Aufgabe 1: Der Umdreher:

Schreibe eine Programm, das ein gegebenes Array umdreht, d. h., dass das erste Element nach der Operation hinten steht und das letzte Element steht vorne.

Aufgabe 2: Primzahlen nach Sieb-Verfahren

Schreiben Sie alle Zahlen (z.B. von 2 bis 100) in ein Array. Beginnend mit der kleinsten Zahl als der Zahl 2, diese wird als Primzahl auf dem Bildschirm ausgegeben und gleichzeitig alle Vielfachen dieser Zahl im Array auf 0 gesetzt d.h. aus der Liste gestrichen. Anschließend wird die nächste Zahl <> 0 im Array auf gleiche Weise bearbeitet.

Aufgabe 3: Bitmuster

a.) Lesen Sie eine Integerzahl ein und geben Sie deren Bitmuster aus (z.B.: 23 → 00010111).
b.) Ausgabe einer 32 Bit Integer mithilfe eines Marko ohne Schleifen (ist ein wenig tricky -☺)

Aufgabe 4: Sortieren

Sortieren Sie ein mit Zufallszahlen belegtes Array beliebiger Größe mit dem Bubble-Sort- Verfahren: jeweils zwei benachbarte Feldelemente werden vertauscht, wenn sie in der falschen Reihenfolge sind.

Aufgabe 5: Selection Sort

Schreiben Sie ein Sortierprogramm, dass nach dem Selection-Sort Algorithmus arbeitet:

1. suche das kleinste/größte Element des Arrays,
2. vertausche dieses mit dem ersten Element des Arrays,
3. gehe wieder zu Schritt 1, jetzt aber mit dem verkürzten Array.

```
Zufalsszahl:

 diese beiden Dateien includieren
 #include <stdlib.h>
 #include <time.h>

 Am Anfang v on main Zufallsgenerator initialisieren:
 ➔ srand(time(NULL));

 Zufallszahl generieren von 0 bis 100 generieren


 ➔ rand() % 100;
```

Aufgabe 6: Primfaktor Zerlegung

Schreiben Sie ein Programm zur Primzahlzerlegung so das das Einlesen der Zahl von einer Funktion *eingabe* erledigt wird und die Berechnung und Ausgabe der Primfaktoren von einer anderen Funktion *zerlegung*. Überlegen Sie zuerst, welche Variablentypen die Funktionen erhalten und zurückliefern sollen!

Ein einfaches Faktorisierungsverfahren basiert auf dem Verfahren der Probedivision. Wenn n die vorgegebene natürlich Zahl bezeichnet, dann dividiert man probeweise n der Reihe nach durch alle Zahlen von 2 aufwärts, bis die Division aufgeht. Wenn auf diese Weise ein Primfaktor gefunden wird, so wird er in einer Liste zur Verwaltung sämtlicher Primfaktoren aufgenommen. Die Ausgangszahl wird durch den Primfaktor divividiert und das Verfahren wird mit dem Quotienten analog durchgeführt. Das Verfahren endet, wenn die zu überprüfende Zahl eine Primzahl ist. Diese wird dann ebenfalls in der Liste der Primfaktoren aufgenommen.

Aufgabe 7: Treppen

Das Treppenproblem: Sie können bei einer Treppe entweder genau eine Stufe pro Schritt nehmen, oder, falls sie sportlich und durchtrainiert sind, 2 Stufen auf einmal. Damit haben sie mehrere Möglichkeiten eine mehrstufige Treppe zu überwinden

z.B.: 1 Stufe → 1 Möglichkeit, 2 Stufen → 2 Möglichkeiten, 3 Stufen → 3 Möglichkeiten, 4 Stufen → 5 Möglichkeiten, ...

Schreiben Sie ein Programm, welches Ihnen rekursiv die Anzahl der Möglichkeiten berechnet, eine Treppe T mit n Stufen zu erklimmen:

$$T_n = \begin{cases} n & \text{für} \quad n < 3 \\ T_{(n-1)} + T_{(n-2)} & \text{für} \quad n \geq 3 \end{cases}$$

Berechnen Sie dann die Anzahl der Möglichkeiten für n=25.
Version 1 rekursiv
optional Version 2, additiv, aber ist sehr komplex!!!!

