Sortări

1. Obiective

În această lucrare, se prezintă multiple metode de sortare implementate în Prolog.

2. Considerații teoretice

1.1 Sortarea prin permutări

Aceasta metodă generează toate permutările posibile ale listei de intrarea până când se ajunge la o permutare care are toate elementele ordonate și atunci se oprește.

```
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([_]). % dacă este doar un element, lista este deja ordonată
```

Predicatul generează o permutare a listei L prin apelul predicatului *perm/2*, după care verifică dacă este ordonată (*is_ordered/1*). Dacă R nu este o listă ordonată, *is_ordered(R)* va da fail. Execuția va face backtracking la *perm(L,R)* și va rezulta într-o permutare nouă. Acest proces continuă până când permutarea ordonată este găsită. În mod evident, această abordare este foarte ineficientă din punct de vedere algoritmic și a fost inclusă în acest laborator datorită simplității.

Numărul de permutări ale unei liste cu *n* elemente este egal cu *n!*. Pentru a genera permutările, vom alege un element H aleatoriu din lista de intrare și îl vom pune pe prima poziție în lista rezultat. Alegerea lui H se va realiza folosind nedeterminismul predicatului *append/3.* Lista de intrare poate fi scrisă ca o concatenare de 2 sub-liste. Elementul H îl alegem să fie primul element din a doua sub-listă.

Dacă am lua exemplul listei [1,2,3] și folosind predicatul *append* o separăm în două sub-liste:

```
?- append(L1, L2, [1,2,3]).
```

L1 = []
L2 = [1,2,3] ;
L1 = [1]
L2 = [2,3] ;
% ... și tot așa

Folosind șablonul *[H|T]* putem separa a doua listă:
?- append(L1, [H|T], [1,2,3]).
L1 = []
H = 1 T = [2,3] ;
L1 = [1]
H = 2 T = [3] ;
% ...

Permutările obținute prin append-ul dintre L1 și T, iar mai târziu concatenarea primul element H la începutul listei rezultat:

De exemplu:
L1 = [] H = 1 T = [2,3]
Prin append-ul listei L1 și T se obține [2,3] iar prin concatenarea lui H la început, se obține [1,2,3], o primă permutare.
L1 = [1] H = 2 T = [3] ;
Prin append-ul listei L1 și T se obține [1,3] iar prin concatenarea lui H la început, se obține [2,1,3].

Observație. Predicatul *perm/2* folosește backtracking pentru a genera toate permutările. Acest exemplu este folosit pentru a exemplifica logica de formare a unei permutări prin concatenări, nu arată logica a cum generează perm permutările într-o anumită ordine prin backtracking. Recomandăm urmărirea execuției acestui predicat.

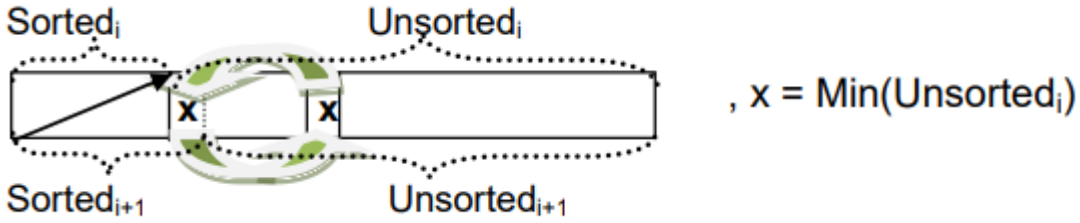
Mai avem de scris predicatul care verifică dacă lista rezultat este ordonată. Alegem ca lista rezultat să fie ordonată crescător.

Observație. Predicatul *is_ordered/1* are un singur argument, lista de intrare care vrem să o verificăm dacă este ordonată. Rezultatele posibile sunt *da* sau *nu*. Orice predicat prolog va returna *true* sau *false*, prin urmare pentru acest predicat, nu este necesar să avem un argument de ieșire deoarece un răspuns de *true* sau *false* este suficient.

