CO202: Coursework 1

Aporva Varshney and Adam Lau

Autumn Term, 2019

```
ghc -fforce-recomp -c Submission.lhs-boot Submission.lhs
```

Problem 1: Dynamic Knapsack

How the code works

Values for the recursive subcalls are stored in the table for fast lookup later on. We effectively use the knapsack function defined above in order to create the table, but with any recursive calls replaced with lookups within the table.

```
knapsack' :: forall name weight value .
  (Ix weight, Num weight, Ord value, Num value) =>
  [(name, weight, value)] -> weight -> value
knapsack' wvs c = table ! c
  where
    table :: Array weight value
    table = tabulate (0,c) mknapsack

    mknapsack :: weight -> value
    mknapsack c = maximum 0 [ v + table ! (c - w) | (_, w, v) <- wvs, w <= c ]
```

Problem 2: Knapsack Elements

In order to also produce a list of elements chosen in order to get the maximum value, we store in the table for any given weight a tuple of the maximum value and also the names already chosen. We compute the possible options at the weight given, prepending the name of the current element to the existing list of names, and take the maximum by comparing the first element (i.e. the value). The base case is a value of 0 and an empty list.

```
knapsack'' :: forall name weight value .
  (Ix weight, Num weight, Ord value, Num value) =>
  [(name, weight, value)] -> weight -> (value, [name])
knapsack'' wvs c = table ! c
  where
    table :: Array weight (value, [name])
    table = tabulate (0,c) mknapsack
```

```
mknapsack :: weight -> (value, [name])
mknapsack c = maximumBy (compare `on` fst) ((0, []):xs)
  where
    xs = [(v + (fst (table ! (c - w))), n:(snd (table ! (c - w))))
          | (n, w, v) <- wvs, w <= c]
```

Problem 3: Bounded Knapsack

To create a bounded knapsack algorithm, we take each element of the list of tuples given, and check whether the weight of the element is greater than the capacity. If it is then we carry on searching through the list. If it isn't, then we compute the values the algorithm gives with the rest of the elements at the same capacity - hence assuming that the current element does not contribute to the optimal solution - and with the rest of the elements at a reduced capacity - hence assuming that the element does contribute to the optimal solution. We choose the approach which then gives the greater weight. Since the optimal solution can be reordered to start from any element, we can simply remove the current element from later recursive calls when assuming that it is not part of the solution.

```
bknapsack
  :: (Ord weight, Num weight, Ord value, Num value)
  => [(name, weight, value)] -> weight -> (value, [name])

bknapsack [] c = (0, [])
bknapsack ((n, w, v):wvs) c
  | w > c       = (v', ns')
  | otherwise
    = if (v + v'') > v' then (v + v'', ns'' ++ [n])
      else (v', ns')
  where
    (v', ns') = bknapsack wvs c
    (v'', ns'') = bknapsack wvs (c - w)
```

Problem 4: Reasonable Indexes

Problem 5: Bounded Knapsack Revisited

```
bknapsack' :: forall name weight value .
  (Ord weight, Num weight, Ord value, Num value) =>
  [(name, weight, value)] -> Int ->
  weight -> (value, [name])
```

```
bknapsack' wvs track c
  | track == (length wvs) = (0, [])
  | w > c                 = (v', ns')
  | otherwise             = if (v + v'') > v' then (v + v'', ns'' ++ [n]) else (v', ns')
  where
    (n, w, v)   = wvs !! track
    (v', ns')   = bknapsack' wvs (track + 1) c
    (v'', ns'') = bknapsack' wvs (track + 1) (c-w)
```

Problem 6: Dynamic Bounded Knapsack

```
bknapsack'' :: forall name weight value .
  (Ord name, Ix weight, Ord weight, Num weight,
    Ord value, Num value) =>
  [(name, weight, value)] -> weight -> (value, [name])
bknapsack'' wvs c = (max, reverse ns)
  where
    (max, ns) = table ! (c, 0)
    m = length wvs

    table :: Array (weight, Int) (value, [name])
    table = tabulate ((0, 0), (c, m)) mbknapsack

    mbknapsack :: (weight, Int) -> (value, [name])
    mbknapsack (c, track)
      | track == m = (0, [])
      | w > c      = (v', ns')
      | otherwise
        = if (v + v'') > v' then (v + v'', n:ns'') else (v', ns')
        where
          (n, w, v)  = wvs !! track
          (v', ns')  = table ! (c, track + 1)
          (v'', ns'') = table ! (c - w, track + 1)
```

Problem 7: Dijkstra Dualized

In order to modify Dijkstra's algorithm so that it returns paths to the vertex, we change it so that v = source p, and then extend p with edges to the vertex instead of edges from the vertex. We also must modify extend so that it prepends the path with the new edge instead of appending to it. TODO: Add code here?

