Sorting Methods

1 Theoretical Considerations

In this session, multiple sorting methods implemented in Prolog are presented.

1.1 Permutation Sort

This method generates, on backtracking, all the permutations of the input list, until the ordered permutation is reached; then it stops.

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

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

is_ordered([H1, H2|T]):- H1 =< H2, is_ordered([H2|T]).
is_ordered([_]). % if only one element remains, the list is already ordered
```

The predicate generates a permutation of the list L by calling the *perm/2* predicate, then checks to see if it is ordered (*is_ordered/1*). 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 and elegant Prolog specification.

The number of permutations of a list with n elements is equal to *n!*. To generate all permutations, we will extract an element *H* randomly from the input list and we will add it to the beginning of the resulting list. The extraction of the random element *H* is achieved through the use of the nondeterminism of the *append/3* predicate. The input list can be written as the concatenation of 2 sub-lists. The *H* element is selected as the first element of the second sub-list (try to think why it needs to be selected like that).

If we were to take the list [1,2,3] and separate it into 2 sub-lists using *append*:

Query
```
?- append(L1, L2, [1,2,3]).
L1 = []
L2 = [1,2,3] ;
L1 = [1]
L2 = [2,3] ;
% ... and so on
```

Using the *[H|T]* template we will decompose the second list:

Query
```
?- append(L1, [H|T], [1,2,3]).
L1 = []
H = 1          T = [2,3] ;
L1 = [1]
H = 2          T = [3] ;
% ...
```

The permutations are obtained by appending *L1* and *T* and then adding *H* at the beginning of the resulting list:

For example:

L1 = []   H = 1   T = [2,3]

The append of L1 and T would result in [2,3] and by adding the H at the beginning, we get [1,2,3], a first permutation.

L1 = [1]   H = 2   T = [3] ;

The append of L1 and T would result in [1,3] and by adding the H at the beginning, we get [2,1,3].

Note. The *perm/2* predicate uses backtracking to generate all permutations. This example is used to show the logic of how a permutation is formed through concatenations, it does not show the logic of how the *perm* predicate generates permutations in a certain order through backtracking. We recommend tracing the execution.

The last step is to write the predicate that verifies if a list is ordered increasingly:

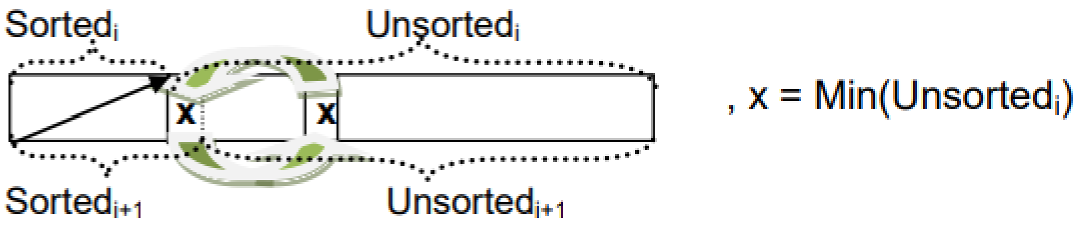
Note. The is_ordered/1 predicate has only one argument, the input list that we want to verify if it is ordered. The possible results are yes or no. Any Prolog predicate returns true or false, therefore for this predicate, we do not need an output argument, as a true or false response is enough.

Follow the execution of:

Query
```
?- trace, append(A, [H|T], [1, 2, 3]), append(A, T, R).
?- trace, perm([1, 2, 3], L).
?- trace, is_ordered([1, 2, 4, 4, 5]).
?- trace, perm_sort([1, 4, 2, 3, 5], R).
```

1.2 Selection Sort

This sorting algorithm *selects* at each step the smallest (or largest) element of the unsorted subpart and moves it at the end of the sorted subpart. The smallest element of the input list will be the first element of the resulting list.



Code

```
sel_sort(L, [M|R]):- min1(L, M), delete1(M, L, L1), sel_sort(L1, R).
sel_sort([], []).

delete1(X, [X|T], T) :- !.
delete1(X, [H|T], [H|R]) :- delete1(X, T, R).
delete1(_, [], []).

min1([H|T], M) :- min1(T, M), M<H, !.
min1([H|_], H).
```

The *min1/2* and *delete1/3* predicates can be found in previous sessions.

This version of the selection sort predicate builds the solution as the 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 (clause 2), and it grows as each recursive call returns, by adding the current minimum in front of the list.

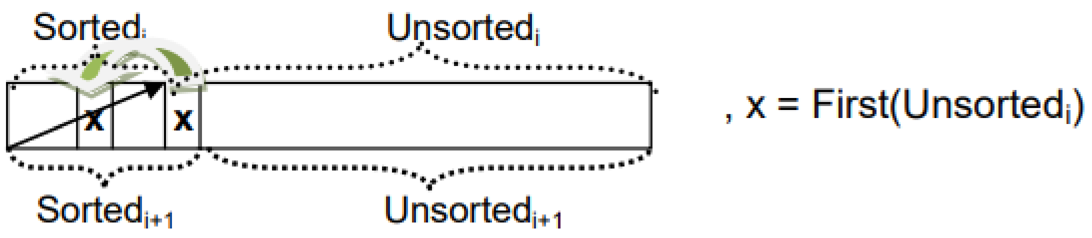
Follow the execution of:

Query

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

1.3 Insertion Sort

This sorting method selects at each step an arbitrary element of the unsorted subpart (usually the first element) and inserts it at the correct position in the sorted subpart (using linear or binary search). All elements greater than the current element (X) are shifted to make room.



Code

```
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: the result of the recursive calls (R1) is obtained first, then we compute the result on the current level (*insert_ord/3 predicate*) by inserting the current element into the correct position in R1.

