Algorithms

Evaluation

Cost model estimates time taken for instructions to be executed. Very general cost model counts number of reductions made.

***Evaluation Order* Applicative Order (strict setting)** leftmost innermost reducible expression (evaluates arguments before function) **Normal Order (lazy setting)** leftmost outermost reducible expression (evaluates function before its arguments). If they terminate both produce values in normal form.

If normal form for expression exists, normal order will always reduce to that normal form, but applicative order may not find that form as it may not terminate.

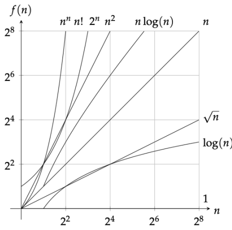
Counting Carefully

Non-primitive function: f a_1 a_2 … a_n = e => T(f) a_1 a_2 … a_n = T(e) + 1; ***Primitive function:*** f x_1 … x_n = 0 => T(f) x_1 … x_n = 0; ***Variable:*** x => T(x) = 0; ***Application:*** f e_1 … e_n => T(f e_1 … e_n) = T(f) + T(e_1) + … + T(e_n); ***Conditional:*** T(if p then e_1 else e_2) = T(p) + if p then T(e_1) else T(e_2).

Asymptotics

Logarithmico-exponential function (L-function) Real, positive, monotonic, one-valued function on real variable defined for all values greater than some definite value by finite combination of algebraic symbols, exponentials, logarithms, operating on real constants and variable.

[THEOREM] Any L-function f is ultimately continuous, of constant sign, monotonic, and as n → inf, the value f(n) tends to one of 0, inf, or some other definite limit.



Du-Bois Reymond Notation

$$f \prec g \iff \lim_{n\to\infty}\frac{f(n)}{g(n)} = 0 \qquad (3.1)$$

$$f \preccurlyeq g \iff \lim_{n\to\infty}\frac{f(n)}{g(n)} < \infty \qquad (3.2)$$

$$f \asymp g \iff 0 < \lim_{n\to\infty}\frac{f(n)}{g(n)} < \infty \qquad (3.3)$$

$$f \succcurlyeq g \iff \lim_{n\to\infty}\frac{f(n)}{g(n)} > 0 \qquad (3.4)$$

$$f \succ g \iff \lim_{n\to\infty}\frac{f(n)}{g(n)} = \infty \qquad (3.5)$$

Trichotomy Theorem mentioned above says one of f<g, f=g, f>g must hold forming a **trichotomy**.
Converse f≺g ⇔ f≻g; ***Transitive*** f≺g ∧ of g≺h => f≺h; f≼g ∧ g≼h => f≼h.
1 ≺ log n ≺ root(n) ≺ n ≺ n log n ≺ n^2 ≺ n^3 ≺ n! ≺ n^n.

Bachmann-Landau Notation

$$f \in o(g(n)) \iff f \prec g \qquad (3.9) \qquad o(g(n)) = \{\, f \mid \forall\delta > 0.\exists n_0 > 0.\forall n > n_0.f(n) < \delta g(n) \,\} \quad (3.14)$$

$$f \in O(g(n)) \iff f \preccurlyeq g \qquad (3.10) \qquad O(g(n)) = \{\, f \mid \exists\delta > 0.\exists n_0 > 0.\forall n > n_0.f(n) \leqslant \delta g(n) \,\} \quad (3.15)$$

$$f \in \Theta(g(n)) \iff f \asymp g \qquad (3.11) \qquad \Theta(g(n)) = O(g(n)) \cap \Omega(g(n)) \quad (3.16)$$

$$f \in \Omega(g(n)) \iff f \succcurlyeq g \qquad (3.12) \qquad \Omega(g(n)) = \{\, f \mid \exists\delta > 0.\exists n_0 > 0.\forall n > n_0.f(n) \geqslant \delta g(n) \,\} \quad (3.17)$$

$$f \in \omega(g(n)) \iff f \succ g \qquad (3.13) \qquad \omega(g(n)) = \{\, f \mid \forall\delta > 0.\exists n_0 > 0.\forall n > n_0.f(n) > \delta g(n) \,\} \quad (3.18)$$

