String Hashing

- Sidhant Bansal

Credits: CP-Algorithms, Codeforces Blogs

Agenda

1. Motivation
2. What is String Hashing
3. How to do String Hashing
4. Analysis
5. Applications
6. Pitfalls

Motivation

Why?

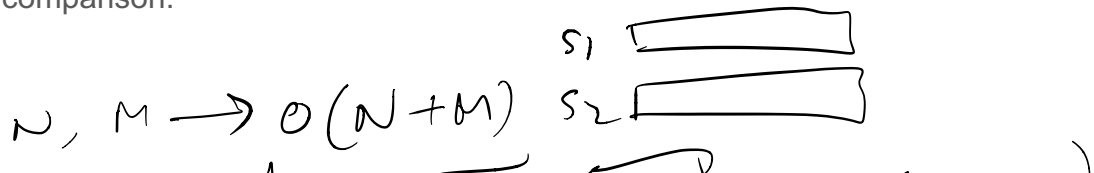
$O(N^2)$ / $O(N^3)$ → TLE.

String Hashing lets you most often than not “optimise” a brute-force approach to a string related question.

How ?

Well at a very high-level, think of it like it speeds-up the “comparison of any two strings”.

Earlier comparing two strings (are equal or not or which one is smaller/larger) used to be **O(N)** now it is **O(1) (NOTE: correct with high probability, NOT 100% of the times)** or **O(logN)** for smaller/larger lexical comparison.



O(1) or O(log N)

Most Powerful String DS

There are many string DS:

- KMP, Suffix Array, Suffix Trees, Palindromic Trees, Aho-Cohrasik, 2 ...

However all problems ≤ Medium-Hard difficulty of these Tags are most often than not solvable by String Hashing as well. (As an alternate solution)

So why doesn’t everyone use it?

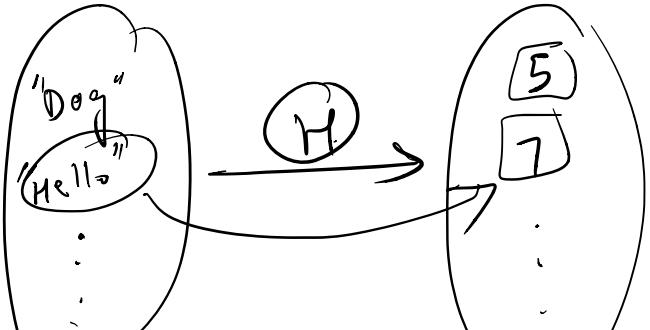
- It can be slightly tedious to implement. And you have to be smart about the modulos, primes, etc you pick.
- Basically your implementation needs to be clean for it to pass (i.e not give WA neither give TLE)

What is String Hashing

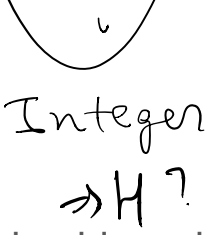
What is String Hashing?

The goal is to be able to have a "fast" method to map any string to an integer, such that

- If $s_1 != s_2$, then $H(s_1) != H(s_2)$
- If $s_1 == s_2$, then $H(s_1) == H(s_2)$
- Given **s**, we should be able to compute **H(s)** fast (i.e. in O(1) or O(logN))
- **H(s)** should be an integer/long long/pair of ints/pair of long long → O(1) in size.



5 != 7 → "Dog" != "Hello"



Poll Question

- Do you think there exists a string hashing which can provide us with all that power mentioned in the previous slide? (True / False)

But wait...

- Let's say we consider all alphabet-based strings of length 100 and boiling them down to an integer.
- How many unique strings can we have? (26^{100})
- And how many different integers do we have? ($2^{32} \sim 10^9$)

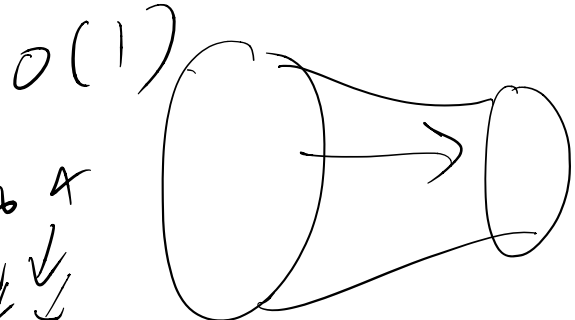
But you can see $26^{100} >> 10^9$

So how can we ensure that "if $\mathbf{s_1 != s_2}$, then $\mathbf{H(s_1) != H(s_2)}$"

Fixed definition of String Hashing

The goal is to be able to have a "fast" method to map any string to an integer, such that

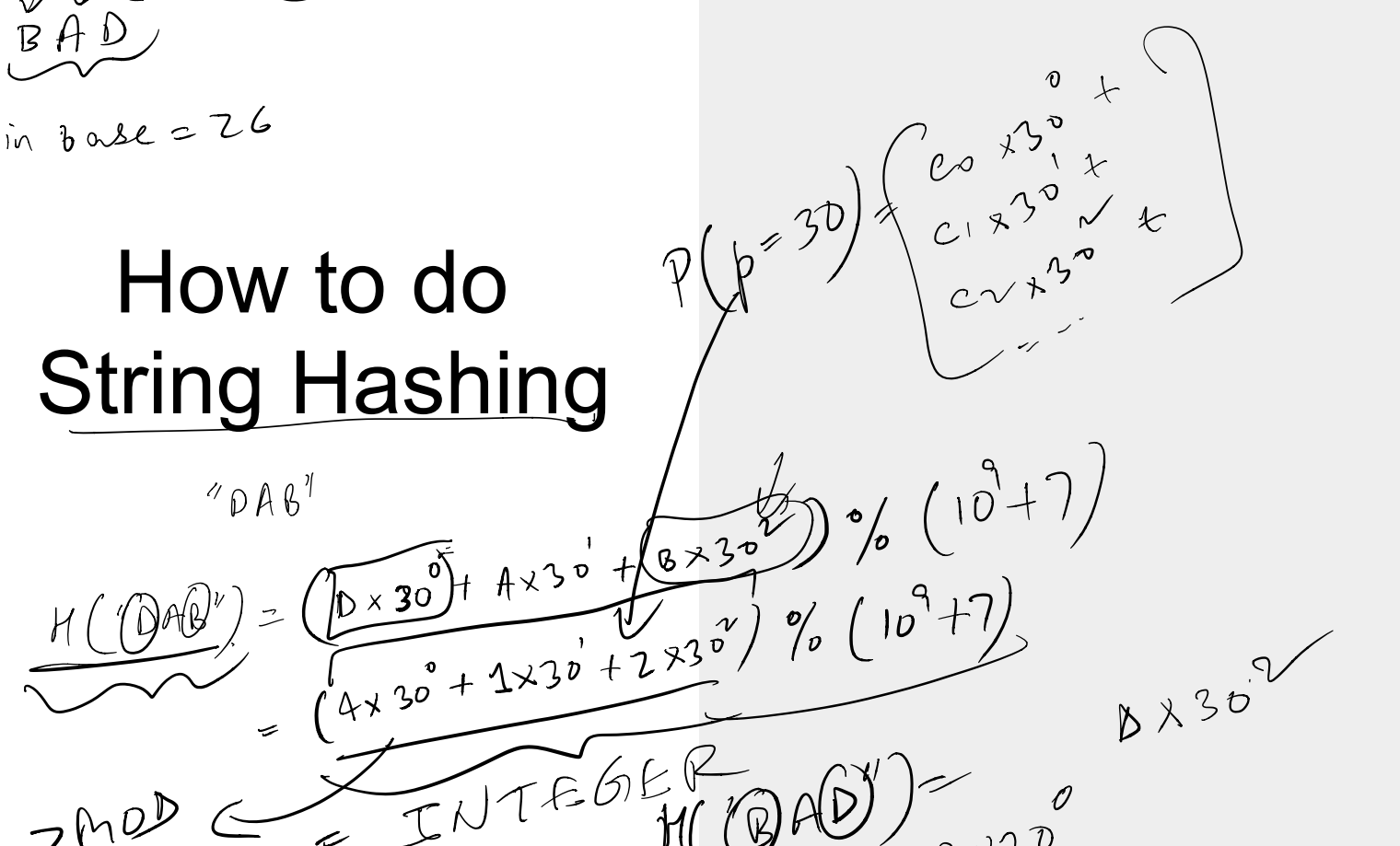
- If s_1 != s_2, then **$H(s_1)$!= $H(s_2)$ (with high probability)**
- If s_1 == s_2, then **$H(s_1)$ == $H(s_2)$ (with 100% probability)**
- Given **s**, we should be able to compute **H(s)** fast (i.e. in O(1) or O(logN))
- **H(s)** should be an integer/long long/pair of ints/pair of long long



