Strings

String

What is a string? A string is an array of characters, so in fact, all algorithms on strings work on any arrays, and sometimes even on more complex objects.

Hash

First of all, let's talk about what a hash is.

Informally speaking, a hash is a mapping of some complex object into simpler objects (possibly with a loss of precision).

A collision is what we call a situation when two different elements have the same hash.

Why?

In fact, hashes have a lot of applications:

1) Checking the condition of a file. We can calculate hash and after transition we can check hash. If it’s broken h(a) != hash.
2) Transferring sufficiently vulnerable data (the hash function should be recoverable with some special data).

Why?

In fact, hashes have a lot of applications:

1) From the previous point we can also say about cryptography.
2) Hash functions sometimes allow checking complex things for equality with precision to a certain property.
3) Hash table

Hash

At the moment, we will consider as a hash, a function that maps a string to a number, and a very restricted one(for example, from the range of natural numbers [0, 10^9 + 7]).

Polynomial hash

Usually, in the context of algorithms, polynomial hash is discussed, but we may touch on others as well; however, let's start with it.

Polynomial hash, as the name suggests, associates a string with a certain polynomial and uses it to compute the hash for the string.

Polynomial hash

There are two main types of Polynomial hash:

1) reversed - (s_0 * p^{n - 1} + s_1 * p^{n - 2} + .. + s_{n - 1} * p^0) % MOD
2) direct - (s_0 * p^0 + s_1 * p^1 + .. + s_{n - 1} * p^{n - 1}) % MOD

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[END OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

```cpp
#include <bits/stdc++.h>

using namespace std;

const long long ALPHA = 29, MOD = 1e9 + 7;

int main() {
    // deg[] = {1, ALPHA, ALPHA^2, ALPHA^3, ...}
    // h[] = {0, s[0], s[0] * ALPHA + s[1], s[0] * ALPHA^2 + s[1] * ALPHA + s[2], ...}

    string s;
    cin >> s;
    int n = s.length();
    vector<long long> h(n + 1), deg(n + 1);
    h[0] = 0;
    deg[0] = 1;
    for (int i = 0; i < n; i++) {
        h[i + 1] = (h[i] * ALPHA + s[i]) % MOD;
        deg[i + 1] = (deg[i] * ALPHA) % MOD;
    }

    for (int i = 0; i <= n; i++) {
        cout << i << " : " << h[i] << " " << deg[i] << "\n";
    }
    return 0;
}
```

Success #stdin #stdout 0.01s 5308KB

stdin

ababa

stdout

```
0 : 0 1
1 : 97 29
2 : 2911 841
3 : 84516 24389
4 : 2451062 707281
5 : 71080895 20511149
```

Polynomial hash

h(s[0..0]) = 97

h(s[0..1]) = h(s[0..0]) * Alpha + s[1] = 97 * 29 + 98 = 2911

```cpp
#include <bits/stdc++.h>

using namespace std;

const long long ALPHA = 29, MOD = 1e9 + 7;

int main() {
    // deg[] = {1, ALPHA, ALPHA^2, ALPHA^3, ...}
    // h[] = {s[0], s[0] + s[1] * P, s[0] + s[1] * P + s[2] * P^2, ...}

    string s;
    cin >> s;
    int n = s.length();
    vector<long long> h(n + 1), deg(n + 1);
    deg[0] = 1;
    for (int i = 0; i < n; i++) {
        deg[i + 1] = (deg[i] * ALPHA) % MOD;
        h[i + 1] = (h[i] + s[i] * deg[i]) % MOD;
    }

    for (int i = 0; i <= n; i++) {
        cout << i << " : " << h[i] << " " << deg[i] << "\n";
    }
    return 0;
}
```

Success #stdin #stdout 0.01s 5288KB

stdin

ababa

stdout

```
0 : 0 1
1 : 97 29
2 : 2939 841
3 : 84516 24389
4 : 2474638 707281
5 : 71080895 20511149
```

Polynomial hash

At first glance, it seems that these methods are very similar, and this is indeed true until we consider specific applications.

We often want a hash to be able to quickly find the hash of a substring in O(1) time.

reversed hash substring

for reversed hash it’s quite simple.

reversed_hash[l..r] = reversed_hash[r + 1] - reversed_hash[l] *

P[r - l + 1]

Reversed hash substring

Math is pretty hard, so let's start with an example.

“ababa”

$h(s) = h(s_{0..4}) = h(ababa) = \{0, 97, 2911, 84516, 2451062, 71080895\}$

$h(s_{0..2}) = h(aba) = 84516$

$h(s_{2..4}) = h(aba) = ?$ (but must be $= 84516$)

Reversed hash substring

reversed_hash[l..r] = reversed_hash[r + 1] - reversed_hash[l] *

P[r - l + 1]

h(s_{2..4}) = 71080895 - 2911 * 24389(P[4 - 2 + 1]) = 84516

And now we will prove it.)

