

Tutorium

Grundlagen: Algorithmen und Datenstrukturen

Übungsblatt Woche 3

Aufgabe 3.1

Zufallsvariablen

Aufgabe 3.1

Gegeben: Zahl $a \in \{0,1\}$, Zahlenfolge (x_1, x_2, \dots, x_k) mit $x_i \in \{0,1\} \quad \forall i \in \{1,2, \dots, k\}$

Gefragt: Kommt a in der Zahlenfolge vor?

Algorithmus:

```
Input: int a, int[] (x1, x2, ..., xk)
1 int i = 1
2 while i ≤ k do
3   if xi = a then
4     return Ja
5   end
6   i = i + 1
7 end
8 return Nein
```

Berechnen Sie die erwartete Anzahl an Vergleichen $x_i = a$ die dieser Algorithmus durchführt, wenn jede Eingabe (nach der gegebenen Spezifikation) mit gleicher Wahrscheinlichkeit auftritt.

Bestimmen Sie außerdem die asymptotisch erwartete Laufzeit.

Hinweis: Verwenden Sie binäre Zufallsvariablen wie in der Vorlesung.

Verwenden Sie auch, dass für eine binäre Zufallsvariable Y gilt:

$$p(Y = 1) = c \quad p(Y = 0) = 1 - c \quad c \in [0,1] \quad \text{und} \quad \mathbb{E}(Y) = \sum_{y \in \{0,1\}} y \cdot p(Y = y) = p(Y = 1) = c$$

$$\text{Sowie: } \sum_{i=0}^n c^i = \frac{c^{n+1} - 1}{c - 1}$$

Aufgabe 3.1

Aufgabe 3.2

Komplexitätsanalyse

Aufgabe 3.2

Gegeben sei eine Zahlenfolge $A[0], \dots, A[n - 1]$ wobei die Länge n ist, und eine Zahl x aus dieser Folge. Gehen Sie im Folgenden davon aus, dass die Wahrscheinlichkeit von Duplikaten in A vernachlässigbar gering ist, das heißt ihre Anzahl ist im Durchschnitt in $\mathcal{O}(1)$.

Betrachten wir den folgenden Algorithmus ***count***(x, A):

```
1  int c = 0;
2  for (int i = 0; i < n; i++)
3      if (A[i] == x) c++;
4  return c;
```

- (a) Bestimmen Sie die Komplexität von ***count*** im worst-case, best-case und average-case.
- (b) Überlegen Sie sich einen alternativen Algorithmus, der auf sortierten Folgen arbeitet, und bestimmen Sie dessen Komplexität im worst-case, best-case und average-case. Vergleichen Sie diese mit den Ergebnissen aus Aufgabenteil (a).
- (c) Lässt sich die Komplexität verbessern, indem man die Folge zunächst sortiert?

Aufgabe 3.2 (a)

Bestimmen Sie die Komplexität von ***count*** im worst-case, best-case und average-case.

Aufgabe 3.2 (b)

Überlegen Sie sich einen alternativen Algorithmus, der auf sortierten Folgen arbeitet, und bestimmen Sie dessen Komplexität im worst-case, best-case und average-case. Vergleichen Sie diese mit den Ergebnissen aus Aufgabenteil (a).

Aufgabe 3.2 (c)

Lässt sich die Komplexität verbessern, indem man die Folge zunächst sortiert?

Aufgabe 3.3

Hands-On Data Structures: Selbstorganisierende Liste

Aufgabe 3.3

(a) Implementieren Sie Methode `void add(...)`, die einen neuen Knoten erstellen soll und an das Ende der Liste anhängt. Die Laufzeit der Funktion soll in $\mathcal{O}(1)$ sein. Sie dürfen hierfür die Klasse `SelfOrganizingList` erweitern.

(b) Implementieren Sie die Methode `Optional findFirst(Predicate p)`. Für den ersten Knoten `n`, für den das Prädikat zu `true` evaluiert (`p.test(n.data)`), soll dessen Wert zurückgegeben werden (`Optional.of(n.data)`).

Außerdem soll dieser Knoten an den Anfang der Liste verschoben werden. Sollte kein Knoten dem Prädikat genügen, so soll `Optional.empty()` zurückgegeben werden und die Liste nicht verändert werden.

