%2 ==1):
            count += 1
        n = n/2
    print count
```

Time Complexity: This requires one iteration per bit and the number of iterations depends on system.

Method3: Using toggling approach: $n \& n-1$

```
def numberOfOnes3(n):
    count=0
    while(n):
            count += 1
            n &= n - 1
    print count
```

Time Complexity: The number of iterations depends on the number of 1 bits in the number.

Method4: Using preprocessing idea. In this method, we process the bits in groups. For example if we process them in groups of 4 bits at a time, we create a table which indicates the number of one's for each of those possibilities (as shown below).

0000→0	0100→1	1000→1	1100→2
0001→1	0101→2	1001→2	1101→3
0010→1	0110→2	1010→2	1110→3
0011→2	0111→3	1011→3	1111→4

The following code to count the number of 1s in the number with this approach:

```
def numberOfOnes4(n):
    Table = [0,1,1,2,1,2,2,3,1,2,2,3,2,3,3,4]
    count = 0
    while (n):
            count = count + Table[n & 0xF]
            n >>= 4
    print count
```

Time Complexity: This approach requires one iteration per 4 bits and the number of iterations depends on system.

21.2.20 Creating Mask for Trailing Zero's

For a given number n, to create a mask for trailing zeros, we can use the expression: $(n \& -n) - 1$

Example:

$$\begin{aligned} n &= & 01001011 \\ -n & & 10110101 \\ n \& -n & & 00000001 \\ (n \& -n) - 1 & & 00000000 \end{aligned}$$

Note: In the above case we are getting the mask as all zeros because there are no trailing zeros.

27.2.21 Swap all odd and even bits

Example:

$$\begin{aligned} n &= & 01001011 \\ \text{Find even bits of given number (evenN)} &= n \& 0xAA & 00001010 \\ \text{Find odd bits of given number (oddN)} &= n \& 0x55 & 01000001 \\ & \text{evenN >>= 1} & 00000101 \\ & \text{oddN <<= 1} & 10000010 \\ & \text{Final Expression: evenN | oddN} & 10000111 \end{aligned}$$

21.2.22 Performing Average without Division

Is there a bit-twiddling algorithm to replace $mid = (low + high) / 2$ (used in Binary Search and Merge Sort) with something much faster?

We can use $mid = (low + high) >> 1$. Note that using $(low + high) / 2$ for midpoint calculations won't work correctly when integer overflow becomes an issue. We can use bit shifting and also overcome a possible overflow issue: $low + ((high - low)/ 2)$ and the bit shifting operation for this is $low + ((high - low) >> 1)$.

21.3 Other Programming Questions with Solutions

Problem-1 Give an algorithm for printing the matrix elements in spiral order.

Solution: Non-recursive solution involves directions right, left, up, down, and dealing their corresponding indices. Once the first row is printed, direction changes (from right) to down, the row is discarded by incrementing the upper limit. Once the last column is printed, direction changes to left, the column is discarded by decrementing the right hand limit.

```
def spiralIterative(n):
    dx,dy = 1,0          # Starting increments
    x,y = 0,0            # Starting location
    matrix = [[None]* n for j in range(n)]
    for i in xrange(n**2):
        matrix[x][y] = i
        nx,ny = x+dx, y+dy
        if 0<=nx<n and 0<=ny<n and matrix[nx][ny] == None:
            x,y = nx,ny
        else:
            dx,dy = -dy,dx
```

```
            x,y = x+dx, y+dy
        return matrix

    def printSpiral(matrix):
        n = range(len(matrix))
        for y in n:
            for x in n:
                print "%2i" % matrix[x][y],
            print
    printSpiral(spiralIterative(5))
```

Recursive:

```
    def spiral(n):
        def spiralPart(x, y, n):
            if x == -1 and y == 0:
                return -1
            if y == (x+1) and x < (n // 2):
                return spiralPart(x-1, y-1, n-1) + 4*(n-y)
            if x < (n-y) and y <= x:
                return spiralPart(y-1, y, n) + (x-y) + 1
            if x >= (n-y) and y <= x:
                return spiralPart(x, y-1, n) + 1
            if x >= (n-y) and y > x:
                return spiralPart(x+1, y, n) + 1
            if x < (n-y) and y > x:
                return spiralPart(x, y-1, n) - 1
        array = [[0] * n for j in xrange(n)]
        for x in xrange(n):
            for y in xrange(n):
                array[x][y] = spiralPart(y, x, n)
        return array
    for row in spiral(5):
        print " ".join("%2s" % x for x in row)
```

Time Complexity: $O(n^2)$. Space Complexity: O(1).

Problem-2 Give an algorithm for shuffling the desk of cards.

Solution: Assume that we want to shuffle an array of 52 cards, from 0 to 51 with no repeats, such as we might want for a deck of cards. First fill the array with the values in order, then go through the array and exchange each element with a randomly chosen element in the range from itself to the end. It's possible that an element will swap with itself, but there is no problem with that.

