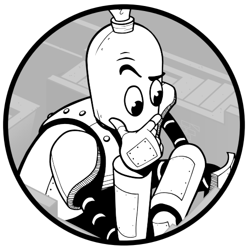
1
HASH TABLES



It's amazing how often computer programs need to search for information, whether it's to find a user's profile in a database or to retrieve a customer's orders. No one likes waiting for a slow search to complete.

In this chapter, we'll solve two problems whose solutions hinge on being able to perform efficient searches. The first problem is determining whether or not all snowflakes in a collection are identical. The second is determining how many passwords can be used to log in to someone's account. We want to solve these problems correctly, but we'll see that some correct approaches are simply too slow. We'll be able to achieve enormous performance increases using a data structure known as a hash table, which we'll explore at length.

We'll end the chapter by looking at a third problem: determining how many ways a letter can be deleted from one word to arrive at another. Here we'll see the risks of uncritically using a new data structure—when learning something new, it's tempting to try to use it everywhere!

Problem 1: Unique Snowflakes

This is DMOJ problem `cco07p2`.

The Problem

We're given a collection of snowflakes, and we have to determine whether any of the snowflakes in the collection are identical.

A snowflake is represented by six integers, where each integer gives the length of one of the snowflake's arms. For example, this is a snowflake:

```
3, 9, 15, 2, 1, 10
```

Snowflakes can also have repeated integers, such as

```
8, 4, 8, 9, 2, 8
```

What does it mean for two snowflakes to be identical? Let's work up to that definition through a few examples.

First, we'll look at these two snowflakes:

```
1, 2, 3, 4, 5, 6
```

and

```
1, 2, 3, 4, 5, 6
```

These are clearly identical because the integers in one snowflake match the integers in their corresponding positions in the other snowflake.

Here's our second example:

```
1, 2, 3, 4, 5, 6
```

and

```
4, 5, 6, 1, 2, 3
```

These are also identical. We can see this by starting at the `1` in the second snowflake and moving right. We see the integers `1`, `2`, and `3` and then, wrapping around to the left, we see `4`, `5`, and `6`. These two pieces together give us the first snowflake.

We can think of each snowflake as a circle as in Figure 1-1.

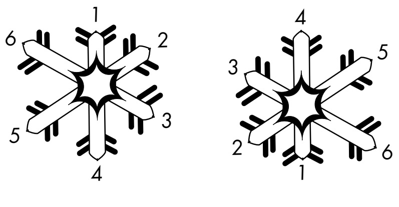


Figure 1-1: Two identical snowflakes

The two snowflakes are identical because we can start at the `1` in the second snowflake and follow it clockwise to get the first snowflake.

Let's try a different kind of example:

```
1, 2, 3, 4, 5, 6
```

and

```
3, 2, 1, 6, 5, 4
```

From what we've seen so far, we would deduce that these are not identical. If we start with the `1` in the second snowflake and move right (wrapping around to the left when we hit the right end), we get `1, 6, 5, 4, 3, 2`. That's not even close to the `1, 2, 3, 4, 5, 6` in the first snowflake.

However, if we begin at the `1` in the second snowflake and move left instead of right, then we do get exactly `1, 2, 3, 4, 5, 6`! Moving left from the `1` gives us `1, 2, 3`, and wrapping around to the right, we can proceed leftward to collect `4, 5, 6`. In Figure 1-2, this corresponds to starting at the `1` in the second snowflake and moving counterclockwise.

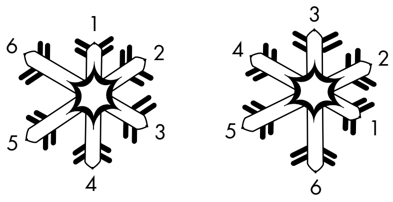


Figure 1-2: Two other identical snowflakes

That's our third way for two snowflakes to be identical: two snowflakes are identical if they match when we move counterclockwise through the numbers.

Putting it all together, we can conclude that two snowflakes are identical if they are the same, if we can make them the same by moving rightward through one of the snowflakes (moving clockwise), or if we can make them the same by moving leftward through one of the snowflakes (moving counterclockwise).

Input

The first line of input is an integer n, the number of snowflakes that we'll be processing. The value n will be between 1 and 100,000. Each of the following n lines represents one snowflake: each line has six integers, where each integer is at least 0 and at most 10,000,000.

Output

Our output will be a single line of text:

- If there are no identical snowflakes, output exactly `No two snowflakes are alike.`
- If there are at least two identical snowflakes, output exactly `Twin snowflakes found.`

The time limit for solving the test cases is one second.

Simplifying the Problem

One general strategy for solving competitive programming challenges is to first work with a simpler version of the problem. Let's warm up by eliminating some of the complexity from this problem.

Suppose that instead of working with snowflakes made of multiple integers, we're working with single integers. We have a collection of integers, and we want to know whether any are identical. We can test whether two integers are identical with C's `==` operator. We can test all pairs of integers, and if we find even one pair of identical integers, we'll stop and output

```
Twin integers found.
```

If no identical integers are found, we'll output

```
No two integers are alike.
```

Let's make an `identify_identical` function with two nested loops to compare pairs of integers, as shown in Listing 1-1.

```
void identify_identical(int values[], int n) {
  int i, j;
  for (i = 0; i < n; i++) {
 ❶ for (j = i + 1; j < n; j++) {
      if (values[i] == values[j]) {
        printf("Twin integers found.\n");
        return;
      }
    }
  }
  printf("No two integers are alike.\n");
}
```

Listing 1-1: Finding identical integers

We feed the integers to the function through the `values` array. We also pass in `n`, the number of integers in the array.

Notice that we start the inner loop at `i + 1` and not `0` ❶. If we started at `0`, then eventually `j` would equal `i`, and we'd compare an element to itself, giving us a false positive result.

Let's test `identify_identical` using this small `main` function:

```
int main(void) {
  int a[5] = {1, 2, 3, 1, 5};
  identify_identical(a, 5);
  return 0;
}
```

Run the code and you will see from the output that our function correctly identified a matching pair of `1`s. In general, I won't provide much test code in this book, but it's important that you play with and test the code yourself as we go along.

Solving the Core Problem

Let's take our `identify_identical` function and try to modify it to solve the Snowflake problem. To do so, we need to make two extensions to our code:

1. We have to work with six integers at a time, not one. A two-dimensional array should work nicely here: each row will be a snowflake with six columns (one column per element).
2. As we saw earlier, there are multiple ways for two snowflakes to be identical. Unfortunately, this means we can't just use `==` to compare snowflakes. We need to take into account our "moving right" and "moving left" criteria (not to mention that `==` in C doesn't compare contents of arrays anyway!). Correctly comparing snowflakes will be the major update to our algorithm.

To begin, let's write a pair of helper functions: one for checking "moving right" and one for checking "moving left." Each of these helpers takes three parameters: the first snowflake, the second snowflake, and the starting point for the second snowflake.

Checking to the Right

Here is the function signature for `identical_right`:

```
int identical_right(int snow1[], int snow2[], int start)
```

To determine whether the snowflakes are the same by "moving right," we scan `snow1` from index `0` and `snow2` from index `start`. If we find corresponding elements that are not equal, then we return `0` to signify that we haven't found identical snowflakes. If all the corresponding elements do match, then we return `1`. Think of `0` as representing false and `1` as representing true.

In Listing 1-2 we make a first attempt at writing this function's code.

```
// bugged!
int identical_right(int snow1[], int snow2[], int start) {
  int offset;
  for (offset = 0; offset < 6; offset++) {
➊ if (snow1[offset] != snow2[start + offset])
      return 0;
  }
  return 1;
}
```

Listing 1-2: Identifying identical snowflakes moving right (bugged!)

As you may notice, this code won't work as we hope. The problem is `start + offset` ➊. If we have `start = 4` and `offset = 3`, then `start + offset = 7`.

The trouble is snow2[7], as snow2[5] is the farthest index to which we are allowed to go.

This code doesn't take into account that we must wrap around to the left of snow2. If our code is about to use an erroneous index of 6 or greater, we should reset our index by subtracting six. This will let us continue with index 0 instead of index 6, index 1 instead of index 7, and so on. Let's try again with Listing 1-3.

```
int identical_right(int snow1[], int snow2[], int start) {
  int offset, snow2_index;
  for (offset = 0; offset < 6; offset++) {
    snow2_index = start + offset;
    if (snow2_index >= 6)
      snow2_index = snow2_index - 6;
        if (snow1[offset] != snow2[snow2_index])
      return 0;
  }
  return 1;
}
```

Listing 1-3: Identifying identical snowflakes moving right

This works, but we can still improve it. One change that many programmers would consider making at this point involves using %, the mod operator. The % operator computes remainders, so x % y returns the remainder of integer-dividing x by y. For example, 9 % 3 is 0, because there is no remainder when dividing 9 by 3. 10 % 4 is 2, because 2 is left over when dividing 10 by 4.

