

## 1 Evaluation

**Definition 1.1 (Counting)** To evaluate  $T(e)$ :

- **Application**  $T(f\ e_1 \dots e_n) = T(f)\ e_1 \dots e_n + T(e_1) + \dots + T(e_n)$
- **Variable**  $T(x) = 0$
- **Primitives** For primitive function  $f$ ,  $T(f)\ x_1 \dots x_n = 0$
- **Conditional**  $T(\text{if } p \text{ then } e_1 \text{ else } e_2) = T(p) + \text{if } p \text{ then } T(e_1) \text{ else } T(e_2)$

## 2 Asymptotics

**Definition 2.1 (L-function)** is a real, positive, monotonic, continuous function where as  $n \rightarrow \infty$ ,  $f(n) \rightarrow$  one of  $0, \infty$  or some value  $k$ .

**Definition 2.2 (Du Bois-Reymond and Bachman-Landau Notation)** For L-functions  $f$  and  $g$ ,

- $f \prec g \Leftrightarrow \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0 \Leftrightarrow f \in o(g(n))$  where  $o(g(n)) = \{f | \forall \delta > 0. \exists n_0 > 0. \forall n > n_0. f(n) < \delta g(n)\}$
- $f \preceq g \Leftrightarrow \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} < \infty \Leftrightarrow f \in O(g(n))$  where  $O(g(n)) = \{f | \exists \delta > 0. \exists n_0 > 0. \forall n > n_0. f(n) \leq \delta g(n)\}$
- $f \asymp g \Leftrightarrow 0 < \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} < \infty \Leftrightarrow f \in \Theta(g(n))$  where  $\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$
- $f \succeq g \Leftrightarrow \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} > 0 \Leftrightarrow f \in \Omega(g(n))$  where  $\Omega(g(n)) = \{f | \exists \delta > 0. \exists n_0 > 0. \forall n > n_0. f(n) \geq \delta g(n)\}$
- $f \succ g \Leftrightarrow \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty \Leftrightarrow f \in \omega(g(n))$  where  $\omega(g(n)) = \{f | \forall \delta > 0. \exists n_0 > 0. \forall n > n_0. f(n) > \delta g(n)\}$

## 3 Data Structures

$\text{foldl } (\diamond) \epsilon$  is extensionally equivalent to  $\text{foldr } (\diamond) \epsilon$  when  $x \diamond (y \diamond z) = (x \diamond y) \diamond z$ ,  $\epsilon \diamond y = y$  and  $x \diamond \epsilon = x$

**Definition 3.1 (Boom Hierachy)**

1. **Associativity**  $(x \cup y) \cup z = x \cup (y \cup z)$  Trees
2. **Identity**  $\emptyset \cup x = x = x \cup \emptyset$  Trees, Lists
3. **Commutativity**  $x \cup y = y \cup x$  Trees, Lists, Bags
4. **Idempotence**  $x \cup x = x$  Trees, Lists, Bags, Sets

**Definition 3.2 (Member)** Using only  $\log(x)+1$  comparisons

```
member x HTip = False
member x (HNode _ lte p gt) | x <= p = go p x lte
                             | otherwise = member x gt
go y x HTip = y <= x
go y x (HNode _ lte p gt) | x <= p = go p x lte
                             | otherwise = go y x gt
```

**Definition 3.3 (Difference Lists)**  $(DList\ fxs) ++ (DList\ fys = DList)\ (fxs.fys)$ ;  $\text{cons } x\ (DList\ fxs) = DList\ ((x:).fxs)$ ;  $\text{empty} = DList\ \text{id}$ ;  $\text{toList } (DList\ fxs) = fxs []$ ;  $\text{fromList } xs = DList\ (xs++)$

**Definition 3.4 (Deque)** data Deque  $a = \text{Deque } [a]\ [a]$ ;  $\text{toList } (\text{Deque } xs\ sy) = xs ++ \text{reverse } sy$ ;  $\text{fromList } xs = \text{Deque } ys\ (\text{reverse } zs)$  where  $(ys, zs) = \text{splitAt } (\text{length } xs \div 2)\ xs$ . ( $\text{isEmpty } xs \Rightarrow \text{isEmpty } sy$  or  $\text{isSingle } sy$  and  $\text{isEmpty } sy \Rightarrow \text{isEmpty } xs$  or  $\text{isSingle } xs$ )

**Definition 3.5 (Random Access List)**  $\text{toList } (RAList\ ts) = (\text{concat} . \text{map } \text{toList})\ ts$

```
cons x xs = RAList (consTree (Leaf x) xs) where
  consTree t (RAList []) = [t]
  consTree t (RAList (Tip:ts)) = t:ts
  consTree t (RAList (t':ts))
    = Tip:consTree (fork t t') (RAList ts)
  RAList (t:ts) !! k
    | isEmpty t = RAList k
    | k < length t = t !! k
    | otherwise = RAList ts !! (k-m)
```

