Introduction to Programming: Lecture 7

K Narayan Kumar

Chennai Mathematical Institute

http://www.cmi.ac.in/~kumar

29 August 2013

Complexity

- We can take the complexity $T(n)$ on inputs of length n to be
  - The maximum among all inputs of length `n`.
    Worst-case complexity

Measuring efficiency in Haskell . . .

- What is the complexity of `reverse`?

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

Measuring efficiency in Haskell ...

- What is the complexity of `reverse`?

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

- Write a recurrence for $T(n)$

$$
\begin{aligned}
T(0) &= 1 \\
T(n) &= T(n-1) + n
\end{aligned}
$$

Measuring efficiency in Haskell . . .

- What is the complexity of `reverse`?

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

- Write a recurrence for $T(n)$

$$\begin{aligned} T(0) &= 1 \\ T(n) &= T(n-1) + n \end{aligned}$$

- Expand and solve

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

The Big-O notation

- The exact value of $T(n)$ is not as interesting as its asymptotic behaviour.

The Big-O notation

- The exact value of $T(n)$ is not as interesting as its asymptotic behaviour.
- Instead of a complex function giving the exact value of $T(n)$ we will be happy with a simple function that is close to $T(n)$, but dominates it.

  Remember that we are interested in the worst-case analysis.

The Big-O notation

- The exact value of $T(n)$ is not as interesting as its asymptotic behaviour.
- Instead of a complex function giving the exact value of $T(n)$ we will be happy with a simple function that is close to $T(n)$, but dominates it.

  Remember that we are interested in the worst-case analysis.

  Example: We will be happy to write $T(n) = n^2$ instead of $\frac{n(n-1)}{2} + 1$.

The Big-O notation

- The exact value of $T(n)$ is not as interesting as its asymptotic behaviour.
- Instead of a complex function giving the exact value of $T(n)$ we will be happy with a simple function that is close to $T(n)$, but dominates it.

  Remember that we are interested in the worst-case analysis.

  Example: We will be happy to write $T(n) = n^2$ instead of $\frac{n(n-1)}{2} + 1$.
- The Big-O notation is a formal treatment of this idea of bounding by a nice function.

The Big-O notation

- $f(n) = O(g(n))$ iff there are constants k and M such that

$$f(n) \;\leq\; k.g(n) \quad \text{for any } n > M$$

The Big-O notation

- $f(n) = O(g(n))$ iff there are constants k and M such that

$$f(n) \ \leq \ k.g(n) \quad \text{for any } n > M$$

- Here are some examples:
  $\frac{n(n-1)}{2} + 1 \ = \ O(n^2)$

The Big-O notation

- $f(n) = O(g(n))$ iff there are constants k and M such that

$$f(n) \;\leq\; k.g(n) \quad \text{for any } n > M$$

- Here are some examples:

$$\frac{n(n-1)}{2} + 1 \;=\; O(n^2)$$

$$\frac{n(n-1)}{2} + 1 \;=\; O(n^4)$$

The Big-O notation

- $f(n) = O(g(n))$ iff there are constants k and M such that

$$f(n) \ \leq \ k.g(n) \quad \text{for any } n > M$$

- Here are some examples:

$$\frac{n(n-1)}{2} + 1 \ = \ O(n^2)$$

$$\frac{n(n-1)}{2} + 1 \ = \ O(n^4)$$

$$n^2 \ = \ O(\frac{n(n-1)}{2} + 1)$$

The Big-O notation

- $f(n) = O(g(n))$ iff there are constants k and M such that
$$f(n) \;\leq\; k.g(n) \quad \text{for any } n > M$$

- Here are some examples:
$$\frac{n(n-1)}{2} + 1 \;=\; O(n^2)$$
$$\frac{n(n-1)}{2} + 1 \;=\; O(n^4)$$
$$n^2 \;=\; O(\frac{n(n-1)}{2} + 1)$$
We will seldom use such a relation though!

The Big-O notation

- $f(n) = O(g(n))$ iff there are constants k and M such that

$$f(n) \;\leq\; k.g(n) \quad \text{for any } n > M$$

- Here are some examples:

  $\frac{n(n-1)}{2} + 1 \;=\; O(n^2)$

  $\frac{n(n-1)}{2} + 1 \;=\; O(n^4)$

  $n^2 \;=\; O(\frac{n(n-1)}{2} + 1)$

  We will seldom use such a relation though!

