1. **Hashing**: The whole process
2. **Hash value/ code**: The index in the Hash Table for storing the value obtained after computing the Hash Function on the corresponding key.
3. **Hash Table**: The data structure associated with hashing in which keys are mapped with values stored in the array.
4. **Hash Function/ Hash**: The mathematical function to be applied on keys to obtain indexes for their corresponding values into the Hash Table.

These are the four Hash Functions we can choose based on the key being numeric or alphanumeric:

1. Division Method
2. Mid Square Method
3. Folding Method
4. Multiplication Method

1. Division Method:

Say that we have a Hash Table of size 'S', and we want to store a (key, value) pair in the Hash Table. The Hash Function, according to the Division method, would be:

1. H(key) = key mod M
* Here M is an integer value used for calculating the Hash value, and M should be greater than S. Sometimes, S is used as M.
* This is the simplest and easiest method to obtain a Hash value.
* The best practice is using this method when M is a prime number, as we can distribute all the keys uniformly.
* It is also fast as it requires only one computation - modulus.

**Let us now take an example to understand the cons of this method:**

Size of the Hash Table = 5 (M, S)

Key: Value pairs: {10: "Sudha", 11: "Venkat", 12: "Jeevani"}

**For every pair:**

1. {10: "Sudha"}
Key mod M = 10 mod 5 = 0
2. {11: "Venkat"}
Key mod M = 11 mod 5 = 1
3. {12: "Jeevani"}
Key mod M = 12 mod 5 = 2

Observe that the Hash values were consecutive. This is the disadvantage of this type of Hash Function. We get consecutive indexes for consecutive keys, leading to poor performance due to decreased security. Sometimes, we need to analyze many consequences while choosing the Hash Table size.

A simple program to demonstrate the mechanism of the division method:

1. #include<stdio.h>
2. **int** main()
3. {
4. **int** size, i, indexes[3];
5. **int** keys[3] = {10, 11, 12};
6. printf("Enter the size of the Hash Table: ");
7. scanf("%d", &size);
8. **int** M = size
9. **for**(i = 0; i < 3; i ++)
10. {
11. indexes[i] = (keys[i] % M);
12. }
13. printf("\nThe indexes of the values in the Hash Table: ");
14. **for**(i = 0; i < 3; i++)
15. {
16. printf("%d ", indexes[i]);
17. }
18. **return** 0;
19. }

**Output:**

Enter the size of the Hash Table: 5

The indexes of the values in the Hash Table: 0 1 2

2. Mid Square Method:

It is a two-step process of computing the Hash value. Given a {key: value} pair, the Hash Function would be calculated by:

1. Square the **key -> key \* key**
2. Choose some digits from the middle of the number to obtain the Hash value.

We should choose the number of digits to extract based on the size of the Hash Table. Suppose the Hash Table size is 100; indexes will range from 0 to 99. Hence, we should select 2 digits from the middle.

Suppose the size of the Hash Table is 10 and the key: value pairs are:

{10: "Sudha, 11: "Venkat", 12: "Jeevani"}

Number of digits to be selected: Indexes: (0 - 9), so 1

H(10) = 10 \* 10 = 100 = 0

H(11) = 11 \* 11 = 121 = 2

H(12) = 12 \* 12 = 144 = 4

* All the digits in the key are utilized to contribute to the index, thus increasing the performance of the Data Structure.
* If the key is a large value, squaring it further increases the value, which is considered the con.
* Collisions might occur, too, but we can try to reduce or handle them.
* Another important point here is that, with the huge numbers, we need to take care of overflow conditions. For suppose, if we take a 6-digit key, we get a 12-digit number that exceeds the range of defined integers when we square it. We can use the long int or string multiplication technique.

