CHAPTER

12 Hash Tables

The new methods are intended to reduce the amount of space required to contain the hash-coded information from that associated with conventional methods. The reduction in space is accomplished by exploiting the possibility that a small fraction of errors of commission may be tolerable in some applications.

— *"Space/time trade-offs in hash coding with allowable errors,"* B. H. BLOOM, 1970

A hash table is a data structure used to store keys, optionally, with corresponding values. Inserts, deletes and lookups run in $O(1)$ time on average.

The underlying idea is to store keys in an array. A key is stored in the array locations ("slots") based on its "hash code". The hash code is an integer computed from the key by a hash function. If the hash function is chosen well, the objects are distributed uniformly across the array locations.

If two keys map to the same location, a "collision" is said to occur. The standard mechanism to deal with collisions is to maintain a linked list of objects at each array location. If the hash function does a good job of spreading objects across the underlying array and take $O(1)$ time to compute, on average, lookups, insertions, and deletions have $O(1 + n/m)$ time complexity, where n is the number of objects and m is the length of the array. If the "load" n/m grows large, rehashing can be applied to the hash table. A new array with a larger number of locations is allocated, and the objects are moved to the new array. Rehashing is expensive ($O(n + m)$ time) but if it is done infrequently (for example, whenever the number of entries doubles), its amortized cost is low.

A hash table is qualitatively different from a sorted array—keys do not have to appear in order, and randomization (specifically, the hash function) plays a central role. Compared to binary search trees (discussed in Chapter 14), inserting and deleting in a hash table is more efficient (assuming rehashing is infrequent). One disadvantage of hash tables is the need for a good hash function but this is rarely an issue in practice. Similarly, rehashing is not a problem outside of realtime systems and even for such systems, a separate thread can do the rehashing.

A hash function has one hard requirement—equal keys should have equal hash codes. This may seem obvious, but is easy to get wrong, e.g., by writing a hash function that is based on address rather than contents, or by including profiling data.

A softer requirement is that the hash function should "spread" keys, i.e., the hash codes for a subset of objects should be uniformly distributed across the underlying array. In addition, a hash function should be efficient to compute.

A common mistake with hash tables is that a key that's present in a hash table will be updated. The consequence is that a lookup for that key will now fail, even though it's still in the hash table. If you have to update a key, first remove it, then update it, and finally, add it back—this ensures it's moved to the correct array location. As a rule, you should avoid using mutable objects as keys.

Now we illustrate the steps for designing a hash function suitable for strings. First, the hash function should examine all the characters in the string. It should give a large range of values, and should not let one character dominate (e.g., if we simply cast characters to integers and multiplied them, a single 0 would result in a hash code of 0). We would also like a rolling hash function, one in which if a character is deleted from the front of the string, and another added to the end, the new hash code can be computed in $O(1)$ time (see Solution 6.13 on Page 80). The following function has these properties:

```
def string_hash(s, modulus):
    MULT = 997
    return functools.reduce(lambda v, c: (v * MULT + ord(c)) % modulus, s, 0)
```

A hash table is a good data structure to represent a dictionary, i.e., a set of strings. In some applications, a trie, which is a tree data structure that is used to store a dynamic set of strings, has computational advantages. Unlike a BST, nodes in the tree do not store a key. Instead, the node's position in the tree defines the key which it is associated with.

Hash tables boot camp

We introduce hash tables with two examples—one is an application that benefits from the algorithmic advantages of hash tables, and the other illustrates the design of a class that can be used in a hash table.

An application of hash tables

Anagrams are popular word play puzzles, whereby rearranging letters of one set of words, you get another set of words. For example, "eleven plus two" is an anagram for "twelve plus one". Crossword puzzle enthusiasts and Scrabble players benefit from the ability to view all possible anagrams of a given set of letters.

Suppose you were asked to write a program that takes as input a set of words and returns groups of anagrams for those words. Each group must contain at least two words.

For example, if the input is "debitcard", "elvis", "silent", "badcredit", "lives", "freedom", "listen", "levis", "money" then there are three groups of anagrams: (1.) "debitcard", "badcredit"; (2.) "elvis", "lives", "levis"; (3.) "silent", "listen". (Note that "money" does not appear in any group, since it has no anagrams in the set.)

Let's begin by considering the problem of testing whether one word is an anagram of another. Since anagrams do not depend on the ordering of characters in the strings, we can perform the test by sorting the characters in the string. Two words are anagrams if and only if they result in equal strings after sorting. For example, sort("logarithmic") and sort("algorithmic") are both "acghiilmort", so "logarithmic" and "algorithmic" are anagrams.

We can form the described grouping of strings by iterating through all strings, and comparing each string with all other remaining strings. If two strings are anagrams, we do not consider the second string again. This leads to an $O(n^2 m \log m)$ algorithm, where n is the number of strings and m is the maximum string length.

Looking more carefully at the above computation, note that the key idea is to map strings to a representative. Given any string, its sorted version can be used as a unique identifier for the anagram group it belongs to. What we want is a map from a sorted string to the anagrams it

corresponds to. Anytime you need to store a set of strings, a hash table is an excellent choice. Our final algorithm proceeds by adding sort(s) for each string s in the dictionary to a hash table. The sorted strings are keys, and the values are arrays of the corresponding strings from the original input.

```
def find_anagrams(dictionary):
    sorted_string_to_anagrams = collections.defaultdict(list)
    for s in dictionary:
        # Sorts the string, uses it as a key, and then appends the original
        # string as another value into hash table.
        sorted_string_to_anagrams[''.join(sorted(s))].append(s)

    return [
        group for group in sorted_string_to_anagrams.values() if len(group) >= 2
    ]
```

The computation consists of n calls to sort and n insertions into the hash table. Sorting all the keys has time complexity $O(nm \log m)$. The insertions add a time complexity of $O(nm)$, yielding $O(nm \log m)$ time complexity in total.

Variant: Design an $O(nm)$ algorithm for the same problem, assuming strings are made up of lower case English characters.