```
    import random
    def shuffle(cards):
        max = len(cards)-1
            while max != 0:
            r = random.randint(0, max)
            cards[r], cards[max] = cards[max], cards[r]
            max = max - 1
        return cards
    data = range(1, 53)
    print shuffle(data)
```

Time Complexity: O(n). Space Complexity: O(1).

Problem-3 Reversal algorithm for array rotation: Write a function rotate(A[], d, n) that rotates A[] of size n by d elements. For example, the array $1, 2, 3, 4, 5, 6, 7$ becomes $3, 4, 5, 6, 7, 1, 2$ after 2 rotations.

Solution: Consider the following algorithm.

Algorithm:

```
rotate(Array[], d, n)
reverse(Array[], 1, d) ;
reverse(Array[], d + 1, n);
reverse(Array[], l, n);
```

Let AB be the two parts of the input Arrays where A = Array[0..d-1] and B = Array[d..n-1]. The idea of the algorithm is:

Reverse A to get ArB. /* Ar is reverse of A */
Reverse B to get ArBr. /* Br is reverse of B */
Reverse all to get (ArBr) r = BA.
For example, if Array[] = [1, 2, 3, 4, 5, 6, 7], d =2 and n = 7 then, A = [1, 2] and B = [3, 4, 5, 6, 7]
Reverse A, we get ArB = [2, 1, 3, 4, 5, 6, 7], Reverse B, we get ArBr = [2, 1, 7, 6, 5, 4, 3]
Reverse all, we get (ArBr)r = [3, 4, 5, 6, 7, 1, 2]

```
def rotateList(A, K):
    n = K % len(A)
    word = A[::-1]    #Reverses the list
    return A[n:] + word[len(A)-n:]
A= [7,3,2,3,3,6,3]
print A, rotateList(A, 3)
```

Problem-4 Suppose you are given an array s[1...n] and a procedure reverse (s,i,j) which reverses the order of elements in between positions i and j (both inclusive). What does the following sequence do, where 1 < k <= n:

```
reverse (s, 1, k);
reverse (s, k + 1, n);
reverse (s, 1, n);
```

(a) Rotates s left by k positions (b) Leaves s unchanged (c) Reverses all elements of s (d) None of the above

Solution: (b). Effect of the above 3 reversals for any k is equivalent to left rotation of the array of size n by k [refer Problem-3].

Problem-5 Finding Anagrams in Dictionary: you are given these 2 files: dictionary.txt and jumbles.txt The jumbles.txt file contains a bunch of scrambled words. Your job is to print out those jumbles words, 1 word to a line. After each jumbled word, print a list of real dictionary words that could be formed by unscrambling the jumbled word. The dictionary words that you have to choose from are in the dictionary.txt file. Sample content of jumbles.:

```
nwae: wean anew wane
eslyep: sleepy
rpeoims: semipro imposer promise
ettniner: renitent
ahicryrhe: hierarchy
dica: acid cadi caid
dobol: blood
 ......
%
```

Solution: Step-By-Step

Step 1: Initialization

- Open the dictionary.txt file and read the words into an array (before going further verify by echoing out the words back from the array out to the screen).
- Declare a hash table variable.

Step 2: Process the Dictionary for each dictionary word in the array. Do the following:
We now have a hash table where each key is the sorted form of a dictionary word and the value associated to it is a string or array of dictionary words that sort to that same key.

- Remove the newline off the end of each word via chomp($word);
- Make a sorted copy of the word - i.e. rearrange the individual chars in the string to be sorted alphabetically
- Think of the sorted word as the key value and think of the set of all dictionary words that sort to the exact same key word as being the value of the key
- Query the hashtable to see if the sortedWord is already one of the keys
- If it is not already present then insert the sorted word as key and the unsorted original of the word as the value
- Else concat the unsorted word onto the value string already out there (put a space in between)

Step 3: Process the jumbled word file

- Read through the jumbled word file one word at a time. As you read each jumbled word chomp it and make a sorted copy (the sorted copy is your key)

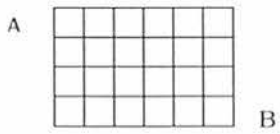
- Print the unsorted jumble word
- Query the hashtable for the sorted copy. If found, print the associated value on same line as key and then a new line.

Step 4: Celebrate, we are all done

Sample code in Perl:

