Problem Solving and Search

Ulle Endriss
Institute for Logic, Language and Computation
University of Amsterdam

[http://www.illc.uva.nl/~ulle/teaching/pss/]

Table of Contents

Lecture 7: Sorting Algorithms and Complexity Analysis 3

Lecture 7: Sorting Algorithms and Complexity Analysis

Plan for Today

Having to *sort a list* is an issue that will come up again and again when you work on somewhat more complex programs.

Sorting thus is a standard topic in introductory programming courses. Also because it demonstrates particularly well that there can be several very different solutions (algorithms) to the same problem, and that it can be useful to systematically compare these alternative approaches.

In this lecture, we are going to:

- introduce different *sorting algorithms* (*bubblesort* and *quicksort*)
- discuss their *implementation* in Prolog
- analyse their *computational complexity* using the *Big-O notation*

Objective

We want to implement a predicate that takes an *ordering relation* and an *unsorted list* and that returns 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 will use this predicate to check whether the terms `A` and `B` are ordered with respect to the ordering relation `Rel`:

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

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

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

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

Bubblesort

Our first sorting algorithm is called *bubblesort*. The way it operates is reminiscent of the bubbles floating upwards in a glass of champagne.

This algorithm works as follows:

- Go through the list from left to right until you hit two 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

The following 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 find 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).
```

Remark: This implementation makes the implicit assumption that `Rel` is *asymmetric*, like > but unlike >= (otherwise there could be a loop).

Examples

Just to show 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 of Algorithms

It is important to understand the *computational complexity* of your algorithm. We may be interested in these *resources:*

- *Time complexity:* How long will it take to compute a solution?
- *Space complexity:* How much memory do we need to do so?

We cannot really talk about *all* cases our algorithm may be applied to. What we can try instead:

- *Worst-case analysis:* How much time/memory will the algorithm take in the worst case (for the most difficult problem of its kind)?
- *Average-case analysis:* How much time/memory will the algorithm take on average (or: for a "typical" problem instance)?

It is usually very difficult to carry out an average-case analysis that is theoretically sound (hard to define "average" problem instances). Experimental studies using real-world data are often the only way.

Complexity Analysis of Sorting Algorithms

We will look into the *worst-case time complexity* of sorting algorithms.

Let n be the *length* of the list to be sorted (i.e., the *problem size*).

We will measure the *complexity of an algorithm* in terms of the *number of primitive comparison operations* (i.e., the number of 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.

Want to understand what happens to the time required to solve a problem with a given algorithm as we increase n. Example:

- for $n = 2$, improved bubblesort makes at most 2 comparisons
- for $n = 3$, improved bubblesort makes at most 6 comparisons
- for $n = 4$, improved bubblesort makes at most 12 comparisons
- ...
- for lists of size n, improved bubblesort makes at most $f(n)$ comparisons

But what is this function f? Is it linear? Quadratic? Exponential?

Big-O Notation

When analysing the complexity of algorithms, small constants and the like don't matter very much. What we are really interested in is the *order of magnitude* with which the complexity of the algorithm increases as we increase the size of the input.

Let n be the *problem size* and let $f(n)$ be the *precise complexity*.

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

Suppose g is a "nice" function that is a "*good approximation*" of f. The Big-O Notation is a way of making this mathematically precise.

> We say that $f(n)$ *is in* $O(g(n))$ *if and only if* there exist an $n_0 \in \mathbb{N}$ and a $c \in \mathbb{R}^+$ such that $f(n) \leqslant c \cdot g(n)$ for all $n \geqslant n_0$.

Thus, 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".

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 the 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.

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

Remark: The complexity of our original version of bubblesort is $O(n^3)$, but we will not prove this here.

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, in part, depend on the choice of `X`.
In Prolog, we simply 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 that preceed `Head` w.r.t. `Rel` (list `Left`) and the rest (list `Right`). Then `Left` and `Right` get 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).
```

Still to do: implement `split/5`

Splitting Lists

The predicate `split/5` 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 w.r.t. the 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 the Splitting Predicate

