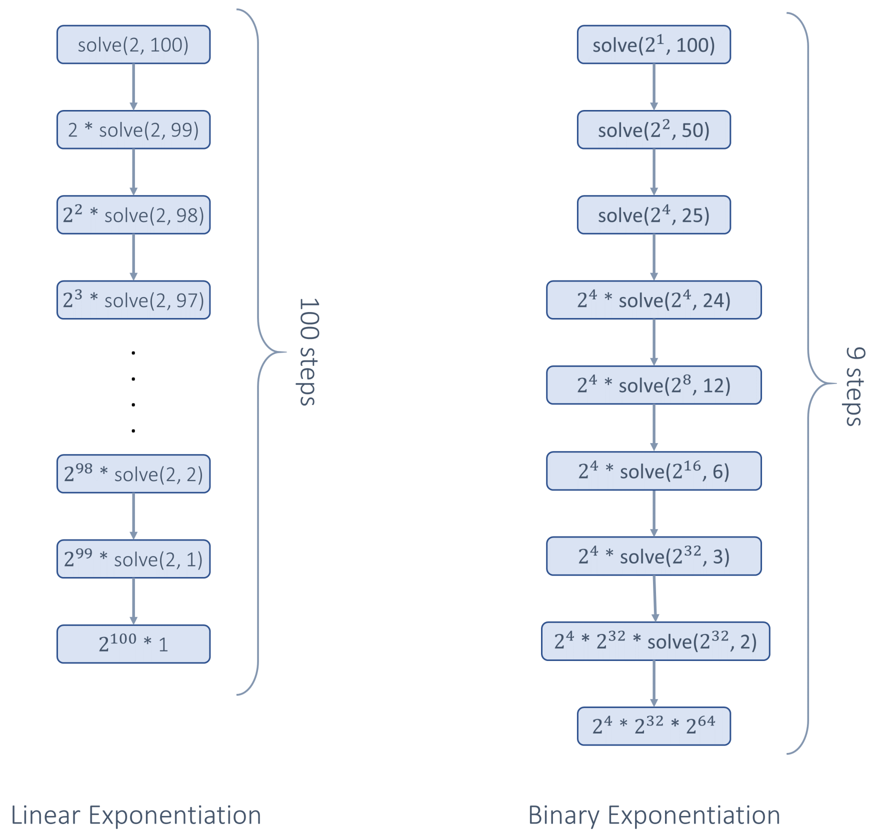
Recursion_1

@kbbhatt04

@August 12, 2023

pow(x, n)

- Approach
    - Brute-force
        - Recursively/Iteratively multiply x, n times
        - Time Complexity: $O(n)$
        - Space Complexity: $O(n)$ for recursive
        - Space Complexity: $O(1)$ for iterative
    - Optimal
        - Fast Exponentiation
            - $(x^2)^{n/2}$ if n is even
            - $x * (x^2)^{(n-1)/2}$ if n is odd



