Sorting Algorithms

Having to sort a list is an issue that comes up all the time when you are writing programs.

Sorting algorithms are a standard topic in introductory courses in Computer Science, not only because sorting itself is an important issue, but also because it is particularly suited to demonstrating that there can be several very different solutions (algorithms) to the same problem, and that it can be useful and instructive to compare these alternative approaches.

In this lecture, we are going to introduce a couple of different sorting algorithms, discuss their implementation in Prolog, and analyse their complexity.

Aim

We want to implement a predicate that will take an *ordering relation* and an *unsorted list* and return a *sorted list*. Examples:

```
?- sort(<, [3,8,5,1,2,4,6,7], List).
List = [1, 2, 3, 4, 5, 6, 7, 8]
Yes

?- sort(>, [3,8,5,1,2,4,6,7], List).
List = [8, 7, 6, 5, 4, 3, 2, 1]
Yes

?- sort(is_bigger, [horse,elephant,donkey], List).
List = [elephant, horse, donkey]
Yes
```

Auxiliary Predicate to Check Orderings

We are going to use the following predicate to check whether two given terms `A` and `B` are ordered with respect to the ordering relation `Rel` supplied:

```
check(Rel, A, B) :-
  Goal =.. [Rel,A,B],
  call(Goal).
```

Remark: In SWI Prolog, you could also use just `Goal` instead of `call(Goal)` in the last line of the program.

Here are two examples of `check/3` in action:

```
?- check(is_bigger, elephant, monkey).
Yes

?- check(<, 7, 5).
No
```

Bubblesort

The first sorting algorithm we are going to look into is called *bubblesort*—the way it operates is supposedly reminiscent of bubbles floating up in a glass of spa rood.

This algorithm works as follows:

- Go through the list from left to right until you find a pair of consecutive elements that are ordered the wrong way round. Swap them.
- Repeat the above until you can go through the full list without encountering such a pair. Then the list is sorted.

Try sorting the list `[3, 7, 20, 16, 4, 46]` this way ...

Bubblesort in Prolog

This predicate calls `swap/3` and then continues recursively. If `swap/3` fails, then the current list is sorted and can be returned:

```
bubblesort(Rel, List, SortedList) :-
  swap(Rel, List, NewList), !,
  bubblesort(Rel, NewList, SortedList).

bubblesort(_, SortedList, SortedList).
```

Go recursively through a list until you finds a pair `A/B` to swap and return the new list, or fail if there is no such pair:

```
swap(Rel, [A,B|List], [B,A|List]) :-
  check(Rel, B, A).

swap(Rel, [A|List], [A|NewList]) :-
  swap(Rel, List, NewList).
```

Examples

Just to prove that it really works:

```
?- bubblesort(<, [5,3,7,5,2,8,4,3,6], List).
List = [2, 3, 3, 4, 5, 5, 6, 7, 8]
Yes

?- bubblesort(is_bigger, [donkey,horse,elephant], List).
List = [elephant, horse, donkey]
Yes

?- bubblesort(@<, [donkey,horse,elephant], List).
List = [donkey, elephant, horse]
Yes
```

An Improvement

The version of bubblesort we have given before can be improved upon. For the version presented, we know that we are going to have to do a lot of redundant comparisons:

> *Suppose we have just swapped elements 100 and 101. Then in the next round, the earliest we are going to find an unordered pair is after 99 comparisons (because the first 99 elements have already been sorted in previous rounds).*

This problem can be avoided, by *continuing* to swap elements and only to return to the front of the list once we have reached its end.

The Prolog implementation is just a little more complicated ...

Improved Bubblesort in Prolog

```
bubblesort2(Rel, List, SortedList) :-
  swap2(Rel, List, NewList), % this now always succeeds
  List \= NewList, !,        % check there's been a swap
  bubblesort2(Rel, NewList, SortedList).

bubblesort2(_, SortedList, SortedList).

swap2(Rel, [A,B|List], [B|NewList]) :-
  check(Rel, B, A),
  swap2(Rel, [A|List], NewList). % continue!

swap2(Rel, [A|List], [A|NewList]) :-
  swap2(Rel, List, NewList).

swap2(_, [], []). % new base case: reached end of list
```

Complexity Analysis

Here we are only going to look into the *time complexity* (rather than the space complexity) of sorting algorithms.

Throughout, let n be the length of the list to be sorted. This is the obvious parameter by which to measure the *problem size*.