BAD

in base = 26

How to do String Hashing



Polynomial Rolling Hash Technique

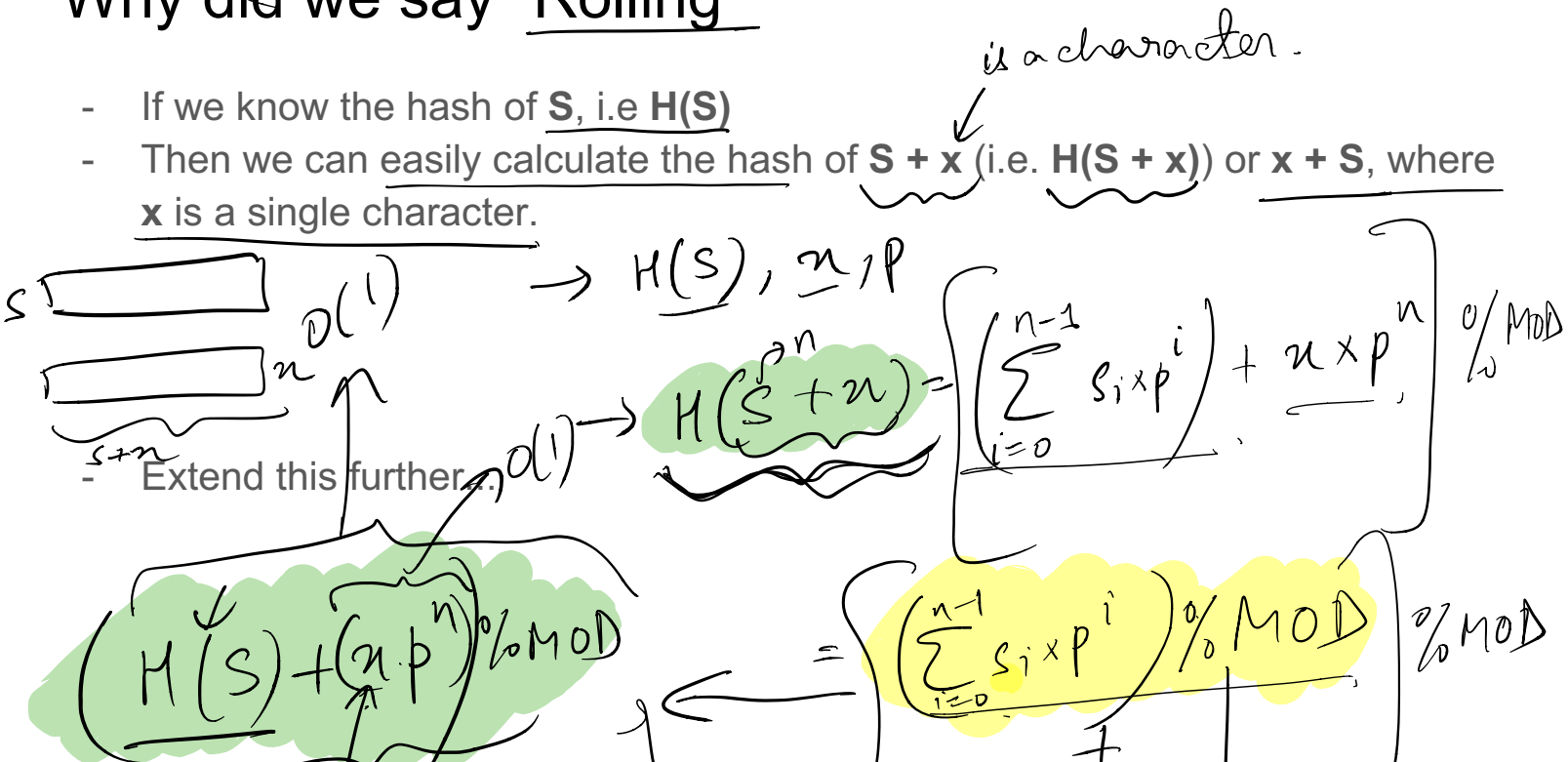
- Given a string **S** = $s_0\, s_1 \ldots s_{N-1}$

$$H(S) = \left(\sum_{i=0}^{N-1} s_i * p^i\right) \% MOD$$

- The hash is a polynomial in **p**
- This hash depends on:
  - The length of the string
  - The order of the characters
  - And is contained within an integer (because of modulo arithmetic)

Why did we say "Rolling"

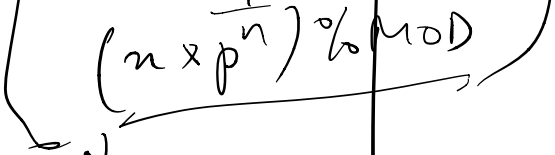
- If we know the hash of **S**, i.e **H(S)**
- Then we can easily calculate the hash of **S + x** (i.e. **H(S + x)**) or **x + S**, where **x** is a single character.

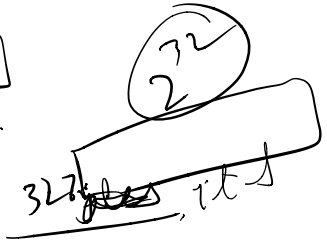


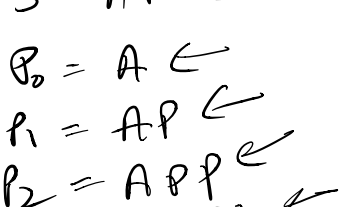
- Extend this further...

Why did we say "Rolling"

- If we know the hash of every prefix of string **S**.
- Let the prefixes of string **S** be P_0, P_1, P_2, … P_{N-1}
- Example: S = "ABCDEF"
  - P_0 = "A", P_1 = "AB", P_2 = "ABC", ..
- Then if we know **$H(P_{i-1})$** then we can calculate **$H(P_i)$** in O(1)
- So given **$H(P_i)$** for all i.
- Now we can calculate hash of any substring of S, i.e S[i...j].
  - But how ?







