# Sorting

# The importance of sorting

Data that is unorganised is a pain to use

It is hence important to know how to sort (put things in order)

This lecture looks at **sorting** lists, with a focus on complexity

# Insertion Sort

Problem: sort a list of numbers into ascending order.

Given a list `[7,3,9,2]`

1. Suppose the tail is sorted somehow to get `[2,3,9]`
2. Insert the head element 7 at the proper position in the sorted tail to get `[2,3,7,9]`

The tail is sorted ‘somehow’ by recursion, of course!

# Insertion

```
insert :: Ord a => a -> [a] -> [a]
insert x list = case list of
  [] -> [x]
  y:ys
    | x <= y    -> x : y : ys
    | otherwise -> y : insert x ys
```

If the input list is sorted, the output list will be also

# Complexity of Insertion

**Best case:** compare only to the head and insert immediately – O(1)

**Worst case:** look through the whole list and insert at the end – O(n)

**Average case:** insert after half the list – O(n)

# Insertion Sort

```
iSort :: Ord a => [a] -> [a]
iSort = foldr insert []
```

Make sure that you understand this definition

- What are the types of its components?
- What does it do on the base case?
- What does it do on the step case?

Most importantly, halfway through the algorithm, which part of the list is sorted, and which is not?

# Complexity of Insertion Sort

**Best case:** the list is already sorted, so each element is put onto the head of the list immediately – O(n)

**Worst case:** the list is sorted in reverse order!

Each element needs to be compared to the tail of the list.

What is the cost of this?

# Worst Case Complexity of Insertion Sort

Say sorting the last element of the list takes 1 step

Then sorting the next element takes 2 steps, and so on, with n steps for the first element of the list

So we have the sum $1 + 2 + 3 + \ldots + (n-2) + (n-1) + n$

$= (n/2) * (n + 1)$

Delete the constants…

$= O(n^2)$

# Average Case Complexity of Insertion Sort

The average case complexity of `insert` was the same (with respect to big-O notation) as the worst case.

So we again have $1 + 2 + 3 + \ldots + (n-2) + (n-1) + n$

(ignoring the constants we don't care about)

$= O(n^2)$

# A Simpler Algorithm? Selection Sort

```
sSort list = case list of
  [] -> []
  _  -> minOfList : sSort (delete minOfList list)
    where
      minOfList = minimum list
```

- `delete` comes from `import Data.List`
- $O(n^2)$ in all cases
  - Think about `minimum`, which is $O(n)$, running over and over

# Merge Sort

Merge Sort is a ‘**divide and conquer**’ algorithm.

Intuition: sorting a list is hard…

But sorting a list of half the size would be easier (**Divide**)

And it is easy to combine two sorted lists into one sorted list (**Conquer**)

# Merge

```
merge :: Ord a => [a] -> [a] -> [a]
merge list1 list2 = case (list1,list2) of
  (list,[]) -> list
  ([],list) -> list
  (x:xs,y:ys)
    | x <= y    -> x : merge xs (y:ys)
    | otherwise -> y : merge (x:xs) ys
```

If we have two lists sorted somehow, their merge is also sorted

Somehow = recursion, of course!

# Complexity of Merge

Assuming that the two lists are of equal length, let n be the list of the two combined.

**Best case:** one list contains elements all smaller than the other. Only need roughly n/2 comparisons – O(n)

**Worst case:** Every element gets looked at, but we are always ‘dealing with’ one element per step – O(n)

**Average case:** Where best case = worst case, this is obvious!

# Merge Sort

```
mSort :: Ord a => [a] -> [a]
mSort list = case list of
  []  -> []
  [x] -> [x]
  _   -> merge (mSort firsthalf) (mSort secondhalf)
    where
      firsthalf  = take half list
      secondhalf = drop half list
      half   = (length list) `div` 2
```

# Complexity of Merge Sort

Much like Insertion Sort, at every step Merge sort is O(n)

- Length, Taking the first half, Taking the second half, Merging all O(n)
- At the next step there are twice as many `mSorts`, but each is working with a list only half the length, so still O(n)!

But how many steps are there?

We half the list lengths repeatedly, i.e. call on n, n/2, ...,16, 8, 4, 2

So if we started with length 16, there would 4 steps = $\log_2(16)$

Cost of O(n) per step * O(log n) steps = O(n log n)

- Best, worst, and average the same!

# Comparison

| $n$ | $n \log_2 n$ | $n^2$ | multiplier |
|---|---|---|---|
| 2 | 2 | 4 | 2 |
| 16 | 64 | 256 | 4 |
| 128 | 896 | 16,384 | 18 |
| 1,024 | 10,240 | 1,048,576 | 102 |
| 8,192 | 106,496 | 67,108,864 | 630 |
| 65,536 | 1,048,576 | 4,294,967,296 | 4,096 |

# Comparison of Sorting Algorithms

In the best case, insertion sort outperforms merge sort. But on average, and in the worst case, merge sort is superior

- In the Data.List library `sort :: Ord a => [a] -> [a]` is merge sort (somewhat optimised versus the one in these slides)

But merge sort is quite space inefficient, so other algorithms, such as ‘quick sort‘, can be preferred sometimes

- Time complexity of average case $O(n \log n)$, but worst case $O(n^2)$

Other algorithms, e.g. ‘radix sort‘, can outperform these generic approaches if you know a lot about the nature of your inputs