HASHING

Basic Idea

The problem a hash table aims to solve is handling a large range of sparse values. For example, suppose we are given pairs of numbers and words (n, w), and we want to retrieve the word *w* given the number *n*. However, the numbers are very large powers of two. In this case, we cannot simply create an array and place the word *w* at position *n*. Instead, we need to condense the values somehow. For instance, we could use the function $h(n) = log_2(n)$, which maps the *k*th power of two to the number *k*. This effectively compresses the range of values, allowing us to store words at corresponding positions (with the nice property that *h* is one-to-one). Another example could involve pairs of words and numbers (w, n), where, given a word, we need to return the corresponding number. If we know that the words contain only the letters a and b, we could represent the word *w* as a binary number by replacing a with 1 and b with 0, and adding a leading 1 to distinguish between similar patterns such as "bba" and "ba". This drastically reduces the range of values. However, the hash functions we just created are tailored to specific problems and require prior knowledge about the data. Ideally, we want a method that works effectively without needing specific knowledge about the keys aside from the fact that they are sparse.

Hash Functions

This is where hash functions come in. While there are countless hash functions, we'll present a few that are simple, efficient, and suitable for programming contests.

- **For Natural Numbers**

The simplest case for hash functions involves natural numbers. Let *M* be the size of the hash table. We can use the hash function $h(n) = n \% M$.
For integers, we can convert them to positive values by adding the maximum absolute value they can take. To improve upon this simple function, we can use a more general form: $h(n) = (a \cdot n + b) \% M$.

- **For Words**

The most important application of hash tables is when the keys are strings. To create a hash function, let's revisit the example where keys contain only the characters a and b. In that case, we represented words as numbers in base 2. If

we had characters a, b, c, d, and e, we could use base 5. To handle the entire alphabet, we can use base 26, and for the full range of ASCII characters, base 256 works well.

- **Theoretical Properties**

This section introduces terminology for analysing hashing algorithms, though it's not essential for contests. A key concept is independence. A hash function h is k - independent if for any k dinstict values x_1, x_2,…, x_k, the outputs $h(x_1)$, $h(x_2)$,…,$h(x_k)$ are independent random variables. In other words, knowing $h(x_1)$, $h(x_2)$,…,$h(x_{k-1})$ provides no information about $h(x_k)$. The most common way to create k - independent functions for integers is by using a polynomial of degree k with random coefficients, taking the remainder modulo a prime number. In practice, 2-independent hash functions are often sufficient. For example, the function for natural numbers we provided earlier is a first-degree polynomial. If collisions are excessive, a second-degree polynomial could be used, though this is rarely needed in contests.

- **With Lists (Cuckoo Hashing)**

The simplest hash table implementation uses an array of lists. When adding a key-value pair, the pair is appended to the list at the corresponding position. This idea can be improved by using two hash functions, which provide two potential positions for the key. The pair is added to the shorter list of the two. To further optimize, cuckoo hashing is used. Here, we use two hash functions and maintain two tables (one for each function). When inserting an item, we compute its position using both hash functions and place it in the table with the smaller list at that position. This method requires 2-independent hash functions, though in practice, simpler ones often suffice. While using a single table is possible for simplicity; it's generally not recommended.

- **Linear Probing**

A simpler but less intuitive implementation uses a single table. To resolve collisions, when a hash value is already occupied, the algorithm increments the hash value by one (linear probing). The main advantage of this approach is speed. Since all elements are stored in a single table and accessed sequentially, the algorithm is cache-efficient—meaning it uses the computer's memory more effectively. However, for this method to be fast, the table size must be twice the maximum number of elements inserted. Theoretically, linear probing requires 5-independent hash functions, but in practice, simpler hash functions (like the one we presented earlier) suffice.