P_3 = APPLE
P_4 = APPLE

Explain how to get **H(S[i...j])** ?

$$H(S[i \dots j]) = (\sum_{k=i}^{\rightarrow j} s_k * p^{k-i}) \% MOD$$

$$= \left(\frac{H(P_j) - H(P_{i-1})}{p^i} \right) \% MOD = ((H(P_j) - H(P_{i-1}))p^{-i}) \% MOD$$

APPLE

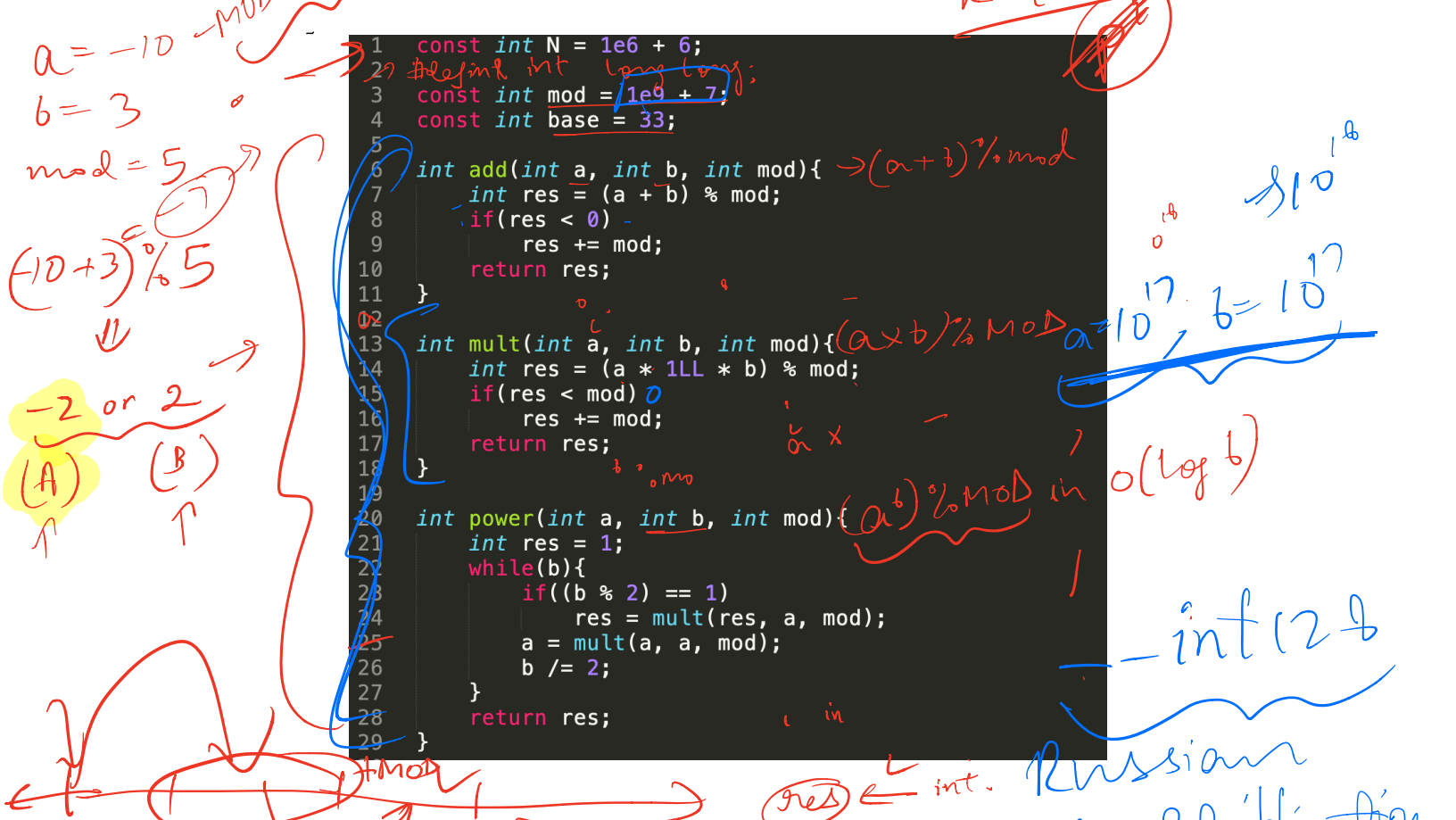
$(P \times 30^0 + P \times 30^1 + L \times 30^2) \% MOD$

$[H(P_i), \forall i]$

$O(1)$

$[H(j) - H(i-1)] \cdot p^{-i} \% MOD$

$\sum_{k=i}^{j} s_k \cdot p^{k-i}$




```cpp
const int N = 1e6 + 6;
#define int long long;
const int mod = 1e9 + 7;
const int base = 33;

int add(int a, int b, int mod){
    int res = (a + b) % mod;
    if(res < 0)
        res += mod;
    return res;
}

int mult(int a, int b, int mod){
    int res = (a * 1LL * b) % mod;
    if(res < mod)
        res += mod;
    return res;
}

int power(int a, int b, int mod){
    int res = 1;
    while(b){
        if((b % 2) == 1)
            res = mult(res, a, mod);
        a = mult(a, a, mod);
        b /= 2;
    }
    return res;
}
```

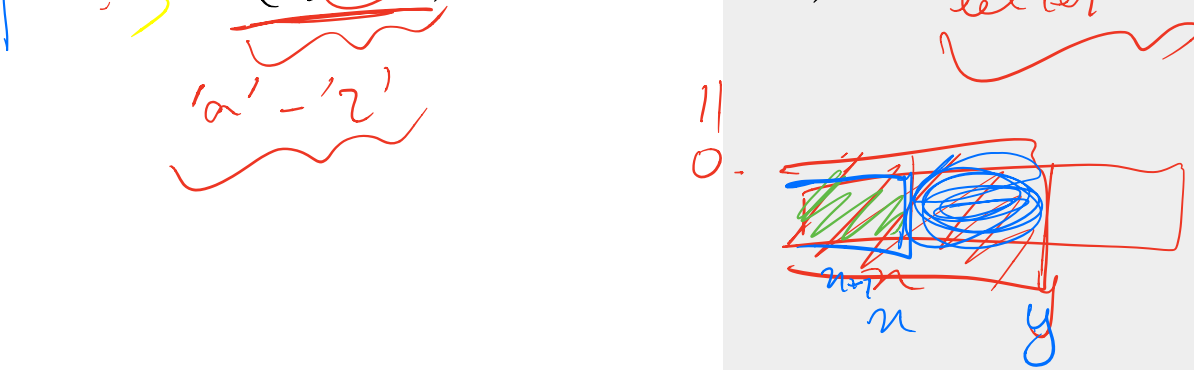
```cpp
int pw[N];
int inv[N];
int hash[N];

void precalc() {
    pw[0] = 1;
    for(int i = 1; i < N; i++)
        pw[i] = mult(pw[i - 1], base, mod);

    int pw_inv = power(base , mod - 2 , mod);
    inv[0] = 1;
    for(int i = 1; i < N; i++)
        inv[i] = mult(inv[i - 1], pw_inv, mod);
}

void build(string s){
    int n = s.length()
    for(int i = 0; i < n ; ++i){
        hash[i] = add((i == 0) 0 ?• hash[i - 1], mult(pw[i], s[i] - 'a' + 1, mod), mod);
    }
}

int getHash(int x , int y){
    int res = add(hash[y], (x == 0) ? 0 : -hash[x - 1], mod);
    res = mult(res , (x == 0) ? 1 : inv[x -1], mod);
    return res;
}
```



Analysis

Analysis

Fix the values first.

- Firstly, your p = base > #alphabets. So generally keep it > 26 or > 36.
- Your MOD should be a prime, so that calculation of (1/base) under MOD is easy.

When mod is prime: $\left(\frac{1}{base}\right) \%MOD = (base^{MOD-2})\%MOD$ ← Proof.

