Programmierkurs (IPK) WS20/21
Lorenz Braun
ZITI, Universität Heidelberg

Aufgabenblatt 3
Abgabedatum 12. November 2020
ipk-exercises:2021-ws-6-g08cd0a5

Aufgabenblatt 3

Allgemeine Hinweise:

- Für die Aufgaben auf diesem Übungsblatt müssen Sie am 12. November votieren.
- Für jede Aufgabe gibt es einen Votierpunkt. Der Punkt für Aufgabe 3 ist ein Bonuspunkt.
- Wenn Sie nicht zu den Fakultäten für Informatik, Mathematik oder Physik gehören, ist es möglich, dass der Login auf dem Gitlab-Server nicht funktioniert. In diesem Falle schicken Sie bitte eine Mail mit Ihrer Uni-ID an: lorenz.braun@ziti.uni-heidelberg.de
- Aufgabe 2 und 3 werden abgegeben indem Sie die Ihre Lösung in ihrem git Repository hinzufügen und mit 'git push' auf unseren Gitlab-Server übertragen.

Aufgabe 1: Versionskontrolle mit git (1 Punkt)

git ist, wie in der Vorlesung vorgestellt, ein verteiltes Versionskontrollsystem. In dieser Übung geht es darum, sich ein wenig mit dem System vertraut zu machen.

(a) Melden Sie sich bei https://edu.ziti.uni-heidelberg.de an.

(b) Konfigurieren Sie git auf Ihrem Computer und legen Sie einen SSH-Key an.
Achtung:

- Ersetzen Sie Name und Email!
- Falls Sie schon einen SSH-Key haben brauchen Sie keinen anlegen.
- Der SSH-Key kann mit einem Passwort gesichert werden, dies ist optional!

```
# git config:
git config --global user.name "Hans Vader"
git config --global user.email "vader@death.star"
# ssh key creation
# default values are fine, but may overwrite existing keys with same name
ssh-keygen -t ed25519
# display Public ssh-key
cat ~/.ssh/id_ed25519.pub
```

(c) Fügen Sie den SSH-Public-Key ihrem Account bei edu.ziti.uni-heidelberg.de hinzu.

- Rufen Sie https://edu.ziti.uni-heidelberg.de/-/profile/keys auf.
- Geben Sie den Output des 'cat' Befehls in das große Textfeld ein.
- Klicken Sie auf 'Add Key'.

(d) Legen Sie ein Projekt für Ihre Gruppe an. Besuchen Sie https://edu.ziti.uni-heidelberg.de/ws2021-ipk/exercise-template/ und folgen der Anleitung dort.
Denken Sie daran auch Ihren Tutor zum Projekt hinzuzufügen, damit dieser Ihre Abgaben bewerten kann!

(e) Kopieren Sie die Quellcodedateien der bisherigen Übungen in die Entsprechenden Ordner, fügen Sie diese zu git hinzu und erstellen einen Commit. Beim Erstellen des Commits müssen Sie eine Commit Message eingeben. Dies können Sie auf zwei verschiedene Arten machen:

- Sie schreiben einfach `git commit`. In diesem Fall öffnet sich im Terminal ein Texteditor mit einigen auskommentierten Zeilen (beginnend mit `"#"`). Schreiben Sie in die erste (leere) Zeile die Commit Message (diese kann auch mehrere Zeilen lang sein), speichern Sie die Datei und verlassen Sie den Editor. Meistens öffnet sich unter Linux der Editor `vim`, der etwas gewöhnungsbedürftig ist:

  1. Drücken Sie die Taste `"i"` (für "insert"), um den Eingabemodus zu aktivieren.
  2. Tippen Sie die commit message.
  3. Drücken Sie die Taste `"ESC"`, um den Editiermodus zu verlassen.
  4. Geben Sie folgendes ein, um die Datei zu speichern und den Editor zu schliessen: `":wq"` und drücken Sie Enter.

- Alternativ können Sie die commit message auch auf der Kommandozeile eingeben:

```
1 git commit -m "hier steht Ihre commit message"
```

  Damit ist es allerdings schwierig, mehrzeilige Commit Messages zu schreiben.