  $n.logn + n \;=\; O(n.logn)$

The Big-O notation

- $f(n) = O(g(n))$ iff there are constants k and M such that

$$f(n) \leq k.g(n) \quad \text{for any } n > M$$

- Here are some examples:

$$\frac{n(n-1)}{2} + 1 = O(n^2)$$

$$\frac{n(n-1)}{2} + 1 = O(n^4)$$

$$n^2 = O(\frac{n(n-1)}{2} + 1)$$

We will seldom use such a relation though!

$$n.logn + n = O(n.logn)$$

$$an^2 + bn.logn + cn + d = O(n^2)$$

Complexity of Insertion Sorting

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

Complexity of Insertion Sorting

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

The complexity of `insert` is `O(n)`.

Complexity of Insertion Sorting

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

The complexity of `insert` is `O(n)`.

- Complexity of insertion sorting:

```
isort [] = []
isort (x:xs) = insert x (isort xs)
```

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

Complexity of Insertion Sorting

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

The complexity of `insert` is `O(n)`.

- Complexity of insertion sorting:

```
isort [] = []
isort (x:xs) = insert x (isort xs)
```

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

- This is the same recurrence as for `reverse` and so $T(n) \;=\; \frac{n(n-1)}{2} \;=\; O(n^2)$

Merge: Complexity

```
merge [] l = l
merge l [] = l
merge (x:xs) (y:ys)
  | (x < y) = x : merge xs (y:ys)
  | otherwise = y : merge (x:xs) ys
```

Merge: Complexity

```
merge [] l = l
merge l [] = l
merge (x:xs) (y:ys)
  | (x < y) = x : merge xs (y:ys)
  | otherwise = y : merge (x:xs) ys
```

- Let n be the sum of the lengths of the two lists. Then

Merge: Complexity

```
merge [] l = l
merge l [] = l
merge (x:xs) (y:ys)
  | (x < y) = x : merge xs (y:ys)
  | otherwise = y : merge (x:xs) ys
```

- Let n be the sum of the lengths of the two lists. Then

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

  Thus, $T(n) = n$.

Mergesorting: Complexity

```
mergesort :: [Int] -> [Int]
mergesort [] = []
mergesort [x] = [x]
mergesort l = merge (mergesort fhalf)
                    (mergesort shalf)
              where
              fhalf = take n l
              shalf = drop n l
              n = div (length l) 2
```

Mergesorting: Complexity

```
mergesort :: [Int] -> [Int]
mergesort [] = []
mergesort [x] = [x]
mergesort l = merge (mergesort fhalf)
                    (mergesort shalf)
                where
                fhalf = take n l
                shalf = drop n l
                n = div (length l) 2
```

- $T(0) = T(1) = 1$ and

Mergesorting: Complexity

```
mergesort :: [Int] -> [Int]
mergesort [] = []
mergesort [x] = [x]
mergesort l = merge (mergesort fhalf)
                    (mergesort shalf)
              where
              fhalf = take n l
              shalf = drop n l
              n = div (length l) 2
```

- $T(0) \;=\; T(1) \;=\; 1$ and
- $T(n) \;=\; 2.T(n/2) + n + n + 1$

Mergesorting: Complexity

```
mergesort :: [Int] -> [Int]
mergesort [] = []
mergesort [x] = [x]
mergesort l = merge (mergesort fhalf)
                    (mergesort shalf)
              where
              fhalf = take n l
              shalf = drop n l
              n = div (length l) 2
```

- $T(0) \;=\; T(1) \;=\; 1$ and
- $T(n) \;=\; 2.T(n/2) + n + n + 1$
- Let us solve the recurrence $T(n) = 2.T(n/2) + c.n + d$ with $T(1) \;=\; 1$

Solving the mergesort recurrence

$$T(n) \; = \; 2.T(n/2) + c.n + d$$

Solving the mergesort recurrence

$$
\begin{aligned}
T(n) &= 2.T(n/2) + c.n + d \\
&= 2.(2.T(n/4) + c.n/2 + d) + c.n + d
\end{aligned}
$$

Solving the mergesort recurrence

$$
\begin{aligned}
T(n) &= 2.T(n/2) + c.n + d \\
&= 2.(2.T(n/4) + c.n/2 + d) + c.n + d \\
&= 4.T(n/4) + c.n + 2.d + c.n + d
\end{aligned}
$$