We are going to measure the *complexity of an algorithm* in terms of the *number of primitive comparison operations* (*i.e.* calls to `check/3` in Prolog) required by that algorithm to sort a list of length n. This is a reasonable approximation of actual runtimes.

As with search algorithms earlier on in the course, we are going to be interested in what happens to the complexity of solving a problem with a given algorithm when we increase the problem size.

Will it be exponential in n? Or linear? Or something in between?

Big-O Notation: Recap

Let $f : \mathbb{N} \to \mathbb{N}$ and $g : \mathbb{N} \to \mathbb{N}$ be two functions mapping natural numbers to natural numbers.

Think of f as computing, for any problem size n, the worst-case time complexity $f(n)$. This may be a rather complicated function.

Think of g as a function that may be a "good approximation" of f and that is more convenient when speaking about complexities. The Big-O Notation is a way of making the idea of a suitable approximation mathematically precise.

We say that $f(n)$ is in $O(g(n))$ iff there exist an $n_0 \in \mathbb{N}$ and a $c \in \mathbb{R}^+$ such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.

That is, from some n_0 onwards, the difference between f and g will be at most some constant factor c.

Examples

(1) Let $f(n) = 5 \cdot n^2 + 20$. Then $f(n)$ is in $O(n^2)$.
Proof: Use $c = 6$ and $n_0 = 5$.

(2) Let $f(n) = n + 1000000$. Then $f(n)$ is in $O(n)$.
Proof: Use $c = 2$ and $n_0 = 1000000$ (or vice versa).

(3) Let $f(n) = 5 \cdot n^2 + 20$. Then $f(n)$ is also in $O(n^3)$, but this is not very interesting. We want complexity classes to be "sharp".

(4) Let $f(n) = 500 \cdot n^{200} + n^{17} + 1000$. Then $f(n)$ is in $O(2^n)$.
Proof: Use $c = 1$ and $n_0 = 3000$. In general, *exponential* functions always grow much faster than any *polynomial* function. So $O(2^n)$ is not at all a sharp complexity class for f. A better choice would be $O(n^{200})$.

Complexity of Bubblesort

How many comparisons does bubblesort perform in the worst case? Suppose we are using the improved version of bubblesort ...

In the worst case, the list is presented exactly the wrong way round, as in the following example:

```
?- bubblesort2(<, [10,9,8,7,6,5,4,3,2,1], List).
```

The algorithm will first move `10` to the end of the list, then `9`, etc.

In each round, we have to go through full list, *i.e.* make $n-1$ comparisons. And there are n rounds (one for each element to be moved). Hence, we require $n \cdot (n-1)$ comparisons.

$\leadsto$ Hence, the complexity of improved bubblesort is $O(n^2)$.

Home entertainment: The complexity of our original version of bubblesort is actually $O(n^3)$. The exact number of comparisons required is $\frac{1}{12} \cdot n \cdot (n-1) \cdot (2n+2) + (n-1)$ ... try to prove it!

Quicksort

The next sorting algorithm we consider is called *quicksort*. It works as follows (for a non-empty list):

- Select an arbitrary element `X` from the list.
- Split the remaining elements into a list `Left` containing all the elements preceding `X` in the ordering relation, and a list `Right` containing all the remaining elements.
- Sort `Left` and `Right` using quicksort (recursion), resulting in `SortedLeft` and `SortedRight`, respectively.
- Return the result: `SortedLeft ++ [X] ++ SortedRight`.

How fast quicksort runs will depend on the choice of `X`. In Prolog, we are simply going to select the head of the unsorted list.

Quicksort in Prolog

Sorting the empty list results in the empty list (base case):

```
quicksort(_, [], []).
```

For the recursive rule, we first remove the `Head` from the unsorted list and split the `Tail` into those elements preceding `Head` wrt. the ordering `Rel` (list `Left`) and the remaining elements (list `Right`). Then `Left` and `Right` are being sorted, and finally everything is put together to return the full sorted list:

```
quicksort(Rel, [Head|Tail], SortedList) :-
  split(Rel, Head, Tail, Left, Right),
  quicksort(Rel, Left, SortedLeft),
  quicksort(Rel, Right, SortedRight),
  append(SortedLeft, [Head|SortedRight], SortedList).
```

