1) Time Complexities

We define the cost for evaluating algorithms based on their input size. We analyse performance on this as it varies. For insert :: int -> [int] -> [int] we work out this cost by writing a recurrence relation.

We perform worst case analysis – for insert this would be an insert right at the end:

$T_{insert}(n) = 1 + T_{insert}(n-1) = 1 + 1 + \ldots 1 + T_{insert}(n-n) = n + T_{insert}(0) = n+1$.

Thus insert is O(n). We do all our calculations under strict evaluation.

```
isort :: [int] -> [int]
isort [] = []
isort (x:xs) = insert x (isort xs)
```

We have: $T_{isort}(0) = 1$, $T_{isort}(n) = 1 + T_{insert}(n-1) + T_{isort}(n-1)$.

So $T_{isort}(n) = 1 + T_{insert}(n-1) + T_{isort}(n-1)$
$= 1 + n + (1 + T_{insert}(n-2) + T_{isort}(n-2))$.
$= 2n+1 + T_{isort}(n-2)$

Isort(0) = 1, (1) = 3, (2) = 6, (3) = 1 + 3 + 6 = 10.

This pattern unfolds to: n+1 + n + n-1 + … 2 + 1, and is thus the sum n+1 numbers. This is (n+1)x(n+2) / 2, and thus $O(n^2)$.

2) Evaluation

In other evaluation models, like lazy evaluation, minimum = head isort is actually O(n). This is hard to deal with thus I'll only do strict evaluation.

2.1) Normal Forms

There are 3 types of normal forms when reducing a function (a function is f(x) = y <-> f = λx y – **Lambda Calculus form**)

Normal Form: Cannot be evaluated any further - no further reductions.

Weak Head Normal Form: Just a constructor, we know what type it is from the constructor, but it is reducible. e.g. [3+2], λx.3+4, Just(8+9)

2.2) Expressions

The key to giving an algorithm an overall cost is to give a series of rules that will assign a cost to a function by allocating a cost to its constituent parts. **Expressions** can be broken down into terms of the following grammar:

```
e ::= x              (variables)
    | k              (constants)
    | f e1...en      (applications)
    | if e then e1 else e2   (conditionals)
```

A **function** f in this language is assume to be defined by the form f x1 … xn = e. **Infix operations** are written as x + y instead of (+) x y. **Primitive constants** such as True, False, 0, 1, 2, are available, as are stand operations on them such as ¬, ⩽, (+), and (×). **List constants and operations** are also primitive, such as [ ], (:), null, head and tail. There are **two main types of evaluation order** which changes runtime as mentioned before: 1) **Applicative order**: Evaluating arguments to a function before evaluating the function itself. 2) **Normal order**: evaluating the function before evaluating its arguments. Applicative order doesn't always terminate, but if it does then these both always reduce to some normal form. In a strict setting we use applicative order, in a lazy setting we use normal order.

2.3) Strict Time Analysis

Given a function f of n arguments, $T(f) x_1 \ldots x_n$ – the number of steps it takes to evaluate f $x_1 \ldots x_n$ can be worked out as follows: 1) For primitive f, $T(f)\ x_1 \ldots x_n = 0$, e.g. T (head) xs = 0, T((+)) x y = 0. 2) For any other function: f $x_1 \ldots x_n$ = e, T(f) $x_1 \ldots x_n$ = 1 + T(e) (We are given that the arguments are already evaluated.

Evaluating a function is 1 step, and then we need to carry out the function body e). Now, T(e) can be defined in terms of expressions, by induction on e:

Primitives and Variables: T(x) = 0, T(k) = 0

Application: T(f e1 … en) = T(f) e1 … en + T(e1) + … + T(en).

Conditional: T(if e then e1 else e2)=T(e)+if e then T(e1)elseT(e2)

Examples:

T(3+4) = T((+) 3 4) + T(3) + T(4) = 0 + 0 + 0 = 0

T(length xs) is:if null xs then 0, else 1 + length (tail xs) =
1 + T(if null xs then 0 else 1++ length (tail xs)) (**cond rule**)
= 1 + ~~T(null xs)~~ +if null xs then ~~T(0)~~ else T(1+length(tail xs)) (**cond**)
= 1 + if null xs then 0 else ~~T(+) 1 (length (tail xs))~~ +T(1)+T(length (tail xs))(**app rule**)
= 1 + if null xs then 0 else T(length (tail xs)) + ~~T(tail xs)~~ (**prim rule**)

We applied **primitive/var rule** throughout multiple times, for terms like T(n).

So T(length) xs = 1 + if null xs then 0 else T (length) (tail xs), generating a recurrence relation.

Composition Rule: The cost of f(g(x)) is T(f(g(x)) = T (f) (g x) + T(g) x

3) Asymptotic Functions

High to low cost: $O(n^n)$, $O(k^n)$, O(n!), $O(n^k)$, $O(n \log_n)$, O(n), O(root n), O(log n), O(1)

When dealing with asymptotics we restrict ourselves to a family of mathematical functions called **L-Functions.**

3.1) L-Functions – an L Function is a **real**, **positive**, **monotonic** (only moves in 1 direction on y axis), one-valued function (each point has a unique value in the range) on a real variable **defined for all values greater than some definite value** by a finite combination of algebraic symbols, exponentials, and logarithms, operating on real constants and the variable. **Theorem**: Any L-function f is ultimately continuous, of constant sign, monotonic, and as n → ∞, the value f(n) tends to one of 0, ∞, or some other definite limit. Can categorise L-Functions with Du Bois Reymond.

3.2) Du-Bois Reymond Notation

The first important concept when looking at complexity is the notion of the rate of increase of a function relative to another: a function can only be seen to grow quickly or slowly with respect to some other function. As is standard, the rate of increase of two functions can be understood as the ratio between them.

Now suppose f and g are L-functions, and consider the ratio of the L-function f /g(n) = f(n)/g(n) as n tends to infinity. This gives rise to a family of operations, ≺, ≼, ≍, ≽, and ≻ that can be used to compare functions:

$f \prec g \iff \lim_{n\to\infty} f(n) / g(n) = 0$
$f \preccurlyeq g \iff \lim_{n\to\infty} f(n) / g(n) < \infty$
$f \asymp g \iff 0 < \lim_{n\to\infty} f(n) / g(n) < \infty$
$f \succcurlyeq g \iff \lim_{n\to\infty} f(n) / g(n) > 0$
$f \succ g \iff \lim_{n\to\infty} f(n) / g(n) = \infty$ (3.5)

These operations have good calculational properties:

We have **trichotomy**: f is less than g, f is comparable to g, f is more than g.