We can use mod here to help with the wraparound behavior. Notice that 0 % 6 is 0, 1 % 6 is 1, . . . , 5 % 6 is 5. Each of these numbers is smaller than 6, and so will itself be the remainder when dividing 6. The numbers 0 to 5 correspond to the legal indices of snow2, so it's good that % leaves them alone. For our problematic index 6, 6 % 6 is 0: 6 divides 6 evenly, with no remainder at all, wrapping us around to the start. That's precisely the wraparound behavior we wanted.

Let's update identical_right to use the % operator. Listing 1-4 shows the new function.

```
int identical_right(int snow1[], int snow2[], int start) {
  int offset;
  for (offset = 0; offset < 6; offset++) {
    if (snow1[offset] != snow2[(start + offset) % 6])
      return 0;
  }
  return 1;
}
```

Listing 1-4: Identifying identical snowflakes moving right using mod

Whether you use this "mod trick" is up to you. It saves a line of code and is a common pattern that many programmers will be able to identify. However, it doesn't always easily apply, even in cases that exhibit similar wraparound behavior, such as `identical_left`. Let's turn to this now.

Checking to the Left

The function `identical_left` is very similar to `identical_right`, except that we need to move left and then wrap around to the right. When traversing right, we had to be wary of erroneously accessing index 6 or greater; this time, we have to be wary of accessing index −1 or less.

Unfortunately, our mod solution won't directly work here. In C, `-1 / 6` is 0, leaving a remainder of −1, and so `-1 % 6` is −1. We'd need `-1 % 6` to be 5.

Let's just do this without using mod. In Listing 1-5, we provide the code for the `identical_left` function.

```
int identical_left(int snow1[], int snow2[], int start) {
  int offset, snow2_index;
  for (offset = 0; offset < 6; offset++) {
    snow2_index = start - offset;
    if (snow2_index <= -1)
      snow2_index = snow2_index + 6;
    if (snow1[offset] != snow2[snow2_index])
      return 0;
  }
  return 1;
}
```

Listing 1-5: Identifying identical snowflakes moving left

Notice the similarity between this function and that of Listing 1-3. All we did was subtract the offset instead of adding it and change the bounds check at `6` to a bounds check at `-1`.

Putting It Together

With these two helper functions, `identical_right` and `identical_left`, we can finally write a function that tells us whether two snowflakes are identical. Listing 1-6 gives the code for an `are_identical` function that does this. We simply test moving right and moving left for each of the possible starting points in `snow2`.

```
int are_identical(int snow1[], int snow2[]) {
  int start;
  for (start = 0; start < 6; start++) {
 ❶ if (identical_right(snow1, snow2, start))
      return 1;
 ❷ if (identical_left(snow1, snow2, start))
      return 1;
  }
  return 0;
}
```

Listing 1-6: Identifying identical snowflakes

We test whether `snow1` and `snow2` are the same by moving right in `snow2` ❶. If they are identical according to that criterion, we return `1` (true). We then similarly check the moving-left criterion ❷.

It's worth pausing here to test the `are_identical` function on a few sample snowflake pairs. Please do that before continuing!

Solution 1: Pairwise Comparisons

When we need to compare two snowflakes, we just deploy our `are_identical` function instead of `==`. Comparing two snowflakes is now as easy as comparing two integers.

Let's revise our earlier `identify_identical` function (Listing 1-1) to work with snowflakes using the new `are_identical` function (Listing 1-6). We'll make pairwise comparisons between snowflakes, printing out one of two messages depending on whether we find identical snowflakes. The code is given in Listing 1-7.

```
void identify_identical(int snowflakes[][6], int n) {
  int i, j;
  for (i = 0; i < n; i++) {
    for (j = i + 1; j < n; j++) {
      if (are_identical(snowflakes[i], snowflakes[j])) {
        printf("Twin snowflakes found.\n");
        return;
      }
    }
  }
  printf("No two snowflakes are alike.\n");
}
```

Listing 1-7: Finding identical snowflakes

This `identify_identical` function on snowflakes is almost, symbol for symbol, the same as the `identify_identical` function on integers in Listing 1-1. All we've done is swap `==` for a function that compares snowflakes.

Reading the Input

We're not quite ready to submit to our judge. We haven't yet written the code to read the snowflakes from standard input. Revisit the problem description at the start of the chapter. We need to read a line containing integer *n* that tells us how many snowflakes there are and then read each of the following *n* lines as an individual snowflake.

Listing 1-8 is a `main` function that processes the input and then calls `identify_identical` from Listing 1-7.

```
#define SIZE 100000

int main(void) {
❶ static int snowflakes[SIZE][6];
  int n, i, j;
  scanf("%d", &n);
  for (i = 0; i < n; i++)
    for (j = 0; j < 6; j++)
      scanf("%d", &snowflakes[i][j]);
  identify_identical(snowflakes, n);
  return 0;
}
```

Listing 1-8: The `main` function for Solution 1

Notice that the `snowflakes` array is a `static` array ❶. This is because the array is huge; without using such a `static` array, the amount of space needed would likely outstrip the amount of memory available to the function. We use `static` to place the array in its own, separate piece of memory, where space is not a concern. Be careful with `static`, though. Regular local variables are initialized on each call of a function, but `static` ones retain whatever value they had on the previous function call (see "Static Keyword" on page xxvi).

Also notice that we've allocated an array of 100,000 snowflakes ❶. You might be concerned that this is a waste of memory. What if the input has only a few snowflakes? For competitive programming problems, it's generally okay to hardcode the memory requirements for the largest problem instance: the test cases are likely to stress test your submission on the maximum size anyway!

The rest of the function is straightforward. We read the number of snowflakes using `scanf`, and we use that number to determine the number of iterations of the outer `for` loop. For each such iteration, we loop six times in the inner `for` loop, each time reading one integer. We then call `identify_identical` to produce the appropriate output.

Putting this `main` function together with the other functions we have written gives us a complete program that we can submit to the judge. Try it out . . . and you should get a "Time-Limit Exceeded" error. It looks like we have more work to do!

Diagnosing the Problem

Our first solution was too slow, so we got a "Time-Limit Exceeded" error. Let's understand why.

For our discussion here, we'll assume that there are no identical snowflakes. This is the worst-case scenario for our code, since then it doesn't stop processing early.

The reason that our first solution is slow is because of the two nested `for` loops in Listing 1-7. Those loops compare each snowflake to every other snowflake, resulting in a huge number of comparisons when the number of snowflakes n is large.

Let's figure out the number of snowflake comparisons our program makes. Since we might compare each pair of snowflakes, we can restate this question as asking for the total number of snowflake pairs. For example, if we have four snowflakes numbered 1, 2, 3, and 4, then our scheme performs six snowflake comparisons: Snowflakes 1 and 2, 1 and 3, 1 and 4, 2 and 3, 2 and 4, and 3 and 4. Each pair is formed by choosing one of the n snowflakes as the first snowflake and then choosing one of the remaining $n - 1$ snowflakes as the second snowflake.

For each of n decisions for the first snowflake, we have $n - 1$ decisions for the second snowflake. This gives a total of $n(n - 1)$ decisions. However, $n(n - 1)$ double-counts the true number of snowflake comparisons that we make—it includes both of the comparisons 1 and 2 and 2 and 1, for example. Our solution compares these only once, so we can divide by 2, giving $n(n - 1)/2$ snowflake comparisons for n snowflakes.

This might not seem so bad, but let's substitute some values of n into $n(n - 1)/2$ and see what happens. Substituting 10 gives $10(9)/2 = 45$. Performing 45 comparisons is a piece of cake for any computer and can be done in milliseconds. How about $n = 100$? That gives 4,950: still no problem. It looks like we're okay for a small n, but the problem statement says that we can have up to 100,000 snowflakes. Go ahead and substitute 100,000 for n in $n(n - 1)/2$: this gives 4,999,950,000 snowflake comparisons. If you run a test case with 100,000 snowflakes on a typical laptop, it will take something like three minutes. That's far too slow—we need at most one second, not several minutes! As a conservative rule of thumb for today's computers, think of the number of steps that we can perform per second as about 30 million. Trying to make nearly 5 billion snowflake comparisons in one second is not doable.

