

Programmierung für Naturwissenschaften 2
Sommersemester 2021
Übungen zur Vorlesung: Ausgabe am 27.05.2021

Aufgabe 7.1 (2 Punkte)

In dieser Peer-Teaching Aufgabe soll in den einzelnen Kleingruppen der zweite Teil des Abschnitts zum n-Queens Problem (siehe `C_slides.pdf`, Abschnitt 14, Frames 320-330) besprochen werden.

In jeder der Kleingruppen mit 2 Personen muss sich der oder die Studierende mit dem lexikographisch kleinsten Nachnamen anhand der oben genannten Frames auf das Thema vorbereiten und das erworbene Wissen an die anderen Mitglieder der Kleingruppe weitergeben. In jeder der Kleingruppen mit 3 Personen gilt das Entsprechende für den oder die Studierende, deren Nachname an Rang 2 der lexikographisch geordneten Nachnamen der Gruppe steht.

Falls jemand von diesen Studierenden nicht zur Übung erscheint, werden die übrigen Mitglieder der betroffenen Kleingruppe am Anfang der Übung auf die anderen Kleingruppen verteilt. Geben Sie bitte in einem solchen Fall in der Dokumentation auch die Namen der externen Mitglieder mit an, damit die Punkte zugeordnet werden können.

Diese Aufgabe soll am Anfang der Übung bearbeitet werden. Der oder die Studierende, die sich vorbereitet hat, kann ggf. bei Unklarheiten Fragen, zumindest zu den Grundbegriffen, beantworten oder Erläuterungen mündlich ergänzen.

Nach der Übung dokumentiert jede Kleingruppe in der Datei `bearbeitung.txt` das Vorgehen bei der Erarbeitung des Themas und ggf. noch bestehende Verständnisfragen oder Hinweise zu Unklarheiten in den Folien. Willkommen sind natürlich auch Bemerkungen zur Lehrform selbst.

Aufgabe 7.2 (3 Punkte) Die meisten von Ihnen kennen bereits die Technik eines Python-Generators, der mit Hilfe von `yield` in einer Funktion

1. Ergebnisse an den aufrufenden Kontext zurückliefert,
2. diesem Kontext (z.B. einer `for`-Schleife) temporär die Kontrolle übergibt, damit dieser die Ergebnisse prozessiert,
3. und sie dann wieder zurück erhält.

Hier ist ein Beispiel eines Generators für Fibonacci-Zahlen:

```
def fib_infinite():
    pp, p = 0, 1 # pp = previous of previous, p = previous
    while True:
        yield pp # iteration will receive pp as value
        current_sum = p + pp
        pp = p
        p = current_sum

for idx, f in enumerate(fib_infinite()):
    print('{}\t{}'.format(idx, f))
    if idx == args.max_idx: # args delivered by option parser
        break
```

In dieser Aufgabe geht es darum, Funktionen zum Generieren von Fibonacci-Zahlen in C zu implementieren. Dabei soll, im Gegensatz zur Lösung von Aufgabe 3.3 die Generator-Funktionalität aus Python so weit wie möglich imitiert werden. Da es in C keine `yield`-Anweisung gibt, muss man die Werte der lokalen Variablen `pp` und `p` in einer Struktur, nennen wir sie `FibGenerator`, speichern, damit sie über verschiedene Generierungsschritte hinweg zur Verfügung stehen.

Implementieren Sie in C in der Datei `fib_generator.c` einen blickdichten Datentyp `FibGenerator` und die drei Funktionen `fib_generator_new`, `fib_generator_delete` und `fib_generator_next`. Die entsprechenden Deklarationen der Prototypen und die Beschreibungen der Funktionalität dieser Funktionen finden Sie in der Datei `fib_generator.h`.

In der Datei `fib_generator_mn.c` finden Sie ein Hauptprogramm, das diese Funktionen aufruft und die ersten 94 Fibonacci-Zahlen generiert. Durch `make test` wird die Korrektheit Ihrer Implementierung getestet. Warum ist es mit dem C-Programm nicht möglich, die nächst größere Fibonacci-Zahl zu berechnen, mit dem Python Programm aber sehr wohl?

