UNIVERSITÀ DI PISA

COMPETITIVE PROGRAMMING - REPORT

Preparation for SWE Interviews

An overview of two Advanced Topics

Antonio Strippoli (625044)

A.Y. 2021/2022

Contents

1 Bit Manipulation

Bit manipulation is the act of algorithmically *manipulating bits* or other pieces of data shorter than a word.[1] Examples of computer programming tasks requiring bit manipulation are:

- Error detection and correction;
- Data compression;
- Embedded systems;
- Networking.

1.1 Background Knowledge

1.1.1 Bitwise Operators

A **bitwise operation** *operates* on a bit string, a bit array or a binary numeral (considered as a bit string) at the level of its *individual bits*. It is a fast and simple action, basic to the higher-level arithmetic operations and directly supported by the processor.[2]

When the strings have **different lengths**, we can imagine padding the shortest one with **leading zeros**, in order to get the same length.

And operation

The **and** operation between two bit strings `x,y` is written as `x & y`. It will **output** a bit string `z` that has **one bits** in positions where **both** `x` **and** `y` have one bits. Example:

```
 101010   &   (42)
   1011   =   (11)
--------------------
 001010
```

Or operation

The **or** operation between two bit strings `x,y` is written as `x | y`. It will **output** a bit string `z` that has **one bits** in positions where **at least one of** `x` **and** `y` have one bits. Example:

```
 101010   |   (42)
    100   =    (4)
--------------------
 101110
```

Xor operation

The **xor** operation between two bit strings `x,y` is written as `x ^ y`. It will **output** a bit string `z` that has **one bits** in positions where **exactly one of** `x` **and** `y` have one bits. Example:

[1] https://en.wikipedia.org/wiki/Bit_manipulation
[2] https://en.wikipedia.org/wiki/Bitwise_operation

$$\begin{array}{rcl} 101010 & \hat{} & (42) \\ 11110 & = & (30) \\ \hline 110100 & & \end{array}$$

Not operation

The **not** operation on a bit string `x` is written as $\sim$`x`. It will **output** a bit string `z` where all the **bits** of x have been **inverted**. The result of the **not** operation depends on the **length** of **the bit representation**, because the operation inverts all bits. Example (with 16-bit short number):

$$\begin{aligned} x &= 42_{10} = 0000000000101010_2 \\ \sim x &= 42_{10} = 1111111111010101_2 \end{aligned}$$

Bit shifts

The **shift** operation on a bit string `x` can be of two types. The **left shift** is written as `x << k` and **shifts** the bits k positions **to the left**, **increasing** the value of the number. The **right shift** is written as `x >> k` and **shifts** the bits k positions **to the right**, **decreasing** the value of the number. During the process, bits moving **out of the length** of the string will **disappear**, while **empty spaces** are **filled with zeroes**. Example (with 8-bit number):

$$\begin{aligned} x &= 42_{10} = 00\underline{101010}_2 \\ x << 3 &= 80_{10} = \underline{0101}0000_2 \\ x >> 3 &= 5_{10} \ \ = 00000\underline{101}_2 \end{aligned}$$

1.1.2 Applications

By combining the bitwise operators, we can achieve interesting results:

- Check if an **integer** is **even or odd**;
- Check if a single **bit** of an integer is **set or not**;
- **Swap** two numbers without using a third variable;
- Generate the **power set** of a given set;
- Implement a **XOR Linked List**.

These applications and tons more can be found at the following links:

- https://graphics.stanford.edu/~seander/bithacks.html;
- https://medium.com/techie-delight/bit-manipulation...;
- https://leetcode.com/tag/bit-manipulation/discuss/1151183/....

1.2 Problems

1.2.1 Bits

Source: codeforces

Problem: Given two integers l and r, find the integer x which has the most number of bits set to 1 in the interval $l \leq x \leq r$. In case there are multiple such numbers, find the smallest of them.

Solution:

A **brute-force** approach would need to consider every possible number in-between l and r, count the number of bits set to 1 for each of them and output the smallest number having the maximum number of bits set to 1, thus having a complexity of $O(r - l)$.

