

Introduction to Computer Science

Author: Daria Shutina

Introduction to Computer Science

Haskell cheat sheet

Lists

Strings

map, filter, zip, zipWith

foldl, foldr

quickSort

Datatypes

Binary tree

Typeclasses

Usage of a constraint in a function declaration

Eq

Creating an instance of Eq

Ord

Functor

Part 1

Maze solving algorithms

Kruskal's algorithm

String search algorithms

Naive String Search

Example

Boyer-Moore algorithm

Bad character rule

Implementation

Landau notations

Th. (Landau Set Ranking)

Th. (Landau Set Computation Rules)

(Non)determinism or randomness

Haskell cheat sheet

A function that takes a Number (Integer, Double, Rational) and returns a Number

```
1 f :: Num a => a -> a
2 f x = x^2
```

Lists

```

1 concat [[1..3],[4..7],[8..10]]
2 reverse [1..10]
3
4 take 5 [1..] -- [1, 2, 3, 4, 5]
5 drop 5 [1..10] -- [6, 7, 8, 9, 10]
6
7 elem 5 [1..10] -- if `5` is in the list
8
9 [1..] !! 9 -- take the 9-th elem, `!!` operation is not safe
10
11

```

Strings

```

1 head "foo" -- 'f'
2 tail "foo" -- "oo"
3 last "foo" -- 'o'
4 init "foo" -- "fo"

```

map, filter, zip, zipWith

```

1 map (\x -> x * 2 + 1) [1..3] -- [3, 5, 7]
2 map ((+ 1) . (* 2)) [1..10] -- [3, 5, 7]
3
4 filter even [1..4] -- [1, 3]
5
6 zip [1..] "hi" -- [ (1,'h'), (2,'i') ]
7 zipWith (\a b -> (show a) ++ [b]) [1..] "hi" -- [ "1h", "2i" ]

```

foldl, foldr

```

1 foldl (-) 0 [1..3] -- ((0 - 1) - 2) - 3 = -6
2 foldr (-) 0 [1..3] -- 1 - (2 - (3 - 0)) = 2

```

quickSort

```

1 import Data.List
2
3 quickSort :: (Ord a) => [a] -> [a]
4 quickSort [] = []
5 quickSort (x:xs) = quickSort smaller ++ [x] ++ quickSort larger
6     where
7         (smaller, larger) = partition (< x) xs

```

Datatypes

```

1 data EmployeeInfo = Employee String Int Double [String] deriving (Show)
2 --      type                      data constructor(s)
3 --      constructor      consists of components or fields of type
4
5 p = Employee "Joe Sample" 22 186.3 []

```

```

1 data Person = Person
2     { name :: String
3     , age  :: Int
4     }
5     deriving (Show, Eq)
6
7 increaseAge :: Person -> Person
8 increaseAge person = person { age = age person + 1 }

```

Binary tree

```

1 data Tree a = Empty
2             | Leaf a
3             | Branch a (Tree a) (Tree a)
4             deriving (Eq, Show)
5
6 toList :: Tree a -> [a]
7 toList Empty = []
8 toList (Leaf x) = [x]
9 toList (Branch x l r) = toList l ++ [x] ++ toList r
10
11 map :: (a -> b) -> Tree a -> Tree b
12 map f Empty = Empty
13 map f (Leaf x) = Leaf (f x)
14 map f (Branch x l r) = Branch (f x) (map f l) (map f r)
15
16 foldr :: (a -> b -> b) -> b -> Tree a -> b

```

```

17 foldr _ z Empty = z
18 foldr f z (Leaf x) = f x z
19 foldr f z (Branch x l r) = foldr f (f x (foldr f z r)) l
20
21 foldl :: (a -> b -> b) -> b -> Tree a -> b
22 foldl _ z Empty = z
23 foldl f z (Leaf x) = f x z
24 foldl f z (Branch x l r) = foldr f (f x (foldl f z l)) r

```

Typeclasses

Typeclasses describe a set of types that have a common interface and behavior. A type belonging to a typeclass implements the functions and behavior defined by the typeclass.

Let us look at the expression:

```
1 (Eq a) => a -> a -> Bool
```

Everything before the symbol `=>` is a *class constraint*.

Usage of a constraint in a function declaration

We want to check whether the element is in the given list. We put a constraint on a type `a` -- it should be an instance of `Eq` typeclass.

```

1 elem :: Eq a => a -> [a] -> Bool
2 elem _ [] = False
3 elem y (x:xs) = y == x || elem y xs

```

Eq

```

1 class Eq a where
2     (==) :: a -> a -> Bool
3     (==) a b = not (a /= b) -- default implementation
4     (/=) :: a -> a -> Bool
5     (/=) a b = not (a == b) -- default implementation

```

`a` is an instance of typeclass `Eq`. It can overload operators `==` and `/=` or use a default version.

