4-arrays-and-strings

Contents

Introduction

One of the most used structures in programming problems, questions with strings usually revolve around working with string permutations, or checking for sub strings in a given string. Questions with arrays, revolve around traversing arrays, maybe mutating them by shifting elements left or right to make space for new elements or remove existing ones, sorting and searching is often also performed on static or dynamic arrays more often than on structures like linked lists for example.

Strings

Most all languages that implement strings are providing immutable string objects, what that means is that that each mutation on the string produces a brand-new string, which is a copy of the original with the mutation applied. Keep that in mind and if the problem revolves around mutating a String, use the correct data structure for the language, for java that would be StringBuilder.

Permutation of a string

The problem here is usually how to find all permutations of a given string, the solution is usually recursive, the way it works, is by cutting the last or the first character from a string, and generating all permutations for that reduced string. Drill down until the input string is of length 1, then in the post recursive calls, stuff the cut character in the end, mid and start of each permutated string in the list of permutations

```
List<String> permutate(String input) {
    if (input.length() == 1) {
        // a string input with length 1, has no permutations
        return Arrays.asList(input);
    }
    // pull the last element from the input, and remember it
    String suffix = input.substring(input.length() - 1);

    // the new input string, is the original with the tail cut off
    String prefix = input.substring(0, input.length() - 1);

    // generate list of permutations for the new substring(n - 1)
    List<String> permutations = permutate(prefix);

    // we hold the actual final result here
    List<String> result = new ArrayList<>();

    // for each permutation of the smaller string, stick last element of
    // the original at both ends first, and then in between the string
    // too. thus the new string result will contain the last character
    // of the original in each position
    for (String perm : permutations) {
        // add at both ends of the permutated string
```

```
            result.add(perm + suffix);
            result.add(suffix + perm);

            // add it, in between the permutated strings
            for (int i = 0; i < perm.length() - 1; i++) {
                String head = perm.substring(0, i + 1);
                String tail = perm.substring(i + 1, perm.length());
                result.add(head + suffix + tail);
            }
        }

        // return the result
        return result;
    }
```

Substring permutation

Another interesting problem, which might look like a tough one is to check if a long string contains a permutation of another shorter string within itself. What we need to realize here is that we can ‘undo’ the permutation in a predictable way, how ? Well by simply sorting the input short string. Then when we traverse the longer string, we take a sub-string of it, as long as the short string, sort that as well and we compare both, if they match then it is indeed a permutation.

This problem is an extension of the clasic sub-string problem where we are asked to check if a given shorter string is a sub-string of another longer string.

Here instead of linear, the complexity is `O(n * slog(s))`. Where N is the length of the long string, and S is the length of the short string. If we know that N is sufficiently bigger than S then we can drop this non-dominant term S. Meaning the complexity will converge to only O(N)

```
int subpermutation(String longString, String shortString) {
    // sort the input, to make sure it is normalized the same way the window
    // string below will be, this we will make use in equalsIgnoreCase
    shortString = shortString.chars()
                    .sorted()
                    .collect(StringBuilder::new, StringBuilder::
                       appendCodePoint, StringBuilder::append)
                    .toString();

    List<String> listOfMatches = new ArrayList<>();
    for (int i = 0; i < longString.length(); i++) {
        // take a sub-string of the original just as long as the input short
           string, which we will sort below
        // account for the fact that the left over of long-string might be
           less than short-string length
        String windowString = longString.substring(i, Math.min(longString.
           length(), i + shortString.length()));

        // unscamble the sub-string, of shortString.length chars, into
        // a predictable ordered sequence of chars this way we can be
        // sure that whatever the permutation in windowString is, it
        // will be normalized, by sorting it, to be ready for comparison
        String sortedString = windowString.chars()
                        .sorted()
                        .collect(StringBuilder::new, StringBuilder::
                           appendCodePoint, StringBuilder::append)
                        .toString();

        // compare against the short string, which is also sorted.
        if (shortString.equalsIgnoreCase(sortedString)) {
            listOfMatches.add(windowString);
        }
    }

    // return the number of matches
    return listOfMatches.size();
}
```

