Armstrong Atlantic State University
Engineering Studies
MATLAB Marina – Sorting Primer

Prerequisites
The Sorting Primer assumes knowledge of the MATLAB IDE, MATLAB help, arithmetic operations, built in functions, scripts, variables, arrays, logic expressions, conditional structures, iteration, functions, debugging, characters and strings, cell arrays, structures, file input and output, and searching. Material on these topics is covered in the MATLAB Marina Introduction to MATLAB module, MATLAB Marina Variables module, MATLAB Marina Arrays module, MATLAB Marina Logic Expressions module, MATLAB Marina Conditional Structures module, MATLAB Marina Iteration module, MATLAB Marina Functions module, MATLAB Marina debugging module, MATLAB Marina Character and Strings module, MATLAB Marina Cell Arrays module, MATLAB Marina Structures module, MATLAB Marina File Input and Output module, and MATLAB Marina Searching module.

Learning Objectives
1. Be able to describe the operation of the different types of sorts: selection, insertion, quick, merge, and bucket.
2. Be able to use MATLAB's `sort` function to sort collections of primitive data types.
3. Be able to write sort functions (selection) for collections of data of non-primitive data types such as cell arrays and structure arrays.

Terms
sorting, selection sort, quick sort

MATLAB Functions, Keywords, and Operators
sort

Sorting
Sorting involves ordering a collection of data in some way. Arranging integers from smallest to largest or largest to smallest and arranging names in alphabetical order are examples of sorting. There are many different types of sorts, some examples of sorts are: selection sort, insertion sort, quick sort, bucket sort, and merge sort.

A selection sort works by searching the collection for the next element in the ordering until the entire collection is ordered. Sorting a collection in increasing order would involve: finding the smallest element and placing that in the first spot, then finding the next smallest and placing it in the second spot, and continuing this procedure until all elements have been placed in order. Figure 1 shows an example of a section sort function that sorts an array of numbers in increasing order.

```
function result = selectionSort(data)
% ------------------------------------------------------
% selectionSort.m
% ------------------------------------------------------
% syntax: result = selectionSort(data)
%  data is the collection of data to sort
%  result is ordered list of numbers (increasing order)
% ------------------------------------------------------
% Notes: data must be a 1D array of numbers
% ------------------------------------------------------

result = data;
% search list number times equal to its size
for pass = 1 : 1 : length(result)-1
    smallindex = pass;
    % search remaining unordered sublist for next smallest element
    for k = pass + 1 : 1 : length(result)
        if (result(k) < result(smallindex))
            % update next smallest element if necessary
            smallindex = k;
        end
    end

    % put next smallest element in correct place by swapping
    if (smallindex ~= pass)
        temp = result(pass);
        result(pass) = result(smallindex);
        result(smallindex) = temp;
    end
end
end
```

Figure 1, `selectionSort` function

Notice that the selection sort requires a nested loop. The outer loop keeps track of which pass and the inner loop is to search the sublist for the desired value. A selection sort must search the collection a number of times equal to the number of elements minus one, when it finds the next smallest element it will switch it with the value in the current position. For example, to sort the array [17, 4, 13, 9, 2, 10] requires five passes:

- After the first pass, the partially sorted list is [2, 4, 13, 9, 17, 10]
- After the second pass, the partially sorted list is [2, 4, 13, 9, 17, 10]
- After the third pass, the partially sorted list is [2, 4, 9, 13, 17, 10]
- After the fourth pass, the partially sorted list is [2, 4, 9, 10, 17, 13]
- And after the fifth pass, the sorted list is [2, 4, 9, 10, 13, 17]

Sorting Examples
Consider sorting the array [17, 7, 3, 14, 9, 2, 21, 8]:

With a selection sort:
[17, 7, 3, 14, 9, 2, 21, 8]
Pass 1, [2, 7, 3, 14, 9, 17, 21, 8]
Pass 2, [2, 3, 7, 14, 9, 17, 21, 8]
Pass 3, [2, 3, 7, 14, 9, 17, 21, 8]
Pass 4, [2, 3, 7, 8, 9, 17, 21, 14]
Pass 5, [2, 3, 7, 8, 9, 17, 21, 14]
etc.