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

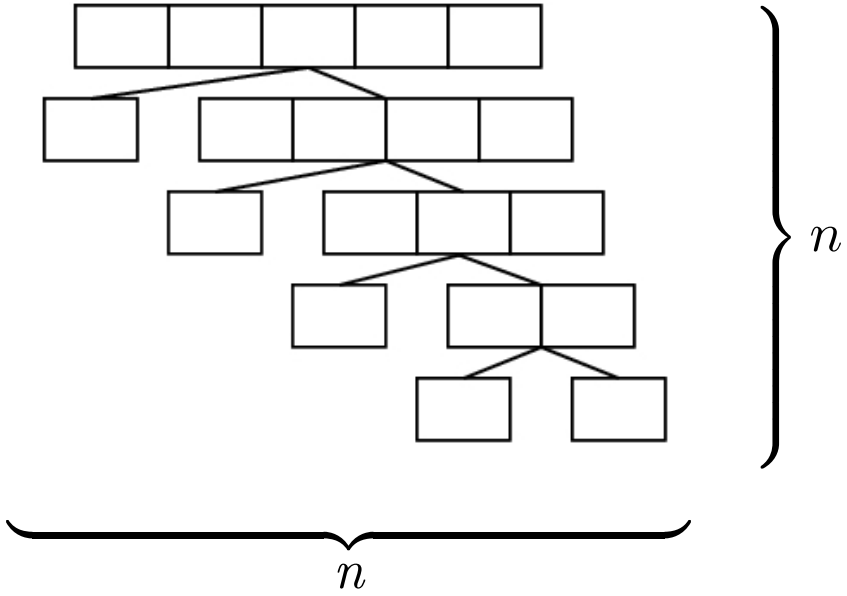
To analyse the complexity of quicksort, we have to check how often quicksort performs splitting, and on lists of what size.

A run of quicksort can be visualised as a *tree*. The height of the tree corresponds to the recursion depth. 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 relatively *balanced* or relatively *unbalanced* ...

Extremely Unbalanced Splitting

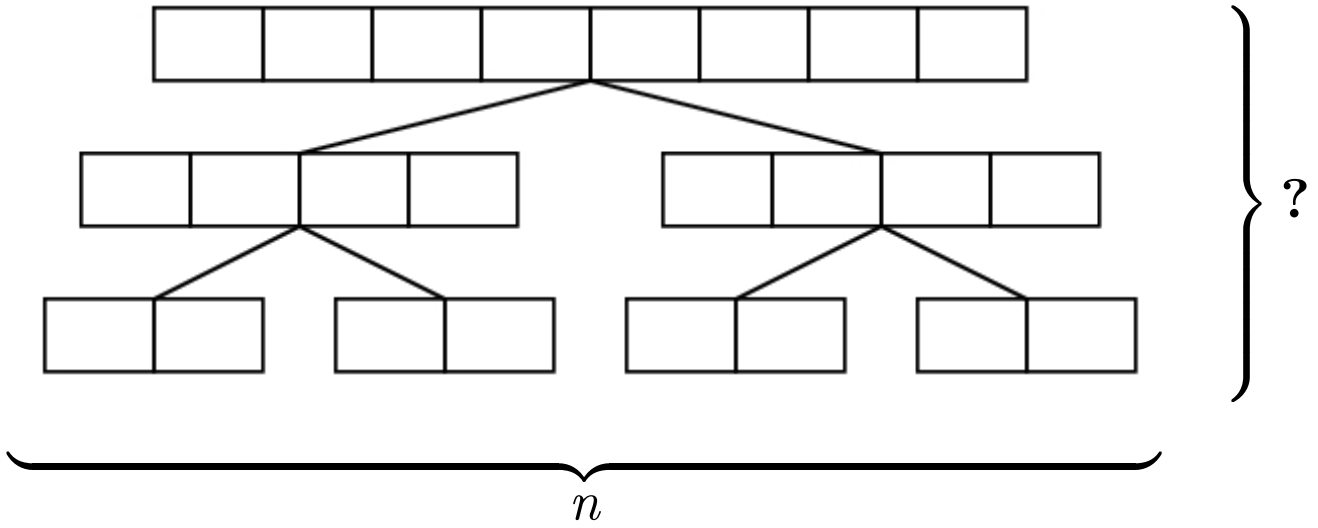
In the case of extremely unbalanced splitting (say, we always select the smallest element and all other elements go into the righthand sublist), quicksort has complexity $O(n^2)$:



Example: This happens when the input list is already sorted and we always select the head of the list for splitting (as we do in Prolog).

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 time complexity of quicksort in the case of (more or less) balanced splitting, we thus 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?

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

- Remark: 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 (continued)

For balanced splitting, we thus end up with an overall complexity of $O(n \log n)$ for quicksort. *This is much better than $O(n^2)$!*

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 can be quadratic for cases where this assumption is not justified.

The assumption of balancedness is justified, for instance, when the input list is randomly ordered. In general, of course, we cannot always make this assumption. In general, always selecting the head of the input list for splitting may not be the best strategy.

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 sorting algorithms:
  naïve bubblesort, *improved bubblesort*, and *quicksort*.
- The Prolog implementations 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 explained in the textbook.