Übungsblatt 12

Aufgabe 12.1

Sei $n = |a|$.
Zunächst ist festzuhalten, dass $\Phi(a) = 0$ ist, genau dann wenn a sortiert ist. Dies liegt daran, dass

$$a \text{ ist sortiert } \iff (\forall i, j \in \{1, \ldots, n\} : i < j \Rightarrow a[i] \leq a[j]) \iff \{(i, j) \mid i < j, a[i] > a[j]\} = \emptyset.$$

Weiterhin gilt, dass $\Phi(a) < n^2$:
Es gibt $n \cdot (n-1)$ viele Tupel der Form $(i, j),\ i \neq j,\ i, j \in \{1, \ldots, n\}$ (n Möglichkeiten für die erste Stelle, $n-1$ für die zweite).
Davon ist bei genau der Hälfte $i > j$, da die Tupel symmetrisch gewählt wurden.
Daraus folgt:

$$\Phi(a) \leq |\{(i, j) \mid i < j\}| = \frac{n \cdot (n-1)}{2} < n^2.$$

Schließlich verringert sich der Wert der Konfliktfunktion mit jeder Iteration:
Seien i und j mit $i < j$ und $a[i] > a[j]$ die Werte, die in Zeile 1 zu Beginn einer Iteration i ausgewählt wurden, und sei $\Phi_i(a)$ die Anzahl der Konflikte zu diesem Zeitpunkt.

Im Laufe dieser Iteration wird das Tupel (i, j) aus der Menge $\{(i, j) \mid i < j,\ a[i] > a[j]\}$ entfernt, da am Ende der Iteration $a[i] < a[j]$.

Es bleibt zu zeigen, dass keine neuen Konflikte dazu gekommen sein können:
Betrachte k mit $j < k \leq n$. Für dieses k können keine neuen Konflikte entstanden sein, da sowohl für $a[i]$ als auch für $a[j]$ gilt: falls sie vor der Iteration mit $a[k]$ in Konflikt standen, tun sie dies auch noch nach der Iteration, da sowohl $i < k$ als auch $j < k$.
Genauso kann für den Fall $1 \leq k < i$ argumentiert werden.
Für den Fall $i < k < j$ können ebenfalls keine neuen Konflikte entstanden sein:

- gilt am Anfang $a[i] > a[k] > a[j]$, werden sowohl die Konflikte (i, k) als auch (k, j) gelöst.
- gilt am Anfang $a[i] < a[k] > a[j]$, bleibt der Konflikt (k, j) bestehen, doch (i, k) ist weiterhin kein Konflikt.
- gilt am Anfang $a[i] > a[k] < a[j]$, bleibt der Konflikt (i, k) bestehen, doch (k, j) ist weiterhin kein Konflikt.

Damit terminiert der Algorithmus `GreedySort` nach höchstens $\frac{n \cdot (n-1)}{2}$ Vertauschungen. $\square$

Aufgabe 12.2

a)

Die Reihenfolge der Flips ist:

$$4, 3, 2, 5, 4, 6.$$

b)

Eine Triangulation beschreibt einen kreuzungsfreien geometrischen Graphen.
Für einen solchen Graphen gilt laut Vorlesung:

$$e \leq n + \frac{2}{3}e - 2 \iff \frac{1}{3}e \leq n - 2 \iff e \leq 3n - 6,$$

wobei e der Anzahl der Kanten entspricht.
Weiterhin gilt $f \leq \frac{2}{3}e$, mit f gleich der Anzahl der Flächen.
Durch Einsetzen erhält man $f \leq 4,5n - 9$. Da alle Flächen einer Triangulation, mit Ausnahme der äußeren Fläche, Dreiecke sind, hat man im Worst Case $d = \lfloor 4,5n - 10 \rfloor$ Dreiecke.

Sei o.B.d.A. n gerade, also $d = \lfloor 4,5n - 10 \rfloor = 4,5n - 10$.
Im schlimmsten Fall steht jeder Punkt mit jedem Dreieck in Konflikt, d.h. man hat $n \cdot (4,5n - 10) = 4,5n^2 - 10n$ Konflikte. Durch jeden Flip verringert sich die Anzahl der Konflikte um mindestens 2, d.h. im Worst Case um genau 2.

Das heißt, es werden schlimmstenfalls $(4,5n^2 - 10n)/2 = 2,25n^2 - 5n \in \Omega(n^2)$ Flips benötigt.

Wie bereits in der Vorlesung gezeigt, benötigt jede Iteration (Breitensuche + Flip) Laufzeit in $\mathcal{O}(n)$. Für die Worst-Case-Analyse gehen wir daher von $\Theta(n)$ pro Iteration aus.
Das heißt, die Worst-Case-Laufzeit liegt in $\Omega(n^3)$.

Aufgabe 13.3

```
# Diese Funktion ermittelt alle Nachbarknoten eines Knotens n
def Get_Adjacent_Nodes(n):
        adjacent_nodes = []
        current_edge = n.edge
        while True:
                new_node = current_edge.next.start
                if (new_node in adjacent_nodes):
                        # Knoten bereits gefunden ->
                        # wir haben gegen den Uhrzeigersinn alle
                        # Nachbarknoten abgearbeitet und können aufhören
                        return adjacent_nodes
                adjacent_nodes += new_node
                current_edge = current_edge.twin.next

def Breitensuche(H, s):
        s.d = 0
        frontier = Queue()
        frontier.push(s)
        while (frontier.count() > 0):
                current_node = frontier.pop()
                for (new_node in (Get_Adjacent_Nodes(current_node))):
                        if (new_node.color == weiß):
                                new_node.color = grau
                                new_node.pi = current_node
                                new_node.d = current_node.d + 1
                                frontier.push(new_node)
                current_node.color = schwarz
```