Reversed hash substring

reversed_hash - (s_0 * p^{n - 1} + s_1 * p^{n - 2} + .. + s_{n - 1} * p^0)

reversed_hash[l] - (s_0 * p^{l - 1} + s_1 * p^{n - 2} + .. + s_{l - 1} * p^0)

reversed_hash[r + 1] - (s_0 * p^{r} + s_1 * p^{n - 2} + .. + s_{r} * p^0)

Reversed hash substring

$$\text{reversed\_hash}[l..r] = \text{reversed\_hash}[r + 1] - \text{reversed\_hash}[l] * P[r - l + 1] = (s_0 * p^{r} + s_1 * p^{r - 1} + .. + s_{r} * p^0) - (s_0 * p^{l - 1} + s_1 * p^{l - 2} + .. + s_{l - 1} * p^0) * P[r - l + 1]$$

$$reversed\_hash[l \cdots r] = reversed\_hash[r+1] - reversed\_hash[l] * P[r-l+1] = (s_0 * p^r + s_1 * p^{r-1} + .. + s_r * p^0) - (s_0 * p^{l-1} + s_1 * p^{l-2} + .. + s_{l-1} * p^0) * P[r-l+1]$$

Reversed hash substring

$$(s_0 * p^{r} + s_1 * p^{r - 1} + .. + s_{r} * p^0) - (s_0 * p^{l - 1} * p^{r - l + 1} + s_1 * p^{n - 2} * p^{r - l + 1} + .. + s_{l - 1} * p^0 * p^{r - l + 1}) = s_0 * p^{r} + .. + s_{l - 1} * p^{r - l + 1} - s_0 * p^{(l - 1 + r - l + 1)} - .. - s_{l - 1} * p^{0 + r - l + 1} + s_l * p^{r - l} + .. s_r = s_l * p^{r - l} + .. s_r$$

$$(s_0 * p^r + s_1 * p^{r-1} + .. + s_r * p^0) - (s_0 * p^{l-1} * p^{r-l+1} + s_1 * p^{n-2} * p^{r-l+1} + .. + s_{l-1} * p^0 * p^{r-l+1}) = s_0 * p^r + .. + s_{l-1} * p^{r-l+1} - s_0 * p^{(l-1+r-l+1)} - .. - s_{l-1} * p^{0+r-l+1} + s_l * p^{r-l} + ..s_r = s_l * p^{r-l} + ..s_r$$

```
    // deg[] = {1, ALPHA, ALPHA^2, ALPHA^3, ...}
    // h[] = {0, s[0], s[0] * ALPHA + s[1], s[0] * ALPHA^2 + s[1] * ALPHA + s[2], ...}
    string s;
    cin >> s;
    int n = s.length();
    vector<long long> h(n + 1), deg(n + 1);
    h[0] = 0;
    deg[0] = 1;
    for (int i = 0; i < n; i++) {
        h[i + 1] = (h[i] * ALPHA + s[i]) % MOD;
        deg[i + 1] = (deg[i] * ALPHA) % MOD;
    }

    auto get_hash = [&]( int l, int r ) { // [l..r]
      return h[r + 1] - h[l] * deg[r - l + 1];
    };

    cout << get_hash(0, 2) << " " << get_hash(2, 4);
    return 0;
```

Success #stdin #stdout 0.01s 5288KB

stdin

ababa

stdout

84516 84516

Direct hash substring

For direct hash it’s harder

direct_hash[l..r] = (h[r + 1] - h[l]) / P[l]

Direct hash substring

Let’s check again example - “ababa”

h(s) = h(s_{0..4}) = h(ababa) = {0, 97, 2939, 84516, 2474638, 71080895}

h(s_{0..2}) = h(aba) = 84516

h(s_{2..4}) = h(aba) = ? (but must be = 84516)

Direct hash substring

$h[2..4] = (h[5] - h[2]) / P(2) = (71080895 - 2939) / 841 = 84516$

Direct hash substring

direct_hash - $(s_0 * p^0 + s_1 * p^1 + .. + s_{n - 1} * p^{n - 1})$

direct_hash[l] - $(s_0 * p^0 + s_1 * p^1 + .. + s_{l - 1} * p^{l - 1})$

direct_hash[r + 1] - $(s_0 * p^0 + s_1 * p^1 + .. + s_{r} * p^{r})$

Direct hash substring

direct_hash[l..r] = (direct_hash[r + 1] - direct_hash[l]) / P[l] = ((s_0 * p^0 + s_1 * p^1 + .. + s_{r} * p^{r}) - (s_0 * p^0 + s_1 * p^1 + .. + s_{l - 1} * p^{l - 1})) / P[l]

