

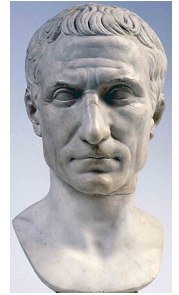
Lecture 11: Cryptography

Written by Ben Yan, from materials by Jerry Cain

Caesar Cipher

A Caesar cipher, unlike a Caesar salad or a Little Caesar's cheese pizza, may actually have been created by Roman general Julius Caesar, who was known to have used the cipher in private correspondence.

The core idea is surprisingly simple: **each letter in the plaintext**—the legible message that we're trying to disguise—is **replaced by a letter a fixed number of positions along the alphabet**. For instance, with a shift of **+3**, **"A"** would be replaced by **"D"**, **"B"** would be replaced by **"E"**, and so on, with **"W"** replaced by **"Z"**. Letters at the end of the alphabet wrap-around, e.g., **"X"** would be replaced by **"A"**, and **"Y"** by **"B"**.



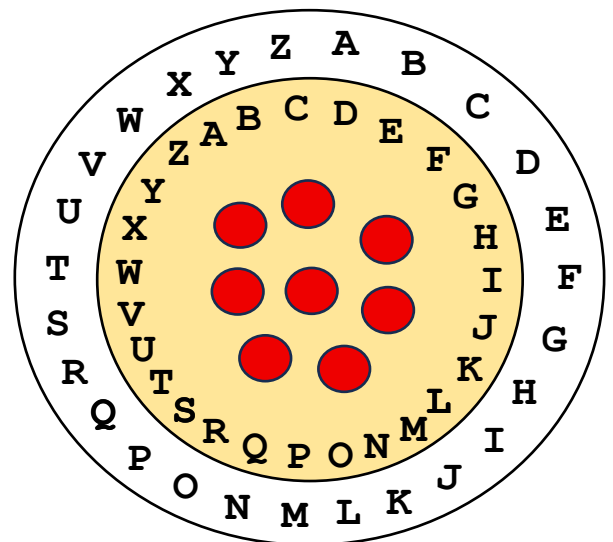
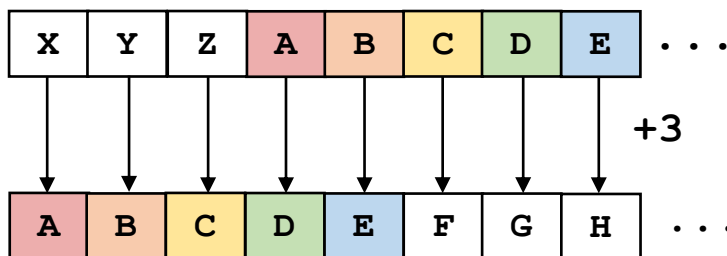
Our goal is to implement a function **caesarCipher**, which accepts a plaintext string and a shift value as input, and **returns the encrypted string that results from shifting each of the lower and uppercase letters in the input string**. Other characters, e.g., punctuation, are left as-is.

Here's a handful of examples, with famous last words from Shakespeare's *Julius Caesar* play included.

- `caesarCipher("abcd", 1)` returns `"bcde"`
- `caesarCipher("pizza", 3)` returns `"slccd"`
- `caesarCipher("Et th, Brute?", -1)` returns `"Ds sg, Aqtsd?"`
- `caesarCipher("Class of 2030", 13)` returns `"Pynff bs 2030"`



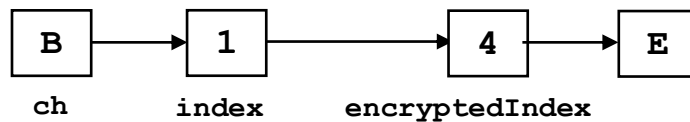
If the shift is negative, we go backward instead. For instance, if **shift = -1**, **"E"** replaced by **"D"**, and **"a"** is replaced by **"z"**. Note that a shift of **-1** is identical to a positive shift of $26 - 1 = +25$.



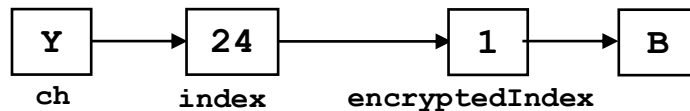
A Caesar cipher can be drawn as a **Little Caesar's pizza**, with a warm outer crust of letters, and an inner rotating circle of letters. Both run alphabetically clockwise. To encode a letter, you find it on the outer crust, and map it inward to its corresponding letter on the inner circle. The above is for a shift of **+3**.