Hinweis: Kompilierte Programme sollten normalerweise nicht eingecheckt werden, da man diese ja einfach durch erneutes Kompilieren wieder aus den Quellcodedateien erzeugen kann.

(f) Pushen Sie den Commit auf den Server.

(g) Während der Arbeit in den Übungen entstehen diverse Dateien, die nicht mit unter Versionskontrolle sollen, z.B. Sicherungsdateien, `.o`-Dateien und CMake-Buildverzeichnisse. Um diese Dateien zu *ignorieren*, gibt es in git den `gitignore`-Mechanismus (siehe `git help gitignore`). Erstellen Sie in Ihrem Arbeitsverzeichnis eine Datei `.gitignore` (Achtung, diese Datei ist versteckt, zum Anzeigen verwenden Sie `ls -a`) mit Dateinamen, die nicht zu git hinzugefügt werden sollen, z.B.

```
# alle Dateien, die auf .o enden
*.o
# eine bestimmte Datei, z.B. ein Executable
statistics
# Sicherungskopien, die auf ~ enden
*~
# das Unterverzeichnis build/ (von CMake)
build/
```

Hierbei werden Zeilen, die mit `#` beginnen, wieder als Kommentare ignoriert. Fügen Sie die Datei zu git hinzu, erstellen Sie einen Commit und pushen Sie diesen.

Von nun an können Sie git verwenden, um Dateien zwischen Ihrem Laptop und dem Rechner im Computer-Pool zu synchronisieren.

(h) Testen Sie, ob alle wichtigen Dateien auf dem Server angekommen sind (am einfachsten über das Webinterface). **Danach** löschen Sie das lokale Verzeichnis mit Ihren Übungen, indem Sie in das übergeordnete Verzeichnis wechseln und den Befehl `rm -rf ipk-exercises` ausführen, wobei Sie `ipk-exercises` durch den Namen ersetzen müssen, den Sie für das Verzeichnis verwendet haben.

Im Anschluss klonen Sie das Repository wieder vom Server und überprüfen, dass alle eingecheckten Dateien weiterhin da sind.

Hinweis: Auch in Zukunft ist es sinnvoll, Ihre Dateien zumindest am Ende der Übungsstunde zu committen und auf den Server zu pushen. Dann können Sie das Repository von zu Hause erneut klonen und an den Übungen weiterarbeiten. Es ist auch eine gute Idee, regelmäßiger Commits zu machen, z.B. immer dann, wenn man eine Teilaufgabe erfolgreich gelöst hat.

Weitere Hilfe zu git finden Sie unter anderem hier:

- https://git-scm.com/doc Ausführliche Dokumentation und einige Einführungsvideos.
- https://git-scm.com/docs/everyday Eine praktische Kurzübersicht wichtiger Befehle.
- https://git-scm.com/docs/gittutorial Das offizielle Tutorial.

Ansonsten ist Google Ihr Freund...

Aufgabe 2: Positive Potenzen von ganzen Zahlen (1 Punkt)

In dieser Aufgabe sollen Sie positive Potenzen $n \in \mathbb{N}_0$ von ganzen Zahlen $q \in \mathbb{Z}$ berechnen:

$$q^n = \prod_{i=1}^{n} q = \underbrace{q \cdot q \cdots q}_{n-\text{mal}}, \qquad q^0 = 1.$$

Alle folgenden Funktionen sollen testen, ob die Eingabe gültig ist. Bei einem Fehler schreiben Sie eine Meldung nach `std::cout` und geben 0 zurück.

(a) Schreiben Sie eine Funktion `int iterative(int q, int n)`, die q^n mit Hilfe einer Schleife berechnet. Diese Funktion soll im namespace `power` liegen.

(b) Schreiben Sie eine Funktion `int recursive(int q, int n)`, die q^n berechnet, indem sie sich wiederholt selbst mit anderen Argumenten aufruft. Hier müssen Sie eine geeignete Abbruchbedingung finden, bei der die Funktion sich nicht mehr weiter selbst aufruft. Diese Funktion soll ebenfalls im namespace `power` liegen.

