COMP 333

Prolog Programming Project #1: Sorting Algorithms (Draft 2 04/17/23)

In this project you will write Prolog predicates that start with lists of random numbers as inputs and transform them via predicates into the corresponding sorted lists. The algorithms to be implemented are **selection sort** and **merge sort**. These algorithms are normally introduced in COMP 182, please review them on your own if you are not already familiar with them.

Prolog is best applied to problems that include a search component, and it is not the first choice for solving traditional computational problems like sorting. But these are a good application area for gaining practice. These problems don't use Prolog's built-in search and backtrack features, which is one of big advantages provided by Prolog. But this project shows how to use Prolog as a prototype checker to demonstrate that your core logical relationships for an algorithm are correct before proceeding to full implementation in a different language. In the next Prolog project, we'll pick a problem that uses search and backtrack more directly to find the solution.

Selection Sort

This is an $O(n^2)$ time complexity sort. The most efficient time complexity sorts are O(n log n), but the $O(n^2)$ ones are good practice problems because they are simple. We'll start with selection sort here.

Given an array of size n of initially random numbers, selection sort will sort the numbers in place by means of swaps. For **n elements** numbered 0 through n-1, there will be **n-1 passes** through the array. We can index each pass with value p from 0 through n-2. **Pass p sweeps through the array from index p through n-1**. The final pass n-2 covers only indexes n-2 and n-1.

During each pass, selection sort keeps track of the **index** of the minimum value observed during the pass. We maintain the **index of the minimum value**, not the value itself. The typical implementation is to keep track of minimum index mi, initialized to p, the starting index of the pass. During the pass, if any value is reached that is smaller than the current minimum, the new value becomes the minimum.
At the end of the pass, we have located the index of the minimum value for that pass. The final task of the pass is to swap the values in positions p and mi, so long as p does not equal mi. In this way the smallest values in the array are gradually pushed to the left end of the array. After all passes are completed the array is sorted.

In summary, the algorithm is implemented with (1) an iteration over p to drive the passes, (2) locating the minimum index during each pass, and (3) swapping the minimum value into its correct position.

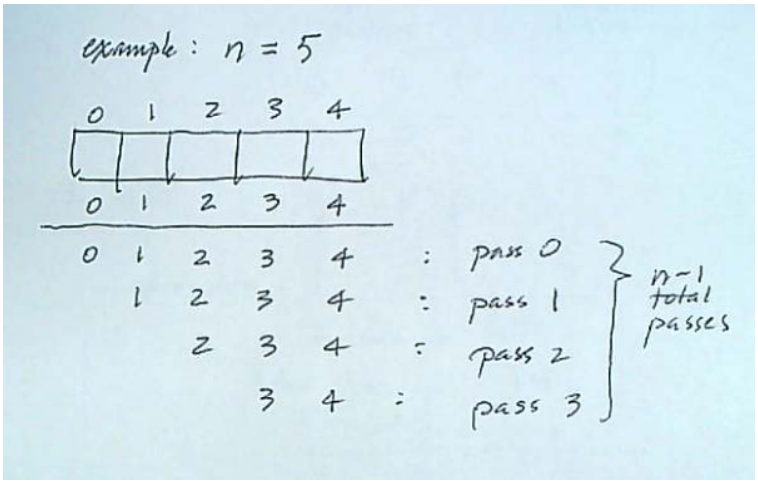

Figure: snapshot of list with n = 5, showing n-1 = 4 passes

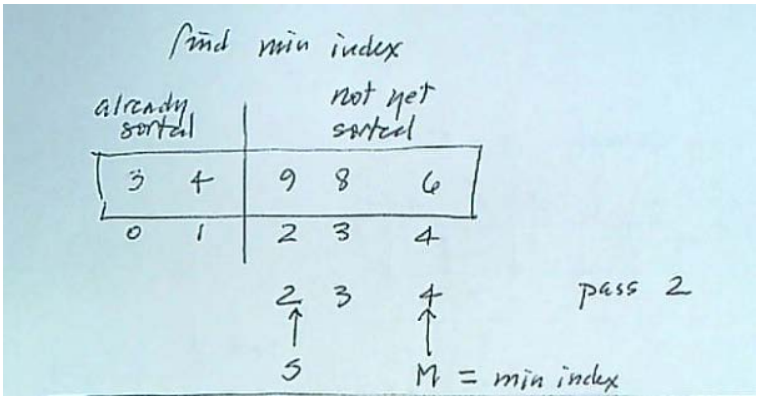

Figure: finding the min index during pass 2

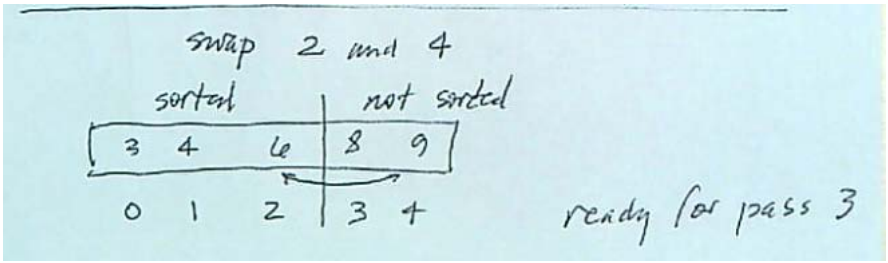

Figure: swapping value at min index M into its proper position at S, completing pass 2

