Übungsblatt 1

Deadline: Freitag, 25.04.2025 um 09:00 Uhr (Lösungsdatei sowie GitHub-Link)

Template und Namensrichtlinie

Die Datei `UE1_Nachname_Vorname.jl` auf Moodle enthält ein Template für dieses erste Übungsblatt. Lade dieses Template herunter und verwende bitte unbedingt dieses Template, da hier die Funktionssignaturen korrekt aufgeschrieben sind.

Benenne sodann diese Datei um, indem du für `Nachname` und `Vorname` deinen echten Namen einsetzt. Der LV-Leiter würde beispielsweise folgenden Dateinamen wählen: `UE1_Obszelka_Daniel.jl`

Namensrichtlinie: Es ist wichtig, dass die Datei exakt nach der Namensrichtlinie benannt ist, da die Lösungsdateien automatisiert downgeloadet und beurteilt werden!

Füge deine Lösungscodes direkt in die Datei `UE1_Nachname_Vorname.jl` ein, indem du die Funktionen mit Code befüllst. Ziel ist es, den Code so zu schreiben, dass möglichst viele Anforderungen erfüllt sind. Löse jedes Beispiel so gut es geht, es kann auch Teilpunkte geben, wenn nicht alle Aspekte (perfekt) erfüllt sind.

Wichtig: Bitte in der Datei `UE1_Nachname_Vorname.jl` ausschließlich Funktionen definieren und keine Tests vornehmen (also keine Funktionsaufrufe einbauen)! Wenn du deinen Code testen willst (was sehr empfohlen wird!), dann kannst du eine weitere Datei schreiben (zB `UE1_Test.jl`, Dateiname beliebig) und ganz oben folgendes schreiben:

```
pfad = ...    # Pfad zu jenem Ordner, in dem UE1_Nachname_Vorname.jl liegt
include(pfad * "/" * "UE1_Nachname_Vorname.jl")
```

Dieser Befehl lädt die Datei `UE1_Nachname_Vorname.jl` und macht die dort geschriebenen Funktionen verfügbar (entspricht im Prinzip der R-Funktion `source()`). Der LV-Leiter würde beispielsweise folgendes in seine Testdatei schreiben:

```
pfad = "C:/Users/Daniel/DanielGit/Lehre/Lehre_Univie/SOLV_Julia/Uebung"
include(pfad * "/" * "UE1_Obszelka_Daniel.jl")
```

Abgabe der Lösungsdatei

Die Übungen werden via GitHub abgegeben und frühestens zur Deadline von der LV-Leitung downgeloadet. Erzeuge zunächst auf deinem Computer einen Ordner, in dem du alle Übungen machen willst. Dieser kann zB `Julia-Exercises` oder `Uebung` heißen (oder anders). In diesem Ordner werden alle Übungsdateien dieses Semesters gesammelt. Dort schiebst du die Datei `UE1_Nachname_Vorname.jl` hinein und bearbeitest sie dort.

Folgende Schritte sind für die Abgabe notwendig:

1. Git lokal installieren (siehe "Hausübung" von Einheit 4 bzw. Git-Skriptum S. 14f)
2. GitHub Account erstellen (siehe "Hausübung" von Einheit 5 bzw. Git-Skriptum S. 32)
3. Den Übungsordner (zB `Julia-Exercises` oder `Uebung`), in dem die Datei `UE1_Nachname_Vorname.jl` liegt, auf GitHub hochladen. Wie das geht, haben wir in Einheit 6 besprochen. Bzw. wird das im Git-Skriptum auf S. 33 und 34 erklärt.
4. Regelmäßig euren aktuellen Stand auf Git pushen (Siehe Git-Skriptum S. 35 - 45)

GitHub-Link per Mail schicken

Den GitHub-Link (siehe S. 35 im Git-Skriptum) zum GitHub-Repository mit euren Lösungen bitte bis spätestens zur Deadline an daniel.obszelka@univie.ac.at schicken:

- Betreff: GitHub-Link für die Julia-Übungen
- Inhalt: Der GitHub-Link (Bauart: https://github.com/username/repositoryname).

Der LV-Leiter würde zB folgenden Link in seiner Mail schicken:
https://github.com/Daniel-Obszelka/Julia-Exercises

Hinweis zu Einzelarbeiten und Plagiate

Dieses Übungsblatt ist einzeln zu lösen! Die Abgaben werden stichprobenartig auf Plagiate getestet. Im Plagiatsfall können alle Punkte abgezogen werden! Im Wiederholungsfall droht ein Ausschluss aus der Lehrveranstaltung.