(c) Implementieren Sie die Methode `void removeDuplicates()`, die gleiche Elemente entfernt und nur das erste dieser Elemente in der Liste belässt. Ihre Funktion soll $\mathcal{O}(1)$ (konstante Menge) Speicher verbrauchen.

```
public class SelfOrganizingList<T> {  
    private static class Node<T> {  
        T data ;  
        Node<T> next ;  
        Node ( T d ) {  
            data = d ;  
            next = null ;  
        }  
    }  
  
    public void add (T data ) {  
  
    }  
  
    public Optional<T> findFirst  
    (Predicate<T> p) {  
  
    }  
  
    public void removeDuplicates () {  
  
    }  
}
```

Aufgabe 3.3 (a)

Implementieren Sie Methode `void add(...)`, die einen neuen Knoten erstellen soll und an das Ende der Liste anhängt. Die Laufzeit der Funktion soll in $\mathcal{O}(1)$ sein. Sie dürfen hierfür die Klasse `SelfOrganizingList` erweitern.

```
private Node<T> head;
private Node<T> end;

public void add(T data) {
    if (end == null) {
        head = end = new Node<T>(data);
        return;
    }

    end.next = new Node<T>(data);
    end = end.next;
}
```

Aufgabe 3.3 (b)

Implementieren Sie die Methode *Optional* *findFirst(Predicate p)*. Für den ersten Knoten *n*, für den das Prädikat zu *true* evaluiert (*p.test(n.data)*), soll dessen Wert zurückgegeben werden (*Optional.of(n.data)*). Außerdem soll dieser Knoten an den Anfang der Liste verschoben werden. Sollte kein Knoten dem Prädikat genügen, so soll *Optional.empty()* zurückgegeben werden und die Liste nicht verändert werden.

```
public Optional<T> findFirst(Predicate<T> p) {
    Node<T> current = head;
    Node<T> previous = null;

    while (current != null) {
        if (p.test(current.data)) {
            T result = current.data;
            if (previous != null) {

                if (current == end)
                    end = previous;
                previous.next = current.next;
                current.next = head;
                head = current;
            }
            return Optional.of(result);
        }
        previous = current;
        current = current.next;
    }
    return Optional.empty();
}
```

Aufgabe 3.3 (c)

Implementieren Sie die Methode `void removeDuplicates()`, die gleiche Elemente entfernt und nur das erste dieser Elemente in der Liste belässt.
Ihre Funktion soll $\mathcal{O}(1)$ (konstante Menge) Speicher verbrauchen.

```
public void removeDuplicates() {  
    Node<T> current = null, runner = null;  
    current = head;  
  
    while (current != null) {  
        runner = current;  
  
        while (runner.next != null) {  
            if (current.data.equals(runner.next.data))  
                runner.next = runner.next.next;  
            else  
                runner = runner.next;  
        }  
        if (current.next == null)  
            end = current;  
  
        current = current.next;  
    }  
}
```

Aufgabe 3.4

Algorithmen entwickeln und Laufzeiten bestimmen

Aufgabe 3.4

Entwerfen Sie im Folgenden zwei einfache Funktionen und erarbeiten Sie korrekten Java-Code. Bestimmen Sie dann die asymptotische Laufzeit Ihres Algorithmus.

Sie dürfen in ihren Algorithmen ausschließlich Schleifen, If-Statements, Additionen, Subtraktion sowie Multiplikation verwenden.

- a) Entwerfen Sie eine Funktion, die für zwei Integer a und b die ganzzahlige Division berechnet. Sie können davon ausgehen, dass $a > 0$ und $b > 0$ gilt.
- b) Entwerfen Sie eine Funktion, die für zwei Integer a und b den Rest berechnet, also $a \bmod b$. Sie können davon ausgehen, dass $a > 0$ und $b \neq 0$ gilt.

Aufgabe 3.4 (a)

Entwerfen Sie eine Funktion, die für zwei Integer a und b die ganzzahlige Division berechnet. Sie können davon ausgehen, dass $a > 0$ und $b > 0$ gilt.

Aufgabe 3.4 (b)

Entwerfen Sie eine Funktion, die für zwei Integer a und b den Rest berechnet, also $a \bmod b$. Sie können davon ausgehen, dass $a > 0$ und $b \neq 0$ gilt.