5 Sorting Methods

Sorting represents one of the most common problems in programming languages. As we will see further in this chapter, various sorting algorithms have a very elegant specification in Prolog.

5.1 Direct sorting methods

Direct sorting methods are the simplest, from the algorithmic point of view, sorting methods. They don't employ any specialized programming technique, but use simple techniques starting from the specifications.

5.1.1 Permutation sort

A possible approach to the sorting problem is to find the ordered permutation of a list. This natural strategy can be easily specified in Prolog:

```
perm_sort(L, R):-perm(L,R), is_ordered(R), !.
```

The predicate generates a permutation of the list L, then checks to see if it is ordered. If R is not ordered, the sub-query to is_ordered(R) will fail. The execution will backtrack with a new resolution for perm(L, R), resulting in a new permutation. This process continues until the ordered permutation is found. Of course, this approach, as natural as it is, is very inefficient from the algorithmic point of view. We have included it here due to its simplicity.

For generating the permutations of a list, we employ the following observation:

$$n! = (n-1)! * n$$

Thus, in order to obtain all the permutations of a list with n elements one has to extract randomly one element from the list (randomly = ensure that each element will be extracted at some point), obtain the permutations of the remaining n-1 elements, and then add the extracted element to the result.

The predicate which generates the permutations of a list:

```
perm(L, [H|R]):-append(A, [H|T], L), append(A, T, L1), perm(L1, R).
perm([], []).
```

The two calls to append in clause one of the predicate are complementary: the first call extracts an element H from the list L randomly, while the second recreates the list without H.

Exercise 5.1: Study the execution of the following queries (repeating the question):

1. ?- append(A, [H|T], [1, 2, 3]), append(A, T, R).
2. ?- perm([1, 2, 3], L).

Exercise 5.2: Reverse the order of the clauses of predicate append and redo the queries in *exercise 5.1*.

The predicate is_ordered is straightforward:

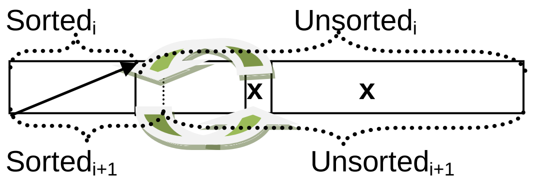
```
is_ordered([_]).
is_ordered([H1, H2|T]):-H1 =< H2, is_ordered([H2|T]).
```

Exercise 5.3: Study the execution of the following queries (repeating the question):

1. ?- is_ordered([1, 2, 4, 4, 5]).
2. ?- is_ordered([1, 2, 4, 2, 5]).
3. ?- perm_sort([1, 4, 2, 3, 5], R).

5.1.2 Selection sort

Selection sort works by selecting, at each step, the minimum (or maximum) element from the unsorted part of the list, and add it to the already sorted part, in the appropriate position:



, x = Min(Unsorted$_i$)

The predicate which performs selection sort:

```
sel_sort(L, [M|R]):- min(L, M), delete(M, L, L1), sel_sort(L1, R).
sel_sort([], []).
```

The auxiliary predicates employed here have already been discussed earlier in this book. For a quick reference, go to chapter 3, sub-sections 1.5 and 1.6. This version of the selection sort predicate builds the solution as recursion returns. Therefore, the result variable holds the sorted part of the list, and the input list holds the unsorted part, which changes with each recursive call. The sorted part is initialized to [] when the recursive calls stop, and it grows as each recursive call returns, by adding the current minimum in front of the list.

Exercise 5.4: Modify the first clause of the predicate sel_sort such that it outputs intermediate results, and study the execution of the following queries:

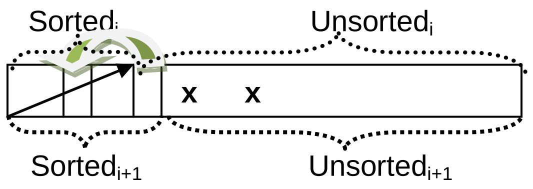
1. ?- sel_sort([3, 2, 4, 1], R).
2. ?- sel_sort([3, 1, 5, 2, 4, 3], R).

Hint: to output the value of a variable onscreen use the function write(). Consult the environment user manual for more information.

Exercise 5.5: Write a predicate which finds the minimum element from a list and deletes it (combines the functionality of min and delete).