Predicates

Predicates for Test Support

Create the following Prolog predicates that will be used for all sorting algorithms.

- rv(R,V): given an input R that defines a range, provide output value V which is a random value on the range 0 to R-1.
- rl(N,R,L): given N for number of values, and R for range, provide output list L that contains N numbers on range R.
- is_sorted(L): given list L of numbers, evaluates to true if values are sorted (non-descending), false otherwise.

rv(R,V)
use built-in predicate random/1 to generate a random number on the range 0 to 1. Then multiply by R and take the floor of the result to produce V.
rl(N,R,L)
use rv(R,V) to generate random values on the range R. Create a list of length N to produce L. The obvious way to generate the list is to use a base case N = 1 rule to create a list of length 1. For the general case, call the predicate recursively on N-1 yielding a temporary list, then generate another random value and append it to the temporary list yielding L. A less obvious but simpler implementation uses the maplist built-in predicate which will be discussed in class.
is_sorted(L)
if the list is length 1 (base case), use a fact to state it is true; then create a rule for the recursive case; length of L > 1, use pattern matching to obtain values of the first two items in the list, item 1 <= item 2, and then call the function recursively on tail of L.

Selection Sort Predicates

- swap(L1,A,B,L2): given a list L1, provide a list L2 with values at indexes A and B or exchanged.
- min_index(L,S,MI): given a list L and starting at index S continuing through to the end of the list, provide MI as the index of the minimum value.
- selection_sort(L1,L2): given list L1, use selection sort algorithm to provide resulting list L2 with values in sorted order.
- selection_sort_test(N,R): uses rl to generate a list of N random numbers on range R, selection_sort to sort them, writeln to display both the original and sorted lists, and is_sorted to confirm that the sorted list is sorted.

swap(L,A,B,M)
use nth0/3 and nth0/4 built-in predicates to find the values in list L at indexes A and B and provide updated list M with the values swapped; this function uses 2 calls to nth0/3 to find values AV and BV at positions A and B in L, two calls to nth0/4 to produce a temporary list with BV replaced with AV, then two calls to nth0/4 to produce final list M with AV replaced with BV. The predicate doesn't need a base case and it is not recursive.
min_index(L,S,MI)
If the list is length 1 (base case), MI is 0. For the rule in the general case, the top level predicate determines the value of two extra index parameters and calls the helper function selection_sort/5 to do the rest of the work. Use S for current index, E for ending index, and M for current minimum. Use different variables for M (current value of minimum index, which might not be the same as the final answer) and MI (the final answer when the pass is completed).
min_index(L,S,E,M,MI)
This is the helper predicate. Determine the values of SV (value in L at position S) and MV (value in L at position M). When min_index/5 is initially called by min_index/3, it uses 1 for S, length of L for E, and 0 for M. If MV <= SV, then call the predicate recursively with S+1, E, and M (L and MI don't change). If SV < MV, then call recursively with S+1, E, and S replacing M. In a separate clause (fact), if S = E or S > E, then we have reached the end of the list and now match MI with M: MI = M.

selection_sort(L,M)

This is the top level predicate. Check for base case L contains 1 element. In a separate rule for the general case, introduce two new parameters S and E for starting and ending indexes. S is initially 0 and E is length of L – 2 (the last pass starts at length-2, not length-1). Call the helper predicate to do the rest of the work.

selection_sort(L,S,E,M)

This is the helper predicate. Use min_index to find index for pass S. Use swap to swap values at S and MI if they are different, yielding a temporary list L1. Call the predicate recursively with L1 and S+1. In a separate clause (fact), when S = E or S > E, match M with L (M = L), yielding final sorted list M.

sort_test(A,N,R,X) or **sort_test(A,N,R)**

A is the symbol defining the algorithm, N is the size of the random number list, R is the range, and X is the final sorted list. X is optional, it's only required if you plan to use it in a follow-on query. Create a list of random numbers. Use call/3 to call the sort predicate named by A to obtain the list of sorted numbers. Use writeln to display the original and sorted lists. Use is_sorted to check that the sorted list is sorted.

