Search

Search

One of the most common problems for an algorithm to solve is to **search** for some data in some data structure.

- E.g. some problems in AI are posed as search through a space of decisions / analyses

We will discuss the (time) complexity of search in

- a list
- a binary search tree

Simplifying assumptions

- Search can be defined so long as the underlying data type is in Eq. But `(==)` might be an expensive operation
  - However this is not relevant for the complexity analysis of the ad hoc polymorphic search algorithm on its own
- We assume that the data structure we are searching in is completely computed before we start searching
  - Not necessarily true due to laziness, but again, not relevant to the search algorithm on its own

Search in a list

```
elem :: Eq a => a -> [a] -> Bool
elem x list = case list of
  []   -> False
  y:ys -> x == y || elem x ys
```

(This is not identical to the definition of `elem` in the Prelude, but it gives the idea)

Best case analysis

Given a finite list of length `n`, what is the best case scenario for `elem`?

- We can assume `n` is large, and so not empty

Best case – the searched-for element is the head of the list

- Check if the list is empty
- Check if its main constructor is cons
- Check if `x == y`
- Run `||`
- Return `True`

Best case analysis

These operations take time

- maybe a lot of time if `(x == y)` is expensive

But the time does not depend on the length of the list.

Therefore in the best case `elem` is **constant** – **O**(1)

Worst case analysis

The worst case – the searched-for element is not in the list at all

- n+1 checks if the list is empty
- n checks if main constructor is cons
- n checks if `x == y`
- n calls to `||`
- n recursive calls
- Return `False`

Worst case analysis

Remember that anything that increases as the input size increases will eventually dominate anything that is constant, so remove all constants

- n+**1** checks if the list is empty
- n checks if main constructor is cons
- n checks if `x == y`
- n calls to `||`
- n recursive calls
- **`Return False`**

Worst case analysis

Remember that anything that increases as the input size increases will eventually dominate anything that is constant, so remove all constants

- **O**(n) checks if the list is empty
- n checks if main constructor is cons
- n checks if x == y
- n calls to ||
- n recursive calls

Worst case analysis

The function describing the time spent searching the list will then be

n * (cost of empty check + cost of cons check + cost of `(==)` + cost of `(||)` + cost of recursive call) + some constant

But complexity analysis asks us to ignore constants, whether they are multiplied or added, so…

In the worst case, `elem` is **linear** – **O**(n).

Note that the case where the element is at the end of the list is also **O**(n).

Average case analysis

Average cases can be tricky – on average, how likely is it that an element will be in a list?

But across the cases that the element is in the list, we can obviously define an average case – where the element is halfway through.

But if searching the whole list takes time **O**(n), then searching half a list takes time proportional to n/2 – and this is also **O**(n)

Search in a binary search tree

```
Data BSTree a = Null | Node (BSTree a) a (BSTree a)

BSElem :: Ord a => a -> BSTree a -> Bool
BSElem x tree = case tree of
  Null -> False
  Node left node right
    | x < node  -> BSElem x left
    | x == node -> True
    | otherwise -> BSElem x right
```

Search in a binary search tree

In the best case, we are again in **O**(1). What about worst case?

Simplifying assumptions:

- The `BSTree` really is a binary search tree – all operations on it have kept it correctly ordered
- The tree is **balanced** – `Node Null 1 (Node Null 2 (Node Null 3 (Node Null 4)))` is technically a `BSTree`, but it is not any faster to search than a list!

Logarithms

Logarithms are the inverse of exponentiation:

If $b^c = a$ then $\log_b a = c$

- $\log_2 1 = 0$ because $2^0 = 1$
- $\mathrm{Log}_2 2 = 1$ because $2^1 = 2$
- $\log_2 4 = 2$ because $2^2 = 4$
- $\log_2 8 = 3$ because $2^3 = 8$
- $\log_2 16 = 4$ …

Height of a binary search tree

In a balanced `BStree` the **height** h as a function of its **size** n is approximately

$$h = \log_2(n + 1)$$

e.g. a tree of height 4 contains (at most) 15 elements.

We can ignore the '+1', and, for large n, $\log_2 n$ will dominate the number of any missing elements at the bottom layer.

In fact the base 2 is also irrelevant for big O analysis – we can say that h is in **O**(log n).

Worst case analysis of `BSElem`

In the worst case BSElem looks at one element for every layer of the tree, so it is in **O**(log n).

In the average height of an element in a tree can be approximated as $\texttt{log}_2$(n-1), so the average case analysis for search is again **O**(log n).

Comparison

n	$\log_2 n$	multiplier
2	1	2
16	4	4
128	7	18
1,024	10	102
8,192	13	630
65,536	16	4,096
524,288	19	27,594