- Time Complexity: $O(logn)$
- Space Complexity: $O(logn)$ for recursive
- Space Complexity: $O(1)$ for iterative

```
# Python3
# Brute-force Solution
class Solution:
    def myPow(self, x: float, n: int) -> float:
        if n == 0:
            return 1
        if n < 0:
            return 1 / self.myPow(x, -n)
        return x * self.myPow(x, n-1)
```

```
# Python3
# Optimal Solution
class Solution:
    def myPow(self, x: float, n: int) -> float:
        if n == 0:
            return 1
        if n < 0:
            return 1 / self.myPow(x, -n)

        if n & 1:
            return x * self.myPow(x, n-1)
        else:
            return self.myPow(x * x, n // 2)
```

```
# Python3
# Optimal Solution
# Iterative
class Solution:
    def binaryExp(self, x: float, n: int) -> float:
        if n == 0:
            return 1

        # Handle case where, n < 0.
        if n < 0:
            n = -1 * n
            x = 1.0 / x

        # Perform Binary Exponentiation.
        result = 1
        while n != 0:
            # If 'n' is odd we multiply result with 'x' and redu
            if n % 2 == 1:
                result *= x
                n -= 1
```

```
            # We square 'x' and reduce 'n' by half, x^n => (x^2)
            x *= x
            n //= 2
        return result

    def myPow(self, x: float, n: int) -> float:
        return self.binaryExp(x, n)
```

```
// C++
// Optimal Solution
// Recursive
class Solution {
public:
        // in question type(n) is given int, I converted it to l
        // as it was failing in a testcase where given n = INT_M
        // and when I did "-n", it was going out of int range (i
        // as range of int is [-2^31, 2^31 - 1]
    double myPow(double x, long long n) {
        if (n == 0) {return 1;}
        if (n < 0) {
            return 1.0 / myPow(x, -n);
        }
        if (n & 1) {
            return x * myPow(x*x, (n-1)/2);
        }
        else {
            return myPow(x*x, n>>1);
        }
    }
};
```

```
// C++
// Optimal Solution
// Iterative
```

```
class Solution {
public:
    double binaryExp(double x, long long n) {
        if (n == 0) {
            return 1;
        }

        // Handle case where, n < 0.
        if (n < 0) {
            n = -1 * n;
            x = 1.0 / x;
        }

        // Perform Binary Exponentiation.
        double result = 1;
        while (n) {
            // If 'n' is odd we multiply result with 'x' and red
            if (n % 2 == 1) {
                result = result * x;
                n -= 1;
            }
            // We square 'x' and reduce 'n' by half, x^n => (x^2
            x = x * x;
            n = n / 2;
        }
        return result;
    }

    double myPow(double x, int n) {
                // this conversion of int n to long long n is ne
                // else it will fail when we do -n as it will go
                // as int range is [-2^31, 2^31 - 1]
        return binaryExp(x, (long long) n);
    }
};
```

Count Good Numbers

A digit string is **good** if the digits **(0-indexed)** at **even** indices are **even** and the digits at **odd** indices are **prime** (`2`, `3`, `5`, or `7`).

- For example, `"2582"` is good because the digits (`2` and `8`) at even positions are even and the digits (`5` and `2`) at odd positions are prime. However, `"3245"` is **not** good because `3` is at an even index but is not even.

Given an integer `n`, return *the **total** number of good digit strings of length* `n`. Since the answer may be large, **return it modulo** `10**9 + 7`.

A **digit string** consists of digits `0` through `9` that may contain leading zeros.

- Approach
    - Optimal
        - Now, we know we have 4 primes = {2, 3, 5, 7} and 5 evens = {0, 2, 4, 6, 8}
        - if `index == 0`, then there can be any of one evens at even position, ans = 5
        - if `index == 1`, then there was 1 even at index = 0, and at this odd index there can be one of 4 primes, therefore ans = 5∗4
        - if `index == 2`, then at this even index there can be one of 5 evens, ans = (5∗4)*5
        - so, continuing the pattern we can see, it's like, `5*4*5*4*5*4*5..... ans so on`
        - here no. of 4s = no. of odd positions = n/2
        - no. of 5s = no. of even positions = (n-n/2)
        - Thus `ans = pow(4,count4) * pow(5,count5)`
        - Time Complexity: $O(logn)$
        - Space Complexity: $O(n)$ for recursive
        - Space Complexity: $O(1)$ for iterative

```python
# Python3
# Optimal Solution
class Solution:
    def myPow(self, x: float, n: int) -> float:
        if n == 0:
            return 1
        if n & 1:
            return x * self.myPow(x, n-1) % 1000000007
        else:
            return self.myPow(x * x % 1000000007, n // 2)


    def countGoodNumbers(self, n: int) -> int:
        fours = n//2
        fives = n - fours

        a = self.myPow(4, fours) % 1000000007
        b = self.myPow(5, fives) % 1000000007

        ans = a * b % 1000000007
        return ans
```

```cpp
// C++
// Optimal Solution
#define ll long long

class Solution {
public:
    // evens  = 0, 2, 4, 6, 8  => 5 evens
    // primes = 2, 3, 5, 7     => 4 primes

    int p = 1e9 + 7;

    // calculating x^y efficeiently
```

```
    ll power(long long x, long long y) {
      long long res = 1;

      x = x % p; // Update x if it is more than or equal to p
      if (x == 0) return 0;

      while (y > 0) {
        // If y is odd, multiply x with result
        if (y & 1) res = (res*x) % p;

        // we have did the odd step is it was odd, now do the ev
        y = y>>1; // dividing y by 2, y>>1 is same as y/2
        x = (x*x) % p;
      }
      return res;
    }

    int countGoodNumbers(long long n) {
      ll count_of_4s = n/2;
      ll count_of_5s = n - count_of_4s;
      ll ans = ((power(4LL, count_of_4s) % p  * power(5LL, count
      return (int)ans;
    }
};
```

