Programming in Haskell: Lecture 13

S P Suresh

September 16, 2019

Measuring efficiency

- Computation is reduction

Measuring efficiency

- Computation is reduction
- Application of definitions as rewriting rules

Measuring efficiency

- Computation is reduction
- Application of definitions as rewriting rules
- Count the number of reduction steps

Measuring efficiency

- Computation is reduction
- Application of definitions as rewriting rules
- Count the number of reduction steps
- Running time is $T(n)$ for input size n

Example: complexity of `(++)`

- `(++)` attaches one list before another

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

  [1,2,3]++[4,5,6]
= 1: [2,3]++[4,5,6]
= 1:2: [3]++[4,5,6]
= 1:2:3: []++[4,5,6]
= 1:2:3:[4,5,6]
```

Example: complexity of `(++)`

- `(++)` attaches one list before another

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

  [1,2,3]++[4,5,6]
= 1: [2,3]++[4,5,6]
= 1:2: [3]++[4,5,6]
= 1:2:3: []++[4,5,6]
= 1:2:3:[4,5,6]
```

- To compute `xs ++ ys`, use the second rule **`length`** `xs` times, and the first rule once

Example: **elem**

```haskell
elem :: Int -> [Int] -> Bool
elem i []     = False
elem i (x:xs) = i == x || elem i xs


  elem 3 [2,3,7,8,9] = elem 3 [3,7,8,9] = True
  elem 3 [2,4,7,8,9] = elem 3 [4,7,8,9]
= elem 3 [7,8,9]     = elem 3 [8,9]
= elem 3 [9] = elem 3 [] = False
```

- Time taken depends on input size as well as the input itself

Variations across inputs

- Worst case complexity

Variations across inputs

- Worst case complexity
- Maximum running time over all inputs of size n

Variations across inputs

- Worst case complexity
- Maximum running time over all inputs of size n
- Pessimistic: may be rare

Variations across inputs

- Worst case complexity
- Maximum running time over all inputs of size n
- Pessimistic: may be rare
- **Average case complexity**: more realistic, but difficult/impossible to compute

Asymptotic complexity

- Interested in $T(n)$ in terms of orders of magnitude

Asymptotic complexity

- Interested in $T(n)$ in terms of orders of magnitude
- $f(n) = O(g(n))$ if there is a constant k and number $N > 0$ such that

Asymptotic complexity

- Interested in $T(n)$ in terms of orders of magnitude
- $f(n) = O(g(n))$ if there is a constant k and number $N > 0$ such that
  - $f(n) \leq k \cdot g(n)$ for all $n \geq N$

Asymptotic complexity

- Interested in $T(n)$ in terms of orders of magnitude
- $f(n) = O(g(n))$ if there is a constant k and number $N > 0$ such that
  - $f(n) \leq k \cdot g(n)$ for all $n \geq N$
- $an^2 + bn + c = O(n^2)$ (take $k = |a| + |b| + |c|$)

Asymptotic complexity

- Interested in $T(n)$ in terms of orders of magnitude
- $f(n) = O(g(n))$ if there is a constant k and number $N > 0$ such that
  - $f(n) \leq k \cdot g(n)$ for all $n \geq N$
- $an^2 + bn + c = O(n^2)$ (take $k = |a| + |b| + |c|$)
- Ignore constant factors, lower-order terms

Asymptotic complexity

- Interested in $T(n)$ in terms of orders of magnitude
- $f(n) = O(g(n))$ if there is a constant k and number $N > 0$ such that
  - $f(n) \leq k \cdot g(n)$ for all $n \geq N$
- $an^2 + bn + c = O(n^2)$ (take $k = |a| + |b| + |c|$)
- Ignore constant factors, lower-order terms
- **Typical complexities**: $O(n)$, $O(n \log n)$, $O(n^k)$, $O(2^n)$, …

Asymptotic complexity

- Interested in $T(n)$ in terms of orders of magnitude
- $f(n) = O(g(n))$ if there is a constant k and number $N > 0$ such that
  - $f(n) \leq k \cdot g(n)$ for all $n \geq N$
- $an^2 + bn + c = O(n^2)$ (take $k = |a| + |b| + |c|$)
- Ignore constant factors, lower-order terms
- **Typical complexities**: $O(n)$, $O(n \log n)$, $O(n^k)$, $O(2^n)$, ...
- Complexity of `++` is $O(n)$, where n is the length of the first list

Asymptotic complexity

- Interested in $T(n)$ in terms of orders of magnitude
- $f(n) = O(g(n))$ if there is a constant k and number $N > 0$ such that
  - $f(n) \leq k \cdot g(n)$ for all $n \geq N$
- $an^2 + bn + c = O(n^2)$ (take $k = |a| + |b| + |c|$)
- Ignore constant factors, lower-order terms
- **Typical complexities**: $O(n)$, $O(n \log n)$, $O(n^k)$, $O(2^n)$, ...
- Complexity of `++` is $O(n)$, where n is the length of the first list
- Complexity of `elem` is $O(n)$ (**worst case**!)