A simple program to demonstrate the mechanism of the mid-square method:

3. Folding Method

Given a {key: value} pair and the table size is 100 (0 - 99 indexes), the key is broken down into 2 segments each except the last segment. The last segment can have less number of digits. Now, the Hash Function would be:

1. H(x) = (sum of equal-sized segments) mod (size of the Hash Table)
* The last carry with fewer digits can be ignored in calculating the Hash value.

For suppose "k" is a 10-digit key and the size of the table is 100(0 - 99), k is divided into:

sum = (k1k2) + (k3k4) + (k5k6) + (k7k8) + (k9k10)

Now, H(x) = sum % 100

**Let us now take an example:**

The {key: value} pairs: {1234: "Sudha", 5678: "Venkat"}

Size of the table: 100 (0 - 99)

**For {1234: "Sudha"}:**

1234 = 12 + 34 = 46

46 % 100 = **46**

**For {5678: "Venkat"}:**

5678 = 56 + 78 = 134

134 % 99 = **35**



4. Multiplication method

Unlike the three methods above, this method has more steps involved:

1. We must choose a constant between 0 and 1, say, A.
2. Multiply the key with the chosen A.
3. Now, take the fractional part from the product and multiply it by the table size.
4. The Hash will be the floor (only the integer part) of the above result.

So, the Hash Function under this method will be:

1. H(x) = floor(size(key\*A mod 1))

**For example:**

{Key: value} pairs: {1234: "Sudha", 5678: "Venkat"}

Size of the table: 100

A = 0.56

**For {1234: "Sudha"}:**

H(1234) = floor(size(1234\*0.56 mod 1))

= floor(100 \* 0.04)

= floor(4) = **4**

**For {5678: "Venkat"}:**

H(5678) = floor(size(5678\*0.56 mod 1))

= floor(99 \* 0.68)

= floor(67.32)

= **67**



* It is considered best practice to use the multiplication method when the Hash Table size is a power of 2 as it makes the access and all the operations faster.

**What after computing the Hash value?**

After computing the Hash value using the hash Function, this value is used as an index in the Hash table. Whenever the user wants to access a value, the corresponding key is hashed using the Hash Function, which gives the index of the key's value in the Hash Table with less cost than regular arrays and linked lists. Hence, Hashing is used to reduce the Time as well as space complexity of the program.

# Hash Functions and list/types of Hash functions

Hashing is the process of generating a value from a text or a list of numbers using a mathematical function known as a [hash function](https://www.geeksforgeeks.org/what-are-hash-functions-and-how-to-choose-a-good-hash-function/).

A **Hash Function**is a function that converts a given numeric or alphanumeric key to a small practical integer value. The mapped integer value is used as an index in the hash table. In simple terms, a hash function **maps** a significant number or string to a small integer that can be used as the **index** in the hash table.

The pair is of the form **(key, value)**, where for a given key, one can find a value using some kind of a “function” that maps keys to values. The key for a given object can be calculated using a function called a hash function. For example, given an array A, if i is the key, then we can find the value by simply looking up A[i].

**Types of Hash functions**

There are many hash functions that use numeric or alphanumeric keys. This article focuses on discussing different hash functions:

1. **Division Method.**
2. **Mid Square Method.**
3. **Folding Method.**
4. **Multiplication Method.**

Let’s begin discussing these methods in detail.

**1. Division Method:**

This is the most simple and easiest method to generate a hash value. The hash function divides the value k by M and then uses the remainder obtained.

**Formula:**

***h(K) = k mod M***

*Here,****k****is the key value, and****M****is the size of the hash table.*

It is best suited that **M**is a prime number as that can make sure the keys are more uniformly distributed. The hash function is dependent upon the remainder of a division.

**Example:**

*k = 12345
M = 95
h(12345) = 12345 mod 95
               = 90*

*k = 1276
M = 11
h(1276) = 1276 mod 11
             = 0*

**Pros:**

1. This method is quite good for any value of M.
2. The division method is very fast since it requires only a single division operation.