Design of a hashable class

Consider a class that represents contacts. For simplicity, assume each contact is a string. Suppose it is a hard requirement that the individual contacts are to be stored in a list and it's possible that the list contains duplicates. Two contacts should be equal if they contain the same set of strings, regardless of the ordering of the strings within the underlying list. Multiplicity is not important, i.e., three repetitions of the same contact is the same as a single instance of that contact. In order to be able to store contacts in a hash table, we first need to explicitly define equality, which we can do by forming sets from the lists and comparing the sets.

In our context, this implies that the hash function should depend on the strings present, but not their ordering; it should also consider only one copy if a string appears in duplicate form. It should be pointed out that the hash function and equals methods below are very inefficient. In practice, it would be advisable to cache the underlying set and the hash code, remembering to void these values on updates.

```
class ContactList:
    def __init__(self, names):
        '''
        names is a list of strings.
        '''
        self.names = names

    def __hash__(self):
        # Conceptually we want to hash the set of names. Since the set type is
        # mutable, it cannot be hashed. Therefore we use frozenset.
        return hash(frozenset(self.names))

    def __eq__(self, other):
        return set(self.names) == set(other.names)
```

```
def merge_contact_lists(contacts):
    '''
    contacts is a list of ContactList.
    '''
    return list(set(contacts))
```

The time complexity of computing the hash is $O(n)$, where n is the number of strings in the contact list. Hash codes are often cached for performance, with the caveat that the cache must be cleared if object fields that are referenced by the hash function are updated.

Hash tables have the **best theoretical and real-world performance** for lookup, insert and delete. Each of these operations has $O(1)$ time complexity. The $O(1)$ time complexity for insertion is for the average case—a single insert can take $O(n)$ if the hash table has to be resized.
Consider using a hash code as a **signature** to enhance performance, e.g., to filter out candidates.
Consider using a precomputed `lookup table` instead of boilerplate if-then code for mappings, e.g., from character to value, or character to character.
When defining your own type that will be put in a hash table, be sure you understand the relationship between **logical equality** and the fields the hash function must inspect. Specifically, anytime equality is implemented, it is imperative that the correct hash function is also implemented, otherwise when objects are placed in hash tables, logically equivalent objects may appear in different buckets, leading to lookups returning false, even when the searched item is present.
Sometimes you'll need a `multimap`, i.e., a map that contains multiple values for a single key, or a bi-directional map. If the language's standard libraries do not provide the functionality you need, learn how to implement a multimap using lists as values, or find a **third party library** that has a multimap.

Table 12.1: Top Tips for Hash Tables

Know your hash table libraries

There are multiple hash table-based data structures commonly used in Python—`set`, `dict`, `collections.defaultdict`, and `collections.Counter`. The difference between `set` and the other three is that is `set` simply stores keys, whereas the others store key-value pairs. All have the property that they do not allow for duplicate keys, unlike, for example, `list`.

In a `dict`, accessing value associated with a key that is not present leads to a `KeyError` exception. However, a `collections.defaultdict` returns the default value of the type that was specified when the collection was instantiated, e.g., if `d = collections.defaultdict(list)`, then if `k not in d` then `d[k]` is `[]`. A `collections.Counter` is used for counting the number of occurrences of keys, with a number of set-like operations, as illustrated below.

```
c = collections.Counter(a=3, b=1)
d = collections.Counter(a=1, b=2)
# add two counters together:  c[x] + d[x], collections.Counter({'a': 4, 'b': 3})
c + d
```

```
# subtract (keeping only positive counts), collections.Counter({'a': 2})
c - d
# intersection:  min(c[x], d[x]), collections.Counter({'a': 1, 'b': 1})
c & d
# union:  max(c[x], d[x]), collections.Counter({'a': 3, 'b': 2})
c | d
```

The most important operations for `set` are `s.add(42)`, `s.remove(42)`, `s.discard(123)`, `x in s`, as well as `s <= t` (is `s` a subset of `t`), and `s - t` (elements in `s` that are not in `t`).

The basic operations on the three key-value collections are similar to those on `set`. One difference is with iterators—iteration over a key-value collection yields the keys. To iterate over the key-value pairs, iterate over `items()`; to iterate over values, use `values()`. (The `keys()` method returns an iterator to the keys.)

Not every type is "hashable", i.e., can be added to a `set` or used as a key in a `dict`. In particular, mutable containers are not hashable—this is to prevent a client from modifying an object after adding it to the container, since the lookup will then fail to find it if the slot that the modified object hashes to is different.

Note that the built-in `hash()` function can greatly simplify the implementation of a hash function for a user-defined class, i.e., implementing `__hash(self)__`.

12.1 Test for palindromic permutations

A palindrome is a string that reads the same forwards and backwards, e.g., "level", "rotator", and "foobaraboof".

Write a program to test whether the letters forming a string can be permuted to form a palindrome. For example, "edified" can be permuted to form "deified".

Hint: Find a simple characterization of strings that can be permuted to form a palindrome.

Solution: A brute-force approach is to compute all permutations of the string, and test each one for palindromicity. This has a very high time complexity. Examining the approach in more detail, one thing to note is that if a string begins with say 'a', then we only need consider permutations that end with 'a'. This observation can be used to prune the permutation-based algorithm. However, a more powerful conclusion is that all characters must occur in pairs for a string to be permutable into a palindrome, with one exception, if the string is of odd length. For example, for the string "edified", which is of odd length (7) there are two 'e', two 'd's, two 'i's, and one 'f'—this is enough to guarantee that "edified" can be permuted into a palindrome.

More formally, if the string is of even length, a necessary and sufficient condition for it to be a palindrome is that each character in the string appears an even number of times. If the length is odd, all but one character should appear an even number of times. Both these cases are covered by testing that at most one character appears an odd number of times, which can be checked using a hash table mapping characters to frequencies.

```
def can_form_palindrome(s):
    # A string can be permuted to form a palindrome if and only if the number
    # of chars whose frequencies is odd is at most 1.
    return sum(v % 2 for v in collections.Counter(s).values()) <= 1
```

The time complexity is $O(n)$, where n is the length of the string. The space complexity is $O(c)$, where c is the number of distinct characters appearing in the string.

12.2 Is an anonymous letter constructible?

Write a program which takes text for an anonymous letter and text for a magazine and determines if it is possible to write the anonymous letter using the magazine. The anonymous letter can be written using the magazine if for each character in the anonymous letter, the number of times it appears in the anonymous letter is no more than the number of times it appears in the magazine.

