20CS6033/4033 Fall 2022

**Assignment #3**

Assigned 9/27/2022

Due 10/6/2022, 11:59PM on Canvas

Sorting algorithms in Prolog

**50 points**

As we know from the study of data structures and algorithms courses, sorting algorithms come in two general classes: iterative and recursive. The first type comprises algorithms such as *Bubble Sort* and *Insertion Sort*. The second type comprises of algorithms such as *Merge Sort* and *Quick Sort*.

The complexity of list processing algorithms such as sorting algorithms depends on the input size – in this case on the list size or the length of the list:

* **Recursive algorithms** are in general preferred: it is easier to prove their correctness and for longer lists their complexity is far superior to that of iterative algorithms.
* However, for short lists, the overhead due to recursion may offset any improvement due to the actual algorithm.
* So, we can imagine hybrid algorithms that **behave** as recursive algorithms for large lists and switch to an iterative type of behavior for short lists.

In this assignment you are provided with incomplete prolog code for

1. Bubble sort
2. Insertion sort
3. Merge sort
4. Quick sort

We will call (1) & (2) “small sorts”, and (3) & (4) “big sorts”.

We wish to design and implement a **hybridSort** as follows:

**hybridSort** takes **FIVE** arguments: hybridSort(LIST,SMALL,BIG,THRESHOLD,SLIST),

where

1. THRESHOLD is a positive integer
2. SMALL is one of the small sorts (Bubble sort, and insertion sort)
3. BIG is one of the big sorts (Merge sort, and Quick sort)
4. SLIST is the sorted version of LIST
5. ~~ORDER is less or greater~~

Thus, **hybridSort** behaves as follows:

* When the length of LIST is less than THRESHOLD, then **hybridSort** **calls** **SMALL**

**example: if the threshold is say 6, and the list L has length 5, then**

**?-hybridSort(L, bubbleSort, mergeSort, 6, SLIST, less)** calls **bubbleSort(L, SLIST)**

* When the length of list LIST is greater than or equal to THRESHOLD, then **hybridSort** **behaves** like one of the **BIG** sorts (attention, it does not call a big sort, for if it did, it will immediately switch to **that** sort).

However, if L has 10 elements

**?-hybridSort(L, bubbleSort, mergeSort, 6, SLIST, less)** behaves like **mergeSort, i.e., we must implement hybridSort to do this; We do NOT call mergeSort!, Why?**

**Because, if we did, then the remainder of the list will be sorted by mergeSort, so even though the halves of L will have length less than the threshold, they will NOT be sorted by bubbleSort, they will be sorted by mergeSort.**

For example:

The query

**?- hybridSort(LIST, bubbleSort, mergeSort, 5, SLIST).**

will be solved as follows: compare the length of LIST to 5: if it is less than 5 then **bubbleSort** is called; if it is greater than or equal to 5 then **hybridSort** behaves like **mergeSort**.

The query

**?-hybridSort(LIST, bubbleSort, quickSort, 5, SLIST).**

will be solved as follows: compare the length of LIST to 5: if it is less than 5 then **bubbleSort** is called; if it is greater than or equal to 5 then **hybridSort** behaves like **quickSort**.

The query

**?-hybridSort(LIST, insertionSort, mergeSort, 5, SLIST).**

will be solved as follows: compare the length of LIST to 5: if it is less than 5 then **insertionSort** is called; if it is greater than or equal to 5 then **hybridSort** behaves like **mergeSort**.

The query

**?-hybridSort(LIST, insertionSort, quickSort, 5, SLIST).**

will be solved as follows: compare the length of LIST to 5: if it is less than 5 then **insertionSort** is called; if it is greater than or equal to 5 then **hybridSort** behaves like e **quickSort**.

In your program, write first the predicate **randomList** which will generate a random list of a given number of elements. That is, **randomList** will have two arguments – the number of elements, and the list it generates: **randomList(N, LIST).** To implement **randomList**, use the built in predicate **random(Lower, Upper, X)**, which generates X as a random value between Lower and Upper.

