🍰 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 an <u>unordered map⅂</u> , 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 unordered map 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 unordered map 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 an unordered map 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 unordered map** with each word
3. **Handling words that are both uppercase and lowercase** in the input string

How would you start the first part?

If C++ had a built-in split function

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

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

C++ ▾

```cpp
vector<string> splitWords(const string& inputString)
{
    vector<string> words;

    size_t currentWordStartIndex = 0;
    size_t currentWordLength = 0;

    for (size_t i = 0; i < inputString.length(); ++i) {
        char character = inputString[i];

        if (isalpha(character)) {
            if (currentWordLength == 0) {
                currentWordStartIndex = i;
            }
            ++currentWordLength;
        }
        else {
            string currentWord = inputString.substr(currentWordStartIndex, currentWordLength);
            words.push_back(move(currentWord));
            currentWordLength = 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, C++ 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 extract the substring with our word and append it to the vector. That keeps our split function at $O(n)$ time.
>
> Another option would be to use a `stringstream`. Lots of different ways to work with strings.

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

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

C++ ▼

```cpp
void addWordToHashMap(unordered_map<string, size_t>& wordsToCounts, const string& word)
{
    // attempt to find word in the map and update count or add
    auto it = wordsToCounts.find(word);

    if (it != wordsToCounts.end()) {
        ++it->second;
    }
    else {
        wordsToCounts.insert(make_pair(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*?

Here are a few options:

1. Only make a word uppercase in our unordered map if it is *always* uppercase in the original string.
2. Make a word uppercase in our unordered map if it is *ever* uppercase in the original string.
3. Make a word uppercase in our unordered map 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 unordered map 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 unordered map 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 unordered map 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 `addWordToHashMap()` function to avoid duplicate words?

Think about the different possibilities:

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

Moving forward, we can either:

1. Check for words that are in the unordered map in **both** cases *when we're done populating the unordered map.* 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 unordered map that's both uppercase and lowercase.* As we add "Vanilla"s and "vanilla"s, we'd *always only ever have one version* in our unordered map.

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

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

There are three possibilities:

1. **A lowercase version is in the unordered map** (in which case we *know* our input word is uppercase, because if it is lowercase and already in the unordered map it would have passed our first check and we'd have just incremented its count)

2. **An uppercase version is in the unordered map** (so we *know* our input word is lowercase)

3. **The word is not in the unordered map** 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.

C++ ▼

```
// current hash map
// {"blue", 3}


// adding
// "Blue"


// code:
++wordsToCounts[toLowerCase(word)];


// updated hash map
// {"blue", 4}
```

What about the second possibility?

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

C++ ▾

```cpp
// current hash map
// {Yellow=6}


// adding
// "yellow"


// code (we will write our "capitalize()" method later)
string key = capitalize(word);
size_t capitalizedCount = wordsToCounts[key];
wordsToCounts.erase(key);
wordsToCounts.insert(make_pair(word, capitalizedCount + 1));


// updated hash map
// {yellow=7}
```

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

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

C++ ▾

```cpp
// current hash map
// {purple=2}


// adding
// "indigo"


// code
wordsToCounts.insert(make_pair(word, 1));


// updated hash map
// {purple=2, indigo=1}
```

Now we have all our pieces! We can split words, add them to an unordered map, 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 vector 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 vector for each word and an unordered map 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 an unordered map 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 vector. That at least doubles the memory we use! How can we eliminate the need for that vector?

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

Solution

In our solution, we make two 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 unordered map 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").

To split the words in the input string and populate an unordered map of the unique words to the number of times they occurred, we:

1. **Split words** by spaces 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 unordered map** as they are identified, checking if the word is already in our unordered map in its current case or another case.