The catch in this problem is to realize that a permutation can be converted to a normalized, predictable sequence, by sorting the items. And to also realize that if the sub-string extracted from the long-string contains a character originally not present in the input short-string, then the comparison would catch that and correctly report that these two cannot be permutations of one-another

Unique characters

Another problem in string space, might revolve around checking if a string has only unique characters. This problem similarly to the one above, can be normalized to make the task easier. Generally speaking when working with strings, we should either think about 'sorting' the string or 'hashing' the string.

To check if a string contains only unique characters we would simply sort the string, and check if each pair of characters are different. The sorting will position the characters such that if we have repeating ones they would be next to one another, meaning that we can compare the `char[i] with char[i + 1]`. If we traverse the entire string without finding a matching pair we can simply conclude that the string contains only unique characters.

Another solution is to put all characters in a hash map, count each occurance and check if we have a character that occurs more than once. This solution trades in space, for time complexity.

```
boolean unique(String input) {
    // sort the string, this can be done with quick sort, to avoid using
       intermediary structures, and wasting space,
    String sorted = input.chars()
                    .sorted()
                    .collect(StringBuilder::new, StringBuilder::
                       appendCodePoint, StringBuilder::append)
                    .toString();

    // one way to do it is to go through the sorted array and check if a
       there are any elements which are the same and are sequential
    for (int i = 0; i < input.length() - 1; i++) {
        if (sorted.charAt(i) == sorted.charAt(i + 1)) {
            // at this point we know that there are duplicate characters in
               the string
            // therefore the string does not contain unique characters only
            return false;
        }
    }
    return true;
}
```

Permutation check

This is a common problem we already saw above, where we check if a given string is a permutation of another. This is a more generalized solution. Which instead of sorting (which will also work) we employ another approach, namely, storing one string (char wise) into a hash map and traversing the other string.

The approach is simple, we store one of the string's characters in a hash map, for each character is a key, and the value is the occurance count of that character.

Then we traverse the second string, we check if the current character is contained in the hash map, if yes we subtract from the count, in the end if we end up with and empty hash map or all characters have a count of 0, we know that the same exact characters are contained in both strings, so that means that both strings are permutations of the same characters. If while traversing the second string we meet a character that is not contained in the hash map, we can early exit because the second string contains a character originally not present in the first string.

```
boolean permutation(String first, String second) {
    // early exit, we know that this can not be true if both have different
       lengths to begin with
    if (first.length() != second.length()) {
        return false;
    }

    // first collect all unique characters in a map, from the first string,
       this way we will have a count of all characters
    // contained in the string
    Map<Character, Integer> countMap = new HashMap<>();
    for (int i = 0; i < first.length(); i++) {
        if (!countMap.containsKey(first.charAt(i))) {
            countMap.put(first.charAt(i), 1);
        } else {
            Integer count = countMap.get(first.charAt(i));
            countMap.put(first.charAt(i), count + 1);
        }
    }

    // use the count from the map and go through the second string, each time
        we meet a char that is present in the map that means
    // that it was present in the first string, decrement the count, if a
       characters exists in second string, but does not in the
    // first, then we can early exit and say that there is no possible way
       that the first string can be a permutation of the second
    for (int i = 0; i < second.length(); i++) {
        if (countMap.containsKey(second.charAt(i))) {
            // decrement the count of each unique character, in the end we
               have to end up with all counts being 0, meaning that not
            // only the same chars were present in the second string but also
                the same number of them as well
            Integer count = countMap.get(second.charAt(i));
            if (count == 1) {
                countMap.remove(second.charAt(i));
            } else {
                countMap.put(second.charAt(i), count - 1);
            }
        } else {
            return false;
        }
    }

    // check if the count map is empty, if yes, then that means we have
       removed all characters exactly count times, therefore both strings are
        permutations of one another
    return countMap.isEmpty();
}
```

Palindrome check

Yet another common problem is working with palindromes, words or strings which are spelt backwards and forwards the same. A problem involving a palindrome might ask of us to check if any permutations of a given string can produce a palindrome.