(c) Eine naive Implementierung obiger Funktionen muss $n-1$ Multiplikationen durchführen. Können Sie eine bessere Implementierung finden? Sie können die verbesserte Version entweder iterativ oder rekursiv implementieren, was immer Ihnen einfacher erscheint. Tipp: $q^{2n} = (q^n)^2$.

Hinweise:

- Ihr Hauptprogramm soll q und n von der Kommandozeile einlesen und das Ergebnis ausgeben.
- Achten Sie darauf, dass die Funktion `main(int argc, char** argv)` nicht innerhalb des namespace liegt, sonst funktioniert Ihr Programm nicht!

Aufgabe 3: Berechnung n-ter Wurzeln (1 Bonuspunkt)

In der vorherigen Aufgabe haben Sie Potenzen von Zahlen berechnet, wobei der Exponent n stets eine ganze Zahl war. Hier wollen wir die Aufgabenstellung quasi umdrehen: wir suchen die n-te Wurzel einer positiven Zahl $q \in \mathbb{R}^+$, definiert durch

$$q^{1/n} = a \in \mathbb{R}^+ \iff a^n = \prod_{i=1}^{n} a = q.$$

Im Gegensatz zur Potenzaufgabe nutzen wir hier Fließkommazahlen (`double`), da die so definierte Wurzel $q^{1/n}$ für die meisten Kombinationen von q und n keine ganze Zahl mehr ist. Eine Formel, mit der man diese n-te Wurzel $q^{1/n}$ näherungsweise berechnen kann, ist

$$a_{k+1} \coloneqq a_k + \frac{1}{n} \cdot \left(\frac{q}{a_k^{n-1}} - a_k \right),$$

wobei a_0 eine erste Schätzung ist, z.B. einfach $a_0 \coloneqq 1$, und die Folge von Werten $a_0, a_1, a_2, a_3, \ldots$ immer bessere Näherungen für den echten Wert von $q^{1/n}$ produziert.

Alle folgenden Funktionen sollen testen, ob die Eingabe gültig ist. Bei einem Fehler schreiben Sie eine Meldung nach `std::cout` und geben ggf. 0 zurück.

(a) Schreiben Sie eine Funktion `double root_iterative(double q, int n, int steps)`, die $q^{1/n}$ näherungsweise berechnet. Dabei ist `steps` die Anzahl an Schritten (und damit Anzahl an Näherungen), die das Programm berechnen soll.

(b) Zum Berechnen einer Iteration $a_k \to a_{k+1}$ benötigen Sie die $(n-1)$-te Potenz von a_k. Eine passende Funktion haben Sie bereits geschrieben; Sie müssen lediglich darauf achten, dass sich die Datentypen von Ein- und Ausgabe geändert haben. Sollten Sie mit der letzten Aufgabe Schwierigkeiten gehabt haben dürfen sie auch `std::pow`[1] verwenden. Vergessen Sie nicht `#include <cmath>` mit einzufügen.

(c) Schreiben Sie eine Funktion `void test_root(double q, int n, int steps)`, die die Genauigkeit Ihrer Wurzelberechnung testet. Dazu soll die Funktion wie oben beschrieben eine Näherung $\tilde{a} \approx q^{1/n}$ berechnen, die Potenz $\tilde{a}^n$ bestimmen, und dann die folgenden Werte ausgeben: q, n, `steps` und $q - \tilde{a}^n$.

Testen Sie Ihr Programm für mehrere zehnstellige Ganzzahlen q als Eingabe, mit $n \in \mathbb{N}$ einstellig. Überprüfen Sie insbesondere, dass für $n = 1$ die Eingabe reproduziert wird ($q^1 = q$), und für $n = 2$ die gewohnte Quadratwurzel berechnet wird. Die Schrittzahl `steps` kann dabei in der Größenordnung von 100 gewählt werden.

[1] https://en.cppreference.com/w/cpp/numeric/math/pow