Problem 8: Heap Operations

The heap invariants for our implementation are that the root of any tree is the minimum of the elements in the left and right subtrees, and that the height of the left and right subtrees differs by at most 1.

To create a heap, we begin by defining some helpful functions. The height function returns the height of a tree in constant time, whereas the hnode smart constructor creates a new node, maintaining the height of the resulting node to be 1 greater than the maximum height of its left and right subtrees.

The fixHeap function fixes a heap so that the root of the tree is the minimum of all elements in the left and right subtrees. Given a comparator, this function assumes that the left and right subtrees are valid heaps, and finds their minimum before pushing the current root down if it is larger than the minimum element. It then recursively calls itself on the modified subtree, in case the element which has been pushed down is not the minimum of that subtree. We can push an element down at most the height of the tree. Since the heap is a balanced tree, the height is logarithmic in the number of nodes. This function therefore runs in $O(log(n))$ time.

The removeDeepest function finds the deepest left most element in the tree and removes it from the tree, returning a tuple of the modified tree and the removed element. At each stage, we descend down the tree into one of the subtrees, so similarly to fixHeap, the function runs in $O(log(n))$ time.

Finally we can begin to implement the functions for the priority queue. Inserting into the heap involves finding the subtree with the least height (choosing the left subtree if they are of equal height), requiring log n steps as the tree is assumed balanced before the insertion. We fix the heap as we recurse, so that the inserted element bubbles up to the right place in the tree. However, since the heap is sorted before the insertion, the new element can only move up to at most the root, which would require log n comparisons. Although fixHeap is itself $O(log(n))$, in this case once we swap the elements no further recursion occurs down the tree, as the replaced element is already the minimum of the subtrees - hence when inserting, fixHeap executes in constant time. Hence, insertion occurs in $O(log(n))$ time.

Creating a priority queue from a list calls insert on every element in the list, so that it takes $O(nlog(n))$ time.