```
open("MYFILE",<dictionary.txt>);          #step 1
while(<MYFILE>){
   $row = $_;
   chomp($row);
   push(@words,$row);
}
my %hashdic = ();
#step 2
foreach $words(@words){
   @not_sorted=split (//, $words);
    @sorted = sort (@not_sorted);
   $name=join("",@sorted);
    if (exists $hashdic{$name})    {
      $hashdic{$name}.=" $words";
   }
   else {
      $hashdic{$name}=$words;
   }
}
$size=keys %hashdic;
#step 3
open("jumbled",<jumbles.txt>);
while(<jumbled>){
   $jum = $_;
   chomp($jum);
  @not_sorted1=split (//, $jum);
   @sorted1 = sort(@not_sorted1);
   $name1=join("",@sorted1);
   if(length($hashdic{$name1})<1)  {
      print "\n$jum : NO MATCHES";
   }
   else  {
      @value=split(/ /,$hashdic{$name1});
      print "\n$jum : @values";
   }
}
```

Problem-6 **Pathways:** Given a matrix as shown below, calculate the number of ways for reaching destination *B* from *A*.



Solution: Before finding the solution, we try to understand the problem with a simpler version. The smallest problem that we can consider is the number of possible routes in a 1×1 grid.

From the above figure, it can be seen that:

- From both the bottom-left and the top-right corners there's only one possible route to the destination.

- From the top-left corner there are trivially two possible routes.

Similarly, for $2x2$ and $3x3$ grids, we can fill the matrix as:

0	1
1	**2**

0	1	1
1	2	3
1	3	**6**

From the above discussion, it is clear that to reach the bottom right corner from left top corner, the paths are overlapping. As unique paths could overlap at certain points (grid cells), we could try to alter the previous algorithm, as a way to avoid following the same path again. If we start filling $4x4$ and $5x5$, we can easily figure out the solution based on our childhood mathematics concepts.

0	1	1	1
1	2	3	4
1	3	6	10
1	4	10	**20**

0	1	1	1	1
1	2	3	4	5
1	3	6	10	15
1	4	10	20	35
1	5	15	35	**70**

Are you able to figure out the pattern? It is the same as *Pascals* triangle. So, to find the number of ways, we can simply scan through the table and keep counting them while we move from left to right and top to bottom (starting with left-top). We can even solve this problem with mathematical equation of *Pascals* triangle.

Problem-7 Given a string that has a set of words and spaces, write a program to move the spaces to *front* of string. You need to traverse the array only once and you need to adjust the string in place.
Input = "move these spaces to beginning" *Output* = " movethesepacestobeginning"

Solution: Maintain two indices i and j; traverse from end to beginning. If the current index contains char, swap chars in index i with index j. This will move all the spaces to beginning of the array.

```
def moveSpacesToBegin(A):
    i=len(A)-1
    datalist = list(A)              # strings atr immutable. Covert it to list
    j=i
    for j in range(i,-1, -1):
        if(not datalist[j].isspace()):
            temp=datalist[i]
            datalist[i]=datalist[j]
            datalist[j]=temp
            i -= 1
    A = ''.join(datalist)
    return A
A = "move these spaces to beginning"
print A, "\n", moveSpacesToBegin(A)
```

Time Complexity: $O(n)$ where n is the number of characters in the input array. Space Complexity: O(1).

Problem-8 For the Problem-7, can we improve the complexity?

Solution: We can avoid a swap operation with a simple counter. But, it does not reduce the overall complexity.

```
def moveSpacesToBegin(A):
    n=len(A)-1
    datalist = list(A)
    count=i = n
    for j in range(i,0, -1):
            if(not datalist[j].isspace()):
                    datalist[count]= datalist[j]
                    count -= 1

    while(count>=0):
            datalist[count]=' '
            count -= 1
    A = ''.join(datalist)
    return A
A = "move these spaces to beginning"
print A, "\n", moveSpacesToBegin(A)
```

Time Complexity: $O(n)$ where n is the number of characters in input array. Space Complexity: O(1).

Problem-9 Given a string that has a set of words and spaces, write a program to move the spaces to *end* of string. You need to traverse the array only once and you need to adjust the string in place.
Input = "move these spaces to end" *Output* = "movethesepacestoend "

Solution: Traverse the array from left to right. While traversing, maintain a counter for non-space elements in array. For every non-space character A[*i*], put the element at A[*count*] and increment *count*. After complete traversal, all non-space elements have already been shifted to front end and *count* is set as index of first 0. Now, all we need to do is run a loop which fills all elements with spaces from *count* till end of the array.

