Algorithms Design & Analysis

1 Introduction

Given example function insert

```
insert :: Int -> [Int] -> [Int]
insert x [] = [x]
insert x (y:ys)
     | x <= y    = x : y : ys
     | otherwise = y : insert x ys
```

By analysing:

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

Solving the recursion:

$$\begin{aligned} T_{insert}(n) &= 1 + T_{insert}(n-1) \\ &= 1 + (1 + T_{insert}(n-2)) \\ &\dots \\ &= 1 + (1 + \ldots + T_{insert}(n-n)) \\ &= n + 1 \end{aligned}$$

Now $T_{insert}(n) = n + 1$.

Then giving isort :

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

By analysing:

$$T_{isort}(0) = 1$$
$$T_{isort}(n) = 1 + T_{insert}(n-1) + T_{isort}(n-1)$$

Solving the recursion:

$$\begin{aligned} T_{isort}(n) &= 1 + T_{insert}(n-1) + T_{isort}(n-1) \\ &= 1 + n + T_{isort}(n-1) \\ &= 1 + n + (1 + (n-1) + \ldots + T_{isort}(n-n)) \\ &= n + \frac{n(n+1)}{2} + 1 \end{aligned}$$

So isort runs approximately in square time.

1.1 Normal Forms

There are three kinds of normal forms.

Normal Form An expression is in **normal form** (**NF**) if it is

- A *constructor* applied to arguments in NF or
- A λ-abstraction whose body is in NF.

An expression in normal form can not be further reduced.
A *constructor* can be empty list [] , a cons :, number literals, …

1.1.1 Examples of Expressions in Normal Form

- 1
- []
- 5:[]
- [1,2,3]

1.2 Weak Head Normal Forms

Weak Head Normal Form An expression is in **weak head normal form** (**WHNF**) if it is

- A *constructor* applied to arguments in any form or

- A λ-abstraction whose body is in any form.

An expression in weak head normal form contains parts that are waiting to be computed.
NF $\subset$ WHNF $\subset$ Expr

1.2.1 Examples of Expressions in Weak Head Normal Form

- (3, 5+2)
- [1+1]
- Just(8 + 9)
- 5:repeat 5

2 Evaluation

2.1 Definition of The Example Language

$$
\begin{aligned}
e \ ::= \ & x \\
| \ & k \\
| \ & f \ e_1 \ e_2 \\
| \ & \text{if } e \text{ then } e_1 \text{ else } e_2
\end{aligned}
$$

Constants include: 0, 3, 5, 7, ..., [], (:), +
Rules:

- $[x_1, \ldots, x_n]$ is in short of $x_1 : x_2 : \ldots : x_n$.

Given insert function:

```
insert x xs =
        if null xs then x : []
                   else if x <= head xs then x: xs
                                        else head xs : insert x (tail xs)
```

2.2 Strict Time Analysis

Given a function f of n argument,s $T(f)x_1 \ldots x_n$ is the number of steps to take to evaluate $fx_1 \ldots x_n$.
For a primitive f we have: $T(f)\ x_1 \ldots x_n = 0$. For example: $T(head)\ xs = 0$, $T((+))\ xy = 0$.
Otherwise, $f\ x_1\ \ldots\ x_n = e$, $T(f)\ x_1\ \ldots\ x_n = 1 + T(e)$

We define T inductively on e:

$$T(x) = 0$$
$$T(k) = 0$$

$$\begin{aligned} T(f\ e_1\ \ldots\ e_n) &= T(f)\ e_1\ \ldots\ e_n \\ &\quad + T(e_1) + \ldots + T(e_n) \end{aligned}$$

$$T(\text{if } e \text{ then } e_1 \text{ else } e_2) = T(e) + \text{if } e \text{ then } T(e_1) \text{ else } T(e_2)$$

2.3 Analysing function length

Given function length:

$$\text{length } xs \ = \text{if null } xs \text{ then } 0 \text{ else } 1 + \text{length}(\text{tail } xs)$$

By analysing:

$$\begin{aligned} &T(\text{length } xs) \\ &\qquad = T(\text{length})\ xs + T(xs) \\ &\qquad = 1 + T(\text{if null } xs \text{ then } 0 \text{ else } 1 + \text{length}(\text{tail } xs)) \\ &\qquad = 1 + T(\text{null } xs) + \text{if null } xs \text{ then } T(0) \text{ else } T(1 + \text{length}(\text{tail } xs)) \\ &\qquad = 1 + \text{if null } xs \text{ then } 0 \text{ else } T(\text{length})(\text{tail } xs) + T(\text{tail } xs) \\ &\qquad = 1 + \text{if null } xs \text{ then } 0 \text{ else } T(\text{length})(\text{tail } xs) \end{aligned}$$

2.4 Composition Rule

The cost of $f(g(x))$ which is $T(f(g(x)))$ is:

$$\begin{aligned} T(f(g\ x)) &= T(f)(g\ x) + T(g\ x) \\ &= T(f)(g\ x) + T(g)\ x + T(x) \\ &= T(f)(g\ x) + T(g)\ x \end{aligned}$$

3 Asymptotics

L-function is a function on a real variable, that is well-defined for all variables greater than some definite value that is real, positive, monotonic, one-valued and given by a finite combination of algebraic symbols, logarithms, exponential, and constants.

