Technique: Permutations/Combinations using Auxiliary Buffer

Level: Medium

Given an array of integers, print all combinations of size X.

Questions to Clarify:
Q. Can the array have duplicates?
A. No, you can assume there are no duplicate numbers.

Q. What to print if X is greater than the size of the array?
A. Print nothing, as there will be no valid combinations.

Q. What to print if the array is empty?
A. Print nothing as there will be no combinations.

Solution:
We use a buffer of size X with a recursive function *"printCombosHelper"*.
In any recursive call to *printCombosHelper()*, the buffer is filled up to a certain index *i-1*, and the task for the function call is to fill index *i*. If *i* is greater than the size of the buffer, then the buffer is full, and we print its contents.

Otherwise, we find the candidates from the input array that can go into index *i*. We place each candidate into index *i* and then call *printCombosHelper()* for *i+1*.

Pseudocode:

```
printCombos(a, X)
     buffer = new array of size X
     printCombos(a, buffer, 0, 0)

printCombosHelper(a, buffer, startIndex, bufferIndex):
      if buffer is full:
         print buffer
         return

      if startIndex is out of bounds:
         return

      for i: startIndex to a.length-1:
         place a[i] into buffer[bufferIndex]
         printCombos(a, buffer, i + 1, bufferIndex + 1)
```

Test Cases:
Edge Cases: a is empty/null, X is 0, X is greater than a.length
Base Cases: a is of size 1 and 2, X is 1
Regular Cases: X is smaller than a.length, X is equal to a.length

Time Complexity: Factorial Complexity

Note: Typically, interviewers will not ask the time complexity of such complex recursions.

However, from combinatorics, we know that there are ${}^{n}C_{r}$ possible combinations, which is in factorials. We need that many leaves of the recursion tree.

Here is another way to look at this. We can calculate the number of nodes in the recursion tree. In our algorithm, we generate every combination of size 2, and then add another index to generate combinations of size 3. So the number of nodes in the recursion tree are:

$${}^{n}C_{1} + {}^{n}C_{2} + {}^{n}C_{3} + {}^{n}C_{4} + {}^{n}C_{5} + .. + {}^{n}C_{r}$$

Hence the factorial complexity.

Space Complexity: O(X). We use O(X) space both in the buffer allocation and on the recursion stack.

```
public static void printCombos(int[] a, int x) {
    if (a == null || a.length == 0 || x > a.length)
        return;

    int[] buffer = new int[x];
    printCombosHelper(a, buffer, 0, 0);
}

public static void printCombosHelper(int[] a, int[] buffer, int startIndex,
int bufferIndex) {
    // termination cases - buffer full
    if (bufferIndex == buffer.length) {
        printArray(buffer);
        return;
    }
    if (startIndex == a.length) {
        return;
    }

    // find candidates that go into current buffer index
    for (int i = startIndex; i < a.length; i++) {
        // place item into buffer
        buffer[bufferIndex] = a[i];

        // recurse to next buffer index
        printCombosHelper(a, buffer, i + 1, bufferIndex + 1);
    }
}
```

Technique: Permutations/Combinations using Auxiliary Buffer

Level: Medium

Phone Number Mnemonics: Given an N digit phone number, print all the strings that can be made from that phone number. Since 1 and 0 don't correspond to any characters, ignore them.

For example:
213 => AD, AE, AF, BD, BE, BF, CE, CE, CF
456 => GJM, GJN, GJO, GKM, GKN, GKO,.. etc.

Questions to Clarify:
Q. Is the phone number of a specific size?
A. No, it can be of any size

Q. Can we assume that the input will have only digits?
A. Yes

Q. Does the string have to be a valid English word?
A. No, the string can be anything.

Q. How do we handle if phone number is empty or null?
A. Print nothing.

Solution:
This problem is a straightforward application of Recursion using Buffer. For each buffer index, the candidates would be the letters corresponding to the digit at that index.

For example, in the array "234", the candidates for index 0 are letters that correspond to 2 - A, B, C.

We simply place each candidate in the current buffer index and recurse to the next index.