Solving the mergesort recurrence

$$\begin{aligned}
T(n) &= 2.T(n/2) + c.n + d \\
&= 2.(2.T(n/4) + c.n/2 + d) + c.n + d \\
&= 4.T(n/4) + c.n + 2.d + c.n + d \\
&= 4.T(n/4) + 2.c.n + 3.d
\end{aligned}$$

Solving the mergesort recurrence

$$\begin{aligned}
T(n) &= 2.T(n/2) + c.n + d \\
&= 2.(2.T(n/4) + c.n/2 + d) + c.n + d \\
&= 4.T(n/4) + c.n + 2.d + c.n + d \\
&= 4.T(n/4) + 2.c.n + 3.d \\
&= 4.(2.T(n/8) + c.n/4 + d) + 2.c.n + (1+2).d
\end{aligned}$$

Solving the mergesort recurrence

$$
\begin{aligned}
T(n) &= 2.T(n/2) + c.n + d \\
&= 2.(2.T(n/4) + c.n/2 + d) + c.n + d \\
&= 4.T(n/4) + c.n + 2.d + c.n + d \\
&= 4.T(n/4) + 2.c.n + 3.d \\
&= 4.(2.T(n/8) + c.n/4 + d) + 2.c.n + (1+2).d \\
&= 8.T(n/8) + c.n + 4.d + 2.c.n + 3.d
\end{aligned}
$$

Solving the mergesort recurrence

$$
\begin{aligned}
T(n) &= 2.T(n/2) + c.n + d \\
&= 2.(2.T(n/4) + c.n/2 + d) + c.n + d \\
&= 4.T(n/4) + c.n + 2.d + c.n + d \\
&= 4.T(n/4) + 2.c.n + 3.d \\
&= 4.(2.T(n/8) + c.n/4 + d) + 2.c.n + (1+2).d \\
&= 8.T(n/8) + c.n + 4.d + 2.c.n + 3.d \\
&= 8.T(n/8) + 3.c.n + (1+2+4).d
\end{aligned}
$$

Solving the mergesort recurrence

$$\begin{aligned}
T(n) &= 2.T(n/2) + c.n + d \\
&= 2.(2.T(n/4) + c.n/2 + d) + c.n + d \\
&= 4.T(n/4) + c.n + 2.d + c.n + d \\
&= 4.T(n/4) + 2.c.n + 3.d \\
&= 4.(2.T(n/8) + c.n/4 + d) + 2.c.n + (1 + 2).d \\
&= 8.T(n/8) + c.n + 4.d + 2.c.n + 3.d \\
&= 8.T(n/8) + 3.c.n + (1 + 2 + 4).d \\
&= 8.(2.T(n/16) + c.n/8 + d) + 3.c.n + (1 + 2 + 4).d
\end{aligned}$$

Solving the mergesort recurrence

$$\begin{aligned}
T(n) &= 2.T(n/2) + c.n + d \\
&= 2.(2.T(n/4) + c.n/2 + d) + c.n + d \\
&= 4.T(n/4) + c.n + 2.d + c.n + d \\
&= 4.T(n/4) + 2.c.n + 3.d \\
&= 4.(2.T(n/8) + c.n/4 + d) + 2.c.n + (1+2).d \\
&= 8.T(n/8) + c.n + 4.d + 2.c.n + 3.d \\
&= 8.T(n/8) + 3.c.n + (1+2+4).d \\
&= 8.(2.T(n/16) + c.n/8 + d) + 3.c.n + (1+2+4).d \\
&= 16.T(n/16) + c.n + 8.d + 3.c.n + (1+2+4).d
\end{aligned}$$

Solving the mergesort recurrence

$$
\begin{aligned}
T(n) &= 2.T(n/2) + c.n + d \\
&= 2.(2.T(n/4) + c.n/2 + d) + c.n + d \\
&= 4.T(n/4) + c.n + 2.d + c.n + d \\
&= 4.T(n/4) + 2.c.n + 3.d \\
&= 4.(2.T(n/8) + c.n/4 + d) + 2.c.n + (1+2).d \\
&= 8.T(n/8) + c.n + 4.d + 2.c.n + 3.d \\
&= 8.T(n/8) + 3.c.n + (1+2+4).d \\
&= 8.(2.T(n/16) + c.n/8 + d) + 3.c.n + (1+2+4).d \\
&= 16.T(n/16) + c.n + 8.d + 3.c.n + (1+2+4).d \\
&= 16.T(n/16) + 4.c.n + (1+2+4+8).d
\end{aligned}
$$