**Cons:**

1. This method leads to poor performance since consecutive keys map to consecutive hash values in the hash table.
2. Sometimes extra care should be taken to choose the value of M.

**2. Mid Square Method:**

The mid-square method is a very good hashing method. It involves two steps to compute the hash value-

1. Square the value of the key k i.e. k2
2. Extract the middle **r** digits as the hash value.

**Formula:**

***h(K) = h(k x k)***

*Here,****k****is the key value.*

The value of **r**can be decided based on the size of the table.

**Example:**

Suppose the hash table has 100 memory locations. So r = 2 because two digits are required to map the key to the memory location.

*k = 60
k x k = 60 x 60
        = 3600
h(60) = 60*

*The hash value obtained is 60*

**Pros:**

1. The performance of this method is good as most or all digits of the key value contribute to the result. This is because all digits in the key contribute to generating the middle digits of the squared result.
2. The result is not dominated by the distribution of the top digit or bottom digit of the original key value.

**Cons:**

1. The size of the key is one of the limitations of this method, as the key is of big size then its square will double the number of digits.
2. Another disadvantage is that there will be collisions but we can try to reduce collisions.

**3. Digit Folding Method:**

This method involves two steps:

1. Divide the key-value **k**into a number of parts i.e. **k1, k2, k3,….,kn**, where each part has the same number of digits except for the last part that can have lesser digits than the other parts.
2. Add the individual parts. The hash value is obtained by ignoring the last carry if any.

**Formula:**

***k = k1, k2, k3, k4, ….., kn******s = k1+ k2 + k3 + k4 +….+ kn******h(K)= s***

*Here,****s****is obtained by adding the parts of the key****k***

**Example:**

*k = 12345
k1 = 12, k2 = 34, k3 = 5
s = k1 + k2 + k3
  = 12 + 34 + 5
  = 51
h(K) = 51*

**Note:**
The number of digits in each part varies depending upon the size of the hash table. Suppose for example the size of the hash table is 100, then each part must have two digits except for the last part which can have a lesser number of digits.

**4. Multiplication Method**

This method involves the following steps:

1. Choose a constant value A such that 0 < A < 1.
2. Multiply the key value with A.
3. Extract the fractional part of kA.
4. Multiply the result of the above step by the size of the hash table i.e. M.
5. The resulting hash value is obtained by taking the floor of the result obtained in step 4.

**Formula:**

***h(K) = floor (M (kA mod 1))***

*Here,****M****is the size of the hash table.****k****is the key value.****A****is a constant value.*

**Example:**

*k = 12345
A = 0.357840
M = 100*

*h(12345) = floor[ 100 (12345\*0.357840 mod 1)]
               = floor[ 100 (4417.5348 mod 1) ]
               = floor[ 100 (0.5348) ]
               = floor[ 53.48 ]
               = 53*

**Pros:**

The advantage of the multiplication method is that it can work with any value between 0 and 1, although there are some values that tend to give better results than the rest.

**Cons:**

The multiplication method is generally suitable when the table size is the power of two, then the whole process of computing the index by the key using multiplication hashing is very fast.

[Next](https://www.geeksforgeeks.org/what-are-hash-functions-and-how-to-choose-a-good-hash-function/)

[**What are Hash Functions and How to choose a good Hash Function?**](https://www.geeksforgeeks.org/what-are-hash-functions-and-how-to-choose-a-good-hash-function/)

**What is a Hash Function?**

A function that converts a given big phone number to a small practical integer value. The mapped integer value is used as an index in the hash table. In simple terms, a hash function maps a big number or string to a small integer that can be used as the index in the hash table.

**What is meant by Good Hash Function?**

A good hash function should have the following properties:

1. Efficiently computable.
2. Should uniformly distribute the keys (Each table position equally likely for each key)

**For example:** For phone numbers, a bad hash function is to take the first three digits. A better function is considered the last three digits. Please note that this may not be the best hash function. There may be better ways.