Hint: Count the number of distinct characters appearing in the letter.

Solution: A brute force approach is to count for each character in the character set the number of times it appears in the letter and in the magazine. If any character occurs more often in the letter than the magazine we return false, otherwise we return true. This approach is potentially slow because it iterates over all characters, including those that do not occur in the letter or magazine. It also makes multiple passes over both the letter and the magazine—as many passes as there are characters in the character set.

A better approach is to make a single pass over the letter, storing the character counts for the letter in a single hash table—keys are characters, and values are the number of times that character appears. Next, we make a pass over the magazine. When processing a character c, if c appears in the hash table, we reduce its count by 1; we remove it from the hash when its count goes to zero. If the hash becomes empty, we return true. If we reach the end of the letter and the hash is nonempty, we return false—each of the characters remaining in the hash occurs more times in the letter than the magazine.

```
def is_letter_constructible_from_magazine(letter_text, magazine_text):
    # Compute the frequencies for all chars in letter_text.
    char_frequency_for_letter = collections.Counter(letter_text)

    # Checks if characters in magazine_text can cover characters in
    # char_frequency_for_letter.
    for c in magazine_text:
        if c in char_frequency_for_letter:
            char_frequency_for_letter[c] -= 1
            if char_frequency_for_letter[c] == 0:
                del char_frequency_for_letter[c]
                if not char_frequency_for_letter:
                    # All characters for letter_text are matched.
                    return True

    # Empty char_frequency_for_letter means every char in letter_text can be
    # covered by a character in magazine_text.
    return not char_frequency_for_letter


# Pythonic solution that exploits collections.Counter. Note that the
# subtraction only keeps keys with positive counts.
def is_letter_constructible_from_magazine_pythonic(letter_text, magazine_text):
    return (not collections.Counter(letter_text) -
```

```
        collections.Counter(magazine_text))
```

In the worst-case, the letter is not constructible or the last character of the magazine is essentially required. Therefore, the time complexity is $O(m+n)$ where m and n are the number of characters in the letter and magazine, respectively. The space complexity is the size of the hash table constructed in the pass over the letter, i.e., $O(L)$, where L is the number of distinct characters appearing in the letter.

If the characters are coded in ASCII, we could do away with the hash table and use a 256 entry integer array A, with $A[i]$ being set to the number of times the character i appears in the letter.

12.3 Implement an ISBN cache

The International Standard Book Number (ISBN) is a unique commercial book identifier. It is a string of length 10. The first 9 characters are digits; the last character is a check character. The check character is the sum of the first 9 digits, mod 11, with 10 represented by 'X'. (Modern ISBNs use 13 digits, and the check digit is taken mod 10; this problem is concerned with 10-digit ISBNs.)

Create a cache for looking up prices of books identified by their ISBN. You implement lookup, insert, and remove methods. Use the Least Recently Used (LRU) policy for cache eviction. If an ISBN is already present, insert should not change the price, but it should update that entry to be the most recently used entry. Lookup should also update that entry to be the most recently used entry.

Hint: Amortize the cost of deletion. Alternatively, use an auxiliary data structure.

Solution: Hash tables are ideally suited for fast lookups. We can use a hash table to quickly lookup price by using ISBNs as keys. Along with each key, we store a value, which is the price and the most recent time a lookup was done on that key.

This yields $O(1)$ lookup times on cache hits. Inserts into the cache are also $O(1)$ time, until the cache is full. Once the cache fills up, to add a new entry we have to find the LRU entry, which will be evicted to make place for the new entry. Finding this entry takes $O(n)$ time, where n is the cache size.

One way to improve performance is to use lazy garbage collection. Specifically, let's say we want the cache to be of size n. We do not delete any entries from the hash table until it grows to $2n$ entries. At this point we iterate through the entire hash table, and find the median age of items. Subsequently we discard everything below the median. The worst-case time to delete becomes $O(n)$ but it will happen at most once every n operations. Therefore, the amortized time to delete is $O(1)$. The drawback of this approach is the $O(n)$ time needed for some lookups that miss on a full cache, and the $O(n)$ increase in memory.

An alternative is to maintain a separate queue of keys. In the hash table we store for each key a reference to its location in the queue. Each time an ISBN is looked up and is found in the hash table, it is moved to the front of the queue. (This requires us to use a linked list implementation of the queue, so that items in the middle of the queue can be moved to the head.) When the length of the queue exceeds n, when a new element is added to the cache, the item at the tail of the queue is deleted from the cache, i.e., from the queue and the hash table.

```
class LRUCache:
    def __init__(self, capacity):
```

```
        self._isbn_price_table = collections.OrderedDict()
        self._capacity = capacity

    def lookup(self, isbn):
        if isbn not in self._isbn_price_table:
            return -1
        price = self._isbn_price_table.pop(isbn)
        self._isbn_price_table[isbn] = price
        return price

    def insert(self, isbn, price):
        # We add the value for key only if key is not present - we don't update
        # existing values.
        if isbn in self._isbn_price_table:
            price = self._isbn_price_table.pop(isbn)
        elif self._capacity <= len(self._isbn_price_table):
            self._isbn_price_table.popitem(last=False)
        self._isbn_price_table[isbn] = price

    def erase(self, isbn):
        return self._isbn_price_table.pop(isbn, None) is not None
```

The time complexity for each lookup is $O(1)$ for the hash table lookup and $O(1)$ for updating the queue, i.e., $O(1)$ overall.

12.4 Compute the LCA, optimizing for close ancestors

Problem 9.4 on Page 118 is concerned with computing the LCA in a binary tree with parent pointers in time proportional to the height of the tree. The algorithm presented in Solution 9.4 on Page 118 entails traversing all the way to the root even if the nodes whose LCA is being computed are very close to their LCA.

Design an algorithm for computing the LCA of two nodes in a binary tree. The algorithm's time complexity should depend only on the distance from the nodes to the LCA.

Hint: Focus on the extreme case described in the problem introduction.

