SoSe 2020

Prof. Dr. Margarita Esponda

Objektorientierte Programmierung

3. Übungsblatt

Lernziel: Sortieralgorithmen in Python und Analyse deren Komplexität.

1. Aufgabe (4 Punkte)

a) Definieren Sie eine **random_list** Hilfsfunktion, die bei Eingabe eines Wertbereichs **(a, b)** und einer ganzen Zahl **n** eine Liste mit **n** zufälligen Zahlen zwischen **a** und **b** generiert.

b) Definieren Sie eine **sorted** Funktion, die bei Eingabe eines Vergleichsoperators und einer Liste überprüft, ob die Liste nach dem eingegebenen Vergleichsoperator (Vergleichsfunktion) sortiert ist.

Anwendungsbeispiele:

```
>>> sorted (operator.lt, [2, 2, 4, 5, 8, 9])
False
>>> sorted (operator.gt,  [2, 2, 4, 5, 8, 9])
False
>>> sorted (operator.le,  [2, 2, 4, 5, 8, 9])
True
```

2. Aufgabe (4 Punkte)

Wir haben in der Vorlesung gelernt, dass sich beim **Bubblesort**-Algorithmus die Zwischenpositionen der Zahlen oft von den richtigen Endposition entfernen.

Programmieren Sie eine Variante des **Bubblesort**-Algorithmus, die diese Verschlechterung berechnet, indem aufgezählt wird, wie groß die Anzahl der Bewegungen ist, die in die falsche Richtung laufen, und wie groß die Anzahl der richtigen Bewegungen ist und beide als Ergebnis der Funktion zurückgibt

3. Aufgabe (4 Punkte)

a) Ersetzen Sie die **for**-Schleife des Insertsort-Algorithmus der Vorlesung mit einer **while**-Schleife.

b) Definieren Sie eine Testfunktion, die mit Hilfe der Funktionen **random_list** und **sorted** den Algorithmus ausführlich testet.

a) Wann ist es sinnvoll, den **Insertsort**-Algorithmus anstatt des **Quicksort**-Algorithmus zu verwenden? Begründen Sie Ihre Antwort.

4. Aufgabe (14 Punkte)

a) Schreiben Sie eine Variante des **Quicksort**-Algorithmus aus der Vorlesung, die den Median-Wert (mittleren Wert) aus drei zufällig gewählten Elementen des zu sortierenden Teilarrays berechnet und diesen Wert als Pivot verwendet.

b) Erläutern Sie mit einem konkreten Zahlenbeispiel, warum der **Quicksort**-Algorithmus der Vorlesung nicht stabil ist.

c) Können Sie den Algorithmus aus der Vorlesung stabil machen? Erläutern Sie Ihre Antwort,

d) Analysieren Sie mit Hilfe der O-Notation den zusätzlichen Speicherverbrauch des **Quick-Sort-Algorithmus á la Haskell** aus der Vorlesung.

e) Wie oft kann während der Ausführung des **Quicksort**-Algorithmus das kleinste Element maximal bewegt werden? Begründen Sie Ihre Antwort.

f) Entwickeln Sie eine Funktion **quick_insert** als Variante des Quicksort-Algorithmus der Vorlesung, die zusätzlich zum sortierenden Array eine ganze Zahl **k** als Parameter bekommt und die Zahlen mit Hilfe des **Insertsort**-Algorithmus sortiert, wenn der noch zu sortierende Subarray kleiner **k** ist.

g) (**6 Bonuspunkte)** Definieren Sie eine Variante der **partition**-Funktion aus der Vorlesung, so dass zwei Positionen (**p**, **q**) als Ergebnis der Funktion zurückgegeben werden mit folgenden Eigenschaften:

   a) Die Elemente des Arrays am Ende der Funktionsausführung sind von Position **p** bis Position **q** des Arrays gleich.

   b) Alle Elemente vor der Position **p** sind kleiner.

   c) Alle Elemente nach der Position **q** sind größer.

   Verbessern Sie mit der neuen **partition**-Funktion den Quicksort-Algorithmus.

   Analysieren Sie die Komplexität des Algorithmus, wenn alle Zahlen gleich sind.

5. Aufgabe (3 Punkte)

Schreiben Sie eine möglichst effiziente Python-Funktion, die bei Eingabe einer Zahlenmenge beliebigen Datentyps die zwei Zahlen findet, deren Werte am nächsten sind. Analysieren Sie die Komplexität Ihres Algorithmus.

Anwendungsbeispiele:

```
>>> min_diff ([3, 10, 6, 9, 5, 1, 2, 7, 6, 8])
(6, 6)
```

6. Aufgabe (5 Punkte)

Programmieren Sie eine iterative (nicht rekursive) Variante des Mergesort-Algorithmus aus der Vorlesung, indem Sie nur ein Hilfsarray verwenden, um die Zwischen-Ergebnisse der Merge-Operationen zu speichern. Keine weiteren Listen oder Arrays sollen während des Sortieren produziert werden.

7. Aufgabe (4 Punkte)

Gegeben seien zwei Listen mit jeweils **n** ganzen Zahlen. Entwerfen Sie einen effizienten Algorithmus, der die Liste aller Elemente, die in beiden Listen enthalten sind, berechnet. Die Ausgabe sollte in sortierter Reihenfolge erfolgen. Ihr Algorithmus sollte mit einer O(n(log(n)) Komplexität ausgeführt werden.

Wichtige Hinweise:

1) Verwenden Sie geeignete Namen für Ihre Variablen und Funktionen, die den semantischen Inhalt der Variablen oder die Funktionalität der Funktionen darstellen.
2) Verwenden Sie vorgegebene Funktionsnamen, falls diese angegeben werden.
3) Kommentieren Sie Ihre Programme.
4) Verwenden Sie geeignete Hilfsvariablen und Hilfsfunktionen in Ihren Programmen.
5) Löschen Sie alle Programmzeilen und Variablen, die nicht verwendet werden.
6) Schreiben Sie getrennte Test-Funktionen für alle Aufgaben.
7) Die Lösungen sollen elektronisch (Whiteboard-Upload) abgegeben werden.