Direct hash substring

$$direct\_hash[l..r] = (direct\_hash[r + 1] - direct\_hash[l]) / P[l] = (s_0 * p^0 - s_0 * p^0 + s_1 * p^1 - s_1 * p^1 + .. + s_{l - 1} * p^{l - 1} - s_{l - 1} * p^{l - 1} + s_{l} * p^{l} + \ldots + s_{r} * p^{r}) / P[l] = (s_{l} * p^{l} + \ldots + s_{r} * p^{r}) / P[l] = (s_l * p^0 + .. + s_{r} * p^{r - l})$$

$$(s_0 * p^0 - s_0 * p^0 + s_1 * p^1 - s_1 * p^1 + .. + s_{l-1} * p^{l-1} - s_{l-1} * p^{l-1} + s_l * p^l + \ldots + s_r * p^r)/P[l] = (s_l * p^l + \ldots + s_r * p^r)/P[l] = (s_l * p^0 + .. + s_r * p^{r-l})$$

Direct hash substring

And why did I say that this method is more complex? Because we don't know how to divide quickly by a prime modulus, so here we usually multiply by some large power.

direct_hash[l..r] = (h[r + 1] - h[l]) * P[BIG_NUMBER - l]

```
    // deg[] = {1, ALPHA, ALPHA^2, ALPHA^3, ...}
    // h[] = {s[0], s[0] + s[1] * P, s[0] + s[1] * P + s[2] * P^2, ...}

    string s;
    cin >> s;
    int n = s.length();
    vector<long long> h(n + 1), deg(n + 1);
    deg[0] = 1;
    for (int i = 0; i < n; i++) {
        deg[i + 1] = (deg[i] * ALPHA) % MOD;
        h[i + 1] = (h[i] + s[i] * deg[i + 1]) % MOD;
    }

    auto get_hash = [&]( int l, int r ) { // [l..r]
        if (l == 0) {
            return h[r + 1];
        }
        return h[r + 1] - h[l];
    };

    cout << get_hash(0, 2) << " " << get_hash(2, 4) << "\n";
    cout << (get_hash(0, 2) * deg[n]) % MOD << " " << (get_hash(2, 4) * deg[n - 2]) % MOD << "\n";
    return 0;
```

Success #stdin #stdout 0s 5300KB

stdin

ababa

stdout

```
2450964 61260710
87445732 87445732
```

Polynomial hash

So what to choose?

I don't know; mostly pick what you like more.

Generally, reversed hash is better, but I've used direct hash all my life.

So it's just a matter of taste.

Polynomial hash

Both hashes have common important requirements:

1) p should be no less than the length of the alphabet (for example, for the English language p >= 26), otherwise, we will get a collision h(aa) = 1 * 1 + 1 * 5 = 6 * 1 = h(f) when p = 5.
2) Mod is usually chosen to be prime (because if it is not coprime with p and any number from 2 to the length of the alphabet, it's easy to find a collision).
3) Mod is usually chosen to be around n * n * 10, where n is the number of strings, the proof - the birthday paradox.

Polynomial hash

"So, what about collisions, which we discussed for a reason.

It is asserted that with the correct choice of MOD and p, there will be no collisions with a high probability, which is why they are usually ignored.

But if you really don't trust in luck, you can take several hashes (with different MOD and p) or compare the original strings as well when the hashes are equal.

Birthday problem

In a group consisting of 23 or more people, the probability that at least two people have the same birthday exceeds 50%.

In fact, here I am making a very strong assumption that people's birthdays are independent and uniformly distributed, which is not the case, but it's easier to bait on such a fact.

Formally

You need to add $O(\sqrt{n})$ random numbers in range [1, n] to a multiset so that any two will coincide with high probability.

Proof

$f(n, d)$ is the probability that in a group of n people, no one has matching birthdays. We will assume that birthdays are independently and uniformly distributed from 1 to d.

Proof

f(n, d) = (1 - 1/d) * (1 - 2/d) * .. * (1 - (n - 1) / d).

This formula arises in the following way:

Suppose we have chosen a particular day for the first person, then with a probability of 1/d, the second person will have the same birthday, and with a probability of (1 - 1/d), they will not.

For the second person, by the same logic, we get (1 - 2/d).

Proof

We can rigorously prove this using the Taylor series for the exponent, or we can go to Wolfram and check.

Input interpretation

$$\left\{\prod_{k=1}^{n-1}\left(1-\frac{k}{d}\right),\ n=23,\ d=365\right\}$$

$$d\left(-\left(-\frac{1}{d}\right)^{n}\right)(1-d)_{n-1} = \frac{36\,997\,978\,566\,217\,959\,340\,182\,499\,134\,166\,757\,044\,383\,351\,847\,256\,064}{75\,091\,883\,268\,515\,350\,125\,426\,207\,425\,223\,147\,563\,269\,805\,908\,203\,125}$$

Exact result

$$\frac{36\,997\,978\,566\,217\,959\,340\,182\,499\,134\,166\,757\,044\,383\,351\,847\,256\,064}{75\,091\,883\,268\,515\,350\,125\,426\,207\,425\,223\,147\,563\,269\,805\,908\,203\,125}$$

(irreducible)

Number line

Task

Why do we need these hashes at all? We've touched on a lot of mathematics, but so far we haven't discussed the purpose.