Wieso und wie müssen Sie ihr Programm modifizieren, damit dieses auch die Anzahl der Möglichkeiten für n=768 (Ulmer Münster) und n=1860 (Empire State Building) berechnen kann?

Achtung: Nehmen Sie zum Testen eine kleine Anzahl von Treppen so gibt es z.B. bei 40 Treppen 165.580.141 Möglichleiten

Aufgabe 8: Base64

Schreibe eine Base64 Kodierer. Er soll einen fest im Programm vorgegebenen String nach Base64 kodieren. Sie könne die „Vorgabe_base64.c“ benutzen. In dieser müssen nur die 4 Funktionen encode_*_output() geschrieben werden. Ersetzen Sie jeweils das return 0. Falls Sie Lust haben, können Sie sich selbst überlegen wie Sie ein solches Programm schreiben würden.

Zur Kodierung werden jeweils drei Byte des Bytestroms/Array (= 24 Bit) in vier 6-Bit-Blöcke aufgeteilt. Jeder dieser 6-Bit-Blöcke bildet eine Zahl von 0 bis 63. Diese Zahlen werden anhand der nachfolgenden Umsetzungstabelle in „druckbare ASCII-Zeichen“ umgewandelt und ausgegeben. Der Name des Algorithmus erklärt sich durch ebendiesen Umstand – jedem Zeichen des kodierten Datenstroms lässt sich eine Zahl von 0 bis 63 zuordnen (siehe Tabelle). Mathematisch betrachtet gleicht dies einem Stellenwertsystem der Basis 64.

Padding: Falls die Gesamtanzahl der Eingabebytes nicht durch drei teilbar ist, wird der zu kodierende Text am Ende mit aus Nullbits bestehenden Füllbytes aufgefüllt, sodass sich eine durch drei teilbare Anzahl an Bytes ergibt. Um dem Dekodierer mitzuteilen, wie viele Füllbytes angefügt wurden, werden die 6-Bit-Blöcke, die vollständig aus Füllbytes entstanden sind, mit = kodiert. Somit können am Ende einer Base64-kodierten Datei kein, ein oder zwei =-Zeichen auftreten. Anders gesagt, es werden so viele =-Zeichen angehängt, wie Füllbytes angefügt worden sind.

Da sich die Anzahl der ursprünglichen Bytes immer eindeutig aus der Anzahl der Base64-Eingabe-Zeichen ermitteln lässt, wird in manchen Kontexten und Protokollen kein Padding verwendet (abweichend von der ursprünglichen Base64 Definition).

Die Funktion encode_first_output soll aus Byte1 Zeichen1 brechnen:

```
     Byte1       |      Byte2       |     Byte3
7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0

7 6 5 4 3 2|1 0   7 6 5 4|3 2 1 0   7 6|5 4 3 2 1 0
  Zeichen1 |   Zeichen2  |   Zeichen3  |  Zeichen4
```

das bedeutet, dass in Zeichen1 die 6 höchsten Bits von Byte1 enthält.

Beispiel:

```
Byte1: 0100 0011 „C“
Byte2: 1001 0101 „•“
Byte3: 0010 0001 „!“
Zeichen1: 010000 -> 16 = Q
Zeichen2: 111001 -> 57 = 5
Zeichen3: 010100 -> 20 = U
Zeichen4: 100001 -> 33 = h
```

Base64-Zeichensatz

Wert			Zeichen	Wert			Zeichen	Wert			Zeichen	Wert			Zeichen
dez.	binär	hex.		dez.	binär	hex.		dez.	binär	hex.		dez.	binär	hex.	
0	000000	00	A	16	010000	10	Q	32	100000	20	g	48	110000	30	w
1	000001	01	B	17	010001	11	R	33	100001	21	h	49	110001	31	x
2	000010	02	C	18	010010	12	S	34	100010	22	i	50	110010	32	y
3	000011	03	D	19	010011	13	T	35	100011	23	j	51	110011	33	z
4	000100	04	E	20	010100	14	U	36	100100	24	k	52	110100	34	0
5	000101	05	F	21	010101	15	V	37	100101	25	l	53	110101	35	1
6	000110	06	G	22	010110	16	W	38	100110	26	m	54	110110	36	2
7	000111	07	H	23	010111	17	X	39	100111	27	n	55	110111	37	3
8	001000	08	I	24	011000	18	Y	40	101000	28	o	56	111000	38	4
9	001001	09	J	25	011001	19	Z	41	101001	29	p	57	111001	39	5
10	001010	0A	K	26	011010	1A	a	42	101010	2A	q	58	111010	3A	6
11	001011	0B	L	27	011011	1B	b	43	101011	2B	r	59	111011	3B	7
12	001100	0C	M	28	011100	1C	c	44	101100	2C	s	60	111100	3C	8
13	001101	0D	N	29	011101	1D	d	45	101101	2D	t	61	111101	3D	9
14	001110	0E	O	30	011110	1E	e	46	101110	2E	u	62	111110	3E	+
15	001111	0F	P	31	011111	1F	f	47	101111	2F	v	63	111111	3F	/