Converse: f ≺ g ↔ g ≻ f.

Transitivity: f ≺ g && g ≺ h → f ≺ h (same with less and equal).

3.3) Bachman-Landau (Big-O) Notation

f ∈ o(g(n)) ⟺ f ≺ g
f ∈ O(g(n)) ⟺ f ≼ g
f ∈ Θ(g(n)) ⟺ f ≍ g
f ∈ Ω(g(n)) ⟺ f ≽ g (3.12)
f ∈ ω(g(n)) ⟺ f ≻ g (3.13)

These sets can also be defined directly:

o(g(n)) = { f |∀δ > 0.∃n0 > 0.∀n > n0. f(n) < δg(n) }
O(g(n)) = { f |∃δ > 0.∃n0 > 0.∀n > n0. f(n) ⩽ δg(n) }
Θ(g(n)) = O(g(n)) ∩ Ω(g(n)) (3.16)
Ω(g(n)) = { f |∃δ > 0.∃n0 > 0.∀n > n0. f(n) ⩾ δg(n) } (3.17)
ω(g(n)) = { f |∀δ > 0.∃n0 > 0.∀n > n0. f(n) > δg(n) } (3.18) (3.19)

4) Lists

Lists in Haskell are defined like so: data [a] = [ ] | (:) a [a]

The data keyword indicates that a new datatype is being introduced. The type itself is called [a], which can be constructed by means of the constructors, [ ] for empty lists, and (:) for adding an element to a list, introduced to the right of the equality symbol. The types of these constructors is: [ ] :: [a] (:) :: a → [a] → [a] The ++ operation is O(n):

```
(++) :: [a] → [a] → [a]
[] ++ ys = ys
(x : xs) ++ ys = x : (xs ++ ys)
```

The whole list must be traversed, so given length xs = n, then T(++)(n) ∈ O(n). This pattern of recursion on the structure of a list itself crops up frequently, and is captured by the **foldr** function:

```
foldr :: (a → b → b) → b → [a] → b
foldr f k [ ] = k
foldr f k (x : xs) = f x (foldr f k xs)
```

foldr applies functions to a list like so f x1 (f x2 (...(f xn k))), effectively applying the two arg function to things adjacently until we get the result. When foldr is used with a binary operator (⋄), the following holds: foldr (⋄) ϵ [x1, x2, ..., xn] = x1 ⋄ (x2 ⋄ (... ⋄ (xn ⋄ ϵ)))

Furthermore, when (⋄) is associative and ϵ is a neutral element, this is simply: foldr (⋄) ϵ [x1, x2, ..., xn] = x1 ⋄ x2 ⋄ ... ⋄ xn One way to interpret this is that applying foldr is a destructor of lists, dual to the operations (:) and [ ] which are constructors. This can be seen by considering the effect of foldr (:) [ ] , it is equal to the identity function. Thus we can define some list operations in terms of foldr: concat [xs, xs1, xs2…] = xs ++xs1++xs2… = foldr (++) [] xss, thus concat = foldr (++) []. It is O(nm), n = num lists, m = length of longest list.

foldl is foldr but left recursive:

```
foldl :: (b → a → b) → b → [a] → b
foldl f k [ ] = k
foldl f k (x : xs) = foldl f (f k x) xs
```

Some functions are **faster/slower using left recursion** rather than right recursion despite producing the same results. (concatl using foldl is much slower, $O(n^2m)$ as we have our resultant list on the left hand side each time, rather than what we're adding. And we see the definition of concat means that we'd thus traverse much more space). **Just because functions produce the same result doesn't mean they're the same speed.**

Monoid: a set X that is equipped with an associative binary operation (⋄) : X × X → X and a neutral (identity) element ϵ : X.

5) Abstract Lists

Abstract interfaces for lists can be created that can be instantiated to **different concrete implementations with varying complexity characteristics.** Our interface looks like this: *class List list where*

fromList :: [a] → list a	*toList :: list a → [a]*
normalize :: list a → list a	*empty :: list a*
single :: a → list a	*cons :: a → list a → list a*
snoc :: list a → a → list a	*head :: list a → a*
tail :: list a → list a	*init :: list a → list a*
last :: list a → a	*isEmpty :: list a → Bool*
isSingle :: list a → Bool	*length :: list a → Int*
(++) :: list a → list a → list a	*(!!) :: list a → Int → a*

fromList takes us to our abstract implementation, toList from it. normalize applies fromList toList. (toList fromList = id, and should be maintained as such) empty is a function constructing the empty list, single is a constructor. cons adds an item to front, snoc to end. !! returns a specific index of the list. We have to use toList and fromList in our concrete definitions usually:

```
cons :: a → list a → list a
cons x xs = fromList (x : toList xs)
single :: a → list a
single x = fromList [x]
snoc :: list a → a → list a
snoc xs x = fromList (toList xs ++ [x])
```

Here are some concrete implementations of lists:

4.1) Default Lists – the standard list implementation. toList and fromList are simply the identity function. We can define all functions by **referring to their Prelude** versions or redefining them in place.

4.2) Tree Lists - A binary tree with values at its leaves can be considered to be a list, where an in-order traversal of the list from left to right corresponds to the order of the list elements. This representation is good as **appending two trees together is achieved by simply placing them under a parent fork**.

4.3) Difference Lists – We get constant time cons, snoc and ++. Other operations become more expensive though. For a difference list we replace (++) with function composition. Thus appending lists together always ends up in a right-associated list which occurs because of the definition of composition.

(◦) :: (b → c) → (a → b) → (a → c)
(g ◦ f) x = g (f x)

If f = (xs++), g = (ys++) and h = (zs++), then their composition: ((zs++) ◦ (ys++) ◦ (xs++)) [ ] = zs ++ (ys ++ (xs ++ [ ]))

Thus we get right associative appends which is desirable as it is O(1).

This is how we implement them:

newtype DList a = DList ([a] → [a])

Note carefully that the append function (++) used here has type:

(++) :: List list ⇒ list a → list a This has a constraint that states that list must be a member of the List class, which allows it to work on different types, as annotated in the calculation. We get:

```
instance List DList where
toList :: DList a → [a]
toList (DList fxs) = fxs [ ]
fromList :: [a] → DList a
fromList xs = DList (xs++)
(++) :: DList a → DList a → DList a
DList fxs ++ DList fys = DList (fxs ◦ fys)
```

fxs represents funcs on lists, fxs = (xs++).