Zum Nachlesen & Vertiefen: Einheiten 1 bis 6 sowie das Julia-Skriptum. Arbeite die Inhalte gründlich genug durch, bevor du mit dem Übungsblatt loslegst.

Fragen?

Das Forum steht euch für Fragen zur Verfügung. Bitte keine konkreten Codes im Forum veröffentlichen und Fragen präzise genug stellen, damit ich zielgerichtet darauf eingehen kann.

Viel Erfolg und Freude beim Bearbeiten von Übungsblatt 1!

Das Übungsblatt enthält 5 Beispiele; aus formattechnischen Gründen beginnt jedes Beispiel auf einer neuen Seite.

1) **Größte Elemente**. Gegeben ist ein Vektor mit Zahlen. Du darfst davon ausgehen, dass die Zahlen alle verschieden sind (das macht die Aufgabe unkomplizierter). Schreibe die Funktion

```
greatest(x::Vector{T}, k::Integer = 1) :: Vector{T} where {T <: Real}
```

welche die größten `k` Zahlen von x extrahiert.

Folgende Anforderungen soll sie erfüllen: **(4 P)**

a) Der Parameter `k` soll überprüft werden. Wenn `k <= 0` oder `k > length(x)` gilt, soll ein geeigneter Fehler geworfen werden mit einem informativen Fehlermeldungstext. (1 P)

b) Der Parameter `k` soll den Standardwert 1 haben und standardmäßig soll die Funktion einen Vektor mit dem größten Element zurückgeben. (1 P)

c) Für allgemeines gültiges `k` sollen die `k` größten Elemente zurückgegeben werden. (1 P)

d) Die Reihenfolge des Auftretens der Elemente soll erhalten bleiben. (1 P)

Der Vektor `x` darf **nicht** verändert werden, es darf also keine Seiteneffekte geben!

Beispiele:

```
julia> println(greatest([4, 0, 2, 3, 1]))
[4]
julia> println(greatest([4, 0, 2, 3, 1], 3))
[4, 2, 3]    # Jede andere Reihenfolge führt zu einem Punkt Abzug wegen d).
julia> println(greatest([4, 0, 2, 3, 1], 8))
ERROR: DomainError with k must be between 1 and length(x): ...
```

Freiwillige Zusatzfragen zum Nachdenken: Angenommen, die Zahlen wären nicht alle verschieden. Inwiefern würde das die Aufgabe erschweren? Was müsste in diesem Fall bedacht werden? Weiters: Angenommen, wir wollten einstellen können, ob wir statt den größten Elementen die k kleinsten Elemente extrahieren können. Wie könnte das umgesetzt werden?

Bitte umblättern für Beispiel 2

2) Index eines ähnlichsten Elements finden. Gegeben ist ein Vektor `x` mit beliebigen Zahlen eines Typs `T` und eine weitere Zahl `y` vom selben Typ `T`. Schreibe die Funktion

```
nearestindex(x::Vector{<:Real}, y::Real) :: Int
```

welche die folgenden Anforderungen erfüllt: **(4 P)**

a) Es soll die Stelle eines Elements von `x`, das die geringste absolute Differenz zu `y` hat, zurückgegeben werden. (2 P)

b) Wenn es mehrere zu `y` gleich ähnliche Elemente in `x` gibt, soll einer dieser Indizes zufällig ausgewählt und zurückgegeben werden. (2 P)

<u>Beispiele</u>:

```
# 7 ist das zu y = 7 eindeutig ähnlichste Element => Return 2 (Index von 7)
julia> println(nearestindex([2, 7, 5, 1, 4, 2], 7))
2
# 7 ist das zu y = 9 eindeutig ähnlichste Element => Return 2 (Index von 7)
julia> println(nearestindex([2, 7, 5, 1, 4, 2], 9))
2
# 7 und 5 sind gleich weit entfernt von y = 6
# => Wähle zufällig 2 oder 3 (Indizes von 7 und 5) aus.
julia> println(nearestindex([2, 7, 5, 1, 4, 2], 6))
3
# 2, 4 und 2 sind gleich weit entfernt von y = 3
# => Wähle zufällig 1, 5 oder 6 (Indizes von 2, 4 und 2) aus.
julia> println(nearestindex([2, 7, 5, 1, 4, 2], 3))
6
```

Freiwillige Zusatz-Forschungsfragen: Ein Kollege schlägt folgende Signatur der Funktion vor:

```
nearestindex(x::Vector{T}, y::T) :: Int where {T <: Real}
```

Was ist der Unterschied zu der Signatur der Beispielangabe? Welche Vor- und Nachteile hat die vom Kollegen vorgeschlagene Signatur?