What to turn in:

* 1. Implement **randomList**
  2. Save the list created into the knowledge base by using the built-in commands: **dynamic/1 assertz/1**
  3. The complete code for all the sorts shown below.
  4. The result of eight runs as follows:
     1. Run each of the simple sorting algorithms individually, on the same list of length at least 50.
     2. Run hybridSort on all the four pairs of the simple algorithms, on the same list used in the previous 4 runs

In the incomplete code below, you must

1. Fill in correctly all the places where FILLINHERE occurs
2. Write succinct, but meaningful comments describing each predicate
3. Complete the definition of hybridSort.

/\*swap the first two elements if they are not in order\*/

swap([X, Y|T], [Y, X | T]):-

Y =< X.

/\*swap elements in the tail\*/

swap([H|T], [H|T1]):-

swap(T, T1).

/\* Comment describing bubbleSort \*/

bubbleSort(L,SL):-

swap(L, L1), % at least one swap is needed

!,

bubbleSort(L1, FILLINHERE).

bubbleSort(L, L). % here, the list is already sorted

/\* Comment describing ordered \*/

ordered([]).

ordered([\_X]).

ordered([H1, H2|T]):-

H1 =< H2,

ordered([H2|T]).

/\*Comment describing insert(E, SL, SLE) …\*/

/\*Comment describing the 1st clause of insert …\*/

insert(X, [],[X]).

insert(E, [H|T], [E,H|T]):-

ordered(T),

FILLINHERE(E, H),

!.

/\*Comment describing the 2nd clause of insert …\*/

insert(E, [H|T], [H|T1]):-

ordered(T),

insert(E, T, FILLINHERE).

/\* Comment describing insertionSort \*/

insertionSort([], []).

insertionSort([H|T], SORTED) :-

insertionSort(T, T1),

insert(H, T1, FILLINHERE).

/\* Comment to describe meregeSort… \*/

mergeSort([], []). %the empty list is sorted

mergeSort([X], [X]):-!.

mergeSort(L, SL):-

split\_in\_half(L, L1, L2),

mergeSort(L1, FILLINHERE),

mergeSort(L2, S2),

merge(S1, FILLINHERE, SL).

/\* Comment to describe split\_in\_half…\*/

intDiv(N,N1, R):- R is div(N,N1).

split\_in\_half([], \_, \_):-!, fail.

split\_in\_half([X],[],[X]).

split\_in\_half(L, L1, L2):-

length(L,N),

intDiv(N,2,N1),

length(L1, FILLINHERE),

append(L1, L2, L).

/\* Comment describing merge(S1, S2, S) \*/

merge([], L, L). % comment

merge(L, [],L). % comment

merge([H1|T1],[H2|T2],[H1| FILLINHERE]):-

H1 FILLINHERE H2,

merge(T1,[H2|T2],T).

merge([H1|T1], [H2|T2], [H2|T]):-

FILLINHERE =< FILLINHERE

merge([H1|T1], T2, FILLINHERE).

/\* Comment describing split for quickSort \*/

split(\_, [],[],[]).

split(X, [H|T], [H|SMALL], BIG):-

H =< X,

split(X, T, SMALL, FILLINHERE).

split(X, [H|T], SMALL, [H|BIG]):-

X =< H,

split(X, T, FILLINHERE, BIG).

/\* Comment describing quickSort \*/

quickSort([], []).

quickSort([H|T], LS):-

split(H, T, SMALL, FILLINHERE),

quickSort(SMALL, S),

quickSort(BIG, B),

append(S, [H|B], FILLINHERE).

/\* Comment describing hybridSort \*/

hybridSort(LIST, bubbleSort, BIGALG, T, SLIST):-

length(LIST, N), N=<T,

bubbleSort(LIST, FILLINHERE).

hybridSort(LIST, insertionSort, BIGALG, T, SLIST):-

length(LIST, N), N=<T,

insertionSort(LIST, SLIST).

hybridSort(LIST, SMALL, mergeSort, T, SLIST):-

length(LIST, N), N>T,

split\_in\_half(LIST, L1, L2),

hybridSort(L1, SMALL, mergeSort, T, S1),

hybridSort(L2, SMALL, mergeSort, T, S2),

merge(S1,S2, SLIST).

hybridSort([H|T], SMALL, quickSort, T, SLIST):-

length(LIST, N), N>T,

split(H, T, L1, L2),

FILLINHERE several lines in the body of this clause

append(S1, [H|S2], SLIST).

hybridSort([H|T], SMALL, quickSort, T, SLIST):-

FILLINHERE the full body of this clause