Solution: The brute-force approach is to traverse upwards from the one node to the root, recording the nodes on the search path, and then traversing upwards from the other node, stopping as soon as we see a node on the path from the first node. The problem with this approach is that if the two nodes are far from the root, we end up traversing all the way to the root, even if the LCA is the parent of the two nodes, i.e., they are siblings. This is illustrated in by L and N in Figure 9.1 on Page 112.

Intuitively, the brute-force approach is suboptimal because it potentially processes nodes well above the LCA. We can avoid this by alternating moving upwards from the two nodes and storing the nodes visited as we move up in a hash table. Each time we visit a node we check to see if it has been visited before.

```
def lca(node_0, node_1):
    iter_0, iter_1 = node_0, node_1
    nodes_on_path_to_root = set()
```

```
    while iter_0 or iter_1:
        # Ascend tree in tandem for these two nodes.
        if iter_0:
            if iter_0 in nodes_on_path_to_root:
                return iter_0
            nodes_on_path_to_root.add(iter_0)
            iter_0 = iter_0.parent
        if iter_1:
            if iter_1 in nodes_on_path_to_root:
                return iter_1
            nodes_on_path_to_root.add(iter_1)
            iter_1 = iter_1.parent
    raise ValueError('node_0 and node_1 are not in the same tree')
```

Note that we are trading space for time. The algorithm for Solution 9.4 on Page 118 used $O(1)$ space and $O(h)$ time, whereas the algorithm presented above uses $O(D0 + D1)$ space and time, where $D0$ is the distance from the LCA to the first node, and $D1$ is the distance from the LCA to the second node. In the worst-case, the nodes are leaves whose LCA is the root, and we end up using $O(h)$ space and time, where h is the height of the tree.

12.5 Find the nearest repeated entries in an array

People do not like reading text in which a word is used multiple times in a short paragraph. You are to write a program which helps identify such a problem.

Write a program which takes as input an array and finds the distance between a closest pair of equal entries. For example, if s = ⟨"All", "work", "and", "no", "play", "makes", "for", "no", "work", "no", "fun", "and", "no", "results"⟩, then the second and third occurrences of "no" is the closest pair.

Hint: Each entry in the array is a candidate.

Solution: The brute-force approach is to iterate over all pairs of entries, check if they are the same, and if so, if the distance between them is less than the smallest such distance seen so far. The time complexity is $O(n^2)$, where n is the array length.

We can improve upon the brute-force algorithm by noting that when examining an entry, we do not need to look at every other entry—we only care about entries which are the same. We can store the set of indices corresponding to a given value using a hash table and iterate over all such sets. However, there is a better approach—when processing an entry, all we care about is the closest previous equal entry. Specifically, as we scan through the array, for each value seen so far, we store in a hash table the latest index at which it appears. When processing the element, we use the hash table to see the latest index less than the current index holding the same value.

For the given example, when processing the element at index 9, which is "no", the hash table tells us the most recent previous occurrence of "no" is at index 7, so we update the distance of the closest pair of equal entries seen so far to 2.

```
def find_nearest_repetition(paragraph):
    word_to_latest_index, nearest_repeated_distance = {}, float('inf')
    for i, word in enumerate(paragraph):
        if word in word_to_latest_index:
```

```
            latest_equal_word = word_to_latest_index[word]
            nearest_repeated_distance = min(nearest_repeated_distance,
                                            i - latest_equal_word)
        word_to_latest_index[word] = i
    return nearest_repeated_distance if nearest_repeated_distance != float(
        'inf') else -1
```

The time complexity is $O(n)$, since we perform a constant amount of work per entry. The space complexity is $O(d)$, where d is the number of distinct entries in the array.

12.6 Find the smallest subarray covering all values

When you type keywords in a search engine, the search engine will return results, and each result contains a digest of the web page, i.e., a highlighting within that page of the keywords that you searched for. For example, a search for the keywords "Union" and "save" on a page with the text of the Emancipation Proclamation should return the result shown in Figure 12.1.

> My paramount object in this struggle is to **save the Union**, and is not either to save or to destroy slavery. If I could save the Union without freeing any slave I would do it, and if I could save it by freeing all the slaves I would do it; and if I could save it by freeing some and leaving others alone I would also do that.

Figure 12.1: Search result with digest in boldface and search keywords underlined.

The digest for this page is the text in boldface, with the keywords underlined for emphasis. It is the shortest substring of the page which contains all the keywords in the search. The problem of computing the digest is abstracted as follows.

Write a program which takes an array of strings and a set of strings, and return the indices of the starting and ending index of a shortest subarray of the given array that "covers" the set, i.e., contains all strings in the set.

Hint: What is the maximum number of minimal subarrays that can cover the query?

Solution: The brute force approach is to iterate over all subarrays, testing if the subarray contains all strings in the set. If the array length is n, there are $O(n^2)$ subarrays. Testing whether the subarray contains each string in the set is an $O(n)$ operation using a hash table to record which strings are present in the subarray. The overall time complexity is $O(n^3)$.

We can improve the time complexity to $O(n^2)$ by growing the subarrays incrementally. Specifically, we can consider all subarrays starting at i in order of increasing length, stopping as soon as the set is covered. We use a hash table to record which strings in the set remain to be covered. Each time we increment the subarray length, we need $O(1)$ time to update the set of remaining strings.

We can further improve the algorithm by noting that when we move from i to $i+1$ we can reuse the work performed from i. Specifically, let's say the smallest subarray starting at i covering the set ends at j. There is no point in considering subarrays starting at $i+1$ and ending before j, since we know they cannot cover the set. When we advance to $i+1$, either we still cover the set, or we have to advance j to cover the set. We continuously advance one of i or j, which implies an $O(n)$ time complexity.