```
def moveSpacesToEnd(A):
    n=len(A)-1
    datalist = list(A)
    count=i = 0
    for i in range(i,n):
        if(not datalist[i].isspace()):
            datalist[count]= datalist[i]
            count += 1
    while(count<=n):
        datalist[count]=' '
        count += 1
    A = ''.join(datalist)
    return A
A = "move these spaces to beginning"
print A, "\n", moveSpacesToEnd(A)
```

Time Complexity: O(*n*) where *n* is number of characters in input array. Space Complexity: O(1).

Problem-10 Moving Zeros to end: Given an array of *n* integers, move all the zeros of a given array to the end of the array. For example, if the given array is {1, 9, 8, 4, 0, 0, 2, 7, 0, 6, 0}, it should be changed to {1, 9, 8, 4, 2, 7, 6, 0, 0, 0, 0}. The order of all other elements should be same.

Solution: Maintain two variables *i* and *j*; and initialize with 0. For each of the array element *A*[*i*], if *A*[*i*] non-zero element, then replace the element *A*[*j*] with element *A*[*i*]. Variable *i* will always be incremented till *n* - 1 but we will increment *j* only when the element pointed by *i* is non-zero.

```
def moveZerosToEnd(A):
    i=j=0
    while (i <= len(A) - 1):
        if (A[i] != 0):
            A[j] = A[i]
            j += 1
        i += 1
    while (j <= len(A) - 1):
        A[j] = 0
        j += 1
    return A
A= [7,0,0,3,0,2,3,3,6,3]
print A,"\n", moveZerosToEnd(A)
```

Time Complexity: O(*n*). Space Complexity: O(1).

Problem-11 For Problem-10, can we improve the complexity?

Solution: Using simple swap technique we can avoid the unnecessary second *while* loop from the above code.

```
def mySwap(A, i, j):
    temp=A[i];A[i]=A[j];A[j]=temp
def moveZerosToEnd2(A):
    i=j=0
    while (i <= len(A) - 1):
        if (A[i] !=0):
            mySwap(A,j,i)
            j += 1
        i += 1
    return A
A= [7,0,0,3,0,2,3,3,6,3]
print A,"\n", moveZerosToEnd2(A)
```

Time Complexity: O(n). Space Complexity: O(1).

Problem-12 Variant of Problem-10 and Problem-11: Given an array containing negative and positive numbers; give an algorithm for separating positive and negative numbers in it. Also, maintain the relative order of positive and negative numbers. Input: -5, 3, 2, -1, 4, -8 Output: -5 -1 -8 3 4 2

Solution: In the *moveZerosToEnd* function, just replace the condition $A[i]$!=0 with $A[i] < 0$.

Problem-13 Given a number represented as an array of digits, plus one to the number.

Solution:

```
from __future__ import division
import random
def plus_one(digits):
    print digits, '+ 1 =',
    carry = 1
    for i in reversed(xrange(len(digits))):
        x = digits[i]
        carry, x = divmod(x+carry, 10)
        digits[i] = x
    if carry > 0: digits.insert(0,carry)
    print digits
    return digits
if __name__ == '__main__':
    plus_one([1,2,3,4])
    plus_one([1,9,9])
    plus_one([9,9,9])
    plus_one([0])
```

Problem-14 Give a shuffle algorithm for an array.

Solution: The Fisher–Yates shuffle algorithm was described by Ronald A. Fisher and Frank Yates in their book Statistical tables for biological, agricultural and medical research. The basic method given for generating a random permutation of the numbers 1 through n goes as follows:

1. Write down the numbers from 1 through n.
2. Pick a random number k between one and the number of unstruck numbers remaining (inclusive).
3. Counting from the low end, strike out the kth number not yet struck out, and write it down elsewhere.
4. Repeat from step 2 until all the numbers have been struck out.
5. The sequence of numbers written down in step 3 is now a random permutation of the original numbers.

The algorithm described by Durstenfeld differs from that given by Fisher and Yates in a small but significant way. Whereas a naïve computer implementation of Fisher and Yates' method would spend needless time counting the remaining numbers in step 3 above, Durstenfeld's solution is to move the *struck* numbers to the end of the list by swapping them with the last unstruck number at each iteration. This reduces the algorithm's time complexity to O(n), compared to O(n^2) for the naïve implementation.

```
import random
def shuffleArray(A):
    n = len(A)
    i = n - 1
    while i>0:
            j = int((random.random())% (i+1))
            tmp = A[j-1];A[j-1] = A[j]; A[j] = tmp
            i -= 1
A = [1,3,5,6,2,4,6,8]
shuffleArray(A)
print A
```