**3. Übungsaufgabe zu**
**Fortgeschrittene funktionale Programmierung**
**Thema: Algorithmenmuster: Backtracking und Prioritätsgesteuerte Suche**
**ausgegeben: Di, 28.03.2017, fällig: Di, 04.04.2017**

Für dieses Aufgabenblatt sollen Sie Haskell-Rechenvorschriften zur Lösung der im folgenden angegebenen Aufgabenstellungen entwickeln und für die Abgabe in einer Datei namens `AufgabeFFP3.hs` auf **oberstem Niveau in Ihrem Gruppenverzeichnis** ablegen. Kommentieren Sie Ihre Programme aussagekräftig und benutzen Sie, wo sinnvoll, Hilfsfunktionen und Konstanten.

Wir betrachten noch einmal das Dart-Scheibenproblem von Aufgabenblatt 2, wollen es aber hier nicht mithilfe von Generatoren, Selektoren, Filtern und Transformatoren lösen, sondern mithilfe der Algorithmenmuster für *Rücksetzsuche* (*Backtracking*) und *prioritätsgesteuerte Suche*.

Zur Erinnerung: Geworfen wird auf eine Dart-Scheibe mit $k$ verschiedenen Punktwerten. Abschnitte oder Ringe mit doppeltem oder dreifachem Punktwert gibt es nicht, auch nicht das Bull's Eye in der Mitte. Jeder Abschnitt kann in einer Wurffolge mehrfach getroffen werden. Kein Wurf geht fehl.

```
type Points      = Int      -- Punktwert einer Dart-Scheibe; echt
                            -- positive Zahl
type Dartboard   = [Points] -- Dart-Scheibe charakterisiert durch
                               Liste echt aufsteigender Punktwerte
type Turn        = [Points] -- Punktwerte einer Wurffolge; nur
                            -- Punktwerte, die auf der Scheibe vorkommen
                            -- (d.h., im Dartboard-Wert vorkommen) sind
                            -- moeglich, auch mehrfach moeglich.
type Turns       = [Turn]   -- Strom von Wurffolgen
type TargetScore = Int      -- Gewuenschte Zielpunktsumme > 0
type Throws      = Int      -- Anzahl von Wuerfen einer Wurffolge > 0
```

Ist es möglich, mit einer Folge von Würfen exakt $m$ Punkte zu erzielen? Ist es möglich, mit genau $n$ Würfen exakt $m$ Punkte zu erreichen? Wieviele Würfe sind mindestens erforderlich, um exakt $m$ Punkte zu erreichen?

- Schreiben Sie zur Beantwortung dieser Fragen Haskell-Rechenvorschriften

  ```
  bt_dart_ts   :: Dartboard -> TargetScore -> Turns
  bt_dart_tst  :: Dartboard -> TargetScore -> Throws -> Turns
  bt_dart_tsml :: Dartboard -> TargetScore -> Turns
  ```

  deren Bedeutungen mit derjenigen der Rechenvorschriften `dart_ts`, `dart_tst` und `dart_tsml` von Aufgabenblatt 2 übereinstimmen, deren Implementierungen sich aber auf das *backtracking*-Funktional

```
searchDfs :: (Eq node) => (node -> [node]) -> (node -> Bool)
                                                -> node -> [node]
```

und dessen Argumentfunktionen

```
succ :: node -> [node]
goal :: node -> Bool
```

und ggf. eine weitere Funktion `sort :: Turn -> Turn` zum aufsteigenden Sortieren einer Wurffolge abstützen.

Überlegen Sie sich dazu einen hinreichend informationsreichen Datentyp

```
data Node = ...
```

informationsreich genug auch für die weiteren Aufgabenteile, machen ihn zu einer Instanz der Typklasse `Eq` und implementieren darüber drei Paare von Funktionen

```
succ_ts :: Node -> [Node]
goal_ts :: Node -> Bool

succ_tst :: Node -> [Node]
goal_tst :: Node -> Bool

succ_tsml :: Node -> [Node]
goal_tsml :: Node -> Bool
```

so dass die sich auf `searchDfs` mit je einem der drei Funktionspaare und `sort` abstützenden Aufrufe von `bt_dart_ts`, `bt_dart_tst` und `bt_dart_tsml` die gewünschte Bedeutung erhalten, d.h.:

- `bt_dart_ts` liefert die (endlich vielen) Wurffolgen mit dem angegebenen Zielpunktewert.
- `bt_dart_tst` liefert die (endlich vielen) Wurffolgen, die den angegebenen Zielpunktewert mit der angegebenen Zahl von Würfen erreichen.
- `bt_dart_tsml` liefert die (endlich vielen) Wurffolgen, die den angegebenen Zielpunktewert mit geringster Wurfzahl ergeben.