If we expand $n(n-1)/2$, we get $n^2/2 - n/2$. The largest exponent there is 2. Algorithm developers therefore call this an $O(n^2)$ algorithm, or a *quadratic-time algorithm*. $O(n^2)$ is pronounced "big O of n squared," and you can think of it as telling you that the rate at which the amount of work grows is quadratic relative to the problem size. For a brief introduction to big O, see Appendix A.

We need to make such a large number of comparisons because identical snowflakes could show up anywhere in the array. If there were a way to get identical snowflakes close together in the array, we could quickly determine whether a particular snowflake was part of an identical pair. Maybe we can try sorting the array to get the identical snowflakes close together?

Sorting Snowflakes

C has a library function called `qsort` that we can use to sort an array. The key requirement is a comparison function: it takes pointers to two elements to sort, and it returns a negative integer if the first element is less than the second, `0` if they are equal, and a positive integer if the first is greater than the second. We can use `are_identical` to determine whether two snowflakes are equal; if they are, we return `0`.

What does it mean, though, for one snowflake to be less than or greater than another? It's tempting to just agree on some arbitrary rule here. We might say, for example, that the snowflake that is "less" is the one whose first differing element is smaller than the corresponding element in the other snowflake. We do that in Listing 1-9.

```
int compare(const void *first, const void *second) {
  int i;
  const int *snowflake1 = first;
  const int *snowflake2 = second;
  if (are_identical(snowflake1, snowflake2))
    return 0;
  for (i = 0; i < 6; i++)
    if (snowflake1[i] < snowflake2[i])
      return -1;
  return 1;
}
```

Listing 1-9: A comparison function for sorting

Unfortunately, sorting in this way will not help us solve our problem. You might try writing a program that uses sorting to put identical snowflakes next to each other so that you can find them quickly. But here's a four-snowflake test case that would likely fail on your laptop:

```
4
3 4 5 6 1 2
2 3 4 5 6 7
4 5 6 7 8 9
1 2 3 4 5 6
```

The first and fourth snowflakes are identical—but the message `No two snowflakes are alike.` may be output. What's going wrong?

Here are two facts that `qsort` might learn as it executes:

1. Snowflake 4 is less than Snowflake 2.
2. Snowflake 2 is less than Snowflake 1.

From this, `qsort` could conclude that Snowflake 4 is less than Snowflake 1, without ever directly comparing Snowflake 4 and Snowflake 1! Here it's relying on the transitive property of less than. If *a* is less than *b*, and *b* is less than *c*, then surely *a* should be less than *c*. It seems like our definitions of "less" and "greater" matter after all.

Unfortunately, it isn't clear how one would define "less" and "greater" on snowflakes so as to satisfy transitivity. If you're disappointed, perhaps you can take solace in the fact that we'll be able to develop a faster solution without using sorting at all.

In general, collecting similar values with sorting can be a useful data-processing technique. As a bonus, good sorting algorithms run quickly—certainly faster than $O(n^2)$, but we aren't going to be able to use sorting here.

Solution 2: Doing Less Work

Comparing all pairs of snowflakes and trying to sort the snowflakes proved to be too much work. To work up to our next, and ultimate, solution, let's pursue the idea of trying to avoid comparing snowflakes that are obviously not identical. For example, if we have snowflakes

```
1, 2, 3, 4, 5, 6
```

and

```
82, 100, 3, 1, 2, 999
```

there's no way that these snowflakes can be identical. We shouldn't even waste our time comparing them.

The numbers in the second snowflake are very different from the numbers in the first snowflake. To devise a way to detect that two snowflakes are different without having to directly compare them, we might begin by comparing the snowflake's first elements, because 1 is very different from 82. But now consider these two snowflakes:

```
3, 1, 2, 999, 82, 100
```

and

```
82, 100, 3, 1, 2, 999
```

These two snowflakes *are* identical even though 3 is very different from 82. We need to do more than just look at first elements.

A quick litmus test for determining whether two snowflakes might be identical is to use the *sum* of their elements. When we sum our two example snowflakes, for `1, 2, 3, 4, 5, 6`, we get a total of 21, and for `82, 100, 3, 1, 2, 999`, we get 1,187. We say that the *code* for the former snowflake is 21 and the code for the latter is 1,187.

Our hope is that we can throw the "21 snowflakes" in one bin and throw the "1,187 snowflakes" in another, and then we never have to compare the 21s to the 1,187s. We can do this binning for each snowflake: add up its elements, get a code of *x*, and then store it along with all of the other snowflakes with code *x*.

Of course, finding two snowflakes with a code of 21 does not guarantee they are identical. For example, both `1, 2, 3, 4, 5, 6` and `16, 1, 1, 1, 1, 1` have a code of 21, and they are surely not identical.

That's okay, because our "sum" rule is designed to weed out snowflakes that are clearly not identical. This allows us to avoid comparing all pairs—the source of the inefficiency in Solution 1—and only compare pairs that have not been filtered out as obviously nonidentical.

In Solution 1, we stored each snowflake consecutively in the array: the first snowflake at index `0`, the second at index `1`, and so on. Here, our storage strategy is different: sum codes determine snowflakes' locations in the array! That is, for each snowflake, we calculate its code and use that code as the index for where to store the snowflake.

We have to solve two problems:

1. Given a snowflake, how do we calculate its code?
2. What do we do when multiple snowflakes have the same code?

Let's deal with calculating the code first.

Calculating Sum Codes

At first glance, calculating the code seems easy. We could just sum all of the numbers within each snowflake like so:

```
int code(int snowflake[]) {
  return (snowflake[0] + snowflake[1] + snowflake[2]
          + snowflake[3] + snowflake[4] + snowflake[5]);
}
```

This works fine for many snowflakes, such as `1, 2, 3, 4, 5, 6`, and `82, 100, 3, 1, 2, 999`, but consider a snowflake with huge numbers, such as

```
1000000, 2000000, 3000000, 4000000, 5000000, 6000000
```

The code that we calculate is `21000000`. We plan to use that code as the *index* in an array that holds the snowflakes, so to accommodate this, we'd have to declare an array with room for 21 million elements. As we're using at most 100,000 elements (one for each snowflake), this is an outrageous waste of memory.

We're going to stick with an array that has room for 100,000 elements. We'll need to calculate a snowflake's code as before, but then we must force that code to be a number between `0` and `99999` (the minimum and maximum index in our array). One way to do this is to break out the `%` (mod) operator again. Taking a nonnegative integer mod x yields an integer between 0 and $x - 1$. No matter the sum of a snowflake, if we take it mod 100,000, we'll get a valid index in our array.

This method has one downside: taking the mod like this will force *more* nonidentical snowflakes to end up with the same code. For example, the sums for `1, 1, 1, 1, 1, 1` and `100001, 1, 1, 1, 1, 1` are different—`6` and `100006`—but once we take them mod 100,000, we get `6` in both cases. This is an acceptable risk to take: we'll just hope that this doesn't happen much; when it does, we'll perform the necessary pairwise comparisons.

We'll calculate the sum code for a snowflake and mod it, as displayed in Listing 1-10.

```
#define SIZE 100000

int code(int snowflake[]) {
  return (snowflake[0] + snowflake[1] + snowflake[2]
          + snowflake[3] + snowflake[4] + snowflake[5]) % SIZE;
}
```

Listing 1-10: Calculating the snowflake code

Snowflake Collisions

In Solution 1, we used the following fragment to store a snowflake at index `i` in the `snowflakes` array:

```
    for (j = 0; j < 6; j++)
      scanf("%d", &snowflakes[i][j]);
```

This worked because exactly one snowflake was stored in each row of the two-dimensional array.

However, now we have to contend with the `1, 1, 1, 1, 1, 1` and `100001, 1, 1, 1, 1, 1` kind of collision, where, because they'll end up with the same mod code and that code serves as the snowflakes index in the array, we need to store multiple snowflakes in the same array element. That is, each array element will no longer be one snowflake but a collection of zero or more snowflakes.

One way to store multiple elements at the same array index is to use a *linked list*, a data structure that links each element to the next. Here, each element in the snowflakes array will point to the first snowflake in the linked list; the remainder of the snowflakes can be accessed through `next` pointers.