To solve this issue we can simply take into account the fact that to have a palindrome we must meet the following conditions

- the input string is of even length - therefore each character MUST occur even number of times (e.g. `a a b b c c` -> `a b c c b a`)
- the input string is of odd length - therefore only one character CAN occur odd number of times, the rest must be even (e.g. `a a b b c c c` -> `a b c c c b a`)

To make this check easy we can simply put all characters in a map and count them up, then we check if our string is of even length, all counts must be even, otherwise we can allow for at most one and only one character to have odd number of occurances in the string.

```
boolean palindrome(String input) {
    // string of input one, has no permutations or we can say the string is
       it's own permutation and the string is a palindrome too
    if (input.length() == 1) {
        return true;
    }

    // how to approach this problem, we know that a palindrom must have an
       even number of characters appearing in it, the only way
    // it can be a plindrome is if the same number of characters appear in
       the "left" and "right" halfs. This means that if the
    // input string's lendth is even i.e has an even number of characters,
       then all characters must appear even number of times, 2,
    // 4, 6, etc. If the input string length is not even, then at most one
       character from those can have a non even count.

    // count all char occurances, and store them in the map
    Map<Character, Integer> countMap = new HashMap<>();
    for (int i = 0; i < input.length(); i++) {
        if (!countMap.containsKey(input.charAt(i))) {
            countMap.put(input.charAt(i), 1);
        } else {
            Integer count = countMap.get(input.charAt(i));
            countMap.put(input.charAt(i), count + 1);
        }
    }

    // check how many non even characters are there hold them in the count,
       we do not care about the even characters, what we mean is that if we
       find 0 odd character occurances, therefore they are all even.
    int counter = 0;
    for (var entry : countMap.entrySet()) {
        // account only for the odd ones
        if (entry.getValue() % 2 != 0) {
            counter++;
        }
    }

    // when the input length is even, we can not have non-even count of
       characters, when the input length is non-even, at most one
    // character can have a non-even count in the original string, below we
       check for exactly those conditions.
    boolean palindrome = false;
    if (input.length() % 2 == 0) {
        palindrome = counter == 0;
    } else {
        palindrome = counter == 1;
    }
    return palindrome;
}
```

Wraparound substring

A modification of the traditional sub-string check, where we try to find if a given sequence of characters is a sub-string inside another string, however while they are sequential, some characters might wrap around to the start of the string.

For example, say we are looking for the sub-string `waffle` inside the given string `a waffle was found on the ground`. In this traditional example, the full sub-string is found in the same sequence in the string starting at range / positions with indices `[1-6]`.

However let us take the same string but do a slight 'rotation' on it, and now we have `fle was found on the ground a wa` - The same sequence of `waffle` is still present however part of it is at the end of the string, another part is at the beginning.

Another example where the input string is the same length as the sub-string, which is really just the same as the general case.

- `waterbottle` <-> `erbottlewat` - is contained completely, but wraps around
- `waterbottle` <-> `orbottlewat` - is not contained, a mismatching character