0.5 Pkt

Punkteverteilung:

- korrekte Deklaration der Struktur: 0.5 Punkte
- Implementierung der drei Funktionen: jeweils 0.5 Punkte
- bestandener Test: 0.5
- Beantwortung der Frage: 0.5 Punkte

Aufgabe 7.3 (6 Punkte)

In der fiktiven Welt Simul lebt die fiktive Bakterienart *Enpe completii*. Das Genom des Bakteriums enthält ein Gen *dolly*, das in zwei Varianten (Allelen) *dolly-0* und *dolly-1* vorkommen kann. Zur Vereinfachung der Beschreibung sprechen wir von Individuen vom Typ 0 (diese haben das Allel *dolly-0*) und Typ 1 (diese haben das Allel *dolly-1*).

Auf Simul läuft die Zeit in diskreten Zeiteinheiten. Wir sprechen daher von Generationen. In jeder Generation hat jedes *Enpe completii* Bakterium eine gewisse Wahrscheinlichkeit sich zu vermehren, d.h. sich in zwei identische Individuen des gleichen Typs zu teilen. Diese Wahrscheinlichkeit p_0 bzw. p_1 ist vom Typ abhängig (p_i für Typ $i \in \{0, 1\}$).

Da die Ressourcen auf Simul stark beschränkt sind, kann die Population nicht wachsen. Falls sich ein Individuum vermehrt, stirbt unmittelbar nach der Teilung ein zufällig ausgewähltes Individuum der Population. Dieses darf auch eines der zwei neu aus der Zellteilung entstandenen Individuen sein.

Die Populationsgröße ist also in allen Generationen konstant. Wenn das sich teilende Individuum vom Typ 0 ist und das sterbende Individuum vom Typ 1, dann erhöht sich die Anzahl der Individuen vom Typ 0 und die Anzahl der Individuen vom Typ 1 verringert sich. Das entsprechende gilt für den umgekehrten Fall.

Implementieren Sie in der Datei `simul_evolution.c` eine Funktion

```
void simulation_run(const size_t *num_individuals, const double *probab_divide,
                  size_t generations, FILE *logfp);
```

zur Simulation der Populationsgrößen entsprechend des obigen Modells und der Parameter

<code>num_individuals[0]</code>	initiale Anzahl der Individuen vom Typ 0,
<code>num_individuals[1]</code>	initiale Anzahl der Individuen vom Typ 1,
<code>prob_divide[0]</code>	Wahrscheinlichkeit, dass sich ein Individuum vom Typ 0 teilt,
<code>prob_divide[1]</code>	Wahrscheinlichkeit, dass sich ein Individuum vom Typ 1 teilt,
<code>generations</code>	maximale Anzahl der simulierten Generationen,
<code>logfp</code>	FILE-pointer zur Ausgabe in eine log-Datei mit <code>fprintf(logfp, ...)</code> .

Siehe hierzu auch die Datei `simulevolution.h`, die in der C-Datei inkludiert werden muss.

Die Simulation soll beendet werden, sobald ausschliesslich Individuen von einem der beiden Typen vorhanden sind (d.h. die Population ist fixiert auf Individuen eines Typs). In einem solchen Fall wird eine Zeile der folgenden Form ausgegeben.

```
fixed:<typ><TAB>steps:<nsteps>
```

Hier kennzeichnen die spitzen Klammern Platzhalter für den Typ (0 oder 1) und die Anzahl der simulierten Generationen bis zur Fixierung auf den angegebenen Typ. Die spitzen Klammern sind nicht Teil der Ausgabe.