We'll use a typical linked list implementation. Each `snowflake_node` contains both a snowflake and a pointer to the next snowflake. To collect these two components, we'll use a struct. We'll also make use of `typedef`, which allows us to later use `snowflake_node` instead of the full `struct snowflake_node`:

```
typedef struct snowflake_node {
  int snowflake[6];
  struct snowflake_node *next;
} snowflake_node;
```

This change necessitates updates to two functions, `main` and `identify_identical`, because those functions use our old two-dimensional array.

The New main Function

You can see the updated `main` code in Listing 1-11.

```
int main(void) {
❶ static snowflake_node *snowflakes[SIZE] = {NULL};
❷ snowflake_node *snow;
  int n, i, j, snowflake_code;
  scanf("%d", &n);
  for (i = 0; i < n; i++) {
 ❸ snow = malloc(sizeof(snowflake_node));
    if (snow == NULL) {
      fprintf(stderr, "malloc error\n");
      exit(1);
    }
    for (j = 0; j < 6; j++)
    ❹ scanf("%d", &snow->snowflake[j]);
 ❺ snowflake_code = code(snow->snowflake);
 ❻ snow->next = snowflakes[snowflake_code];
 ❼ snowflakes[snowflake_code] = snow;
  }
  identify_identical(snowflakes);
  // deallocate all malloc'd memory, if you want to be good
  return 0;
}
```

Listing 1-11: The `main` *function for Solution 2*

Let's walk through this code. First, notice that we changed the type of our array from a two-dimensional array of numbers to a one-dimensional array of pointers to snowflake nodes ❶. We also declare `snow` ❷, which will point to snowflake nodes that we allocate.

We use `malloc` to allocate memory for each `snowflake_node` ❸. Once we have read in and stored the six numbers for a snowflake ❹, we use `snowflake_code` to hold the snowflake's code ❺, calculated using the function we wrote in Listing 1-10.

The last thing to do is to add the snowflake to the `snowflakes` array, which amounts to adding a node to a linked list. We do this by inserting the snowflake at the beginning of the linked list. We first point the inserted node's `next` pointer to the first node in the list ❻, and then we set the start of the list to point to the inserted node ❼. The order matters here: if we had reversed the order of these two lines, we would lose access to the elements already in the linked list!

Notice that, in terms of correctness, it doesn't matter where in the linked list we add the new node. It could go at the beginning, the end, or somewhere in the middle—it's our choice. So we should do whatever is fastest, and adding to the beginning is fastest because it doesn't require us to traverse the list at all. If we instead chose to add an element to the end of a linked list, we'd have to traverse the entire list. If that list had a million elements, we'd have to follow the next pointers a million times until we got to the end—that would be very slow!

Let's work on a quick example of how this `main` function works. Here's the test case:

```
4
1 2 3 4 5 6
8 3 9 10 15 4
```

```
16 1 1 1 1 1
100016 1 1 1 1 1
```

Each element of `snowflakes` begins as `NULL`, the empty linked list. As we add to `snowflakes`, elements will begin to point at snowflake nodes. The numbers in the first snowflake add up to 21, so it goes into index `21`. The second snowflake goes into index `49`. The third snowflake goes into index `21`. At this point, index `21` is a linked list of *two* snowflakes: `16, 1, 1, 1, 1, 1` followed by `1, 2, 3, 4, 5, 6`.

How about the fourth snowflake? That goes into index `21` again, and now we have a linked list of three snowflakes there. See Figure 1-3 for the hash table that we've built.

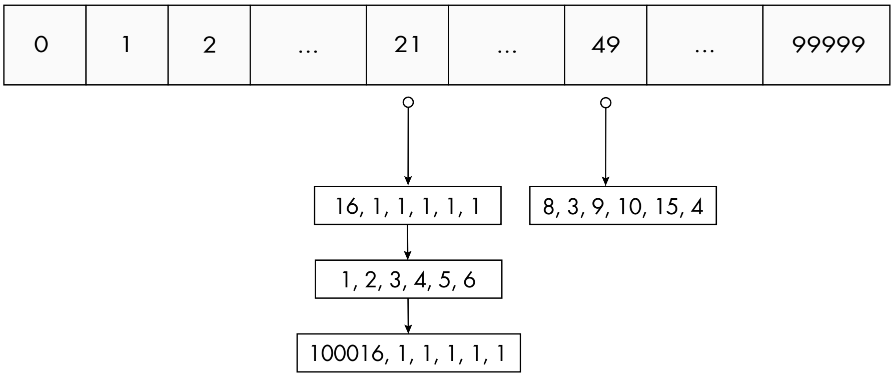


Figure 1-3: A hash table with four snowflakes

There are multiple snowflakes in index `21`. Does this mean that we have identical snowflakes? No! This emphasizes the fact that a linked list with multiple elements is not sufficient evidence to claim that we have identical snowflakes. We have to compare each pair of those elements to correctly state our conclusion. That's the final piece of the puzzle.

The New identify_identical Function

We need `identify_identical` to make all pairwise comparisons of snowflakes within each linked list. Listing 1-12 shows the code to do so.

```
void identify_identical(snowflake_node *snowflakes[]) {
  snowflake_node *node1, *node2;
  int i;
  for (i = 0; i < SIZE; i++) {
 ❶ node1 = snowflakes[i];
    while (node1 != NULL) {
   ❷ node2 = node1->next;
      while (node2 != NULL) {
        if (are_identical(node1->snowflake, node2->snowflake)) {
          printf("Twin snowflakes found.\n");
          return;
        }
        node2 = node2->next;
      }
   ❸ node1 = node1->next;
    }
  }
  printf("No two snowflakes are alike.\n");
}
```

Listing 1-12: Identifying identical snowflakes in linked lists

We begin with `node1` at the first node in a linked list ❶. We use `node2` to traverse from the node to the right of `node1` ❷ all the way to the end of the linked list. This compares the first snowflake in the linked list to all other snowflakes in that linked list. We then advance `node1` to the second node ❸, and we compare that second snowflake to each snowflake to its right. We repeat this until `node1` reaches the end of the linked list.

This code is dangerously similar to `identify_identical` from Solution 1 (Listing 1-7), which made all pairwise comparisons between any two snowflakes. By contrast, our new code only makes pairwise comparisons within a single linked list. But what if someone crafts a test case where all snowflakes end up in the same linked list? Wouldn't the performance then be as bad as in Solution 1? It would, yes, but absent such nefarious data, we're in great shape. Take a minute to submit Solution 2 to the judge and see for yourself. You should see that we've discovered a much more efficient solution! What we've done is use a data structure called a hash table. We'll learn more about hash tables next.

Hash Tables

A *hash table* consists of two things:

1. An array. Locations in the array are referred to as *buckets*.
2. A *hash function*, which takes an object and returns its code as an index into the array.

The code returned by the hash function is referred to as a *hashcode*; that code determines at which index an object is stored or *hashed*.

Look closely at what we did in Listings 1-10 and 1-11 and you'll see that we already have both of these things. That `code` function, which took a snowflake and produced its code (a number between 0 and 99,999), is a hash function; and that `snowflakes` array is the array of buckets, where each bucket contains a linked list.

Hash Table Design

Designing a hash table involves many design decisions. Let's talk about three of them here.

The first decision concerns size. In Unique Snowflakes, we used an array size of 100,000. We could have instead used a smaller or larger array. A smaller array saves memory. For example, on initialization, a 50,000-element array stores half as many NULL values as does a 100,000-element array. However, a smaller array leads to more objects ending up in the same bucket. When objects end up in the same bucket, we say that a *collision* has occurred. The problem with having many collisions is that they lead to long linked lists. Ideally, all of the linked lists would be short so that we wouldn't have to walk through and do work on many elements. A larger array avoids some of these collisions.

To summarize, we have a memory–time tradeoff here. Make the hash table too small and collisions run rampant. Make the hash table too big and memory waste becomes a concern. In general, try to choose an array size that's a reasonable percentage—such as 20 percent or 50 percent or 100 percent—of the maximum number of elements you expect to insert into the hash table.

In Unique Snowflakes, we used an array size of 100,000 to match the maximum number of snowflakes; had we been constrained to use less memory, smaller arrays would have worked just fine as well.

The second consideration is our hash function. In Unique Snowflakes, our hash function adds up a snowflake's numbers mod 100,000. Importantly, this hash function guarantees that, if two snowflakes are identical, they will end up in the same bucket. (They might also end up in the same bucket if they are not identical, of course.) This is the reason why we can search within linked lists, and not between them, for identical snowflakes.