Recursive Bitonic

```
bitonic::(Int→Int→Double)→Int→Double
bitonic δ 0 = 0
bitonic δ 1 = 2 × δ 0 1
bitonic δ n =
   minimum [ bitonic δ k – δ (k – 1) k
   + δ (k – 1) n
   + sum [δ i (i + 1) | i ← [k . . n – 1]]
   | k ← [1 . . n – 1]]
```

6) Divide and Conquer – fundamental algorithmic strategy. Consists of three parts: 1) Divide a problem into subproblems 2) Solve subproblems into subsolutions 3) Combine subsolutions into a solution

An example is merge sort:

```
msort :: [Int] → [Int]
msort [ ] = [ ]
msort [x] = [x]
msort xs = merge (msort us) (msort vs)
where
   (us, vs) = splitAt (n 'div' 2) xs
   n = length xs
merge :: [Int] → [Int] → [Int]
merge [ ] ys = ys
merge xs [ ] = xs
merge (x : xs) (y : ys)
 | x ⩽ y = x : merge xs (y : ys)
 | otherwise = y : merge (x : xs) ys
```

Memoized Bitonic

```
bitonic'' :: (Int → Int → Double)→Int → Path
bitonic'' δ n = table ! n where
  table = tabulate (0, n)
  mbitonic mbitonic :: Int → Path
  mbitonic 0 = Path 0 [ (0, 0)]
  mbitonic 1 = Path (2 × δ 0 1) [(0,1),(0,1)]
  mbitonic n = minimum [table ! k-δ' (k-1) k
          + δ ' (k – 1) n
          + sum [δ' i (i+1) | i ← [k..n-1]]
           | k ← [1 . . n – 1]]
where
  δ' :: Int → Int → Path
  δ' i j = Path (δ i j) [(min i j, max i j)]
```

msort splits the list in halves, and then we merge back up starting with the smallest lists. We see that merge does the bulk of the work in its recursive case, swapping the value of x and y. This repeated splitting gives us $T_{msort}(0) = 1$, $T_{msort}(1) = 1$ $T_{msort}(n) = T_{length}(n) + T_{splitAt}(n/2) + T_{merge}(n/2) + 2 \times T_{msort}(n/2)$, and solving it gives us O(n log n). The key is that to merge two lists of size n (we only merge lists of similar size), it takes at most n/2 comparisons. And we build the list up merging as we go so we are comparing using small lists. Quicksort is another divide and conquer algorithm. The time complexity is usually O(n log n), but with bad pivot choice is $O(n^2)$ and thus the worst case (and so time complexity) is $O(n^2)$. Here we choose pivot = first element always, split into a list of items smaller than it and one bigger. We then qsort these two lists which are split, and combine results.

```
qsort :: [Int] → [Int]
qsort [ ] = [ ]
qsort [x] = [x]
qsort (x : xs) = qsort us ++ [x] ++ qsort vs
   where (us, vs) = partition (<x) xs
partition :: (a → Bool) → [a] → ([a], [a])
partition p xs = (filter p xs, filter (¬ ◦ p) xs)
```

7) Dynamic Programming

We *trade storage for speed* using **memoization**. The speedup comes from caching subsolutions with memorization and later looking them up rather than recomputing these subsolutions. Our strategy is:

1. Write an inefficient recursive algorithm that solves the problem.
2. Improve efficiency by storing intermediate shared results. (using tabulate and memo)

We should use an array to store results in Haskell as arrays are O(1) compared to a list's O(n). We make arrays from lists using this function

array :: Ix i ⇒ (i, i) → [(i, a)] → Array i a. The array type has (!) :: Ix i ⇒ Array i a → i → a, which can be used to look things up in constant time.

```
fib :: Int → Integer
fib 0 = 0
fib 1 = 1
fib n = fib (n – 1) + fib (n – 2)
```

Here's an example, computing the nth Fibonacci number in O(n), using bottom up DP. **memo** must be in the same scope as table. We see memo mirrors our old fib recursion.

```
fib' :: Int → Integer
fib' n = table ! n
  where
   table :: Array Int Integer
   table = tabulate (0, n) memo
   memo 0 = 0
   memo 1 = 1
   memo n = table ! (n – 1) + table ! (n – 2)

tabulate :: Ix i ⇒ (i, i) → (i → a) → Array i a
tabulate (u, v) f = array (u, v) [ (i, f i) | i ← range (u, v)]
```

Here's the standard tabulate function which produces our table by applying our function f to all values between x and y. (0, and n in this case). The magic comes from tabulating with our memo function. All we have to do now is customise our function supplied to tabulate, to get the values we need (it should follow the pattern of the old recursive version), and get the correct array index as the final answer.

7.1) Edit Distance Problem – this is a complex DP problem which requires a 2d array, and thus indexing. In this problem we need to find the number of insertions, deletions and updates it takes to turn one string into another. The problem is simplified by considering only deletions and updates - insertion of a char into a string is the same as deleting the char from the other. One way to visualise it is moving to the left = deletion of first char in first string, right = deletion of first from right, middle = deletion of both first chars if they match. We capture this with this recursive algorithm:

```
dist :: String → String → Int
dist xs [ ] = length xs
dist [ ] ys = length ys
dist xs@(x:xs) ys@(y:ys) = minimum [dist xxs ys + 1, dist xs yys + 1,
                      dist xs ys + if x ≡ y then 0 else 1]
```

Minimum lets us consider the cheapest cost of three choices at each node. This is very inefficient - $O(3^{|m+n|})$. We can fix this with DP, but we need to make our problem amenable to DP as strings can't be used to index into an array. We do this by measuring our progress across xs and ys with indices i and j rather than cutting the head of the string away:

```
dist' :: String → String → Int → Int → Int
dist' xs ys i 0 = i
dist' xs ys 0 j = j
dist' xs ys i j = minimum [dist' xs ys i (j–1) +1, dist' xs ys (i–1) j +1,
dist' xs ys (i–1) (j–1) + if x≡y then 0 else 1]
where
  m = length xs
  n = length ys
  x = xs !! (m – i)
  y = ys !! (n – j)
```

Tabulating this is quite easy now. Just take our template, replace memo with the definition of dist', pass in a 2d array index into tabulate (doesn't need to be redefined) We do have to use fromList to make arrays rather than lists – to remove the !!.

```
dist'' :: String → String → Int
dist'' xs ys = table ! (m, n)
 where
   table = tabulate ((0, 0),(m, n)) (uncurry memo)
   memo :: Int → Int → Int
   memo i 0 = i
   memo 0 j = j
   memo i j = minimum [table ! (i, j – 1) + 1, table ! (i – 1, j) + 1,
                  table ! (i – 1, j – 1) + if x ≡ y then 0 else 1]
    where x = axs ! (m – i)
          y = ays ! (n – j)
          m = length xs
          n = length ys
          axs, ays :: Array Int Char
          axs = fromList xs
          ays = fromList ys
```

We can also do axs = listArray (0, length xs-1) xs

8) Amortized Analysis - Sometimes the cost of an algorithm can't be derived by its singular runtime – we need the wider context. This is done with **Amortized Analysis**.

