**COMP 333**

**Prolog Programming Project #1: Sorting Algorithms (Draft 04/03/23)**

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, insertion sort, merge sort, and quick 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.

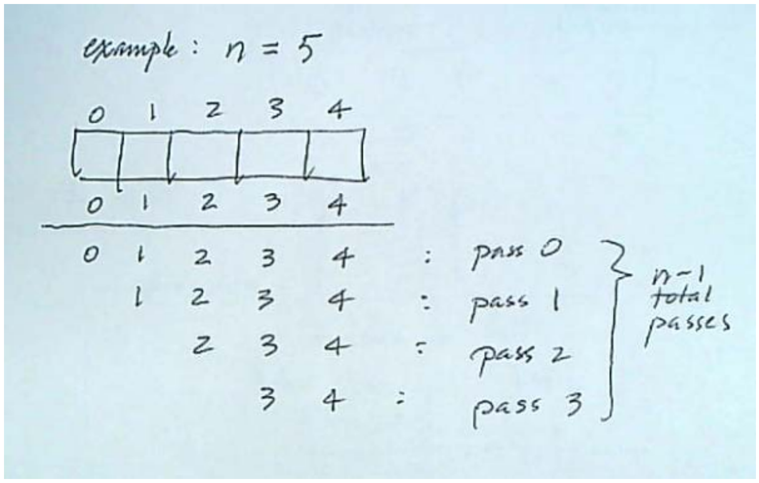

Figure: snapshot of list with n = 5, showing n-1 = 4 passes

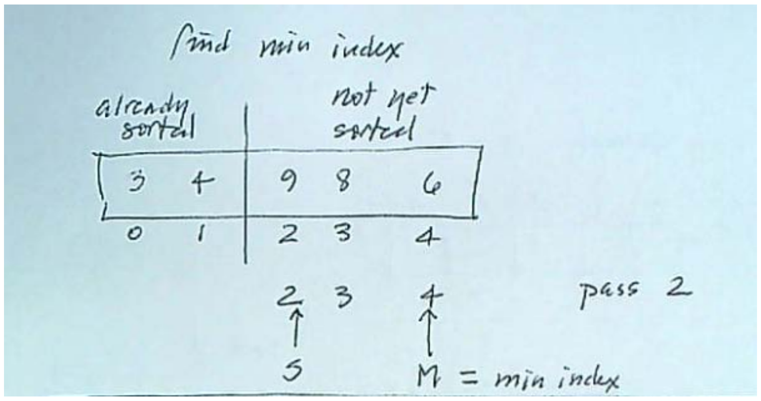

Figure: finding the min index during pass 2

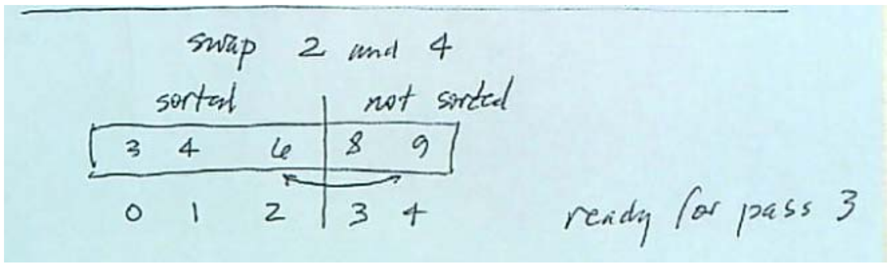

Figure: swapping value at min index M into its proper position at S, completing pass 2