Creating an instance of Eq

```

1  -- custom type
2  data Weekday = Mon | Tue | Wed | Thu | Fri | Sat | Sun
3
4  instance Eq Weekday where
5      Mon == Mon = True
6      Tue == Tue = True
7      Wed == Wed = True
8      Thu == Thu = True
9      Fri == Fri = True
10     Sat == Sat = True
11     Sun == Sun = True
12     _ == _ = False

```

```

1  -- lists
2  -- we create an instance of a list and also put a constraint on a type `a`
3  instance Eq a => Eq [a] where
4      [] == [] = True
5      (x:xs) == (y:ys) = x == y && xs == ys
6      _ == _ = False

```

Ord

It is an extension of `Eq`, so `Ord` is a subclass of `Eq`.

Default implementation:

```

1  data Ordering = LT | EQ | GT
2
3  instance Eq Ordering where
4      LT == LT = True
5      EQ == EQ = True
6      GT == GT = True
7      _ == _ = False
8
9  class (Eq a) => Ord a where
10     (<), (<=), (>=), (>) :: a -> a -> Bool
11     compare :: a -> a -> Ordering
12     max, min :: a -> a -> a
13
14     compare x y | x == y = EQ
15                 | x <= y = LT
16                 | otherwise = GT
17
18     x <= y = compare x y /= GT
19     x < y = compare x y == LT

```

```

20     x >= y = compare x y /= LT
21     x > y = compare x y == GT
22
23     max x y | x <= y = y
24             | otherwise = x
25
26     min x y | x <= y = x
27             | otherwise = y

```

Functor

Part 1

Maze solving algorithms

We can imagine that a maze is a graph. The cells are vertexes of the graph. If you can get from one cell to another, then there is an edge between the corresponding vertexes.

More formally, for a maze M we have a tuple $M = (T, S, X)$ where

- $T = (V, E)$ is a graph with the vertexes V and edges E ;
- $S \in V$ is the start node;
- $X \in V$ is the exit node.

We want to find a solution for the maze \Leftrightarrow we want to build a spanning tree.

Kruskal's algorithm

We want to build an MST T in the connected graph G .

Initially, T is an empty graph and every vertex forms a subset of a size 1.

Edges of G are sorted in an increasing order. We consider an edge and add it to the answer, if it connects vertexes from different subsets. Then two subsets are merged into one subset.

```

1  std::vector<int> parent(n, -1);
2  std::vector<int> rank(n, 0); // ну типа высота дерева
3
4  int find_set(int v) {
5      if (parent[v] == v) {

```

```

6         return v;
7     }
8     return parent[v] = find_set(parent[v]);
9 }
10
11 void union_set(int a, int b) {
12     int v_a = find_set(a);
13     int v_b = find_set(b);
14     if (v_a == v_b) {
15         return;
16     }
17     if (rank[v_a] < rank[v_b]) {
18         std::swap(v_a, v_b);
19     }
20     parent[v_b] = v_a;
21     if (rank[v_a] == rank[v_b]) {
22         ++rank[v_a];
23     }
24 }
25
26 int kraskal(std::set<std::pair<int, std::pair<int, int>>> &Edges) {
27     int cost = 0;
28     for (const auto &edge : Edges) {
29         int w = edge.first, u = edge.second.first, v = edge.second.second;
30         if (find_set(u) != find_set(v)) {
31             union_set(u, v);
32             cost += w;
33         }
34     }
35     return cost;
36 }

```

Total complexity is $O(\log n)$.

String search algorithms

We need a program to find a (relatively short) string in a (possibly long) text.

Problem formalization

Σ is an alphabet. $k = |\Sigma|$.

Σ^* is a set of all words that can be created out of Σ .

$$\Sigma^0 = \{\epsilon\}$$

$$\Sigma^1 = \Sigma$$

$$\Sigma^i = \{wv : w \in \Sigma^{i-1} \wedge v \in \Sigma\}, i > 1$$

$$\Sigma = \bigcup_{i \geq 0} \Sigma^i$$

$t \in \Sigma^*$ is a text and $p \in \Sigma^*$ is a pattern.

$|t| = n$, $|p| = m$, $n > m$.

We need to find the first occurrence of p in t .

Naive String Search

Check at each position whether the pattern matches.

Lowercase characters indicate comparisons that were skipped.

Total complexity is $O(nm)$.

Example

Example: $t = \text{FINDANEEDLEINAHAYSTACK}$, $p = \text{NEEDLE}$

F I N D A N E E D L E I N A H A Y S T A C K

N e e d l e

N e e d l e

N E e d l e

N e e d l e

N e e d l e

N E E D L E

Boyer-Moore algorithm

The idea is to compare the pattern right to left instead left to right. If there is a mismatch, try to move the pattern as much as possible to the right.

