1

HASH TABLES

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 which words are compound words. 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. These two snowflakes are identical because we can choose a starting point for the second snowflake and follow it to the right to get the first snowflake.

Let's try a trickier 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`.

That's our third way for two snowflakes to be identical: two snowflakes are called identical if they match when we move leftward 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, or if we can make them the same by moving leftward through one of the snowflakes.

Input

The first line of input is an integer *n*, which gives 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 zero and at most 10,000,000.

Output

Our output will be a single line of text:

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

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

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 three 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 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 header 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 the code for this function.

```
int identical_right(int snow1[], int snow2[],
                    int start) { //bugged!
  int offset;
  for (offset =0; offset < 6; offset++) {
❶ 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` ❶. 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, `6 % 3` is zero, because there is no remainder when dividing six by three. `6 % 4` is two, because there is two left over when dividing six by four.

We can use mod here to help with the wraparound behavior. Notice that `0 % 6` is zero, `1 % 6` is one, . . ., `5 % 6` is five. Each of these numbers is smaller than six and so will itself be the remainder when dividing six. The numbers zero to five correspond to the legal indices of `snow2`, so it's good that `%` leaves them alone. For our problematic index `6`, `6 % 6` is zero: six divides six 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 to problems that exhibit this same wraparound behavior—for example, `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 work here. In C, `-1 / 6` is zero, leaving a remainder of −1, and so `-1 % 6` is −1. We'd need `-1 % 6` to be five.

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 < 0)
      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 add 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

Whenever 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` on snowflakes is almost, symbol for symbol, the same as the `identify_identical` 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. First, 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 now 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" in the Introduction).

Also notice that I've allocated an array of `100000` 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 OK to hard-code 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 control the number of iterations of the for loop. For each such iteration, we iterate six times, 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. The trouble is the two nested for loops, which 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 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 slow, but let's substitute some values of n into $n(n - 1)/2$. 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 OK 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 maybe four minutes. That's far too slow—we need at most two seconds, 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 4 billion snowflake comparisons in two seconds 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 the rate at which the amount of work grows 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. To get the identical snowflakes close together, we can try sorting the array.

Sorting Snowflakes

C has a library function called `qsort` that makes it easy 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: Comparison function for sorting

Unfortunately, sorting in this way will not help us solve our problem. 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." gets 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. 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.

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 1187. We say that the *code* for the former snowflake is 21 and the code for the latter is 1187.

Our hope is that we can throw the "21 snowflakes" in one bin and throw the "1187 snowflakes" in another, and then we never have to compare the 21s to the 1187s. 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 OK, because our "sum" rule is designed to weed out snowflakes that are clearly not identical. This allows us to avoid comparing all pairs—that's 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,` 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. When we're using at most 100,000 elements, 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 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 in the same location 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 element. To collect these two components of a node, 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 previously used arrays.

The New main

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 adding the snowflake to 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 tra-

verse the list at all. If we had 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` as well. Thus now 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. Incidentally, do we have any 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

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. This 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?

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, in which 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*; the index that a hash function returns for an object is where that object is *hashed*.

Look closely at the code 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 because that's the maximum number of snowflakes that can be presented to our program (according to the problem specification). 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.

The second consideration is our hash function. In our example, 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 they should get hashed 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` 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
  long snowflake[] = {1, 2, 3, 4, 5, 6};
  //2^17 is the smallest power of 2 that is at least 100000
  unsigned long code = oaat((char *)snowflake,
                            sizeof(snowflake), 17);
  printf("%u\n", code);
  return 0;
}
```

Listing 1-13: An intricate hash function

To call `oaat`, we pass three parameters:

`key` The data that we want to hash

`len` The length of those data

`bits` The number of bits that we want in the resulting hashcode

Raising 2 to the power of `bits` tells us the maximum value that a hashcode could have. For example, if we choose 17, then $2^{17} = 131,072$ 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 our hash function will spread the data around.

Indeed, this is why 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 how to cause millions of collisions.