To solve this issue we have to simply modify the sub-string algorithm, instead of looping through the string from start to finish, we loop through it with wrap around. The main thing to consider is that we might end up looping infinitely, to stop that we have to keep track of how many times we loop through the source string, the one inside which we are looking for a sub-string match. Worse case scenario is we loop through it at most 2 times, which is O(2*N) which is O(N), we can drop the constants.

```
boolean substrting(String orig, String rotated) {
    int i = 0;
    int j = 0;

    int loops = 0;
    int count = 0;

    // loop over the original string, and and check how many consequtive
       characters from the original match with the
    // rotated string, the caveat here is to just loop around the original,
       using mod, to make sure all characters
    // are inspected in the correct order. (we do simple wrap around)
    while (i < orig.length()) {
        // if two loops over the rotated string were made, and we still have
           not found the original string, then we can bail out,
        // there is no match
        if (loops > 2) {
            break;
        }

        // each time a full loop is made, account for it, a loop is made when
            the next j would become greater than the original
        // length, and a wrap around would occur
        loops += (j + 1) / orig.length();

        // check against the current pos in original and rotated, if not
           equal move the orig string pointer j
        // forward, accounting for index loop around, using mod
        if (orig.charAt(i) != rotated.charAt(j)) {
            // when count is already bigger than 1
            if (count > 1) {
                // this case is mandatory to handle, since we have already
                   started 'counting' and we encounter a non-matching
                // character, that means that the string partially matched,
                   but not the entire sub-string, we bail here
                break;
            }
            j = (j + 1) % orig.length();
            continue;
        }

        // both chars at the given position match, increment the count, and
           move both pointers forward, at some
        // point if count == orig.length == rotated.length we know that the
           rotation is valid
        j = (j + 1) % rotated.length();
        i++;
        count++;
    }

    // reaching this point here if the count is exactly the length of the
       original sub-string, that means the rotated string
    // contained the orig as a sub-string, somewhere in its representation
    return count == orig.length();
}
```

Since we are looking for a sequential sub-string, the moment we start incrementing the count, and we notice a non-matching character, we can early bail out, the sub-string sequence is broken, meaning it is not contained.

Arrays

Reversing array

One of the most common algorithms, that is not that complex but comes up often and is something that should be understood. The approach basically walks the array from both ends, and swaps elements at both ends of the array, while the two pointers, end and start meet. The moment start overlaps with end we can stop with the swapping. Key thing to note here is that start has to become strictly bigger or equal than end

```
int[] reverse(int[] input, int start, int end) {
    while (start < end) {
        // general swap method, nothing special, swap the elements at both
           ends
        int tmp = input[start];
        input[start] = input[end];
        input[end] = tmp;

        // move start forward and end backward, do this until they basically
           meet
        start++;
        end--;
    }
    return input;
}
```

```
   [1, 2, 3, 4, 5, 6] s = 0, e = 5

   [6, 2, 3, 4, 5, 1] s = 1, e = 4

   [6, 5, 4, 5, 2, 1] s = 2, e = 3

                      s = 3, e = 2
```

Rotating array

This problem extends off of the base problem of rotating an array, what rotating an array means is rotating it such that the elements are pushed forward, from some position `k` and they loop around the end of the array and are `pushed` at the beginning of the array

```
   [1, 2, 3, 4, 5, 6]

   [5, 6, 1, 2, 3, 4]
```

In the example above the rotation factor `k` is 2, note that if the rotation factor is 0, that would imply we simply have to only reverse the array, and that is that. That is simply a special case of the rotation problem, but the rotation problem itself is solved by reversing the array around the rotation factor `k`

```
int[] rotate(int[] input, int k) {
    // first reverse the entire array, this is the base case
    input = reverse(input, 0, input.length - 1);

    // then reverse from the start to the `k`-th element
    input = reverse(input, 0, k - 1);

    // finally reverse the elements from `k` to the end of the array
    input = reverse(input, k, input.length - 1);
    return input;
}
```

```
   [1, 2, 3, 4, 5, 6] - input starting array

   [6, 5, 4, 3, 2, 1] - reverse entire array

   [5, 6, 4, 3, 2, 1] - reverse from 0 - (k - 1)

   [5, 6, 1, 2, 3, 4] - reverse from k - (l - 1)
```