When solving a problem with a hash table, the hash function that we use should take into account what it means for two objects to be identical. If two objects are identical, then the hash function must hash them to the same bucket. In the case in which two objects must be exactly equal to be considered "identical," we can scramble things so extensively that the mapping between object and bucket is far more intricate than what we did with the snowflakes. Check out the oaat (one-at-a-time) hash function in Listing 1-13 for an example.

```
#define hashsize(n) ((unsigned long)1 << (n))
#define hashmask(n) (hashsize(n) - 1)

unsigned long oaat(char *key, unsigned long len, unsigned long bits) {
  unsigned long hash, i;
  for (hash = 0, i = 0; i < len; i++) {
    hash += key[i];
    hash += (hash << 10);
    hash ^= (hash >> 6);
  }
  hash += (hash << 3);
  hash ^= (hash >> 11);
  hash += (hash << 15);
  return hash & hashmask(bits);
}
```

```
int main(void) { // sample call of oaat
  char word[] = "hello";
  // 2^17 is the smallest power of 2 that is at least 100000
❶ unsigned long code = oaat(word, strlen(word), 17);
  printf("%u\n", code);
  return 0;
}
```

Listing 1-13: An intricate hash function

To call `oaat` ❶ as we do in the `main` function, we pass three parameters:

`key`	The data that we want to hash (here, we're hashing the `word` string)
`len`	The length of those data (here, the length of the `word` string)
`bits`	The number of bits that we want in the resulting hashcode (here, 17)

The maximum value that a hashcode could have is one less than 2 to the power of `bits`. For example, if we choose 17, then $2^{17} - 1 = 131,071$ is the maximum that a hashcode could be.

How does `oaat` work? Inside the `for` loop, it starts by adding the current byte of the key. That part is similar to what we did when adding up the numbers in a snowflake (Listing 1-10). Those left shifts and exclusive ors are in there to put the key through a blender. Hash functions do this blending to implement an *avalanche effect*, which means that a small change in the key's bits makes a huge change to the key's hashcode. Unless you intentionally created pathological data for this hash function or inserted a huge number of keys, it would be unlikely that you'd get many collisions. This highlights an important point: with a single hash function, there is *always* a collection of data that will lead to collisions galore and subsequently horrible performance. A fancy hash function like `oaat` can't protect against that. Unless we're concerned about malicious input, though, we can often get away with using a reasonably good hash function and can assume that it will spread the data around.

Indeed, this is why using our hash table solution (Solution 2) for Unique Snowflakes was so successful. We used a good hash function that distributes many nonidentical snowflakes into different buckets. Since we're not securing our code from attack, we don't have to worry about some evil person studying our code and figuring out a way to cause millions of collisions.

For our third and final design decision, we have to think about what we want to use as our buckets. In Unique Snowflakes, we used a linked list as each bucket. Using linked lists like this is known as a *chaining* scheme.

In another approach, known as *open-addressing*, each bucket holds at most one element, and there are no linked lists. To deal with collisions, we search through buckets until we find one that is empty. For example, suppose that we try to insert an object into bucket number 50, but Bucket 50 is already occupied. We might then try Bucket 51, then 52, then 53, stopping when we find an empty bucket. Unfortunately, this simple sequence can lead to poor performance when a hash table has many elements stored in it, so more nuanced search schemes are often used in practice.

Chaining is generally easier to implement than open-addressing, which is why we used chaining for Unique Snowflakes. However, open-addressing does have some benefits, including saving memory by not using linked list nodes.

Why Use Hash Tables?

Using a hash table turbocharges our solution to Unique Snowflakes. On a typical laptop, a test case with 100,000 elements will take only a fraction of a second to run! No pairwise comparisons of all elements and no sorting is needed, just a little processing on a bunch of linked lists.

Recall that we used an array size of 100,000. The maximum number of snowflakes that can be presented to our program is also 100,000. If we're given 100,000 snowflakes and assume the perfect scenario of each one going into its own bucket, then we'd have only one snowflake per linked list. If we have a little bad luck, then maybe a few of those snowflakes will collide and end up in the same bucket. In the absence of pathological data, though, we expect that each linked list will have at most a few elements. As such, making all pairwise comparisons within a bucket will take only a small, constant number of steps. We expect hash tables to give us a *linear-time* solution, because we take a constant number of steps in each of the n buckets. So we take something like n steps, in comparison to the $n(n - 1)/2$ formula we had for Solution 1. In terms of big O, we'd say that we expect an $O(n)$ solution.

Whenever you're working on a problem and you find yourself repeatedly searching for some element, consider using a hash table. A hash table takes a slow array search and converts it into a fast lookup. For some problems, you may be able to sort an array rather than use a hash table. A technique called binary search (discussed in Chapter 7) could then be used to quickly search for elements in the sorted array. But often—such as in Unique Snowflakes and the problem we'll solve next—that won't work. Hash tables to the rescue!

Problem 2: Login Mayhem

Let's go through another problem and pay attention to where a naive solution would rely on a slow search. We'll then drop in a hash table to cause a dramatic speedup. We'll go a little more quickly than we did for Unique Snowflakes because now we know what to look for.

This is DMOJ problem `coci17c1p3hard`.

The Problem

To log in to your account on a social network website, you'd expect that only your password would work—no one should be able to use a different password to get into your account. For example, let's say that your password is `dish`. (That's a terribly weak password—don't actually use that anywhere!) To log in to your account, someone would need to enter exactly `dish` as the password. That's just how logins work.

But now imagine that you are wanting to join a (hopefully theoretical) social network website that has a major security concern: other passwords besides yours can be used to get into your account! Specifically, if someone tries a password that has your password as a substring, then they're in. If your password were `dish`, for example, then passwords like `brandish` and `radishes` would work to get into your account because the string `dish` is in them. You don't know what password to choose for your account—so at various points you will ask: "If I chose this password, how many current users' passwords would get in to my account?"

We need to support two types of operations:

Add Sign up a new user with the given password.

Query Given a proposed password *p*, return the number of current users' passwords that could be used to get into an account whose password is *p*.

Input

The input consists of the following lines:

- A line containing *q*, the number of operations to be performed. *q* is between 1 and 100,000.
- *q* lines, each giving one add or query operation to be performed.

Here are the operations that can be performed in those *q* lines:

- An add operation is specified as the number `1`, a space, and then the new user's password. It indicates that a new user has joined with the provided password. This operation doesn't result in any output.
- A query operation is specified as the number `2`, a space, and then a proposed password *p*. It indicates that we should output the number of current users' passwords that could be used to get into an account whose password is *p*.

All passwords provided in these operations are between 1 and 10 lowercase characters.

Output

Output the result of each query operation, one per line.

The time limit for solving the test case is three seconds.

Solution 1: Looking at All Passwords

Let's work through a test case to make sure that we know exactly what we're being asked to do.

```
❶ 6
❷ 2 dish
  1 brandish
  1 radishes
  1 aaa
❸ 2 dish
❹ 2 a
```

We can tell from the first line ❶ that there are 6 operations for us to perform. The first operation ❷ asks us how many of the existing users' passwords would get into an account whose password is `dish`. Well, there are no existing users, so the answer is `0`!

Next, we add three user passwords, and then we get to our next query operation ❸. Now we're being asked about `dish` in the context of these three passwords. You might be thinking that we need to search through the existing passwords to count up the ones that have `dish` in them. (Hmmm, searching! That's our first inkling that a hash table may be needed here.) If you do that, you'll find that two of the passwords—`brandish` and `radishes`—have `dish` in them. The answer is therefore `2`.

And what about the final query ❹? We're looking for passwords that have an `a` in them. If you search through the three existing passwords, you'll find that all three of them do! The answer is therefore `3`.

We're done! The correct output for the full test case is:

```
0
2
3
```

If we implement the solution strategy that we just used, we might arrive at something like Listing 1-14.

```
❶ #define MAX_USERS 100000
  #define MAX_PASSWORD 10

  int main(void) {
    static char users[MAX_USERS][MAX_PASSWORD + 1];
    int num_ops, op, op_type, total, j;
    char password[MAX_PASSWORD + 1];
    int num_users = 0;
    scanf("%d", &num_ops);
    for (op = 0; op < num_ops; op++) {
      scanf("%d%s", &op_type, password);

   ❷ if (op_type == 1) {
        strcpy(users[num_users], password);
        num_users++;
❸ } else {
      total = 0;
      for (j = 0; j < num_users; j++)
        if (strstr(users[j], password))
          total++;
      printf("%d\n", total);
    }
  }
  return 0;
}
```

Listing 1-14: Solution 1

The problem description says that we'll have at most 100,000 operations. If each is an add operation, then we get 100,000 users ❶, and we can't have any more than that.

