Programming in Haskell: Lecture 14

S P Suresh

September 18, 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

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

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

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 $O(n)$

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, $T(n) = O(n^2)$

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
```

Merge Sort: analysis

- $T(n)$: time taken by `sort` on input of length n

Merge Sort: analysis

- $T(n)$: time taken by `sort` on input of length n
- Assume, for simplicity, that n is a power of 2

Merge Sort: analysis

- $T(n)$: time taken by `sort` on input of length n
- Assume, for simplicity, that n is a power of 2
- Then the lengths of `front` and `back` are $\frac{n}{2}$

Merge Sort: analysis

- $T(n)$: time taken by `sort` on input of length n
- Assume, for simplicity, that n is a power of 2
- Then the lengths of `front` and `back` are $\frac{n}{2}$
- There are two recursive sorts

Merge Sort: analysis

- $T(n)$: time taken by `sort` on input of length n
- Assume, for simplicity, that n is a power of 2
- Then the lengths of `front` and `back` are $\frac{n}{2}$
- There are two recursive sorts
- Overall time taken by **`length`**, **`splitAt`** and `merge` is $O(n)$

Merge Sort: analysis

- $T(n)$: time taken by `sort` on input of length n
- Assume, for simplicity, that n is a power of 2
- Then the lengths of `front` and `back` are $\frac{n}{2}$
- There are two recursive sorts
- Overall time taken by **`length`**, **`splitAt`** and `merge` is $O(n)$
- Let us assume it is cn, for some constant c

Merge Sort: analysis

- $T(n)$: time taken by `sort` on input of length n
- Assume, for simplicity, that n is a power of 2
- Then the lengths of `front` and `back` are $\frac{n}{2}$
- There are two recursive sorts
- Overall time taken by **`length`**, **`splitAt`** and `merge` is $O(n)$
- Let us assume it is cn, for some constant c
- Recurrence: $T(n) = 2T(n/2) + cn$

Merge Sort: analysis

- **Recurrence**: $T(n) = 2T(n/2) + cn$

Merge Sort: analysis

- **Recurrence**: $T(n) = 2T(n/2) + cn$
- Expanding ...

$$\begin{aligned} T(1) &= 1 \\ T(n) &= 2T(n/2) + cn \\ &= 2(2T(n/4) + cn/2) + cn \\ &= 2^2 T(n/2^2) + 2cn \\ &= 2^2(2T(n/2^3) + cn/2^2) + 2cn \\ &= 2^3 T(n/2^3) + 3cn \\ &= \dots \\ &= 2^j T(n/2^j) + jcn \end{aligned}$$

Merge Sort: analysis

- **Recurrence**: $T(n) = 2T(n/2) + cn$

Merge Sort: analysis

- **Recurrence**: $T(n) = 2T(n/2) + cn$
- Expanding ...

$$\begin{aligned} T(1) &= 1 \\ T(n) &= 2T(n/2) + cn \\ &= \dots \\ &= 2^j T(n/2^j) + jcn \end{aligned}$$

Merge Sort: analysis

- **Recurrence**: $T(n) = 2T(n/2) + cn$
- Expanding ...

$$
\begin{aligned}
T(1) &= 1 \\
T(n) &= 2T(n/2) + cn \\
&= \ldots \\
&= 2^j T(n/2^j) + jcn
\end{aligned}
$$

- When $j = \log n$, $2^j = n$ and $n/2^j = 1$

Merge Sort: analysis

- **Recurrence**: $T(n) = 2T(n/2) + cn$
- Expanding ...

$$
\begin{aligned}
T(1) &= 1 \\
T(n) &= 2T(n/2) + cn \\
&= \ldots \\
&= 2^j T(n/2^j) + jcn
\end{aligned}
$$

- When $j = \log n$, $2^j = n$ and $n/2^j = 1$
- Thus $T(n) = 2^{\log n} T(1) + cn\log n = n + cn\log n = O(n\log n)$

Avoiding merge

- Merge is needed because some elements in `front` are greater than some elements in `back`

Avoiding merge

- Merge is needed because some elements in `front` are greater than some elements in `back`
- Can we ensure that everything in `front` is smaller than everything in `back`?

Avoiding merge

- Merge is needed because some elements in `front` are greater than some elements in `back`
- Can we ensure that everything in `front` is smaller than everything in `back`?
- Suppose the median value in list is `m`

Avoiding merge

- Merge is needed because some elements in `front` are greater than some elements in `back`
- Can we ensure that everything in `front` is smaller than everything in `back`?
- Suppose the median value in list is `m`
- Move all values `<= m` to `front`

Avoiding merge

- Merge is needed because some elements in `front` are greater than some elements in `back`
- Can we ensure that everything in `front` is smaller than everything in `back`?
- Suppose the median value in list is `m`
- Move all values `<= m` to `front`
- `back` has values `> m`

Avoiding merge

- Merge is needed because some elements in `front` are greater than some elements in `back`
- Can we ensure that everything in `front` is smaller than everything in `back`?
- Suppose the median value in list is `m`
- Move all values `<= m` to `front`
- `back` has values `> m`
- Recursively sort `front` and `back`

Avoiding merge

- Merge is needed because some elements in `front` are greater than some elements in `back`
- Can we ensure that everything in `front` is smaller than everything in `back`?
- Suppose the median value in list is `m`
- Move all values `<= m` to `front`
- `back` has values `> m`
- Recursively sort `front` and `back`
- List is now sorted! No need to merge

Avoiding merge

- How do we find the median?

Avoiding merge

- How do we find the median?
- Sort and pick up middle element

Avoiding merge

- How do we find the median?
- Sort and pick up middle element
- But our aim is to sort!

Avoiding merge

- How do we find the median?
- Sort and pick up middle element
- But our aim is to sort!
- Instead, pick up some value in list – **pivot**

Avoiding merge

- How do we find the median?
- Sort and pick up middle element
- But our aim is to sort!
- Instead, pick up some value in list – **pivot**
- Split list with respect to the pivot

Avoiding merge

- How do we find the median?
- Sort and pick up middle element
- But our aim is to sort!
- Instead, pick up some value in list – **pivot**
- Split list with respect to the pivot
- Usually we pick the first element as pivot

Quicksort