Solving the mergesort recurrence

$$
\begin{aligned}
T(n) &= 2.T(n/2) + c.n + d \\
&= 2.(2.T(n/4) + c.n/2 + d) + c.n + d \\
&= 4.T(n/4) + c.n + 2.d + c.n + d \\
&= 4.T(n/4) + 2.c.n + 3.d \\
&= 4.(2.T(n/8) + c.n/4 + d) + 2.c.n + (1+2).d \\
&= 8.T(n/8) + c.n + 4.d + 2.c.n + 3.d \\
&= 8.T(n/8) + 3.c.n + (1+2+4).d \\
&= 8.(2.T(n/16) + c.n/8 + d) + 3.c.n + (1+2+4).d \\
&= 16.T(n/16) + c.n + 8.d + 3.c.n + (1+2+4).d \\
&= 16.T(n/16) + 4.c.n + (1+2+4+8).d
\end{aligned}
$$

- Let $k = log_2(n)$. Then,

Solving the mergesort recurrence

$$
\begin{aligned}
T(n) &= 2.T(n/2) + c.n + d \\
&= 2.(2.T(n/4) + c.n/2 + d) + c.n + d \\
&= 4.T(n/4) + c.n + 2.d + c.n + d \\
&= 4.T(n/4) + 2.c.n + 3.d \\
&= 4.(2.T(n/8) + c.n/4 + d) + 2.c.n + (1+2).d \\
&= 8.T(n/8) + c.n + 4.d + 2.c.n + 3.d \\
&= 8.T(n/8) + 3.c.n + (1+2+4).d \\
&= 8.(2.T(n/16) + c.n/8 + d) + 3.c.n + (1+2+4).d \\
&= 16.T(n/16) + c.n + 8.d + 3.c.n + (1+2+4).d \\
&= 16.T(n/16) + 4.c.n + (1+2+4+8).d
\end{aligned}
$$

- Let $k = log_2(n)$. Then,

$$T(n) = n.T(1) + k.c.n + (1 + 2 + 4 \ldots 2^{k-1}).d$$

Solving the mergesort recurrence

$$
\begin{aligned}
T(n) &= 2.T(n/2) + c.n + d \\
&= 2.(2.T(n/4) + c.n/2 + d) + c.n + d \\
&= 4.T(n/4) + c.n + 2.d + c.n + d \\
&= 4.T(n/4) + 2.c.n + 3.d \\
&= 4.(2.T(n/8) + c.n/4 + d) + 2.c.n + (1+2).d \\
&= 8.T(n/8) + c.n + 4.d + 2.c.n + 3.d \\
&= 8.T(n/8) + 3.c.n + (1+2+4).d \\
&= 8.(2.T(n/16) + c.n/8 + d) + 3.c.n + (1+2+4).d \\
&= 16.T(n/16) + c.n + 8.d + 3.c.n + (1+2+4).d \\
&= 16.T(n/16) + 4.c.n + (1+2+4+8).d
\end{aligned}
$$

- Let $k = log_2(n)$. Then,

$$
\begin{aligned}
T(n) &= n.T(1) + k.c.n + (1+2+4\ldots 2^{k-1}).d \\
T(n) &\leq n + c.n.log_2(n) + d.2^k
\end{aligned}
$$

Solving the mergesort recurrence

$$\begin{aligned}
T(n) &= 2.T(n/2) + c.n + d \\
&= 2.(2.T(n/4) + c.n/2 + d) + c.n + d \\
&= 4.T(n/4) + c.n + 2.d + c.n + d \\
&= 4.T(n/4) + 2.c.n + 3.d \\
&= 4.(2.T(n/8) + c.n/4 + d) + 2.c.n + (1+2).d \\
&= 8.T(n/8) + c.n + 4.d + 2.c.n + 3.d \\
&= 8.T(n/8) + 3.c.n + (1+2+4).d \\
&= 8.(2.T(n/16) + c.n/8 + d) + 3.c.n + (1+2+4).d \\
&= 16.T(n/16) + c.n + 8.d + 3.c.n + (1+2+4).d \\
&= 16.T(n/16) + 4.c.n + (1+2+4+8).d
\end{aligned}$$