Abstract Datatypes

```
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

tail = toList ∘ tail ∘ fromList
```

```
import Prelude hiding
  (head, tail, init, last, (!!), length, (++))
```

An isomorphism between two types a and b is given by a pair of functions $f :: a \to b$ and $g :: b \to a$ such that $f \circ g = id$ and $g \circ f = id$.

```
normalize :: list a → list a
normalize = fromList ∘ toList
length :: [a] → Int
length []       = 0
length (x : xs) = 1 + length xs
length :: [a] → Int
length = Prelude.length
```

```
empty :: list a
empty = fromList []
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])
```

Divide and Conquer

Three parts Divide a problem into subproblems, divide and conquer subproblems into sub solutions, conquer sub solutions into solution.

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
```

$$T_{msort}(0) = 1$$
$$T_{msort}(1) = 1$$
$$T_{msort}(n) = T_{length}(n) + T_{splitAt}(\tfrac{n}{2}) + T_{merge}(\tfrac{n}{2}) + 2 \times T_{msort}(\tfrac{n}{2})$$

Solving this recurrence gives $T_{msort}(n) \in \Theta(n \log n)$.

Quick Sort

```
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)
```

$$\begin{aligned} T_{qsort}(n) &= T_{partition}(n-1) + T_{++}(n-1) + T_{qsort}(n-1) + T_{qsort}(0) \\ &= c \times n + T_{qsort}(n-1) \\ &= c \times n + c \times (n-1) + \ldots + c \times 1 \\ &= c \times \frac{n \times (n+1)}{2} \\ &= c \times \frac{n^2 + n}{2} \end{aligned}$$

In other words, $T_{qsort}(n) \in O(n^2)$ in the worst case.

Dynamic Programming

Used to efficiently calculate exact solutions to certain recursive problems ('trade space for speed').
Memoization Storing result of function call so it can be used again later.
Strategy Write inefficient recursive algorithm that solves problem, improve efficiency by storing intermediate shared results.

Fibonacci example

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

fibs :: [Integer]
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)

fib3 :: Int → Integer
fib3 n = fibs !! n

fib2 :: Int → Integer
fib2 n = go n 0 1 where
  go 0 x y = x
  go n x y = go (n − 1) y (x + y)
```

Array – comes equipped with operation to look up values in constant time, Table, Tabulate

```
array :: Ix i ⇒ (i, i) → [(i, a)] → Array i a

table :: Int → Array Int Integer
table n = array (0, n) [(0, 0)
                       , (1, 1)
                       , (2, table ! 0 + table ! 1)
                       , (3, table ! 1 + table ! 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)]
```

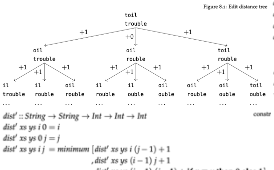
```
fib' :: Int → Integer
fib' n = table ! n
  where
    table :: Array Int Integer
    table = tabulate (0, n) memo
    memo 0 = 0
    memo 1 = 1
    memo n = table ! (n − 1) + table ! (n − 2)
```

Array function Builds array from list containing indices and their values, a!i returns value at index i and fails if i out of bounds; ***Table function*** Constructs table in array; ***Tabulate function*** Results of applying f to all values between x/y. Implemented as array so constant time access.

Edit-Distance

Levenshtein Distance between two strings: no of insertions, deletions, updates taken to turn one into other.

Figure 8.1: Edit distance tree




```
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)
```

constr

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

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
```

Bitonic Travelling Salesman

Travelling Salesman Problem Finds shortest possible route that visits set of cities exactly once and returns to starting city; ***Bitonic Travelling Salesman Problem*** Variation of TSP where salesman must visit cities in bitonic manner, first travelling along monotonic path and then reversing to return to starting city.