With a merge sort:
[17, 7, 3, 14, 9, 2, 21, 8]
[17, 7, 3, 14] [9, 2, 21, 8]
[17, 7] [3, 14] [9, 2] [21, 8]
[17] [7] [3] [14] [9] [2] [21] [8]
[7, 17] [3, 14] [2, 9] [8, 21]
[3, 7, 14, 17] [2, 8, 9, 21]
[2, 3, 7, 8, 9, 14, 17, 21]

With a quick sort using the median of sublists as pivot (best case):
[17, 7, 3, 14, 9, 2, 21, 8]
[7, 3, 2, 8] [9] [17, 14, 21]
[3, 2] [7] [8] [14] [17] [21]
[2] [3]
[2, 3]
[2, 3, 7, 8] [14, 17, 21]
[2, 3, 7, 8, 9, 14, 17, 21]

With a quick sort using first element of sublists as pivot:
[17, 7, 3, 14, 9, 2, 21, 8]
[7, 3, 14, 9, 2, 8] [17] [21]
[3, 2] [7] [14, 9, 8]
[2][3] [9, 8] [14]
[8][9]
[2, 3] [8,9]
[8, 9, 14]
[2, 3, 7, 8, 9, 14]
[2, 3, 7, 8, 9, 14, 17, 21]

A quick sort works well if it gets good pivots for each sublist. The pivot can be the median of the sublist (best case) or could use the first element of list and hope it is close to the median.

Estimating Efficiency of Algorithms

Big O notation allows one to characterize functions by their growth rates. Big O notation is commonly used to analyze algorithms in terms of utilization of computer resources and can be used to estimate the worst case performance (computing cost in terms of execution time, memory required) of an algorithm independent of the computer architecture it is run on.

The Big O of an algorithm is expressed as O(function of N), where N is size of the collection to be processed. Typically we are concerned with the computation cost as N gets large (approaches infinity).

A simplified set of rules for Big O notation is:

- For sequential operations, the Big O values of each operation are added
- For nested operations the Big O values of each operation are multiplied
- If the Big O function is a sum of terms, keep only the term with the largest growth rate
- If the Big O function is a product of terms, any constants that do not depend upon the size of the data being processed are omitted.

O(1) computation cost of the algorithm is independent of the size of the data, for example accessing an element in an array. O(N) computation cost of the algorithm is proportional to size of the data, for example copying an array . $O(N^2)$ computation cost of the algorithm is proportional to the square of the size of the data, for example gamma correction on an image Other Big O examples: finding an item in a sorted list using a binary search is O(log N), finding an item in an unsorted list is O(N), sorting a list using an insertion sort is $O(N^2)$, and sorting a list using a quick sort is O(N log N).

The MATLAB code segment of Figure 2a generates a multiplication table for up to N*N. The algorithm for the code of Figure 2a is $O(N^2)$, the disp operation is performed N^2 times.

```
N = 3;
for k1 = 1 : 1 : N
    for k2 = 1 : 1 : N
        disp([int2str(k1), '*', int2str(k2), '=', int2str(k1*k2)]);
    end
end
```

Figure 2a, MATLAB Code to Generate Multiplication Table

The MATLAB code segment of Figure 2b displays the elements of an array of size N. The algorithm for the code of Figure 2b is O(N), the disp operation is performed N times.

```
arr = [1:1:N];
for k = 1 : 1 : length(arr)
        disp(arr(k));
end
```

Figure 2b, MATLAB Code to Display the Elements of an Array

A somewhat simplistic view of Big O (at least for basic looping algorithms) is that for p nested loops, the algorithm is $O(N^p)$. This simplistic view can overestimate the worst case efficiency.

When there is a choice of algorithms to perform the same operation, the more efficient algorithm is generally preferable especially if there are resource constraints such as limited memory, storage, or processor time. For example, the worst case to search a sorted list of 10000 items using a binary search is 10000 comparisons (N = 10000) whereas the worst case for searching the same list using a binary search is 14 comparisons (log2(N) = 13.29).

Last modified Monday, November 18, 2013