- Let $k = log_2(n)$. Then,

$$T(n) = n.T(1) + k.c.n + (1+2+4\ldots 2^{k-1}).d$$

$$\begin{aligned}
T(n) &\leq n + c.n.log_2(n) + d.2^k \\
T(n) &\leq c.n.log_2(n) + (2.d)n
\end{aligned}$$

Solving the mergesort recurrence

$$
\begin{aligned}
T(n) &= 2.T(n/2) + c.n + d \\
&= 2.(2.T(n/4) + c.n/2 + d) + c.n + d \\
&= 4.T(n/4) + c.n + 2.d + c.n + d \\
&= 4.T(n/4) + 2.c.n + 3.d \\
&= 4.(2.T(n/8) + c.n/4 + d) + 2.c.n + (1+2).d \\
&= 8.T(n/8) + c.n + 4.d + 2.c.n + 3.d \\
&= 8.T(n/8) + 3.c.n + (1+2+4).d \\
&= 8.(2.T(n/16) + c.n/8 + d) + 3.c.n + (1+2+4).d \\
&= 16.T(n/16) + c.n + 8.d + 3.c.n + (1+2+4).d \\
&= 16.T(n/16) + 4.c.n + (1+2+4+8).d
\end{aligned}
$$

- Let $k = log_2(n)$. Then,

$$T(n) = n.T(1) + k.c.n + (1 + 2 + 4 \ldots 2^{k-1}).d$$

$$
\begin{aligned}
T(n) &\leq n + c.n.log_2(n) + d.2^k \\
T(n) &\leq c.n.log_2(n) + (2.d)n
\end{aligned}
$$

- Thus $T(n) = O(n.logn)$.

Complexity of Quicksort

```
quicksort :: [Int] -> [Int]
quicksort [] = []
quicksort (x:xs) = (quicksort lower) ++
                   [splitter] ++
                   (quicksort upper)
  where
    splitter = x
    lower    = filter (<= x) xs
    upper    = filter (> x) xs
```

Complexity of Quicksort

```
quicksort :: [Int] -> [Int]
quicksort [] = []
quicksort (x:xs) = (quicksort lower) ++
                   [splitter] ++
                   (quicksort upper)
  where
    splitter = x
    lower    = filter (<= x) xs
    upper    = filter (> x) xs
```

- If the splitter is always the median, the recurrence is $T(n) = 2.T(n/2) + c.n + d$

Complexity of Quicksort

```
quicksort :: [Int] -> [Int]
quicksort [] = []
quicksort (x:xs) = (quicksort lower) ++
                   [splitter] ++
                   (quicksort upper)
  where
    splitter = x
    lower    = filter (<= x) xs
    upper    = filter (> x) xs
```

- If the splitter is always the median, the recurrence is $T(n) = 2.T(n/2) + c.n + d$
- If the splitter is always the smallest (or largest) element then the recurrence is $T(n) = T(n-1) + c.n$

Complexity of Quicksort

```
quicksort :: [Int] -> [Int]
quicksort [] = []
quicksort (x:xs) = (quicksort lower) ++
                   [splitter] ++
                   (quicksort upper)
  where
    splitter = x
    lower    = filter (<= x) xs
    upper    = filter (> x) xs
```

- If the splitter is always the median, the recurrence is
  $T(n) = 2.T(n/2) + c.n + d$
- If the splitter is always the smallest (or largest) element then the recurrence is
  $T(n) = T(n-1) + c.n$
- Thus the worst case complexity of Quicksort is $O(n^2)$.

Sorting by keys

- Student names and Marks.
  - maintain names and marks of a group of students
  - list of pairs `(name,marks)` where `name::String` and `marks::Int`

Sorting by keys

- Student names and Marks.
  - maintain names and marks of a group of students
  - list of pairs `(name,marks)` where `name::String` and `marks::Int`
- What if we want the list of students sorted by marks?
- Equip `isort` with a function to pick up the key.

Sorting by keys

- Student names and Marks.
  - maintain names and marks of a group of students
  - list of pairs `(name,marks)` where `name::String` and `marks::Int`
- What if we want the list of students sorted by marks?
- Equip `isort` with a function to pick up the key.

```
isort f [] = []
isort f (x:xs) = insert f x (isort f xs)

insert f x [] = [x]
insert f x (y:ys)
        | (f x) < (f y) = x:y:ys
        | otherwise     = y:(insert f x ys)
```

- `isort snd [("Nikhil", 75), ("Lavanya", 71)]`

Stable sorting

- Sort by marks . . .