Splitting Lists

We still need to implement `split/5`. This predicate takes an ordering relation, an element, and a list, and returns two lists: one containing the elements from the input list preceding the input element wrt. the input ordering relation, and one containing the remaining elements from the input list (both unsorted).

```
split(_, _, [], [], []).

split(Rel, Middle, [Head|Tail], [Head|Left], Right) :-
  check(Rel, Head, Middle), !,
  split(Rel, Middle, Tail, Left, Right).

split(Rel, Middle, [Head|Tail], Left, [Head|Right]) :-
  split(Rel, Middle, Tail, Left, Right).
```

Testing `split/5`

The following example demonstrates how `split/5` works:

```
?- split(<, 20, [18,7,21,15,20,55,7,8,87], X, Y).
X = [18, 7, 15, 7, 8]
Y = [21, 20, 55, 87]
Yes
```

Quicksort Examples

A couple of examples demonstrating that quicksort works:

```
?- quicksort(>, [2,4,5,3,6,5,1], List).
List = [6, 5, 5, 4, 3, 2, 1]
Yes

?- quicksort(is_bigger, [elephant,donkey,horse], List).
List = [elephant, horse, donkey]
Yes
```

Complexity of Splitting

To analyse the complexity of quicksort, we first analyse the *complexity of splitting*, a crucial sub-routine of the algorithm.

Given a list `L` and an element `X`, how many comparisons are required to divide the elements in `L` into those that are to be placed to the left and those that are to be placed to the right of `X`?

Let n be the length of `[X|L]`. Clearly, we require exactly $n-1$ comparison operations. Hence, the complexity of splitting in $O(n)$.

Complexity of Quicksort

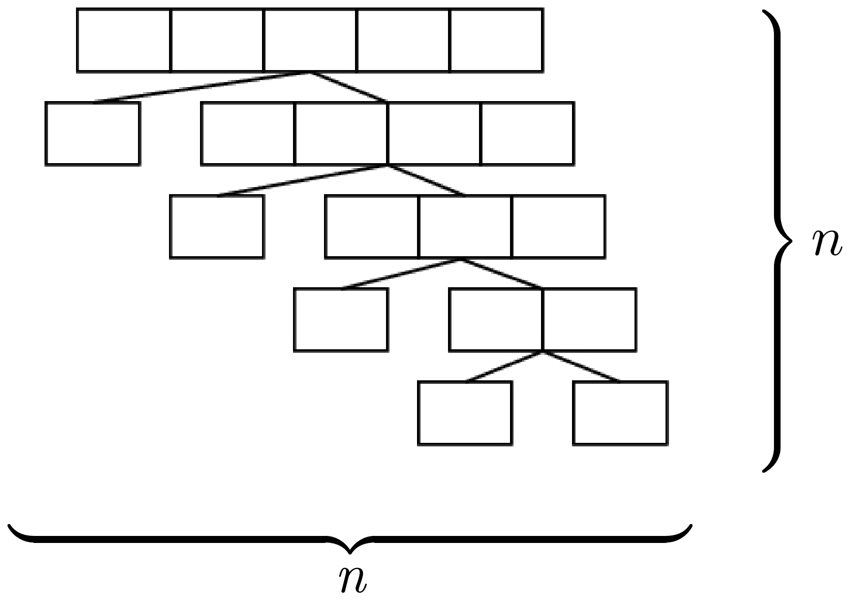
To find out what the complexity of quicksort is, we have to check how often quicksort performs a splitting operation, and on lists of how many elements it does so.

A run of quicksort can be visualised as a tree. The height of the tree corresponds to the recursion depth; and the width of the tree corresponds to the work done by the splitting sub-routine at each recursion level . . .

This will crucially depend on what elements we select for splitting. Splitting could be either more or less *balanced* or more or less *unbalanced* . . .

