Aufgabe A1

In dieser Aufgabe geht es um vollkommene Zahlen. Definition: “Eine natürliche Zahl n wird vollkommene Zahl genannt, wenn sie gleich der Summe aller ihrer positiven Teiler außer sich selbst ist.” Beispiel: 6 = 1+2+3 ist eine vollkommene Zahl, ebenso 28 = 1+2+4+7+14.

a) Die in cpp_A1.hpp angegebene Funktion

```
unsigned int sum_factors_serial(unsigned int n)
```

berechnet die Summe aller positiven Teiler einer Zahl n (außer n selbst) und gibt diese Summe als Ergebnis zurück, so dass für vollkommene Zahlen

```
n == sum_factors_serial(n)
```

gilt. **[8 P.]**

b) Die in cpp_A1.hpp angegebene Funktion

```
unsigned int sum_factors_parallel(unsigned int n)
```

berechnet, wie in a), die Summe aller positiven Teiler einer Zahl n (außer n selbst) und gibt diese Summe als Ergebnis zurück - aber in einer parallelen, thread-sichereren und sinnvollen Variante.
Nutzen Sie mindestens zwei Threads, um die Summe zu berechnen. **[12 P.]**

Die Testfälle in `tc[]` in cpp_A1.cpp enthalten jeweils die Zahl, deren Teiler summiert werden (`n`), sowie das erwartete Ergebnis der Untersuchung (`expected`).

Bewertungsschema			
A1	**Punkte maximal**	**Punkte erreicht**	**Kommentar**
a)	**8**		
b)	**12**		
	20		

Aufgabe A2

In dieser Aufgabe geht es um (Pseudo)Zufallszahlengeneratoren (Pseudo Random Number Generator PRNG), genauer um lineare Kongruenzgeneratoren (LKG) und etwas allgemeinere, nicht notwendigerweise lineare Zufallsgeneratoren, die zufällig aussehende Zahlenfolgen, die sogenannten Pseudo-Zufallszahlen, erzeugen.

Eine Folge (x_n) eines linearen Generators ist beispielsweise durch die Vorschrift

$$x_{n+1} = (a\, x_n + b) \bmod m$$

beschrieben. Beispiel: für a=6, b=0, m=13 und dem Startwert x_0=1 ergibt sich die Folge 1,6,10,8,9,2,12,7,3,5,4,11,1,... die auch in den Testfällen als Referenzfolge betrachtet wird.

Gefragt sind hier nun zwei Klassen, die jeweils einen bestimmten Typ von Zufallsgenerator repräsentieren - einmal einen lineare Kongruenzgenerator `LKG`, der unter Angabe der Parameter a,b,m und x_0 eine nächste Zufallszahl x_{n+1} nach obiger Vorschrift erzeugt, sowie eine allgemeine Variante `PRNG`, die unter Angabe der Vorschrift selber und eines Startwertes x_0 eine nächste Zufallszahl x_{n+1} erzeugt.

Der Testcode in cpp_A2.cpp ist in Teilen auskommentiert, da die Klassen nicht implementiert sind. Kommentieren Sie ihn je nach Fortschritt ein.

Es bietet sich nun an, zunächst eine abstrakte Klasse `PRNG_Base` zu definieren, so dass beide Klassen davon erben und eine Funktion `next()` implementieren, die gerade die nächste Zufallszahl x_{n+1} berechnet.

a) Implementieren Sie eine Basisklasse `PRNG_Base`, die zunächst eine rein abstrakte Funktion
```
unsigned int next()
```
enthält. **[4 P.]**

b) Implementieren Sie eine Klasse `LKG`, die von `PRNG_Base` erbt. **[4 P.]**

c) `LKG` enthält einen Konstruktor, der `unsigned int` Werte a,b,m und x_0 übergeben bekommt. **[4 P.]**

d) `LKG` überschreibt `next()` nach obiger Vorschrift für lineare Kongruenzgeneratoren. **[4 P.]**

Wenn Sie c) und d) umgesetzt haben, können Sie in cpp_A2.cpp ein Objekt Ihrer Klasse `LKG` dynamisch erzeugen und testen lassen. Dazu brauchen Sie nur die Kommentare beim Erzeugen des Objektes (Zeile 41) und beim ersten Testcode (Zeilen 51, 52) zu entfernen und Ihr Zufallsgenerator wird gegen die genannte Beispielfolge getestet.

e) Implementieren Sie eine weitere Klasse `PRNG`, die ebenfalls von `PRNG_Base` erbt. **[4 P.]**

f) `PRNG` enthält einen Konstruktor, der eine Funktion für die Berechnungsvorschrift, z.B. die Funktion `next_random_number` in cpp_A2.cpp, sowie einen Startwert übergeben bekommt (siehe Zeile 42). **[8 P.]**

g) `PRNG` implementiert `next()` durch Anwendung der übergebenen Berechnungsvorschrift. **[4 P.]**

Wenn Sie f) und g) umgesetzt haben, können Sie in cpp_A2.cpp ein Objekt Ihrer Klasse `PRNG` dynamisch erzeugen und testen lassen. Dazu brauchen Sie nur die Kommentare beim Erzeugen des Objektes (Zeile 42) zu entfernen.

h) Erzeugen Sie nun dynamisch ein weiteres Objekt der Klasse `PRNG`, aber geben Sie diesem im Konstruktor statt eines Funktionszeigers einen äquivalenten Lambdaausdruck mit. Dies ist schon in Zeile 43 vorbereitet und erwartet von Ihnen den Lambdaausdruck. **[4 P.]**

i) Erweitern Sie Ihre Basisklasse um eine Funktion

```
unsigned int * generate_seq(unsigned int count)
```

die ebenfalls in abgeleiteten Klassen überschrieben werden könnte. Diese Funktion legt eine Anzahl (`count`) Zufallszahlen in einem dynamisch erzeugten Feld dieser Länge durch wiederholtes Aufrufen von `next()` ab und gibt das Feld zurück. **[4 P.]**

Um diese Funktionalität zu testen, brauchen Sie nur die Kommentare in den Zeilen 54-56 zu entfernen und `generate_seq` funktioniert für alle Zufallsgeneratoren.

Die Testfälle in `tc[]` in cpp_A2.cpp enthalten jeweils die Folgenglieder der Beispielfolge: Index (`no`) und Wert (`expected`).

Bewertungsschema			
A2	**Punkte maximal**	**Punkte erreicht**	**Kommentar**
a)	**4**		
b)	**4**		
c)	**4**		
d)	**4**		
e)	**4**		
f)	**8**		
g)	**4**		
h)	**4**		
i)	**4**		
	40		