COMP50001: Algorithm Design & Analysis

Sheet 7 (Week 8)

Exercise 7.1

There is an algorithm related to quick sort called quick select: given an unsorted array, it returns the kth smallest element, in $\mathcal{O}(n)$ time.

Just as with quick sort the algorithm partitions the input according to a pivot. Unlike quick sort, the algorithm does not recurse on both partitions: given that the kth element is to be found, if the returned pivot index is bigger than k, only the left partition is needed; otherwise, if it is smaller only the right partition is needed; otherwise they are equal and the pivot is the element required.

Implement $qselect :: Ord\ a \Rightarrow Int \to [a] \to Maybe\ a$, a *persistent* version of quick select that works directly on lists. Also implement $qselect'$ with the same signature, but that makes use of $aqselect :: Ord\ a \Rightarrow Int \to STArray\ s\ Int\ a \to Int \to Int \to ST\ s\ (Maybe\ a)$ to perform an in-place quick select, and comment on the performance difference between the two implementations.

Exercise 7.2

Often it is useful to give efficient but mutation-heavy algorithms a pure interface. Using the functions *runSTArray* and *thaw*, implement a pure interface to the *qsort* presented in lectures:

$$qsortp :: Ord\ a \Rightarrow Array\ Int\ a \to Array\ Int\ a$$

Explain if it is possible to implement a similar pure interface to the functions $index :: Int \to Array\ Int\ a \to a$ and $update :: a \to Int \to Array\ Int\ a \to Array\ Int\ a$ which maintains their $\mathcal{O}(1)$ running time.

Exercise 7.3

Consider the following type which represents sets of strings of as:

$$\textbf{data}\ Trie\ a = Bool \prec [(a, Trie\ a)]$$

1. Calculate the time complexity of *member*, in terms of both the size of the trie and the length of the input list:

$$\begin{aligned}
&member :: Eq\ a \Rightarrow [a] \to Trie\ a \to Bool \\
&member\ [\,]\quad\ (e \prec _) = e \\
&member\ (x : xs)\ (_ \prec ys) = maybe\ False\ (member\ xs)\ (lookup\ x\ ys)
\end{aligned}$$

2. If a is a finite type like *Word4*, the time complexity of *member* changes. Calculate its time complexity when $a \equiv Word4$.

3. Implement $insert :: Eq\ a \Rightarrow [a] \to Trie\ a \to Trie\ a$.

4. Assume you have a perfect hashing function on a, called $hashList :: Hashable\ a \Rightarrow a \to [Bool]$. Implement $nubT :: Hashable\ a \Rightarrow [a] \to [a]$ using this hash function and the trie.

The following is a slow, but correct, implementation of selection.

```
slowselect :: Ord a ⇒ Int → [a] → Maybe a
slowselect i xs
  | i < length xs = Just (sort xs !! i)
  | otherwise     = Nothing
```

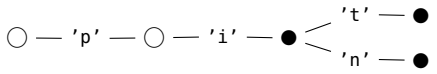
```
swap :: STArray s Int a → Int → Int → ST s ()
swap axs i j = do
  x ← readArray axs i
  y ← readArray axs j
  writeArray axs i y
  writeArray axs j x
```

```
apartition :: Ord a
           ⇒ STArray s Int a
           → Int
           → Int
           → ST s Int
apartition axs p q = do
  x ← readArray axs p
  let loop i j
    | i > j = do swap axs p j
                 return j
    | otherwise = do u ← readArray axs i
                     if u < x
                       then do loop (i + 1) j
                       else do swap axs i j
                               loop i (j − 1)
  loop (p + 1) q
```

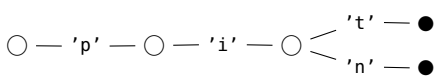
```
runSTArray :: (∀s.ST s (STArray s i a)) → Array i a
thaw       :: Ix i ⇒ Array i a → ST s (STArray s i a)
```

```
fromList :: Eq a ⇒ [[a]] → Trie a
fromList = foldr insert (False ≺ [])
```

```
fromList ["pi","pin","pit"] =
  False ≺
    [('p', False ≺
      [('i', True ≺
        [('n', True ≺ [])
        ,('t', True ≺ [])])])]
```




```
fromList ["pin","pit"] =
  False ≺
    [('p', False ≺
      [('i', False ≺
        [('n', True ≺ [])
        ,('t', True ≺ [])])])]
```



Word4 is the type of 4-bit words, i.e. unsigned integers ranging in value from 0 to 15.