Stable sorting

- Sort by marks . . .

  . . . and students with the same marks should be sorted alphabetically by name

Stable sorting

- Sort by marks . . .

  . . . and students with the same marks should be sorted alphabetically by name

```
isort snd ( isort fst [("Nikhil",75), ("Lavanya",71),
    ("Anirudha",70), ("Ananya",75), ("Badri",70)]   )
```

Stable sorting

- Sort by marks ...

  ...and students with the same marks should be sorted alphabetically by name

  ```
  isort snd ( isort fst [("Nikhil",75), ("Lavanya",71),
      ("Anirudha",70), ("Ananya",75), ("Badri",70)]   )
  ```

- Does not work!

Stable sorting

- Sort by marks . . .

  . . . and students with the same marks should be sorted alphabetically by name

```
isort snd ( isort fst [("Nikhil",75), ("Lavanya",71),
    ("Anirudha",70), ("Ananya",75), ("Badri",70)]   )
```

- Does not work!
- `isort` messes around the order between equal elements!

Stable sorting

- Sort by marks ...

  ...and students with the same marks should be sorted alphabetically by name

  ```
  isort snd ( isort fst [("Nikhil",75), ("Lavanya",71),
      ("Anirudha",70), ("Ananya",75), ("Badri",70)]   )
  ```

- Does not work!

- `isort` messes around the order between equal elements!

- A sorting algorithm is stable if relative order of equal elements is left unaltered.

Stable Sorting

- It is easy to turn `isort` into a stable sort.

Stable Sorting

- It is easy to turn `isort` into a stable sort.

```
isort f [] = []
isort f (x:xs) = insert f x (isort f xs)

insert f x [] = [x]
insert f x (y:ys)
        | (f x) <= (f y) = x:y:ys
        | otherwise      = y:(insert f x ys)
```

Stable Sorting

- It is easy to turn `isort` into a stable sort.

```
isort f [] = []
isort f (x:xs) = insert f x (isort f xs)

insert f x [] = [x]
insert f x (y:ys)
        | (f x) <= (f y) = x:y:ys
        | otherwise      = y:(insert f x ys)
```

- With a similar change `mergesort` can also be made stable.

Stable Sorting

- It is easy to turn `isort` into a stable sort.

```
isort f [] = []
isort f (x:xs) = insert f x (isort f xs)

insert f x [] = [x]
insert f x (y:ys)
        | (f x) <= (f y) = x:y:ys
        | otherwise      = y:(insert f x ys)
```

- With a similar change `mergesort` can also be made stable.
- Same with `quicksort`

minout

- `minout :: [Int] -> Int`
  `minout l` is the minimum nonnegative number not in `l` assuming that all elements in `l` are nonnegative and distinct.
  - `minout [3,1,2] = 0`
  - `minout [1,5,3,0,2] = 4`
  - `minout [11,5,3,0] = 1`
- How do we compute `minout`?

`minout`: direct solution

- Here is one way to do this:

`minout`: direct solution

- Here is one way to do this:

```
minoutAux i l
  | (elem i l) = minoutAux (i+1) l
  | otherwise  = i

minout l = minoutAux 0 l
```

minout: direct solution

- Here is one way to do this:

```
minoutAux i l
  | (elem i l) = minoutAux (i+1) l
  | otherwise  = i

minout l = minoutAux 0 l
```

- What is the complexity of this program?

`minout`: direct solution

- Here is one way to do this:

```
minoutAux i l
  | (elem i l) = minoutAux (i+1) l
  | otherwise  = i

minout l = minoutAux 0 l
```

- What is the complexity of this program?
  - Observe that the answer lies within `0 ... length(l)`.

`minout`: direct solution

- Here is one way to do this:

```
minoutAux i l
  | (elem i l) = minoutAux (i+1) l
  | otherwise  = i

minout l = minoutAux 0 l
```

- What is the complexity of this program?
  - Observe that the answer lies within `0 ... length(l)`.
  - So, `minoutAux i l` is evaluated for at most `n+1` values of `i`

`minout`: direct solution

- Here is one way to do this:

```
minoutAux i l
  | (elem i l) = minoutAux (i+1) l
  | otherwise  = i

minout l = minoutAux 0 l
```

- What is the complexity of this program?
  - Observe that the answer lies within `0 ... length(l)`.
  - So, `minoutAux i l` is evaluated for at most `n+1` values of `i`
  - Each evaluation takes $O(n)$ steps (since `elem` takes $O(n)$ steps).