**Definition 3.6 (AVL Tree)** *data HTree a = HTip | HNode Height (HTree a) a (HTree a)*

```

insert x HTip = hnode HTip x HTip          balancel lt y rt
insert x t@(HNode _ lt y rt)                | height lt - height rt <= 1 = hnode lt y rt
      | x == y = t                          | otherwise = case lt of HNode _ llt x rlt
      | x < y = balancel (insert x lt) y rt  | height llt >= height rlt -> rotr (hnode lt y rt)
      | otherwise = balancer lt y (insert x rt) | otherwise -> rotr (hnode (rotr lt) y rt)

rotr (HNode _ (HNode _ p x q) y r) = hnode p x (hnode q y r)
rotr (HNode _ p x (HNode _ q y r)) = hnode (hnode p x q) y r

fromOrdList xs = fst (go (length xs) xs where)
  go 0 xs = (HTip, xs)
  go n xs = let m = n `div` 2
              (lhs, x:xs') = go m xs
              (rhs, xs'') = go (n-m-1) xs'
              in (HNode (1+max [height lhs, height rhs]) lhs x rhs, xs'')

hTreeToList rt = go rt [] where
  go HTip ks = ks
  go (HNode _ lt p gt) ks = go lt (p:go gt ks)

insert x DTip = (True, DNode Zero DTip x DTip)
insert x (DNode bl lt y rt) = case compare x y of
  EQ -> (False, DNode bl lt y rt)
  LT -> case insert x lt of
    (False, lt') -> (False, DNode bl lt' y rt)
    (True, lt') -> case bl of
      MinusOne -> (False, DNode Zero lt' y rt)
      Zero -> (True, DNode PlusOne lt' y rt)
      PlusOne -> rotr lt' y rt
  rotr (DNode PlusOne a y b) x c = (False, DNode Zero a y (DNode Zero b x c))
  rotr (DNode Zero a y b) x c = (True, DNode MinusOne a y (DNode PlusOne b x c))
  rotr (DNode MinusOne a y (DNode bl b z c)) x d =
    (False, DNode Zero (DNode (balr bl) a y b) z (DNode (ball bl) c x d))
  balr PlusOne = Zero; balr Zero = Zero; balr MinusOne = PlusOne
  ball PlusOne = MinusOne; ball Zero = Zero; ball MinusOne = Zero

```

**Definition 3.7 (Red Black Tree)** *data RBTREE a = E | N Colour (RBTREE a) a (RBTREE a). Every red node must have a black parent and every path from root to leaf must have same number of black nodes.*

```

insert x t = blacken (go t) where
  go E = N R E x E
  go t@(N c lt y rt)                blacken (N R lt x rt) = N B lt x rt
      | x < y = balance c (go lt) y rt  blacken t = t
      | x == y = t
      | otherwise = balance c lt y (go rt)

balance B (N R (N R 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 (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 (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

```

**Definition 3.8 (Treap)** *data Treap a = Empty | Node (Treap a) a Int (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 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
  | otherwise = Node a x p (Node b y q c)

```

**Definition 3.9 (Tries)** *data Trie a = (Bool, [a, Trie a])*

```

insert [] (e, ts) = (True, ts)
insert (x:xs) (e, ts) = e, ins ts where

```

```

ins [] = [(x, insert xs (False, []))]
ins ((y,ys):ts) | x == y = (y, insert xs yt):ts
                  | otherwise = (y,yt): ins ts

```

## 4 Divide & Conquer

1. Divide a problem into subproblems.
2. Divide and conquer subproblems into subsolutions.
3. Conquer subsolutions into a solution.

**Algorithm 4.1 (Merge Sort)**  $msort [] = []$ ;  $msort [x] = [x]$ ;  $msort xs = merge (msort us) (msort vs)$  where  $(us,vs) = splitAt ((length xs) \div 2) xs$ ;  $merge (x:xs) (y:ys) = \text{if } x \leq y \text{ then } x : merge xs (y:ys) \text{ else } y : merge (x:xs) ys$ ;  $merge xs [] = xs$ ;  $merge [] ys = ys$ ;

*Worst Case:*  $n \log n$ .  $T_{msort}(n) = T_{length}(n) + T_{splitAt}(\frac{n}{2}) + T_{merge}(\frac{n}{2}) + 2 \times T_{msort} \frac{n}{2}$

**Algorithm 4.2 (Quicksort)** *Worst Case:*  $n^2$ .  $qsort [] = []$ ;  $qsort [x] = [x]$ ;  $qsort (x:xs) = qsort us ++ [x] ++ qsort vs$  where  $(us,vs) = partition (\leq x) xs$ ;  $partition p xs = (filter p xs, filter (not p) xs)$

## 5 Dynamic Programming

1. Write an inefficient recursive algorithm that solves the problem.
2. Improve inefficiency by storing intermediate shared results.