A *perfect* hashing function is one that maps elements without any collisions.

5. Consider another hashing function which returns the bits of the hash in four-bit chunks: $hashChunks :: Hashable\ a \Rightarrow a \to [Word4]$. Calculate the time complexity of $insert \circ hashChunks$ in terms of the cardinality of a.

6. Calculate the time complexity when *hashChunks* returns n-bit chunks.

Exercise 7.4

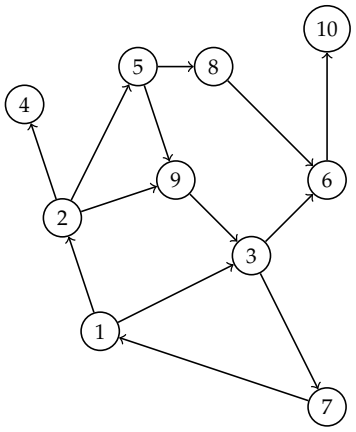
Consider the following representation of graphs:

type $Graph\ a = a \to [a]$

Implement the following two versions of depth-first-search, where $dfs\ g\ r$ returns a list of all the nodes in the graph g visited in depth-first-search order starting from a root r.

$dfs\ :: Ord \quad a \Rightarrow Graph\ a \to a \to [a]$
$dfs' :: Hashable\ a \Rightarrow Graph\ a \to a \to [a]$

dfs should have complexity $\mathcal{O}(n \log n)$, and *dfs'* should be $\mathcal{O}(n)$. You may use data structures implemented in other assignments to implement this solution.



$graph :: Graph\ Int$
$graph\ 1 = [2,3]$
$graph\ 2 = [4,5,9]$
$graph\ 3 = [6,7]$
$graph\ 4 = [\,]$
$graph\ 5 = [8,9]$
$graph\ 6 = [10]$
$graph\ 7 = [1]$
$graph\ 8 = [6]$
$graph\ 9 = [3]$
$graph\ 10 = [\,]$

A valid result for $dfs\ \ graph\ \ 3$ is $[3,6,10,7,1,2,4,5,8,9]$.

Exercise 7.5

Given a uniformly-distributed hash function and n buckets, calculate the chance of a collision happening after inserting m elements.

Exercise 7.6

A bloom filter is a *probabilistic* data structure which can act as a set, with some small probability of false positives in the set. It works by storing a bit array; to insert a value into the set, take multiple hash functions, run each of them on the value, and set the corresponding bits in the bit array.

Implement $nubB :: [a \to Int] \to [a] \to [a]$ which uses a bloom filter of size 256. The first argument is the list of hash functions that should be used.

Exercise 7.7

Cuckoo hashing is an efficient way to handle hash collisions in a hash table. It works by storing two hash tables, with two different hash functions. When inserting, if an element already is in the bucket being inserted into, it is kicked out and inserted into the other hash table. This procedure repeats until an empty bucket is found or the whole hash table has been traversed, at which point two new tables (double the size) are allocated.

Implement a set as a mutable hash table with *insert* and *member* functions using cuckoo hashing.

data $HashSet\ s\ a = ...$
$newHashSet :: ST\ s\ (HashSet\ s\ a)$
$insert\ \ :: Hashable\ a \Rightarrow a \to HashSet\ s\ a \to ST\ s\ ()$
$member :: Hashable\ a \Rightarrow a \to HashSet\ s\ a \to ST\ s\ Bool$

Solutions to the Exercises

Solution 7.1

```
qselect :: Ord a ⇒ Int → [a] → Maybe a
qselect k [ ] = Nothing
qselect k (x : xs)
    | k < p      = qselect k us
    | k > p      = qselect (k − p − 1) vs
    | otherwise = Just x
  where
    (us, vs) = partition (⩽ x) xs
    p = length us
qselect′ :: Ord a ⇒ Int → [a] → Maybe a
qselect′ k xs = runST $ do
  axs ← newListArray (0, n) xs
  mx ← aqselect k axs 0 n
  return mx
where
  n = length xs − 1
aqselect :: Ord a ⇒ Int → STArray s Int a → Int → Int → ST s (Maybe a)
aqselect k axs i j
      | i > j      = return Nothing
      | otherwise = do
        p ← apartition axs i j
        if | k < p      → aqselect k axs i (p − 1)
           | k > p      → aqselect k axs (p + 1) j
           | otherwise → do x ← readArray axs p
                            return (Just x)
```