Urmăriți execuția la:
?- append(A, [H|T], [1, 2, 3]), append(A, T, R).
?- perm([1, 2, 3], L).
?- is_ordered([1, 2, 4, 4, 5]).
?- perm_sort([1, 4, 2, 3, 5], R).

1.2 Sortarea prin selecție

Această sortare alege la fiecare pas cel mai mic element din lista nesortată și îl mută la capătul liste deja sortate, după care continuă pe listă fără acel element. Cel mai mic element din lista de intrarea reprezintă primul element din lista rezultat.




```
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).
```

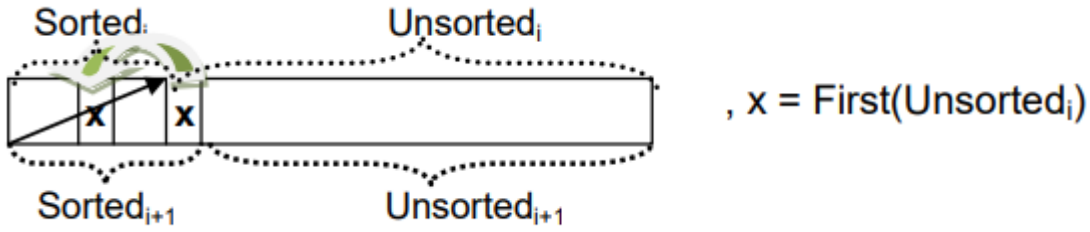
Predicatele *min1/2* și *delete1/3* se pot găsi în lucrările precedente.

Această versiunea a sortării prin selecție construiește soluția la întoarcerea din apelul recursiv. Prin urmare, variabila care conține rezultatul reține partea sortată a listei și lista de intrare conține partea nesortată, care se schimbă cu fiecare apel recursiv. Partea sortată este inițializată cu [] când recursivitatea se oprește (clauza 2), și crește la fiecare apel care se întoarce din recursivitate prin adăugarea minimului curent la începutul listei.

Urmăriți execuția la:
?- sel_sort([3, 2, 4, 1], R).
?- sel_sort([3, 1, 5, 2, 4, 3], R).

1.3 Sortarea prin inserție

Această metodă de sortare extrage la fiecare pas un element din lista nesortată (de obicei primul element) și îl inserează în poziția corectă în lista sortată (folosind căutare liniară sau binară). Implementarea de mai jos corespunde căutării liniare.




```
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]).
```

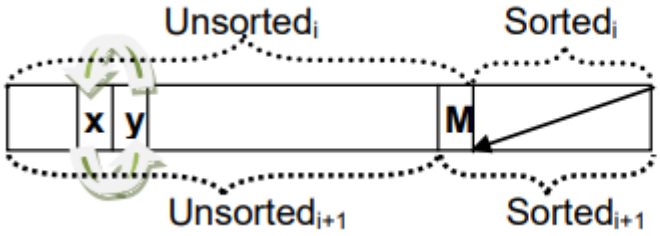
Acestă implementare conține o abordare de recursivitate înapoi: rezultatul apelurilor recursive (R1) este obținut primul, după care se obține rezultatul nivelului curent (*predicatul insert_ord/3*) prin inserarea elementului current la poziția corectă în R1.

Urmăriți execuția la:

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

1.4 Sortarea bulelor

Sortarea bulelor este una dintre cele mai simple metode directe de sortare, dar și cea mai ineficientă. Sortarea bulelor realizează mai multe treceri peste lista de intrare. La fiecare trecere verifică două câte două elemente consecutive și le interschimbă dacă nu respectă condiția de ordine. Prin acest proces, fiecare trecere garantează că elementul maxim a părții nesortate ajunge la începutul părții sortate. Astfel, la fiecare trecere, coada listei este sortată.