Bitte umblättern für Beispiel 3

3) Sortieren mit Bubblesort. Ziel ist es, einen einfachen (nicht effizienten) Sortieralgorithmus umzusetzen. Führe zunächst eine Internetrecherche zu Bubblesort durch. Schreibe dann die Funktion

```
bubblesort!(x::Vector{<:Real}; rev::Bool = false) :: Nothing
```

welche die folgenden Anforderungen erfüllt: **(4 P)**

a) Im Zuge von Bubblesort ist es notwendig, Elemente in einem Array zu vertauschen. Schreibe dazu die Hilfsfunktion

```
swap!(x::Vector, i::Integer, j::Integer) :: Nothing
```

welche in einem Vektor `x` inplace die Elemente an den Stellen `i` und `j` vertauscht. (1 P)

b) Standardmäßig soll `bubblesort!()` den Vektor `x` aufsteigend sortieren. Dabei soll *inplace* sortiert werden, dh die Änderungen sollen nach außen sichtbar sein (Seiteneffekt). (2 P)

c) Wenn der Parameter `rev` auf `true` gesetzt wird, dann soll absteigend sortiert werden. (1 P)

Die Funktionen `sort()` und `sort!()` dürfen **nicht** verwendet werden!

<u>Beispiele zu `swap!()`</u>

```
julia> x = [2, 9, 8, 1, 6, 4];
julia> println(x)
[2, 9, 8, 1, 6, 4]

julia> swap!(x, 1, length(x))
julia> println(x)
[4, 9, 8, 1, 6, 2]

julia> swap!(x, 3, 4)
julia> println(x)
[4, 9, 1, 8, 6, 2]
```

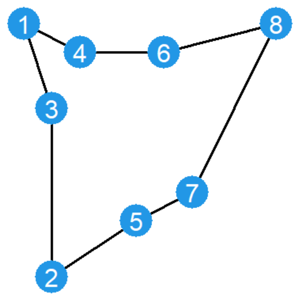
<u>Beispiele zu `bubblesort!()`</u>

```
julia> x = [2, 9, 8, 1, 6, 4];
julia> bubblesort!(x)
julia> println(x)
[1, 2, 4, 6, 8, 9]

julia> x = [2, 9, 8, 1, 6, 4];
julia> bubblesort!(x, rev = true)
julia> println(x)
[9, 8, 6, 4, 2, 1]
```

Freiwillige Zusatzfrage zum Nachdenken: Angenommen, es soll nur ein Teilbereich des Vektors (zB nur der Bereich zwischen den Positionen a und b mit `1 <= a <= b <= length(x)`) sortiert werden. Wie könnte das umgesetzt werden?

4) **Kanonische Form eines Vektors**. Hintergrund: Wir behandeln in dieser Lehrveranstaltung das TSP (Travelling Salesperson Problem). Gegeben sind n Knoten (auch Städte genannt) mit der Knotenmenge $V = \{1, \ldots, n\}$. Alle Knoten sind direkt durch Kanten verbunden. Beim TSP gilt es, eine Rundreise (auch Tour genannt) durch alle Knoten zu finden.[1] Im Folgenden ist ein Beispielproblem mit n = 8 Städten mitsamt einer möglichen Rundreise abgebildet:



Eine Rundreise wird als Permutation der Zahlen von 1 bis n dargestellt, wobei an der Stelle i der Index jener Stadt steht, die in der Rundreise an i. Stelle besucht wird. Die Stelle 1 markiert den Startknoten der Rundreise. Um den Kreis zu schließen, wird am Ende von der letzten Stadt noch zum Startknoten zurückgekehrt.

Für den Startknoten 2 gibt es im obigen Beispiel zwei mögliche Darstellungen (einmal im Uhrzeigersinn und einmal gegen den Uhrzeigersinn):

Index	1	2	3	4	5	6	7	8
Knoten	2	3	1	4	6	8	7	5

Index	1	2	3	4	5	6	7	8
Knoten	2	5	7	8	6	4	1	3

Im Prinzip kann jeder Knoten als Startknoten definiert werden. ZB lautet die Rundreise für den Startknoten 6 im Uhrzeigersinn:

Index	1	2	3	4	5	6	7	8
Knoten	6	8	7	5	2	3	1	4

Ziel dieses Beispiels ist es, eine **kanonische Form** einer gegebenen Rundreise zu bestimmen mit folgenden Eigenschaften:

1. Der Startknoten (die Stadt an Stelle 1) ist immer Stadt Nummer 1.
2. In der Rundreise der kanonischen Form werden dieselben Kanten verwendet wie in der gegebenen Rundreise.
3. Die Stadt an Stelle 2 hat eine niedrigere Nummer als die Stadt an Stelle n.