But if we move to the **bits domain**, we can do better. Let's consider an example:

```
l = 01 = 0001
r = 12 = 1100
```

The idea is to start from l and **greedily set its bits to 1**, starting from the least significant bit. We will continue until we get a number greater than r. Let's check this idea with the example:

```
l = 01 = 0001 (already set)

l = 03 = 0011

l = 07 = 0111

l = 15 = 1111 (stop, l > r)
```

In this case, 7 will be our final answer.

Now, how do we **implement** this idea? We can exploit two bitwise operators: **OR** and **LEFT SHIFT**. Consider iterating over all the positions of the binary as described before and let c be the position that must be set to 1. Then, to update l, we can just perform:

```
l | (1 << c)
```

`1 << c` will produce a number having only the c bit set and by doing the OR between l and this number we are setting the c bit of l to 1.

A simple **follow-up** to this problem could be to also consider negative range extremes. My solution coded in C++ is available here.

1.2.2 Missing and duplicate elements in an array

Source: techiedelight

Problem: Given an integer array A of size n, where $1 <= A_i <= n$, there is an element in the range $1...n$ missing in the array, and another element duplicated. Return these two elements.

Solution:

We will exploit **4 properties** of the XOR operation:

1. $x \wedge 0 = x$;
2. $x \wedge x = 0$;
3. Associativity;
4. Commutativity.

We can exploit properties 1. and 2. to "cancel out" the numbers that are not feasible candidates to the solution. The idea is to **XOR all the elements in** A **and numbers in the range** $1...n$.

Let's consider a simple instance of the problem:

```
A = [1, 2, 3, 3]
```

By XOR-ing as described we obtain:

```
1 ^ 2 ^ 3 ^ 3 ^ 1 ^ 2 ^ 3 ^ 4 =
```

By **commutativity**:

```
= 1 ^ 1 ^ 2 ^ 2 ^ 3 ^ 3 ^ 3 ^ 4 =
```

By **associativity**:

```
= (1^1) ^ (2^2) ^ (3^3) ^ (3^4) =
```

Finally, by **properties 1 and 2**:

```
= 0 ^ 0 ^ 0 ^ (3^4) =
```

```
= 3 ^ 4 = 7_10 = 111_2
```

We can notice that we're left with the **XOR** between the **missing** number and the **duplicate** number.

Now, how do we **"decompose"** this XOR-ed result in order to get back the two numbers? Generally, the XOR is not a reversible operation.

But we know that given two numbers x and y, any set bit in `(x ^ y)` will be either set in x or y, **but not in both**. By using this fact, let's identify the **right-most set bit** and its position k in `(x ^ y)`. Note that given the constraints we're sure there will always be at least one bit set. Then, we can **partition** the original array in two groups: elements having the k-th bit set and those not having the k-th bit set.

Now, we will **XOR** the first group with all the **numbers** in the **range** $1...n$ having the **k-th bit set**, and do the same thing for the second group with the remaining elements in the range $1...n$.

In the end, one of the two groups will contain the missing element, the other will contain the duplicated element.

Let's look at that by continuing the previous example.

The **rightmost bit** in 7_{10} is set, so $k = 3$. Let's create the **two partitions**:

$A_{3rd\ bit=1} = [1, 3, 3], \quad A_{3rd\ bit=0} = [2]$

By performing the above explained **XOR** on the **first group** we obtain:

```
(1 ^ 3 ^ 3) ^ (1 ^ 3) = 3
```

Which is the **duplicated** element. On the second group instead we obtain:

```
(2) ^ (2 ^ 4) = 4
```

Which is the **missing** element.

Coding this idea in C++ is really straightforward and my code is available here.
This is a well-known problem and several variants can be found online and proposed at an interview. For example:

- https://leetcode.com/problems/missing-number/;
- https://leetcode.com/problems/find-the-duplicate-number/.

1.2.3 Find element occurring once when all others are present thrice

Source: geeksforgeeks

Problem: Given an integer array A where every element occurs thrice except for one that occurs once, find and return the element that occurs once.

Solution:

This problem is a harder follow-up of the previous one. Indeed, it is not possible to apply the same XOR technique again, since both the target and the other elements are appearing an odd number of times.