`minout`: direct solution

- Here is one way to do this:

```
minoutAux i l
  | (elem i l) = minoutAux (i+1) l
  | otherwise  = i

minout l = minoutAux 0 l
```

- What is the complexity of this program?
  - Observe that the answer lies within `0 ... length(l)`.
  - So, `minoutAux i l` is evaluated for at most `n+1` values of `i`
  - Each evaluation takes $O(n)$ steps (since `elem` takes $O(n)$ steps).
- Thus this program takes $O(n^2)$ steps.

`minout` via sorting

- Here is another way to solve the problem.

`minout` via sorting

- Here is another way to solve the problem.
    - Sort the list in ascending order.

`minout` via sorting

- Here is another way to solve the problem.
  - Sort the list in ascending order.
  - `zip` it with the list `[0,1,...,(length l)-1]`

`minout` via sorting

- Here is another way to solve the problem.
    - Sort the list in ascending order.
    - `zip` it with the list `[0,1,...,(length l)-1]`
    - Find the first pair in this list of pairs which is unequal values and report the second component.

`minout` via sorting

- Here is another way to solve the problem.
  - Sort the list in ascending order.
  - `zip` it with the list `[0,1,...,(length l)-1]`
  - Find the first pair in this list of pairs which is unequal values and report the second component.

```
minout l = funequal (zip (sort l) [0..(length l)])
funequal ((x,y):ls)
  | (x /= y)   = y
  | otherwise = funequal ls
```

`minout` via sorting

- Here is another way to solve the problem.
  - Sort the list in ascending order.
  - `zip` it with the list `[0,1,...,(length l)-1]`
  - Find the first pair in this list of pairs which is unequal values and report the second component.

```
minout l = funequal (zip (sort l) [0..(length l)])
funequal ((x,y):ls)
  | (x /= y)   = y
  | otherwise = funequal ls
```

- Fix this to work when the answer is `length l`.

`minout` via sorting

- Here is another way to solve the problem.
  - Sort the list in ascending order.
  - `zip` it with the list `[0,1,...,(length l)-1]`
  - Find the first pair in this list of pairs which is unequal values and report the second component.

```
minout l = funequal (zip (sort l) [0..(length l)])
funequal ((x,y):ls)
  | (x /= y)   = y
  | otherwise = funequal ls
```

- Fix this to work when the answer is `length l`.

  What is the complexity of this function?

`minout` via sorting

- Here is another way to solve the problem.
  - Sort the list in ascending order.
  - `zip` it with the list `[0,1,...,(length l)-1]`
  - Find the first pair in this list of pairs which is unequal values and report the second component.

```
minout l = funequal (zip (sort l) [0..(length l)])
funequal ((x,y):ls)
  | (x /= y)  = y
  | otherwise = funequal ls
```

- Fix this to work when the answer is `length l`.

  What is the complexity of this function?

  $O(n.logn) + O(n) = O(n.logn)$

The Divide and Conquer Solution

The Divide and Conquer Solution

- Construct two lists
  - `fhalf` containing all the elements smaller than `(length l) `div` 2`
  - `shalf` containing all the elements at least as big as `(length l) `div` 2`

The Divide and Conquer Solution

- Construct two lists
  - `fhalf` containing all the elements smaller than `(length l) ‘div‘ 2`
  - `shalf` containing all the elements at least as big as `(length l) ‘div‘ 2`
- Check if `(length fhalf) == (length l) ‘div‘ 2`
  - If no, answer lies in first half
  - If yes, answer lies in second half

`minout` in Haskell

```
minout [] = 0
minout [x]
  | (x==0) = 1
  | otherwise = 0

minout l
  | length fhalf < m = minout fhalf
  | otherwise = m + minout (shift m shalf)
  where
  m = div (length l) 2
  fhalf = filter (< m) l
  shalf = filter (>= m) l
  shift i [] = []
  shift i (x:xs) = (x-i) : shift i xs
```

`minout` in Haskell

```
minout [] = 0
minout [x]
  | (x==0) = 1
  | otherwise = 0

minout l
  | length fhalf < m = minout fhalf
  | otherwise = m + minout (shift m shalf)
  where
  m = div (length l) 2
  fhalf = filter (< m) l
  shalf = filter (>= m) l
  shift i [] = []
  shift i (x:xs) = (x-i) : shift i xs
```

- What is its complexity?