Extremely Unbalanced Splitting

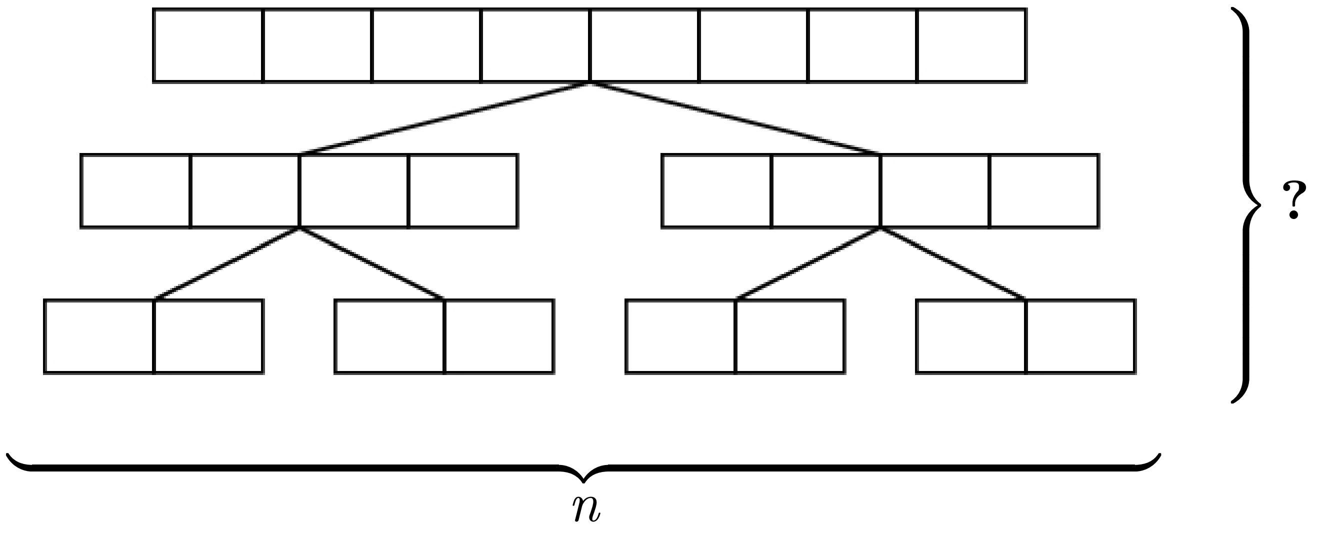
In the case of extremely unbalanced splitting (say, we always select the smallest element and all other elements end up in the righthand sublist), quicksort has got a complexity of $O(n^2)$.



This situation occurs, for instance, if the input list is already sorted and we always select the head of the list for splitting (as in our Prolog implementation).

Balanced Splitting

For the case of balanced splitting (the number of elements ending up to the left of the selected element is always roughly equal to the number of elements ending up on the righthand side), the following figure depicts the situation:



To find out about the complexity of quicksort in the case of (more or less) balanced splitting, we need to know what the height of such a tree is (with respect to n).

Height of a Binary Tree

- How high is a binary tree of width n?
- There is 1 root node. Each time we go down one level, the number of nodes per level doubles. On the final level, there are n nodes (= width of the tree).
- So, how many times do we have to multiply 1 by 2 to get n?

$$\begin{aligned} 1 \cdot \underbrace{2 \cdot 2 \cdot \cdots \cdot 2}_{x} &= n \\ 2^x &= n \\ x &= \underline{\underline{\log_2 n}} \end{aligned}$$

- <u>Remark:</u> Logarithms with different bases just differ by a constant factor (e.g. $\log_2 n = 5 \cdot \log_{32} n$). So, in particular, when we use the Big-O Notation, the basis of logarithms does not matter and we are simply going to write "$\log n$".

Complexity of Quicksort (cont.)

For balanced splitting, we end up with an overall complexity of $O(n \log n)$ for quicksort.

In practice, we can usually assume that splitting will occur in more or less balanced a fashion. This is why quicksort is usually regarded as an $O(n \log n)$ algorithm, although we have seen that complexity will be quadratic in the very worst case.

The assumption of balancedness is justified, for instance, when the input list is randomly ordered. In general, of course, we cannot make that assumption. In general, always selecting the head of the input list for splitting may not be a good strategy. In some cases it may be possible to devise a heuristic to decide which element to select (not discussed here).

Summary: Sorting Algorithms

- Sorting a list is a fundamental algorithmic problem that comes up again and again in Computer Science and AI.
- We have discussed three searching algorithms: naïve bubblesort, improved bubblesort, and quicksort.
- The *Prolog implementation* of each of these take an ordering relation and a list as input, and return the sorted list.
- The complexity of (improved) *bubblesort* is $O(n^2)$. Slow.
- The complexity of *quicksort* is $O(n \log n)$, at least under the assumption of reasonably balanced splitting. Fast.
- There are many other sorting algorithms around. Two of them, *insert-sort* and *merge-sort* are also explained in the textbook.