Interview Cake

# You want to build a word cloud, an infographic where the size of a word corresponds to how often it appears in the body of text.

To do this, you'll need data. Write code that takes a long string and builds its word cloud data in a dictionary.

A **hash table** organizes data so you can quickly look up values for a given key.

| | Average | Worst Case |
|---|---|---|
| **space** | $O(n)$ | $O(n)$ |
| **insert** | $O(1)$ | $O(n)$ |
| **lookup** | $O(1)$ | $O(n)$ |
| **delete** | $O(1)$ | $O(n)$ |

**Strengths:**

- **Fast lookups**. Lookups take $O(1)$ time *on average*.
- **Flexible keys**. Most data types can be used for keys, as long as they're hashable (/concept/hashing).

**Weaknesses:**

- **Slow worst-case lookups**. Lookups take $O(n)$ time *in the worst case*.
- **Unordered**. Keys aren't stored in a special order. If you're looking for the smallest key, the largest key, or all the keys in a range, you'll need to look through every key to find it.
- **Single-directional lookups**. While you can look up the *value* for a given key in $O(1)$ time, looking up the *keys* for a given *value* requires looping through the whole dataset—$O(n)$ time.
- **Not cache-friendly**. Many hash table implementations use linked lists (/concept/linked-list), which don't put data next to each other in memory.

**In Python 3.6**

In Python 3.6, hash tables are called dictionaries.

Python 3.6

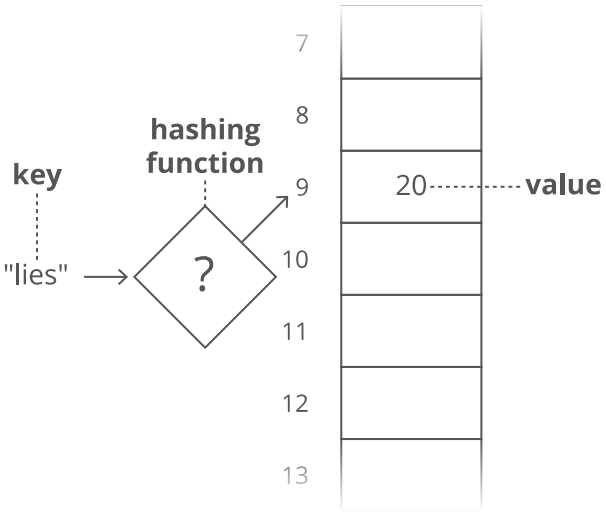
```
light_bulb_to_hours_of_light = {
    'incandescent': 1200,
    'compact fluorescent': 10000,
    'LED': 50000,
}
```

### Hash maps are built on arrays

Arrays (/concept/array) are pretty similar to hash maps already. Arrays let you quickly look up the value for a given "key" . . . except the keys are called "indices," and we don't get to pick them—they're always sequential integers (0, 1, 2, 3, etc).

Think of a hash map as a "hack" on top of an array to let us use flexible keys instead of being stuck with sequential integer "indices."

All we need is a function to convert a key into an array index (an integer). That function is called a **hashing function (/concept/hashing)**.




To look up the value for a given key, we just run the key through our hashing function to get the index to go to in our underlying array to grab the value.

How does that hashing function work? There are a few different approaches, and they can get pretty complicated. But here's a simple proof of concept:

Grab the number value for each character and add those up.

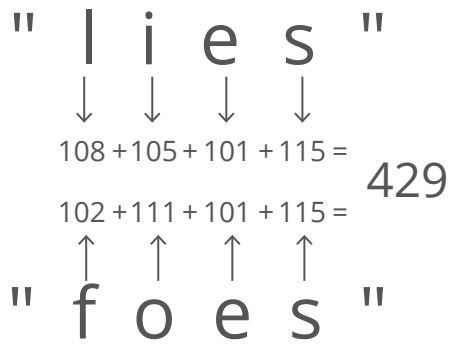

The result is 429. But what if we only have *30* slots in our array? We'll use a common trick for forcing a number into a specific range: the modulus operator (%). (/concept/modulus) Modding our sum by 30 ensures we get a whole number that's less than 30 (and at least 0):

$$429 \% 30 = 9$$

> The hashing functions used in modern systems get pretty complicated—the one we used here is a simplified example.