8.1) Deques - in normal lists, adding to the back is expensive (O(n)). This is shown with snoc – based on ++. A double ended queue is a queue where elements can be added to the front or back of the list. data Deque a = Deque [a][a] Deques have two lists – xs and sy. This has a list of elements xs, and reversed elements sy. To add to the back we add to the second list. This is because toList is implemented by xs ++ reverse sy, so sy is always reversed when we use it (and thus these operations are more expensive but adding to back is cheap). Two invariants are maintained to keep operations efficient **as a whole**: 1) **isEmpty xs ⇒ isEmpty sy ∨ isSingle sy**

2) isEmpty sy ⇒ isEmpty xs ∨ isSingle xs

If one of the lists is empty the other one has at most one element. fromList is then:

```
fromList xs = Deque ys (reverse zs)
   where (ys, zs) = splitAt (length xs'div' 2) xs
empty :: Deque a
empty = Deque [ ] [ ]
```

snoc must be implemented like so to maintain the invariant:

```
snoc :: Deque a → a → Deque a
snoc (Deque [ ] sy) x = Deque sy [x]
snoc (Deque xs sy) x = Deque xs (x : sy)
```

Following this implementation: checking if a deque is empty / single:

```
isEmpty :: Deque a → Bool
isEmpty (Deque xs sy) = isEmpty xs ∧ isEmpty sy
isSingle :: Deque a → Bool
isSingle (Deque xs sy) = (isEmpty xs∧isSingle sy)∨(isSingle xs∧isEmpty sy)
```

The important thing is tail:

```
tail :: Deque a → Deque a
tail (Deque [ ] [ ]) = error "tail: empty list"
tail (Deque [ ] sy) = empty
tail (Deque [x] sy) = fromList (reverse sy)
tail (Deque (x : xs) sy) = Deque xs sy
```

The deque [x] sy case is expensive – O(n) (uses fromList and reverse). Consider carrying out tail xs for a deque xs of length n. Based off this worst case we'd assume the cost of tail is O(n). However consider repeated calls of tail in a chain until the list is exhausted. We'd only ever encounter the 3rd case ONCE – our cost being $O(n^2)$ for having n calls of an algorithm of worst case cost O(n) is misleading. We should do **Amortized Analysis.**

8.2) Amortization – The general setting is a sequence of operations $op_0 \ldots op_n$ acting on an initial datastructure xs_0. To perform amortized analysis we must:

1. Define a **cost function** $C_{opi}(xs_i)$ **for each operation op_i on data xs_i**.
2. Define an **amortized cost function $A_{opi}(xs_i)$ for each operation op_i on data xs_i**
3. A size function S(xs) that calculates the size of data xs.

The costs functions estimate how many steps it would take for each operation to execute. The goal is to define these functions so that they can do an accounting of how much work needs to be done to execute an operation on a datastructure. They should be defined so that the following holds: $C_{opi}(xs_i) \leqslant A_{opi}(xs_i) + S(xs_i) - S(xs_{i+1})$ **(*)**

This says for any given data xsi, the cost of executing the operation opi is less than the amortized cost, plus the difference between the datastructure before and after the operation. If this inequality can be shown to be true, then the summation over a series of operations:

$$\sum_{i=0}^{n-1} C_{op_i}(xs_i) \leqslant \sum_{i=0}^{n-1} A_{op_i}(xs_i) + S(xs_0) - S(xs_n)$$

When $S(xs_n) = 0$ then this implies: **(**)**

$$\sum_{i=0}^{n-1} C_{op_i}(xs_i) \leqslant \sum_{i=0}^{n-1} A_{op_i}(xs_i)$$

This means the sum of the cost functions is less than the sum of the amortized costs. **For example if $A_{opi}(xs) = 1$, then the cost function is bounded by O(n) – this is how we derive the Amortized cost.**

Example:

1) Assign costs: $C_{cons}(xs) = 1$, $C_{snoc}(xs) = 1$ $C_{head}(xs) = 1$, $C_{last}(xs) = 1$ C_{tail}(Deque xs sy) = if length xs > 1 then 1 else length sy

2) We assign an amortized cost that is higher than some operations and lower than others: $A_{opi}(xs) = 2$.

3) We assign a size S(Deque xs sy) = |length xs – length sy| (measures imbalance between two lists as tail is expensive when xs = 1)

Now, if **(*)** can be shown to hold then these are valid:

Consider Deque xs' sy' = tail (Deque xs sy), where length sy = k.

In the worst case, when xs is a singleton list, this implies that:

S(Deque xs sy) = k – 1

S(Deque xs' sy') = 1 (because we encounter fromList reverse sy, and fromList constructs a new deque with the elements split across the two lists evenly). So, substituting into (*) this results in:

Ctail(Deque xs sy) ⩽ Atail(Deque xs sy) + S(Deque xs sy) – S(Deque xs' sy'). The RHS evals to: 2 + |length xs – length sy| – length xs' – length sy'| = (k – 1) – 1. The LHS is k. LHS <= RHS.

This is clearly true. Therefore our choice of C, A and S are correct, and the time complexity of these instructions is bounded by O(n) (as A was constant time), and the amortized cost of tail is O(1) (******)

Using the triangle inequality (our thing on RHS ⩾ {|a – b| ⩽ |a| + |b|} is often important).

9) Random Access Lists

9.1) Peano Numbers are a simplistic way of counting natural numbers: a number is either zero, or one more than some other number. This represented by the Peano datatype with some simple functions below:

```
data Peano = Zero | Succ Peano
inc :: Peano → Peano
inc n = Succ n
dec :: Peano → Peano
dec (Succ n) = n
add :: Peano → Peano → Peano
add Zero n = n
add (Succ m) n = Succ (add m n)
```

This is essentially the same as the structure of lists, except lists have some data involved. (inc = cons, dec = tail, add = ++, zero = empty, succ peano = cons a list a). A better counting system is Binary:

9.2) Binary Numbers:

```
type Binary = [Digit]
data Digit = O | I
   deriving Eq
```

We use a list of digits to store our number, LSB first ([I,O,I,I] = $2^0 + 2^2 + 2^3$ = 13). Add, sub are defined standardly. inc is defined as follows:

```
inc :: Binary → Binary
inc [ ] = [I]
inc (O : bs) = I : bs
inc (I : bs) = O : (inc bs)
```

The worst case (I : bs) is O(n). But this case doesn't always occur so we can use Amortized Analysis:

1) The cost function: $C_{inc}(bs)$ = t + 1 where t = length (takeWhile (≡ I) bs)

2) $A_{inc}(bs) = 2$

3) $S_{inc}(bs)$ = b where b = length (filter (≡ I) bs) (aka the absolutely explosively bad case).