```
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]]

bitonic' :: (Int → Int → Double) → Int → Path
bitonic' δ 0 = Path 0 [(0, 0)]
bitonic' δ 1 = Path (2 × δ 0 1) [(0, 1), (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]]
  where
    δ' :: Int → Int → Path
    δ' i j = Path (δ i j) [(min i j, max i j)]
```

```
data Path = Path Double [(Int, Int)]
  deriving (Show, Eq, Ord)
instance Num Path where
  Path d1 ps1 + Path d2 ps2 = Path (d1 + d2) (ps1 ++ ps2)
  Path d1 ps1 − Path d2 ps2 = Path (d1 − d2) (ps1 \\ ps2)
  fromInteger 0 = Path 0 []
```

```
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)]
```

Amortized Analysis

Amortized analysis Gives cost of operation in context of sequence of previous operations on data structure.
Deque Double ended queue (symmetric list) is a queue where elements can be added both at front/back efficiently and is in this sense double ended. Deque contains xs and sy, which together form list with all elements in xs followed by reversed elements in sy.

```
data Deque a = Deque [a] [a]

instance List Deque where
  toList :: Deque a → [a]
  toList (Deque xs sy) = xs ++ reverse sy
isEmpty xs ⇒ isEmpty sy ∨ isSingle sy
isEmpty sy ⇒ isEmpty xs ∨ isSingle xs

fromListNaive :: [a] → Deque a
fromListNaive xs = Deque xs []

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

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
```

Amortization
***Three things to define* Cost function** $C_{op\_i}(xs_i)$ for each operation op_i on data xs_i; **Amortized cost function** $A_{op\_i}(xs_i)$ for each operation op_i on data xs_i; **Size function** S(xs) that calculates sizes of data xs.

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

Furthermore, when $S(xs_0) = 0$, then this implies:

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

$$C_{cons}(xs) = 1 \qquad C_{snoc}(xs) = 1$$
$$C_{head}(xs) = 1 \qquad C_{last}(xs) = 1$$

The *tail* function is a more expensive operation: when *xs* is a si gleton list it will cost as many steps as there are elements in the reversed:

$$C_{tail}(Deque\ xs\ sy) = \textbf{if } length\ xs > 1 \textbf{ then } 1 \textbf{ else } length\ sy$$

Now a simple amortized cost is given to all the operations.

$$A_{op}(xs) = 2$$

This cost is obviously higher than the real cost of some operations, and lower than the real cost of others.
Finally, a size function is assigned to the data:

$$S(Deque\ xs\ sy) = |length\ xs - length\ sy|$$

With these pieces in place, it is easy to verify that Equation 10.1 holds. Consider the situation $Deque\ xs'\ sy' = tail\ (Deque\ xs\ sy)$, where $length\ sy = k$. In the worst case, when *xs* is a singleton list, this implies that:

$$S(Deque\ xs\ sy) = k - 1$$
$$S(Deque\ xs'\ sy') = 1$$

So, substituting into Equation 10.1 this results in:

$$C_{tail}(Deque\ xs\ sy) \leqslant A_{tail}(Deque\ xs\ sy) + S(Deque\ xs\ sy) - S(Deque\ xs'\ sy')$$
$$\iff$$
$$k \leqslant 2 + (k-1) - 1$$

This is clearly true. Therefore the time complexity of these instructions is bounded by $O(n)$, and the amortized cost of *tail* is $O(1)$.

Random Access Lists

Peano Numbers Simplistic way of counting natural numbers, number is either 0 or succ of some other number.
Binary Numbers List of digits with LSB first representation, e.g., [O, O, I, I] = 12.