For our 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 that 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 tremendously accelerates Unique Snowflakes. On a typical laptop, a test case with 100,000 elements will take only two seconds to run! No pairwise comparisons of all elements and no sorting is needed, just

a little processing on a bunch of linked lists. In the absence of pathological data, we expect that each linked list will have only a few elements. As such, making all pairwise comparisons within a bucket will take only a small, constant number of steps. We therefore expect hash tables to give us a *linear-time* solution—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. Unlike for Unique Snowflakes, when solving other problems you may be able to begin by sorting the array. A technique called binary search (discussed in Chapter 6) could then be used to quickly search for elements in the sorted array. However, even sorting an array and then using binary search can't compete with the speed of a hash table.

Problem 2: Compound Words

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 for a dramatic speedup. We'll go more quickly than we did for Unique Snowflakes because now we know what to look for.

This is UVa problem 10391.

The Problem

We are given a wordlist in which each word is a lowercase string. For example, we might be given the wordlist containing `crea`, `create`, `open`, and `te`. We'll assume that the strings aren't very long. Our task is to determine the strings in the wordlist that are *compound words*: the concatenation of exactly two other strings in the wordlist. For the given example, only the string `create` is such a compound word, because it is the concatenation of `crea` and `te`.

Input

The input is one string (word) per line, in alphabetical order. We'll get at most 120,000 strings.

Output

The problem requires us to output each compound word on its own line, in alphabetical order.

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

Identifying Compound Words

Once the words have been read in, how do we identify the compound words? Think of the word `create`. There are five opportunities for `create` to be a compound word:

1. If `c` is a word and `reate` is a word
2. If `cr` is a word and `eate` is a word
3. If `cre` is a word and `ate` is a word
4. If `crea` is a word and `te` is a word
5. If `creat` is a word and `e` is a word

On the first iteration, we should search the wordlist for `c` and `reate`. If both searches are successful, we have found a compound word. On the second iteration, we should search the wordlist for `cr` and `eate`. We continue to do this until we have tried all five opportunities, and that's only for the string `create`. Presumably, we'll have other words to check, too, up to 120,000 of them. That's a lot of searching, and searching over and over in a huge wordlist is very time consuming. We'll speed things up with a hash table.

Solution

Our solution will again use a hash table of linked lists. As expected, we also need a hash function.

We won't use something like the snowflake hash function here, because it would lead to collisions between words like `cat` and `act` that are anagrams. Unlike in the Unique Snowflakes problem, words 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.

We use four helper functions in our solution.

Reading a Line

We begin with a helper function to read a line (Listing 1-14).

```
/*based on https://stackoverflow.com/questions/16870485 */
char *read_line(int size) {
  char *str;
  int ch;
  int len = 0;
  str = malloc(size);
  if (str == NULL) {
    fprintf(stderr, "malloc error\n");
    exit(1);
  }
❶ while ((ch = getchar()) != EOF && (ch != '\n')) {
    str[len++] = ch;
    if (len == size) {
      size = size * 2;
   ❷ str = realloc(str, size);
      if (str == NULL) {
        fprintf(stderr, "realloc error\n");
```

```
        exit(1);
      }
    }
  }
❸ str[len] = '\0';
  return str;
}
```

Listing 1-14: Reading a line

Unfortunately, the problem specification does not tell us the maximum length of a line.

We cannot hard-code some maximum word length like 16 or even 100, because we have no control over the input. The `read_line` function therefore takes an initial size that we hope suffices for most or all of the lines. When we call the function, we give an initial size of 16, because that covers most English words we're likely to see. We can use `read_line` to read characters ❶ up until the array reaches its maximum length; if the array fills up and the word still isn't over, it then uses `realloc` to double the array's length ❷, thereby creating more space to read more characters. We're careful to terminate `str` with a null character ❸; otherwise, it wouldn't be a valid string!