, M = Max(Unsorted$_i$)

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

Dacă la o trecere nu s-a făcut nici o interschimbare atunci sortarea s-a terminat. O versiunea îmbunătățită a sortării bulelor se folosește de acest fapt și oprește algoritmul în acest caz. Pentru a verifica dacă s-a făcut o interschimbare ne vom folosi de un *flag (F).* Când se produce o interschimbare, *F* este inițializat cu o valoare constantă (1). La fiecare trecere, vom verifica acest flag:

- Dacă *F* a fost inițializat (nu a rămas o variabilă liberă), atunci cel puțin o interschimbare a fost făcută, ceea ce înseamnă că lista s-ar putea să nu fie ordonată. Deci, un nou apel al *bubble_sort/2* este necesar.
- Dacă *F* a rămas o variabilă liberă după apelul *one_pass/3*, atunci apelul către *nonvar(F)* va da fail. Ceea ce înseamnă că lista L este sortată și va fi pasată către rezultat (clauza 2 a predicatului *bubble_sort/2*)

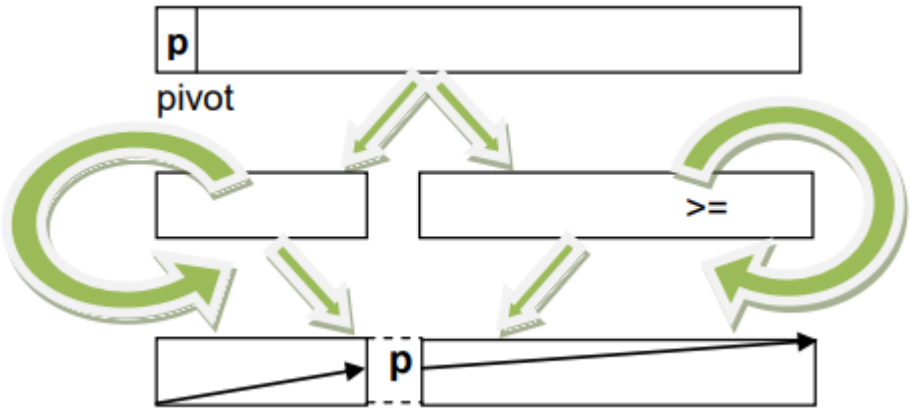
Urmăriți execuția la:

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

1.5 Sortare rapidă

Sortarea rapidă (*Quick sort*) se folosește de un pivot care împarte lista de intrare în două sub-liste (tehnică *divide et impera*). O sub-listă cu elementele mai mici decât pivotul și o a doua sub-listă cu elementele mai mari decât pivotul. În cazul implementării din laborator folosim ca și pivot primul element. Repetăm acest

proces până când sub-listele devin vide. Rezultatul este compus prin concatenarea *sub-listei cu elemente mai mici* cu *pivotul* și cu *sub-lista elementelor mai mari* (append-ul din prima clauză a predicatului *quick_sort*/2).




```
quick_sort([H|T], R):- % alegem pivot ca primul element
   partition(H, T, Sm, Lg),
   % sortăm sublista cu elemente mai mici decât pivotul
   quick_sort(Sm, SmS),
   % sortăm sublista cu elemente mai mari decât pivotul
   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(_, [], [], []).
```

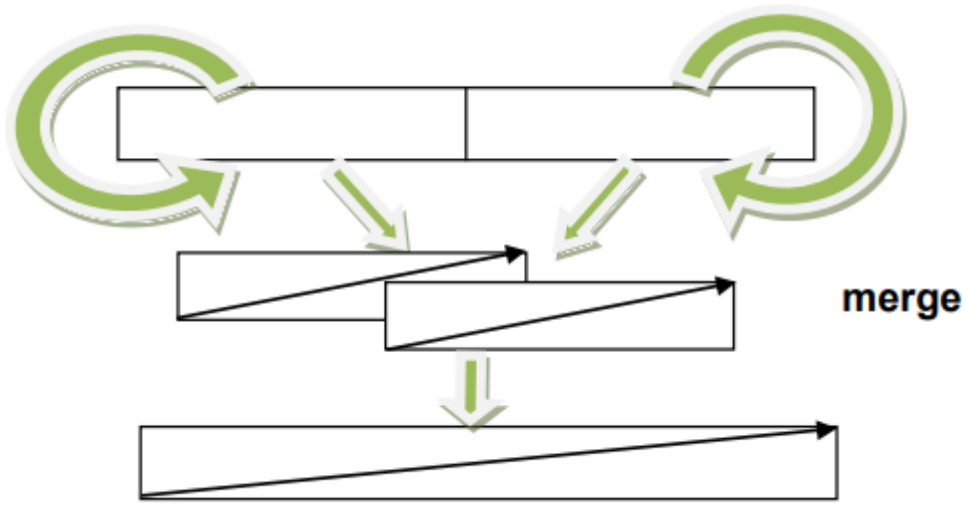
Predicatul *partition/4* are ca pivot primul argument și lista de intrare ca al doilea argument, când elementul curent (*X*) al listei este mai mic decât pivotul (*P*) este adăugat (clauza 1) la lista elementelor mai mici (al treilea argument), altfel este adăugat (clauza 2) la lista elementelor mai mari (al patrulea argument). Acest proces recursiv se oprește când lista de intrare (al doilea argument) ajunge la lista vidă.