Sort a Stack

- Approach
  - Optimal
    - The idea of the solution is to hold all values in Function Call Stack until the stack becomes empty
    - When the stack becomes empty, insert all held items one by one in sorted order
    - Time Complexity: $O(n^2)$

- Space Complexity: $O(n)$

```python
# Python3
# Optimal Solution
def sortedInsert(s, element):

    # Base case: Either stack is empty or newly inserted
    # item is greater than top (more than all existing)
    if len(s) == 0 or element > s[-1]:
        s.append(element)
        return
    else:

        # Remove the top item and recur
        temp = s.pop()
        sortedInsert(s, element)

        # Put back the top item removed earlier
        s.append(temp)

def sortStack(s):

    # If stack is not empty
    if len(s) != 0:

        # Remove the top item
        temp = s.pop()

        # Sort remaining stack
        sortStack(s)

        # Push the top item back in sorted stack
        sortedInsert(s, temp)
```

```
// C++
// Optimal Solution
void sortStack(stack<int> &s) {
    // If the stack is empty, return
    if (s.empty())
        return;

    // Remove the top element of the stack
    int x = s.top();
    s.pop();

    // Sort the remaining elements in the stack using recursion
    sortStack(s);

    // Create two auxiliary stacks
    stack<int> tempStack;

    // Move all elements that are greater than x from main stack
    while (!s.empty() && s.top() > x) {
        tempStack.push(s.top());
        s.pop();
    }

    // Push x back into the main stack
    s.push(x);

    // Move all elements from tempStack back to the main stack
    while (!tempStack.empty()) {
        s.push(tempStack.top());
        tempStack.pop();
    }
}
```

Reverse a Stack

- Approach
    - Optimal
        - The idea of the solution is to hold all values in Function Call Stack until the stack becomes empty
        - When the stack becomes empty, insert all held items one by one in sorted order
        - Time Complexity: $O(n^2)$
        - Space Complexity: $O(n)$

```
# Python3
# Optimal Solution
def insertAtBottom(stack, item):
    if isEmpty(stack):
        push(stack, item)
    else:
        temp = pop(stack)
        insertAtBottom(stack, item)
        push(stack, temp)

def reverse(stack):
    if not isEmpty(stack):
        temp = pop(stack)
        reverse(stack)
        insertAtBottom(stack, temp)
```

```
// C++
// Optimal Solution
void insert_at_bottom(stack<int>& st, int x)
{
    if (st.size() == 0) {
        st.push(x);
    }
    else {
```

```
        // All items are held in Function Call
        // Stack until we reach end of the stack
        // When the stack becomes empty, the
        // st.size() becomes 0, the above if
        // part is executed and the item is
        // inserted at the bottom

        int a = st.top();
        st.pop();
        insert_at_bottom(st, x);

        // push allthe items held in
        // Function Call Stack
        // once the item is inserted
        // at the bottom
        st.push(a);
    }
}

void reverse(stack<int>& st)
{
    if (st.size() > 0) {
        // Hold all items in Function
        // Call Stack until we
        // reach end of the stack
        int x = st.top();
        st.pop();
        reverse(st);

        // Insert all the items held
        // in Function Call Stack
        // one by one from the bottom
        // to top. Every item is
        // inserted at the bottom
        insert_at_bottom(st, x);
    }
```

```
    return;
}
```