In practice, we can often employ ***heuristic techniques*** to create a hash function that performs well. Qualitative information about the distribution of the keys may be useful in this design process. In general, a hash function should depend on every single bit of the key, so that two keys that differ in only one bit or one group of bits (regardless of whether the group is at the beginning, end, or middle of the key or present throughout the key) hash into different values. Thus, a hash function that simply extracts a portion of a key is not suitable. Similarly, if two keys are simply digited or character permutations of each other *(such as 139 and 319)*, they should also hash into different values.

**Const string & key**

it is called a read only reference. i.e you cannot change it.

eg:

1. string a = "str1";
2. const string& b = a; //you can never initialize a reference without a r val.
3. cout << b ; // prints str1.
4. b = "str2"; //throws an error.
5. a = "changed str"; //you can change this at will.
6. cout << b ; // prints "changed str".

when you want to pass an object there are two ways of doing it, pass by reference, i.e you pass the object itself and pass by value, i.e you make a copy of the object and then pass it to a function.

difference being, any change that is done on a copy doesn’t reflect on the original, but any change on the original remains.

if we are not changing the state of the object that is being passed it is easier, faster and cheaper to pass itself rather than its copy.

but the called function might accidentally change the passed object by reference, so we make it const. i.e we are telling the compiler that we do not expect any one to make any changes in the object, if some one tries then make noise.

eg:

1. void passCopy(string a)
2. {
3. a = "changed";
4. }
5.
6. void passRef(string& a)
7. {
8. a = "changed";
9. }
10.
11. void passConstRef(const string& a)
12. {
13. a = "changed 2"; // compilation error, since it is a read only reference.
14. }
15.
16. int main()
17. {
18. string a = "str";
19. cout << a; //prints "str"
20.
21. passCopy(a);
22. cout << a; // still prints "str", as we have made changes on copy not
23. //on original code.
24.
25. passRef(a);
26. cout << a; // prints "changed" , as we passed the original (reference).
27.
28. passConstRef(a); //throws error.
29. //because we are trying to edit the read only reference.
30.
31. return 0;

f(const string&) takes string by const reference: f is operating directly on the string object passed by reference: there is no copy involved. const prevents modifications to the original object though.

f(const string) takes a string value, which means f is given a copy of the original string. Even if you drop const, when passing by value, any modification to the string is lost when f returns.

1. string g\_value;
2. void callback()
3. { g\_value = "blue";
4. }
5. void ProcessStringByRef(const string &s)
6. { callback();
7. std::cout << s << "\n";
8. }
9. void ProcessStringByValue(const string s)
10. { callback();
11. std::cout << s << "\n";
12. }
13. int main()
14. {
15. g\_value = "red";
16. ProcessStringByValue(g\_value);
17. g\_value = "red";
18. ProcessStringByRef(g\_value);
19. }

Output:

red

blue

iven a string S of length N consisting of only lower-case English alphabets, you will be asked to process Q queries over it . In each query you will be given two lower case characters X and Y. Your task is to find out the number of such substrings of the the string S which have the characters X and Y on either of its end points, both X...Y and Y...X are considered to be valid.

Note : Substrings length should be greater than 1.

**Input Format**

The first line of the input will contain N , the length of the string.

Next line will contain as string of length N. Next line will will contain Q , the number of queries.

Then Q subsequent lines will contain two lowercase characters X and Y separated by a space.

**Constraints**

1<= N <= 10^6

1<= Q <= 10^3

**Output Format**

For each query , output the answer in a separate line.

**Sample Input 0**

5

aacbb

2

a c

a b

**Sample Output 0**

2

4

**Explanation 0**

For the first query, the possible substrings are aac and ac. Hence the answer is 2.

For the second query, the possible substrings are aacbb, aacb , acbb , and , acb hence making a total of 4 substrings.

<https://leetcode.com/tag/hash-table/>