```
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)
```

```
type Binary = [Digit]
data Digit = O | I
  deriving Eq
inc :: Binary → Binary
inc []       = [I]
inc (O : bs) = I : bs
inc (I : bs) = O : (inc bs)
```

$$C_{inc}(bs) = t + 1 \textbf{ where } t = length\ (takeWhile\ (\equiv I)\ bs)$$
$$A_{inc}(bs) = 2$$
$$S_{inc}(bs) = b \textbf{ where } b = length\ (filter\ (\equiv I)\ bs)$$

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

Binary Tree Lookup Balanced binary trees allow efficient access to their elements.
Random Access Lists Efficient data structures that combine benefits of lists/binary numbers, allowing for quick random access/modification operations.

```
data Tree a = Tip | Leaf a | Fork Int (Tree a) (Tree a)
fork :: Tree a → Tree a → Tree a
fork l r = Fork (length l + length r) l r

instance List Tree where
  toList :: Tree a → [a]
  toList (Tip)        = []
  toList (Leaf x)     = [x]
  toList (Fork n l r) = toList l ++ toList r
length :: Tree a → Int
length (Tip)        = 0
length (Leaf x)     = 1
length (Fork n l r) = n
```

```
(!!) :: 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
newtype RAList a = RAList [Tree a]
instance List RAList where
  toList :: RAList a → [a]
  toList (RAList ts) = (concat ∘ map toList) ts
```

```
(!!) :: 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
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)
```

Searching

Equality

$$x \equiv x \qquad \text{(reflexivity)}$$
$$x \equiv y \Leftrightarrow y \equiv x \qquad \text{(symmetry)}$$
$$x \equiv y \wedge y \equiv z \Rightarrow x \equiv z \qquad \text{(transitivity)}$$

```
data Fruit = Apple | Orange
instance Eq Fruit where
  Apple  ≡ Apple  = True
  Orange ≡ Orange = True
  _      ≡ _      = False
```

```
rummage :: Eq a ⇒ a → [a] → Bool
rummage x [] = False
rummage x (y : ys)
  | x ≡ y     = True
  | otherwise = rummage x ys
```

Ordering

$$x \leqslant x \qquad \text{(reflexivity)}$$
$$x \leqslant y \wedge y \leqslant z \Rightarrow x \leqslant z \qquad \text{(transitivity)}$$
$$x \leqslant y \wedge y \leqslant x \Rightarrow x \equiv y \qquad \text{(antisymmetry)}$$
$$x \leqslant y \vee y \leqslant x \qquad \text{(connexity)}$$

```
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
```

```
instance Poset [] where
  member :: Ord a ⇒ a → [a] → Bool
  member x [] = False
  member x (y : ys)
    | x ≡ y     = True
    | x < y     = member x ys
    | otherwise = False
```

Search Trees

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

```
member :: Ord a ⇒ a → Tree a → Bool
member x Nil = False
member x (Node lt y rt)
  | x ≡ y     = True
  | x < y     = member x lt
  | otherwise = member x rt
```

```
type Height = Int
data HTree a = HTip
             | HNode Height (HTree a) a (HTree a)
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
```

```
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)
  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
```

```
balancel :: HTree a → a → HTree a → HTree a
balancel lt y rt
  | height lt − height rt ⩽ 1 = hnode lt y rt
  | otherwise = case lt of
      HNode _ llt x rlt | height llt ⩾ height rlt → rotr (hnode lt y rt)
                        | otherwise               → rotr (hnode (rotl lt) y rt)
```

Red-Black Trees

They are another means of creating balanced trees. They do not need to store height of current tree (unlike AVL/Binary Search Tree) instead stores colour of node: red/black.

Two invariants: Every red node must have black parent node; Every path from root to leaf must have same number of black nodes.

Ensures tree is at most imbalanced by factor of at most two in one of its branches.