Solution 7.2

The idea is that *thaw* takes an immutable array and produces a mutable one in some context *s*. Then, the *aqsort* function from lectures can perform an in-place quicksort on this new array. Once the work is complete, the array is returned using *runSTArray*.

```
qsortp axs = runSTArray $ do
      axsm ← thaw axs
      aqsort axsm i j
      return axsm
  where
    (i, j) = bounds axs
aqsort :: Ord a ⇒ STArray s Int a → Int → Int → ST s ()
aqsort axs i j
  | i ⩾ j      = return ()
  | otherwise = do
    k ← apartition axs i j
```

$aqsort\ axs\ i\ (k-1)$
$aqsort\ axs\ (k+1)\ j$

The *index* function poses no problem, since the array is not mutated in any way. In contrast, the *update* function will produce a copy of the array, which cannot be done in $O(1)$ time.

Solution 7.3

1. If a is finite and small enough then there are only finitely many children in the tree. Storing these children in an array makes the lookup at each node constant. In this case, the complexity of *member* is $O(n)$, where the word has length n.

2. The time complexity of the *member* function is bounded, in part, by the branching factor (i.e. the number of children) of each node. By limiting the keys to *Word4*, we have limited the number of children in each node to 16, making the complexity of *member* $\mathcal{O}(n \times 16)$, i.e. $\mathcal{O}(n)$.

3. The *insert* function works by recursing over then input key that is to be inserted into the tree.

   In the base case the list is empty, which means that the tree must be updated so that the current node is *True*

   $$\begin{aligned}&insert :: \forall a.Eq\ a \Rightarrow [a] \to Trie\ a \to Trie\ a\\&insert\ [\,]\ (e \prec ts) = True \prec ts\end{aligned}$$

   The *True* signifies that the key is present at this node in the trie.

   Otherwise, the key is some $x : xs$ and is being inserted into some node $(e \prec ts)$. The value of e remains unchanged: only the last node on the path from the root of the trie taken by following the key is affected: all other node values remain the same. The result of *ins ts* is to find the child whose edge is labelled with x so that the insertion can continue there.

   $$\begin{aligned}&insert\ (x : xs)\ (e \prec ts) = e \prec ins\ ts\\&\quad \textbf{where}\\&\qquad ins :: [(a, Trie\ a)] \to [(a, Trie\ a)]\\&\qquad ins\ [\,] = [(x, insert\ xs\ (False \prec [\,]))]\\&\qquad ins\ ((y, yt) : ts)\\&\qquad\quad \mid x \equiv y \quad\ = (y, insert\ xs\ yt) : ts\\&\qquad\quad \mid otherwise = (y, yt) : ins\ ts\end{aligned}$$

   The definition of *ins* inspects each of the children of the current node. If there are no children, then a new edge is created with the label x, and its child is the tree given by inserting xs into the empty tree. Otherwise there is some edge (y, yt) to consider. If $x \equiv y$ then yt is the tree that corresponds to x. The label y remains unchanged, since it is equal to x, and $insert\ \ xs\ \ yt$ is performed to update the rest of yt. The other siblings ts are left unchanged. If $x \not\equiv y$ then (y, yt) is left unchanged and *ins* continues to search in the siblings ts.

The perfect *hashList* can be simulated as follows, although this implementation is beyond the scope of this course.

$$\begin{aligned}&getBits :: FiniteBits\ a \Rightarrow a \to [Bool]\\&getBits\ x = map\ (testBit\ x)\ [0\,..\,finiteBitSize\ x - 1]\\&hashList :: Hashable\ a \Rightarrow a \to [Bool]\\&hashList = getBits \circ hash\end{aligned}$$

4. The idea here is to build a trie using *insert* and the hash function. Since each element in the list can be turned into a $[Bool]$, this list can be used as the keys in the trie.