Basic approaches to solve this problem involves **sorting+scanning** in $O(n\,log\,n)$ time or **hashing** in $O(n)$ time but also $O(n)$ space. We can do **better** by using **bitwise operators**.

The idea is to make use of **2 variables**:

- `once`: at each iteration, holds the **XOR** of the elements which have appeared only **once**;
- `twice`: at each iteration, holds the **XOR** of the elements which have appeared only **twice**;

Then, we need to implement different behaviours depending on how many times we have seen a number:

- 1^{st} time seeing a number: XOR it to `once`;
- 2^{nd} time seeing a number: remove it from `once` and XOR it to `twice`;
- 3^{rd} time seeing a number: remove it from both `once` and `twice`.

If we manage to implement these definitions, after iterating over all the elements we will just need to **return the content of the variable** `once`.

Let's call `x` an element appearing thrice in the array. To implement the previously described steps, we will do something like:

1. `twice = twice | (once & x);`
2. `once = once ^ x;`
3. `thrice = once & twice;`
4. `once = once & ~thrice;`
5. `twice = twice & ~thrice;`

To better understand why these 5 steps work, let's execute them three times with $x = 42_{10} = 101010_2$. Note that both `once` and `twice` are initialized to 0.

First time

First we compute the & between `once` and `x`:

$once \ \& \ x = 000000 \ \& \ 101010 = 000000$

Then, XORing 0 with `twice` will return `twice`. So, `twice` has **not** been **updated**, as we wanted.

Instead, next step will **XOR** `once` with x, as we wanted:

$once \ \hat{} \ x = 000000 \ \hat{} \ 101010 = 101010$

Note that the next three steps will not change the values of `once` and `twice`, so we can skip them for now.

Second time

Let's compute once more the first two steps. Now, with the first step `twice` will get updated:

$once \ \& \ x = 101010 \ \& \ 101010 = 101010$

Then, by XORing this result with `twice` we get:

$twice \ \hat{} \ 101010 = 101010$

Next, the second step will remove `x` from `once`, since they contain the same value. So, we ended up with having **`x` "stored" in `twice`** and **removed** from `once`, as we wanted. Again, it can be proved that the last three steps will not change anything.

Third time

Now, since the element appears the third time, we will need to remove it from both `once` and `twice`.

After the first two steps, `twice` will not get updated while `once` will get once more the bit representation of x. That's exactly when the **last three steps** start to become useful. The third step will get the **bits that have appeared thrice** (so, that are in common between `once` and `twice`):

$thrice = once \ \& \ twice = 101010 \ \& \ 101010 = 101010$

Then, the last two steps will **remove those bits** from both `once` and `twice`:

$once \ \& \ \sim thrice = 000000$

$twice \ \& \ \sim thrice = 000000$

Which performs exactly what we wanted. In a real instance of the problem however, most likely we will not get the repeated elements in sequence, but the code will still work. To better understand why, think of it this way: `once` will contain the **bits** that have appeared once, while `twice` will contain the **bits** that have appeared twice. **So, it is not a matter of numbers anymore, but bits**!

My solution coded in C++ is available here. An interesting follow-up for this problem could be to generalize to k the number of times the repeated elements appear.

1.2.4 Maximum subset XOR

Source: `geeksforgeeks`

Problem: Given an integer array A, return an integer denoting the maximum XOR subset value.

Solution:

A **brute-force** approach consists in computing the XOR of every possible subset of A and taking the maximum out of all of them. The time complexity will be dominated by the number of subsets, resulting in $O(2^n)$. Looking at the problem constraints ($<= 10^5$ elements in A), we can infer that a solution in $O(n)$ is required.

The key idea is that **the more we manage to have highly significant bits set in the result, the more we will "score points"**. Now, we're trying to maximize a single number, which is composed of 32 bits (at least in `g++ 5.4`).

Then, let's start by try and have the 32th bit set in the result. **Are there elements in the set having this bit set?** To check if a number x has the i-th bit set, we can make use of the **AND** and **LEFT SHIFT** bitwise operators:

```
x & (1 << i) != 0
```

If we find a single element having that bit set, then we're sure that it will be part of the final result. But what if we have **multiple candidates** $A_1, A_2, ..., A_k$? In this case, we will perform the following:

- **Pick the maximum** among all the candidates;
- Change all the **elements in the array having the i-th bit** set by **XORing** them with the maximum. With this step, we remove the information contained in the maximum from the other elements.