```
data Colour = R | B
data RBTree a = E
             | N Colour (RBTree a) a (RBTree a)
instance Poset RBTree where
  insert :: 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

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
```

Randomized Algorithms

Randomized algorithm Produce results quickly and with high probability (mostly correct) using random values. **Monte Carlo** Predictable running time/unpredictable correct result; **Las Vegas** Opposite.

Leibniz's Law/Identity of indiscernibles Functions always map same inputs to same outputs (x=y => fx=fy).

No function can return truly random result due to Leibniz's law but can exhibit pseudo-random behaviour (Depends on some input that varies either explicitly/implicitly***).***

```
mkStdGen :: Int → StdGen
randoms :: StdGen → [Int]
randoms seed = x : randoms seed′
  where (x, seed′) = random seed
inside :: (Double, Double) → Bool
inside (x, y) = x × x + y × y ⩽ 1
montePi′ :: MonadRandom m ⇒ m Double
montePi′ = loop samples 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′
getRandomR :: MonadRandom m ⇒ (Int, Int) → m Int
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]
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
montePi‴ :: MonadRandom m ⇒ m Double
montePi‴ = do
  rxys ← getRandomRs (0,1)
  let xys = take samples (pairs (rxys))
  return (4 × fromIntegral (length (filter inside xys)) / fromIntegral samples)

random :: StdGen → (Int, StdGen)
class Random a where
  random  :: StdGen → (a, StdGen)
  randoms :: StdGen → [a]
  randomR  :: (a, a) → StdGen → (a, StdGen)
  randomRs :: (a, a) → StdGen → [a]
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
evalRand :: Rand StdGen a → StdGen → a
```

Treaps

Treap Combination of tree and heap, all values to the left are smaller and values to the right are larger, equally parent node has higher priority in heap than children.

```
data Treap a = Empty | Node (Treap a) a Int (Treap a)
  deriving Show
member :: Ord a ⇒ a → Treap a → Bool
member x Empty = False
member x (Node a y _ b)
  | x < y = member x a
  | x ≡ y = True
  | x > y = member x b
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)
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

delete :: Ord a ⇒ a → Treap a → Treap a
delete x Empty = Empty
delete x (Node a y q b)
  | x < y = Node (delete x a) y q b
  | x ≡ y = merge a b
  | x > y = Node a y q (delete x b)
merge :: Treap a → Treap a → Treap a
merge Empty r = r
merge l Empty = l
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 Empty = []
toList (Node a x p b) = toList a ++ [x] ++ toList b
toList :: Treap a → [a]
toList t = toList′ t []
  where
    toList′ :: Treap a → [a] → [a]
    toList′ Empty xs = xs
    toList′ (Node a x p b) xs = toList′ a (x : (toList′ b xs))
fromList :: Ord a ⇒ [a] → Treap a
fromList xs = foldr (uncurry insert) Empty (zip xs (randoms seed))
  where seed = mkStdGen 42
```

Randomized Treaps

```
data RTreap a = RTreap StdGen (Treap a)
insert′ :: Ord a ⇒ a → RTreap a → RTreap a
insert′ x (RTreap seed t) = RTreap seed′ (insert x p t)
  where (p, seed′) = random seed
empty′ :: RTreap a
empty′ = RTreap (mkStdGen 42) Empty
empty″ :: StdGen → RTreap a
empty″ seed = RTreap seed Empty

fromList′ :: Ord a ⇒ [a] → RTreap a
fromList′ xs = foldr insert′ empty′ xs
toList′ :: RTreap a → [a]
toList′ (RTreap seed t) = toList t
rquicksort :: Ord a ⇒ [a] → [a]
rquicksort xs = toList′ (fromList′ xs)
```

Randomized Binary Search Trees

Behaves like ordinary binary search tree most of the time, but with some probability will insert a value at its root. Underlying data type is ordinary binary tree.