Die kanonische Form der obigen Rundreise lautet etwa:

Index	1	2	3	4	5	6	7	8
Knoten	1	3	2	5	7	8	6	4

[1] Beim TSP gilt es genauer gesagt, eine Rundreise mit minimalen Kosten zu finden. Den Kostenfaktor brauchen wir aber für dieses Beispiel nicht.

In diesem Beispiel wird die Rundreise in der kanonischen Form gegen den Uhrzeigersinn vorgenommen, weil 3 kleiner als 4 ist.

Jetzt zur Aufgabe: Sei n ∈ ℕ. Gegeben ist ein Vektor mit einer beliebigen Permutation der Zahlen von 1 bis n. Schreibe die Funktion

```
canonicaltour(x::Vector{T}) :: Vector{T} where {T <: Integer}
```

welche die **kanonische Form** von `x` bestimmt und zurückgibt.

Folgende Anforderungen soll die Funktion erfüllen: **(4 P)**

a) Falls `x` keine Permutation der Zahlen von `1` bis `length(x)` ist, soll ein geeigneter Fehler geworfen werden mit einem informativen Fehlermeldungstext. (1 P)

b) Die Startstadt ist immer die 1 (Eigenschaft 1 der kanonischen Form) und es wird eine gültige Permutation zurückgegeben. (1 P)

c) Die zurückgegebene Permutation erfüllt Eigenschaft 2 der kanonischen Form. (1 P)

d) Die zurückgegebene Permutation erfüllt Eigenschaft 3 der kanonischen Form. (1 P)

<u>Beispiele</u>:

Die folgenden Aufrufe führen allesamt zur selben kanonischen Form:

```
julia> println(canonicaltour([1, 3, 2, 5, 7, 8, 6, 4]))
julia> println(canonicaltour([1, 4, 6, 8, 7, 5, 2, 3]))
julia> println(canonicaltour([2, 3, 1, 4, 6, 8, 7, 5]))
julia> println(canonicaltour([2, 5, 7, 8, 6, 4, 1, 3]))
julia> println(canonicaltour([3, 2, 5, 7, 8, 6, 4, 1]))
julia> println(canonicaltour([4, 6, 8, 7, 5, 2, 3, 1]))
[1, 3, 2, 5, 7, 8, 6, 4]

julia> canonicaltour([3, 1, 4, 0])
ArgumentError("x must be a permutation of the numbers 1:length(x).")
```

Bitte umblättern für Beispiel 5

5) **Distanzfunktion**. Sei $n \in \mathbb{N}$. Seien $x \in \mathbb{R}^n$ und $y \in \mathbb{R}^n$ zwei gleich lange Vektoren. Die p-Distanz zwischen `x` und `y` ist definiert als:

$$\|x - y\|_p = \left(\sum_{i=1}^{n} |x_i - y_i|^p\right)^{\frac{1}{p}}$$

Schreibe die Funktion

```
distance(x::Vector{<:Real}, y::Vector{<:Real}; p::Real = 2)
```

welche die folgenden Anforderungen erfüllt: **(4 P)**

a) Längenvergleich: Die Funktion soll mit einem Fehler (inkl. einer informativen Fehlermeldung) abgebrochen werden, wenn `x` und `y` nicht dieselbe Länge haben. (1 P)

b) `p > 0` soll gelten. Wenn `p <= 0` ist, soll die Funktion mit einem geeigneten Fehler mitsamt einer informativen Fehlermeldung abgebrochen werden. (1 P)

c) Standardmäßig soll die euklidische Distanz (`p = 2`) zwischen `x` und `y` berechnet und zurückgegeben werden. (1 P)

d) Die Funktion soll für beliebiges `p > 0` die p-Distanz zwischen `x` und `y` berechnen und zurückgeben. (1 P)

<u>Beispiele</u>:

```
julia> x = [2, 6, 0]
julia> y = [-1, 4, 5]

julia> distance(x, y)          # p = 2 (Standardwert)
6.164414002968976
julia> distance(x, y, p = 1)   # Manhattan-Distanz
10.0
julia> distance([1, 2, 3], [1, 2])
ArgumentError("x and y should have the same length.")
julia> distance(x, y, p = 0)
ArgumentError("p > 0 must hold!")
```