On Size Functions: we're looking for some thing which gets "worse and worse" until we reach the "bad state" from which an expensive operation puts us into "the good state". Our size function is the measure of the potential for an expensive operation to occur. For the dequeue case, we took |length xs – sy| as this measures the balance between the two, which gets worse before exploding and being repaired. For the binary number case we measure (length (filter (== I) bs) as we measure the number of Is which gets worse before exploding and being repaired. With Amortized Analysis, we just want to prove that our explosion happens over enough time and is cheap enough that our time complexity isn't fucked.

Now apply these to (*): Given a list of binary digits bs and another bs' = inc bs, the following holds:

$C_{inc}(bs) \leqslant A_{inc}(bs) + S_{inc}(bs) - S_{inc}(bs') \Leftrightarrow$
t + 1 ⩽ 2 + b – b ' where b ' = b – t + 1 ⇔
t + 1 ⩽ 2 + b – (b – t + 1) ⇔ t + 1 ⩽ t + 1

This is true so the approx. amortized cost is O(1).

9.3) Binary Tree Lookup – Balanced Binary Trees are efficient.

data Tree a = Tip | Leaf a | Fork Int (Tree a) (Tree a)

Three constructors, 2 base cases. Tip = tree with no data, Leaf x only has one item. Fork n l r puts two trees together and the size of the tree. We only store data in the leaves, not in each node. We use a **smart constructor** to maintain that the size is properly stored in n:

```
fork :: Tree a → Tree a → Tree a
fork l r = Fork (length l + length r) l r
```

We can construct lists from these quite easily, the base cases are easy and the recursive case just does toList l ++ toList r. The length function is simply 0 and 1 for the base cases, and n for the Fork case.

The lookup function is slightly unusual:

```
(!!) :: Tree a → Int → a
Tip !! n = error "(!!): no values in a Tip!"
Leaf x !! 0 = x
Fork n l r !! k
 | k < m = l !! k
 | otherwise = r !! (k – m)
 where m = length l
```

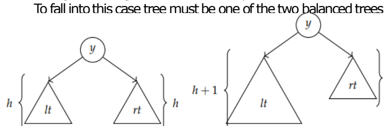
Balanced tree -> O(log n) time for as we recurse into one half.

9.4) Random Access Lists
The standard list representation models itself on Peano Numbers. Random Access Lists model the structure on **Binary Numbers** instead, which has its own benefits.
```
newtype RAList a = RAList [Tree a]
```
The RAList has the same complexity as a tree. e.g: !!:
```
(!!) :: RAList a → Int → a
RAList (t : ts) !! K
 | isEmpty t = RAList ts !! k
 | k < m = t !! K
 | otherwise = RAList ts !! (k − m)
where m = length t
```
Given xs :: RAList a, the cost of performing xs !! k is O(log k) in the worst case. The interesting operation is the cons function:
```
cons :: a → RAList a → RAList a
cons x xs = RAList (consTrees (Leaf x) xs)
 where
  consTrees :: Tree a → RAList a → [Tree a]
  consTrees t (RAList []) = [t]
  consTrees t (RAList (Tip:ts)) = t : ts
  consTrees t (RAList (t':ts)) = Tip :
            consTrees (fork t t')(RAList ts)
```
Notice that this follows the structure of the inc :: Binary → Binary function, benefiting from similar amortized complexity.

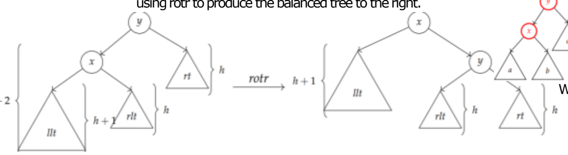
10) Searching
To search for an item in a structure, we must have some notion of **equality**:
```
class Eq a where
  (≡) :: a → a → Bool
```
Valid implementations of eq need to have an equality operator that behaves well - we need **reflexity, transitivity and antisymmetry**. The language doesn't enforce this, we must implement it validly.
Here's an example implementation of Eq:
```
data Fruit = Apple | Orange
instance Eq Fruit where
   Apple ≡ Apple = True
   Orange ≡ Orange = True
      _ ≡ _ = False
```
The simplest way to search a list is to just query the entire thing until we find the elem:
```
rummage x [ ] = False
rummage x (y : ys) = x ≡ y ∨ rummage x ys
```
We can improve on this by using a data structure with more order

10.1) Ordered Lists: The assumption that the elements can be ordered is recorded with an Ord constraint, which expects the (⩽) relation to be defined. The Ord class itself relies on the existence of Eq, so that the order (⩽) can be compatible with equality (≡).
```
class Eq a ⇒ Ord a where
  (⩽) :: a → a → Bool
```
We require that the ⩽ relation is a partial order – transitive, reflexive, antisymmetric (x ⩽ y ∧ y ⩽ x). We could implement a poset:
```
class Poset poset where
  toPoset :: Ord a ⇒ [a] → poset a
  fromPoset :: poset a → [a]
  empty :: poset a
  insert :: Ord a ⇒ a → poset a → poset a
  delete :: Ord a ⇒ a → poset a → poset a
  member :: Ord a ⇒ a → poset a → Bool
  union :: Ord a ⇒ poset a → poset a → poset a
  inter :: Ord a ⇒ poset a → poset a → poset a
```
We write member like so, but it has the same time complexity
```
member :: Ord a ⇒ a → [a] → Bool
member x [ ] = False
member x (y : ys) = x
   ≡ y ∨ (x < y ∧ member x ys)
```
10.2) Search Trees – Quicksort works by taking a pivot that partitions data into two parts. The structure of this recursion can be captured in this tree:
```
data Tree a = Nil | Node (Tree a) a (Tree a)
```
Constructing this tree is like the splitting step of quicksort:
```
instance Poset Tree where
toPoset :: Ord a ⇒ [a] → Tree a
toPoset [ ] = Nil toPoset (x : xs) = Node (toPoset us) x (toPoset vs)
  where (us, vs) = partition (⩽ x) xs
```
This allows faster access to elements when balanced.
```
member :: Ord a ⇒ a → Tree a → Bool
member x Nil = False
member x (Node l y rt)
 | x ≡ y = True
 | x < y = member x lt
 | otherwise = member x rt
```
If the tree is balanced, then its depth will force $T_{member}(n) \in O(\log(n))$, where n is the number of elements in the tree. The worst case is still linear though.