After each iteration, the chosen element (if any), **must not be considered in the next iterations**. An idea to **not waste extra-space** for this task, is to have an increasing **"dead zone"** at the beginning of the array. We can move elements there once selected.

Then, we will perform the same process with the 31th bit, 30th, 29th and so on. After completing the last iteration, we will return as a result the **XOR of all the elements in the array**.

Let's now walk through an **example**:

$A = [3, 5, 4]$ — $A_{bin} = [011, 101, 100]$

We can skip the first steps until we get to work with the 3rd most significant bit, since there are no elements making use of more significant bits.

For the 3rd bit, we have two candidates: 5 and 4. The maximum in this case is 5, so it will be the one moved to the dead zone, while 4 will get XORed with 5. After performing these two operations, the array now looks like this:

$A = [5, 3, 1]$ — $A_{bin} = [101, 011, 001]$

For the 2nd bit there is no ambiguity: we will take 3 (as it is the maximum) and we just need to move it to the dead zone:

$A = [5, 3, 1]$ — $A_{bin} = [101, 011, 001]$

For the 1st and last bit, there is no ambiguity about choosing 1 as the maximum (only candidate), but we still need to change 5 and 3, as they both have the 1st bit set as well. In the end, we obtain:

$A = [4, 2, 1]$ — $A_{bin} = [100, 010, 001]$

By XORing the elements of A all together, we obtain 7, which is the expected answer.

My solution coded in C++ is available here. An interesting follow-up for this problem could be to return the subset of elements instead of the XOR of them.

1.2.5 Chandan and balanced strings

Source: hackerearth

Problem: Given a string s, count the number of sub-strings in s that are balanced strings. A string is balanced if and only if the characters of the string can be partitioned into two multi-sets M_1, M_2 such that $M_1 = M_2$.

Solution:

A **brute-force** solution is to consider all the possible $O(n^2)$ sub-strings and check whether they are balanced or not. Implementing the check efficiently using a sliding-window approach will result in an overall $O(n^2)$ time complexity, but we can do better by exploiting bit manipulation.

In order to exploit them, let's define a **numerical encoding** for a certain character c:

$c_{enc} = 1 << (c -' a')$

That is, **each character will be represented by a number having a single bit set**. The encoding will be different for each number, with no set bit in common.

Now, we can notice that a condition for the string to be balanced is to have the **characters appearing an even number of times**. And the XOR operation is the perfect ally for this kind of tasks. Indeed, if we **XOR** the **numerical encoding of all the characters** in the string and we obtain **0**, the string is balanced.

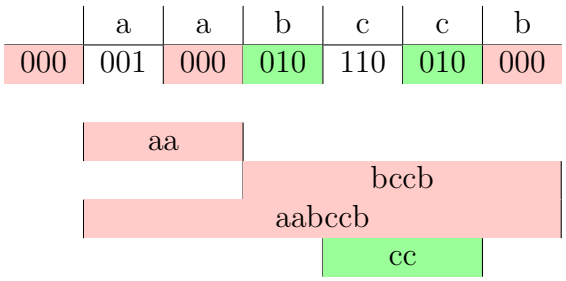
Let's walk through a quick example for this:

$s = aabccb$ — $s_{enc} = [001, 001, 010, 100, 100, 010]$

Notice that the characters in s can be divided in two multi-sets ($M_1 = \{a, b, c\}, M_2 = \{a, b, c\}$) and that the XOR of all the numbers in s_{enc} is 0. But there's more: things become interesting by taking a look at the **intermediate XOR values**:

$000 \rightarrow_a 001 \rightarrow_a 000 \rightarrow_b 010 \rightarrow_c 110 \rightarrow_c 010 \rightarrow_b 000$

As we can see, there are **values that appear multiple times** (000 and 010). To help explain what use can we make of them, let's visualize the situation in a different way:



As we can see, **a pair of equal XOR number corresponds to a balanced sub-string**. For example, the first 000 and the second 000 correspond to the balanced string "aa". Since the two XOR value are equal, that must mean that the XOR of every character (numerically encoded) in-between is 0: in this case, $001_a \,\hat{}\, 001_a = 000$. And if the XOR is 0, we already discussed that the considered string is a balanced one.