`minout` in Haskell

```haskell
minout [] = 0
minout [x]
  | (x==0) = 1
  | otherwise = 0

minout l
  | length fhalf < m = minout fhalf
  | otherwise = m + minout (shift m shalf)
  where
  m = div (length l) 2
  fhalf = filter (< m) l
  shalf = filter (>= m) l
  shift i [] = []
  shift i (x:xs) = (x-i) : shift i xs
```

- What is its complexity?
- `T(n) = T(n/2) + c.n`

Complexity of `minout`

$$T(n) \;=\; T(n/2) + c.n$$

Complexity of `minout`

$$
\begin{aligned}
T(n) &= T(n/2) + c.n \\
&= T(n/4) + c.n/2 + c.n
\end{aligned}
$$

Complexity of `minout`

$$
\begin{aligned}
T(n) &= T(n/2) + c.n \\
&= T(n/4) + c.n/2 + c.n \\
&= T(n/8) + c.n/4 + c.n/2 + c.n
\end{aligned}
$$

...

Complexity of `minout`

$$
\begin{aligned}
T(n) &= T(n/2) + c.n \\
&= T(n/4) + c.n/2 + c.n \\
&= T(n/8) + c.n/4 + c.n/2 + c.n
\end{aligned}
$$

...

$$
= T(n/2^{logn}) + c.1 + \ldots c.n/4 + c.n/2 + c.n
$$

Complexity of `minout`

$$\begin{aligned} T(n) &= T(n/2) + c.n \\ &= T(n/4) + c.n/2 + c.n \\ &= T(n/8) + c.n/4 + c.n/2 + c.n \end{aligned}$$

...

$$\begin{aligned} &= T(n/2^{logn}) + c.1 + \ldots c.n/4 + c.n/2 + c.n \\ &= d.n \;=\; O(n) \end{aligned}$$

Complexity of `minout`

$$\begin{aligned} T(n) &= T(n/2) + c.n \\ &= T(n/4) + c.n/2 + c.n \\ &= T(n/8) + c.n/4 + c.n/2 + c.n \\ \ldots \\ &= T(n/2^{logn}) + c.1 + \ldots c.n/4 + c.n/2 + c.n \\ &= d.n = O(n) \end{aligned}$$

Divide and Conquer need not always work. See lecture notes.

Testing if a number is prime

- Check if a given number n is a prime.

Testing if a number is prime

- Check if a given number n is a prime.
- Naive algorithm:

```
prime n = and (map f [1..(div n 2)])
  where
    f i = ((n mod i) /= 0)
```

Testing if a number is prime

- Check if a given number n is a prime.
- Naive algorithm:

```
prime n = and (map f [1..(div n 2)])
  where
    f i = ((n mod i) /= 0)
```

- This function takes $O(n)$ to decide if n is a prime.

Testing if a number is prime

- Check if a given number n is a prime.
- Naive algorithm:

```
prime n = and (map f [1..(div n 2)])
  where
    f i = ((n mod i) /= 0)
```

- This function takes $O(n)$ to decide if n is a prime.
- Is this an efficient algorithm?

Testing if a number is prime

- Check if a given number n is a prime.
- Naive algorithm:

```
prime n = and (map f [1..(div n 2)])
  where
    f i = ((n mod i) /= 0)
```

- This function takes $O(n)$ to decide if n is a prime.
- Is this an efficient algorithm?
- Is it efficient to take 2^{64} steps to decide if a 64 bit number is prime?

Testing if a number is prime

- Check if a given number n is a prime.
- Naive algorithm:

```
prime n = and (map f [1..(div n 2)])
  where
    f i = ((n mod i) /= 0)
```

- This function takes $O(n)$ to decide if n is a prime.
- Is this an efficient algorithm?
- Is it efficient to take 2^{64} steps to decide if a 64 bit number is prime?
- The size of the input is the number of bits required to write it down.

Testing if a number is prime

- Check if a given number n is a prime.
- Naive algorithm:

```
prime n = and (map f [1..(div n 2)])
  where
    f i = ((n mod i) /= 0)
```

- This function takes $O(n)$ to decide if n is a prime.
- Is this an efficient algorithm?
- Is it efficient to take 2^{64} steps to decide if a 64 bit number is prime?
- The size of the input is the number of bits required to write it down.
- The above algorithm is takes 2^n steps to decide if a number of size n is prime.