```

tabulate :: Ix i => (i,i) -> (i->a) -> Array i a
tabulate (u,v) f = array (u,v) [(i, f i) | i <- range(u,v)]

```

**Algorithm 5.1 (Fibonacci Numbers)**  $fib' n = table ! n$  where  
 $table = tabulate (0,n) memo :: Array Int Integer$   
 $memo 0 = 0$ ;  $memo 1 = 1$ ;  $memo n = table ! (n-1) + table ! (n-2)$

**Algorithm 5.2 (Edit Distance)**  $dist xs ys = table ! (m,n)$  where  
 $table = tabulate ((0,0),(m,n)) (uncurry memo)$   
 $memo i 0 = i$ ;  $memo 0 j = j$ ;  
 $memo i j = \text{minimum } [table ! (i,j-1) + 1, table ! (i-1,j) + 1,$   
 $\quad \quad \quad table ! (i-1,j-1) + \text{if } (axs!(m-i)) == (ays!(n-j)) \text{ then } 0 \text{ else } 1]$   
 $m = \text{length } xs$ ;  $n = \text{length } ys$ ;  $axs = \text{fromList } xs$ ;  $ays = \text{fromList } ys$

**Algorithm 5.3 (Bitonic Travelling Salesman)**  $bitonic d n = table ! n$  where  
 $table = tabulate (0,n) memo$   
 $memo 0 = Path 0 [(0,0)]$ ;  $memo 1 = Path (2 * d 0 1) [(0,1),(0,1)]$   
 $memo n = \text{minimum } [table ! k - d' (k-1) k + d' (k-1) n + \text{sum}[d' i (i+1) | i <- [k..n-1]]$   
 $\quad \quad \quad | k <- [1..n-1]]$   
 $\text{where } d' i j = Path (d i j) [(min i j, max i j)]$

## 6 Amortized Analysis

**Definition 6.1 (Amortization)**  $C_{op_i}(xs_i) \leq A_{op_i}(xs_i) + S(xs_i) - S(xs_{i+1})$

- A cost function  $C_{op_i}(xs_i)$  for each operation  $op_i$  on data  $xs_i$ .
- An amortized cost function  $A_{op_i}(xs_i)$  for each operation  $op_i$  on data  $xs_i$ .
- A size function  $S(xs)$  that calculates the size of data  $xs$ .  
 If  $S(xs_0) = 0$  then  $\sum_{i=0}^{n-1} C_{op_i}(xs_i) \leq \sum_{i=0}^{n-1} A_{op_i}(xs_i)$

## 7 Randomized Algorithm

Monte Carlo algorithms have predictable run time but unpredictable results. Las Vegas algorithms vice versa.

**Algorithm 7.1 (Randomised Pi)** montePi = loop (mkStdGen 42) 1000 0 where  
 loop seed 0 m = 4 \* fromIntegral m / fromIntegral 1000  
 loop seed n m = let (x,seed') = randomR (0,1) seed  
                   (y,seed'') = randomR (0,1) seed'  
                   in loop seed'' (n-1) (if inside (x,y) then m+1 else m)

**Algorithm 7.2 (Randomised Treap)** data RTreap a = RTreap StdGen (Treap a)

insert' x (RTreap seed t) = RTreap seed' (insert x p t) where (p,seed') = random seed  
 rquicksort xs = toList' (fromList' xs)

## 8 Mutable Data Structures

To get STArray from list,  $\text{axs} \leftarrow \text{newListArray } (0, \text{length } \text{xs} - 1) \text{ xs}$

**Algorithm 8.1 (Checklist)** minfree xs = length (takeWhile id (checklist xs))  
 checklist xs = runST \$ do  
   ays <- newArray (0,length xs - 1) False :: ST s (STArray s Int Bool)  
   sequence [writeArray ays x True | x <- xs, x < length xs]  
   getElem ays

**Algorithm 8.2 (Quicksort)** qsort xs = runST \$ do  
 axs <- newListArray (0,(length xs - 1)) xs  
 aqsort axs 0 (length xs - 1)  
 getElem axs  
 aqsort axs i j | i >= j = return ()  
                   | otherwise = do  
     k <- apartition axs i j  
     aqsort axs i (k-1)  
     aqsort (k+1) j  
 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 loop (i+1) j  
         else do swap axs i j  
               loop i (j-1)  
   in loop (p+1) q

**Algorithm 8.3 (Quickselect)** qselect k xs = runST \$ do  
 axs <- newListArray (0,(length xs-1)) xs  
 mx <- aqselect k axs 0 (length xs-1)  
 return mx  
 aqselect k axs i j | i > j = return Nothing  
                   | otherwise = do  
     p <- apartition axs i j  
     if | k < p -> aqselect k axs i (p-1)  
       | k > p -> aqselect k (p+1) j  
       | otherwise -> do x <- readArray axs p  
                         return x