Therefore, to obtain the answer we need to compute the **number of possible pairs of equal XOR numbers**. Remember that given the frequency f of a specific XOR number we can compute the number of possible pairs in $O(1)$ by using:

$\frac{f(f-1)}{2}$

To stay within the target of $O(n)$ time, we can simply keep track of the **frequency** using an **hashmap**. My solution coded in C++ is available here.

2 Rolling Hash

Rolling Hash (or Polynomial Hash) is a particular **hash function** that can be applied on a **sequence of elements** and only uses multiplications and additions.

Given 2 sequence of elements s_1 and s_2 having some elements in common, consider h_1 the rolling hash value of s_1: with rolling hash, we can compute the rolling hash value h_2 of s_2 **without the need of hashing the whole sequence** s_2, effectively re-using the hash value h_1. The more elements s_1 and s_2 have in common, the faster the computation will happen.

This feature takes on particular relevance when used in combination with a **sliding-window technique**, since two consecutive windows generally differ by one element only, thus allowing an O(1) computation of the next hash value. Indeed, the most popular application of rolling-hash is the **Rabin–Karp string search algorithm**.

2.1 Background Knowledge

2.1.1 Definition

Let's consider the integer sequence $A = \{a_0, a_1, a_2, ..., a_{n-1}\}$.
The **rolling hash** (or polynomial hash) of A is defined as:

$$h(A, B, M) = (a_0 \cdot B^{n-1} + a_1 \cdot B^{n-2} + ... + a_{n-2} \cdot B + a_{n-1}) \; mod \; M$$

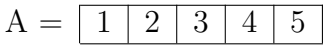
Where B is a constant named **base** and M is a constant named **module**, with
$max(a_i) < B < M$ and $gcd(B, M) = 1$.

The **choice of the values** for B and M is **critical**: rolling hash is still subject to **collisions**, and poorly choosing those parameters may result in errors and **hacking attacks**. You can learn more about it here. Note that in the context of competitive programming it is generally possible to bypass this step by trying and testing different values for the 2 parameters, until all the test-cases for the specific problem pass.

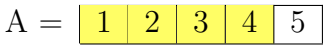
However, even with carefully picked parameters collisions may still arise. One way to deal with them is to **double-check if we have obtained the correct result**. E.g.: in the Rabin-Karp algorithm, once we have found a match using the hash values we could double-check if the match is indeed the string we were searching for.

2.1.2 Usage with a sliding-window technique

Let's consider the following sequence A of integer elements:



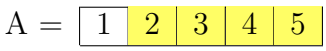
Suppose that we want to compute the hash values of all the sub-arrays of size 4. To accomplish it, we can use a sliding window of size 4. Let's consider the first window:



We can compute the rolling hash of the first 4 elements using the formula described above. We will use $B = 10$ and $M = 41$:

$h_1 = h([1, 2, 3, 4], 10, 41) = (1 \cdot 10^3 + 2 \cdot 10^2 + 3 \cdot 10 + 4)\ mod\ 41 = 1234\ mod\ 41 = 4$

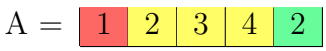
Then, we will move the window 1 position to the right, onto the next sequence:



If we didn't have h_1, we would need to compute h_2 the same way we did for h_1:

$h_2 = h([2, 3, 4, 5], 10, 41) = (2 \cdot 10^3 + 3 \cdot 10^2 + 4 \cdot 10 + 5)\ mod\ 41 = 2345\ mod\ 41 = 8$

Thinking about how the window moves, we can notice that by moving the window 1 position to the right we are **removing the first element** and **adding another one**:



We can use this observation at our advantage, reflecting it onto the second rolling hash value computation. **We don't need to re-compute the hash all over again**. Instead, we can:

- **drop** the first element (with appropriate power of base B);
- **multiply** the left out value with the base number B;
- **add** the new element to the rightmost position;
- **apply the modulo** M.