Well, let's solve the first problem - to find the number of unique strings.

Example

string = abac

unique = {a, b, c, ab, ba, aba, bac, abac}

Solution

Let's find the hashes of all substrings and then put them in a set; this way, we will find the number of different strings.

Check s < t

Suppose we have two strings, and we want to quickly compare them for < using hashes. For example, we want to quickly sort the strings.

That is, for instance, the string 'abcf' < 'abde'.

Check s < t

First of all, let’s understand that there is such an i that s[0..i - 1] = t[0..i - 1] and s[i] != t[i]; accordingly, if we find such an i, then to compare two strings it is enough to compare s[i] and t[i].

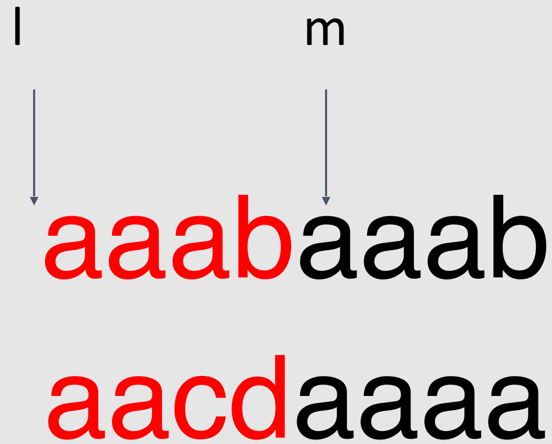
For simplicity, let's assume that len(s) = len(t) = n; if this is not, then we need to add extra check in solution.

Check s < t

How can we find such an i? For example, we can use binary search. Let's do a binary search on the function (are the hashes of s[0..i - 1] equal to t[0..i - 1]).

The left boundary (definitely fits) - 0

The right boundary (definitely not fits) - n + 1.


l
m
r
aaabaaab
aacdaaaa

l m r

aaabaaab

aacdaaaa

l m r

aaabaaab

aacdaaaa

l r

aaabaaab

aacdaaaa

Task

Let's now solve the next problem. We have an initially empty set of strings and there are three types of queries:

1) Add a string to the set of strings.
2) Remove a string from the set of strings.
3) Check if a string is in the set."

Solution

Let's maintain a set of unique hashes.

Actually, for the first type of query, we will add an element to the set.

For the second type of query, we will remove it.

For the third type of query, we will also remove it.

insert(abc), rev_h(abc) = 84518

s = {84518}

insert(aba), rev_h(aba) = 84516

s = {84518, 84516}

insert(ba), rev_h(ba) = 2939

s = {2939, 84518, 84516}

insert(ba), rev_h(ba) = 2939

s = {2939, 84518, 84516}

remove(ba), rev_h(ba) = 2939

s = {84518, 84516}

find(ba), rev_h(ba) = 2939, no

s = {84518, 84516}

Separate chaining

Important: any hash function

h(s) = s_0 + s_1 + .. + s_n

Separate chaining

Actually, a hash table is usually used for such purposes.

This structure stores all the different strings for each hash, but we pay for it with memory.

Because we need to allocate O(HASH_MOD) memory cells.

HASH_MOD = 8, strings = {}

0	1	2	3	4	5	6	7
{}	{}	{}	{}	{}	{}	{}	{}

add(aba), h[aba] = 1, strings = {“aba”}

0	1	2	3	4	5	6	7
{}	{aba}	{}	{}	{}	{}	{}	{}

add(aca), h[aca] = 1, strings = {“aba”, “aca”}

0	1	2	3	4	5	6	7
{}	{aba, aca}	{}	{}	{}	{}	{}	{}

?ada, h[ada] = 1, strings = {“aba”, “aca”}, hash_table[1] = {“aba”, “aca”},

answer = no

0	1	2	3	4	5	6	7
{}	{aba, aca}	{}	{}	{}	{}	{}	{}

?bda, h[bda] = 2, strings = {“aba”, “aca”}, hash_table[2] = {}, answer = no

0	1	2	3	4	5	6	7
{}	{aba, aca}	{}	{}	{}	{}	{}	{}

Details

This is only the first version of implementing a hash table, but let's stop here and discuss all its aspects.

Collision

First - what to do with collisions? As we have already mentioned due to the birthday paradox, collisions are quite likely.