For each add operation ❷, we copy the new password into our users array. And for each query operation ❸, we loop through all of the existing user passwords, checking how many of them have the proposed password as a substring.

Like our first solution to Unique Snowflakes, this solution is not fast enough to pass the test cases in time. That's because we have an $O(n^2)$ algorithm here, where n is the number of queries.

We are able to quickly add user passwords to our array—no problem there. What slows us down are the query operations, because each of them has to scan through all existing user passwords. That's where the quadratic-time behavior comes from. Suppose, for example, that a test case starts by adding 50,000 user passwords, and then hammers us with 50,000 queries. Taken together, that would require about 50,000 × 50,000 = 2,500,000,000 steps. That's over 2 billion steps; there's no way that we can do that many in our allowed time limit of three seconds.

Solution 2: Using a Hash Table

We need to speed up the query operations. And we're going to use a hash table to do so. But how? Isn't it just a fact of life that we need to compare each query password with each existing password? No! Read on as we turn the problem on its head.

How to Use the Hash Table

For each query operation, it would be nice if we could just look up the needed password in a hash table to determine how many existing user passwords could get into its account. For example, once we add the users with passwords `brandish`, `radishes`, and `aaa`, then it would be nice to be able to look up `dish` in the hash table and get a value of `2`. But while we're adding those three user passwords, how are we supposed to know to be keeping track of what's going on with `dish`? We don't know which passwords are going to be queried later.

Well, since we don't know the future, let's just add one to the total for every single substring of each user password. That way we'll be ready if we ever need to look any of them up.

Focus on the `brandish` password. If we consider each substring, then we'll increment the total for `b`, `br`, `bra`, `bran`, `brand`, `brandi`, `brandis`, `brandish`, `r`, `ra`, and so on. Don't worry: if we process them all, we'll definitely hit `dish` and increment it. We'll increment `dish` again when we do the same kind of substring processing on `radishes`. So, `dish` will end up with a total of 2, as needed.

You might worry that we're being excessive here, processing a ton of substring passwords, the vast majority of which are not going to be queried. However, remember from the problem description that passwords can be at most 10 characters. Each substring has a starting point and an ending point. In a password of 10 characters, there are only 10 possible starting points and 10 possible ending points, so an upper bound on the number of substrings in a password is $10 \times 10 = 100$. As we have at most 100,000 user passwords, each of which has at most 100 substrings, we'll store at most $100{,}000 \times 100 = 10{,}000{,}000$ substrings in our hash table. That'll take up a few megabytes of memory, for sure, but that's nothing to worry about. We're trading a little memory for the ability to look up any password's total when we need it.

As with Unique Snowflakes, our solution will use a hash table of linked lists. We also need a hash function. We won't use something like the snowflake hash function here, because it would lead to collisions between passwords like `cat` and `act` that are anagrams. Unlike in the Unique Snowflakes problem, passwords should be distinguished not just by their letters but by the locations of those letters. Some collisions are inevitable, of course, but we should do what we can to limit their prevalence. To that end, we'll wield that wild `oaat` hash function from Listing 1-13.

Searching the Hash Table

We'll use the following node to store passwords in our hash table:

```
#define MAX_PASSWORD 10

typedef struct password_node {
  char password[MAX_PASSWORD + 1];
  int total;
  struct password_node *next;
} password_node;
```

This node is similar to `snowflake_node` from Unique Snowflakes, but we now also have a `total` member to keep track of the total count for this password.

Now we can write a helper function to search the hash table for a given password. See Listing 1-15 for the code.

```
#define NUM_BITS 20

password_node *in_hash_table(password_node *hash_table[], char *find) {
  unsigned password_code;
  password_node *password_ptr;
❶ password_code = oaat(find, strlen(find), NUM_BITS);
❷ password_ptr = hash_table[password_code];
  while (password_ptr) {
  ❸ if (strcmp(password_ptr->password, find) == 0)
      return password_ptr;
    password_ptr = password_ptr->next;
  }
  return NULL;
}
```

Listing 1-15: Searching for a password

This `in_hash_table` function takes a hash table and a password to find in the hash table. If the password is found, the function returns a pointer to the corresponding `password_node`; otherwise, it returns `NULL`.

The function works by calculating the hashcode of the password ❶ and using that hashcode to find the appropriate linked list to search ❷. It then checks each password in the list, looking for a match ❸.

Adding to the Hash Table

We also need a function that will add one to a given password in the hash table. See Listing 1-16 for the code.

```
void add_to_hash_table(password_node *hash_table[], char *find) {
  unsigned password_code;
  password_node *password_ptr;
❶ password_ptr = in_hash_table(hash_table, find);
  if (!password_ptr) {
    password_code = oaat(find, strlen(find), NUM_BITS);
    password_ptr = malloc(sizeof(password_node));
    if (password_ptr == NULL) {
      fprintf(stderr, "malloc error\n");
      exit(1);
    }
    strcpy(password_ptr->password, find);
  ❷ password_ptr->total = 0;
    password_ptr->next = hash_table[password_code];
    hash_table[password_code] = password_ptr;
  }
❸ password_ptr->total++;
}
```

Listing 1-16: Adding one to a password's total

We use our in_hash_table function ❶ to determine whether the password is already in the hash table. If it isn't, we add it to the hash table and give it a count of 0 for now ❷. The technique for adding each password to the hash table is the same as for the Unique Snowflakes problem: each bucket is a linked list, and we add each password to the beginning of one of those lists.

Next, whether the password was already in there or not, we increment its total ❸. In that way, a password that we just added will have its total increased from 0 to 1, whereas existing passwords will simply have their total incremented.

The main Function, Take 1

Ready for the main function? Our first attempt is in Listing 1-17.

```
// bugged!
int main(void) {
❶ static password_node *hash_table[1 << NUM_BITS] = {NULL};
  int num_ops, op, op_type, i, j;
  char password[MAX_PASSWORD + 1], substring[MAX_PASSWORD + 1];
  password_node *password_ptr;
  scanf("%d", &num_ops);
  for (op = 0; op < num_ops; op++) {
    scanf("%d%s", &op_type, password);

  ❷ if (op_type == 1) {
      for (i = 0; i < strlen(password); i++)
        for (j = i; j < strlen(password); j++) {
          strncpy(substring, &password[i], j - i + 1);
          substring[j - i + 1] = '\0';
        ❸ add_to_hash_table(hash_table, substring);
        }

  ❹ } else {
    ❺ password_ptr = in_hash_table(hash_table, password);
    ❻ if (!password_ptr)
        printf("0\n");
      else
        printf("%d\n", password_ptr->total);
    }
  }
  return 0;
}
```

Listing 1-17: The main function (bugged!)

To determine the size of the hash table, we've used this strange bit of code: 1 << NUM_BITS ❶. We set NUM_BITS to 20 in Listing 1-15; 1 << 20 is a shortcut for computing 2^{20}, which is 1,048,576. (The oaat hash function requires that the hash table have a number of elements that is a power of 2.) Remember that the maximum number of users we'll have is 100,000; the hash table size that I chose is about 10 times this maximum to account for the fact that we insert multiple strings for each password. Smaller or larger hash tables would have worked fine, too.

For each add operation ❷, we increment the total for each substring by using our `add_to_hash_table` helper function ❸. And for each query operation ❹, we use our `in_hash_table` helper function ❺ to retrieve the total for the password; if the password isn't in the hash table ❻ then we output `0`.

Put all of our functions together and let's try running our code! Remember this test case?

```
6
2 dish
1 brandish
1 radishes
1 aaa
2 dish
2 a
```

The output is supposed to be:

```
0
2
3
```

Unfortunately, our code gives this instead:

```
0
2
5
```

Wait, `5`? Where's that `5` coming from?

Look at the password `aaa`. How many `a` substrings are in there? There are three! And we're going to find each of them, resulting in three increments to the total for `a`. But that doesn't make sense: `aaa` should be able to bump up the total for `a` at most once, not multiple times. After all, `aaa` is only one password.

The main Function, Take 2

What we need to do is make sure that, for each password, each of its substrings counts only once. To do that, we'll maintain an array of all of the substrings that we've generated for the current password. Prior to using a substring, we'll search to make sure that we haven't used that substring yet.

We're introducing a new search here, so it's worth thinking about whether we need a new hash table of substrings. While we could indeed add another hash table for that, we don't need to: as we already argued, each password won't have too many substrings, so a *linear search* (that is, an element-by-element search) through them is going to be fast enough.