```
data BTree a = BNil
             | BNode (BTree a) a (BTree a)
insert :: Ord a ⇒ a → BTree a → BTree a
insert x BNil = BNode BNil x BNil
insert x (BNode l y r)
  | x < y = BNode (insert x l) y r
  | x ≡ y = BNode l y r
  | x > y = BNode l y (insert x r)
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
data RBTree a = RBTree StdGen Int (BTree a)
empty :: RBTree a
empty = RBTree (mkStdGen 42) 0 BNil
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
```

Mutable Data Structures

Mutable References

```
fib :: Int → Integer
fib n = loop n 0 1
  where
    loop 0 x y = x
    loop n x y = loop (n − 1) y (x + y)

newSTRef   :: a → ST s (STRef s a)
readSTREf  :: STRef s a → ST s a
writeSTRef :: STRef s a → a → ST s ()
runST :: (forall s ∘ ST s a) → a

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
```

Checklist

```
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)
minfree :: [Int] → Int
minfree xs = head ([0..] \\ xs)
(\\) :: Eq a ⇒ [a] → [a] → [a]
us \\ vs = filter (¬ ∘ flip elem vs) us
minfree′ :: [Int] → Int
minfree′ xs = length (takeWhile id (checklist xs))
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]
    getElems ays
  where
    m = length xs
```

Useful Haskell Functions

```
fst, snd        :: (a,b) -> a
fst (x, _)      = x
snd (_, y)      = y

id              :: a -> a
id    x         = x

(.)             :: (b -> c) -> (a -> b) -> (a -> c)
(f . g) x       = f (g x)

head, last      :: [a] -> a
head (x:_)      = x

last [x]        = x
last (_:xs)     = last xs

tail, init      :: [a] -> [a]
tail (_:xs)     = xs

init [x]        = []
init (x:xs)     = x : init xs

null            :: [a] -> Bool
null []         = True
null (_:_)      = False

(++)            :: [a] -> [a] -> [a]
[]     ++ ys    = ys
(x:xs) ++ ys    = x : (xs ++ ys)

map             :: (a -> b) -> [a] -> [b]
map f [ ]       = [ ]
map f (x:xs)    = f x : map f xs

filter          :: (a -> Bool) -> [a] -> [a]
filter _ []     = []
filter p (x:xs)
      | p x        = x : filter p xs
      | otherwise  = filter p xs

concat          :: [[a]] -> [a]
concat          = foldr (++) []

length          :: [a] -> Int
length          = foldl (\x _ ->x+1) 0

(!!)            :: [a] -> Int -> a
(x:_)  !! 0     = x
(_:xs) !! n | n>0 = xs !! (n-1)
(_:_)  !! _     = error "Prelude.!!: negative index"
[]     !! _     = error "Prelude.!!: index too large"

and = ...
or = ...

ord :: Char -> Int
chr :: Int -> Char
toUpper, toLower :: Char -> Char
isAscii, isDigit :: Char -> Bool
isUpper, isLower :: Char -> Bool

foldl           :: (a -> b -> a) -> a -> [b] -> a
foldl f z []    = z
foldl f z (x:xs) = foldl f (f z x) xs

foldr           :: (a -> b -> b) -> b -> [a] -> b
foldr f z []    = z
foldr f z (x:xs) = f x (foldr f z xs)

iterate         :: (a -> a) -> a -> [a]
iterate f x     = x : iterate f (f x)

take             :: Int -> [a] -> [a]
take n _ | n <= 0 = []
take _ []        = []
take n (x:xs)    = x : take (n-1) xs

drop             :: Int -> [a] -> [a]
drop n xs | n <= 0 = xs
drop _ []        = []
drop n (_:xs)    = drop (n-1) xs

zip             :: [a] -> [b] -> [(a,b)]
zip             = zipWith (\a b -> (a,b))

zipWith                  :: (a->b->c) -> [a]->[b]->[c]
zipWith z (a:as) (b:bs)  = z a b : zipWith z as bs
zipWith _ _ _            = []

takeWhile :: (a->Bool) -> [a] -> [a]
takeWhile p [ ] = [ ]
takeWhile p (x:xs)
      | p x        = x : takeWhile p xs
      | otherwise  = [ ]

dropWhile :: (a->Bool) -> [a] -> [a]
dropWhile p [ ] = [ ]
dropWhile p (x:xs)
      | p x        = dropWhile p xs
      | otherwise  = x:xs

flip :: (a -> b -> c) -> b -> a -> c
flip f x y = f y x

until :: (a -> Bool) -> (a -> a) -> a -> a
until p f x   = if p x then x else until p f (f x)

sort :: Ord a => [a] -> [a]
sort xs = ...