In the case where we have n strings and len elements in the hash_table, since the strings have a sufficiently random hash, we can expect that each cell should contain O(n / len) elements. We will call the quantity O(n / len) the load factor a.

Expected time

Although some queries may take a very long time if you are unlucky, on average, the queries will work in O(1) if the load factor is good enough. Acceptable figures of load factor alpha should range around 0.6 to 0.75.

Therefore, all operations with random strings will work in O(1).

Some practical example

unordered_map - c++

dict - pyhton

unordered_map.rehash()

unordered_map.rehash(random(p1, p2, p3, p4))

std::unordered_map<Key,T,Hash,KeyEqual,Allocator>::rehash

```
void rehash( size_type count );
```
(since C++11)

Changes the number of buckets to a value n that is not less than `count` and satisfies `n >= size() / max_load_factor()`, then rehashes the container, i.e. puts the elements into appropriate buckets considering that total number of buckets has changed.

Parameters

count - lower bound for the new number of buckets

Return value

(none)

Complexity

Average case linear in the size of the container, worst case quadratic.

Notes

`rehash(0)` may be used to force an unconditional rehash, such as after suspension of automatic rehashing by temporarily increasing `max_load_factor()`.

See also

reserve (C++11)	reserves space for at least the specified number of elements and regenerates the hash table (public member function)

z-function

Let there be a string s of length n. Then the Z-function of this string is an array of length n, the i-th element of which is equal to the greatest number of characters starting from position i that match the first characters of the string s.

In other words, z[i] is the longest common prefix of the string s and its i-th suffix.

aaabaab

z-function

string - aaaaa

z[0] = any, z[1] = 4(common(aaaaa, aaaa) = aaaa), z[2] = 3(common(aaaaa, aaa) = aaa), z[3] = 2(common(aaaaa, aa) = aa), z[4] = 1(common(aaaaa, a) = a).

z-function

string - aaabaab

z[0] = any,

z[1] = 2(common(aaabaab, aabaab) = aa),

z[2] = 1(common(aaabaab, abaab) = a),

z-function

z[3] = 0(common(aaabaab, baab) = {}),

z[4] = 2(common(aaabaab, aab) = aa),

z[5] = 1(common(aaabaab, ab) = a),

z[6] = (common(aaabaab, b) = {}).

z-function

string - abacaba

z[0] = any,

z[1] = 0(common(abacaba, bacaba) = {}),

z[2] = 1(common(abacaba, acaba) = a),

z[3] = 0(common(abacaba, caba) = {}),

z[4] = 3(common(abacaba, aba) = aba),

z[5] = 0(common(abacaba, ba) = ba),

z[6] = 1(common(abacaba, a) = a).

z-function

Since we know the hashes, we could solve it with them, but let's come up with something more interesting.

Let's start with a trivial algorithm.

We will simply iterate over the answer and check if it is possible to achieve it.

```cpp
#include <bits/stdc++.h>

using namespace std;

int main() {
    string s;
    cin >> s;
    int n = s.length();
    vector<int> z(n);

    for (int i = 1; i < n; i++) {
        // we try to find such answer j, that i + j < n (symbol still in string)
        // and s[ans] == s[i + ans](symbol in prefix of s ans substr of s from i)
        while (i + z[i] < n && s[z[i]] == s[i + z[i]]) {
            z[i]++;
        }
    }

    for (int i = 0; i < n; i++) {
        cout << z[i] << "\n";
    }
    return 0;
}
```

stdin

abacaba

stdout

```
0
0
1
0
3
0
1
```

z-function

How can we speed up such a solution? Suppose we have already found the z-function for all characters from 0 to $i - 1$.

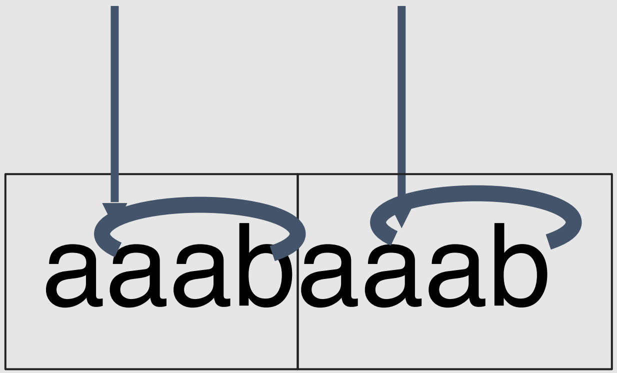
Also, suppose we know the rightmost segment $[L, R]$ such that it is equal to the prefix of the string, that is, $z[L] = R - L + 1$.

any, 2, 1, 0, 3; segment with biggest R = [4, 7]

aaab$\boxed{\text{aaab}}$

z-function

Then the answer for the i-th character is not less than the answer for the character min(z[i - L], R - i + 1). You can see a more detailed explanation in the following figure


aaab
aaab

z-function

But then we can take, as the initial value for z[i], min(r - i + 1, z[i - l]) and increase it if necessary.

z-function

This will work in linear time, as we will not increase the length of the rightmost segment more than n times.