As a concrete example, consider the array $\langle apple, banana, apple, apple, dog, cat, apple, dog, banana, apple, cat, dog\rangle$ and the set $\{banana, cat\}$. The smallest subarray covering the set starting at 0 ends at 5. Next, we advance to 1. Since the element at 0 is not in the set, the smallest subarray covering the set still ends at 5. Next, we advance to 2. Now we do not cover the set, so we advance from 5 to 8—now the subarray from 2 to 8 covers the set. We update the start index from 2 to 3 to 4 to 5 and continue to cover the set. When we advance to 6, we no longer cover the set, so we advance the end index till we get to 10. We can advance the start index to 8 and still cover the set. After we move past 8, we cannot cover the set. The shortest subarray covering the set is from 8 to 10.

```
Subarray = collections.namedtuple('Subarray', ('start', 'end'))


def find_smallest_subarray_covering_set(paragraph, keywords):
    keywords_to_cover = collections.Counter(keywords)
    result = Subarray(-1, -1)
    remaining_to_cover = len(keywords)
    left = 0
    for right, p in enumerate(paragraph):
        if p in keywords:
            keywords_to_cover[p] -= 1
            if keywords_to_cover[p] >= 0:
                remaining_to_cover -= 1

        # Keeps advancing left until keywords_to_cover does not contain all
        # keywords.
        while remaining_to_cover == 0:
            if result == (-1, -1) or right - left < result[1] - result[0]:
                result = (left, right)
            pl = paragraph[left]
            if pl in keywords:
                keywords_to_cover[pl] += 1
                if keywords_to_cover[pl] > 0:
                    remaining_to_cover += 1
            left += 1
    return result
```

The complexity is $O(n)$, where n is the length of the array, since for each of the two indices we spend $O(1)$ time per advance, and each is advanced at most $n - 1$ times.

The disadvantage of this approach is that we need to keep the subarrays in memory. We can achieve a streaming algorithm by keeping track of latest occurrences of query keywords as we process A. We use a doubly linked list L to store the last occurrence (index) of each keyword in Q, and hash table H to map each keyword in Q to the corresponding node in L. Each time a word in Q is encountered, we remove its node from L (which we find by using H), create a new node which records the current index in A, and append the new node to the end of L. We also update H. By doing this, each keyword in L is ordered by its order in A; therefore, if L has n_Q words (i.e., all keywords are shown) and the current index minus the index stored in the first node in L is less than current best, we update current best. The complexity is still $O(n)$.

```
def find_smallest_subarray_covering_subset(stream, query_strings):
    class DoublyLinkedListNode:
        def __init__(self, data=None):
```

```
        self.data = data
        self.next = self.prev = None

class LinkedList:
    def __init__(self):
        self.head = self.tail = None
        self._size = 0

    def __len__(self):
        return self._size

    def insert_after(self, value):
        node = DoublyLinkedListNode(value)
        node.prev = self.tail
        if self.tail:
            self.tail.next = node
        else:
            self.head = node
        self.tail = node
        self._size += 1

    def remove(self, node):
        if node.next:
            node.next.prev = node.prev
        else:
            self.tail = node.prev
        if node.prev:
            node.prev.next = node.next
        else:
            self.head = node.next
        node.next = node.prev = None
        self._size -= 1

# Tracks the last occurrence (index) of each string in query_strings.
loc = LinkedList()
d = {s: None for s in query_strings}
result = Subarray(-1, -1)
for idx, s in enumerate(stream):
    if s in d:  # s is in query_strings.
        it = d[s]
        if it is not None:
            # Explicitly remove s so that when we add it, it's the string most
            # recently added to loc.
            loc.remove(it)
        loc.insert_after(idx)
        d[s] = loc.tail

        if len(loc) == len(query_strings):
            # We have seen all strings in query_strings, let's get to work.
            if (result == (-1, -1)
                    or idx - loc.head.data < result[1] - result[0]):
                result = (loc.head.data, idx)
return result
```

Variant: Given an array A, find a shortest subarray $A[i, j]$ such that each distinct value present in A is also present in the subarray.

Variant: Given an array A, rearrange the elements so that the shortest subarray containing all the distinct values in A has maximum possible length.

Variant: Given an array A and a positive integer k, rearrange the elements so that no two equal elements are k or less apart.

12.7 Find smallest subarray sequentially covering all values

In Problem 12.6 on Page 168 we did not differentiate between the order in which keywords appeared. If the digest has to include the keywords in the order in which they appear in the search textbox, we may get a different digest. For example, for the search keywords "Union" and "save", in that order, the digest would be "**Union, and is not either to save**".

Write a program that takes two arrays of strings, and return the indices of the starting and ending index of a shortest subarray of the first array (the "paragraph" array) that "sequentially covers", i.e., contains all the strings in the second array (the "keywords" array), in the order in which they appear in the keywords array. You can assume all keywords are distinct. For example, let the paragraph array be ⟨apple, banana, cat, apple⟩, and the keywords array be ⟨banana, apple⟩. The paragraph subarray starting at index 0 and ending at index 1 does not fulfill the specification, even though it contains all the keywords, since they do not appear in the specified order. On the other hand, the subarray starting at index 1 and ending at index 3 does fulfill the specification.

Hint: For each index in the paragraph array, compute the shortest subarray ending at that index which fulfills the specification.

Solution: The brute-force approach is to iterate over all subarrays of the paragraph array. To check whether a subarray of the paragraph array sequentially covers the keyword array, we search for the first occurrence of the first keyword. We never need to consider a later occurrence of the first keyword, since subsequent occurrences do not give us any additional power to cover the keywords. Next we search for the first occurrence of the second keyword that appears after the first occurrence of the first keyword. No earlier occurrence of the second keyword is relevant, since those occurrences can never appear in the correct order. This observation leads to an $O(n)$ time algorithm for testing whether a subarray fulfills the specification, where n is the length of the paragraph array. Since there are $O(n^2)$ subarrays of the paragraph array, the overall time complexity is $O(n^3)$.

The brute-force algorithm repeats work. We can improve the time complexity to $O(n^2)$ by computing for each index, the shortest subarray starting at that index which sequentially covers the keyword array. The idea is that we can compute the desired subarray by advancing from the start index and marking off the keywords in order.

The improved algorithm still repeats work—as we advance through the paragraph array, we can reuse our computation of the earliest occurrences of keywords. To do this, we need auxiliary data structures to record previous results.

Specifically, we use a hash table to map keywords to their most recent occurrences in the paragraph array as we iterate through it, and a hash table mapping each keyword to the length of the shortest subarray ending at the most recent occurrence of that keyword.

These two hash tables give us is the ability to determine the shortest subarray sequentially covering the first k keywords given the shortest subarray sequentially covering the first $k-1$ keywords.

