# 1. Arrays and Strings

**Note that array questions and string questions are often interchangeable. That is, a question that this book states using an array may be asked instead as a string question, vice versa**

## Hash Tables

A hash table is a data structure that maps keys to values for highly efficient lookup. It offers fast insertion and searching. They are limited in size (can be resized, to be avoided). It is hard to order.

### How it works:

1. Key values are assigned to indices (hash code) in a Hash Table using a Hash Function
2. A Hash Function helps calculate the best index (hash code) an item should go into.

There are a number of implementing this.

### Simple implementation: Use an array of linked lists and a hash code function. To insert a key and a value, we do the following:

1. Compute the key's hash code (usually an int or long).
2. Then, map the hash code to an index in the array. This could be done with something like hash(key) % array_length.
3. At this index, there is a linked list of keys and values. Store the key and value in this index. We must use a linked list because of collisions.

To retrieve the value pair by its key:

1. Compute the hash code from the key.
2. Compute the index from the hash code.
3. Search through the linked list for the value with this key.

If the # of collisions is very high, the worst case runtime is `O(N)`, where N is the number of keys. If the collisions are kept to a minimum, the lookup case is `O(1)`.

### Alternative implementation: Use a balanced binary search tree

This will give us a `O(log N)` lookup time. This method potentially uses less space, since we no longer allocate a large array. We can also iterate through the keys in order, which can be useful sometimes.

### Arrays vs ArrayList

1. An array is a basic functionality provided in Java. ArrayList is part of collections framework in Java. Array members are accessed using [], while ArrayList has a .get() method.
2. Arrays are created with `int[] arr = new int[10]`.
   ArrayLists are created with `ArrayList<Type> arr = new ArrayList<Type>()`.
3. Array is a fixed data structure, while ArrayList is not.
4. Array can contain both primitive data types as well as objects of a class, depending on the definition of the array. ArrayList only supports object entries.
   **Note**: When we call arrayList.add(1), it casts the primitive int data type into an Integer object.
5. Since ArrayList can't be created for primitive data types, members of ArrayList are always references to objects at different memory locations. Therefore, the actual objects are not necessarily stored at contiguous locations. Only the references of the actual objects are stored at contiguous locations.
6. In array, in case of primitive types, actual values are stored in contiguous locations. In case of object type, allocation is similar to ArrayList.

## ArrayList & Resizable Arrays

An ArrayList is an array that resizes itself when needed while still providing O(1) access. When the array is full, the array doubles in size. Each size takes O(n) time, but happens rarely that its amortized insertion time is still O(1).

*Why is the amortized insertion runtime O(1)?* Suppose you have an array of size N. We can work backwards to compute how many elements we copied at each capacity increase. Observe that when we increase the array to K elements, the array was previously half that size (we needed to copy K/2 elements).

Therefore, in total, the number of copies to insert N elements is roughly N/2 + N/4 + N/8 + ... + 2 + 1, which is ~N elements.

Therefore, inserting N elements takes O(N) work total. Each insertion is O(1) on average, even though some insertions take O(N) time in the worst case (when you need to double the array size).

## StringBuilder

```
String joinWords(String[] words){
  String sentence = "";
  for (String word : words){
    sentence = sentence + word;
  }
  return sentence;
}
```

On each concatenation, a new copy of the string is created, and the two strings are copied over. The first iteration requires us to copy x characters. The second iteration requires copying 2x characters, and so on. Total time is O(x + 2x + ... + nx) → O(x * n^2)

StringBuilder can help you avoid this problem. StringBuilder simply creates a resizable array of all the strings, copying them back to a string only when necessary.

# Questions

## Q1. Implement an algorithm to determine if a string has all unique characters. What if you cannot use additional data structures?

A: You should first ask your interviewer if the string is an ASCII string or Unicode string. Asking this question will show an eye for detail and a solid foundation in computer science. In this case, we assume ASCII.

One solution is to create an array of boolean values, where the flag at index `i` indicates whether character `i` in the alphabet is contained in the string. The second time you see this character you can just return `false`.

We can also immediately return `false` if the string length exceeds the # of unique characters in the alphabet.

```
boolean isUniqueChars(String str){
  if (str.length() > 128) return false;

  boolean[] char_set = new boolean[128];
  for (int i = 0; i < str.length(); i++){
    int val = str.charAt(i);
    if (char_set[val]){
      return false;
    }
    char_set[val] = true;
  }
  return true;
}
```

Time complexity is `O(n)`, where n is the length of string. Space complexity is `O(1)`.

If we can't use additional data structures, we can do:

1. Compare every character of the string to every other character of the string. This will take `O(n^2)` time and `O(1)` space.
2. If we're allowed to modify the input string, we can sort it in `O(n log n)` time and then linearly check the string for neighboring characters that are identical.