Searching the Hash Table

Now we'll create a function in Listing 1-15 to search the hash table for a given word.

```
#define NUM_BITS 17

typedef struct word_node {
  char **word;
  struct word_node *next;
} word_node;

int in_hash_table(word_node *hash_table[], char *find,
                  unsigned find_len) {
  unsigned word_code;
  word_node *wordptr;
❶ word_code = oaat(find, find_len, NUM_BITS);
❷ wordptr = hash_table[word_code];
  while (wordptr) {
  ❸ if ((strlen(*(wordptr->word)) == find_len) &&
        (strncmp(*(wordptr->word), find, find_len) == 0))
      return 1;
    wordptr = wordptr->next;
  }
  return 0;
}
```

Listing 1-15: Searching for a word

The `in_hash_table` function takes a hash table and a word to find in the hash table. If the word is found, the function returns `1`; otherwise, it returns `0`. The third parameter, `find_len`, gives the number of characters in `find` that constitutes the word we're searching for. We need that third parameter because we'll want to be able to search for the beginning of a string; without it, we wouldn't know how many characters we needed to compare.

The function works by calculating the hashcode of the word ❶ and using that hashcode to find the appropriate linked list to search ❷. The hash table contains pointers to strings, not strings themselves, hence the leading * in `*(wordptr->word)` ❸. (As you'll see when studying `main` in Listing 1-17, the hash table contains pointers rather than strings so that we don't store duplicate copies of strings.)

Identifying Compound Words

Now we're ready to check all possible splits of a word to determine whether the word is a compound word; Listing 1-16 does the job.

```
void identify_compound_words(char *words[],
                             word_node *hash_table[],
                             int total_words) {
  int i, j;
  unsigned len;
❶ for (i = 0; i < total_words; i++) {
    len = strlen(words[i]);
  ❷ for (j = 1; j < len; j++) {
    ❸ if (in_hash_table(hash_table, words[i], j) &&
          in_hash_table(hash_table, &words[i][j], len - j)) {
        printf("%s\n", words[i]);
      ❹ break;
      }
    }
  }
}
```

Listing 1-16: Identifying compound words

The function `identify_compound_words` is like `identify_identical` from Unique Snowflakes (Listing 1-12). For each word ❶, it generates all possible splits ❷, and then it searches the hash table for both the prefix string (prior to the split point) and suffix string (from the split point on). We use `j` as the split point ❸. The first search is for the first `j` characters of word `i`. The second searches for the piece of word `i` beginning at index `j` (which has length `len - j`). If both searches are successful, then the current word is a compound word. Notice the use of `break` ❹; without that, a word would be printed multiple times if it had multiple valid splits.

You might be surprised at the use of both a hash table *and* a `words` array. The nodes in the hash table will point to the strings in `words`, but why are there two structures here? Why can't we just use the hash table and not use the `words` array? The reason is that we're required to output the words in

sorted order! Hash tables do not maintain any semblance of sorting—they blast the elements around. We could sort the compound words as a postprocessing step, but we'd be doing work that's already done for us. The words are already sorted when we read them from the input. By going through the words in the `words` array in order, we get the sorting for free.

The main Function

The `main` function is given in Listing 1-17.

```
#define WORD_LENGTH 16

int main(void) {
❶ static char *words[1 << NUM_BITS] = {NULL};
❷ static word_node *hash_table[1 << NUM_BITS] = {NULL};
  int total = 0;
  char *word;
  word_node *wordptr;
  unsigned length, word_code;
  word = read_line(WORD_LENGTH);
  while (*word) {
  ❸ words[total] = word;
    wordptr = malloc(sizeof(word_node));
    if (wordptr == NULL) {
      fprintf(stderr, "malloc error\n");
      exit(1);
    }
    length = strlen(word);
    word_code = oaat(word, length, NUM_BITS);
    wordptr->word = &words[total];
  ❹ wordptr->next = hash_table[word_code];
  ❺ hash_table[word_code] = wordptr;
    word = read_line(WORD_LENGTH);
    total++;
  }
  identify_compound_words(words, hash_table, total);
  return 0;
}
```