In practice, h_2 can be computed as:

$h_2 = ((h_1 - 1 * 10^3) * 10 + 5)\ mod\ 41 = ((4 - 1000) * 10 + 5)\ mod\ 41 = -9955\ mod\ 41 = 8$

As you can see, the results of the two different computations of h_2 are identical. The previous formula can be generalized as:

$$h_{i+1} = ((h_i - A[i] \cdot B^{k-1}) \cdot B + A[i + k])\ mod\ M$$

where h_i represents the rolling hash of the i-th sliding window, and k is the size of the window.

This formula is the **essence of the rolling hash**: provided h_i, we can compute h_{i+1} in $O(1)$ time, instead of $O(k)$.

2.2 Problems

2.2.1 Implement strStr()

Source: leetcode

Problem: Given two strings s_1 and s_2, with $|s_1| = n, |s_2| = m$, return the index of the first occurrence of s_1 in s_2. If s_1 is not found in s_2, return -1.

Solution:

A **brute-force** approach is to traverse all the possible starting points of s_2 (from 0 to m-n) and see if the following characters in s_2 match those of s_1. This would require $O(nm)$, but we can do better using rolling hashes.

In order to check whether two strings are equal, we can also use hashing:

- If $h(s_1) \neq h(s_2) \Rightarrow s_1 \neq s_2$;
- If $h(s_1) = h(s_2) \wedge s_1 \neq s_2 \Rightarrow$ false positive (hash collision);
- If $h(s_1) = h(s_2) \wedge s_1 = s_2 \Rightarrow$ true positive.

We can use this string-matching technique in a **sliding-window** environment, exploiting **rolling hash** to quickly compute the hashes.

Note that rolling hash can also be computed on strings: we simply need to **interpret each character as an integer**. We could exploit the **ASCII** or Unicode value of the character, but since the problem's constraints specify that the 2 strings consist of only lowercase english characters, we can **reduce the risk of overflow** with the following mapping:

$a \rightarrow 1$

$b \rightarrow 2$

$c \rightarrow 3$

$\dots$

$z \rightarrow 26$

We will start by **computing the rolling hash** of s_1 and the first n characters of s_2 (the first window). Then, we can **compare the hashes of each window**, and return the starting index at the first true positive match. The mechanism of hashing computation is the exact same we explained above.

If we assume a **low-probability of collisions**, the time complexity would drop to $O(n + m)$, which also matches the theoretical lower-bound for this problem.

My solution coded in C++ is available here. This problem was the most immediate application of Rolling Hash. It is a nice algorithm to have under our belt, since it is really fast to code. In an interview context, there are several similar questions:

- https://leetcode.com/problems/find-substring-with-given-hash-value/
  Classified as **hard**, but it is exactly the problem we just solved, except that on this one you're **forced to use rolling hash**;

- https://leetcode.com/problems/subtree-of-another-tree/
  Could appear totally unrelated, but once you figure a way to **serialize the tree as a string**, you can reuse the solution to the problem we just solved. **Takeaway message**: even if a problem seems totally unrelated to strings, perhaps there is a way to **translate** it into a string problem and re-use well-known algorithms, instead of memorizing tons of data-specific solutions (e.g. Merkle Tree can also solve this problem in $O(n+m)$).

2.2.2 Palindromic Sub-strings

Source: leetcode

Problem: Given a string s, return the number of palindromic sub-strings in it. A string is said to be palindromic when it reads the same backward as forward (e.g. abba).

Solution:

A **brute-force** approach is to consider every possible sub-string in s and check whether it is a palindrome or not. The number of possible sub-strings is $O(n^2)$, while checking if a string is a palindrome without additional knowledge require $O(n)$ time, thus we end up with $O(n^3)$ time.

Now, let's focus on the **definition of palindrome**. Every string of length 1 is trivially a palindrome. For those longer than 2, we can come up with a **recursive definition**. A string s with length n is a palindrome iff:

- $s[0] = s[n-1]$;
- $s[1...n-2]$ is a palindrome.

Given this definition, one could think to go for a **DP solution**: by memorizing whether a particular sub-string is palindrome, we can reduce the time required for the palindromic property check to $O(1)$, lowering the overall algorithm cost to $O(n^2)$ (which would be an accepted solution from leetcode). Still, faster solutions exist: we can exploit **rolling hash** to further reduce the time complexity to $O(nlogn)$.

