Übungsblatt 5 für "Diskrete und geometrische Algorithmen"

1.) In der Vorlesung wurde der Algorithmus RANDOMIZE-IN-PLACE zum zufälligen Permutieren eines Feldes $A[1..n]$ vorgestellt. Betrachten wir nun den "leicht" abgewandelten Algorithmus, wo das Element $A[i]$ jeweils mit einem aus dem gesamten Feld zufällig gewählten Element vertauscht wird:

```
PERMUTE-WITH-ALL(A)
n := A.länge
FOR i = 1 TO n DO
  vertausche A[i] mit A[RANDOM(1, n)]
END DO
```

Erzeugt dieser Algorithmus ebenfalls eine zufällige Permutation von A? Weshalb oder weshalb nicht?

2.) Noch eine "leichte" Abwandlung des Algorithmus RANDOMIZE-IN-PLACE. Diesmal wird das Element $A[i]$ jeweils mit einem zufällig gewählten Element aus dem Teilfeld $A[i+1..n]$ vertauscht:

```
PERMUTE-WITHOUT-IDENTITY(A)
n := A.länge
FOR i = 1 TO n − 1 DO
  vertausche A[i] mit A[RANDOM(i + 1, n)]
END DO
```

Man könnte zunächst meinen, daß dadurch alle von der Identität verschiedene Permutationen zufällig erzeugt werden. Überlegen Sie sich, daß dies aber nicht der Fall ist. Können Sie herausfinden, welche Klasse von Permutationen mit diesem Algorithmus tatsächlich zufällig erzeugt wird?

3.) Nehmen Sie an, wir wollten eine zufällige Stichprobe der Größe m aus $\{1, 2, \ldots, n\}$ entnehmen, das heißt eine m-elementige Teilmenge S mit $0 \leq m \leq n$, sodaß jede m-elementige Teilmenge mit der gleichen Wahrscheinlichkeit erzeugt wird. Zeigen Sie, daß die folgende Prozedur mit nur m Aufrufen von RANDOM eine zufällige m-elementige Teilmenge S von $\{1, 2, \ldots, n\}$ berechnet, also jede m-elementige Teilmenge mit der gleichen Wahrscheinlichkeit durch die Prozedur erzeugt wird:

```
RANDOM-SAMPLE(m, n)
IF m = 0 THEN
  RETURN ∅
ELSE   S := RANDOM-SAMPLE(m − 1, n − 1)
  i := RANDOM(1, n)
  IF i ∈ S THEN
    S := S ∪ {n}
  ELSE
    S := S ∪ {i}
  END IF
  RETURN S
END IF
```

4.) Das Coupon-Sammel-Problem: Eine Person versucht eine vollständige Sammlung von n verschiedene Coupons zu erhalten. Bei jedem Kauf erhält er zufällig (also jedes Coupon mit Wahrscheinlichkeit $1/n$ und unabhängig von allen anderen Käufen) eines von n Coupons. Man berechne die erwartete Anzahl an Coupons, die diese Person erwerben muß, um eine vollständige Sammlung zu erhalten.
Hinweis: Man teile das Problem in n "Etappen" auf, d.h., man betrachte $\mathbb{E}(X_j)$, für $1 \leq j \leq n$, wobei die Zufallsvariable X_j die Anzahl an Coupons angibt, die man benötigt, um ausgehend von einer Sammlung von $j-1$ verschiedenen Coupons ein weiteres von diesen verschiedenes Coupon zu erhalten.

5.) In der Vorlesung haben wir die Datenstruktur der (Max-)Heaps kennengelernt, welche z.B. für Prioritätswarteschlangen wichtig sind. Eine Operation, die wir bislang nicht betrachtet haben, ist HEAP-DELETE(A, i), welche das Element in Knoten i aus dem Heap A löscht. Geben Sie eine Implementierung von HEAP-DELETE an, die für einen Heap mit n Elementen die Zeit $O(\log n)$ benötigt.

6.) Erzeugen eines Heaps mittels Einfügens: Wir können einen Heap bauen, indem wir wiederholt die in der Vorlesung kennengelernte Prozedur MAX-HEAP-INSERT aufrufen, um ein Element in den Heap einzufügen. Betrachten Sie die wie folgt geänderte BUILD-MAX-HEAP-Prozedur:

```
BUILD-MAX-HEAP'(A)
A.heap-size := 1
FOR i = 2 TO A.länge DO
  MAX-HEAP-INSERT(A, A[i])
END DO
```

(a) Erzeugen die Prozeduren BUILD-MAX-HEAP und BUILD-MAX-HEAP' immer den selben Heap, wenn sie auf das gleiche Eingabefeld angewendet werden? Beweisen Sie, daß sie dies tun, oder geben Sie ein Gegenbeispiel an.

(b) Zeigen Sie, daß BUILD-MAX-HEAP' im schlechtesten Fall die Laufzeit $\Theta(n \log n)$ benötigt, um einen Heap mit n Elementen zu erzeugen.

7.) Überlegen Sie sich, daß ein n-elementiger Heap die Höhe $\lfloor \log_2 n \rfloor$ hat und daß es höchstens $\lceil \frac{n}{2^{h+1}} \rceil$ Knoten der Höhe h gibt.

8.) Stapeltiefe (stack size) von Quicksort: Der Quicksort-Algorithmus, welcher in der Vorlesung vorgestellt wurde, ruft sich zweimal rekursiv selbst auf. Allerdings ist der zweite rekursive Aufruf von Quicksort nicht wirklich notwendig und man kann die Rekursion vermeiden, indem man eine iterative Kontrollstruktur verwendet (diese Methode heißt Endrekursion). Folgende Version von Quicksort verwendet diese:

```
TAIL-RECURSIVE-QUICKSORT(A, p, r)
WHILE p < r DO
  q := PARTITION(A, p, r)
  TAIL-RECURSIVE-QUICKSORT(A, p, q − 1)
  p := q + 1
END DO
```

(a) Überlegen Sie sich, daß TAIL-RECURSIVE-QUICKSORT(A,1,A.länge) das Feld a korrekt sortiert.

(b) Compiler führen rekursive Prozeduren für gewöhnlich aus, indem sie einen Stapel (= stack) verwenden, auf dem für jeden rekursiven Aufruf die zugehörigen Informationen einschließlich der Parameterwerte abgespeichert werden. Die Informationen zu dem aktuellen Aufruf befinden sich im Stapel ganz oben und die Informationen zu dem ersten Aufruf ganz unten. Wenn eine Prozedur aufgerufen wird, werden ihre Informationen auf den Stapel gelegt, wenn sie beendet wird, werden ihre Daten vom Stapel genommen. Wir nehmen an, daß die Informationen für jeden Prozeduraufruf im Stapel den Speicherplatz $O(1)$ benötigen. Die Stapeltiefe ist der maximale Umfang, den der Stapelspeicher zu irgendeinem zeitpunkt der Berechnung belegt.

Beschreiben Sie ein Szenario, bei dem die Stapeltiefe von TAIL-RECURSIVE-QUICKSORT angewendet auf ein Eingabefeld mit n Elementen $\Theta(n)$ ist.

(c) Modifizieren Sie den Algorithmus für TAIL-RECURSIVE-QUICKSORT so, daß die Stapeltiefe im schlechtesten Fall $\Theta(\log n)$ ist. Sorgen Sie dafür, daß die erwartete Laufzeit des Algorithmus in $O(n \log n)$ bleibt.