and, or :: [Bool] -> Bool
```

Master Theorem

T(n) = aT(n/b) + f(n) has solutions as follows, where E = log a/ log b is the critical exponent: (1) If $n^{E+\varepsilon} = O(f(n))$ for some $\varepsilon > 0$ then $T(n) = \Theta(f(n))$. (2) If $f(n) = \Theta(n^E)$ then $T(n) = \Theta(f(n)\log n)$. (3) If $f(n) = O(n^{E-\varepsilon})$ for some $\varepsilon > 0$ then $T(n) = \Theta(n^E)$.

Examples: Binary Search, MergeSort, Strassen's Algorithm

$$W(n) = W(n/2) + 1$$

Here $a = 1$ and $b = 2$ and $f(n) = \Theta(n^0)$.

Then $E = \log a/\log b = 0$. So

$$W(n) = \Theta(n^0 \log n) = \Theta(\log n)$$

$$W(n) = 2W(n/2) + (n-1)$$

Here $a = 2$ and $b = 2$ and $f(n) = \Theta(n^1)$.

Then $E = \log a/\log b = 1$. So

$$W(n) = \Theta(n \log n)$$

$$A(n) = 7A(n/2) + 18(n/2)^2$$

Here $a = 7$ and $b = 2$, $f(n) = \Theta(n^2)$.

Then $E = \log a/\log b = \log 7 > 2$. So

$$A(n) = \Theta(n^{\log 7})$$

Array Resizing (Mutable Data Structures Continuation)

```
data ArrayList s a = ArrayList (STRef s Int) (STRef s Int) (STRef s (STArray s Int a))
newArray_ :: Ix i ⇒ (i, i) → ST s (STArray s i a)
empty :: ST s (ArrayList s a)
empty = do pn   ← newSTRef 0
           pm   ← newSTRef m
           axs  ← newArray_ (0, m − 1)
           paxs ← newSTRef axs
           return (ArrayList pn pm paxs)
  where m = 8
toList :: ArrayList s a → ST s [a]
toList (ArrayList rn rm raxs) = do
  n ← readSTRef rn
  m ← readSTRef rm
  axs ← readSTRef raxs
  sequence [readArray axs i | i ← [m − n .. m − 1]]

insert :: a → (ArrayList s a) → ST s ()
insert x (ArrayList pn pm paxs) = do
  n   ← readSTRef pn
  m   ← readSTRef pm
  axs ← readSTRef paxs
  writeSTRef pn (n + 1)
  if n < m
    then do
      writeArray axs (m − n − 1) x
    else do
      let m′ = 2 × m
  writeSTRef pm m′
  axs′ ← newArray_ (0, m′ − 1)
  writeSTRef paxs axs′
  sequence [do x′ ← readArray axs i
               writeArray axs′ (m + i) x′
    | i ← [0 .. m − 1]]
  writeArray axs′ (m − 1) x

reverse :: [Int] → [Int]
reverse xs = runST $ do
  pxs ← empty
  sequence [insert x pxs | x ← xs]
  toList pxs
```

Mutable Data Structures Continuation

Hashing

```
class Hashable a where
  hash :: a → Int
nub :: (Hashable a, Eq a) ⇒ [a] → [a]
nub xs = concat (runST $ do
  axss ← newListArray (0,255) (replicate 256 []) :: ST s (STArray s Int [a])
  sequence [do let hx = hash x ‘mod‘ 255
               xs ← readArray axss hx
               unless (x ∈ xs) $ do
                 writeArray axss hx (x : xs)
           | x ← xs]
  getElems axss)
```

1s

Quicksort

```
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
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

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
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

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
```