When processing the ith string in the paragraph array, if that string is the jth keyword, we update the most recent occurrence of that keyword to i. The shortest subarray ending at i which sequentially covers the first j keywords consists of the shortest subarray ending at the most recent occurrence of the first $j-1$ keywords plus the elements from the most recent occurrence of the $(j-1)$th keyword to i. This computation is implemented below.

```
Subarray = collections.namedtuple('Subarray', ('start', 'end'))


def find_smallest_sequentially_covering_subset(paragraph, keywords):
    # Maps each keyword to its index in the keywords array.
    keyword_to_idx = {k: i for i, k in enumerate(keywords)}

    # Since keywords are uniquely identified by their indices in keywords
    # array, we can use those indices as keys to lookup in an array.
    latest_occurrence = [-1] * len(keywords)
    # For each keyword (identified by its index in keywords array), the length
    # of the shortest subarray ending at the most recent occurrence of that
    # keyword that sequentially cover all keywords up to that keyword.
    shortest_subarray_length = [float('inf')] * len(keywords)

    shortest_distance = float('inf')
    result = Subarray(-1, -1)
    for i, p in enumerate(paragraph):
        if p in keyword_to_idx:
            keyword_idx = keyword_to_idx[p]
            if keyword_idx == 0:  # First keyword.
                shortest_subarray_length[keyword_idx] = 1
            elif shortest_subarray_length[keyword_idx - 1] != float('inf'):
                distance_to_previous_keyword = (
                    i - latest_occurrence[keyword_idx - 1])
                shortest_subarray_length[keyword_idx] = (
                    distance_to_previous_keyword +
                    shortest_subarray_length[keyword_idx - 1])
            latest_occurrence[keyword_idx] = i

            # Last keyword, for improved subarray.
            if (keyword_idx == len(keywords) - 1
                    and shortest_subarray_length[-1] < shortest_distance):
                shortest_distance = shortest_subarray_length[-1]
                result = Subarray(i - shortest_distance + 1, i)
    return result
```

Processing each entry of the paragraph array entails a constant number of lookups and updates, leading to an $O(n)$ time complexity, where n is the length of the paragraph array. The additional space complexity is dominated by the three hash tables, i.e., $O(m)$, where m is the number of keywords.

12.8 Find the longest subarray with distinct entries

Write a program that takes an array and returns the length of a longest subarray with the property that all its elements are distinct. For example, if the array is $\langle f, s, f, e, t, w, e, n, w, e \rangle$ then a longest subarray all of whose elements are distinct is $\langle s, f, e, t, w \rangle$.

Hint: What should you do if the subarray from indices i to j satisfies the property, but the subarray from i to $j+1$ does not?

Solution: We begin with a brute-force approach. For each subarray, we test if all its elements are distinct using a hash table. The time complexity is $O(n^3)$, where n is the array length since there are $O(n^2)$ subarrays, and their average length is $O(n)$.

We can improve on the brute-force algorithm by noting that if a subarray contains duplicates, every array containing that subarray will also contain duplicates. Therefore, for any given starting index, we can compute the longest subarray starting at that index containing no duplicates in time $O(n)$, since we can incrementally add elements to the hash table of elements from the starting index. This leads to an $O(n^2)$ algorithm. As soon as we get a duplicate, we cannot find a longer beginning at the same initial index that is duplicate-free.

We can improve the time complexity by reusing previous computation as we iterate through the array. Suppose we know the longest duplicate-free subarray ending at a given index. The longest duplicate-free subarray ending at the next index is either the previous subarray appended with the element at the next index, if that element does not appear in the longest duplicate-free subarray at the current index. Otherwise it is the subarray beginning at the most recent occurrence of the element at the next index to the next index. To perform this case analysis as we iterate, all we need is a hash table storing the most recent occurrence of each element, and the longest duplicate-free subarray ending at the current element.

For the given example, $\langle f, s, f, e, t, w, e, n, w, e \rangle$, when we process the element at index 2, the longest duplicate-free subarray ending at index 1 is from 0 to 1. The hash table tells us that the element at index 2, namely f, appears in that subarray, so we update the longest subarray ending at index 2 to being from index 1 to 2. Indices 3–5 introduce fresh elements. Index 6 holds a repeated value, e, which appears within the longest subarray ending at index 5; specifically, it appears at index 3. Therefore, the longest subarray ending at index 6 to start at index 4.

```
def longest_subarray_with_distinct_entries(A):
    # Records the most recent occurrences of each entry.
    most_recent_occurrence = {}
    longest_dup_free_subarray_start_idx = result = 0
    for i, a in enumerate(A):
        # Defer updating dup_idx until we see a duplicate.
        if a in most_recent_occurrence:
            dup_idx = most_recent_occurrence[a]
            # A[i] appeared before. Did it appear in the longest current
            # subarray?
            if dup_idx >= longest_dup_free_subarray_start_idx:
                result = max(result, i - longest_dup_free_subarray_start_idx)
                longest_dup_free_subarray_start_idx = dup_idx + 1
        most_recent_occurrence[a] = i
    return max(result, len(A) - longest_dup_free_subarray_start_idx)
```

The time complexity is $O(n)$, since we perform a constant number of operations per element.

12.9 Find the length of a longest contained interval

Write a program which takes as input a set of integers represented by an array, and returns the size of a largest subset of integers in the array having the property that if two integers are in the subset, then so are all integers between them. For example, if the input is $\langle 3, -2, 7, 9, 8, 1, 2, 0, -1, 5, 8\rangle$, the largest such subset is $\{-2, -1, 0, 1, 2, 3\}$, so you should return 6.

Hint: Do you really need a total ordering on the input?

Solution: The brute-force algorithm is to sort the array and then iterate through it, recording for each entry the largest subset with the desired property ending at that entry.

On closer inspection we see that sorting is not essential to the functioning of the algorithm. We do not need the total ordering—all we care are about is whether the integers adjacent to a given value are present. This suggests using a hash table to store the entries. Now we iterate over the entries in the array. If an entry e is present in the hash table, we compute the largest interval including e such that all values in the interval are contained in the hash table. We do this by iteratively searching entries in the hash table of the form $e+1, e+2, \ldots$, and $e-1, e-2, \ldots$. When we are done, to avoid doing duplicated computation we remove all the entries in the computed interval from the hash table, since all these entries are in the same largest contained interval.