Complexity of `reverse`

- Naive reverse

```
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

Complexity of `reverse`

- Naive reverse

```
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

- Write a recurrence for $T(n)$

$$T(0) = 1$$
$$T(n) = T(n-1) + n$$

Complexity of `reverse`

- Naive reverse

```
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

- Write a recurrence for $T(n)$

$$T(0) = 1$$
$$T(n) = T(n-1) + n$$

- Solve by expanding the recurrence

Complexity of `reverse`

- Solving the recurrence

$$
\begin{aligned}
T(n) &= T(n-1)+n \\
&= (T(n-2)+(n-1))+n \\
&= ((T(n-3)+(n-2))+(n-1))+n \\
&= \ldots \\
&= ((\cdots(T(0)+1)+\cdots(n-2))+(n-1))+n \\
&= 1+1+2+\cdots+(n-2)+(n-1)+n \\
&= 1+\frac{n(n+1)}{2} \\
&= O(n^2)
\end{aligned}
$$

Speeding up **reverse**

- Reverse into the empty list

```
reverse            = revInto []
revInto a []       = a
revInto a (x:xs) = revInto (x:a) xs
```

Speeding up **reverse**

- Reverse into the empty list

```
reverse            = revInto []
revInto a []       = a
revInto a (x:xs) = revInto (x:a) xs
```

- Complexity of `revInto a xs`

Speeding up `reverse`

- Reverse into the empty list

```
reverse             = revInto []
revInto a []        = a
revInto a (x:xs) = revInto (x:a) xs
```

- Complexity of `revInto a xs`
  - Let n be `length xs`

Speeding up **reverse**

- Reverse into the empty list

```
reverse              = revInto []
revInto a []         = a
revInto a (x:xs)     = revInto (x:a) xs
```

- Complexity of `revInto a xs`
  - Let n be `length xs`
  - $T(n) = T(n-1) + 1$

Speeding up `reverse`

- Reverse into the empty list

```
reverse              = revInto []
revInto a []         = a
revInto a (x:xs) = revInto (x:a) xs
```

- Complexity of `revInto a xs`
  - Let n be `length xs`
  - $T(n) = T(n-1) + 1$
  - Expanding, $T(n) = O(n)$

Speeding up `reverse`

- Reverse into the empty list

```
reverse            = revInto []
revInto a []       = a
revInto a (x:xs) = revInto (x:a) xs
```

- Complexity of `revInto a xs`
  - Let n be `length xs`
  - $T(n) = T(n-1) + 1$
  - Expanding, $T(n) = O(n)$
- Thus `reverse` has complexity $O(n)$

Sorting

- Goal is to arrange a list in ascending order

Sorting

- Goal is to arrange a list in ascending order
- How would we sort a hand of cards?

Sorting

- Goal is to arrange a list in ascending order
- How would we sort a hand of cards?
- A single card is sorted, by definition

Sorting

- Goal is to arrange a list in ascending order
- How would we sort a hand of cards?
- A single card is sorted, by definition
- Put second card before/after first

Sorting

- Goal is to arrange a list in ascending order
- How would we sort a hand of cards?
- A single card is sorted, by definition
- Put second card before/after first
- "Insert" third, fourth, ...card in correct place

Sorting

- Goal is to arrange a list in ascending order
- How would we sort a hand of cards?
- A single card is sorted, by definition
- Put second card before/after first
- "Insert" third, fourth, …card in correct place
- Insertion sort

Insertion sort: `insert`

- Insert an element into a sorted list:

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

Insertion sort: `insert`

- Insert an element into a sorted list:

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

- $T(n) = O(n)$

Insertion sort: `isort`

- The sorting procedure:

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

Insertion sort: `isort`

- The sorting procedure:

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

- Alternatively:

```
isort = foldr insert []
```

Insertion sort: `isort`

- The sorting procedure:

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

- Alternatively:

