Digitale Systeme SS2016

Aufgabe zum C-Programmierpraktikum

---

1 Organisatorisches

Implementieren Sie in Einzelarbeit ein Programm in der Programmiersprache C, das die folgende Problemstellung löst. Bitte wenden Sie sich bei Fragen zur Aufgabenstellung an den Betreuer des C-Programmierpraktikums, *Daniel Cagara*[1].

Jede abgegebene Lösung wird in einem kurzen Einzelgespräch besprochen, in dem Sie ihre Lösung am Quellcode vorstellen. Weitere Informationen bezüglich der Termine für die Einzelgespräche werden auf der Veranstaltungswebsite[2] bekanntgegeben.

Die Aufgabe muss in der Endfassung bis zum **01.08.2016** abgegeben werden. Wir empfehlen Ihnen eine erste Abgabe mindestens zwei Wochen vor dem endgültigen Termin, also bis zum 18.07.2016, so dass wir Ihnen frühzeitig Rückmeldung geben und Sie bis zum endgültigen Abgabetermin ihr Programm ggf. noch einmal nachbessern können. Nach dem 18.07.2016 kann ein Feedback nicht mehr garantiert werden. Sie dürfen gerne auch jederzeit früher eine Version abgeben und sich Feedback holen.

Für das erfolgreiche Bestehen der C-Programmieraufgabe müssen Sie zwei Kriterien erfüllen:

1. Ihr Programm muss funktionieren. Wir überprüfen dies anhand einiger Testdatensätze und automatisierter Tests, beispielsweise auf Speicherlecks.
2. Sie müssen uns im Einzelgespräch Ihr Programm und Ihren Algorithmus erläutern und begründen können.

Ihre C-Quellcode-Datei senden Sie bitte mit Ihrem Namen sowie Ihrer Matrikelnummer an die Adresse **cagara@psychologie.hu-berlin.de**.

2 Aufgabenstellung

Die bekannte Professorin für Archäologie und freiberufliche Abenteurerin *Windy-Hannah Scones* ist einmal mehr auf der Suche nach einem besonders alten Inka-Artefakt, bekannt unter dem Namen *Klausu 'Zu Lasse-ung*. Sie steht kurz vor ihrem Ziel!

Auf seiner Suche ist sie auf einen Hinweis gestoßen, der sich auf das letzte Hindernis bezieht, welches Dr. „Windy" Scones vom Erreichen des Artefaktes trennt. Der Zugang zum Artefakt soll sich hinter einer gigantischen Halle befinden, auf deren Boden es von zahllosen Schlangen wimmelt (Windy hasst Schlangen!). Glücklicherweise sind in dieser Halle aber kleine Plattformen auf Pfählen angebracht, welche wiederum durch Hängebrücken miteinander verbunden sind. Genau in der Mitte dieser Hängebrücken

[1] `cagara@psychologie.hu-berlin.de`

[2] `https://www.informatik.hu-berlin.de/de/forschung/gebiete/ti/teaching/ss/ds/prgprj`

befindet sich jeweils ein Schlüssel. Windy braucht *alle* Schlüssel, um damit die letzte Tür zu öffnen, hinter der sich das Artefakt verbirgt. Leider sind die Hängebrücken aber mittlerweile so alt und marode, dass man, sobald man eine betreten hat, diese nur noch am anderen Ende wieder verlassen kann, bevor sie in sich zusammenbricht.

Da Sie als Informatiker/in zu Dr. Scones' Expeditionsteam gehören, sollen Sie ihr ein Programm schreiben, welches ihr verrät, wie man *alle* Schlüssel einsammeln kann, ohne eine Hängebrücke zweimal benutzen zu müssen. Windy kann sich zu Beginn auf eine beliebige Startplattform abseilen und das Einsammeln der Schlüssel auf einer beliebigen Plattform abschließen. Start- und Zielplattform sind also nicht vorgegeben.

Für diese Planungsaufgabe sind folgende Daten verfügbar:

- Die Anzahl der Plattformen. Die Plattformen sind mit Ganzzahlen, beginnend bei 0, laufend durchnummeriert.
- Eine Liste der Hängebrücken zwischen den Plattformen, wobei es zwischen zwei Plattformen immer nur maximal eine Hängebrücke gibt.