Aufgabe 9: Palindrom

Schreiben Sie eine Funktion `int palin(char c[] , int erstes, int letztes)`, die rekursiv prüfen soll, ob ein gegebenes Wort oder ein Satz ein Palindrom[i] ist. Ihr wird das Wort oder der Satz in einem char-Array c sowie die Stelle des ersten und letzten Zeichens als int über- geben. Stimmen beide Zeichen überein, so kann man mit dem Prüfen des verkürzten **Wortes oder Satzes fortfahren.**

Aufgabe 10: Wurzel berechnen

Zur Berechnung der Wurzel einer Zahl gibt es ein rekursives Verfahren. Die Rekursionsformel lautet:

$$\mathrm{w}(n,x) = \begin{cases} \frac{1}{2}\left[\mathrm{w}(n-1,x) + \frac{x}{\mathrm{w}(n-1,x)}\right] & \text{für} \quad n \geq 1 \\ 1 & \text{für} \quad n = 0 \end{cases}$$

Schreiben Sie eine Funktion double wurzel(int n, double x), die rekursiv die Wurzel einer eingegebenen Zahl x berechnet. Die Zahl n gibt die Rekursionstiefe an. Vergleichen Sie das Ergebnis mit dem exakten Wert mit den Funktion `sqrt(double)` aus <math.h>.

Aufgabe 11 : Ackermann Funktion

Man berechne für kleine ganze `Zahlen m, n (m ≤ 3, n ≤ 8)` die Ackermann-Funktion `a(m, n)`:

$$a(m,n) := \begin{cases} n+1 & \text{falls} \quad m = 0 \\ a(m-1,1) & \text{falls} \quad n = 0 \\ a(m-1, a(m,n-1)) & \text{sonst} \end{cases}$$

Extra Aufgabe 12: Zahlenreihe berechnen

Erstellen Sie eine Funktion zur Berechnung der Folgenden mathematischen Reihe. Die Funktion soll so gestaltet werden , dass Sie die Anzahl (a) der Reihenelement als Parameter übergeben werde kann und folgende Ausgabe für Beispielsweise a = 10 produziert wird.
`0,1,1,2,3,5,8,13,21,34,55`

$$f(n) := \begin{cases} 0 \, f\ddot{u}r \, n = 0 \\ 1 \, f\ddot{u}r \, n = 1 \\ f(n-1) + f(n-2) f\ddot{u}r \, n > 1 \end{cases}$$

Extra Aufgabe 13: Zahlenpyramide (rekursiv) 🤯

Erstellen Sie eine Rekursive Funktion, welche eine auf dem Kopf stehende Zahlenpyramide ausgibt. Die Höhe der Pyramide soll über die Funktionsparameter gesteuert werden.
Höhe 4:

```
444444444
 3333333
  22222
   111
    0
```

Höhe 8:

```
88888888888888888
 777777777777777
  6666666666666
   55555555555
    444444444
     3333333
      22222
       111
        0
```

Extra Aufgabe 14: ggT (größter gemeinsamer Teiler) rekursiv

Erstellen Sie eine Funktion welche das ggt rekursiv berechnet:
Die Rekursionsformel für das ggt ist:

$$ggt(n\,,m) = \begin{cases} 0\ für\ m = n \\ ggt(m\,,n-m)\ für\ m < n \\ ggt(m-n\,,n) für\ m > n \end{cases}$$