# 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 $k$th 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 $k$th 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 $qselect :: Ord\ a \Rightarrow Int \rightarrow [a] \rightarrow Maybe\ a$, a *persistent* version of quick select that works directly on lists. Also implement $qselect'$ with the same signature, but that makes use of $aqselect ::$ $Ord\ a \Rightarrow Int \rightarrow STArray\ s\ Int\ a \rightarrow Int \rightarrow Int \rightarrow ST\ s\ (Maybe\ a)$ 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:

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

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

*Exercise 7.3*

Consider the following type which represents sets of strings of $a$s:

**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:

   $member :: Eq\ a \Rightarrow [a] \rightarrow Trie\ a \rightarrow Bool$
   $member\ []\qquad (e \prec \_) = e$
   $member\ (x : xs)\ (\_ \prec ys) = maybe\ False\ (member\ xs)\ (lookup\ x\ ys)$

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 $insert :: Eq\ a \Rightarrow [a] \rightarrow Trie\ a \rightarrow Trie\ a$.

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

5. Consider another hashing function which returns the bits of the hash in four-bit chunks: *hashChunks* :: *Hashable a* $\Rightarrow$ *a* $\rightarrow$ [*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* $\rightarrow$ [*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*      *a* $\Rightarrow$ *Graph a* $\rightarrow$ *a* $\rightarrow$ [*a*]
*dfs'* :: *Hashable a* $\Rightarrow$ *Graph a* $\rightarrow$ *a* $\rightarrow$ [*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.

*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* $\rightarrow$ *Int*] $\rightarrow$ [*a*] $\rightarrow$ [*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.
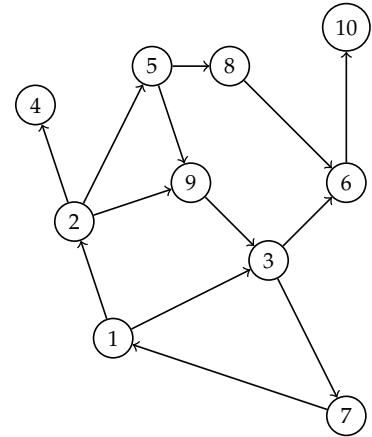


*graph* :: *Graph Int*
*graph* 1  = [2, 3]
*graph* 2  = [4, 5, 9]
*graph* 3  = [6, 7]
*graph* 4  = []
*graph* 5  = [8, 9]
*graph* 6  = [10]
*graph* 7  = [1]
*graph* 8  = [6]
*graph* 9  = [3]
*graph* 10 = []

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

**data** *HashSet s a* = ...

*newHashSet* :: *ST s* (*HashSet s a*)

*insert*   :: *Hashable a* $\Rightarrow$ *a* $\rightarrow$ *HashSet s a* $\rightarrow$ *ST s* ()
*member* :: *Hashable a* $\Rightarrow$ *a* $\rightarrow$ *HashSet s a* $\rightarrow$ *ST s Bool*

## Exercise 7.1

There is an algorithm related to quick sort called quick select: given an unsorted array, it returns the $k$th 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 $k$th 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 $qselect :: Ord\ a \Rightarrow Int \rightarrow [a] \rightarrow Maybe\ a$, a *persistent* version of quick select that works directly on lists. Also implement $qselect'$ with the same signature, but that makes use of $aqselect ::$ $Ord\ a \Rightarrow Int \rightarrow STArray\ s\ Int\ a \rightarrow Int \rightarrow Int \rightarrow ST\ s\ (Maybe\ a)$ to perform an in-place quick select, and comment on the performance difference between the two implementations.

```
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
```
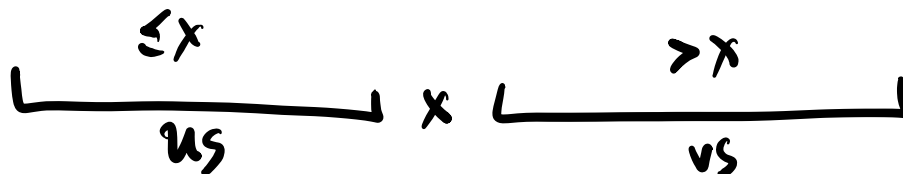
```
qselect' :: Ord a => Int -> [a] -> Maybe a
qselect' k xs = runST $ do
      axs <- newListArray (0, n-1) xs
      mx <- aqselect k axs 0 n
      return mx
  where
      n = length xs


aqselect :: 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 -> do aqselect k axs i (p-1)
             | k > p -> do aqselect k axs (p+1) j
             | otherwise -> do x <- readArray axs p
                               return (Just x)
```

{-# LANGUAGE
      MultiWayIf #-}

*Exercise 7.2*

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

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

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

```haskell
qsortp :: Ord a => Array Int a -> Array Int
qsortp axs = runSTArray $ do
       axsm <- thaw axs
       aqsort axsm i j
       return axsm
   where
       (i,j) = bounds axs
```

$$\text{data } Trie\ a = Bool\ \lessdot\ [(a, Trie\ a)]$$

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

```haskell
                    ↙ Res
insert :: Eq a => [a] -> Trie a -> Trie a
insert []     (e < ts) = True < ts
insert (x:xs) (e < ts) = ins ts
    where
        ins :: [(a, Trie a)] -> [(a, Trie a)]
        ins []        = [(x, insert xs (False < []))]
        ins ((y,yt):ts)
            | x == y    = (y, insert xs yt) :ts
            | otherwise = (y, yt) : ins ts
```