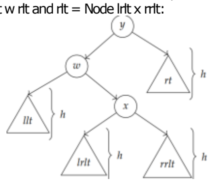
10.2.1) Binary Search (AVL) Trees - balanced trees by keeping track of height:
```
type Height = Int
data HTree a=Htip|HNode Height (HTree a) a (Htree a)
```
To make sure trees are constructed in the proper way where height is preserved, a smart constructor is used:
```
hnode :: HTree a → a → HTree a → HTree a
hnode lt x rt = HNode h lt x rt
   where h = (height lt ⊔ height rt) + 1
height :: HTree a → Int
height HTip = 0
height (HNode h lt x rt) = h
```
Insert is the difficult case, as we must maintain balancedness of our tree. We use the balancel and balancer smart constructors, to maintain the invariant:
1) **The difference in height between siblings is at most 1:**
```
instance Poset HTree where
insert :: Ord a ⇒ a → HTree a → HTree a
insert x HTip = hnode HTip x HTip
insert x t@(HNode lt y rt)
 | x ≡ y = t
 | x < y = balancel (insert x lt) y rt
 | otherwise = balancer lt y (insert x rt)
```
There are multiple cases to consider for balancel and balancer:
1) The height of lt and rt differ by at most 1 already. Focusing on the balanced case – inserting into the left tree, we only need compare height lt – height rt:
```
balancel :: HTree a → a → HTree a → HTree a
balancel lt y rt
 | height lt − height rt ⩽ 1 = hnode lt y rt
```
To fall into this case tree must be one of the two balanced trees



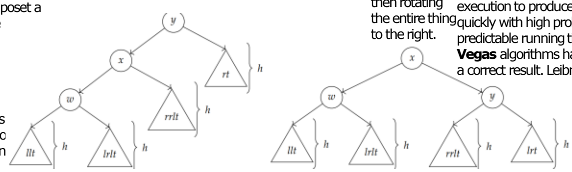
2) Assume the difference is exactly 2, this occurs if the initial tree has an imbalance of at most 1 to begin with. Inserting another element will increase the imbalance by 1 level. If height lt = height rt + 2 then further analysis on the subchildren of lt = Node llt x rlt yields some interesting cases.
Suppose that height llt > height rlt. Assuming that height rt = h, then height lt = h + 2, and height llt = h + 1. This is depicted by the left hand diagram below, and can be rotated to the right using rotr to produce the balanced tree to the right.



The final case is when height llt < height rlt. Assuming h = height rt it's possible to reason the tree is shaped as so, where lt = Node llt w rlt and rlt = Node lrlt x rrlt:



To balance this tree we need to place x as the root and have subtrees of equal height h+1. We do this by first rotating lt to the left, and then rotating the entire thing to the right.



These cases can be encoded by:
```
 | otherwise = case lt of HNode llt x rlt
   | height llt ⩾ height rlt→ rotr (hnode lt y rt)
   | otherwise → rotr (hnode (rotl lt) y rt)
```
All that is needed is a suitable definition of rotl and rotr, which can be done by simple pattern matching:
```
rotr :: HTree a → HTree a
rotr (HNode (HNode p x q) y r)
      = hnode p x (hnode q y r)
rotl :: HTree a → HTree a
rotl (HNode p x (HNode q y r))
      = hnode (hnode p x q) y r
```
The code for balancer follows a similar reasoning. rotr, rotl, balancel and balancer are all O(1), the insert function costs only O(log(n)) time, assuming a balanced tree, to create a balanced tree with an element inserted. This all works as our value size invariants are maintained.

11) Red Black Trees
Another way of creating balanced trees, they don't store height but rather the colour of a node - red/black.
```
data Colour = R | B
data RBTree a = E | N Colour (RBTree a) a (RBTree a)
```
We have **two invariants:**
1) **Every red node has a black parent node**
2) **Every path from the root node to a leaf must have the same number of black nodes**
These invariants enforce that the tree is imbalanced by at most a factor of 2 in one of the branches. Grants fast searching.

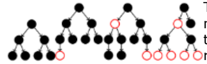


Figure 13.1: Valid Red-Black trees

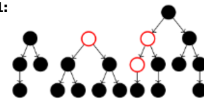
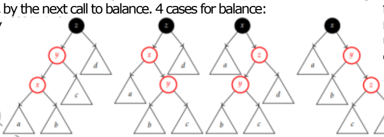
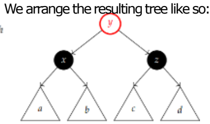


Figure 13.2: Invalid Red-Black trees

The insert x function inserts a new red leaf at the bottom of the tree that contains x. We recursively call the go function on the appropriate subtree until an empty node is found. Every node along the path to that leaf is balanced by applying the balance function. To ensure that the parent node is not red, the blacken function is applied to the final result.
```
instance Poset RBTree where I
nsert :: Ord a ⇒ a → RBTree a → RBTree a
insert x t = blacken (go t)
   where
     go :: RBTree a → RBTree a
     go E = N R E x E
     go t@(N c lt y rt)
       | x < y = balance c (go lt) y rt
       | x ≡ y = t
       | x > y = balance c lt y (go rt)
blacken :: RBTree a → RBTree a
blacken (N R lt x rt) = N B lt x rt
blacken t = t
```
Blacken is only ever applied to the result of go t, so it will only ever affect the root node, enforcing variant 1 for that node. The balance function balances the tree by ensuring no red nodes with red children. Assuming that the tree is valid to start with, the only new red node will have been inserted at one of the leaves. The innermost application of balance will be able to fix a potential red-red conflict, but may itself create a new red node that needs fixing by the next call to balance. 4 cases for balance:



We arrange the resulting tree like so:



We do this with pattern matching all 4 cases:
```
balance :: Colour → RBTree a → a → RBTree a → RBTree a
balance B (N R (NR a x b) y c) z d = N R (N B a x b) y (N B c z d)
balance B (N R a x (NR b y c)) z d = N R (N B a x b) y (N B c z d)
balance B a x (N R (N R b y c) z d) = N R (N B a x b) y (N B c z d)
balance B a x (N R b y (N R c z d)) = N R (N B a x b) y (N B c z d)
balance c lt x rt = N c lt x rt
```
Building an RBT from a list can be achieved by recursively inserting all the elements from the list into a tree. This costs O(n log n) but is O(n) if sorted list