### Hash collisions

What if two keys hash to the same index in our array? In our example above, look at "lies" and "foes":

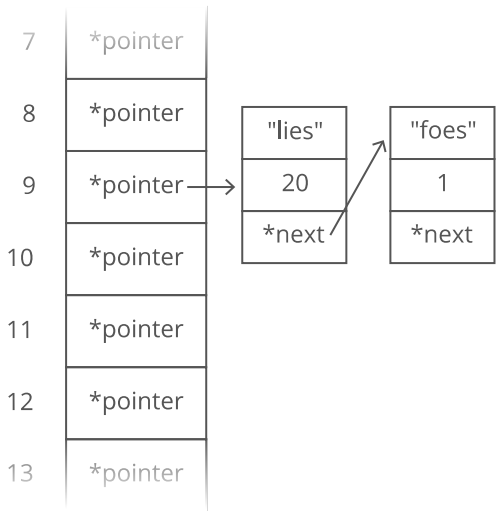



They both sum up to 429! So of course they'll have the same answer when we mod by 30:

$$429 \% 30 = 9$$

This is called a **hash collision**. There are a few different strategies for dealing with them.

Here's a common one: instead of storing the actual values in our array, let's have each array slot hold a *pointer* to a *linked list (/concept/linked-list)* holding the values for all the keys that hash to that index:




Notice that we included the *keys* as well as the values in each linked list node. Otherwise we wouldn't know which key was for which value!

> There are other ways to deal with hash collisions. This is just one of them.

### When hash table operations cost $O(n)$ time

### Hash collisions

If *all* our keys caused hash collisions, we'd be at risk of having to walk through all of our values for a single lookup (in the example above, we'd have one big linked list). This is unlikely, but it *could* happen. That's the worst case.

### Dynamic array resizing

Suppose we keep adding more items to our hash map. As the number of keys and values in our hash map exceeds the number of indices in the underlying array, hash collisions become inevitable.

To mitigate this, we could expand our underlying array whenever things start to get crowded. That requires allocating a larger array and rehashing all of our existing keys to figure out their new position—$O(n)$ time.

### Sets

A **set** is like a hash map except it only stores keys, without values.

Sets often come up when we're tracking groups of items—nodes we've visited in a graph, characters we've seen in a string, or colors used by neighboring nodes. Usually, we're interested in whether something is in a set or not.

Sets are usually implemented very similarly to hash maps—using hashing to index into an array—but they don't have to worry about storing values alongside keys. In Python, the set implementation is largely copied from the dictionary implementation (https://markmail.org/message/ktzomp4uwrmnzao6).

Python 3.6

```
light_bulbs = set()

light_bulbs.add('incandescent')
light_bulbs.add('compact fluorescent')
light_bulbs.add('LED')

'LED' in light_bulbs  # True
'halogen' in light_bulbs  # False
```

, where the keys are words and the values are the number of times the words occurred.

**Think about capitalized words**. For example, look at these sentences:

```
'After beating the eggs, Dana read the next step:'
'Add milk and eggs, then add flour and sugar.'
```

What do we want to do with "After", "Dana", and "add"? In this example, your final dictionary should include *one* "Add" or "add" with a value of 2. Make *reasonable* (not necessarily *perfect*) decisions about cases like "After" and "Dana".

Assume the input will only contain words and standard punctuation.

> You could make a reasonable argument to use **regex** in your solution. We won't, mainly because performance is difficult to measure and can get pretty bad (http://blog.codinghorror.com/regex-performance/).

## Gotchas

Are you sure your code handles hyphenated words and standard punctuation?

Are you sure your code **reasonably handles the same word with different capitalization?**

Try these sentences:

```
"We came, we saw, we conquered...then we ate Bill's (Mille-Feuille) cake."
'The bill came to five dollars.'
```

We can do this in $O(n)$ runtime and space.

The final dictionary we return should be the ***only*** data structure whose length is tied to $n$.

We should only iterate through our input string **once**.

## Breakdown

We'll have to go through the entire input string, and we're returning a dictionary with every unique word. In the worst case every word is different, so our runtime and space cost will both be at least $O(n)$.

This challenge has several parts. Let's break them down.

1. **Splitting the words** from the input string
2. **Populating the dictionary** with each word
3. **Handling words that are both uppercase and lowercase** in the input string

How would you start the first part?

We could use the built-in `split()` function to separate our words, but if we just split on spaces we'd have to iterate over all the words before or after splitting to clean up the punctuation. And consider em dashes or ellipses, which *aren't* surrounded by spaces but nonetheless separate words. Instead, we'll make our *own* `split_words()` function, which will let us iterate over the input string only once.