As a concrete example, consider $A = \langle 10, 5, 3, 11, 6, 100, 4\rangle$. We initialize the hash table to $\{6, 10, 3, 11, 5, 100, 4\}$. The first entry in A is 10, and we find the largest interval contained in A including 10 by expanding from 10 in each direction by doing lookups in the hash table. The largest set is $\{10, 11\}$ and is of size 2. This computation updates the hash table to $\{6, 3, 5, 100, 4\}$. The next entry in A is 5. Since it is contained in the hash table, we know that the largest interval contained in A including 5 has not been computed yet. Expanding from 5, we see that $3, 4, 6$ are all in the hash table, and 2 and 7 are not in the hash table, so the largest set containing 5 is $\{3, 4, 5, 6\}$, which is of size 4. We update the hash table to $\{100\}$. The three entries after 5, namely $3, 11, 6$ are not present in the hash table, so we know we have already computed the longest intervals in A containing each of these. Then we get to 100, which cannot be extended, so the largest set containing it is $\{100\}$, which is of size 1. We update the hash table to $\{\}$. Since 4 is not in the hash table, we can skip it. The largest of the three sets is $\{3, 4, 5, 6\}$, i.e., the size of the largest contained interval is 4.

```
def longest_contained_range(A):
    # unprocessed_entries records the existence of each entry in A.
    unprocessed_entries = set(A)

    max_interval_size = 0
    while unprocessed_entries:
        a = unprocessed_entries.pop()

        # Finds the lower bound of the largest range containing a.
        lower_bound = a - 1
        while lower_bound in unprocessed_entries:
            unprocessed_entries.remove(lower_bound)
            lower_bound -= 1
```

```
        # Finds the upper bound of the largest range containing a.
        upper_bound = a + 1
        while upper_bound in unprocessed_entries:
            unprocessed_entries.remove(upper_bound)
            upper_bound += 1

        max_interval_size = max(max_interval_size,
                                upper_bound - lower_bound - 1)
    return max_interval_size
```

The time complexity of this approach is $O(n)$, where n is the array length, since we add and remove array elements in the hash table no more than once.

12.10 Compute all string decompositions

This problem is concerned with taking a string (the "sentence" string) and a set of strings (the "words"), and finding the substrings of the sentence which are the concatenation of all the words (in any order). For example, if the sentence string is "amanaplanacanal" and the set of words is {"can","apl","ana"}, "aplanacan" is a substring of the sentence that is the concatenation of all words.

Write a program which takes as input a string (the "sentence") and an array of strings (the "words"), and returns the starting indices of substrings of the sentence string which are the concatenation of all the strings in the words array. Each string must appear exactly once, and their ordering is immaterial. Assume all strings in the words array have equal length. It is possible for the words array to contain duplicates.

Hint: Exploit the fact that the words have the same length.

Solution: Let's begin by considering the problem of checking whether a string is the concatenation strings in words. We can solve this problem recursively—we find a string from words that is a prefix of the given string, and recurse with the remaining words and the remaining suffix.

When all strings in words have equal length, say n, only one distinct string in words can be a prefix of the given string. So we can directly check the first n characters of the string to see if they are in words. If not, the string cannot be the concatenation of words. If it is, we remove that string from words and continue with the remainder of the string and the remaining words.

To find substrings in the sentence string that are the concatenation of the strings in words, we can use the above process for each index in the sentence as the starting index.

```
def find_all_substrings(s, words):
    def match_all_words_in_dict(start):
        curr_string_to_freq = collections.Counter()
        for i in range(start, start + len(words) * unit_size, unit_size):
            curr_word = s[i:i + unit_size]
            it = word_to_freq[curr_word]
            if it == 0:
                return False
            curr_string_to_freq[curr_word] += 1
            if curr_string_to_freq[curr_word] > it:
                # curr_word occurs too many times for a match to be possible.
```

```
                return False
        return True

    word_to_freq = collections.Counter(words)
    unit_size = len(words[0])
    return [
        i for i in range(len(s) - unit_size * len(words) + 1)
        if match_all_words_in_dict(i)
    ]
```

We analyze the time complexity as follows. Let m be the number of words and n the length of each word. Let N be the length of the sentence. For any fixed i, to check if the string of length nm starting at an offset of i in the sentence is the concatenation of all words has time complexity $O(nm)$, assuming a hash table is used to store the set of words. This implies the overall time complexity is $O(Nnm)$. In practice, the individual checks are likely to be much faster because we can stop as soon as a mismatch is detected.

The problem is made easier, complexity-wise and implementation-wise, by the fact that the words are all the same length—it makes testing if a substring equals the concatenation of words straightforward.

12.11 Test the Collatz conjecture

The Collatz conjecture is the following: Take any natural number. If it is odd, triple it and add one; if it is even, halve it. Repeat the process indefinitely. No matter what number you begin with, you will eventually arrive at 1.

As an example, if we start with 11 we get the sequence $11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1$. Despite intense efforts, the Collatz conjecture has not been proved or disproved.

Suppose you were given the task of checking the Collatz conjecture for the first billion integers. A direct approach would be to compute the convergence sequence for each number in this set.

Test the Collatz conjecture for the first n positive integers.

Hint: How would you efficiently check the conjecture for n assuming it holds for all $m < n$?

Solution: Often interview questions are open-ended with no definite good solution—all you can do is provide a good heuristic and code it well.

The Collatz hypothesis can fail in two ways—a sequence returns to a previous number in the sequence, which implies it will loop forever, or a sequence goes to infinity. The latter cannot be tested with a fixed integer word length, so we simply flag overflows.

The general idea is to iterate through all numbers and for each number repeatedly apply the rules till you reach 1. Here are some of the ideas that you can try to accelerate the check:

- Reuse computation by storing all the numbers you have already proved to converge to 1; that way, as soon as you reach such a number, you can assume it would reach 1.
- To save time, skip even numbers (since they are immediately halved, and the resulting number must have already been checked).
- If you have tested every number up to k, you can stop the chain as soon as you reach a number that is less than or equal to k. You do not need to store the numbers below k in the hash table.