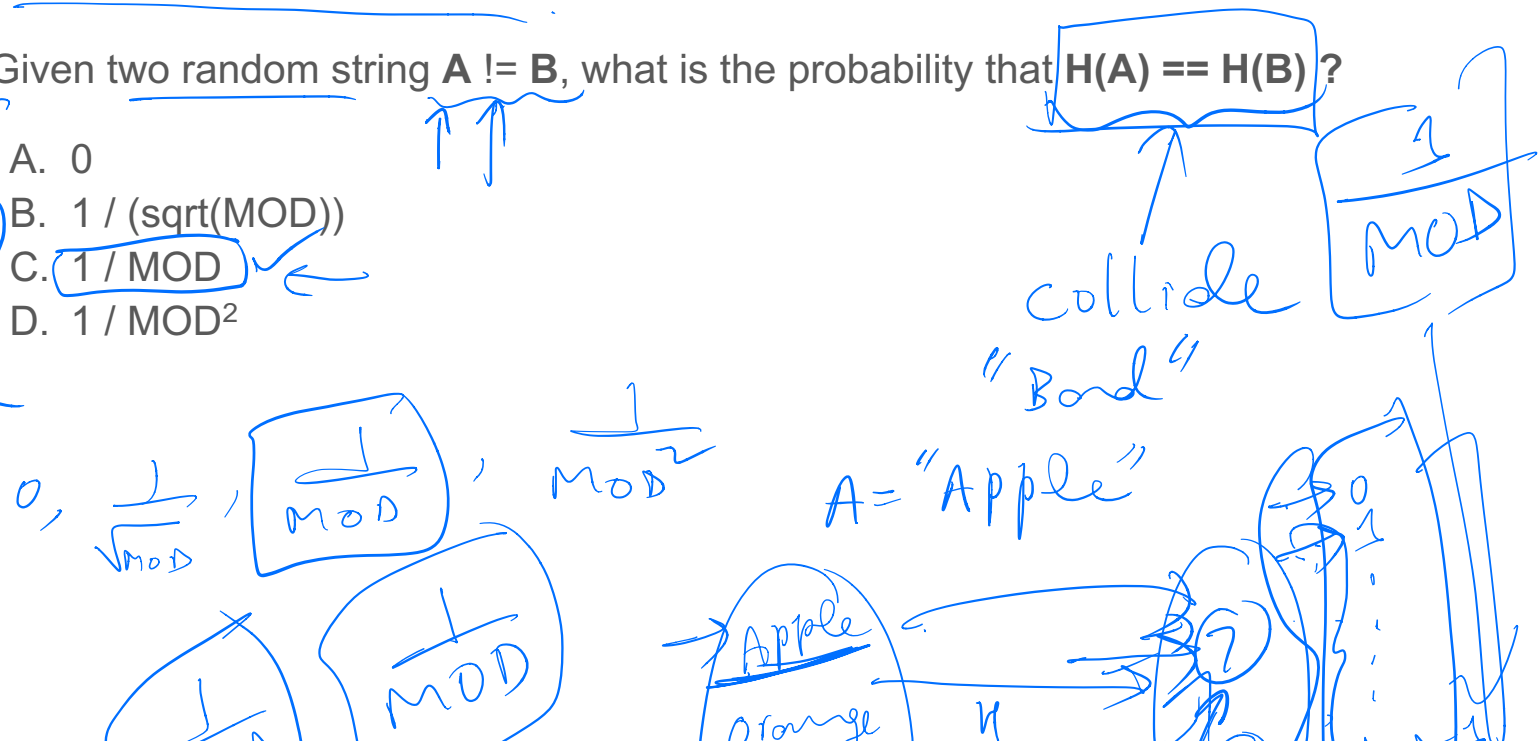
(Using Euler's Theorem)

```
34
35  void precalc() {
36      pw[0] = 1;
37      for(int i = 1; i < N; i++)
38          pw[i] = mult(pw[i - 1], base, mod);
39
40      int pw_inv = power(base , mod - 2 , mod);
41      inv[0] = 1;
42      for(int i = 1; i < N; i++)
43          inv[i] = mult(inv[i - 1], pw_inv, mod);
44  }
```

Why high probability? (Poll Question)

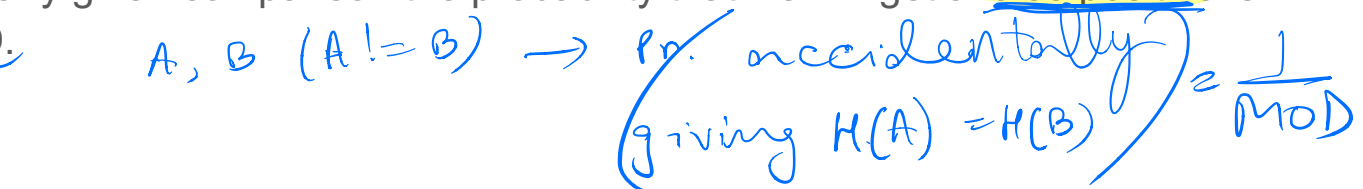
Given two random string **A** != **B**, what is the probability that **H(A) == H(B)** ?

A. 0
B. 1 / (sqrt(MOD))
C. 1 / MOD
D. 1 / MOD^2

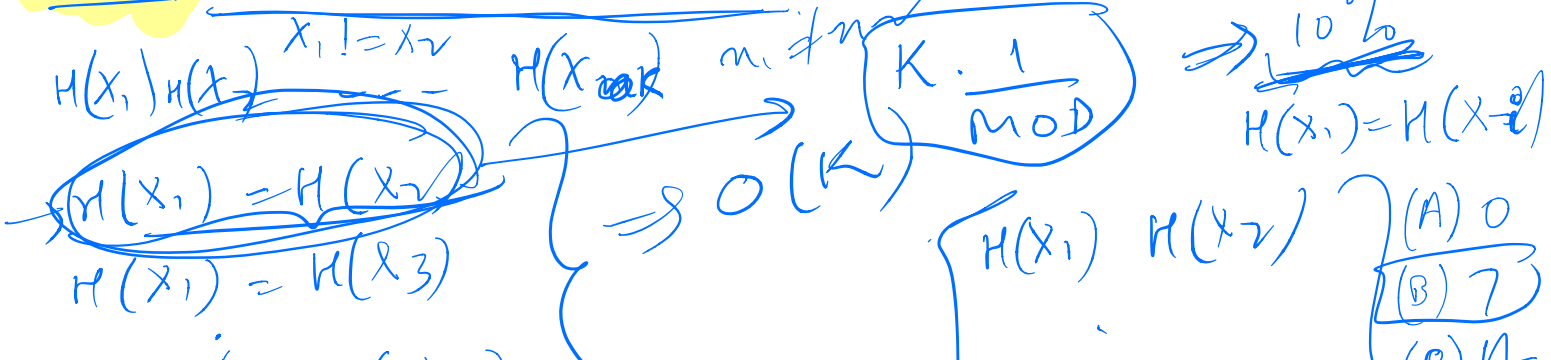


But wait...it amplifies

So for any given comparison the probability that we will get a false positive is 1/MOD.



If we do K such comparisons in our algorithm, then on average (i.e. in expectation) K/MOD of them will give us false positive.



Example where it may break

- Let's say your MOD = $\frac{1}{10^9 + 7} \approx 10^{-9}$
- Maybe you hashed a string **S** of length 10^6 and put all the **100** length substrings of **S** in a set to find the number of distinct **100** length substrings
- How many comparisons did you do?
- Umm, roughly $(10^6 - 100 + 1)^2 \sim 10^{12}$
- And each comparison had a fail probability of ~ 10^{-9}
- So roughly how many failed comparisons did you do? $10^{12-9} = 1000 > 0$.