12) Randomized Algorithms
– an algorithm that uses random values in its execution to produce a result. They're desirable when they produce results quickly with high probability. Two types: **Monte Carlo** algorithms have a predictable running time but unpredictably compute a correct result, and **Las Vegas** algorithms have an unpredictable running time, but predictably compute a correct result. Leibniz's law (identity of indiscernibles): x = y ⇒ f x = f y. This holds true for any function. Thus the idea for random value generation is to start with a seed value, from which a random value and a new seed can be extracted. Seed values have type StdGen and can be created with mkStdGen function:
```
import System.Random
mkStdGen :: Int → StdGen
```
Once created this can be passed into the random function.
```
random :: StdGen → (Int, StdGen)
```
We can easily produce a list of randoms by just calling random seed, and then doing x : randoms seed'. We can do other types by implementing this interface:
```
class Random a where
random :: StdGen → (a, StdGen)     randoms :: StdGen → [a]
randomR :: (a, a) → StdGen → (a, StdGen)
randomRs :: (a, a) → StdGen → [a]
```
Randomized Π – to compute pi, we can generate a random coordinate pair (range 0 to 1), and see if the Pythagorean distance from 0 is less than 1. If it is then we have a hit. We compute the num hits / sample size * 4, to get pi. (as the ratio of circlesize/squaresize = pi/4).
The important point to note here is that the variables seed, seed', and seed'' must be carefully scheduled to happen sequentially. Inside checks if we're inside the circle:
```
inside :: (Double, Double) → Bool
inside (x, y) = x × x + y × y ⩽ 1
montePi :: Double
montePi = loop (mkStdGen 42) samples 0
  where
    loop :: StdGen → Int → Int → Double
    loop seed 0 m = 4 × fromIntegral m / fromIntegral samples
    loop seed n m =
      let (x,seed') = randomR (0, 1) seed
          (y,seed'') = randomR (0, 1) seed'
          m' = if inside (x, y) then m + 1 else m
          n' = n − 1
      in loop seed'' n' m'
samples :: Int
samples = 10000
```
12.2) Sequencing Random Generators: Threading seeds around can be error prone, so instead we can handle seed gen automatically by using a context m. The key change in the following code is the use of the do keyword, which indicates that the following block of code is to be executed sequentially, one line at a time:
```
montePi' :: MonadRandom m ⇒ m Double
montePi' = loop 10000000 0
 where
  loop :: MonadRandom m ⇒ Int → Int → m Double
  loop 0 m = return (4 × fromIntegral m / fromIntegral samples)
  loop n m = do
   x ← getRandomR (0, 1)
   y ← getRandomR (0, 1)
   let m' = if inside (x, y) then m + 1 else m
     n' = n − 1
   loop n' m
```
The base case has a return, and the assignment of x and y is through special notation that indicates they're of a sequential operation getRandomR (0, 1):
```
getRandomR :: MonadRandom m ⇒ (Int, Int) → m Int
```
Its pure counterpart is randomR. Since we're wrapping a type into a higher level monad type, all random functions have a monad version:
```
class Monad m ⇒ MonadRandom m where
getRandom :: Random a ⇒ m a
getRandoms :: Random a ⇒ m [a]
getRandomR :: Random a ⇒ (a, a) → m a
getRandomRs :: Random a ⇒ (a, a) → m [a]
```
(remember a monad consists of a type constructor that takes a type as argument and returns a context for the monad computations which can involve side effects or state, a unit function – return or pure which takes a type and wraps it in the monad lifting it to the monadic context, and a bind function that takes a monad and transforms its inner value and returns a new monad – allowing for sequencing of computations.) A different approach is to use **12.3) Random Streams** - In this version, all of the random values are generated before being transformed into an appropriate sample. In this version, we use the **randomRs** function and all rand vals are generated before transformation into a sample:
```
montePi'' :: Double
montePi'' = 4 × fromIntegral (length (filter inside xys)) /
fromIntegral samples
  where xys = take samples (pairs (randomRs (0, 1)
              (mkStdGen 42) :: [Double]))
  pairs :: [a] → [(a, a)]
  pairs (x : y : xys) = (x, y) : pairs xys
```
This works nicely but the seed 42 is hard coded in – we can instead make it a parameter. Either that or we use a MonadRandom constraint:

15) Treaps
We can randomise so that the tree is balanced on construction, rather than carefully balance. **Treaps** let us do this – which are combos of binary trees and heaps. We have the **invariant** that values are stored in symmetric order – **the value on the left of a node is less than it and the value on the right is greater than it.** We also have the invariant that **parent nodes have a higher priority than their children:**
```
data Treap a = Empty | Node (Treap a) a Int (Treap a)
   deriving Show
```
Node l v p r holds a left child l, a value v, a priority p, and a right child r. This allows for an efficient member function, which recurses into the left or right depending on the value of x vs the comparing node.
insert inserts a value (with its priority) maintaining our invariants:
```
insert :: Ord a ⇒ a → Int → Treap a → Treap a
insert x p Empty = Node Empty x p Empty
insert x p (Node a y q b)
 | x < y = lnode (insert x p a) y q b
 | x ≡ y = Node a y q b
 | x > y = rnode a y q (insert x p b)
```
Smart constructors:
```
lnode :: Treap a → a → Int → Treap a → Treap a
lnode Empty y q c = Node Empty y q c
lnode l@(Node a x p b) y q c
 | q ⩽ p = Node l y q c -- = Node (Node a x p b) y q c
 | otherwise = Node a x p (Node b y q c)
rnode :: Treap a → a → Int → Treap a → Treap a
rnode a x p Empty = Node a x p Empty
rnode a x p r@(Node b y q c)
 | p ⩽ q = Node a x p r -- = Node a x p (Node b y q c)
 | otherwise = Node (Node a x p b) y q c
```
To delete :: Ord a ⇒ a → Treap a → Treap a a node, we follow much the same structure as insert, we call delete recursively, returning Empty in the Empty case, and using Node rather than lnode or rnode. We use merge in the recursive case but merge :: Treap a → Treap a → Treap a in the equals case: which merges two treaps by doing two easy base cases, and in the recursive case we make a new node and merge into the left or right by comparing on priority:
```
merge l@(Node a x p b) r@(Node c y q d)
 | p < q = Node a x p (merge b r)
 | otherwise = Node (merge l c) y q d
```
ToList can be implemented efficiently using function composition and an accumulating parameter. fromList can be done to but we need val-param pairs, if the priorities are randomly distributed, the tree is on average the resulting tree is balanced.