- If multiplication and division are expensive, use bit shifting and addition.
- Partition the search set and use many computers in parallel to explore the subsets, as show in Solution 19.9 on Page 299.

Since the numbers in a sequence may grow beyond 32 bits, you should use 64-bit integer and keep testing for overflow; alternately, you can use arbitrary precision integers.

```
def test_collatz_conjecture(n):
    # Stores odd numbers already tested to converge to 1.
    verified_numbers = set()

    # Starts from 3, hypothesis holds trivially for 1.
    for i in range(3, n + 1):
        sequence = set()
        test_i = i
        while test_i >= i:
            if test_i in sequence:
                # We previously encountered test_i, so the Collatz sequence has
                # fallen into a loop. This disproves the hypothesis, so we
                # short-circuit, returning False.
                return False
            sequence.add(test_i)

            if test_i % 2:  # Odd number.
                if test_i in verified_numbers:
                    break  # test_i has already been verified to converge to 1.
                verified_numbers.add(test_i)
                test_i = 3 * test_i + 1  # Multiply by 3 and add 1.
            else:
                test_i //= 2  # Even number, halve it.
    return True
```

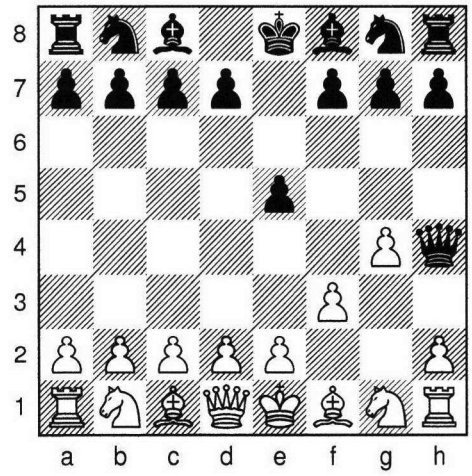
We cannot say much about time complexity beyond the obvious, namely that it is at least proportional to n.

12.12 Implement a hash function for chess

The state of a game of chess is determined by what piece is present on each square, as illustrated in Figure 12.2 on the following page. Each square may be empty, or have one of six classes of pieces; each piece may be black or white. Thus $\lceil \log(1+6\times 2) \rceil = 4$ bits suffice per square, which means that a total of $64 \times 4 = 256$ bits can represent the state of the chessboard. (The actual state of the game is slightly more complex, as it needs to capture which side is to move, castling rights, *en passant*, etc., but we will use the simpler model for this question.)

Chess playing computers need to store sets of states, e.g., to determine if a particular state has been evaluated before, or is known to be a winning state. To reduce storage, it is natural to apply a hash function to the 256 bits of state, and ignore collisions. The hash code can be computed by a conventional hash function for strings. However, since the computer repeatedly explores nearby states, it is advantageous to consider hash functions that can be efficiently computed based on incremental changes to the board.

Design a hash function for chess game states. Your function should take a state and the hash code for that state, and a move, and efficiently compute the hash code for the updated state.



1. f3, e5 2. g4, ♕h4

Figure 12.2: Chessboard corresponding to the fastest checkmate, *Fool's Mate*.

Hint: XOR is associative, commutative, and fast to compute. Additionally, $a \oplus a = 0$.

Solution: A straightforward hash function is to treat the board as a sequence of 64 base 13 digits. There is one digit per square, with the squares numbered from 0 to 63. Each digit encodes the state of a square: blank, white pawn, white rook,. . . ,white king, black pawn, . . . , black king. We use the hash function $\sum_{i=0}^{63} c_i p^i$, where c_i is the digit in location i, and p is a prime (see on Page 159 for more details).

Note that this hash function has some ability to be updated incrementally. If, for example, a black knight taken by a white bishop the new hash code can be computed by subtracting the terms corresponding to the initial location of the knight and bishop, and adding a term for a blank at the initial location of the bishop and a term for the bishop at the knight's original position.

Now we describe a hash function which is much faster to update. It is based on creating a random 64-bit integer code for each of the 13 states that each of the 64 squares can be in. These $13 \times 64 = 832$ random codes are constants in the program. The hash code for the state of the chessboard is the XOR of the code for each location. Updates are very fast—for the example above, we XOR the code for black knight on i_1, white bishop on i_2, white bishop on i_1, and blank on i_2.

Incremental updates to the first hash function entail computing terms like p^i which is more expensive than computing an XOR, which is why the second hash function is preferable. The maximum number of word-level XORs performed is 4, for a capture or a castling.

As an example, consider a simpler game played on a 2×2 board, with at most two pieces, P and Q present on the board. At most one piece can be present at a board position. Denote the board positions by $(0,0)$, $(0,1)$, $(1,0)$, and $(1,1)$. We use the following random 7-bit codes for each individual position:

- For $(0,0)$: $(1100111)_2$ for blank, $(1011000)_2$ for P, $(1100010)_2$ for Q.
- For $(0,1)$: $(1111100)_2$ for blank, $(1000001)_2$ for P, $(0001111)_2$ for Q.
- For $(1,0)$: $(1100101)_2$ for blank, $(1101101)_2$ for P, $(0011101)_2$ for Q.
- For $(1,1)$: $(0100001)_2$ for blank, $(0101100)_2$ for P, $(1001011)_2$ for Q.

Consider the following state: P is present at $(0, 0)$ and Q at $(1, 1)$, with the remaining positions blank. The hash code for this state is $(1011000)_2 \oplus (1111100)_2 \oplus (1100101)_2 \oplus (1001011)_2 = (0001010)_2$. Now to compute the code for the state where Q moves to $(0, 1)$, we XOR the code for the current state with $(1001011)_2$ (removes Q from $(1, 1)$), $(0100001)_2$ (adds blank at $(1, 1)$), $(1111100)_2$ (removes blank from $(0, 1)$), and $(0001111)_2$ (adds Q at $(0, 1)$). Note that, regardless of the size of the board and the number of pieces, this approach uses four XORs to get the updated state.

Variant: How can you include castling rights and *en passant* information in the state?