Generate all Binary Strings of length N with no consecutive 1's

- Approach
    - Brute-force
        - Generate all Binary Strings
        - Iterate through all and remove the strings that has consecutive 1's
        - Time Complexity: $O(n * 2^n)$
        - Space Complexity: $O(n)$
    - Optimal
        - Generate Binary String such that if string[-1] == '1', then next char can only be '0'
        - But if string[-1] == '0', then next char could be any
        - Time Complexity: $O(2^n)$
        - Space Complexity: $O(n)$

```
# Python3
# Brute-force Solution
def generateString(N: int) -> List[str]:
    temp = []

    def rec(n, i, curr):
        nonlocal temp
        if i == n:
            temp += curr,
            return

        rec(n, i+1, curr+"0")
        rec(n, i+1, curr+"1")
```

```
    rec(N, 1, "0")
    rec(N, 1, "1")

    def check(s):
        for i in range(len(s)-1):
            if s[i] == "1" and s[i+1] == "1":
                return True
        return False

    ans = []
    for i in temp:
        if check(i):
            continue
        ans += i,

    ans.sort()
    return ans
```

```
# Python3
# Optimal Solution
def generateString(N: int) -> List[str]:
    ans = []

    def rec(n, i, curr):
        nonlocal ans
        if i == n:
            ans += curr,
            return

        if curr[-1] != "1":
            rec(n, i+1, curr+"1")
        rec(n, i+1, curr+"0")

    rec(N, 1, "0")
```

```
    rec(N, 1, "1")

    ans.sort()
    return ans
```

```
// C++
// Optimal Solution
void rec(int n, int i, string curr, vector<string>& ans) {
    if (curr.size() == n) {ans.push_back(curr); return;}

    if (curr.back() != '1') {
        rec(n, i+1, curr+"1", ans);
    }
    rec(n, i+1, curr+"0", ans);
}

vector<string> generateString(int N) {
    vector<string> ans;
    rec(N, 1, "0", ans);
    rec(N, 1, "1", ans);
    sort(ans.begin(), ans.end());
    return ans;
}
```

Generate Parenthesis

- Approach
    - Optimal
        - The idea is to add `')'` only after valid `'('`
        - We use two integer variables `left` & `right` to see how many `'('` & `')'` are in the current string
        - If `left < n` then we can add `'('` to the current string
        - If `right < left` then we can add `')'` to the current string

For `n = 2`, the recursion tree will be something like this,

```
                    (0, 0, '')
                         |
                    (1, 0, '(')
                   /           \
           (2, 0, '((')      (1, 1, '()')
               /                  \
       (2, 1, '(()')           (2, 1, '()(')
           /                        \
   (2, 2, '(())')                (2, 2, '()()')
              |                             |
   res.append('(())')            res.append('()()')
```

- Time Complexity: $O(2^n)$
- Space Complexity: $O(n)$

```
# Python3
# Optimal Solution
# Iterative Solution
class Solution:
    def generateParenthesis(self, n: int) -> List[str]:
        result = []
        left = right = 0
        q = [(left, right, '')]
        while q:
            left, right, s = q.pop()
            if len(s) == 2 * n:
                result.append(s)
            if left < n:
                q.append((left + 1, right, s + '('))
            if right < left:
                q.append((left, right + 1, s + ')'))
        return result
```

```
# Python3
# Optimal Solution
# Recursive Solution
class Solution:
    def generateParenthesis(self, n: int) -> List[str]:
        ans = []

        def rec(n, left, right, curr):
            nonlocal ans
            if len(curr) == n * 2:
                ans += curr,
                return

            if left < n:
                rec(n, left+1, right, curr+"(")
            if right < left:
                rec(n, left, right+1, curr+")")

        rec(n, 0, 0, "")
        return ans
```