Implement: Shift One Character

First, we'll write a helper function to **shift a single uppercase letter at a time**. To do this, we convert the character to an **index**, translate that index by the shift **key**, then convert the resulting encrypted index back into a character. Here's an example for shifting "B" by 3 positions right in the alphabet.



Here's a second example, but with wrapping around the end of the alphabet. Note that $(24 + 3) \% 26 = 1$, where the modulo operator $\%$ can be used to reset 26 to 0 ("A"), 27 to 1 ("B"), and so forth.



As we observed in the previous lecture, we compute a letter's index by taking the difference of its ASCII code with "A"'s ASCII code. Thus, "A" maps to index 0, "B" maps to 1, "C" maps to 2, and so on.

```
let index = ch.charCodeAt(0) - "A".charCodeAt(0);
```

From there, we can translate that **index** by the **key** shift, wrapping around the end if necessary.

```
let encryptedIndex = (index + key) % 26;
```

The ASCII code of the new letter is given by `"A".charCodeAt(0) + encryptedIndex`, and we use `String.fromCharCode` to retrieve the actual letter. Here's an implementation that achieves just that.

```
function shiftUpperCase(ch, key) {
  let code = ch.charCodeAt(0);
  let base = "A".charCodeAt(0);
  let index = code - base;
  let encryptedIndex = (index + key) % 26;
  return String.fromCharCode(base + encryptedIndex);
}
```

Next, we implement `shiftLowerCase`, which pantomimes the code above to work for lowercase letters instead. As we may expect, the only difference that we replace "A" with lowercase "a" 😊.

```
function shiftLowerCase(ch, key) {
  let code = ch.charCodeAt(0);
  let base = "a".charCodeAt(0); // uses lowercase "a" not uppercase!
  let index = code - base;
  let encryptedIndex = (index + key) % 26;
  ch = String.fromCharCode(base + encryptedIndex);
}
```

You could also 🧙 marry `shiftUpperCase` and `shiftLowerCase` into one function (as much code is repeated), or do `shiftUpperCase(ch.toUpperCase(), key).toLowerCase()` for a lowercase letter `ch`.

Implement: Shift Entire String

Next, we write the `caesarCipher` function itself. This encrypts a string `str` of arbitrary length by shifting each of its upper and lowercase letters by the `key`. To handle negative shifts, we adjust `key` to an equivalent positive shift between `+0` and `+25`, inclusive, using a bit of clever arithmetic.

```
if (key < 0) key = 26 - (-key % 26);
```

For instance, if `key = -1`, then `-key % 26 = 1 % 26 = 1`, and `26 - 1 = +25`, and indeed, shifting backward by `1` position is equivalent to shifting forward by `25` positions in the alphabet.

When in doubt on how some code works, trace through a small, tangible example for 🦋 intuition 🦋.

From here, the implementation of `caesarCipher` is, I dare say, delightfully straightforward. We iterate over each character, checking uppercase letters with `ch >= "A" && ch <= "Z"`, lowercase letters similarly with `ch >= "a" && ch <= "z"`, and shifting them using our tag team of helper functions.

```
function caesarCipher(str, key) {
  if (key < 0) key = 26 - (-key % 26); // ensures shift is positive

  let result = "";
  for (let i = 0; i < str.length; i++) { // shift each letter by key
    let ch = str.charAt(i);

    if (ch >= "A" && ch <= "Z")
      ch = shiftUpperCase(ch);

    } else if (ch >= "a" && ch <= "z")
      ch = shiftLowerCase(ch);
    }

    result += ch; // note: ch stays the same if not a letter!
  }
  return result;
}
```

And now, back in the Roman Empire, Caesar can disguise his last words to not give Brutus satisfaction.



Letter Substitution Cipher

A **cryptogram** is a puzzle in which a message is disguised by replacing each letter in the original letter with some other letter. Spooky right? The substitution pattern is consistent through the message, and exposing this pattern solves the cryptogram.