Therefore, the final complexity is O(n).

```
int main() {
    string s;
    cin >> s;
    int n = s.length();
    vector<int> z(n);
    int left_bound = 0, right_bound = 0;
    for (int i = 1; i < n; i++) {
        // if i is inside the rightest known block
        if (i <= right_bound) {
            z[i] = min(right_bound - i + 1, z[i - left_bound]);
        }
        // if we can make answer bigger -> we will do it
        // if the position of end is still in string
        // and symbols are equal
        while (i + z[i] < n && s[z[i]] == s[i + z[i]]) {
            z[i] += 1;
        }
        // if we find righter block we change the answer
        if (i + z[i] - 1 > right_bound) {
            left_bound = i;
            right_bound = i + z[i] - 1;
        }
    }

    for (int i = 0; i < n; i++) {
        cout << z[i] << " ";
    }
    return 0;
```

Success #stdin #stdout 0.01s 5304KB

stdin

```
aaabaaab
```

stdout

```
0 2 1 0 4 2 1 0
```

z-function

How can we speed up such a solution? Let's assume we have already found the z-function for all characters from 0 to $i - 1$.

Let's also assume that we know the furthest to the right segment $[L, R]$ such that it is equal to the prefix of the string, meaning $z[L] = R - L + 1$.

Knuth–Morris–Pratt algorithm

Given a text t and a pattern s, the task is to find and output the positions of all occurrences of the string s in the text t.

For example, in the text “aababaab” the positions of the string “aa” are {0, 5}.

Let's construct the string s#t (# is a delimiter symbol, some character that is definitely not in either s or t).

Knuth–Morris–Pratt algorithm

Let's calculate the z-function for the string aa#aababaab.

[0, 1, 0, 2, 1, 0, 1, 0, 2, 1, 0], in those i where the z-function equals length(s), it means that the substring starting at the i-th character of length(s) is equal to the prefix of length(s), that is, it is equal to s.

```cpp
    string s, t;
    cin >> s >> t;
    string sum = t + '#' + s;
    int n = sum.length();
    vector<int> z(n);
    int left_bound = 0, right_bound = 0;
    for (int i = 1; i < n; i++) {
        if (i <= right_bound) {
            z[i] = min(right_bound - i + 1, z[i - left_bound]);
        }
        while (i + z[i] < n && sum[z[i]] == sum[i + z[i]]) {
            z[i] += 1;
        }
        if (i + z[i] - 1 > right_bound) {
            left_bound = i;
            right_bound = i + z[i] - 1;
        }
    }
    for (int i = 0; i < n; i++) {
        if (z[i] == t.length()) {
            cout << i - t.length() - 1 << "\n";
        }
    }
    return 0;
```

Success #stdin #stdout 0s 5288KB

stdin

aababaab
aa

stdout

```
0
5
```

pi-function

Given a string s, it is required to compute its prefix function, i.e., an array of numbers pi[0 .. n - 1], where pi[i] is defined as follows: it is the greatest length of the longest proper suffix of the substring s[0 .. i] that matches its prefix (a proper suffix means not coinciding with the entire string). In particular, the value of pi[0] is considered to be zero.

Example

For example, for the string "abcabcd" the prefix function is: [0, 0, 0, 1, 2, 3, 0], which means:

the strings "a", "ab", "abc", and "abcabcd" do not have a non-trivial prefix that matches a suffix;

the string "abca" has a prefix of length 1 that matches the suffix;

the string "abcab" has a prefix of length 2 that matches the suffix;

the string "abcabc" has a prefix of length 3 that matches the suffix;

aaabaaab

```cpp
#include <bits/stdc++.h>

using namespace std;

int main() {
    string s;
    cin >> s;
    int n = s.length();
    vector<int> pi(n);
    for (int i = 0; i < n; i++) {
        for (int k = 0; k <= i; k++) {
            if (s.substr(0, k) == s.substr(i - k + 1, k)) {
                pi[i] = k;
            }
        }
    }

    for (int i = 0; i < n; i++) {
        cout << pi[i] << " ";
    }
    return 0;
}
```

Success #stdin #stdout 0.01s 5304KB

stdin

```
aaabaaab
```

stdout

```
0 1 2 0 1 2 3 4
```

Faster

We've got an algorithm with a complexity of n^3, let's make it faster.

The first important note is that the value of pi[i + 1] is at most one more than the value of pi[i] for any i.

pi[i + 1] > pi[i] + 1

pi[i + 1] = 3, s(pi[i + 1]) = “aaa” = s(0..2)

aaabaaab

pi[i + 1] > pi[i] + 1 -> len(s(pi[i + 1]).pop_back) > pi[i]

pi[i + 1] = 3, s(pi[i + 1]).pop_back = “aa” = s(0..1)

$\boxed{\text{aaabaaa}}\text{b}$

Faster

We've got an algorithm with a complexity of n^2, since we only decrease pi[i] and therefore get n positions and for each a check in O(n).

