Hochschule Darmstadt
Prof. Dr. Benjamin Meyer

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
fbi
FACHBEREICH INFORMATIK

3. Dezember 2020

Programmieren, Algorithmen und Datenstrukturen I, WS 2020/21
Aufgabenblatt 3

3.1 Caesar-Chiffre

Die Caesar-Chiffre ist eine der ältesten dokumentiertn Verschlüsselungsverfahren der Welt. Der Name leitet sich dabei vom römischen Feldherrn Gaius Julius Caesar ab, der diese Art der geheimen Kommunikation für seine militärische Korerspondenz verwendete. Die Grundidee dabei ist, dass jeder Buchstabe einer zu verschlüsselnden Nachricht zyklisch im Alphabet um einen bestimmten Wert (den sog. Schlüssel) verschoben wird. Bei einem Schlüssel von 3 wird beispielsweise aus einem a ein d, aus einem m ein p, und aus einem x wieder ein a. Klein- und GroSSschreibung wird erstmal ignoriert, verwenden Sie daher nur Kleinbuchstaben.

Abbildung 1: Chiffrierscheibe für Caesar-Verschlüsselung

Diese Verschlüsselung wollen wir nachprogrammieren.
Mithilfe eines C-Arrays und std::cin können Sie mehrere Buchstaben (hier bis zu neun, an Stelle 10 steht /n) auf einmal einlesen (bis zum nächsten Leerzeichen):

```
char input[10]; cin » input;
```

Initialisieren Sie das Array sinnvoll, sodass Sie merken, wie viele Buchstaben wirklich eingelesen wurden. Gehen Sie nun jeden Buchstaben durch und verschieben ihn im Alphabet (wandeln Sie ihn hierfür zeitweise in einen Integer um). Die Kleinbuchstaben liegen hierbei im Bereich 97 bis 122.
Schreiben Sie abschließend ein kleines Programm drum herum, dass den Benutzer auffordert, erst den Schlüssel einzugeben, dann die Anzahl an Wörtern, die verschlüsselt werden sollen, und dann nacheinander jeweils ein Wort. Geben Sie für jedes Wort die verschlüsselte Version wieder aus.

Können Sie diesen Text entschlüsseln (wenn ja, mit welchem Schlüssel wurde er verschlüsselt)?

```
dqmt mznwto jmq lmz sticacz
```

3.2 Sieb des Eratosthenes

Heutige, modernere Verschlüsselungsverfahren basieren häufig auf Primzahlen und dem Problem, bei (sehr) großen Zahlen überhaupt festzustellen, ob es sich um eine Primzahl handelt. Ein sehr einfaches Verfahren für die Identifikation aller Primzahlen bis zu einer festgelegten Schranke ist das Sieb des Eratosthenes, benannt nach einen im 3. Jahrhundert vor Christus lebenden Mathematiker gleichen Namens. Hierbei werden alle Zahlen von `1` bis `n` notiert und, ähnlich den Löchern eines Siebes, nach und nach weggestrichen, wenn sie durch eine andere Zahl (also außer durch 1 und durch die Zahl selber) teilbar sind. Genauer werden zuerst alle Vielfachen von zwei aus der Liste gestrichen, dann alle Vielfachen von 3. Vielfache von 4 müssen nicht mehr betrachtet werden, weil diese ja schon vorher als Vielfaches von 2 eliminiert wurden. Und so geht es weiter, bis die Wurzel von n überschritten wurde.
Ihre Aufgabe ist es nun, dieses Verfahren nachzuprogrammieren. Speichern Sie sich hierzu alle Zahlen von 1-n in einem `std::array`. Durchlaufen Sie das Array und eliminieren Sie alle Werte, die durch zwei teilbar sind (z.B. setzen Sie die Einträge auf -1) Wiederholen Sie dieses Vorgehen für die jeweils kleinste noch nicht gestrichene Zahl (nach der, die Sie als letztes getestet hatten). Also nach der 2 die 3, nach der 3 die 5 (weil die 4 schon gestrichen war). Geben Sie zum Schluss alle verbleibenden Zahlen nacheinander aus.
Überlegen Sie sich auf Papier, wie viele Berechnungsschritte Ihr Algorithmus wohl braucht. Was bedeutet das in der O-Notation?.

3.3 Sortieren und Zusammenführen

Füllen Sie zwei Vektoren mit n = 100 ganzen Zufallszahlen im Bereich 0-999. Wie das geht, haben Sie im 2. Aufgabenblatt gelernt - vergessen Sie das Setzen des Seeds nicht.
Füllen Sie nun ein Array mit den sortierten Zahlen: Gehen Sie also beide Vektoren durch und suchen die jeweils geringste Zahl. Die kleinere von beiden wird in das Array geschrieben und in dem Vektor eliminiert. (Tipp: Um ein Element eines Vektors `vec` an der Stelle `index` zu löschen, rufen Sie folgende Funktion auf: `vec.erase(vec.begin() + index);`)
Fahren Sie damit solange fort, bis alle Zahlen der Vektoren in dem Array stehen.
Überlegen Sie sich wieder auf Papier, wie viele Berechnungsschritte Ihr Algorithmus wohl braucht und was das in der O-Notation bedeutet.

3.4 Bonusaufgabe: Bubble-Sort

Verwenden Sie nun den Bubble-Sort, der aus der Vorlesung bekannt ist. Sortieren Sie hiermit die beiden Vektoren aus Aufgabe 3.3 im Vorfeld und lösen Sie die Aufgabe dann erneut - also die Zahlen sortiert in Ihr Array einfügen. Wie können Sie klug die Tatsache nutzen, dass die Vektoren schon vorsortiert waren? Verändert sich hierdurch die O-Notation und die Laufzeit des Programmes? Messen Sie die Systemzeit vor und nach der Berechnung und geben Sie die Differenz aus.