One of the most famous cryptograms was written by American writer Edgar Allan Poe, who just looks haunted (see right). In Poe's characteristically mysterious and macabre short story "*The Gold Bug*", a parchment of scrambled letters and symbols was said to unearth the location of an ancient, buried pirate treasure. Was the treasure found? Idk.



Poe's cryptograph is a wicked example of a **letter-substitution cipher**, in which **each letter in the original message is replaced by some different letter** in the coded version of that message. For this conspiracy of raven's ciphers, it's standard to represent the **key** as a string sequence of 26 letters, where **key[0]** is the letter that "A" maps to, **key[1]** is the letter that "B" is encrypted to, and so on.

As an example,

```
const KEY = "QWERTYUIOPASDFGHJKLZXCVBNM";
```

would correspond to a cipher with these letter-to-letter mappings, e.g., "A" -> "Q", "B" -> "W", etc.

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
Q	W	E	R	T	Y	U	I	O	P	A	S	D	F	G	H	J	K	L	Z	X	C	V	B	N	M

This is the arrangement of letters on the **QWERTY** keyboard, by the way, if you're curious ☺. Our task is to write a function **encrypt**, which accepts a string **str** and substitution pattern **key** as input, and substitutes each letter in the input string in accordance with the **key**. For instance, with the **KEY** above,

- **encrypt("ABCDEF", KEY)** returns "QWERTY"
- **encrypt("You Had Me at Hello", KEY)** returns "Ngx Iqr Dt qz Itssg". The famous quote from the 1996 rom-com *Jerry Maguire*, starring Tom Cruise and Renée Zellweger.
- **encrypt("Nevermore...Nevermore", KEY)** returns "Ftctkdgkt...Ftctkdgkt". The repeated, haunted cries of the raven in Poe's famous poem titled, no surprise, *The Raven* 🦇.

Like the Caesar cipher, we first write a helper function to **substitute a single uppercase letter**. Again, we calculate the index of the letter, and use that index to lookup the replacement letter in the **key** string.



On the next page is an implementation of **substituteUpperCase** that accomplishes exactly this.

```
function substituteUpperCase(ch, key) {
  let base = "A".charCodeAt(0);
  let code = ch.charCodeAt(0);
  let index = code - base;
  return key.charAt(index);
}
```

You won't be astonished the implementation of **substituteLowerCase** is very, very similar.

```
function substituteLowerCase(ch, key) {
  let base = "a".charCodeAt(0); // uses lowercase "a" not uppercase!
  let code = ch.charCodeAt(0);
  let index = code - base;
  return key.charAt(index);
}
```

From here, the implementation of our cryptogram-creating **encrypt** mimics that of **caesarCipher**.

```
function encrypt(str, key) {
  let result = "";
  for (let i = 0; i < str.length; i++) {
    let ch = str.charAt(i);

    if (ch >= "A" && ch <= "Z") {
      ch = substituteUpperCase(ch, key)
    } else if (ch >= "a" && ch <= "z") {
      ch = substituteLowerCase(ch, key);
    }

    result += ch;
  }
  return result;
}
```

I love you.
Ngx Egdhstzt Dt

Shut up, just shut up.
Ngx lqr Dt qz ltssg ♥

In case you're wondering how to break cryptograms, look back to no other than Poe's "*The Gold Bug*." In this story, Poe uncovers the art of **frequency analysis**: assuming the most common letters in the cryptogram likely map to the most common letters in English, i.e. **E, T, A, O, I, N, S, H, R, D, L, and U**.

For reference, here's the pirate cryptogram puzzle from the story, and a table of symbol frequencies. It may not shock you that **8** is "**E**", the most common letter in English, and the runner-up **;** is "**T**" ☺.

53†††305)) 6*;4826) 4†.) 4†) ;80
6*;48†8†60)) 85;1† (; :†*8†83(88)
5*†;46(;88*96*?;8)*†(;485);5*†
2:*†(;4956*2(5*-4)8†8*;40692
85);) 6†8) 4††;1(†9;48081;8:8†1
;48†85;4) 485†528806*81(†9;48
;(88;4(†?34;48) 4†;161;;:188;†?;

8	33	*	13	1	8	3	4
;	26	5	12	0	6	?	3
4	19	6	11	9	5	†	2
†	16	(	10	2	5	-	1
)	16	†	8	:	4	.	1