5.1.3 Insertion sort

Insertion sort, as its name implies, inserts each element from the unsorted part in the appropriate position in the sorted part. All the elements (in the sorted part) which are greater than the current element considered (x) are shifted to the right in order to make room for x:



, x = First(Unsorted_i)

The predicate which performs insertion sort can be written as:

```
ins_sort([H|T], R):- ins_sort(T, R1), insert_ord(H, R1, R).
ins_sort([], []).

insert_ord(X, [H|T], [H|R]):-X>H, !, insert_ord(X, T, R).
insert_ord(X, T, [X|T]).
```

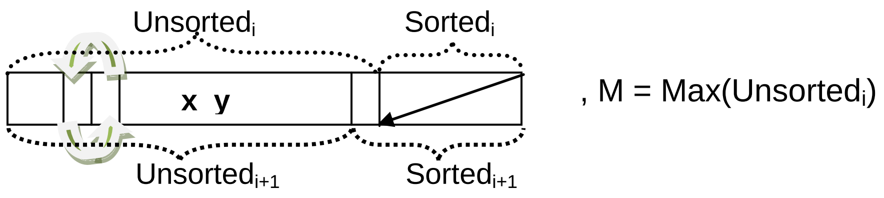
This is a backward recursive approach: first we obtain the result of the recursive call (R_1), then compute the result on the current level by inserting the current element in its right position in R_1 (predicate insert_ord).

Exercise 5.6: Modify the first clause of the predicate ins_sort such that it outputs intermediate results, and study the execution of the following queries:

1. ?- insert_ord(3, [], R).
2. ?- insert_ord(3, [1, 2, 4, 5], R).
3. ?- insert_ord(3, [1, 3, 3, 4], R).
4. ?- ins_sort([3, 2, 4, 1], R).
5. ?- ins_sort([3, 1, 5, 2, 4, 3], R).

5.1.4 Bubble sort

Bubble sort is one of the simplest direct sorting methods, but also the least efficient. It performs several passes through the data. In each pass it compares adjacent elements two by two and, if necessary, it swaps them. This approach ensures that, in each pass, at least the maximum element of the unsorted part reaches its final position – at the end of the unsorted part, or, better yet, the beginning of the sorted part. Therefore, in each pass at least the tail of the sequence is already sorted.



A possible Prolog specification for bubble sort:

```
bubble_sort(L, R):-one_pass(L, R1, F), nonvar(F), !, bubble_sort(R1, R).
bubble_sort(L, L).

one_pass([H1, H2|T], [H2|R], F):- H1>H2, !, F = 1, one_pass([H1|T], R, F).
one_pass([H1|T], [H1|R], F):-one_pass(T, R, F).
one_pass([], [] ,_).
```

We have selected the version which performs only as many passes through the data as necessary. When the list becomes sorted, the algorithm stops. This is controlled through a flag, F. Before each new pass, the flag is reset to a free variable. Whenever a swap is performed, F is instantiated to a constant value (1). After the pass we check the flag: if it has been instantiated (no longer a free variable), then at least a swap has been performed, meaning that the list may not be ordered. Thus, a new call to bubble_sort is required. If F remains free after the call to one_pass, then the call to nonvar(F) will fail. This means the list L is sorted, so we need to pass it to the result (second clause of the predicate bubble_sort).

Exercise 5.7: Study the execution of the following queries (outputting the intermediate results of bubble_sort if necessary):

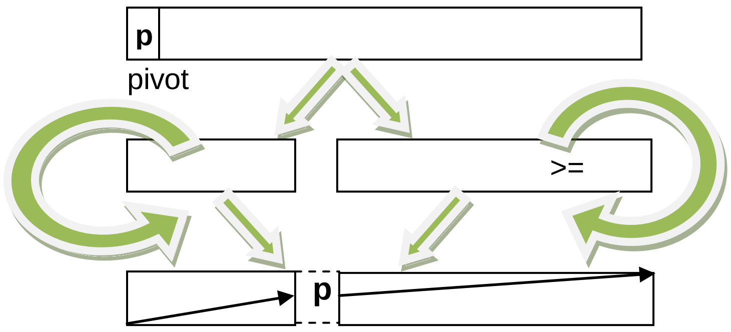
1. ?- one_pass([1, 2, 3, 4], R, F).
2. ?- one_pass([2, 3, 1, 4], R, F).
3. ?- bubble_sort([1, 2, 3, 4], R).
4. ?- bubble_sort([2, 3, 1, 4], R).
5. ?- bubble_sort([2, 3, 3, 1], R).

5.2 Advanced Sorting Methods

We will present two advanced sorting methods, based on the "*divide et impera*" technique: quicksort and merge sort. For the first technique the *impera* part of the algorithm is performed in *O(1)*, while the second performs the *divide* operation in constant time.

5.2.1 Quicksort

Quicksort's divide et impera strategy consists in partitioning the sequence in two, according to a pivot element: a sub-sequence containing the elements smaller than the pivot, and the sub-sequence containing the elements which are larger than or equal to the pivot. Then, apply the same strategy for each of the two sub-sequences. The process stops when the sequence to partition is empty. To compose the result, simply append the sorted sub-sequence of the elements which are smaller than the pivot with the pivot and the sub-sequence of larger elements:



The Prolog predicates which perform quicksort are presented below:

```
quick_sort([H|T], R):-partition(H, T, Sm, Lg), quick_sort(Sm, SmS),
                      quick_sort(Lg, LgS), append(SmS, [H|LgS], R).

quick_sort([], []).
```