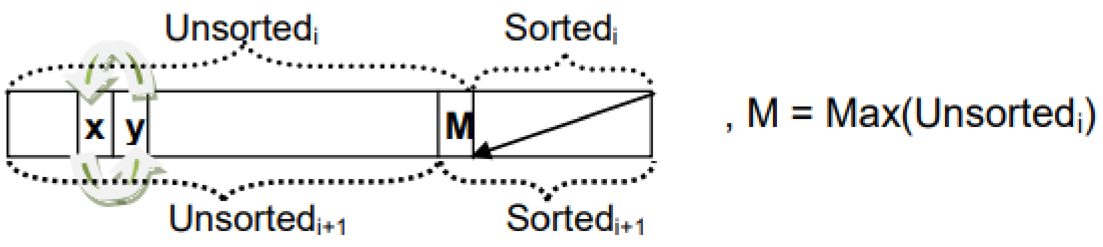
Follow the execution of:

Query

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

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 input list. At each pass, it compares adjacent elements two by two and swaps them if the ordering condition is not satisfied. Through this process, each pass ensures that the maximum element of the unsorted part reaches the beginning of the sorted part. Thus, at each pass, the tail of the list is sorted.



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

If at one pass no switch is made, it means that the list is already ordered. An improved version of bubble sort makes use of this and stops the algorithm in such cases. To verify that any swaps were made, we use a *flag (F)*. Whenever a swap is performed, *F* is instantiated to a constant value (1). After each pass, we check the flag:

if *F* 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/2* is required.

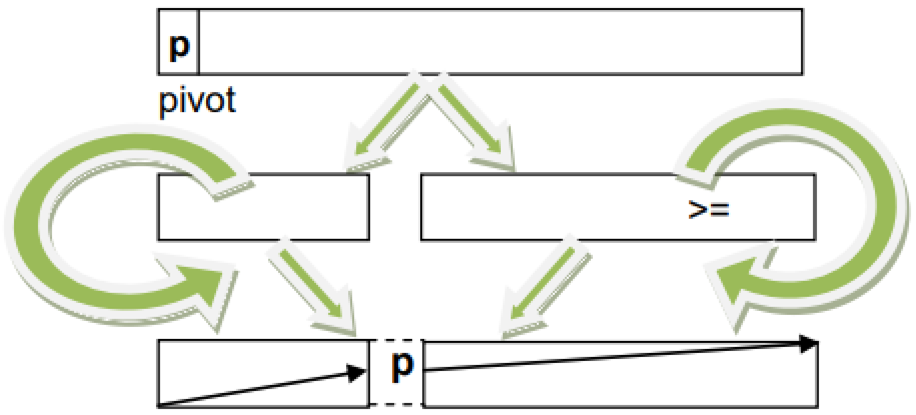
If *F* remains free after the call to *one_pass/3*, then the call to *nonvar(F)* will fail. This means the list *L* is sorted, so we need to pass it to the result (second clause of the predicate *bubble_sort/2*).

Follow the execution of:

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

1.5 Quick Sort

Quick sort uses a pivot to split the input list into 2 sub-lists (*divide et impera* technique). A sub-list with elements smaller than the pivot and a sub-list with elements larger than the pivot. It repeats this process until the sub-lists become empty. For the implementation in this laboratory, the pivot is the first element of the list. The result is composed by appending the sorted *sub-list of smaller elements* with the *pivot* and then with the *sub-list of larger elements* (the append of the first clause of the *quick_sort/2* predicate).