Extract simply returns the root of the tree, and is therefore a constant time function. Discard deletes the root of the tree, taking $log(n)$ steps to find the deepest element, and a further $log(n)$ steps to fix the heap afterwards. Hence it has a complexity of $O(log(n))$.

```
data Heap a = Heap (a -> a -> Ordering) (Tree a)
```

```
data Tree a = Nil | Node Int (Tree a) a (Tree a)

height :: Tree a -> Int
height Nil = 0
height (Node h _ _ _) = h

hnode :: Tree a -> a -> Tree a -> Tree a
hnode l x r = Node h l x r
  where
    h = 1 + max (height l) (height r)

showTree :: (Show a) => Tree a -> String
showTree (Nil) = ""
showTree (Node _ Nil x Nil) = show x
showTree (Node _ l x r) = show x ++ "(" ++ showTree l ++ ")(" ++ showTree r ++ ")"

showHeap :: (Show a) => Heap a -> String
showHeap (Heap _ t) = showTree t

-- Fixes the heap so minimum element at the top
-- Assumes that the subtrees have minimum element at the top
fixHeap :: (a -> a -> Ordering) -> Tree a -> Tree a
fixHeap cmp Nil = Nil
fixHeap cmp t@(Node h Nil x Nil) = t
fixHeap cmp t@(Node h (Node lh ll lx lr) x Nil)
  | cmp lx x == LT = hnode l' lx Nil
  | otherwise      = t
  where
    l' = fixHeap cmp (hnode ll x lr)
fixHeap cmp t@(Node h Nil x (Node rh rl rx rr))
  | cmp rx x == LT = hnode Nil rx r'
  | otherwise      = t
  where
    r' = fixHeap cmp (Node rh rl x rr)
fixHeap cmp t@(Node h l@(Node lh ll lx lr) x r@(Node rh rl rx rr))
  | cmp lx rx == LT = if cmp lx x == LT then hnode l' lx r else t
  | cmp rx lx == LT = if cmp rx x == LT then hnode l rx r' else t
  | otherwise       = t
  where
    l' = fixHeap cmp (hnode ll x lr)
    r' = fixHeap cmp (hnode rl x rr)

removeDeepest :: Tree a -> (Tree a, a)
removeDeepest Nil = error "Can't remove deepest element: tree has no elements"
```

```
removeDeepest (Node _ Nil x Nil) = (Nil, x)
removeDeepest (Node _ l x Nil) = (hnode l' x Nil, x')
  where
    (l', x') = removeDeepest l
removeDeepest (Node _ Nil x r) = (hnode Nil x r', x')
  where
    (r', x') = removeDeepest r
removeDeepest (Node _ l x r)
  | height l >= height r = (hnode l' x r, x')
  | otherwise            = (hnode l x r', x'')
  where
    (l', x') = removeDeepest l
    (r', x'') = removeDeepest r


instance PQueue Heap where
  toPQueue cmp [] = empty cmp
  toPQueue cmp (x:xs)
    = insert x (toPQueue cmp xs)

  fromPQueue (Heap _ Nil) = []
  fromPQueue ps
    = e:(fromPQueue es)
    where
      (e, es) = detach ps

  priority :: Heap a -> (a -> a -> Ordering)
  priority (Heap cmp _) = cmp

  empty :: (a -> a -> Ordering) -> Heap a
  empty cmp = Heap cmp Nil

  isEmpty :: Heap a -> Bool
  isEmpty (Heap _ Nil) = True
  isEmpty _ = False

  insert :: a -> Heap a -> Heap a
  insert x (Heap cmp t)
    = Heap cmp (insert' t)
    where
      insert' Nil = hnode Nil x Nil
      insert' (Node h l y r)
        | height l <= height r = fixHeap cmp (hnode (insert' l) y r)
        | otherwise            = fixHeap cmp (hnode l y (insert' r))
```

```
delete :: a -> Heap a -> Heap a
delete x (Heap cmp t)
  | elemDepth == -1 = error "Element␣is␣not␣in␣the␣list"
  | elemDepth /= h  = Heap cmp (fst $ deleteAndReplace t')
  | elemDepth == h  = Heap cmp (fst $ deleteDeepestLeaf t)
  where
    h = height t
    (t', d) = removeDeepest t
    elemDepth = depthToRemove t 1

    depthToRemove Nil _ = -1
    depthToRemove (Node _ l y r) depth
      | cmp x y == EQ = depth
      | depthL == -1  = depthR
      | otherwise     = depthL
      where
        depthL = depthToRemove l (depth + 1)
        depthR = depthToRemove r (depth + 1)

    deleteAndReplace Nil = (Nil, False)
    deleteAndReplace t@(Node h l y r)
      | cmp x y == EQ = (fixHeap cmp (hnode l d r), True)
      | bl == True    = (fixHeap cmp (hnode l' y r), bl)
      | br == True    = (fixHeap cmp (hnode l y r'), br)
      | otherwise     = (t, False)
      where
        (l', bl) = deleteAndReplace l
        (r', br)  = deleteAndReplace r

    deleteDeepestLeaf t@(Node h Nil y Nil)
      | cmp x y == EQ = (Nil, True)
      | otherwise     = (t, False)
    deleteDeepestLeaf t@(Node h l y r)
        | bl == True = (hnode l' y r, True)
        | br == True = (hnode l y r', True)
        | otherwise  = (t, False)
      where
        (l', bl) = deleteDeepestLeaf l
        (r', br) = deleteDeepestLeaf r

extract :: Heap a -> a
extract (Heap cmp Nil) = error "Cannot␣extract␣from␣empty␣heap"
extract (Heap cmp (Node _ _ x _)) = x
```

```
  discard :: Heap a -> Heap a
  discard (Heap cmp Nil) = error "Cannot␣extract␣from␣empty␣heap"
  discard (Heap cmp (Node _ Nil x Nil)) = Heap cmp Nil
  discard (Heap cmp t)
    = Heap cmp (fixHeap cmp (hnode l d r))
    where
      (Node _ l x r, d) = removeDeepest t

  detach :: Heap a -> (a, Heap a)
  detach h = (extract h, discard h)
```

Problem 9: Adjacency List Graphs

```
newtype AdjList e v = AdjList [(v, [e])]

instance (Eq e, Edge e v) => Graph (AdjList e v) e v where
  vertices (AdjList ves)    = map fst ves
  edges (AdjList ves)       = concat [snd ve | ve <- ves]
  edgesFrom (AdjList ves) s
    = case es of
        Just e  -> e
        Nothing -> []
    where
      es = lookup s ves
  edgesTo   (AdjList ves) t = filter ((== t) . target) (edges (AdjList ves))
  velem v                   = elem v . vertices
  eelem e                   = elem e . edges
```

Problem 10: Conflict Zones

```
conflictZones :: GameState -> PlanetId -> PlanetId
  -> ([PlanetId], [PlanetId], [PlanetId])
conflictZones g p q = undefined
```