Ihr Programm soll eine mögliche Folge von Plattformen ausgeben, mit der alle Schlüssel eingesammelt werden können. Gleichzeitig soll Ihr Programm auch erkennen, wenn es keine solche Folge geben kann.

Wichtig: Beachten Sie, dass die Schlüssel auf den Hängebrücken liegen, nicht auf den Plattformen. Es geht also insbesondere *nicht* darum, jede Plattform einmal zu erreichen!

3 Ein- und Ausgabeformate

Ihr Programm soll seine Eingabe aus einer Datei lesen. Der Name der zu lesenden Datei wird als erste (und einzige) Kommandozeilenoption beim Aufruf übergeben. Die Eingabedatei ist eine Textdatei, die die *Topologie* der Halle beschreibt: In ihr steht in der ersten Zeile die Anzahl der aufgestellten Plattformen. Jede weitere Zeile beschreibt eine Hängebrücke in der Form: `von nach`. Alle Zeilen (einschließlich der letzten) sind durch einen Zeilenumbruch (`'\n'`) abgeschlossen.

Eine Eingabedatei mit dem Inhalt

```
4
0 1
1 2
2 3
3 0
```

würde also eine Topologie definieren, in der es vier Plattformen gibt, die kreisförmig durch Hängebrücken miteinander verbunden sind.

Verwenden Sie für die Ausgabe des Ergebnisses genau eine Zeile. Diese Zeile soll eine Liste von Plattformnummern enthalten, die die eingelesene Probleminstanz löst. Aufeinanderfolgende Plattformnummern sollen durch genau ein Leerzeichen getrennt sein. Die Zeile soll durch ein Newline-Zeichen abgeschlossen werden.

Können die Plattformen nicht abgearbeitet werden, so ist `-1` mit anschließendem Newline auszugeben. Ein fehlerhaftes Eingabedateiformat soll mit dem Text „`Ungueltiges Eingabeformat`“ und abschließendem Newline quittiert werden.

Unmittelbar nach der Ausgabe soll sich das Programm beenden. Es werden also zu keinem Zeitpunkt Eingaben oder sonstige Aktionen des Benutzers erwartet.

Für die obige Eingabe gibt es also acht korrekte Ausgaben:

```
0 1 2 3 0
1 2 3 0 1
2 3 0 1 2
3 0 1 2 3
3 2 1 0 3
0 3 2 1 0
1 0 3 2 1
2 1 0 3 2
```

Ihr Programm soll *eine* dieser Möglichkeiten auf die Standardausgabe (`stdout`) ausgeben.

4 Beispieldaten

Wir stellen ihnen auf der Homepage der Lehrveranstaltung einige Beispiel-Eingabedateien zum Testen Ihres Programms zur Verfügung. Auch eine „Musterlösung“ für jeden Eingabedatensatz finden Sie dort. Denken Sie beim Vergleich der Ausgabe Ihres Programms mit diesen Lösungen daran, dass mehrere zulässige Möglichkeiten existieren können.

Für die Überprüfung Ihrer Abgabe werden andere Datensätze zum Einsatz kommen.

5 Hinweise zum Algorithmus

Es ist möglich, das Problem durch folgenden *Backtracking*-Algorithmus zu lösen:

Zuerst wählt man eine beliebige Startplattform (z. B. Plattform 0). Anschließend wird eine beliebige von dieser Plattform ausgehende Hängebrücke ausgewählt. Dies setzt man für die Plattform, auf der man dann ankommt fort: Wieder wird eine von dieser ausgehende, noch intakte Brücke gewählt. Wurde diese Hängebrücke noch nicht begangen, folgt man ihr auf die nächste Plattform und markiert sie als beschritten. Hatte man sie allerdings vorher schon einmal überquert, kann die Brücke als zusammengebrochen und nicht mehr verwendbar angesehen werden.

Wird eine Plattform erreicht, von der keine intakten Hängebrücken mehr ausgehen, muss geprüft werden, ob es irgendwo anders noch intakte Hängebrücken gibt. Ist dies nicht der Fall, wurde eine korrekte Lösung gefunden, die dann nur noch ausgegeben werden muss. Andernfalls wurden die noch intakten Brücken ausgelassen, es wurde

also *kein* korrekter Pfad gefunden. In diesem Fall setzt der *Backtracking*-Algorithmus ein: Das Programm muss „die Zeit zurückspulen“ und zwar bis zu dem Zeitpunkt, an dem man das letzte Mal eine Wahl zwischen zwei oder mehr Hängebrücken hatte, und wo man noch nicht alle möglichen Wege ausprobiert hat. Dieses Mal muss dann eine neue, bislang noch nicht getestete Variante versucht werden.