Let's go back to the **palindrome definition**. Let t be the reversed string of s: then, **a string is a palindrome iff** $s = t$. This is also true for every sub-string $s[i...j]$: it is a palindrome iff $s[i...j] = t[j...i]$. In the previous problem, we showed that hashing can be used to check whether 2 strings are equal.

So, a **first solution to this problem using rolling hash** is to iterate over all the sub-strings $s[i...j]$ of s and count the number of cases where $h(s[i...j]) = h(t[j...i])$. This would still take $O(n^2)$, but we can locate **two phases that we need to work on**:

- We need to **retrieve the hash** of every sub-string of s and t. We will demonstrate that by **pre-processing** the strings in $O(n)$ we can later recover the hash values in $O(1)$;
- We can't iterate over every possible sub-string, as they are $O(n^2)$. We will demonstrate that **counting** the number of palindromic sub-strings can be done in $O(nlogn)$.

Let's see how to **improve the first phase**. The idea is the following:

- Pre-compute the **hash of every prefix** of s and t in $O(n)$ time, storing them in 2 arrays $prehash$ and $posthash$, where $prehash[i+1]$ contains the hash of $s[0...i]$;
- If we want to know the **hash of the sub-string** $s[i...j]$, we can use the following **formula**:

$$h(s[i...j], B, M) = (prehash[j] - prehash[i-1] \cdot B^{j-i+1}) \,\%\, M$$

  Since we are using a rolling hash, we only need to remove the hash of the first i characters of the string, multiplied by the appropriate base.

Next, the **second phase** can be improved using **binary search**.

The initial idea is similar to the "**expand around center**" approach: a palindrome string can either have odd or even length. For odd strings we define its **center** as its central character (e.g. ab<u>c</u>ba), while for even strings it is defined as the two characters in the middle (e.g. a<u>bb</u>a). **Consider every center** of a potential palindrome in s (so, all the individual characters and all the pairs of consecutive characters). We can count the number of palindromes by **expanding outwards** from the selected center (1 char to the left and 1 char to the right at a time) and checking at each step if the 2 new added characters match. The process will stop as soon as the new added characters don't match anymore or we reach the end of the string.

The "expand around center" approach would require $O(n^2)$ time, but thanks to the work done in the first phase we can do it in $O(nlogn)$, by slightly changing it. The idea is to simply use **binary search** to find the max expansion from each center:

- If s[i...j] is palindrome (checked using rolling hashes), then try to **expand** the range;
- Else, **shrink** the range.

Let's test this final idea with an **example**: let `s = "xyzABCDCBAx"` and consider the center to be the character `D`. From this center, we can expand at most by 4 characters, since on the left we have 6 characters and on the right we have 4 characters, setting the search space for the binary search to `[0...4]`.

This first value tested for the expansion's range by the binary search is `2`, because of `(0+4)/2 = 2`. As we can see, the expanded string `BCDCB` is palindrome (and so are all its sub-expansions). The check will be done by computing the hashes of `BCDCB` and its reverse - using the pre-computed prefix hash arrays - and comparing them.

Thus, the search space now becomes `[2...4]`, and the next value to be tested is `3` (because of `(2+4)/2 = 3`). The expanded string `ABCDCBA` is once more a palindrome.

Finally, we can see that if we try to expand more and test the value `4` (because of `(3+4)/2 = 4`), we will find that the expansion `zABCDCBAx` is not palindrome. The search can't proceed anymore, and we've found that the palindromes having center `D` are exactly 4 (value of the expansion + 1).

My solution coded in C++ is available here. Note that while being an already fast and accepted solution, a faster $O(n)$ solution does exists for this problem: the Google's "Manacher's algorithm".

2.2.3 Longest Common Sub-path

Source: `leetcode`

Problem: Given n integer arrays of variable length, find their longest common sub-array.