   Each element x in the list turns into a function that both looks up x in a tree t by using its hashlist. If the element is not present, then the trie is modified to contain that element.

```
nubT :: ∀a.Hashable a ⇒ [a] → [a]
nubT xs = foldr f b xs (False ≺ [])
  where
    b :: Trie Bool → [a]
    b = const []
    f :: a → (Trie Bool → [a]) → (Trie Bool → [a])
    f x k t
      | member hx t = k t
      | otherwise = x : k (insert hx t)
      where hx = hashList x
```

5. To calculate the worst-case complexity of *insert*, we consider the case when the tree is at its maximum size (i.e. it has every element of a in it), and every node has the maximum number of children. In this case, inserting into the tree is bounded only by the depth of the tree, which is on the order of $\mathcal{O}(\log_{16}|a|)$.

6. Using similar reasoning, the depth of the tree (and therefore the complexity of *insert*), is of the order $\mathcal{O}(\log_{2^{n+1}}|a|)$.

Solution 7.4

The first version uses a *Set* impelmentation, since there is an *Ord* constraint. Given a graph g and a root r, we perform *search Set.empty* $[r]$, where *search* repeatedly pops nodes off of the stack (its second argument), and searches from them if they're not in the set of items it has already seen.

If the element has not been visited yet, then it is added to the output as the next node to be visited, and all of its children are pushed onto the top of the stack.

```
dfsStack :: Ord a ⇒ Graph a → a → [a]
dfsStack g r = search Set.empty [r]
  where
    search _ [] = []
    search seen (x : stack)
      | Set.member x seen = search seen stack
      | otherwise         = x : search (Set.insert x seen) (g x ++ stack)
```

Now there are n nodes to be searched, and assuming the set takes $O(\log n)$ time to check for membership the overall cost is as required.

The recursion pattern over *stack* here is a familiar one, and is actually encapsulated by *foldr*:

```
dfs :: ∀a.Ord a ⇒ Graph a → a → [a]
dfs g r = search r (const [ ]) Set.empty
  where
    search :: a → (Set a → [a]) → (Set a → [a])
    search x k xs
      | Set.member x xs = k xs
      | otherwise       = x : foldr search k (g x) (Set.insert x xs)
```

The expensive operation in the previous code is *member*. The idea is to replace this with a hash table where values are inserted. The insert and the lookup should be $O(1)$.

```
dfs′ :: ∀a.(Hashable a, Eq a) ⇒ Graph a → a → [a]
dfs′ g r = runST $ do
  ht ← HT.new
  search ht r (pure [ ])
where
  search :: HT.HashTable s a () → a → ST s [a] → ST s [a]
  search ht x p = do isMember ← memberHT x ht
                     if isMember
                       then p
                       else do HT.insert ht x ()
                               ys ← foldr (search ht) p (g x)
                               return (x : ys)
```

The *memberHT* function checks for membership by doing a lookup, and returning *True* or *False* accordingly.

```
memberHT :: (Hashable a, Eq a) ⇒ a → HT.HashTable s a b → ST s Bool
memberHT x ht = do
  mx ← HT.lookup ht x
  case mx of
    Just _ → return True
    Nothing → return False
```

Solution 7.5

This is a restating of the well-known *birthday problem*.

It's easier to understand if we first calculate the probability of not finding a collision: for n buckets and 2 elements, that simply means that the second element must not go into the same bucket as the first, which has a probability of $\frac{n-1}{n}$. If we were to add a third element, there would be one fewer bucket to choose from, meaning the new probability is $\frac{n-1}{n} \times \frac{n-2}{n}$. In general, we can calculate the probability of *not* finding a collision with the following:

$$\prod_{i=1}^{m-1} \frac{n-i}{n} \tag{1}$$

And then the probability of a collision is simply the inverse of that, which is the following:

$$1 - \prod_{i=1}^{m-1} \frac{n-i}{n} \tag{2}$$

Or, in Haskell code:

```
collision n m = 1 − product (map (%n) (take m [n, n − 1 . .]))
```

Solution 7.6

This solution will first build an array of the required size (this will be our bit-array):

```
falses :: Int → ST s (STArray s Int Bool)
falses n = newArray (0, n − 1) False
```

Next, we will write a function that returns all of the indices into the bit array given the list of hash functions:

```
bloomIndices :: [a → Int] → a → [Int]
bloomIndices xs x = map (λh → h x `mod` 256) xs
```

Notice that we take the hash modulo the size of the array.