**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.
- find_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). Initially, S is 1, E is length of L – 1, and M is 0.
**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). 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).
**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|...] .
```

**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.

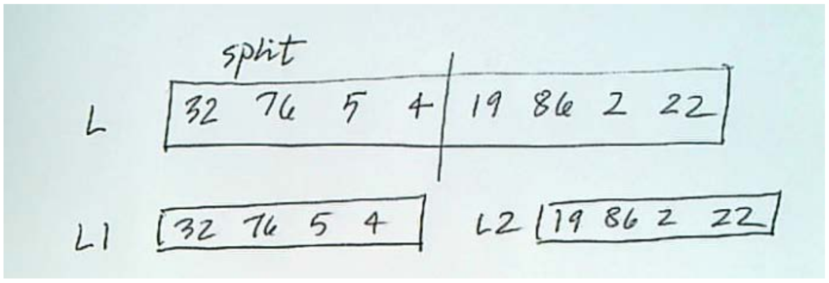

Figure: split original list into two lists of approximately equal lengths

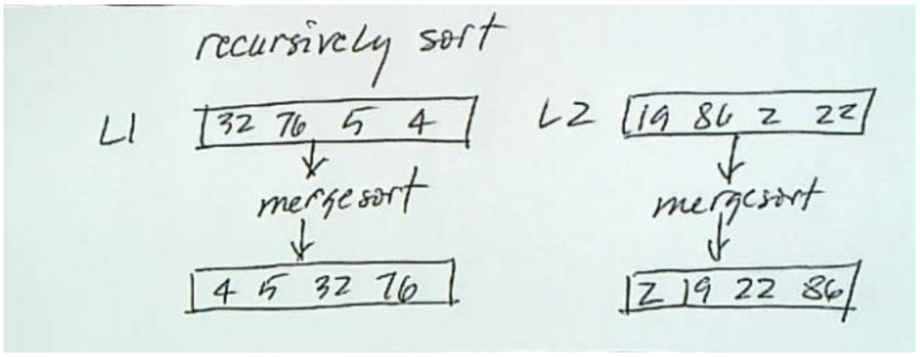

Figure: recursively sort each half separately.

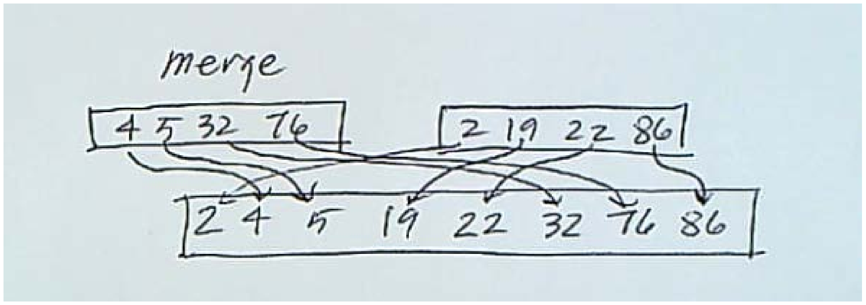

Figure: merge the two sorted halves into final sorted list

**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.

**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.

**Quick Sort**

Quick Sort is another O(n log n) algorithm that has some similarities to merge sort. But instead of using the split operation, it uses the **partition** operation. To partition the array or portion of the array, we pick a pivot value. Ideally the best performance requires the pivot to be the median value of the segment, but any pivot will work but with less-than-optimal performance. Once the pivot is determined, divide the array into three subarrays: values < pivot, values = pivot, values > pivot. Recursively sort the first and third subarrays, then append the sorted first array, the second array (which doesn't need to be sorted), and the sorted third array.

Famous implementations of Quick Sort have produced some of the fastest of all sorts based on comparisons. Ironically, Quick Sort has poor worst-case performance, specifically when the original values are already sorted. The adaptive $O(n^2)$ sort insertion sort does better than the O(n log n) quicksort for sorting lists that are "partially" sorted. Look up C.A.R. Hoare, the computer scientist who invented quicksort and who has written extensively on it, and Robert Sedgewick, who introduced improvements and optimizations of the original. Much of the history of the algorithm is focused on squeezing out incrementally improved performance. For our Prolog version, performance isn't the focus, so work on making the partition step faster is not relevant.

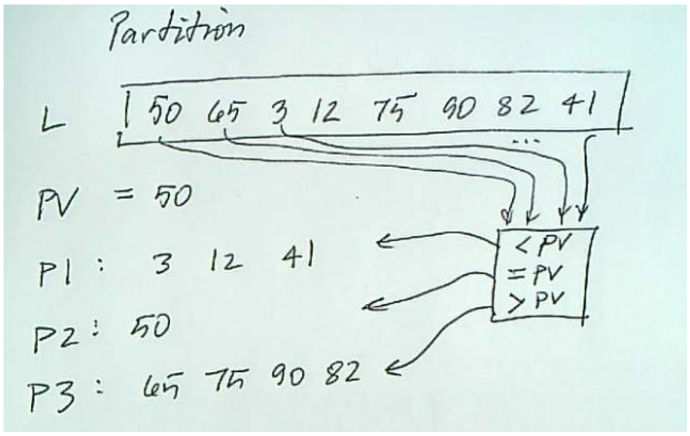

Figure: partition L by dividing into lists P1 P2 P3 based on pivot value PV = 50

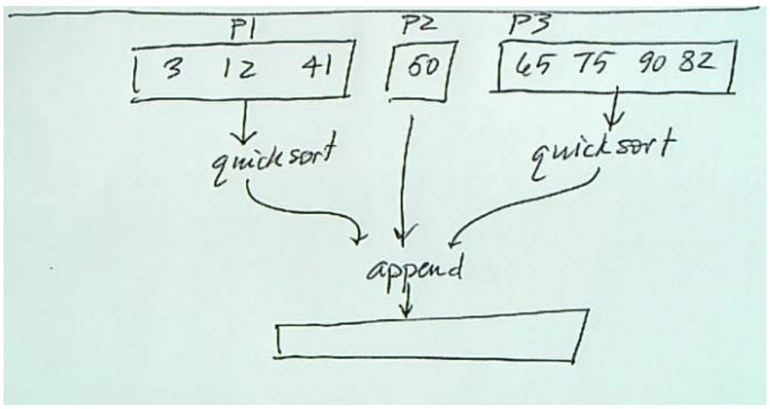

Figure: after partitioning, recursively sort P1 and P3, then append all partitions to get result.

**Predicates**

**partition(L,P1,P2,P3)**

This is the top level partition predicate. Given list L, find pivot value, then partition L into three lists P1 with values less than pivot, P2 with values = pivot, and P3 with values > pivot. First, use pattern matching to divide L into head and tail. Assume head is the pivot value. Call the helper function with C = 0, E = length of L – 1, L = tail of L, PV = head of L, temporary partitions P1T, P2T, P3T all [], and P1, P2, P3 final values left unchanged.

**partition(C,E,L,PV,P1T,P1,P2T,P2,P3T,P3)**

The helper function has 10 parameters because each of the 3 partitions needs a temporary value (P1T,P2T,P3T) and a final value (P1,P2,P3). The other parameters are C for current index, E for ending index, L for list to partition, and PV for pivot value. Check value of L at position C (CV) and compare it to PV (pivot value). Call predicate recursively with C+1. Update temporary partitions in the recursive call as follows. If CV < PV, append CV to L1T. If CV = PV, append to L2T, else append to L3T. Also define a base case (fact) when C = E or C > E. When partition is complete, match temporary partitions with final partitions:  P1 = P1T, P2 = P2T, P3 = P3T.

**quick_sort(L,M)**

Partition L into L1, L2, and L3. Call quick_sort recursively on L1 and L2 yielding L1S and L2S. Append L1S, L2 (which does not need to be sorted) and L3 yielding M. This will require multiple calls to append with temporary list results.

**Insertion Sort TBD**