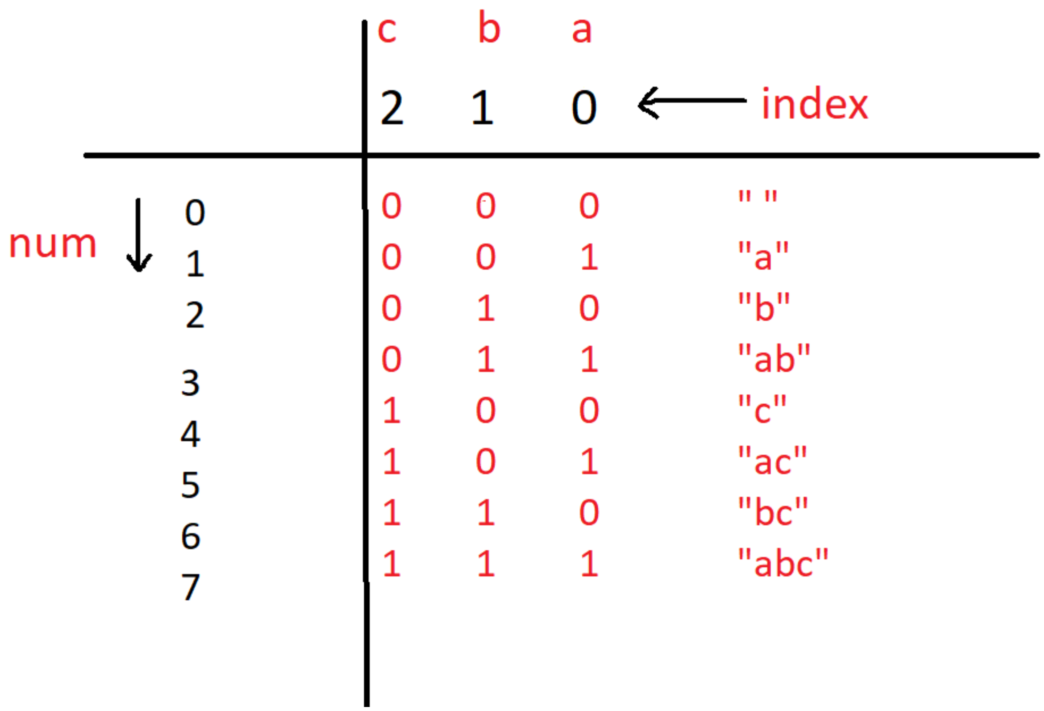
```
// C++
// Optimal Solution
// Recursive Solution
class Solution {
public:
    vector<string>result;

    void helper(int open,int close,int n,string current)
    {
        if(current.length()==n*2)
        {
            result.push_back(current);
            return;
```

```
        }
        if(open<n)  helper(open+1,close,n,current+"(");
        if(close<open)  helper(open,close+1,n,current+")");
    }
    vector<string> generateParenthesis(int n) {
        helper(0,0,n,"");
        return result;
    }
};
```

Generate all Subsequences

- Approach
  - Brute-force
    - Using bit manipulation
    - If n&(1<<i) != 0,then the ith bit is set
    - Traverse from 0 to 2^(n)-1 and check for every number if their bit is set or not. If the bit is set add that character to your subsequence