Example Runs

```
?- sort_test(selection_sort, 10, 10000, S).
        [325,5451,4388,6525,7451,1568,8898,443,4835,3489]

        [325,443,1568,3489,4388,4835,5451,6525,7451,8898]

        S = [325, 443, 1568, 3489, 4388, 4835, 5451, 6525, 7451|...] .

?- sort_test(selection_sort, 10, 10000, S).
        [7405,7291,3862,824,9233,3154,3047,7113,9135,1335]

        [824,1335,3047,3154,3862,7113,7291,7405,9135,9233]

        S = [824, 1335, 3047, 3154, 3862, 7113, 7291, 7405, 9135|...] .

?- sort_test(selection_sort, 100, 10000, S).
        [7655,9359,1201,8494,6773,9208,521,5510,7580,3487,3314,7632,9335,3860,2954,2349,2320,9943,5698,1599,6537,
        3439,4311,1021,8326,4569,4505,3275,2699,56,3870,1553,368,8653,2359,6736,1591,6326,4613,47,2567,1467,7269,
        1527,6916,615,6312,3876,8151,9319,3757,8124,9821,2801,5609,8710,4152,396,4877,9132,5187,8689,8003,7506,35
        59,6824,2979,8127,9181,2345,9748,721,2913,6846,4260,9145,7987,5056,9022,9054,1945,7578,7655,1212,2668,945
        7,270,4842,8448,180,4530,4651,4851,3365,5579,9520,1443,3739,1494,8889]

        [47,56,180,270,368,396,521,615,721,1021,1201,1212,1443,1467,1494,1527,1553,1591,1599,1945,2320,2345,2349,2
        359,2567,2668,2699,2801,2913,2954,2979,3275,3314,3365,3439,3487,3559,3739,3757,3860,3870,3876,4152,4260,4
        311,4505,4530,4569,4613,4651,4842,4851,4877,5056,5187,5510,5579,5609,5698,6312,6326,6537,6736,6773,6824,6
        846,6916,7269,7506,7578,7580,7632,7655,7655,7987,8003,8124,8127,8151,8326,8448,8494,8653,8689,8710,8889,9
        022,9054,9132,9145,9181,9208,9319,9335,9359,9457,9520,9748,9821,9943]

        S = [47, 56, 180, 270, 368, 396, 521, 615, 721|...] .
```

Alternative Implementation

The predicates can be written as described above by following the general computational steps in procedural style as usual in Java or other procedural language. But Prolog naturally supports declarative style, so let's try an alternative implementation using declarative style when possible.

min_index(L,S,MI)

First use append/3 and length/2 to split the list L into L1 of length S, and L2. Then use min_list/3 built-in to find the minimum value in L2. Next use nth0/3 to find the index position of the minimum value in L2. Finally compute the desired minimum index value MI which is index in L2 + S.

swap(L,A,B,M)

No suggested changes from the earlier implementation.

selection_sort(L,M)

Use min_index and swap starting at position 0 to get an updated list. Call predicate recursively on tail of updated list, yielding a sorted list. Finally cons head of list to sorted list yielding final result M.

Merge Sort

Merge Sort is an O(n log n) algorithm that works by recursively splitting the array into two halves, sorting each half, then merging the two halves back into the final array. Merging is O(n), and the number of merges is O(log n). Be sure to distinguish between the **merge** operation and the **mergesort** operation.

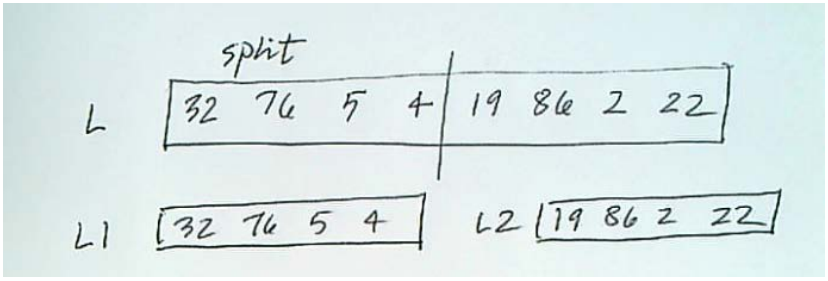

Figure: split original list into two lists of approximately equal lengths

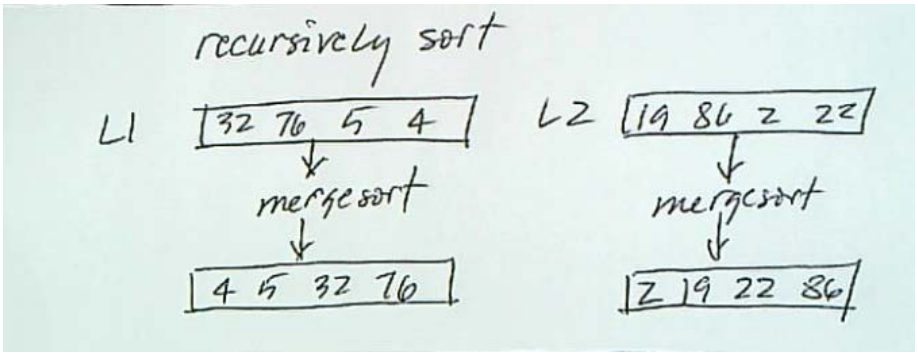

Figure: recursively sort each half separately.