Pseudocode:

```
printWords(a)
    buffer = new array of size a.length
    printWordsHelper(a, buffer, 0, 0)

printWordsHelper(a, buffer, aIndex, bufferIndex)
    if buffer is full or aIndex is equal to a.length:
        print buffer

    candidates = get letters for a[aIndex]

    if candidates is empty, it must be a 0 or 1:
        continue to next index
```

```
    for each candidate:
        place candidate at buffer[bufferIndex]
        printWordsHelper(a, buffer, aIndex + 1, bufferIndex + 1)
```

Test Cases:
Edge Cases: a is empty/null
Base Cases: a has one number, a has only 1's and 0's
Regular Cases: a with 1's and 0's, without 1's and 0's

Time Complexity: Exponential Complexity - $O(4^n)$, where *n* is the size of the phone number.

At each function call, we can call at most 4 function calls. We do this at most N times, so the total number of function calls is:

4 X 4 X 4 .. (n times), which is 4^n

Space Complexity: O(n), where *n* is the size of the phone number. The O(n) space is taken both by the buffer and the call stack.

```
public static void printWords(int[] phoneNumber) {
    if (phoneNumber == null || phoneNumber.length == 0)
        return;

    char[] buffer = new char[phoneNumber.length];
    printWordsHelper(phoneNumber, buffer, 0, 0);
}

public static void printWordsHelper(int[] a, char[] buffer, int aIndex, int
bufferIndex) {
    // termination case
    if (bufferIndex >= buffer.length || aIndex >= a.length) {
        printArray(buffer, bufferIndex);
        return;
    }

    // find candidates for buffer position
    char[] letters = getLetters(a[aIndex]);

    if (letters.length == 0)
        printWordsHelper(a, buffer, aIndex + 1, bufferIndex);

    // place candidates in buffer
    for (char letter: letters) {
        buffer[bufferIndex] = letter;
        printWordsHelper(a, buffer, aIndex + 1, bufferIndex + 1);
    }
```

```
}

/*
 * Helper Function. Ask interviewer if he/she wants you to implement this.
 */
public static char[] getLetters(int digit) {
    switch(digit) {
        case 0: return new char[]{};
        case 1: return new char[]{};
        case 2: return new char[]{'a', 'b','c'};
        case 3: return new char[]{'d', 'e','f'};
        case 4: return new char[]{'g', 'h','i'};
        case 5: return new char[]{'j', 'k','l'};
        case 6: return new char[]{'m', 'n','o'};
        case 7: return new char[]{'p', 'q','r','s'};
        case 8: return new char[]{'t','u', 'v'};
        case 9: return new char[]{'w','x', 'y', 'z'};
    }
    throw new IllegalArgumentException("Invalid Digit " + digit);
}
```

Technique: Permutations/Combinations using Auxiliary Buffer

Level: Medium

Given an array of integers A, print all its subsets.

For example:
Input: [1, 2, 3]

Output:
[]
[1]
[2]
[3]
[1, 2]
[1, 3]
[2, 3]
[1, 2, 3]

Questions to Clarify:
Q. Do we print out the empty set?
A. Yes

Q. Can we assume there are no duplicates?
A. Yes

Q. If the input is an empty array, do we just print out the empty set?
A. Yes

Q. If the input is null, should we throw an exception of simply print nothing?
A. Let's just print nothing.

Solution:
The solution is to simply modify one line in the "print all combinations of size X" code.

One way to print all subsets is to print all combinations of size 0, then all combinations of size 1, then combinations of size 2, all the way up to A.length.

However, we can simply modify the code that prints combinations of size A.length. How is that? Well, in order to print combinations of size A.length, we first come up with all combinations of size A.length-1, then add characters to the last index. So on the way, we are coming up with all combinations from 0 to A.length.

Instead of just printing when the buffer is full, we print the buffer at every recursive call. That gives us all the subsets.

The one line change: Move the print statement outside the if condition.

Pseudocode:

```
printSubsets(a)
     buffer = new array of size a.length
     printSubsets(a, buffer, 0, 0)

printSubsetsHelper(a, buffer, startIndex, bufferIndex):
      print buffer

      if buffer is full:
         return

      if startIndex is out of bounds:
         return

      for i: startIndex to a.length-1:
         place a[i] into buffer[bufferIndex]
         printSubsetsHelper(a, buffer, i + 1, bufferIndex + 1)
```

Test Cases:
Edge Cases: A is empty/null
Base Cases: A is of size 1 and 2
Regular Cases: A is of size greater than 2

Time Complexity: Factorial Complexity

Note: Typically, interviewers will not ask the time complexity of such complex recursions.