We'll check if each character is a letter with the string method `isalpha()`.

Now how can we split each word? Let's assume, for now, that our helper function will return a list of words.

We'll iterate over all the characters in the input string. How can we identify when we've reached the end of a word?

Here's a simple example. It doesn't work perfectly yet—you'll need to add code to handle the end of the input string, hyphenated words, punctuation, and edge cases.

Python 3.6

```
def split_words(input_string):
    words = []
    current_word_start_index = 0
    current_word_length = 0

    for i, char in enumerate(input_string):
        if char.isalpha():
            if current_word_length == 0:
                current_word_start_index = i
            current_word_length += 1
        else:
            word = input_string[current_word_start_index:
                current_word_start_index + current_word_length]
            words.append(word)
            current_word_length = 0

    return words
```

> Careful—if you thought of building up the word character by character (using +=), you'd be doing a lot more work than you probably think. Every time we append a character to a string, Python makes a whole new string. If our input is one long word, then creating all these copies is $O(n^2)$ time.

> Instead, we keep track of the *index* where our word starts and its current length. Once we hit a space, we can use string slicing to extract the current word to append to the list. That keeps our split function at $O(n)$ time.

Now we've solved the first part of the challenge, splitting the words. The next part is **populating our dictionary with unique words.** What do we do with each word?

If the word is in the dictionary, we'll increment its count. Otherwise, we'll add it to the dictionary with a count of 1.

Python 3.6

```
words_to_counts = {}

def add_word_to_dictionary(word):
    if word in words_to_counts:
        words_to_counts[word] += 1
    else:
        words_to_counts[word] = 1
```

Alright, last part! **How should we handle words that are uppercase and lowercase?**

Consider these sentences:

```
'We came, we saw, we ate cake.'
'Friends, Romans, countrymen! Let us eat cake.'
'New tourists in New York often wait in long lines for cronuts.'
```

**Take some time to think of possible approaches.** What are some other sentences you might run into. What are all your options?

When are words that *should be* lowercase *not*?
*Why* not?
What are the *ideal* cases we'd want in our dictionary?

Here are a few options:

1. Only make a word uppercase in our dictionary if it is *always* uppercase in the original string.
2. Make a word uppercase in our dictionary if it is *ever* uppercase in the original string.
3. Make a word uppercase in our dictionary if it is ever uppercase in the original string *in a position that is not the first word of a sentence.*

4. Use an API or other tool that identifies proper nouns.
5. Ignore case entirely and make every word lowercase.

**What are the pros and cons for each one?**

Pros and cons include:

1. **Only make a word uppercase in our dictionary if it is *always* uppercase in the original string:** this will have reasonable accuracy in very long strings where words are more likely to be included multiple times, but words that *only* ever occur as the first word in a sentence will always be included as uppercase.
2. **Make a word uppercase in our dictionary if it is *ever* uppercase in the original string:** this will ensure proper nouns are *always* uppercase, but any words that are *ever* at the start of sentences will always be uppercase too.
3. **Make a word uppercase in our dictionary if it is ever uppercase in the original string *in a position that is not the first word of a sentence:*** this addresses the problem with option (2), but proper nouns that are *only* ever at the start of sentences will be made lowercase.
4. **Use an API or other tool that identifies proper nouns:** this has a lot of potential to give us a high level of accuracy, but we'll give up control over decisions, we'll be relying on code we didn't write, and our practical runtime may be significantly increased.
5. **Ignore case entirely and make every word lowercase:** this will give us simplicity and consistency, but we'll lose all accuracy for words that should be uppercase.

Any of these could be considered reasonable. Importantly, **none of them are perfect**. They all have tradeoffs, and it is very difficult to write a highly accurate algorithm. Consider "cliff" and "bill" in these sentences:

```
'Cliff finished his cake and paid the bill.'
'Bill finished his cake at the edge of the cliff.'
```

You can choose whichever of the options you'd like, or another option you thought of. For this breakdown, we're going to choose option (1).

Now, how do we update our `add_word_to_dictionary()` function to avoid duplicate words?

Think about the different possibilities:

1. The word is **uppercase or lowercase.**
2. The word is **already in the dictionary** or not.
3. **A different case of the word is already in the dictionary** or not.