The second important note is that let's say we have computed the value of the prefix function pi[i] for some i. Now, if s[i+1] = s[pi[i]], we can confidently say that pi[i+1] = pi[i] + 1.

pi[5] = 2, if s[6] = s[2] -> pi[6] = 3

aaabaaab

Faster

And what if s[i+1] != s[pi[i]], let's check s[pi[pi[i] - 1]].

Where did I get this formula from?)

Well, look, we know that s[0..pi[i] - 1] = s[i - pi[i] + 1 .. i], and also that s[0..pi[pi[i] - 1] - 1] = s[pi[i] - pi[pi[i] - 1] .. pi[i] - 1], which means s[0..pi[pi[i] - 1] - 1] = s[i - pi[pi[i] - 1] + 1 .. i].

(It seems I didn't mess up the formulas, it's better to look at an image here).

pi[5] = 2, if s[6] = s[2] -> pi[6] = 3

abaaababaabb

s[4] != s[10]

abaaababaabb

pi[3] = 1 => s[0..0] = s[3..3] = s[10..10]

abaaababaabb

s[1] = s[11] => pi[11] = 2

abaaababaabb

Faster

It turns out that such a solution already works in O(n) because either we increase pi[i] by 1 or decrease it by some amount. From this, it follows that in total, pi[i] will not increase more than n times, and therefore, they will not decrease more than n times either.

```cpp
#include <bits/stdc++.h>

using namespace std;

int main() {
    string s;
    cin >> s;
    int n = s.length();
    vector<int> pi(n);
    for (int i = 1; i < n; i++) {
        int j = pi[i - 1];
        while (j > 0 && s[i] != s[j]) {
            j = pi[j - 1];
        }
        if (s[i] == s[j]) {
            j++;
        }
        pi[i] = j;
    }
    for (int i = 0; i < n; i++) {
        cout << pi[i] << " ";
    }
    return 0;
}
```

Success #stdin #stdout 0s 5296KB

stdin

aaabaaab

stdout

```
0 1 2 0 1 2 3 4
```

Knuth–Morris–Pratt algorithm

Given a text t and a pattern s, the task is to find and output the positions of all occurrences of the string s in the text t.

For example, in the text “aababaab”, the positions of the string “aa” are {0, 5}.

Let's construct the string s#t (# is a separator symbol, a certain symbol that is definitely not present in either s or t).

Knuth–Morris–Pratt algorithm

Let's calculate the prefix function for the string aa#aababaab.

We get [0, 1, 0, 1, 2, 0, 1, 0, 1, 2, 0].

We will find all positions where the prefix function equals the length of the pattern. These are positions [4, 9] in the combined string, or [1, 6], meaning we have found the positions of the end of the occurrences of the pattern we are looking for.

```
int main() {
    string s, t;
    cin >> s >> t;
    string sum = t + '#' + s;
    int n = sum.length();
    vector<int> pi(n);
    for (int i = 1; i < n; i++) {
        int j = pi[i - 1];
        while (j > 0 && sum[i] != sum[j]) {
            j = pi[j - 1];
        }
        if (sum[i] == sum[j]) {
            j++;
        }
        pi[i] = j;
    }
    for (int i = 0; i < n; i++) {
        if (pi[i] == t.length()) {
            int pos_in_sum = i - t.length() - 1;
            cout << pos_in_sum - t.length() + 1 << " " << pos_in_sum << "\n";
        }
    }
    return 0;
}
```

Success #stdin #stdout 0.01s 5308KB

stdin

aababaab

aa

stdout

```
0 1
5 6
```

Task 1

You need to hash the string with precision to permutation.

That is, so that abac = acab = acba.

Hashing 1

a = 97, b = 98, a = 97, c = 99, hash = 97 + 98 + 97 + 99

a = 97, c = 99, a = 97, b = 98, hash = 97 + 98 + 97 + 99

hash(abac) = 391

hash(acab) = 391

hash(ac) = hash(bb) - it's bad

Hashing 2

a = random_1, b = random_2, a = random_3, c = random_4

random_1 + random_2 = random_3

Hashing 3

a = random_prime_1, b = random_prime_2, a = random_prime_1, c = random_prime_4

random_1 * random_2 != random_3

First right solution.

Hashing 4

abac

[2,1,1,0,0,0]

h([2,1,1,0,0,0])

Second right solution.

Xor is bad

3^3^3 = 3

3^3^3^3^3 = 3^3^3

3^3^3^3 = 3^3 = 0

a = 1, b = 2, c = 3

ab = 1^2 = 3 = c

Task 2

Let's create another realization of hash table.