Any L-function is ultimately continuous, of constant sign, monotonic, and as $n \to \infty$, $f(n)$ will be 0, ∞, or some value k.

3.1 Du Bois-Reymond Notation

$$
\begin{aligned}
f \prec g &\iff \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \\
f \preccurlyeq g &\iff \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty \\
f \asymp g &\iff 0 < \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty \\
f \succcurlyeq g &\iff \lim_{n\to\infty} \frac{f(n)}{g(n)} > 0 \\
f \succ g &\iff \lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty
\end{aligned}
$$

converse $f \prec g \iff g \succ f$

transitivity $f \prec g \wedge g \prec h \Rightarrow f \prec h$

$f \preccurlyeq g \wedge g \preccurlyeq h \Rightarrow f \preccurlyeq h$

3.2 Bachman-Landau Notation

Bachman-Landau Notation is a more standard notation for function complexity.

$$
\begin{aligned}
f(n) \in o(g(n)) &\iff f \prec g \\
f(n) \in O(g(n)) &\iff f \preccurlyeq g \\
f(n) \in \Theta(g(n)) &\iff f \asymp g \\
f(n) \in \Omega(g(n)) &\iff f \succcurlyeq g \\
f(n) \in \omega(g(n)) &\iff f \succ g
\end{aligned}
$$

The sets can also be defined directly:

$$o(g(n)) = \{f | \forall \delta > 0. \exists n_0 > 0. \forall n > n_0. f(n) < \delta \cdot g(n)\}$$
$$O(g(n)) = \{f | \exists \delta > 0. \exists n_0 > 0. \forall n > n_0. f(n) \Leftarrow \delta \cdot g(n)\}$$
$$\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$$
$$O(g(n)) = \{f | \exists \delta > 0. \exists n_0 > 0. \forall n > n_0. f(n) >= \delta \cdot g(n)\}$$
$$O(g(n)) = \{f | \forall \delta > 0. \exists n_0 > 0. \forall n > n_0. f(n) > \delta \cdot g(n)\}$$

4 Lists

4.1 Common Monoids

Common monoids include:

- $+$ and 0
- x and 1
- $\cup$ and $\emptyset$
- $\cap$ and powerset of X
- $\vee$ and $false$
- $\wedge$ and $true$
- $++$ and []
- $\cdot$ and id (function composition and identity function)

4.2 Two Kinds of fold Functions

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

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

foldr and foldl are **extensionally** equivalent when f and k forms monoid. They are extensionally equivalent means their result are equal, but there can be difference in complexity.

4.2.1 Implementing concat

```
concat :: [[a]] -> [a]
concat []  = []
concat (xs : xss) = xs ++ concat xss
```

++ and [] forms a monoid. So concat can be implemented by foldl or foldr.

```
concatr :: [[a]] -> [a]
concatr xss = foldr (++) [] xss

concatl :: [[a]] -> [a]
concatl xss = foldl (++) [] xss
```

Expansion of contactr is $((xs_1 + +xs_2) + +xs_3) + + \ldots + +xs_{m-1}$.
++ takes time only to copy its left operand then add a reference to right operand.
concatr uses $O(nm^2)$ while concatl uses only $O(nm)$ where m is the size of xss.

4.2.2 Re-association of List Concatenation Improves Performance

The original behavior of ++ is right-associative:

```
((ws ++ xs) ++ ys) ++ zs
```

Since function composition is left associative, i.e. $h \cdot (g \cdot f) = (h \cdot g) \cdot f$, we can replace concatenation with function compositions.

$$(ws + +) \cdot (xs + +) \cdot (ys + +) \cdot (zs + +) \cdot []$$

This is equivalent to

```
ws ++ (xs ++ (ys ++ zs))
```

5 Divide and Conquer

Divide and Conquer is an algorithmic strategy in 3 parts:

1. Divide a problem into smaller sub-problems.
2. Turn sub-problems into sub-solutions.
3. Conquer sub-solutions into a solution.

6 Dynamic Programming

6.1 Strategies

- Write an inefficient recursive function
- Improve efficiency by storing intermediate shared results.

Need to choose how to index the table. Indexes should be simple.

7 Amortised Analysis

Amortised Cost measures average cost in long term, compared to pessimistic worst-case analysis. We need amortised analysis because worst-case analysis often over-measure complexity since in most times we won't reach the worst case. In amortised analysis, operations must be understood in a wider context, rather than treating them in isolation.

7.1 Defining Amortised Cost

The goal is to define the 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:

$$C_{op_1}(xs_i) \Leftarrow A_{op_1}(xs_i) + S(xs_i) - S(xs_{i+1})$$

where:

- $C_{op_i}(xs_i)$ is cost for each operation op_i on data xs_i.
- $A_{op_i}(xs_i)$ is amortised cost for that operation.
- $S(xs)$ is the size of xs, that usually changes steady, but suddenly changes drastically.

7.2 Finding Amortised Cost

We find amortised cost by guessing a $A_{op_1}(xs_i)$ and test if the above equation **??** holds for all cases of $C_{op_1}(xs_i)$. Usually we can verify by testing worst case C and best case C.
$S(xs)$ should be defined as length of list

7.2.1 Example: Finding Amortised Cost of inc on Binary

We will analyse amortised cost for this

```
type Binary = [Bit]
data Bit = O | I

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