Moving forward, we can either:

1. Check for words that are in the dictionary in **both** cases *when we're done populating the dictionary*. If we add "Vanilla" three times and "vanilla" eight times, we'll combine them into *one* "vanilla" at the end with a value 11.
2. *Avoid **ever** having a word in our dictionary that's both uppercase and lowercase.* As we add "Vanilla"s and "vanilla"s, we'd *always only ever have one version* in our dictionary.

We'll choose the second approach since it will save us a walk through our dictionary. How should we start?

If the word we're adding is already in the dictionary in its current case, let's increment its count. What if it's *not* in the dictionary?

There are three possibilities:

1. **A lowercase version is in the dictionary** (in which case we *know* our input word is uppercase, because if it is lowercase and already in the dictionary it would have passed our first check and we'd have just incremented its count)
2. **An uppercase version is in the dictionary** (so we *know* our input word is lowercase)
3. **The word is not in the dictionary** in any case

Let's start with the first possibility. What do we want to do?

Because we only include a word as uppercase if it is always uppercase, we simply increment the lowercase version's count.

Python 3.6

```
# Current dictionary
# {'blue': 3}

# Adding
# 'Blue'

# Code
words_to_counts[word.lower()] += 1

# Updated dictionary
# {'blue': 4}
```

What about the second possibility?

This is a little more complicated. We need to remove the uppercase version from our dictionary if we encounter a lowercase version. **But we still need the uppercase version's count!**

Python 3.6

```
# Current dictionary
# {'Yellow': 6}

# Adding
# 'yellow'

# Code (we will write our "capitalize()" method later)
words_to_counts[word] = 1
words_to_counts[word] += words_to_counts[word.capitalize()]
del words_to_counts[word.capitalize()]

# Updated dictionary
# {'yellow': 7}
```

Finally, what if the word is not in the dictionary at all?

Easy—we add it and give it a count of 1.

Python 3.6

```
# Current dictionary
# {'purple': 2}

# Adding
# 'indigo'

# Code
words_to_counts[word] = 1

# Updated dictionary
# {'purple': 2, 'indigo': 1}
```

Now we have all our pieces! We can split words, add them to a dictionary, and track the number of times each word occurs without having duplicate words of the same case. Can we improve our solution?

Let's look at our runtime and space cost. We iterate through every character in the input string once and then every word in our list once. That's a runtime of $O(n)$, which is the best we can achieve for this challenge (we *have* to look at the entire input string). The space we're using includes a list for each word and a dictionary for every unique word. Our worst case is that every word is different, so our space cost is also $O(n)$, which is also the best we can achieve for this challenge (we *have* to return a dictionary of words).

**But we can still make some optimizations!**

How can we make our *space cost* even smaller?

We're storing all our split words in a separate list. That at least doubles the memory we use! How can we eliminate the need for that list?

Right now, we store each word in our list *as we split them*. Instead, let's just immediately populate each word in our dictionary!

# Solution

In our solution, we make three decisions:

1. **We use a class.** This allows us to tie our methods together, calling them on instances of our class instead of passing references.
2. To handle duplicate words with different cases, **we choose to make a word uppercase in our dictionary only if it is *always* uppercase in the original string**. While this is a reasonable approach, it is *imperfect* (consider proper nouns that are also lowercase words, like "Bill" and "bill").
3. **We build our own `split_words()` method** instead of using a built-in one. This allows us to pass each word to our `add_word_to_dictionary()` method *as it was split*, and to split words and eliminate punctuation in *one* iteration.

To split the words in the input string and populate a dictionary of the unique words to the number of times they occurred, we:

1. **Split words** by spaces, em dashes, and ellipses—making sure to include hyphens surrounded by characters. We also include all apostrophes (which will handle contractions nicely but will break possessives into separate words).
2. **Populate the words in our dictionary** as they are identified, checking if the word is already in our dictionary in its current case or another case.

If the input word is *uppercase* and *there's a lowercase version in the dictionary*, we increment the lowercase version's count. If the input word is *lowercase* and *there's an uppercase version in the dictionary*, we "demote" the uppercase version by adding the lowercase version and giving it the uppercase version's count.