Wurden alle Möglichkeiten für den gewählten Startknoten geprüft, aber keine gültige Lösung gefunden, so wird der Algorithmus für einen neuen, vorher noch nicht verwendeten Startknoten wiederholt. So werden nach und nach alle möglichen Reihenfolgen, über die Brücken zu laufen, ausprobiert.

Wurde für alle Startknoten kein einziger möglicher Weg entdeckt, so gibt es für diese Konstellation von Plattformen und Hängebrücken keine Lösung.

Diese Vorgehensweise benötigt unter Umständen viel Rechenzeit, wird aber – richtig implementiert – zu einem korrekten Ergebnis führen. Es ist möglich, diesen Ansatz rekursiv zu implementieren.

Wenn Sie keine andere Lösung finden, dann können Sie dieses Brute-Force-Backtracking-Verfahren implementieren. Es gibt allerdings auch (mindestens) eine sehr viel elegantere Lösung für das Problem, die außerdem auch etwas einfacher umzusetzen ist. Sie dürfen jedes (korrekte) Lösungsverfahren einsetzen, das zwei grundlegende Bedingungen erfüllt:

- Es soll nicht noch ineffizienter sein als das oben skizzierte Brute-Force-Verfahren.
- Sie müssen es uns bei der Vorstellung Ihres C-Programms erläutern und begründen können.

6 Wichtige Hinweise

Beachten Sie bei der Erstellung Ihres Programms **unbedingt** folgende Punkte:

- Kommentieren Sie Ihren Quelltext in angemessenem Umfang.
- Halten Sie sich strikt an das oben definierte Ausgabeformat und die Aufrufkonventionen. Wir überprüfen Ihr Programm halbautomatisch auf die Korrektheit der ausgegebenen Lösungen. *Eine falsch formatierte Ausgabe wird nicht als korrekt anerkannt!* Im Zweifelsfall fragen Sie *rechtzeitig vor Abgabe* nach!
- Reagieren Sie auf falsch formatierte Eingabedaten mit der oben angegebenen Fehlermeldung.
- Das Programm soll sich nach Ausgabe des Ergebnisses auf `stdout` sofort beenden.
- Ihr Programm muss auf der Kommandozeile ohne grafische Oberfläche lauffähig sein und darf keine interaktiven Elemente (Warten auf Benutzereingaben,...) enthalten.

- Verwenden Sie keine Bibliotheken außer der C89/C90-Standardbibliothek. (Diese umfasst die folgenden Header: `assert.h`, `errno.h`, `float.h`, `limits.h`, `locale.h`, `math.h`, `setjmp.h`, `signal.h`, `stdarg.h`, `stddef.h`, `stdio.h`, `stdlib.h`, `string.h`, `time.h`.)
- Achten Sie darauf, dass Ihr Programm auf unterschiedlichen Architekturen lauffähig ist (z. B. mit unterschiedlich langen Integer-Typen).
- Verwenden Sie eine einzige C-Quelltextdatei. Ihr Programm muss sich mit einem einzelnen GCC-5.3-Compileraufruf ohne weitere Optionen fehlerfrei übersetzen und linken lassen.
- Legen Sie Ihr Programm so aus, dass es mit beliebig großen Eingabedaten umgehen kann. Setzen Sie hierfür unbedingt dynamische Speicherverwaltung mit `malloc/free` ein. Die Verwendung von `realloc` ist nicht vorgesehen. Wir werden Ihr Programm mit *großen* Datensätzen testen! Das bedeutet, dass (beliebig) lange Ein- und Ausgabedateien auftreten werden.
- Ihr abgegebenes Programm wird automatisch auf Speicherlecks überprüft. Stellen Sie sicher, dass es keine Speicherlecks aufweist. Geben Sie jeglichen dynamisch angeforderten Speicher vor der Terminierung des Programms korrekt wieder frei. Für Ihre Suche nach Speicherlecks eignet sich das Programm *valgrind*[2]. Nützliche Parameter sind `-leak-check=full`, `-track-origins=yes` und `-show-reachable=yes`. Dies schließt nicht aus, das weitere Paramter sinnvoll sind.

[2] http://www.valgrind.org