```
sort :: [Int] -> [Int]
sort []     = []
sort (x:xs) = sort front ++ [pivot] ++ sort back
    where
        pivot = x
        front = [y <- xs, y <= x]
        back  = [y <- xs, y > x]
```

Quicksort: analysis

- **Worst case**: `pivot` is maximum or minimum

Quicksort: analysis

- Worst case: `pivot` is maximum or minimum
- One partition is empty while the other is of size $n-1$

Quicksort: analysis

- **Worst case**: `pivot` is maximum or minimum
- One partition is empty while the other is of size $n-1$
- Partitioning takes time $O(n)$, say cn

Quicksort: analysis

- **Worst case**: `pivot` is maximum or minimum
- One partition is empty while the other is of size $n - 1$
- Partitioning takes time $O(n)$, say cn
- $T(n) = T(n-1) + cn$

Quicksort: analysis

- **Worst case**: `pivot` is maximum or minimum
- One partition is empty while the other is of size $n-1$
- Partitioning takes time $O(n)$, say cn
- $T(n) = T(n-1) + cn$
- Expanding, $T(n) = O(n^2)$

Quicksort: analysis

- **Worst case**: `pivot` is maximum or minimum
- One partition is empty while the other is of size $n-1$
- Partitioning takes time $O(n)$, say cn
- $T(n) = T(n-1) + cn$
- Expanding, $T(n) = O(n^2)$
- But **average case complexity** is $O(n \log n)$

Quicksort: analysis

- **Worst case**: `pivot` is maximum or minimum
- One partition is empty while the other is of size $n-1$
- Partitioning takes time $O(n)$, say cn
- $T(n) = T(n-1) + cn$
- Expanding, $T(n) = O(n^2)$
- But **average case complexity** is $O(n \log n)$
- Quicksort is one of the few examples amenable to average case analysis

Average case analysis

- Assume input is a permutation of $1, 2, \ldots, n$

Average case analysis

- Assume input is a permutation of $1, 2, \ldots, n$
- Actual values not important

Average case analysis

- Assume input is a permutation of $1, 2, \ldots, n$
- Actual values not important
- Only relative order matters

Average case analysis

- Assume input is a permutation of $1, 2, \ldots, n$
- Actual values not important
- Only relative order matters
- Each permutation is equally likely as input (uniform probability)

Average case analysis

- Assume input is a permutation of $1, 2, \ldots, n$
- Actual values not important
- Only relative order matters
- Each permutation is equally likely as input (uniform probability)
- Calculate running time across all inputs

Average case analysis

- Assume input is a permutation of $1, 2, \ldots, n$
- Actual values not important
- Only relative order matters
- Each permutation is equally likely as input (uniform probability)
- Calculate running time across all inputs
- Expected running time can be shown to be $O(n \log n)$