Check out Listing 1-18 for the finishing touch.

```
❶ int already_added(char all_substrings[][MAX_PASSWORD + 1],
                    int total_substrings, char *find) {
    int i;
    for (i = 0; i < total_substrings; i++)
      if (strcmp(all_substrings[i], find) == 0)
        return 1;
    return 0;
  }

  int main(void) {
    static password_node *hash_table[1 << NUM_BITS] = {NULL};
    int num_ops, op, op_type, i, j;
    char password[MAX_PASSWORD + 1], substring[MAX_PASSWORD + 1];
    password_node *password_ptr;
    int total_substrings;
    char all_substrings[MAX_PASSWORD * MAX_PASSWORD][MAX_PASSWORD + 1];
    scanf("%d", &num_ops);
    for (op = 0; op < num_ops; op++) {
      scanf("%d%s", &op_type, password);

      if (op_type == 1) {
        total_substrings = 0;
        for (i = 0; i < strlen(password); i++)
          for (j = i; j < strlen(password); j++) {
            strncpy(substring, &password[i], j - i + 1);
            substring[j - i + 1] = '\0';
          ❷ if (!already_added(all_substrings, total_substrings, substring)) {
              add_to_hash_table(hash_table, substring);
              strcpy(all_substrings[total_substrings], substring);
              total_substrings++;
            }
          }

      } else {
        password_ptr = in_hash_table(hash_table, password);
        if (!password_ptr)
          printf("0\n");
        else
          printf("%d\n", password_ptr->total);
      }
    }
    return 0;
  }
```

Listing 1-18: A new helper function and fixed `main` *function*

We have a new `already_added` helper function here ❶ that we'll use to tell us whether the `find` substring is already in the `all_substrings` array for the current password.

In the `main` function itself, notice now that we check whether we've seen the current substring ❷. If we have not, only then do we add it to the hash table.

It's time to submit our code to the judge. Go for it! As with Unique Snowflakes, the speedup from using a hash table amounts to an improvement from $O(n^2)$ to $O(n)$, which is plenty fast for the three-second time limit.

Problem 3: Spelling Check

Sometimes, problems look like they can be solved in a particular way because they bear resemblance to other problems. Here's a problem where it seems that a hash table is appropriate, but on further reflection we see that hash tables vastly overcomplicate what is required.

This is Codeforces problem `39J` (Spelling Check). (The easiest way to find it is to search online for *Codeforces 39J*.)

The Problem

In this problem, we are given two strings where the first string has one more character than the second. Our task is to determine the number of ways in which one character can be deleted from the first string to arrive at the second string. For example, there is one way to get from `favour` to `favor`: we can remove the `u` from the first string.

There are three ways to get from `abcdxxxef` to `abcdxxef`: we can remove any of the `x` characters from the first string.

The context for the problem is a spellchecker. The first string might be `bizzarre` (a misspelled word) and the second might be `bizarre` (a correct spelling). In this case, there are two ways to fix the misspelling—by removing either one of the two `z`s from the first string. The problem is more general, though, having nothing to do with actual English words or spelling mistakes.

Input

The input is two lines, with the first string on the first line and the second string on the second line. Each string can be up to one million characters.

Output

If there is no way to remove a character from the first string to get the second string, output `0`. Otherwise, output two lines:

- On the first line, output the number of ways in which a character can be deleted from the first string to get the second string.
- On the second line, output a space-separated list of the indices of the characters in the first string that can be removed to get the

second string. The problem requires we index a string from `1`, not `0`. (That's a bit annoying, but we'll be careful.)

For example, for this input:

```
abcdxxxef
abcdxxef
```

we would output:

```
3
5 6 7
```

The `5 6 7` are the indices of the three `x` characters in the first string, since we are counting from one (not zero).

The time limit for solving the test cases is two seconds.

Thinking About Hash Tables

I spent a truly embarrassing number of hours searching for the problems that drive the chapters in this book. The problems dictate what I can teach you about the relevant data structure or algorithm. I need the problem solutions to be algorithmically complex, but the problems themselves need to be sufficiently simple so that we can understand what is being asked and keep the relevant details at hand. I really thought I had found exactly that kind of hash table problem I needed for this section. Then I went to solve it.

In Problem 2, Login Mayhem, we were given the passwords as part of the input. That was nice, because we just jammed each substring from the passwords into a hash table and then used the hash table to search for them as needed. Here, in Problem 3, we're not given any such list of strings to insert. Unfazed, when I first tried solving this problem, I created a hash table and I inserted into it each prefix of the second (that is, shorter) string. For example, for the word `abc`, I would have inserted `a`, `ab`, and `abc`. I also created another hash table for the suffixes of the second string. For the word `abc`, I would have inserted `c`, `bc`, and `abc`. Armed with those hash tables, I proceeded to consider each character of the first string. Removing each character is tantamount to splitting the string into a prefix and a suffix. We can just use the hash tables to check whether both the prefix and suffix are present. If they are, then removing that character is one of the ways in which we can transform the first string into the second.

This technique is tempting, right? Want to give it a try?

The thing I had failed to keep in mind was that each string could be up to a million characters long. We certainly can't store all of the prefixes and suffixes themselves in the hash table—that would take up way too much memory. I played around with using pointers in the hash table to point to both the start and end of the prefixes and suffixes. That solves the concerns of memory use, but it doesn't free us from having to compare these extra-long strings whenever we perform a search using the hash table. In Unique Snowflakes and Login Mayhem, the elements in the hash table were small:

6 integers for a snowflake and 10 characters for a password. That's nothing. However, here, the situation is different: we might have strings of a million characters! Comparing such long strings is very time-consuming.

Another timesink here is computing the hashcode of prefixes and suffixes of these strings. We might call `oaat` on a string of length 900,000, and then call it again on a string with one additional character. That duplicates all of the work from the first `oaat` call, when all we wanted was to incorporate one more character into the string being hashed.

Yet, I persisted. I had it in my mind that a hash table was the way to go here, and I failed to consider alternatives. At this point, I probably should have taken a fresh look at the problem. Instead, I learned about *incremental hash functions*, hash functions that are very fast when generating the hashcode for an element that is very similar to the previously hashed element. For example, if I already have the hashcode for `abcde`, then computing the hashcode for `abcdef` using an incremental hash function will be very fast, because it can lean on the work already done for `abcde` rather than starting from scratch.

Another insight was that, if it is too costly to compare extra-long strings, we should try to avoid comparing them at all. We could just hope that our hash function is good enough and that we're lucky enough with the test cases so that no collisions occur. If we look for some element in the hash table, and we find a match . . . well, let's hope it was an actual match and not us getting unlucky with a false positive. If we're willing to make this concession, then we can use a structure that's simpler than the hash table array that we used up to this point in the chapter. In array `prefix1`, each index `i` gives the hashcode for the prefix of length `i` from the first string. In array `prefix2`, each index `i` gives the hashcode for the prefix of length `i` from the second string. In each of two other arrays, we can do similarly for the suffixes of the first string and suffixes of the second string.

Here is some code that shows how the `prefix1` array can be built:

```
// long long is a very large integer type in C99
unsigned long long prefix1[1000001];
prefix1[0] = 0;
for (i = 1; i <= strlen(first_string); i++)
❶ prefix1[i] = prefix1[i - 1] * 39 + first_string[i];
```

The other arrays can be built similarly.

It's important that we use unsigned integers here. In C, overflow is well defined on unsigned integers but not signed integers. If a word is long enough, we'll definitely get overflow, so we don't want to allow undefined behavior.

Now we can use these arrays to determine whether prefixes or suffixes match. For example, to determine whether the first `i` characters of the first string equal the first `i` characters of the second string, just check whether `prefix1[i]` and `prefix2[i]` are equal.

Note how little work it takes to calculate the hashcode for `prefix1[i]` given the hashcode for `prefix1[i - 1]`: it's just a multiplication, followed by adding the new character ❶. Why multiply by 39 and add the character? Why not use something else for the hash function? Honestly, because what I chose didn't lead to any collisions in the Codeforces test cases. Yes, I know, it's unsatisfying.

Not to worry, though: there's a better way! To get there, we'll stare at the problem a little more closely, instead of just jumping to a hash table solution.

An Ad Hoc Solution

Let's think more carefully through an earlier example:

```
abcdxxxef
abcdxxef
```

Suppose that we remove the `f` from the first string (index `9`). Does this make the first string equal the second? No, so `9` will not show up in our space-separated list of indices. The strings have a long prefix of matching characters. There are six such characters to be exact: `abcdxx`. After that, the two strings diverge, where the first string has an `x` and the second has an `e`. If we don't fix that, then we have no hope that the two strings will be equal. The `f` is too far to the right for its deletion to produce equal strings.