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

C++ ▾

```cpp
class WordCloudData
{
private:
    unordered_map<string, size_t> wordsToCounts_;


    void populateWordsToCounts(const string& inputString)
    {
        // iterates over each character in the input string, splitting
        // words and passing them to addWordToHashMap()

        size_t currentWordStartIndex = 0;
        size_t currentWordLength = 0;

        for (size_t i = 0; i < inputString.length(); ++i) {
            char character = inputString[i];

            // if we reached the end of the string we check if the last
            // character is a letter and add the last word to our hash map
            if (i == inputString.length() - 1) {
                if (isalpha(character)) {
                    if (currentWordLength == 0) {
                        currentWordStartIndex = i;
                    }
                    ++currentWordLength;
                }
                if (currentWordLength > 0) {
                    string currentWord = inputString.substr(currentWordStartIndex, currentWordLength);
                    addWordToHashMap(currentWord);
                }
            }

            // if we reach a space we know we're at the end of a word
            // so we add it to our hash map and reset our current word
            else if (character == ' ') {
                if (currentWordLength > 0) {
                    string currentWord = inputString.substr(currentWordStartIndex, currentWordLength);
                    addWordToHashMap(currentWord);
                    currentWordLength = 0;
                }
            }
```

```cpp
        // 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 hash map and reset our current word
        else if (character == '.') {
            if (i < inputString.length() - 1 && inputString[i + 1] == '.') {
                if (currentWordLength > 0) {
                    string currentWord = inputString.substr(currentWordStartIndex, currentWordLer
                    addWordToHashMap(currentWord);
                    currentWordLength = 0;
                }
            }
        }


        // if the character is a letter or an apostrophe, we add it to our current word
        else if (isalpha(character) || character == '\'') {
            if (currentWordLength == 0) {
                currentWordStartIndex = i;
            }
            ++currentWordLength;
        }


        // 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
        else if (character == '-') {
            if (i > 0 && isalpha(inputString[i - 1])
                    && isalpha(inputString[i + 1])) {
                if (currentWordLength == 0) {
                    currentWordStartIndex = i;
                }
                ++currentWordLength;
            }
            else if (currentWordLength > 0) {
                string currentWord = inputString.substr(currentWordStartIndex, currentWordLength)
                addWordToHashMap(currentWord);
                currentWordLength = 0;
            }
        }
    }
}


void addWordToHashMap(const string& word)
```

```cpp
    {
        // if the word is already in the hash map we increment its count

        auto it = wordsToCounts_.find(word);

        if (it != wordsToCounts_.end()) {

            ++it->second;

        }


        // if a lowercase version is in the hash map,

        // we know our input word must be uppercase

        // but we only include uppercase words if they're always uppercase

        // so we just increment the lowercase version's count

        else if ((it = wordsToCounts_.find(toLowerCase(word))) != wordsToCounts_.end()) {

            ++it->second;

        }


        // if an uppercase version is in the hash map,

        // we know our input word must be lowercase.

        // since we only include uppercase words if they're always uppercase,

        // we add the lowercase version and give it the uppercase version's count

        else if ((it = wordsToCounts_.find(capitalize(word))) != wordsToCounts_.end()) {

            size_t newCount = it->second + 1;

            wordsToCounts_.erase(it);

            wordsToCounts_.insert(make_pair(word, newCount));

        }


        // otherwise, the word is not in the hash map at all, lowercase or uppercase

        // so we add it to the hash map

        else {

            wordsToCounts_.insert(make_pair(word, 1));

        }

    }


    static string toLowerCase(const string& word)

    {

        string result = word;

        transform(result.begin(), result.end(), result.begin(), ::tolower);

        return result;

    }


    static string capitalize(const string& word)

    {
```

```
            string result = word;

            result[0] = toupper(result[0]);

            return result;

        }


    public:

        WordCloudData (const string& inputString)

        {

            populateWordsToCounts(inputString);

        }


        const unordered_map<string, size_t> getWordsToCounts() const

        {

            return wordsToCounts_;

        }

    };
```

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 an unordered map 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 unicode input? 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.

Want more coding interview help?

Check out **interviewcake.com** for more advice, guides, and practice questions.