12.4) Randomised Treaps
Treaps have a specialised insert function which means they don't adhere to the standard interface. **Randomised Treaps** are treaps who's priorities are independent CRVs. These can be extracted from a standard RNGer, which is abstracted by the type StdGen: data RTreap a = RTreap StdGen (Treap a)
The StdGen is used to create a random variable whenever random:: StdGen → (Int, StdGen). insert' as such uses the RNGer to create a new priority for each insertion, and updates the generator in the structure accordingly:
```
insert' :: Ord a ⇒ a → RTreap a → RTreap a
insert' x (RTreap s t) = RTreap s' (insert x p t)
   where (p, seed') = random s
```
To instantiate a new randomized treap, we use the empty constructor with a deterministic seed:
```
empty' :: RTreap a
empty' = RTreap (mkStdGen 42) Empty
```
To allow for a different tree for each instance, we require a seed threaded through (rather than 42...)
```
fromList' :: Ord a ⇒ [a] → RTreap a
fromList' xs = foldr insert' empty' xs
toList' :: RTreap a → [a]
toList' (RTreap seed t) = toList t
```
12.5) Randomised Quicksort – has worstcase runtime O(nlogn). The function rquicksort below uses a fixed seed, since fromList' uses empty'.
```
rquicksort :: Ord a ⇒ [a] → [a]
rquicksort xs = toList' (fromList' xs)
```

13) Randomized Binary Search Trees
– behaves like a BST, but has a probability of having a value inserted at the root. Here's a normal BST below:
```
data BTree a = BNil | BNode (BTree a) a (BTree a)
```
We have two inserts, one is the normal one on BSTs, the other places an element at the root accordingly with rotations:
```
insertRoot :: Ord a ⇒ a → BTree a → BTree a
insertRoot x BNil = BNode BNil x BNil
insertRoot x (BNode l y r)
 | x < y = rotr (insertRoot x l) y r
 | x ≡ y = BNode l y r
 | x > y = rotl l y (insertRoot x r)
rotr :: BTree a → a → BTree a → BTree a
rotr (BNode a x b) y c = BNode a x (BNode b y c)
rotl :: BTree a → a → BTree a → BTree a
rotl a x (BNode b y c) = BNode (BNode a x b) y c
```
Here's a Randomized Binary Search Tree:
```
data RBTree a = RBTree StdGen Int (BTree a)
```
The empty RBTree simply instantiates the seed
```
empty :: RBTree a empty = RBTree (mkStdGen 42) 0 BNil
```
insert' inserts an element into a tree with n elements in the ordinary way, but with probability 1/n+1 will instead insert the value at the root.
```
insert' :: Ord a ⇒ a → RBTree a → RBTree a
insert' x (RBTree seed n t)
 | p ≡ 0 = RBTree seed' (n + 1) (insertRoot x t)
 | otherwise = RBTree seed' (n + 1) (insert x t)
 where
   (p,seed') = randomR (0, n) seed
```
This maintains balance with a very high probability, but only returns correct results when distinct elements inserted using this function at most once.

14) Mutable Datastructures
We could implement fibonacci using mutable state to keep track of the previous two fibs: A mutable value of type STRef s a holds a mutable reference to a that can be created, read, and modified with three primitive functions:
```
newSTRef :: a → ST s (STRef s a)
readSTRef :: STRef s a → ST s a
writeSTRef :: STRef s a → a → ST s ()
```
These operations all return values that work within an ST s context. Such values can only be extracted with the runST function:
```
runST :: (forall s ◦ ST s a) → a
```
The runST function processes a sequential computation, but remains a pure value by preventing any of its internal state from escaping to the outside world. Here's an imperative fib in Haskell. The dollar means a set of parenthesis wrapping our do block
```
fib' :: Int → Integer
fib' n = runST $ do
  rx ← newSTRef 0
  ry ← newSTRef 1
  let loop 0 = do
     x ← readSTRef rx
     return x
  loop n = do
     x ← readSTRef rx
     y ← readSTRef ry
     writeSTRef rx y
     writeSTRef ry (x + y)
     loop (n − 1)
  loop n
```
Sequence does a set of instructions. All this code sat inside a runST clause. Do indicates that the code that follows should be executed in sequence, one line after the other. The definition of loop reveals that in the base case the value x is returned. This is the value that is extracted by runST. Just as we had mutable references, we also have operations for mutable arrays:
```
newArray :: Ix i⇒(i, i)→a→ST s (STArray s i a)
readArray :: Ix i⇒STArray s i a→i→ST a
writeArray :: Ix i⇒STArray s i a→i→a→ST s
(STArray s i a)
```
We could implement a checklist:
```
checklist xs = runST $ do
  ays ← newArray (0, m − 1) False :: ST s
(STArray s Int Bool)
   sequence [writeArray ays x True
         | x ← xs, x < m]
   getElem s ays
where
   m = length xs
```
This particular algorithm relies on mutability. To indicate this the main body of the code is wrapped within runST, which indicates that the code that follows is to be executed sequentially. We use these array funcs:
```
newArray :: Ix i ⇒ (i, i) → a → ST s (STArray s i a)
readArray :: Ix i ⇒ STArray s i a → i → ST a
writeArray :: Ix i ⇒ STArray s i a → i → a → ST s
(STArray s i a)
```
14.1) Mutable Quicksort: Works in place w/o intermediate data structures. Swapping in place:
```
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
```
The mutable array axs is taken as a parameter, as well as two indices i and j. The variables x and y are read from the array, and a written into the swapped positions. The qsort function takes in a list xs and sets up the array axs with the appropriate size. This is then fed into the function aqsort which does the real work.
```
qsort :: Ord a ⇒ [a] → [a]
qsort xs = runST $ do
  axs ← newListArray (0, n) xs
  aqsort axs 0 n getElems axs
where n = length xs − 1
```
The fact that there is mutation happening is hidden from the rest of the system by wrapping sequential steps within a runST. The aqsort function is relatively simple: it takes in the array axs as an argument as well as the two i and j that indicate the range of values that should be sorted.
```
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
```
When i ⩾ j then this represents either a singleton or no value, in which case the work is complete. Otherwise, we call apartition which does the real work of partitioning the array between i and j. It returns some index k which indicates the index of the pivot that was chosen. The aqsort function is then recursively called on the two partitions on either side of k. The apartition function is used to work with the mutable array and perform the partitioning in place. Calling apartition axs p q will partition the values in the array axs between the indexes p and q, using the pivot at index p as the pivot.
```
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
```
Version which does the partitioning on ordinary lists. It does this differently, by first converting the list into an array, before calling apartition.
```
partition :: Ord a ⇒ [a] → [a]
partition [ ] = [ ]
partition xs = runST $ do
  axs ← newListArray (0, n) xs
  apartition axs 0 n
  getElems axs
where n = length xs − 1.
```