This is bad...

The Fix

Use two (or more) mods.

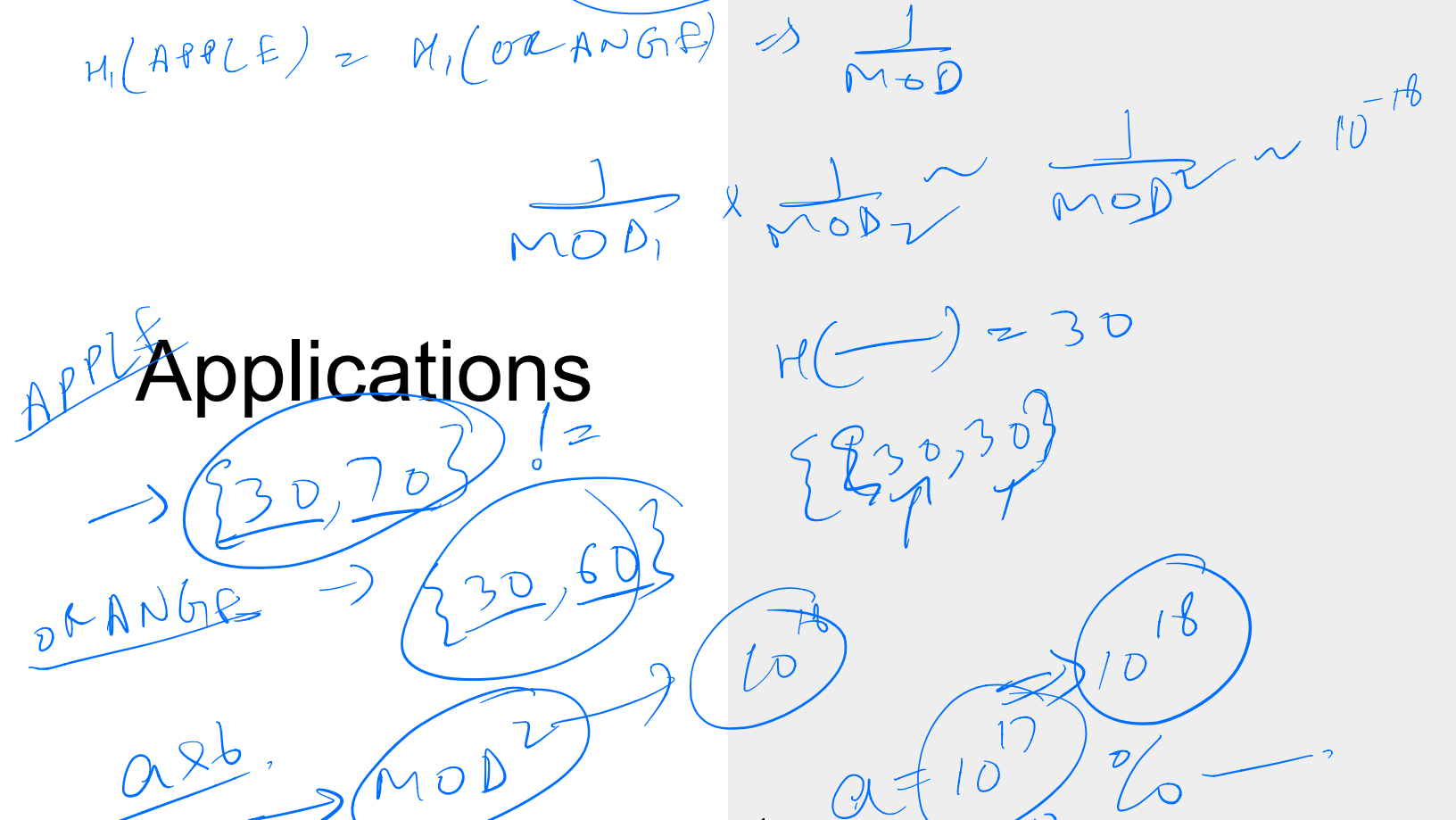
Let **H(S) = (H_1(S), H_2(S))** using two different modulos

Ex. $MOD_1 = 10^9 + 7$, $MOD_2 = 10^9 + 9$

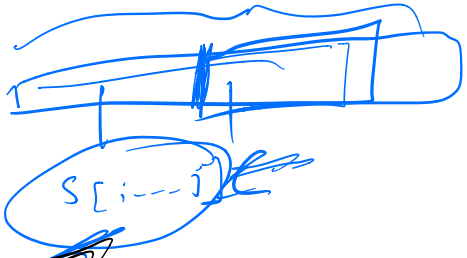
Now probability of accidentally matching for two strings is roughly 10^{-18}.

So # of false positive ~ $10^{12 - 18} \sim 10^{-6}$ (super low)

Applications



Given a string **S**, find the largest substring which occurs at-least twice in the string ?



$O(N^2 \cdot N \cdot N) \rightarrow O(N^4)$

Is there a substring of length K, that occurs ≥ 2 times.