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

- Optimal
    - Since we are generating subsets two cases will be possible, either you can pick the character or you cannot pick the character and move to the next character
    - Maintain a temp string (say f), which is empty initially
    - Now you have two options, either you can pick the character or not pick the character and move to the next index
    - Firstly we pick the character at ith index and then move to the next index (f + s[i])
    - If the base condition is hit, i.e. `i == s.length()`, then we print the temp string and return
    - Now while backtracking we have to pop the last character since now we have
      to implement the non-pick condition and then move to the next index.
    - Time Complexity: $O(2^n)$
    - Space Complexity: $O(2^n)$

```
# Python3
# Brute-force Solution
def func(s):
    n = len(s)
    ans = []
    for i in range(1 << n):
        temp = ""
        for j in range(n):
            if i & (1 << j):
                temp += s[j]

        if len(temp):
```

```
            ans += temp,
    ans.sort()
    return ans
```

```
// C++
// Brute-force Solution
vector<string> AllPossibleStrings(string s) {
    int n = s.length();
    vector<string>ans;
    for (int num = 0; num < (1 << n); num++) {
        string sub = "";
        for (int i = 0; i < n; i++) {
            // check if the ith bit is set or not
            if (num & (1 << i)) {
                sub += s[i];
            }
        }
        if (sub.length() > 0) {
            ans.push_back(sub);
        }
    }
    sort(ans.begin(), ans.end());
    return ans;
}
```

```
# Python3
# Optimal Solution
class Solution:
    def subsets(self, nums: List[int]) -> List[List[int]]:
        ans = list()
        temp = list()

        def allSubsets(index, nums, nums_size, temp, ans):
            if(index == nums_size):
```

```
                ans.append(temp[:])
                return
            temp.append(nums[index])
            allSubsets(index+1, nums, nums_size, temp, ans)
            temp.pop()
            allSubsets(index+1, nums, nums_size, temp, ans)

        allSubsets(0, nums, len(nums), temp, ans)
        return ans
```

```
// C++
// Optimal Solution
void solve(int i, string s, string &f) {
    if (i == s.length()) {
        cout << f << " ";
        return;
    }
    //picking
    f = f + s[i];
    solve(i + 1, s,  f);
    //poping out while backtracking
    f.pop_back();
    solve(i + 1, s,  f);
}
```

Print all Sub-sequences with Sum K

- Approach
    - Brute-force
        - Generate all sub-sequences
        - Iterate through all and calculate the sum
        - Time Complexity: $O(n * 2^n) + O(n * 2^n)$ for generating all sub-sequences and iterating over all to calculate sum

- Space Complexity: $O(n * 2^n)$

- Optimal
    - Since we are generating subsets two cases will be possible, either you can pick the element or you dont pick the element and move to the next index
    - Maintain a temp data structure, which is empty initially
    - Now you have two options, either you can pick the element or not pick the element and move to the next index
    - Firstly we pick the element at ith index and then move to the next index
    - If the base condition is hit, i.e. `i == nums.length()`, and then if `current_sum == K`, we print the temp data structure and return
    - Now while backtracking we have to pop the last element taken since now we have
      to implement the non-pick condition and then move to the next index.
    - Time Complexity: $O(n * 2^n)$
    - Space Complexity: $O(n)$

```
# Python3
# Brute-force Solution
def func(s):
    n = len(s)
    ans = []
    for i in range(1 << n):
        temp = []
        for j in range(n):
            if i & (1 << j):
                temp += s[j],

        if len(temp):
            ans += temp,
    return ans
```

```
s = [1,2,2,2,3,3,4,5,6]
K = 6
ans = func(s)
for i in ans:
    if sum(i) == K:
        print(i)
```

```
# Python3
# Optimal Solution
def func(nums, temp_ds, ind, K, temp_sum):
    if ind == len(nums):
        if temp_sum == K:
            print(temp_ds)
        return

    # take current element
    temp_ds.append(nums[ind])
    temp_sum += nums[ind]
    func(nums, temp_ds, ind+1, K, temp_sum)
    temp_sum -= nums[ind]
    temp_ds.pop()

    # not take current element
    func(nums, temp_ds, ind+1, K, temp_sum)
```

```
// C++
// Optimal Solution
void func(int ind, vector<int>& v, int arr[], int arr_len, int K
{
    if(ind == arr_len){
     if(temp_sum == K){
         for(auto x: v){
             cout << x << " ";
         }
```

```
        cout << endl;
      }
      return;
    }
    // take current element
    v.push_back(arr[ind]);
    temp_sum += arr[ind];
    func(ind+1, v, arr, arr_len, K, temp_sum);
    temp_sum -= arr[ind];
    v.pop_back();

    // not take current element
    func(ind+1, v, arr, arr_len, K, temp_sum);
}
```