However, if you're still interested, the complexity is described in the "find combinations" problem above.

Space Complexity: O(N), where N is A's length. We use O(N) space both in the buffer allocation and on the recursion stack.

```
public static void printSubsets(int[] a) {
    if (a == null || a.length == 0)
        return;

    int[] buffer = new int[a.length];
    printSubsetsHelper(a, buffer, 0, 0);
}

public static void printSubsetsHelper(int[] a, int[] buffer, int startIndex,
int bufferIndex) {
    printArray(buffer, bufferIndex);

    // termination cases - buffer full
```

```
    if (bufferIndex == buffer.length) {
        return;
    }
    if (startIndex == a.length) {
        return;
    }

    // find candidates that go into current buffer index
    for (int i = startIndex; i < a.length; i++) {
        // place item into buffer
        buffer[bufferIndex] = a[i];

        // recurse to next buffer index
        printSubsetsHelper(a, buffer, i + 1, bufferIndex + 1);
    }
}
```

Technique: Permutations/Combinations using Auxiliary Buffer

Level: Medium

Given an array A, print all permutations of size X.

For example,

Input:
A = [1,2,3]
X = 2

Output:
[1, 2]
[1, 3]
[2, 1]
[2, 3]
[3, 1]
[3, 2]

Questions to Clarify:
Q. Can we assume there are no duplicates?
A. Yes

Q. If the input is an empty array, do we just print nothing?
A. Yes

Q. If the input is null, should we throw an exception of simply print nothing?
A. Let's just print nothing.

Solution:
While computing combinations, we went forward from the previous number. So if we are in the following situation:
X = 3
A = [1,2,3,4,5,6,7],
Buffer = [1,4,_] (index 2 is empty)

We start filling index 2 with 5, 6 and 7.

For permutations, however, we can fill index 2 with any number except 1 and 4. This is because [1,4,2] and [1,2,4] are different permutations.

So, we need to keep track of which numbers are already in our buffer. Our candidates for buffer index 2 will be all the other numbers.

Pseudocode:

```
printPermsHelper(a, buffer, bufferIndex, isInBuffer)
   if buffer is full:
        print buffer
        return

   for i: 0 to a.length-1:
       if (isInBuffer[i] is false) // meaning index i is not in buffer
              place index i in buffer
              mark isInBuffer[i] to true
              // recurse to next level
              printPermsHelper(a, buffer, bufferIndex + 1, isInBuffer)
              mark isInBuffer[i] to false
```

Test Cases:
Edge Cases: a is empty/null, X is 0, X is greater than a.length
Base Cases: a is of size 1 and 2, X is 1
Regular Cases: X is smaller than a.length, X is equal to a.length

Time Complexity: Factorial Complexity

Note: Typically, interviewers will not ask the time complexity of such complex recursions.

The first level of the recursion tree has *n* function calls. At the second level, each function call spawns *n-1* more calls. The total number of calls looks as follows:

n * (n - 1) * (n - 2) * (n - 3) * … * (n-X)

This is factorial time - you can also write it as `n!/(n-X-1)!`

Space Complexity: O(N), where N is A.length. We use O(N) space both in the buffer allocation

```
public static void printPerms(int[] a, int x) {
    if (a == null || a.length == 0 || x > a.length)
        return;

    int[] buffer = new int[x];
    boolean[] isInBuffer = new boolean[a.length];
    printPermsHelper(a, buffer, 0, isInBuffer);
}

public static void printPermsHelper(int[] a, int[] buffer, int bufferIndex,
boolean[] isInBuffer) {
    // termination case - buffer full
    if (bufferIndex == buffer.length) {
```

```
        printArray(buffer);
        return;
    }

    // find candidates that go into current buffer index
    for (int i = 0; i < a.length; i++) {
        if (!isInBuffer[i]) {
            // place candidate into buffer index
            buffer[bufferIndex] = a[i];
            isInBuffer[i] = true;
            // recurse to next buffer index
            printPermsHelper(a, buffer, bufferIndex + 1, isInBuffer);
            isInBuffer[i] = false;
        }
    }
}
```