Falls nach `generations` Generationen keine Fixierung auf einen Typ vorliegt, wird eine Zeile der folgenden Form ausgegeben und die Simulation beendet:

```
simulation stopped after <nsteps> steps (0:<num_0>,1:<num_1>)
```

Auch hier sind die Platzhalter in spitzen Klammern angegeben. Falls `logfp` nicht `NULL` ist, werden über `logfp` nach der Kopfzeile `step<TAB>num_0<TAB>num_1` für jede Generation die aktuellen Werte für die Anzahl der Individuen vom Typ 0 bzw. 1 ausgegeben und zwar in folgendem Format:

```
nstep<TAB>num_0<TAB>num_1
```

z.B.:

```
0<TAB>100<TAB>100
1<TAB>99<TAB>101
2<TAB>98<TAB>102 ...
```

Sei `popsiz` die Größe der Population. Am einfachsten ist es, Die Population als Array der Größe `popsiz` mit einem geeigneten Basistyp zu repräsentieren. Sie müssen in der genannten Funktion für eine Generation und für alle Individuen der Generation folgende Ereignisse simulieren:

1. Teilung eines Individuums in zwei Individuen des gleichen Typs, entsprechend der beiden vorgegebenen Wahrscheinlichkeiten p_0 und p_1 . Dazu generiert man für ein Individuum des Typs i eine Zufallszahl r im Intervall $[0, 1)$. Falls $p_i \geq r$ ist, tritt dieses Ereignis ein.
2. Falls eine Teilung erfolgt, bestimmt man durch Zufallsauswahl den Index des in diesem Schritt sterbenden Individuum unter `popsiz+1` möglichen Individuen. Die `+1` ergibt sich aus der Tatsache, dass nach der Teilung zunächst ein zusätzliches Individuum vom gleichen Typ, wie das sich teilende Individuum, entsteht, bevor ein Individuum stirbt. Eine Zufallsauswahl unter n Werten erhält man durch Multiplikation von n mit einer Zufallszahl im Intervall $[0, 1)$ und eines `type-casts` des Produktes in eine ganze Zahl. Je nach zufälliger Auswahl des sterbenden Individuums ergibt sich möglicherweise eine Veränderung der Größen der Einzelpopulationen, über die man Buch führt.

Verwenden Sie zur Erzeugung der Zufallszahlen für die Simulation die Funktion `drand48()`. Der Aufruf von `srand48()` zur Initialisierung des Zufallsgenerators erfolgt bereits im Hauptprogramm.

Zur besseren Strukturierung des Programms implementieren Sie die Simulation einer neuen Generation in einer eigenen Funktion `simulation_step`, die in `simulation_run` wiederholt aufgerufen wird. `simulation_step` benötigt als Parameter einen Verweis auf das genannte Array, die Anzahl der Individuen vom Typ 0, die gesamte Anzahl aller Individuen, sowie einen Verweis auf ein Array mit zwei `double`-Werten, das die Wahrscheinlichkeiten p_0 und p_1 enthält. Die Funktion liefert als `return`-Wert die aktualisierte Anzahl der Individuen vom Typ 0 entsprechend der Werte im genannten Array zurück. In `simulation_run` erfolgt dann die Allokation und Initialisierung des Arrays, die Erzeugung der Ausgaben entsprechend der oben beschriebenen Fälle und die Freigabe des dynamisch allokierten Speicherplatzes.

In den Materialien zu dieser Aufgabe finden Sie ein Hauptprogramm mit einem Optionsparser, ein Makefile zum Kompilieren Ihres Programms sowie ein Shell-Skript mit Testfällen. Durch `make test` verifizieren Sie die Korrektheit Ihres Programms für einige Testfälle.

Zur Bearbeitung dieser Aufgabe sollten Sie insbesondere den Abschnitt 1 aus `numerical.pdf` kennen.

Punkteverteilung:

- nachvollziehbare Implementierung von `simulation_step` entsprechend der Vorgaben: 4 Punkte
- nachvollziehbare Implementierung von `simulation_run` entsprechend der Vorgaben: 1 Punkt
- bestandene Tests: 1 Punkt

Bitte die Lösungen zu diesen Aufgaben bis zum 01.06.2021 um 18:00 Uhr an `pfn2@zbh.uni-hamburg.de` schicken.