```
isort = foldr insert []
```

- Recurrence: $T(n) = T(n-1) + O(n)$

Insertion sort: `isort`

- The sorting procedure:

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

- Alternatively:

```
isort = foldr insert []
```

- Recurrence: $T(n) = T(n-1) + O(n)$
- Expanding, $T(n) = O(n^2)$

A better strategy?

- Divide list in two equal parts

A better strategy?

- Divide list in two equal parts
- Separately sort left and right half

A better strategy?

- Divide list in two equal parts
- Separately sort left and right half
- Merge the two sorted halves to get the full list sorted

A better strategy?

- Divide list in two equal parts
- Separately sort left and right half
- Merge the two sorted halves to get the full list sorted
    - Given two sorted lists `xs` and `ys`, merge into a sorted list `zs`

A better strategy?

- Divide list in two equal parts
- Separately sort left and right half
- Merge the two sorted halves to get the full list sorted
  - Given two sorted lists `xs` and `ys`, merge into a sorted list `zs`
  - Compare first element of `xs` and `ys`

A better strategy?

- Divide list in two equal parts
- Separately sort left and right half
- Merge the two sorted halves to get the full list sorted
  - Given two sorted lists `xs` and `ys`, merge into a sorted list `zs`
  - Compare first element of `xs` and `ys`
  - Move it into `zs`

A better strategy?

- Divide list in two equal parts
- Separately sort left and right half
- Merge the two sorted halves to get the full list sorted
    - Given two sorted lists `xs` and `ys`, merge into a sorted list `zs`
    - Compare first element of `xs` and `ys`
    - Move it into `zs`
    - Repeat until all elements in `xs` and `ys` are processed

Merging two sorted lists

- Merging `[32, 74, 89]` and `[21, 55, 64]`

```
  merge [32, 74, 89] [21, 55, 64]
= 21: merge [32,74,89] [55,64]
= 21: 32: merge [74, 89] [55, 64]
= 21: 32: 55: merge [74, 89] [64]
= 21: 32: 55: 64: merge [74, 89] []
= 21: 32: 55: 64: [74, 89]
= [21, 32, 55, 64, 74, 89]
```

Merge sort

- Sort `l!!0` to `l!!(n/2-1)`

Merge sort

- Sort `l!!0` to `l!!(n/2-1)`
- Sort `l!!(n/2)` to `l!!(n-1)`

Merge sort

- Sort `l!!0` to `l!!(n/2-1)`
- Sort `l!!(n/2)` to `l!!(n-1)`
- Merge sorted halves into `l`

Merge sort

- Sort `l!!0` to `l!!(n/2-1)`
- Sort `l!!(n/2)` to `l!!(n-1)`
- Merge sorted halves into `l`
- How do we sort the halves?

Merge sort

- Sort `l!!0` to `l!!(n/2-1)`
- Sort `l!!(n/2)` to `l!!(n-1)`
- Merge sorted halves into `l`
- How do we sort the halves?
- Recursively, using the same strategy!

Merge Sort

- Sorting `[43, 32, 22, 78, 63, 57, 91, 13]`

```
  sort [43, 32, 22, 78, 63, 57, 91, 13]
= merge (sort [43, 32, 22, 78]) (sort [63, 57, 91, 13])
= merge (merge (sort [43,32]) (sort [22, 78]))
        (merge (sort [63,57]) (sort [91, 13]))
= ...
= merge (merge [32,43] [22, 78]) (merge [57,63] [13,91])
= merge [22, 32, 43, 78] [13, 57, 63, 91]
= [13, 22, 32, 43, 57, 63, 78, 91]
```

Merge Sort: merge

- Merging two sorted lists:

```
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
```

Merge Sort: *merge*

- Merging two sorted lists:

```
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
```

- Each comparison adds at least one element to the output list

Merge Sort: merge

- Merging two sorted lists:

```
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
```

- Each comparison adds at least one element to the output list
- Number of steps in `merge xs ys` is $O(n)$

Merge Sort: merge

- Merging two sorted lists:

```
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
```

- Each comparison adds at least one element to the output list
- Number of steps in `merge xs ys` is $O(n)$
    - n is `length xs + length ys`

Merge Sort

- Sorting a list:

```
sort :: [Int] -> [Int]
sort []  = []
sort [x] = [x]
sort xs  = merge (sort front) (sort back)
    where
        n = (length xs) `div` 2
        (front, back) = splitAt n xs
```