Die Standardbibliothek von C ist in der Standardbibliothek von C++ enthalten. Auf der Seite https://en.cppreference.com/w/c/header findest Du eine brauchbare Übersicht.

Aufgabe 3.1 Klassifizierung und Verarbeitung von Zeichen

Vereinfache Deine Lösungen zu den Aufgaben 1.1, 1.2 und 1.6 mit Hilfe von Funktionen aus `<ctype.h>` (siehe Folie 26).

Aufgabe 3.2 Größter gemeinsamer Teiler

Die mathematische Funktion *ggt* berechnet den größten gemeinsamen Teiler von 2 Zahlen:

ggt(x, 0) = x

ggt(x, y) = ggt(y, x % y) für y > 0

Implementiere *ggt* in einer C-Funktion mit der Signatur `int ggt(int x, int y)`.

Die rekursive mathematische Definition legt eine rekursive Implementation nahe, eine iterative Implementation (d.h. mittels Schleife statt Selbstaufruf) ist aber ebenfalls möglich.

Aufgabe 3.3 Lineare Suche

`int linear_search(int key, int array[], int len)` soll das Element `key` im Array namens `array` der Länge `len` finden. Fange dazu ganz links im Array an und untersuche ein Element nach dem anderen. Sobald Du das gesuchte Element findest, gib dessen Position zurück. Falls das Element nicht in dem Array vorkommt, gib `-1` zurück.

Hier ist eine iterative Lösung vorzuziehen, die rekursive Variante ist aber auch interessant.

Aufgabe 3.4 Binäre Suche

`int binary_search(int key, int array[], int lower, int upper)` soll `key` im Intervall [`lower`, `upper`] des **aufsteigend sortierten** `array`s finden. Untersuche dazu das Element in der Mitte von `lower` und `upper` – falls dies der Schlüssel ist, gib die entsprechende Position zurück. Wenn der Schlüssel kleiner als dieses Element ist, kannst Du Dich auf die linke Hälfte des untersuchten Intervalls beschränken, ansonsten auf die rechte Hälfte. Wenn das Intervall leer ist, existiert der Schlüssel offenbar nicht in dem Array. Dieser Fall soll durch den Rückgabewert `-1` signalisiert werden.

Auch hier kannst Du Dich wieder zwischen Iteration und Rekursion entscheiden.

Du solltest die Funktion gründlich testen. Wird das kleinste Element gefunden? Das größte? Terminiert die Suche korrekt, falls man noch kleinere (größere) Elemente sucht?

Bonusaufgabe: Binäre Suche funktioniert nur auf sortierten Arrays. Kannst Du in der Implementation von `binary_search` einfache Sicherheitsüberprüfungen einbauen, die das Programm abbrechen, sobald festgestellt wird, dass das Array gar nicht sortiert ist?

Auf keinen Fall solltest Du vor der eigentlichen Suche das *komplette* Array auf Sortierung prüfen, weil `binary_search` dann nicht mehr effizienter als lineare Suche wäre!

Aufgabe 3.5 Ganzzahlen dezimal schreiben

Schreibe eine Funktion `void print_decimal(int x)`, die eine Zahl zwischen 0 und 9 entgegennimmt und auf die Konsole schreibt, **ohne** `printf("%d")` zu verwenden. Stattdessen darfst du nur `putchar` verwenden. Du kannst aber nicht einfach nur `putchar(x)` aufrufen. Warum nicht? Hast du noch die ASCII-Tabelle im Sinn?

Erweitere die Funktion so, dass sie eine Zahl zwischen 100 und 999 rückwärts auf die Konsole schreibt. `print_decimal(123)` soll zum Beispiel 321 auf die Konsole schreiben. Dazu ist folgender mathematischer Zusammenhang interessant:

```
123 % 10 =  3     (letzte Ziffer  betrachten)
123 / 10 = 12   (führende Ziffern betrachten)
```

Hebe die Beschränkung auf 3 Ziffern auf. Alle Zahlen bis INT_MAX sollen funktionieren.

Bisher sind die Ziffern rückwärts auf die Konsole geschrieben worden. Behebe diesen Schönheitsfehler. Dazu wirst du wahrscheinlich die Ziffern in einem Array zwischenspeichern müssen. (Es gibt aber auch eine sehr elegante rekursive Lösung.)

Sorge zuletzt dafür, dass auch negative Zahlen von INT_MIN bis -1 korrekt funktionieren.

Aufgabe 3.6 T9

Das T9-System ermöglicht die Abbildung von Tastenfolgen zu Wörtern auf Handys. In der Datei `t9.c` ist ein Grundgerüst für ein solches System vorgegeben.

Implementiere zunächst die Funktion `void map_keys_to_first_word(char s[])`. Aus der Tastenfolge `"56646"` soll beispielsweise die Zeichenkette`"jmmgm"` werden, weil `j` der erste Buchstabe auf der Taste 5 ist, `m` der erste Buchstabe auf der Taste 6 usw.

Implementiere anschließend die Funktion `int generate_next_word(char s[])`. Diese soll `"jmmgm"` auf `"jmmg`**`n`**`"` abbilden, `"jmmgn"` auf `"jmmg`**`o`**`"`, `"jmmgo"` auf `"jmm`**`hm`**`"` usw. Es soll also immer der Buchstabe ganz rechts erhöht werden, und falls dieser den maximalen Wert überschreitet, wird er auf den Anfangswert zurückgesetzt und der Algorithmus wird beim linken Nachbarn fortgesetzt. (Dieses Prinzip kennst Du bereits vom Zählen in jedem beliebigen Zahlensystem.) Die Funktion soll 0 (false) zurückgeben, wenn *alle* Buchstaben auf den Anfangswert zurückgesetzt wurden, ansonsten 1 (true).

In `void print_all_words(char s[])` musst Du jetzt nur noch in einer Schleife das aktuelle Wort schreiben und `generate_next_word` aufrufen, bis alle Wörter durch sind.

Bisher werden hauptsächlich unsinnige Buchstabenfolge generiert. Sorge dafür, dass nur echte Wörter geschrieben werden. Zu diesem Zweck gibt es das Array `dictionary`. Führe einfach eine lineare Suche auf diesem Array durch, um ein Wort darin zu finden.