Move zeroes

Another interesting problem, is moving all zero (or any other value) elements in an array at the end, in-place. The way this is achieved is by keeping a pointer where non-zero elements are placed, starting from the beginning of the array. When the current element is a zero and the next one is not, we put the non-zero element at the current pointer position `s`, then we move the pointer `s` forward, and zero out the next element i.e. `[i - 1]`. A side case that we need to handle is when the current element is non zero, we also need to move the pointer `s` forward, this will make sure that we skip over non-zero elements and both `i` and `s` are incrementing in sync, so we do not override the actual non-zero element later on.

```
int[] movez(int[] input) {
    int s = 0; // pointer into last non zero element in the array
    for (int i = 0; i < input.length - 1; i++) {
        if (input[i] == 0 && input[i + 1] != 0) {
            // when the current element is a 0 and the next one is not,
            // swap with the 0th element, and move the current non-zero
            // element's pointer 's' forward
            input[s] = input[i + 1];
            input[i + 1] = 0;
            s += 1;
        } else if (input[i] != 0) {
            // when the current element is not a 0, simply move the pointer
            // forward, since it must not be moved from it's current position
            s += 1;
        }
    }
    return input;
}
```

Sum target

Another problem which pops up real often is to find two elements in an array which sum to a given target sum. This is quite an interesting one, the way to solve it is to realize that we already have at least two of the variables, the target sum and one of the values, `s = x + y`. In this equation, the value of `x` is the current element from the array (while traversing), the value of `y` we can calculate by re-arranging it such that `y = s - x`. Meaning that we simply search in the array for the value of `y`. If there is such we simply remember the index of that value or print it out, whatever the solution requires. In these tasks it is often assumed that a pair will exist, meaning that if the current value of `x` is the same as the sum we have to find a `0` in the array, that would be the second part of the pair

Note that in the inner loop where we look for the value, we already start from `i + 1`, why is that the case ? Well since `i` starts from the beginning of the array, we are always going to find the second value in the `i + 1` subarray. If such a value exists, of course. The break below is not needed, but we assume the elements are unique, in any case if they are not, the array can be made unique or the break can be removed

```
int[] sum(int[] input, int target) {
    List<Integer> pairs = new LinkedList<>();
    for (int i = 0; i < input.length; i++) {
        // since we have the target sum, and the current element
        // we can find the second element we need, by subtracting
        // the current element from the target sum
        int result = target - input[i];

        // the result is the element we are going to look for in
        // the sub-array from i - length. We also assume that
        // a pair should exist if the target == input[i], we should
        // look for an element with a value of 0 in the array
        for (int j = i + 1; j < input.length; j++) {
            if (result == input[j]) {
                // add the pairs of elements which sum up to target
                // note that we assume the array does not contain
                // duplicates, for simplicty, break
                pairs.add(i);
                pairs.add(j);
                break;
            }
        }
    }

    // return the array of pairs where each two indices are pairs
    return pairs.stream().mapToInt(Integer::intValue).toArray();
}
```

Sub array

Another very often asked problem is to find of sub-array which produces the biggest sum. While the array can have negative numbers. The way this solution works, is by tracking the current max sum, and resetting the sum every time a better sum is reached

The most important part here is the current sum tracking. What happens is we compare the local sum accumulated so far with the current element, if the current element is greater than the local sum + current element, the sequence 'resets', at the current index `i`, otherwise the sequence continues. By resets, what we really mean is that the current element is greater than the local sum accumulated so far meaning that that accumulation can be discarded, we have found a better local sum sequence

```
    [-2, 1, -3, 4, -1, 2, 1, -5, 4]
```