Python 3.6

```
class WordCloudData:

    def __init__(self, input_string):
        self.words_to_counts = {}
        self.populate_words_to_counts(input_string)

    def populate_words_to_counts(self, input_string):
        # Iterates over each character in the input string, splitting
        # words and passing them to add_word_to_dictionary()
        current_word_start_index = 0
        current_word_length = 0
        for i, character in enumerate(input_string):

            # If we reached the end of the string we check if the last
            # character is a letter and add the last word to our dictionary
            if i == len(input_string) - 1:
                if character.isalpha():
                    current_word_length += 1
                if current_word_length > 0:
                    current_word = input_string[current_word_start_index:
                        current_word_start_index + current_word_length]
                    self.add_word_to_dictionary(current_word)

            # If we reach a space or emdash we know we're at the end of a word
            # so we add it to our dictionary and reset our current word
            elif character == ' ' or character == '\u2014':
                if current_word_length > 0:
                    current_word = input_string[current_word_start_index:
                        current_word_start_index + current_word_length]
                    self.add_word_to_dictionary(current_word)
                    current_word_length = 0

            # We want to make sure we split on ellipses so if we get two periods in
            # a row we add the current word to our dictionary and reset our current word
            elif character == '.':
                if i < len(input_string) - 1 and input_string[i + 1] == '.':
                    if current_word_length > 0:
                        current_word = input_string[current_word_start_index:
                            current_word_start_index + current_word_length]
                        self.add_word_to_dictionary(current_word)
```

```
                    current_word_length = 0

            # If the character is a letter or an apostrophe, we add it to our current word
            elif character.isalpha() or character == '\'':
                if current_word_length == 0:
                    current_word_start_index = i
                current_word_length += 1

            # If the character is a hyphen, we want to check if it's surrounded by letters
            # If it is, we add it to our current word
            elif character == '-':
                if i > 0 and input_string[i - 1].isalpha() and \
                        input_string[i + 1].isalpha():
                    if current_word_length == 0:
                        current_word_start_index = i
                    current_word_length += 1
                else:
                    if current_word_length > 0:
                        current_word = input_string[current_word_start_index:
                            current_word_start_index + current_word_length]
                        self.add_word_to_dictionary(current_word)
                        current_word_length = 0


    def add_word_to_dictionary(self, word):
        # If the word is already in the dictionary we increment its count
        if word in self.words_to_counts:
            self.words_to_counts[word] += 1

        # If a lowercase version is in the dictionary, we know our input word must be upper
        # but we only include uppercase words if they're always uppercase
        # so we just increment the lowercase version's count
        elif word.lower() in self.words_to_counts:
            self.words_to_counts[word.lower()] += 1

        # If an uppercase version is in the dictionary, we know our input word must be lowe
        # since we only include uppercase words if they're always uppercase, we add the
        # lowercase version and give it the uppercase version's count
        elif word.capitalize() in self.words_to_counts:
            self.words_to_counts[word] = 1
```

```
            self.words_to_counts[word] += self.words_to_counts[word.capitalize()]
            del self.words_to_counts[word.capitalize()]

        # Otherwise, the word is not in the dictionary at all, lowercase or uppercase
        # so we add it to the dictionary
        else:
            self.words_to_counts[word] = 1
```

## Complexity

Runtime and memory cost are both $O(n)$. This is the best we can do because we have to look at *every* character in the input string and we have to return a dictionary of *every* unique word. We optimized to only make *one* pass over our input and have only *one* $O(n)$ data structure.

## Bonus

1. We haven't explicitly talked about how to handle more complicated character sets. How would you make your solution work with more unicode characters? What changes need to be made to handle silly sentences like these:

   I'm singing 🎵 on a 🌂 day.

   ☹️ + ☕ = ☺️ .

2. We limited our input to letters, hyphenated words and punctuation. How would you expand your functionality to include numbers, email addresses, twitter handles, etc.?
3. How would you add functionality to identify phrases or words that belong together but aren't hyphenated? ("Fire truck" or "Interview Cake")
4. How could you improve your capitalization algorithm?
5. How would you avoid having duplicate words that are just plural or singular possessives?

## What We Learned

To handle capitalized words, there were lots of heuristics and approaches we could have used, each with their own strengths and weaknesses. Open-ended questions like this can really separate good engineers from great engineers.

Good engineers will come up with *a solution*, but great engineers will come up with *several solutions*, weigh them carefully, and choose the best solution for the given context. So as you're running practice questions, challenge yourself to keep thinking even after you have a first solution. See how many solutions you can come up with. This will grow your ability to quickly see multiple ways to solve a problem, so you can figure out the *best* solution. And use the hints and gotchas on each Interview Cake question—they're designed to help you cultivate this skill.