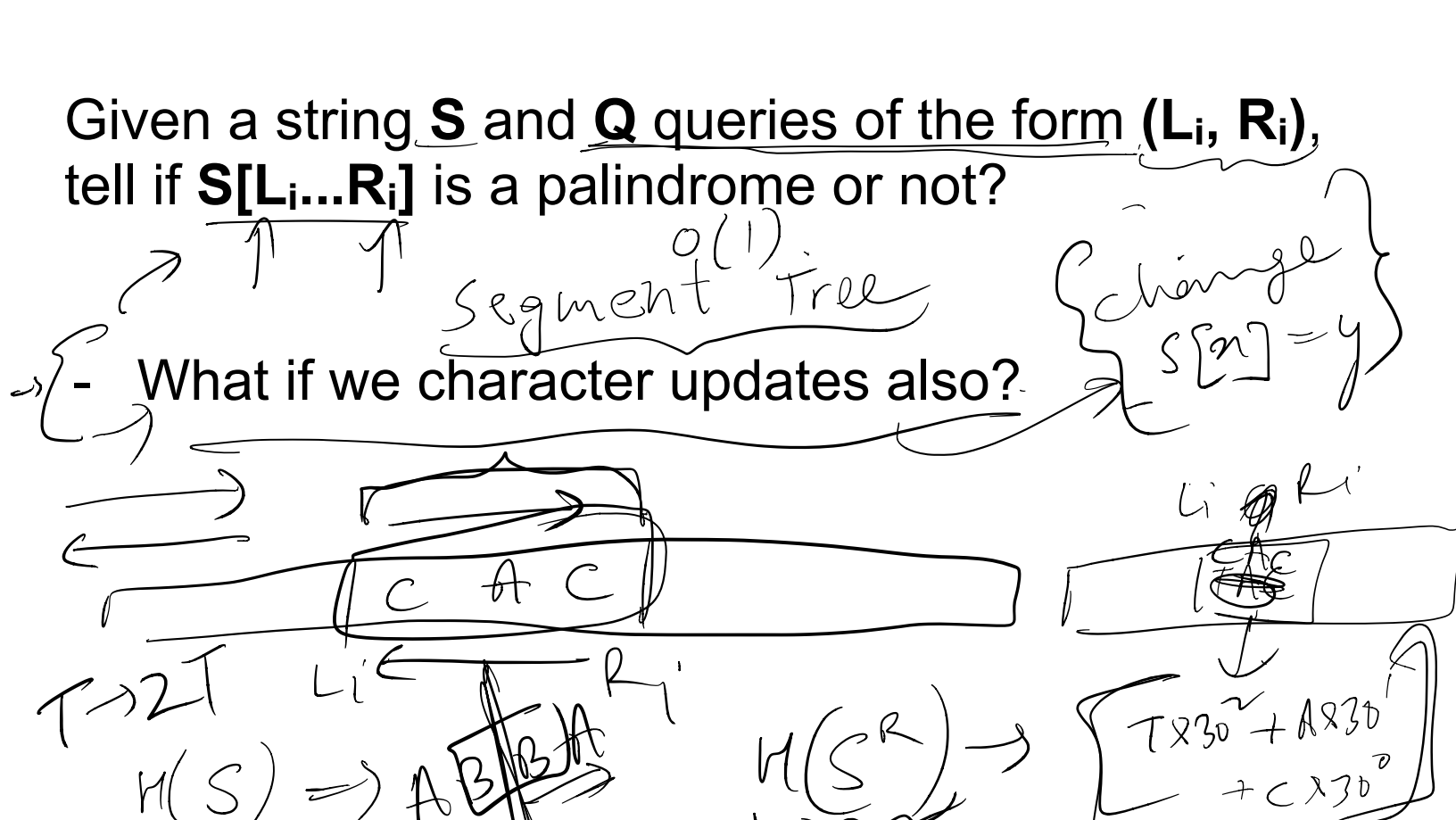
Binary Searching

$O(N \log N)$

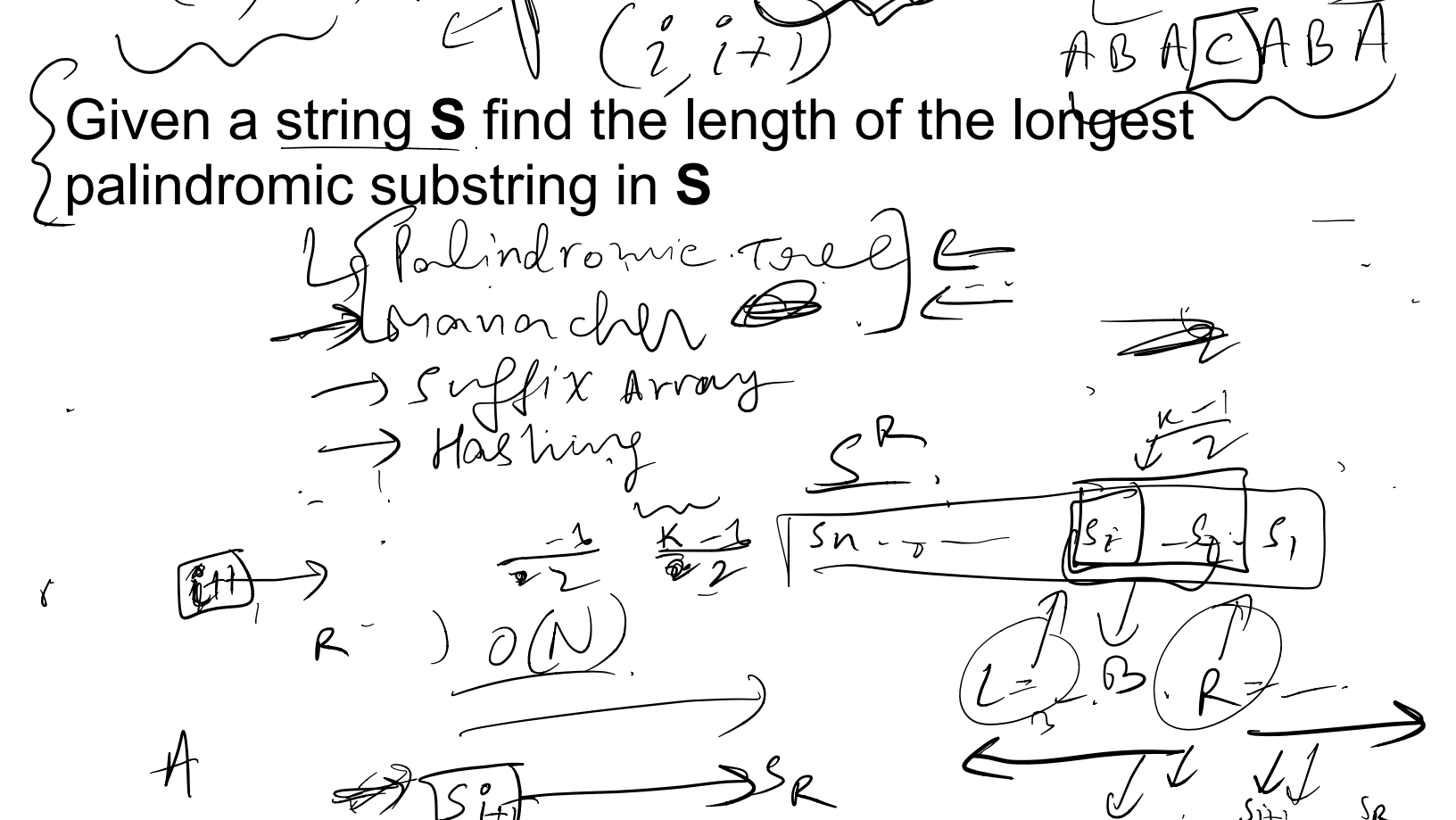
Yes/No. → $O(N \log N)$

Given a string **S** and **Q** queries of the form **(L_i, R_i)**, tell if **$S[L_i...R_i]$** is a palindrome or not?

- What if we character updates also?



Given a string **S** find the length of the longest palindromic substring in **S**



Largest common prefix of two strings **S** and **T**. If we are allowed to swap two characters in one of the strings ?

Pitfalls

Pitfalls

- Slow implementation (unnecessary modulo operations)
- Off-by-one errors
- Always picking $10^9 + 7$ and $10^9 + 9$ as your mods and 26/30/36 as your base.
  Well then you can be hacked by the author.

  Our randomisation argument wasn't completely correct (It was a simplified version and had some loopholes).
  Ideally you want to **randomly** pick the base as well as the mod. See this **Codeforces Blog Post** for precise details.

  So rule of thumb is: Submit the code once with BASE = 33/36 (some typical value) and $MOD_1 = 10^9 + 7$ and $MOD_2 = 10^9 + 9$. If it WAs then either pick some other primes from your codebook or during runtime randomise the picking of base and mod from a hard-coded list.