Next we will write a function which reads the presence of our element from the *STArray*:

```
bloomMember :: STArray s Int Bool → [Int] → ST s Bool
bloomMember axs is = do
  ps ← sequence [readArray axs i | i ← is]
  return (and ps)
```

And we also need a function which *inserts* into the bloom filter:

```
bloomInsert :: STArray s Int Bool → [Int] → ST s ()
bloomInsert axs is = do
  sequence [writeArray axs i True | i ← is]
  return ()
```

Finally, we can tie all of this together like so:

```
nubB :: [a → Int] → [a] → [a]
nubB hashes xs = runST $ do
  axs ← falses 256
  go axs xs
  where
    go axs [ ] = pure [ ]
    go axs (x : xs) = do
      let is = bloomIndices hashes x
      r ← bloomMember axs is
      if r
        then go axs xs
        else do
          bloomInsert axs is
          fmap (x:) (go axs xs)
```

Solution 7.7

As hash functions, we'll use the following two functions to demonstrate the implementation (although it is important to note that these would not be appropriate for a full implementation).

```
hash1, hash2 :: Hashable a ⇒ a → Int
hash1 = hash
hash2 = hashWithSalt 4
```

We then have our definition of the hash set. Since this is basically a mutable object, the whole type will be held behind an *STRef*:

```
data HashSetS s a
  = HashSet Int
            (STArray s Int (Maybe a))
            (STArray s Int (Maybe a))
type HashSet s a = STRef s (HashSetS s a)
```

The first field of the *HashSet* constructor holds the size of the hash tables.

We can extract the elements in the two hash tables with the following function:

```
elems' :: HashSet s a → ST s [a]
elems' hs = do
  HashSet _ xs ys ← readSTRef hs
  exs ← getElems xs
  eys ← getElems ys
  return (catMaybes (exs ++ eys))
```

And we can create an empty hash table with the following:

```
make' :: Int → ST s (HashSet s a)
make' n = do
  axs ← newArray (0, n − 1) Nothing
  ays ← newArray (0, n − 1) Nothing
  newSTRef (HashSet n axs ays)
```

The *bool* function is a helper function defined as so:

```
bool :: a → a → Bool → a
bool x _ False = x
bool _ x True  = x
```

With those tools in place, the *insert* function is not too difficult:

```
insert' :: Hashable a ⇒ a → HashSet s a → ST s ()
insert' x hs = do
  HashSet sz _ _ ← readSTRef hs
  go False x (sz ∗ 2)
  where
    go _ x 0 = do
      HashSet sz _ _ ← readSTRef hs
      hs' ← make' (sz ∗ 2)
      exs ← elems' hs
      insert' x hs'
      sequence [insert' e hs' | e ← exs]
```

```
    nhs ← readSTRef hs'
    writeSTRef hs nhs
  go s x v = do
    HashSet sz xs ys ← readSTRef hs
    let n = bool hash1 hash2 s x `mod` sz
    let axs = bool xs ys s
    xv ← readArray axs n
    writeArray axs n (Just x)
    case xv of
      Nothing → return ()
      Just y → go (¬ s) y (v − 1)
```

The *go* helper function does the bulk of the work here. The first parameter is a boolean, telling the function which of the two arrays it's currently focused on; the second parameter is the element actually being inserted, and the final parameter is a counter tracking how often the *go* function has been called, so that we know when to stop and reallocate two new arrays.

```
member' :: (Hashable a, Eq a) ⇒ a → HashSet s a → ST s Bool
member' x hs = do
  HashSet sz _ _ ← readSTRef hs
  go False (hash1 x) sz
  where
    go _ h 0 = return False
    go s h i = do
      HashSet asz xs ys ← readSTRef hs
      let axs = bool xs ys s
      x' ← readArray axs (h `mod` asz)
      case x' of
        Nothing → return False
        Just y
          | x ≡ y → return True
          | otherwise → go (¬ s) (bool hash2 hash1 s y) (i − 1)
```