Wie auf Aufgabenblatt 2 ist die Reihenfolge der Wurffolgen in den Resultatlisten der Funktionen `bt_dart_ts`, `bt_dart_tst` und `bt_dart_tsml` beliebig, allerdings soll jede Wurffolge in einer Resultatliste aufsteigend geordnet sein. Je nach Wahl der Argumente kann die Ergebnisliste jeder Funktion auch leer sein, wenn es keine passenden Wurffolgen gibt.

*Aufrufbeispiel:*

```
db = [6,7,16,17,26,27,36,37,46,47]
bt_dart_ts db 23    ->> [[6,17],[7,16]]
bt_dart_tst db 55 4 ->> [[7,16,16,16],[6,16,16,17],[6,6,7,36],[6,6,6,37],...]
bt_dart_tsml db 100 ->> [[6,47,47],[7,46,47],[16,37,47],[17,36,47],[17,37,46],...]
bt_dart_ts db 15    ->> []
```

- Das Funktional `searchPfs` zur prioritätsgesteuerten Suche aus Kapitel 3.3 der Vorlesung ist so konzipiert, dass es alle Lösungen innerhalb des Suchraums bestimmt.

  Wandeln Sie die Implementierung von `searchPfs` zu einem Funktional

  ```
  searchPfsFst :: (Ord node) => (node -> [node]) -> (node -> Bool)
                                                    -> node -> [node]
  ```

  so ab, dass die prioritätsgesteuerte Suche nach dem Auffinden der ersten Lösung abbricht. Da es möglicherweise keine Lösung im Suchraum gibt, wird der Resulttyp `[node]` von `searchPfs` für `searchPfsFst` beibehalten, um das Ergebnis einer solchen fehlgeschlagenen Suche durch Ausgabe der leeren Liste angeben zu können.

- Schreiben Sie mithilfe von `searchPfsFst` Haskell-Rechenvorschriften

  ```
  psf_low  :: Dartboard -> Targetscore -> Turns
  psf_high :: Dartboard -> Targetscore -> Turns
  ```

  so dass `psf_low` die Wurffolge mit den niedrigstwertigen Würfen mit dem gewünschten Punktwert liefert, `psf_high` umgekehrt die Wurffolge mit den höchstwertigen Würfen mit dieser Eigenschaft. D.h., die niedrigstwertige Wurffolge enthält angefangen vom kleinsten Scheibenwert jeden Wert so oft, dass die weitere Hinzunahme dieses Werts das Erreichen des Zielwerts verhindert. Umgekehrt enthält die höchstwertige Wurffolge angefangen vom größten Scheibenwert jeden Wert so oft, dass die weitere Hinzunahme dieses Werts das Erreichen des Zielwerts verhindert. In jedem Fall sollen die Resultatwurflisten aufsteigend geordnet sein.

  Machen Sie dazu ihren Datentyp `Node` zu einer Instanz der Typklasse `Ord` und schreiben Sie zwei Paare von Argumentfunktionen

  ```
  succ_low :: Node -> [Node]
  goal_high :: Node -> Bool

  succ_low :: Node -> [Node]
  goal_high :: Node -> Bool
  ```

  für den Aufruf von `searchPfsFst` in `psf_low` und `psf_high`. Überlegen Sie sich, ob Sie für `psf_low` oder `psf_high` ohne Abstützung auf `sort` auskommen und möglicherweise einer gemeinsamen Funktion `goal` für bzw. anstelle von

`goal_low` und `goal_high`. In diesem Fall können Sie eine der beiden Funktionen durch die andere implementieren.

*Aufrufbeispiel:*

```
db = [6,7,16,17,26,27,36,37,46,47]
psf_low db 55 ->> [[6,6,6,6,6,6,6,6,7]]
psf_high db 55 ->> [[6,6,6,37]]
```