Print any 1 Sub-sequence with sum K

- Approach
    - Optimal
        - Same as Print all Sub-Sequences with Sum K
        - Just return true when a sub-sequence with sum K is found else return false
        - Time Complexity: $O(2^n)$
        - Space Complexity: $O(n)$

```
# Python3
# Optimal Solution
def func(nums, temp_ds, ind, K, temp_sum):
    if ind == len(nums):
        if temp_sum == K:
            print(temp_ds)
            return True
        return False
```

```
    # take current element
    temp_ds.append(nums[ind])
    temp_sum += nums[ind]
    if func(nums, temp_ds, ind+1, K, temp_sum):
        return True
    temp_sum -= nums[ind]
    temp_ds.pop()

    # not take current element
    if func(nums, temp_ds, ind+1, K, temp_sum):
        return True

    return False
```

```
// C++
// Optimal Solution
void func(int ind, vector<int>& v, int arr[], int arr_len, int K
{
    if(ind == arr_len){
     if(temp_sum == K){
         for(auto x: v){
             cout << x << " ";
         }
         cout << endl;
         return true;
     }
     return false;
    }
    // take current element
    v.push_back(arr[ind]);
    temp_sum += arr[ind];
    if(func(ind+1, v, arr, arr_len, K, temp_sum))
                return true;
    temp_sum -= arr[ind];
```

```
    v.pop_back();

    // not take current element
    if(func(ind+1, v, arr, arr_len, K, temp_sum))
            return true;

    return false;
}
```

Count Sub-sequences with Sum K

- Approach
    - Brute-force
        - Generate all sub-sequences
        - Count sub-sequences having sum K
        - Time Complexity: $O(n * 2^n) + O(n * 2^n)$ for generating all sub-sequences and counting all sub-sequences with sum K
        - Space Complexity: $O(n * 2^n)$
    - Optimal
        - Same as Print all Sub-Sequences with Sum K
        - Just return 1 when sub-sequence having K is found, else return 0
        - Time Complexity: $O(2^n)$
        - Space Complexity: $O(1)$

```
# Python3
# Brute-force Solution
def func(s):
    n = len(s)
    ans = []
    for i in range(1 << n):
        temp = []
```

```
        for j in range(n):
            if i & (1 << j):
                temp += s[j],

        if len(temp):
            ans += temp,
    return ans

s = [1,2,2,2,3,3,4,5,6]
K = 6
ans = func(s)
count = 0
for i in ans:
    if sum(i) == K:
        count += 1
print(count)
```

```
# Python3
# Optimal Solution
def func(nums, temp_ds, ind, K, temp_sum):
    if ind == len(nums):
        if temp_sum == K:
            return 1
        return 0

    # take current element
    temp_ds.append(nums[ind])
    temp_sum += nums[ind]
    l = func(nums, temp_ds, ind+1, K, temp_sum)
    temp_sum -= nums[ind]
    temp_ds.pop()

    # not take current element
    r = func(nums, temp_ds, ind+1, K, temp_sum)
```

```
    return l + r
```

```
// C++
// Optimal Solution
void func(int ind, int arr[], int arr_size, int K, int temp_sum
    if(ind == arr_size){
     if(temp_sum == K)    return 1;
     return 0;
    }
    // take current element
    temp_sum += arr[ind];
    int l = func(ind+1, arr, arr_size, K, temp_sum);
    temp_sum -= arr[ind];

    // not take current element
    int r = func(ind+1, arr, arr_size, K, temp_sum);

    return l + r;
}
```

Template

- Approach
    - Brute-force
        -
        - Time Complexity: $O(n^3)$
        - Space Complexity: $O(1)$
    - Better
        -
        - Time Complexity: $O(n^3)$

        - Space Complexity: $O(1)$
    - Optimal
        -
        - Time Complexity: $O(n^3)$
        - Space Complexity: $O(1)$

```python
# Python3
# Brute-force Solution
```

```python
# Python3
# Better Solution
```

```python
# Python3
# Optimal Solution
```

```cpp
// C++
// Optimal Solution
```