## Q2. Given two strings, write a method to decide if one is a permutation of the other

We should confirm some details about the question, for example: is the permutation comparison case sensitive? Is whitespace significant?
For this problem, we will assume that the comparison is case sensitive and whitespace is significant. We also assume that the character set was ASCII.

A: Observe that strings of different lengths cannot be permutations of each other.

Solution 1: Sort the strings, then compare the sorted versions of the strings.
While this algorithm is clean, simple, and easy to understand, it is not the most efficient way to solve the problem.
Time Complexity: O(n log n) for sorting

Solution 2: Check if two strings have identical character counts.

```
boolean isPermutation(String s, String t){
  if (s.length() != t.length()) return false;

  int[] letters = new int[128];

  char[] s_array = s.toCharArray();
  for (char c : s_array){
    letters[c]++;
  }
  for (int i = 0; i < t.length(); i++){
    int c = (int) t.charAt(i);
    letters[c]--;
    if (letters[c] < 0) {
      return false;
    }
  }

  return true;
}
```

Time Complexity: O(2n) to walk through the arrays → O(n)

## Q3: Replace empty spaces with %20. Note: if implementing in Java, use a character array so that you can perform this operation in place.

A common approach in string manipulation problems is to edit the string starting from the end and working backwards.

The algorithm employs a two-scan approach. In the first scan, we count the number of spaces to get the number of characters we will have in the final string. In the second pass, we replace the space with %20. If there is no space, then we copy the original character.

Implement the solution to this problem using character arrays, because Java strings are immutable. If we used strings directly, the function would have to return a new copy of the string, but it would allow us to implement this in just one pass.

## Q4: Palindrome permutation. Given a string, write a function to check if it is a permutation of a palindrome.

A: We need to have an even number of almost all characters, and at most 1 character (the middle one) can have an odd count (unless the string has even length, then all characters have to have an event count).

*Solution 1:*
Use a hash table to count how many times each character appears. Then, iterate through the hash table and ensure that no more than one character has an odd count.
Time complexity: O(N), where N is the length of the string.

*Solution 2:*

We can't optimize the big O time, since O(N) is the BCR. Instead, we check the number of odd counts as we along (as opposed to checking it after we put everything into the hash table).

## Q5. One Away. Given two strings, write a funciton to check if they are one edit (or 0 edits) away.

Example:

- pale, ple -> true
- pales, pale -> true
- pale, bale -> true
- pale, bae -> false

*Solution 1:* Think about the meaning of: what does it mean for two strings to be one insertion, replacement, or removal away from each other?

- Replacement: The two strings are only different in one place.
- Insertion: The strings are identical except for a shift at some point in the strings.
- Removal: This is just inverse of insertion.

```
boolean oneEditAway(String first, String second) {
  if (first.length() == second.length()) {
    return oneEditReplace(first, second);
  } else if (first.length() + 1 == second.length()) {
    return oneEditInsert(first, second);
  } else if (first.length() - 1 == second.length()) {
    return oneEditInsert(second, first);
  }
  return false;
}

boolean oneEditReplace(String s1, String s2) {
  boolean foundDifference = false;
  for (int i = 0; i < s1.length(); i++) {
    if (s1.charAt(i) != s2.charAt(i)) {
      if (foundDifference) {
        return false;
      }
      foundDifference = true;
    }
  }
  return true;
}

// Check if you can insert a character into s1 to make s2
boolean oneEditInsert(String s1, String s2) {
  int index1 = 0;
  int index2 = 0;
  while (index2 < s2.length() && index1 < s1.length()) {
    if (s1.charAt(index1) != s2.charAt(index2)){
      if (index1 != index2) {
        return false;
      }
      index2++; // Just add index to one of the indexes, to indicate that they are already out of sync
    } else {
      index1++;
      index2++;
    }
  }
  return true;
}
```

Time complexity: O(n), n is the length of the shorter string.

We can optimize this further by merging oneEditReplace with oneEditInsert. oneEditReplace only flags the difference, whereas oneEditInsert increments the pointer to the longer string.

```
boolean oneEditAway(String first, String second) {
  if (Math.abs(first.length() - second.length()) > 1) {
    return false;
  }

  String s1 = first.length() < second.length() ? first : second;
  String s2 = first.length() < second.length() ? second : first;

  int index1 = 0;
  int index2 = 0;

  boolean foundDifference = false;

  while (index2 < s2.length() && index1 < s1.length()) {
    if (s1.charAt(index1) != s2.charAt(index2)) {
      // Ensure that this is the first difference found
      if (foundDifference) return false;
      foundDifference = true;

      if (s1.length() == s2.length()) { // On replace, move shorter pointer
        index1++;
      }
    } else {
      index1++; // if matching, move shorter pointer
    }
    index2++; // always move pointer for longer string
  }
```

```
    return true;
  }
```