Solution:

A **brute-force** approach is to consider every possible sub-array s of the smallest array, and check whether it is contained in all the other arrays (by searching only for sub-arrays of length equal to s and comparing the strings char by char). Among all the possible matches, we will return the size of the largest one. The time complexity of this solution will be $O(nmM^3)$, with m being the size of the smallest array and M being the size of the largest. This will of course result in a TLE error: we need a faster approach.

A solution involving rolling hash is to **binary search for the answer**. Indeed, we will demonstrate that given a value k we can verify if there exists a common sub-array with size k in $O(mn)$.

We could **compute the hash value of all the sub-arrays** having size k in $O(mn)$ using the sliding-window technique previously introduced. Then, if the arrays have **at least one sub-array (hash) in common**, we have found a solution. This can be accomplished using two `unordered_sets` to find and keep the common elements between consecutive arrays. If we manage to iterate all over the arrays and reach the end with some **non-empty** `unordered_set`, we can conclude that an answer exists.

The full algorithm will result in a $O(nmlogm)$ time complexity.

My solution coded in C++ is available here. For this specific problem, test cases were carefully designed and it is important to put extra care in the selection of the base and modulo numbers. An alternative to further reduce the number of collisions is to employ **two different rolling hashes** (with different bases and modulos).

2.2.4 Minimal Rotation

Source: `cses`

Problem: Given a string s, find its lexicographically minimal rotation. A rotation of a string s is the string $s_{k+1}s_{k+2}\cdots s_n s_1 s_2 \cdots s_k$ for some k $(0 \leq k < n)$, with n = length of the string s.

Solution:

A **brute-force** approach is to iterate through successive rotations while keeping track of the most lexicographically minimal rotation encountered. As every comparison between 2 strings will require $O(n)$ time worst case, the brute-force ends up requiring $O(n^2)$ time.

The **bottleneck** can be found in the **comparison between two strings**: in this exercise, we will see how to reduce the cost for this specific operation - using rolling hash - down to $O(logn)$.

Consider two possible rotation of the string s: s_1 and s_2. Thus, to determine the minimum out of the two, we can find their **largest common prefix** and we can **compare the two strings' respective next character**.

We previously saw that by paying $O(n)$ time for a pre-processing phase, we can later recover the hash of every sub-string in $O(1)$. Then, **binary search** could be exploited to search the **largest common prefix**: we just need to **search for the length** of it. Given an hypothetical length of the largest common prefix, we can check in $O(1)$ if the prefix of s_1 and s_2 is equal (thanks to the pre-processing phase and rolling hash). If it is, then we will need to increase the hypothesis, otherwise we will decrease it.

Let's visualize this idea with an **example**:

$s =$

B	C	A	B	D	A	B	D	A

Let's suppose we reached a certain step where we need to compare these two specific rotations:

$s_1 =$

A	B	D	A	B	D	A	B	C

$s_2 =$

A	B	C	A	B	D	A	B	D

The search space for the binary search is `[0...8]`. Thus, the first value to test is `4` (because of `(0+8)/2 = 4`).

$s_1 =$

A	B	D	A	B	D	A	B	C

$s_2 =$

A	B	C	A	B	D	A	B	D

Rolling hash will be used to compare the hashes between the two prefixes, finding that they are not equal. Thus, the right value of the range must be reduced to 3. The range is now `[0...3]` and `1` is picked.

$s_1 =$

A	B	D	A	B	D	A	B	C

$s_2 =$

A	B	C	A	B	D	A	B	D

This time, the two prefixes are equal. As a consequence, the left value of the range must be increased to 2. The range is now `[2...3]` and `2` is picked.

$s_1 =$

A	B	D	A	B	D	A	B	C

$s_2 =$

A	B	C	A	B	D	A	B	D

The prefixes are once more different, changing the range to `[2...1]`. This concludes the search: we have found that the largest common prefix is the string `"AB"` and we can compare the two strings' respective next character - `D` and `C` - to conclude that s_2 is lexicographically smaller.

$s_1 =$

A	B	D	A	B	D	A	B	C

$s_2 =$

A	B	C	A	B	D	A	B	D

The full algorithm will result in an $O(nlogn)$ time complexity. My solution coded in C++ is available here. Note that while being an already fast and accepted solution, a faster $O(n)$ solution does exists for this problem: the "Booth's Algorithm".