That leads to our first observation: if the length of the *longest common prefix* (in our example, six, the length of `abcdxx`) is p, then our only options for deleting characters are those with indices of $\leq p + 1$. In our example, we should consider deleting the characters whose indices are ≤ 7: `a`, `b`, `c`, `d`, the first `x`, the second `x`, and the third `x`. Deleting anything to the right of index $p + 1$ doesn't fix the diverging character at index $p + 1$ and hence can't make the strings equal.

Notice that only some of these deletions actually work. For example, deleting the `a`, `b`, `c`, or `d` from the first string does not give us the second string. Only each of the three deletions of `x` gives us the second string. So, while we've got an upper bound for indices to consider ($\leq p + 1$), we also need a lower bound.

To think about a lower bound, consider removing the `a` from the first string. Does that make the two strings equal? Nope. The reasoning is similar to that in the previous paragraph: there are diverging characters to the right of the `a` that can't possibly be fixed by removing the `a`. If the length of the *longest common suffix* (in our example, four, the length of `xxef`) is s, then we should consider deleting each of the final $s + 1$ characters of the first string. In terms of indices, we're interested only in those that are $\geq n - s$, where n is the length of the first string. In our example, this tells us to consider only indices that are $\geq 9 - 4 = 5$. In the above paragraph, we argued that we should look at only indices that are ≤ 7. Together, we see that indices `5`, `6`, and `7` are the ones whose deletion transforms the first string into the second. As can be seen in Figure 1-4, what matters here are the indices that are included in both the prefixes and suffixes: each of those characters is a valid deletion.

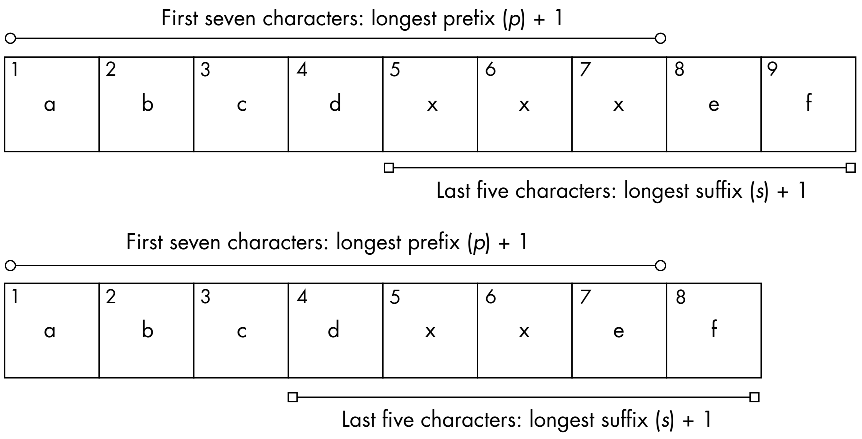

First seven characters: longest prefix (p) + 1
1 a
2 b
3 c
4 d
5 x
6 x
7 x
8 e
9 f
Last five characters: longest suffix (s) + 1
First seven characters: longest prefix (p) + 1
1 a
2 b
3 c
4 d
5 x
6 x
7 e
8 f
Last five characters: longest suffix (s) + 1

Figure 1-4: Overlap between the longest prefix and longest suffix

In general, the indices of interest go from $n - s$ to $p + 1$. For any index in this range, we know from $p + 1$ that the two strings are the same prior to the index. We also know from $n - s$ that the two strings are the same after the index. Therefore, once we remove the index, the two strings are identical. If the range is empty, then there are *no* indices whose deletion transforms the first string into the second, so `0` is output in this case. Otherwise, we use a `for` loop to loop through the indices and `printf` to produce the space-separated list. Let's take a look at the code!

Longest Common Prefix

We have a helper function in Listing 1-19 to calculate the length of the longest common prefix of two strings.

```
int prefix_length(char s1[], char s2[]) {
  int i = 1;
  while (s1[i] == s2[i])
    i++;
  return i - 1;
}
```

Listing 1-19: Calculating the longest common prefix

Here `s1` is the first string and `s2` is the second string. We use `1` as the starting index of the strings. Starting at index `1`, the loop continues as long as corresponding characters are equal. (In a case such as `abcde` and `abcd`, the `e` will fail to match the null terminator at the end of `abcd`, so `i` will correctly end up with value `5`.) When the loop terminates, index `i` is the index of the first mismatched character; therefore, `i - 1` is the length of the longest common prefix.

Longest Common Suffix

Now, to calculate the longest common suffix, we use Listing 1-20.

```
int suffix_length(char s1[], char s2[], int len) {
  int i = len;
  while (i >= 2 && s1[i] == s2[i - 1])
    i--;
  return len - i;
}
```

Listing 1-20: Calculating the longest common suffix

The code is quite similar to Listing 1-19. This time, however, we compare from right to left, rather than left to right. For this reason, we need the `len` parameter, which gives us the length of the first string. The final comparison that we're allowed to make is `i == 2`. If we had `i == 1`, then we'd be accessing `s2[0]`, which is not a valid element of the string!

The main Function

Finally, we have our `main` function in Listing 1-21.

```
#define SIZE 1000000

int main(void) {
❶ static char s1[SIZE + 2], s2[SIZE + 2];
  int len, prefix, suffix, total;
❷ gets(&s1[1]);
❸ gets(&s2[1]);

  len = strlen(&s1[1]);
  prefix = prefix_length(s1, s2);
  suffix = suffix_length(s1, s2, len);
❹ total = (prefix + 1) - (len - suffix) + 1;
❺ if (total < 0)
  ❻ total = 0;

❼ printf("%d\n", total);
❽ for (int i = 0; i < total; i++) {
    printf("%d", i + len - suffix);
    if (i < total - 1)
      printf(" ");
    else
      printf("\n");
  }
  return 0;
}
```

Listing 1-21: The `main` function

We use `SIZE + 2` as the size of our two character arrays ❶. The maximum number of characters that we're required to read is one million, but we need an extra element for the null terminator. And we need one element on top of that because we start indexing our strings at index `1`, "wasting" index `0`.

We read the first ❷ and second string ❸. Notice we pass a pointer to index `1` of each string: `gets` therefore starts storing characters at index `1` rather than index `0`. After calling our helper functions, we calculate the number of indices that can be deleted from `s1` to give us `s2` ❹. If this number is negative ❺, then we set it to `0` ❻. This makes the `printf` call correct ❼. We use a `for` loop ❽ to print the correct indices. We want to start printing at `len - suffix`, so we add `len - suffix` to each integer `i`.

When submitting to the judge, you may need to choose GNU G++ rather than GNU GCC.

There we have it: a linear-time solution. We had to perform some tough analysis, but after that we were able to proceed without complex code and without the need for a hash table. Before considering a hash table, ask yourself, is there anything about the problem that would make hash tables unwieldy? Is a search really necessary, or are there features of the problem that obviate such searching in the first place?

Summary

A hash table is a data structure: a way to organize data so that certain operations are fast. Hash tables speed up the search for some specified element. To speed up other operations, we need other data structures. For example, in Chapter 8, we'll learn about a heap, which is a data structure that can be used when we need to quickly identify the maximum or minimum element in an array.

Data structures are general approaches to organizing and manipulating data. Hash tables apply to all kinds of problems beyond what is shown here; I hope that you now have good intuition for when a hash table can be used. Be on the lookout for other problems where otherwise efficient solutions are hampered by repeated, slow searches.

Notes

Unique Snowflakes is originally from the 2007 Canadian Computing Olympiad.

Login Mayhem is based on a problem from the 2017 Croatian Open Competition in Informatics, Round 1.

Spelling Check is originally from the 2010 School Team Contest #1, hosted by Codeforces. The prefix-suffix solution (used after I finally gave up on a hash table solution) originates from a note posted at *https://codeforces.com/blog/entry/786*.

In our hash table code, we used `malloc` to allocate nodes of our linked lists. It's sometimes possible to avoid using `malloc` and node structures altogether. See "Unique Snowflakes: Implicit Linked Lists" in Appendix B if you're interested in how that can be done.

The `oaat` hash function is by Bob Jenkins (see *http://burtleburtle.net/bob/hash/doobs.html*).

For additional information about hash table applications and implementations, see *Algorithms Illuminated (Part 2): Graph Algorithms and Data Structures* by Tim Roughgarden (2018).