Listing 1-17: The main function

To determine the size of the hash table and `words` array, we've used this strange bit of code: `1 << NUM_BITS` at ❶ and ❷. We set `NUM_BITS` to 17 in Listing 1-15; `1 << 17` is a shortcut for computing 2^{17}, which is 131,072. This is the smallest power of 2 larger than 120,000 (the maximum number of words we'll read). The `oaat` hash function requires that the hash table have a number of elements that is a power of 2, so we use 2^{17} for the size of the hash table and `words` array.

With our data structures declared, we can start using our helper functions to populate them. We store each word in the `words` array ❸ and store a

pointer to that word in the hash table at ❹ and ❺. The technique for adding each pointer to the hash table is the same as for the Unique Snowflakes problem: each bucket is a linked list, and we add each pointer to the beginning of one of those lists. Once all words have been read in, we call `identify_compound_word` to produce the desired output.

In summary, the hash table and `words` arrays work in parallel to facilitate a blazing-fast implementation: the hash table gives us a fast search, and `words` helps us process the words in sorted order. Using a naive solution, with no hash table, would be far slower. Consider again the code in Listing 1-16, and suppose we have n words. With a hash table, each search ❸ is expected to take a small constant number of steps. With no hash table, each such search would require a scan of the `words` array and would therefore take up to n steps! As with Unique Snowflakes, the speedup from using a hash table amounts to an improvement from $O(n^2)$ to $O(n)$.

Problem 3: Spelling Check: Deleting a Letter

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). (Probably the easiest way to find it is to Google *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.

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

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).

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, Compound Words, we were given a wordlist as part of the input. That was nice, because we just jammed each word from the wordlist into a hash table and then used the hash table to search for prefixes and suffixes of each word. Here, in Problem 3, we're not given a wordlist. Unfazed, when I first tried solving this problem, I created a hash table, and I inserted into it each prefix and suffix of the second (that is, shorter) string. For example, for the word `abc`, I would have inserted `a`, `ab`, and `abc` (the prefixes) and `c` and `bc` (the suffixes); `abc` is a suffix too but it is already inserted. Armed with that hash table, 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. Now we're back in Compound Words land: we can just use the hash table to check whether both the prefix and suffix are in the hash table. 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? You can even reuse some of the code from Compound Words!

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 Compound Words, the elements in the hash table were small: 6 integers for a snowflake and 16 or so characters for a really long English word. 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 cannot compare strings at all. We could just hope that our hash function is good enough and that we're lucky enough with the test cases and 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 each of three other arrays, we can do similarly for the suffixes of the first string, prefixes of the second 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++)
  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.

Note how easy it is 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. Why did I multiply by 39 and add the character? Why didn't

I 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 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 `a`, `b`, `c`, `d`, the first `x`, the second `x`, and the third `x`. Deleting anything to the right of $p + 1$ 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`s that can't possibly be fixed by removing the `a`s. If the length of the *longest common suffix* (in our example, four, the length of `xxef`) is s, then we should consider only indices 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 ≥ 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.

In summary, 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 1-18 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 1-18: 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; the length of the longest common prefix is therefore `i - 1`.

Longest Common Suffix

Now, to calculate the longest common suffix, we use Listing 1-19.

```
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 1-19: Calculating the longest common suffix

The code is quite similar to Listing 1-18. 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 1-20.

```
#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 1-20: 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. We need just one final element because we start indexing our strings at index `1`, "wasting" index `0`.

We read the first ❷ and second string ❸. Notice that we pass a pointer to index `1` of each string: `gets` will therefore start 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`.

There we have it: a linear-time solution, with no complex code, and no 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 7, 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 storing and manipulating data.The problems in this chapter should give you good intuition for when a hash table can be used, because hash tables apply to all kinds of problems beyond what is shown here. 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.

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*.

The `oaat` (one-at-a-time) 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).