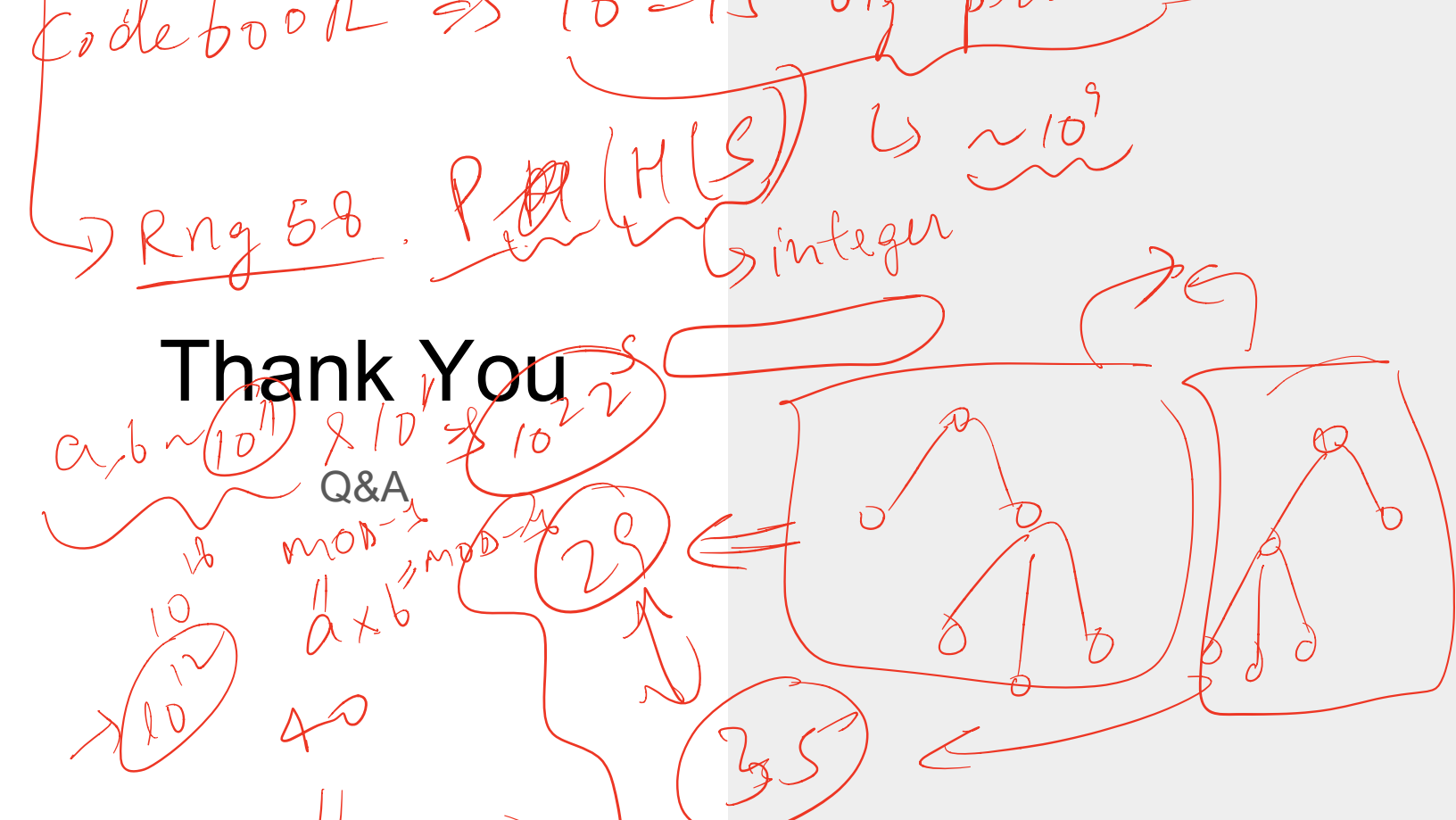
Further Readings

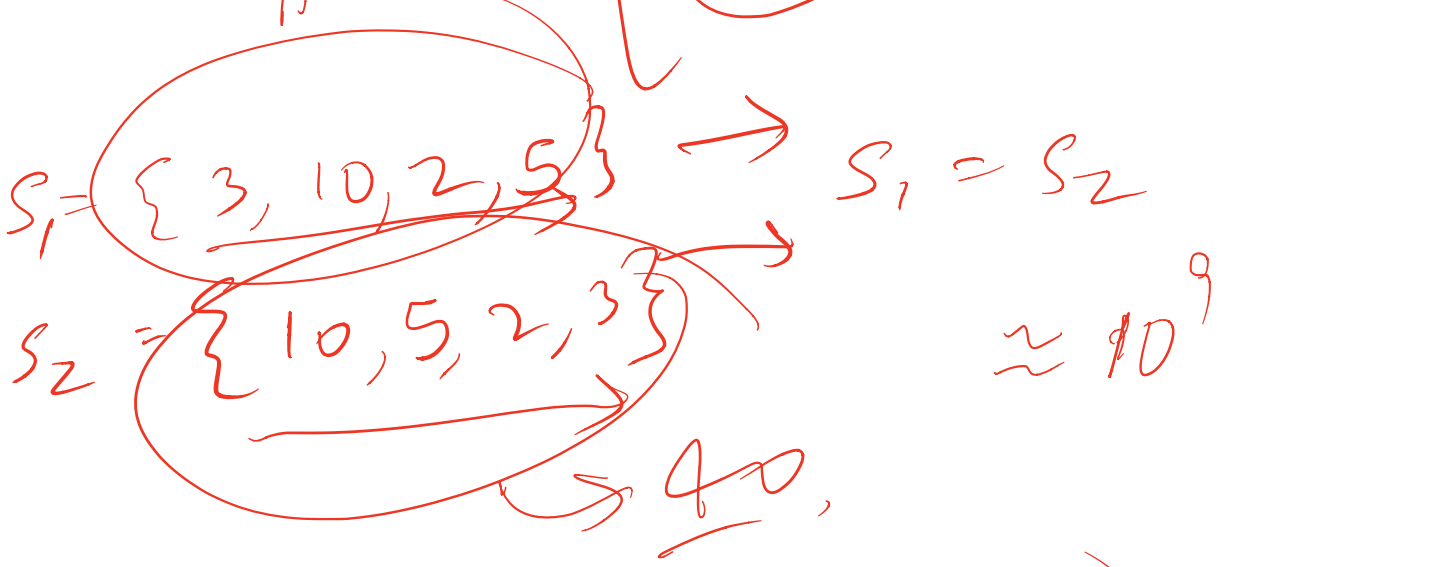
- https://codeforces.com/blog/entry/60445 (More in-depth)
- https://www.quora.com/q/threadsiiithyderabad/String-Hashing-for-competitive-programming (Quora Post)
- https://cp-algorithms.com/string/string-hashing.html (CP Algorithms)

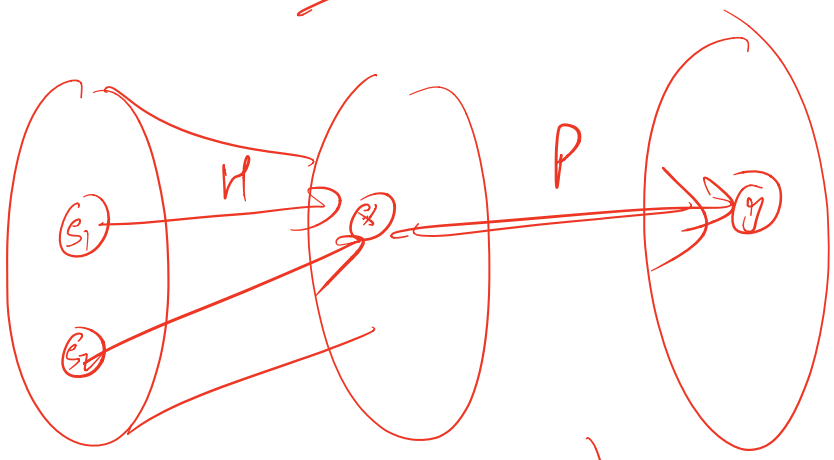
- https://codeforces.com/blog/entry/4898 (Anti-Hash tests)
- https://codeforces.com/blog/entry/60442 (Extensive discussion on anti-hash testing)
- http://rng-58.blogspot.com/2017/02/hashing-and-probability-of-collision.html (Hashing for unordered elements)