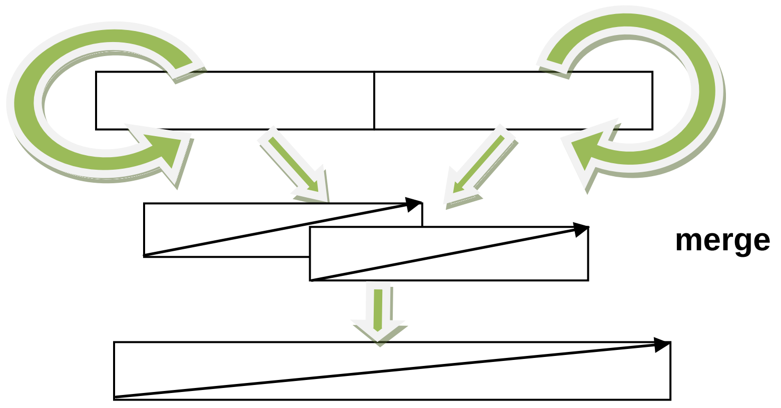
```
partition(H, [X|T], [X|Sm], Lg):-X<H, !, partition(H, T, Sm, Lg).
partition(H, [X|T], Sm, [X|Lg]):-partition(H, T, Sm, Lg).
partition(_, [], [], []).
```

Exercise 5.8: Study the execution of the following queries (if necessary alter the predicates to allow you to visualize the intermediate results):

1. partition(3, [4, 2, 6, 1, 3], Sm, Lg).
2. quick_sort([3, 2, 5, 1, 4, 3], R).
3. quick_sort([1, 2, 3, 4], R).

5.2.2 Merge sort

Merge sort splits the sequence in two equal parts, applies the procedure recursively on each part to obtain the two sorted sub-sequences, which are then merged at the end:



In Prolog, we can specify the merge sort procedure in the following manner:

```
merge_sort(L, R):-split(L, L1, L2), merge_sort(L1, R1), merge_sort(L2, R2),
                              merge(R1, R2, R).
merge_sort([H], [H]).
merge_sort([], []).

split(L, L1, L2):-length(L, Len), Len>1, K is Len/2, splitK(L, K, L1, L2).

splitK([H|T], K, [H|L1], L2):- K>0, !, K1 is K-1, splitK(T, K1, L1, L2).
splitK(T, _, [], T).
```

```
merge([H1|T1], [H2|T2], [H1|R]):-H1<H2, !, merge(T1, [H2|T2], R).
merge([H1|T1], [H2|T2], [H2|R]):-merge([H1|T1], T2, R).
merge([], L, L).
merge(L, [], L).
```

The predicate splitK takes the first K elements in the input list L and adds them to L_1. The rest of L will be put in L_2. Therefore, the predicate split divides the list in two equal parts by calling splitK with argument K equal to the length of the list over 2 (K = length(L)/2). If the length of L is 0 or 1, then the query to split will fail, causing the resolution through clause 1 of merge_sort to fail – this is when the recursion should stop. Therefore, those calls will be matched through clauses 2 or 3 of the predicate merge_sort. The predicate merge performs the merging of two ordered lists. Its specification is straightforward.

Exercise 5.9: Study the execution of the following queries (if necessary alter the predicates to allow you to visualize the intermediate results):

1. ?- split([2, 5, 1, 6, 8, 3], L1, L2).
2. ?- split([2], L1, L2).
3. ?- merge([1, 5, 7], [3, 6, 9], R).
4. ?- merge([1, 1, 2], [1], R).
5. ?- merge([], [3], R).
6. ?- merge_sort([4, 2, 6, 1, 5], R).

Quiz exercises

q5-1. For the predicate perm, the two calls to append, extract en element from a list randomly, and recreate the list without the selected element. Write the predicate(s) which perform these operations without using append, then write a new predicate, perm1, which generates the permutations of a list, using the new predicate(s) for extracting/deleting an element from a list.

q5-2. Write a predicate which performs selection sort by selecting, in each step, the maximum element from the unsorted part, and not the minimum. Analyze its efficiency.

q5-3. Write a forward recursive predicate which performs insertion sort. Analyze its efficiency in comparison with the backward recursive version.

q5-4. Implement a predicate which performs bubble sort, using a fixed number of passes through the input sequence.

5.4 Problems

p5-1. Suppose we have a list of ASCII characters. Sort the list according to their ASCII codes.

```
?- sort_chars([e, t, a, v, f], L).
L = [a, e, f, t, v] ? ;
no
```

Hint: search for char_code in the environment manual.

p5-2. Suppose we have a list whose elements are lists containing atomic elements. Write a predicate(s) which sorts such a list according to the length of the sub-lists.

```
?- sort_len([[a, b, c], [f], [2, 3, 1, 2], [], [4, 4]], R).
R = [[], [f], [4, 4], [a, b, c], [2, 3, 1, 2]] ? ;
no
```

Hint: the ordering relation should be changed from the normal ordering relation on numbers to the one required here, i.e. on list lengths.