Take the example above, starting off the local sum is `-2`, then the local sum will be summed with `1`, producing `-2 + 1 = -1`, however, the element itself is `1`, greater than the local sum, meaning that between the sequence of elements `-2, 1` and simply sequence starting at element `1`, the second one is producing a greater total sum = 1. Then in the next step we go to `-3`, where the current local sum will be `1 + (-3)= -2`, which is better than the element itself `-3`, meaning the sequence will be extended to elements `1, -3` not reset. Third step the element is `4`, the sum so far is `-2`, between `-2 + 4 = 2` and the element `4` itself, we can see the local sum will be greater if we reset the sequence to start at the current element `4`. This process of either extending the sequence or resetting it at the current element keeps going on until we finish with all elements, at the end the `max` value will hold the biggest local sum ever encountered

```java
int subarray(int[] nums) {
    if (nums == null || nums.length == 0) {
        return 0;
    }

    int max = nums[0];
    int current = nums[0];

    for (int i = 1; i < nums.length; i++) {
        // either extend the current subarray or start a new subarray from
           nums[i]
        current = Math.max(nums[i], current + nums[i]);
        // update the maximum sum found so far
        max = Math.max(max, current);
    }

    return max;
}
```

A brief follow up of the steps above, could look like this

```
   inut-array: n[-2, 1, -3, 4, -1, 2, 1, -5, 4]
   curr-sum:   c = -2
   max-sum:    m = -2

   looping i = 1 -> 1
        c = max(1, -2 + 1) -> 1 - seq reset, starts at i = 1
        m = max(-2, 1) -> 1 - new total max is found, set it

    looping i = 2 -> 1
        c = max(-3, 1 + (-3)) -> -2 - seq continues, starts at i = 1, ends in
            j = 2
        m = max(1,  1) -> 1 -> total max is retained so far no bigger sum is
           found

    looping i = 3 -> 1
        c = max(4, -2 + 4) -> 4 - seq resets, starts at i = 3
        m = max(1, 4) -> 4 -> new total max is found, set it

    looping i = 4 -> 1
        c = max(-1, 4 + (-1)) -> 3 - seq continues, starts at i = 3, ends in
           j = 4
        m = max(4, 3) -> 4 -> total max is retained so far no bigger sum is
           found

    looping i = 5 -> 1
        c = max(2, 3 + 2) -> 5 - seq continues, starts at i = 3, ends in j =
           5
        m = max(4, 5) -> 5 -> new total max is found, set it

    the steps keep repeating, until the entire array is exhaused, and finally
       we find
    in the end the max sum is actually 5, produced by these 4 elements - [4,
       -1, 2, 1]
```

Array removal

There are basically two general ways to do array removal, for a given index element that has to be removed, assume that the implementations account for re-sizing the capacity of the array to reduce the capacity if needed

- swap the element to be removed with the last element, when order is not important
- shift down all elements right - left, starting from the index of the element to be removed

Swap solution

This is usually included in gotcha questins, where the order of array elements often does not matter, meaning that we can simply place the last element over the element to be removed, which is often enough to consider this implementation as 'removal'

```
void remove(int[] n, int i) {
    n[i] = n[n.size - 1];
    n.size--;
}
```

Shift solution

Following elements are `pulled` down overriding elements starting at index `k`, which is the index to be removed, the element at that index is assigned the next element, and so on, until the end of the array, this has the effect of pulling down all elements by one, shifting them to the left.

```
void remove(int[] n, int k) {
    for (int i = k; i < n.size - 1; i++) {
        n[k] = n[k + 1];
    }
    n.size--;
}
```

Array insert

Similarly we have two options here, assume that the implementations account for re-sizing the capacity of the array to fit the new elements, if needed

- inserted at the end of the array, set element at [len - 1] = value, when array is big enough
- inserted at some given index position, where elements are shifted left - right, starting from the index to be inserted at

Append solution

Simply append the element at the last possible, free position, which in this case is index `size`, that is the new last free space in the array.

```
void insert(int[] n, int v) {
    n[n.size] = v
    n.size++;
}
```

Shift solution

In this solution the elements are pulled towards the end of the array, note that `size` below refers to the actual number of elements present in the array, we start off from index `size` due to the fact we are inserting one element, meaning that index `size` will be the new last index after the insertion is done.

```
void insert(int[] n, int v, int k) {
    for (int i = n.size; i > k; --i) {
        n[i] = n[i - 1];
    }
    n[k] = v;
    n.size++;
}
```