Thank You

Q&A

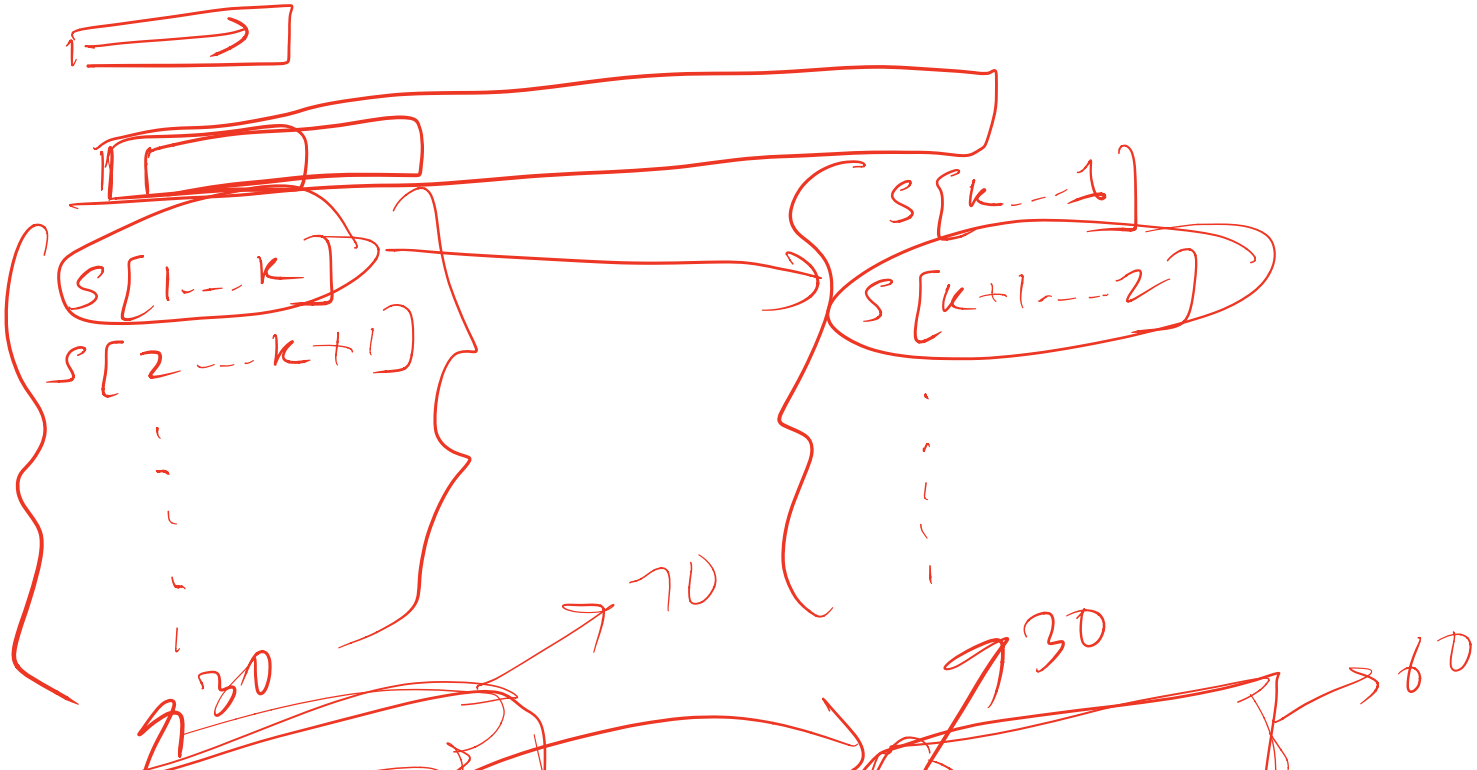


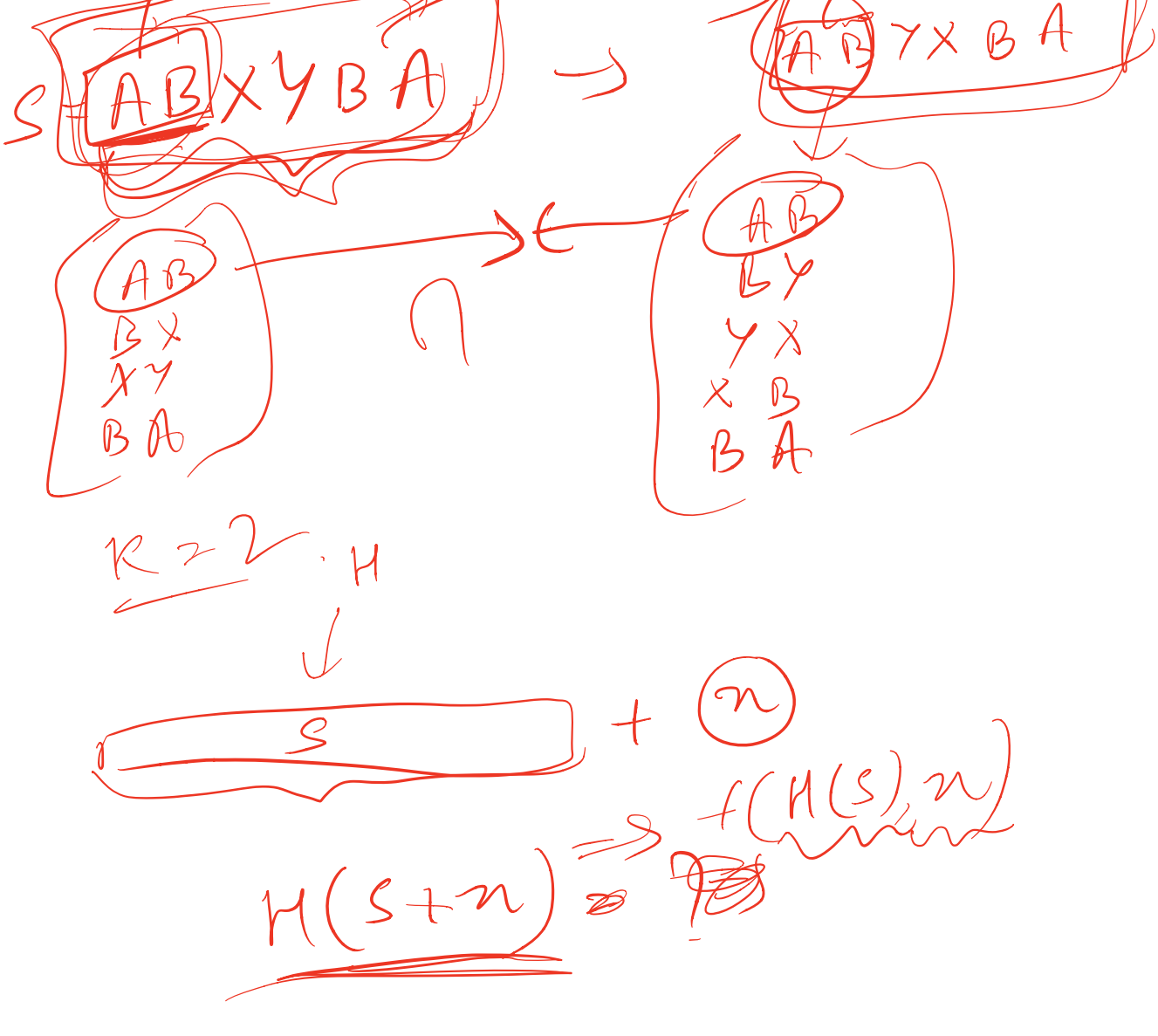




$$H(S_1) = x = H(S_2)$$

$$P(H(S_1)) = y = P(H(S_2))$$





$R = 2$. H

$f(H(s), n)$

$H(s+n) \Rightarrow$