Urmăriți execuția la:

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

1.6 Sortare prin interclasare

Sortarea prin interclasare împarte lista de intrare în doua sub-liste de lungimi egale (tehnica *divide et impera*). Apoi sortează cele două sub-liste și în final interclasează sub-listele deja sortate.




```
merge_sort(L, R):-
   split(L, L1, L2), % împarte L în doua subliste de lungimi egale
   merge_sort(L1, R1),
   merge_sort(L2, R2),
   merge(R1, R2, R). % interclasează sublistele ordonate
% split returnează fail dacă lista ii vidă sau are doar un singur element
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).
```

Predicatul *splitK/4* ia primele **K** elemente din lista de intrare **L** și le inserează în lista L1 (clauza 1), pe când restul elementelor sunt inserate în lista L2 (clauza 2). Predicatul *split* separă lista în două părți egale prin apelarea predicatului *splitK/4*, unde K este jumătate din lungimea listei (*K is Len/2*). Dacă lungimea listei de intrare este 0 sau 1, atunci apelul predicatului *split* va da fail, cauzând astfel rezoluția prin clauza 1 a *merge_sort/2* să dea fail – în acest punct, recursivitatea ar trebui să se oprească. Prin urmare, aceste apeluri vor fi unificate cu clauza 2 sau 3 a predicatului *merge_sort/2*. Predicatul *merge/3* execută unirea a două liste ordonate, prin adăugarea primului element din prima sau a doua listă la lista rezultată în funcție de care este mai mic.

Urmăriți execuția la:

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

3. Exerciții

1. Rescrieți predicatul *sel_sort/2* astfel încât să selecționeze cea mai mare valoare din partea nesortată, prin urmare să sorteze descrescător, folosind denumirea *sel_sort_max/2*. Predicatul *max1/2* poate fi creat prin modificarea predicatului *min1/2* din laboratorul anterior.

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

2. Rescrieți predicatul *ins_sort* utilizând recursivitate forward, folosind denumirea *ins_sort_fwd/2*.

   Recomandare: predicatul de insertion sort este format din două predicate, ambele folosind o abordare backwards, incercați să le modificați pe ambele într-o abordare forwards.

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

3. Implementați bubble sort cu un număr fix de treceri prin lista de intrare, folosind denumirea *bubble_sort_fixed/3*.

```
% bubble_sort_fixed(L, K, R). – K este numărul de treceri
?- bubble_sort_fixed([3,5,4,1,2], 2, R).
R = [3, 1, 2, 4, 5]
```

4. Scrieți un predicat care să sorteze o listă de caractere ASCII. (Puteți folosi o metodă de sortare la alegere).
   Sugestie: folosiți predicatul predefinit *char_code/2*

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

5. Scrieți un predicat care să sorteze o lista de sub-liste în funcție de lungimea sub-listelor.

```
?- 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
```

Opțional. Acest predicat poate fi îngreunat atunci când luăm în considerare cazul a două subliste cu lungimi egale, luăm cazul [1,1,1] și [1,1,2], predicatul ar trebui să analizeze suplimentar cele două liste în cazul de lungimi egale și să compare element cu element.

```
?- 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
```

6. Rescrieți predicatul *perm/2* fără a apela predicatul *append/3*, folosind denumirea *perm1/2*. Extragerea și ștergerea unui element trebuie realizate altfel.

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