Bad character rule

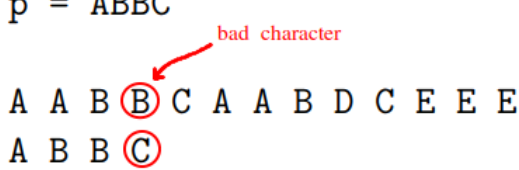
$s := \text{AABBCCAABDCEEE}$, $p := \text{ABBC}$.

We begin to compare the first alignment from right to left. **B** from s and **C** from p mismatch, so **B** is called a *bad character*.

There are two cases of a bad character:

1. The bad character is in p . Then we move p till the bad character matches with some letter from p .

$s = \text{AABBCCAABDC}$
 $p = \text{ABBC}$



bad character

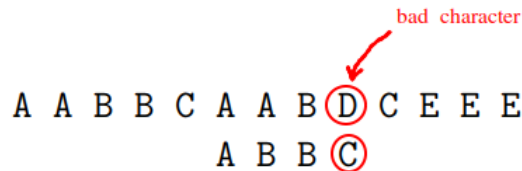
A A B **B** C A A B D C E E E
 A B B **C**

'B' is in p. Move p till 'B' matches with smth from 'ABB'

A A B B C A A B D C E E E
 A B B C

2. The bad character is not in p . Then we move p so that it starts from the position which has not been considered yet.

$s = \text{AABBCCAABDCEEE}$
 $p = \text{ABBC}$



bad character

A A B B C A A B **D** C E E E
 A B B **C**

'D' is a bad character.
 It is not in p, so we just start from the letter after 'D'.

A A B B C A A B D C E E E
 A B B C

Implementation

For the bad character rule, we need a function that takes the bad character and returns the number of alignments that can be skipped. We can pre-compute all possible skips and store the skips in a two-dimensional table. The table can be seen as a function that maps the unmatched character and the position in the pattern to the number of alignments that can be skipped.

For example, $p := \text{NEED}$. The complete table is shown on the left of the picture. The behaviour is the same for letters which are not present in p , so we can make the table shorter, like on the right of the picture.

	0	1	2	3
<i>A</i>	0	1	2	3
<i>B</i>	0	1	2	3
<i>C</i>	0	1	2	3
<i>D</i>	0	1	2	—
<i>E</i>	0	—	—	0
\vdots	\vdots	\vdots	\vdots	\vdots
<i>N</i>	—	0	1	2
\vdots	\vdots	\vdots	\vdots	\vdots
<i>Z</i>	0	1	2	3

	0	1	2	3
<i>D</i>	0	1	2	—
<i>E</i>	0	—	—	0
<i>N</i>	—	0	1	2
*	0	1	2	3

Example using the table:

```

1  s = D D D D D D D
2  p = N E E D
3      0 1 2 3
4
5
6  1)  D D D [D] D D D
7      N E E [D]
8
9  D:D --> matched
10
11
12  2)  D D [D] D D D D
13      N E [E] D
14
15  D:E --> table[D][2] = 2
16      --> 2 alignments are skipped.
17
18
19  3)  D D D D D D [D]
20      N E E [D]
21
22
23  4) etc.
```

Landau notations

$$f(n) = O(g(n)) \Leftrightarrow \exists C \in R, n_0 : f(n) < C \cdot g(n), \forall n \geq n_0.$$

$$f(n) = \Omega(g(n)) \Leftrightarrow \exists C \in R, n_0 : f(n) > C \cdot g(n), \forall n \geq n_0.$$

$$f(n) = \Theta(g(n)) \Leftrightarrow \exists C_1, C_2 \in R, n_0 : C_1 \cdot g(n) < f < C_2 \cdot g(n), \forall n \geq n_0.$$

Th. (Landau Set Ranking)

The commonly used Landau Sets establish a ranking such that

$$O(1) \subset O(\log n) \subset O(n) \subset O(n \log n) \subset O(n^2) \subset O(n^k) \subset O(I^n)$$

for all $k > 2$ and $I > 1$.

Th. (Landau Set Computation Rules)

$$1. \left. \begin{array}{l} k \neq 0 \\ f = O(g) \end{array} \right\} \Rightarrow kf = O(g)$$

$$2. \left. \begin{array}{l} f_1 = O(g_1) \\ f_2 = O(g_2) \end{array} \right\} \Rightarrow f_1 + f_2 = O(\max(g_1, g_2))$$

$$3. \left. \begin{array}{l} f_1 = O(g_1) \\ f_2 = O(g_2) \end{array} \right\} \Rightarrow f_1 f_2 = O(g_1 g_2)$$

(Non)determinism or randomness

A *deterministic algorithm* is an algorithm which, given a particular input, will always produce the same output, with the execution always passing through the same sequence of states.

A *nondeterministic algorithm* is an algorithm that can exhibit different behaviors on the same input.

A *randomized algorithm* is a (nondeterministic) algorithm that employs a degree of randomness as part of its logic. Random number generators often use algorithms to produce so called *pseudo random numbers* -- sequences of numbers that "look" random but that are not really random (since they are calculated using a deterministic algorithm).