Code

```
quick_sort([H|T], R):- % pivot is chosen as first element
    partition(H, T, Sm, Lg),
    % sort sublist with elements smaller than pivot
    quick_sort(Sm, SmS),
    % sort sublist with elements larger than pivot
    quick_sort(Lg, LgS),
    append(SmS, [H|LgS], R).
quick_sort([], []).

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

The *partition/4* predicate has the pivot as the first argument and the input list as the second argument, when the current element (*X*) of the list is smaller than the pivot (*P*) then it is added (clause1) to the list of smaller elements (third argument), otherwise it is added (clause 2) to the list of larger elements (fourth argument). This recursive process stops when the input list (second argument) becomes empty.

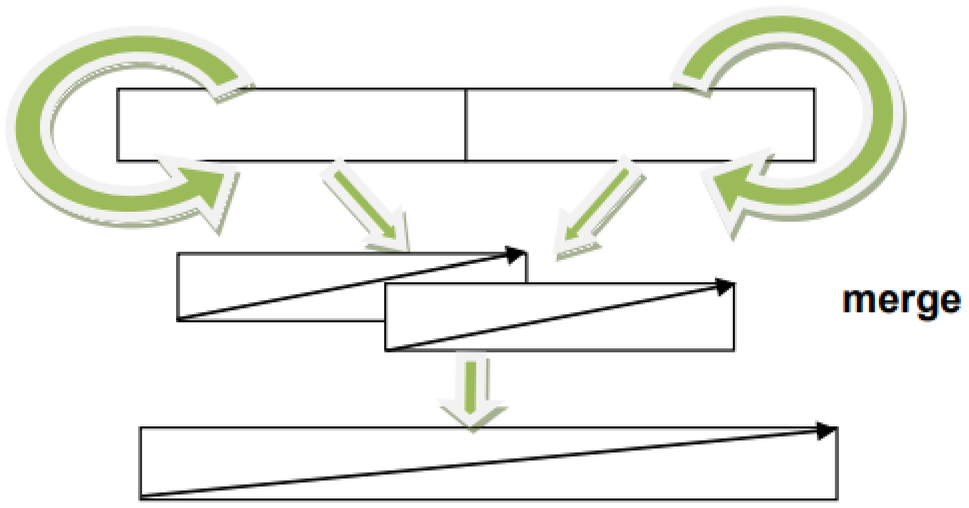
Follow the execution of:

Query

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

1.6 Merge Sort

Merge sort splits the input list into two sub-lists of equal lengths (*divide et impera* technique). By applying the procedure recursively, it sorts the two sub-lists and then merges the two sorted sub-lists.



Code

```
merge_sort(L, R):-
   split(L, L1, L2), % splits L into 2 equal-length sub-lists
   merge_sort(L1, R1),
   merge_sort(L2, R2),
   merge(R1, R2, R). % merges the ordered sub-lists
% split fails if the list contains one or no elements
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 *splitK/4* predicate takes the first **K** elements of the input list **L** and inserts them into L1 (clause 1) while the rest are inserted into L2 (clause 2). The *split* predicate divides the list into two equal parts by calling *splitK/4* with K being half of the length of the list (*K is Len / 2*). If the length of the input list L is 0 or 1, then the query of *split* fails, causing the resolution through clause 1 of *merge_sort/2* to fail – at this point the recursion should stop. Therefore, those calls will be matched through clauses 2 or 3 of the *merge_sort/2* predicate. The *merge/3* predicate performs the merging of two ordered lists, by adding the head of the first or second list to the output list depending on which is smaller.

Follow the execution of:

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

2 Exercises

1. Rewrite the *sel_sort/2* predicate such that it selects the maximum value from the unsorted part, therefore it sorts the list decreasingly, name it *sel_sort_max/2*. The *max1/2* predicate can be created by modifying the *min1/2* predicate from the last laboratory.

```
Query
?- sel_sort_max([3,4,1,2,5], R).
R = [5, 4, 3, 2, 1];
false
```

2. Implement bubble sort using a fixed number of passes through the input sequence, name it *bubble_sort_fixed/3*.

```
Query
% bubble_sort_fixed(L, K, R). – K is the number of passes
?- bubble_sort_fixed([3,5,4,1,2], 2, R).
R = [3, 1, 2, 4, 5]
```

3. Write a predicate that sorts a list of ASCII characters. (You can use a sorting method of your choice).

Suggestion: use the *char_code/2* built-in predicate

```
Query
?- sort_chars([e, t, a, v, f], L).
L = [a, e, f, t, v] ;
false
```

4. Write a predicate that sorts a list of sub-lists according to the lengths of the sub-lists. (You can use a sorting method of your choice).

```
Query
?- sort_lens([[a, b, c], [f], [2, 3, 1, 2], [], [4, 4]], R).
R = [[], [f], [4, 4], [a, b, c], [2, 3, 1, 2]] ;
false
```

Optional. This predicate can be made more complex when considering the case of two sub-lists with equal lengths, take the case of [1,1,1] and [1,1,2], the predicate would have to additionally parse the two lists in the case of equal lengths and compare them element-by-element.

```
Query
?- sort_lens2([[], [1], [2, 3, 1, 2], [2, 3, 5, 2], [7,6,8], [4, 4]], R).
R = [[], [1], [4, 4], [7, 6, 8], [2, 3, 1, 2], [2, 3, 5, 2]];
```

```
false
```

5. Write a *forward* recursive implementation of the *ins_sort/2* predicate, name it *ins_sort_fwd/2*.

Recommendation: the insertion sort predicate is made of two predicates, both using a backwards approach, try to modify them both into a forwards approach.

Query

```
?- ins_sort_fwd([3,4,1,2,5], R).
R = [1, 2, 3, 4, 5];
false
```

6. Rewrite the *perm/2* predicate without using the *append/3* predicate, name it *perm1/2*. Extracting and deleting an element must be achieved differently.

Query

```
?- perm1([1,2,3], R).
R = [1, 2, 3];
R = [1, 3, 2];
R = [2, 1, 3];
…
```