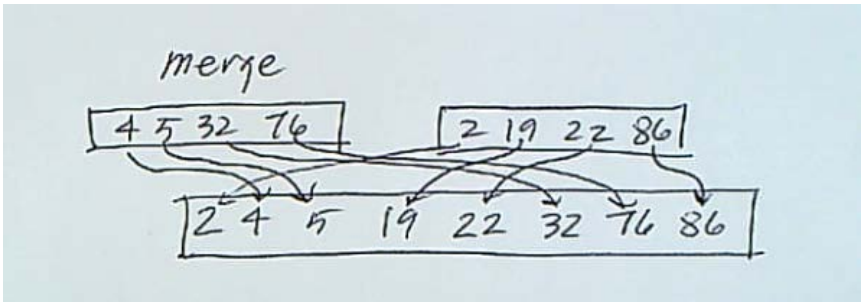

Figure: merge the two sorted halves into final sorted list

Predicates

split(L,L1,L2)

Split list L into two approximately equal-sized halves L1 and L2. Use append/3 (running backwards) and length/2. Note that Prolog provides both floating point divide "/" and integer divide "//". Use integer divide when calculating half the length of L.

merge(L1,L2,M)

Merge two lists L1 and L2 **which must already be sorted** into sorted list M (merging two unsorted lists is a harder problem!) If head of L1 < head of L2, call predicate recursively on tail L1, L2, producing temporary list L3. Then append head of L1 to head of L3 producing M. If head of L2 < head of L1, then proceed similarly with the roles of the lists reversed.

merge_sort(L,M)

Split L into two lists L1 and L2. Call merge_sort recursively on each of L1 and L2 producing L1S and L2S. Merge L1S and L2S yielding M.

To Submit

Create one kb file containing definitions for all required predicates for both selection sort and merge sort. Each predicate can be unit tested from the interpreter.

Typical Test Cases

Predicate Unit Test

```
rv(1000,X).
rl(10,1000,X), writeln(X).
is_sorted([3,4,5]).
is_sorted([3,4,2]).
min_index([50,40,30,70,90], 0, X).
min_index([20,40,30,10,70,90], 1, X).
swap([a,b,c,d,e],1,3,X).
split([a,b,c,d,e],X,Y).
merge([10,20,30,40],[5,6,25,42],X).
```

Complete Sort Tests

```
rl(100,10000,L1), selection_sort(L1, L2), is_sorted(L2), writeln(L1), writeln(L2).
rl(1000,100000,L1), merge_sort(L1,L2), is_sorted(L2), writeln(L1), writeln(L2).
```

When I am evaluating your programs, I will generally stick to the complete sort tests. I may need to run unit tests of support predicates if the top level predicates don't seem to be correct. You should run your own predicate unit tests during development. It's one of the best strategies for developing programs in Prolog. It's a big waste of your time trying to debug more complex predicates unless you have already demonstrated that the simpler ones are correct and working.

Timing (Optional)

Prolog uses lists for aggregate data storage, which are implemented as linked lists. There's no real equivalent to an array in which items are reached via direct indexing. Sorting algorithms are mostly optimized for array storage. Running them in Prolog by sorting linked lists can never be as fast as sorting arrays of items. But there is still a significant difference between selection_sort $O(n^2)$ and merge_sort O(n log n) predicates even when run on linked lists. If you're curious, you can use the **time/1** predicate to estimate execution time for a predicate. If you are timing a compound predicate with multiple goals, put an extra set of parens around the whole compound query so that it appears as a single argument to **time/1**.

```
time( ( rl(1000, 100000, L1), selection_sort(L1,L2) ) ).    % time both rl and selection_sort
rl(1000,100000,L1), time( selection_sort(L1,L2) ).          % time only selection_sort
```

Example Formatting Style for "sorting.pl" Source File

```
%-------------------------------------
% Prolog Sorting Programming Project:  selection sort and merge sort
%-------------------------------------

%---------------------------------------
% support predicates rv/2, rl/3, is_sorted/1
%---------------------------------------

…

%---------------------------------------
% Selection Sort Predicates:  swap/4, min_index/3, selection_sort/2
%---------------------------------------

…

%---------------------------------------
% Merge Sort Predicates:  split/3, merge/3, merge_sort/2
%---------------------------------------

…
```