HASH_MOD = 8,h(s) -> elem in [0..HASH_MOD]

insert(“aba”) h[“aba”] = 1

0	1	2	3	4	5	6	7
	aba						

insert(“aca”) h[“aca”] = 1

0	1	2	3	4	5	6	7
	aba	aca					

insert(“ada”) h[“ada”] = 1

0	1	2	3	4	5	6	7
	aba	aca	ada				

?afa h[“afa”] = 1

0	1	2	3	4	5	6	7
	aba	aca	ada				

insert(‘aka’) h[‘aka’] = 2

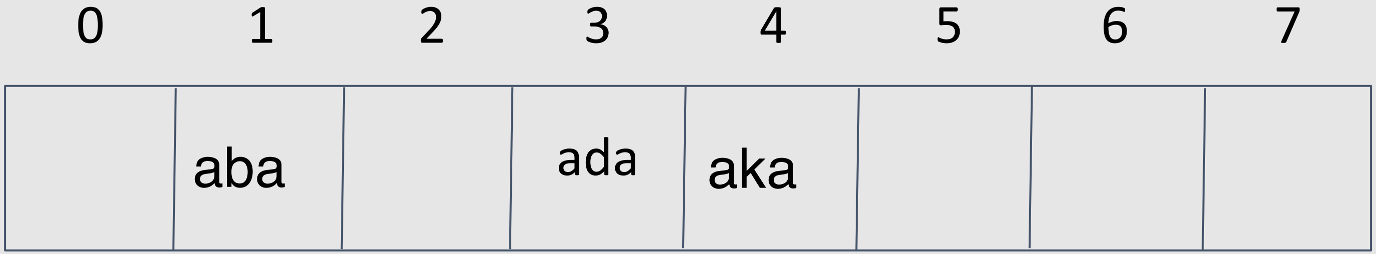
0	1	2	3	4	5	6	7
	aba	aca	ada	aka			

Improvement

step[1] = random_prime_number_1

step[2] = random_prime_number_2

remove('aca') h['aca'] = 1



?(‘ada’) h[‘ada’] = 1, no, but it’s exists

0	1	2	3	4	5	6	7
	aba		ada	aka			
0	1	1	1	1	0	0	0

IMHO

i use load balance = 0.5

If you have (len / n > load balance) -> rebuild hash table.

We can rebuild online

old

0	1	2	3	4	5	6	7
0	0	0	0		0	0	0

new

0	1	2	3	4	5	6	7	8	9	10	11
	aba	ada	aka								

Task 3

Finding amount of palindromic substring with hashes.

aaaaaa -> O(n^2) palindromic substring

radius - length of palindromic subparts

(aaaaa) -> radius = 2

(aaaa) -> radius = 2

Task 3

function - radius i is good

l = 0, because it’s always good

r = length(s), because it’s always bad

hash(mid-radius..mid) = hash(mid + 1..mid + radius)

bbbaaaaacac

Task 4

Reconstruct the string from the prefix function in O(n), assuming the alphabet is unlimited.

prefix function is: [0, 0, 0, 1, 2, 3, 0]

one of options is abcabcd

Task 4

prefix function is: [0, 0, 0, 1, 2, 3, 0]

if pi[i] = 0 we just put new symbol

if pi[i] != 0 we will put symbol s[pi[i] - 1]

because pi-function - is correct -> pi[i] is good

abcabcd

Task 5

Reconstruct the prefix function from the z function in O(n).

s = aaabaaab

z = [0 2 1 0 4 2 1 0]

pi = [0 1 2 0 1 2 3 4]

Task 5

s = aaabaaab

z = [0 2 1 0 4 2 1 0]

pi = [0 1 2 0 1 2 3 4]

z[4] = 4 -> [4, 8) = [0, 4)

pi[7] >= 4, pi[6] >= 3, pi[5] >= 2, pi[4] >= 1.

Task 5

we go from i = 0 .. n and check z[i]

if we already had some pi[j] filled, we don’t need to refill it.

Why -> z[i] can cover pi[j], but if pi[j] is filled, it’s already covered by some z[x] (x < i)

now u can get [i, j), but you have [x, j)

```
for(int i = 1; i < n; i++)
        if(Z[i])
                for(int j = Z[i] - 1; j >= 0 && !(P[i + j]); j--)
                        P[i + j] = j + 1;
```

Task 5

Finding the number of palindromes without using hashes.

All codes

z-function(naive) - https://ideone.com/58gcCa

z-function(good) - https://ideone.com/TQeOBN

pi-function(good) - https://ideone.com/bVQV0x

pi-function(naive) - https://ideone.com/GdoQr1

Knut(pi) - https://ideone.com/5pcA2Y

Knut(z) - https://ideone.com/4vaUUC