SORTING

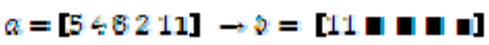
Sorting lists of numbers is a commonly discussed skill covered in introductory computer programming courses. In terms of engineering calculations, it is not the most important topic that will be covered. However, sorting is used by engineers just enough to make it important and certain tricks exist in sorting programming that are useful in other types of programs.

In this tutorial, two different sorting techniques will be covered though many exist. In terms of introductory material, two techniques is plenty.

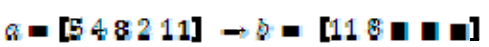
A good exercise to attempt before considering refined techniques is to take a list of numbers and sort them by hand. A list of 5 numbers is plenty. The student should take a list of 5 numbers and sort them. Then consider the question: How would I tell MATLAB to do what I just did (possibly using pseudocode)? Unlike the numerical methods discussed in previous chapters, each step should be easy to see by physically solving the problem by hand.

Sledgehammer Sorting

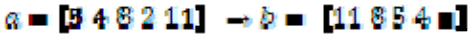
One technique for sorting an array is called sledgehammer sorting (other names may exist). The idea is that one finds the largest value in the array, puts that value in the left most empty slot in a new array, and then deletes that maximum from the original array. This is repeated until all of the values in the original array are gone. If one were to sort the values 5, 4, 8, 2, and 11, the first iteration would look something like:

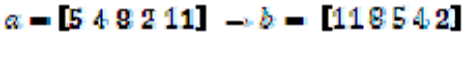


All three steps have been carried out. 11 was identified as the largest value in *a*. 11 was placed in the left most empty slot in *b* (the dark squares indicate empty spots). Finally, the 11 was crossed out from the original array. Now when the second iteration is carried out, 8 is seen as the largest value in *a*, because 11 does not exist anymore in *a*:



If this were continued for the final three elements in *a*, the last three iterations would look like:





The sorted vector is a new vector, *b*. The original vector, *a*, now has nothing left in it. If one were to write pseudocode for this method, it might look something like:

Pseudocode for Sledgehammer Sort

Input vector of numbers to sort
Identify the largest element in the vector
Make that value the 1st element in the sorted array
Cross out the element in the original array
Identify the new largest element in the original vector
Make that value the 2nd element in the sorted array
Cross out the element in the original array
Repeat so that these three steps are carried out a number of times equal to the length of the original array
Output the sorted array

Now this could be translated into a MATLAB function. One change from the pseudocode involves crossing out an element. The normal convention is to reassign the value to an insignificant number rather than actually crossing it out. In a sorting function that starts with the maximum value, the best thing to do is reassign the value to a very large, negative value. For this code, the author has used -10000 as the reassigned value. More elegant approaches would reassign the value to *-1/eps* or *–Inf*.

MATLAB Code for Sledgehammer Sorting

```
function [s]=sledgesort(a)
% a – the array to be sorted
% s – the sorted array from highest to lowest
len=length(a); % all elements will be sorted, so using the length will be helpful
for ii=1:len
        [m,p]=findmax(a);
        s(i)=m;
        a(p)=-10000;
end
```

In this code, the *findmax* function written in a previous tutorial is being used to find the maximum value rather than cutting and pasting that code in this m-file. The *findmax* function has been modified to return *m*, the maximum value, and *p*, the position of the maximum value. The max value is assigned to *s* in the 7th line of code. (There are 9 lines of code total counting the commented lines.) Then in line 8 the value in *a* that was extracted and placed in *s* is reassigned to a very large negative value. Each iteration resets a value in *a* to -10000. At the end, all elements of *a* are -10000.

If one were to identify the values of *m*, *p*, etc. for each iteration as they might do in a quiz, they would find the following values:

1st iteration
        m=11

p=5
s(1)=11
a(5)=-10000

2nd iteration
m=8
p=3
s(2)=8
a(3)=-10000

3rd iteration
m=5
p=1
s(3)=5
a(1)=-10000

4th iteration
m=4
p=2
s(4)=4
a(2)=-10000

5th iteration
m=2
p=4
s(5)=2
a(4)=-10000

One characteristic of this technique that is important to recognize is the fact that it actually stores 2 arrays that are the size of *a*, *a* itself and the new vector, *s*. Generally speaking, this has little effect if the array has roughly 100-1000 elements or less. In the days of room-sized computers that used punch cards, sorting a list of 100 numbers would have been difficult with this approach due to memory storage. For modern computers, larger arrays can be handled with ease using this approach, but it is important to realize that there is a limit. The major benefit of this sorting approach is the ease in programming and the logic that is relatively easy to follow.

Bubble Sorting

In bubble sorting, the approach to sort the array involves comparing the first two numbers in the array. If the one on the left is larger, then nothing happens. If the number on the right is larger, then they are switched. This is repeated down the length of the array. In the first iteration, the smallest number ends up in the right hand most place, while the larger number moves slowly or bubbles toward the beginning. In each subsequent iteration, the comparisons stop closer and closer to the beginning. In other words, each iteration does not require comparison of all of the elements, because the smallest elements have been placed.

If one were to display each step, i.e. each comparison of two numbers, they would look like the following:

Start: [5 4 8 2 11]
1st iteration (march through the array)
[5 4 8 2 11]
[5 8 4 2 11]
[5 8 4 2 11]
[5 8 4 11 2]
2nd iteration
[8 5 4 11 2]
[8 5 4 11 2]
[8 5 11 4 2]
3rd iteration
[8 5 11 4 2]
[8 11 5 4 2]
4th iteration
[11 8 5 4 2]

There are several important things to notice. The first is that the very first step yielded no change in the array. 5 is already bigger than 4, so nothing needed to happen. Second, four marches down the array, i.e. iterations, to compare values were carried out. This is equal to the length of the array minus one. That is not a coincidence. Thirdly, the first iteration made four comparisons, the second iteration made three comparisons, etc. The number of comparisons made for each subsequent iteration decreases by one. If it did not decrease by one each time, the correct answer would still be found. However, comparisons of previously sorted data would be carried out, which would decrease the efficiency of the method.

The pseudocode for this method would look something like the following:

Pseudocode for Bubblesort

Input the array to be sorted

Compare a(1) and a(2)
If a(1)<a(2)
Switch them
Compare a(2) and a(3)
If a(2)<a(3)
Switch them
Compare a(3) and a(4)
If a(3)<a(4)
Switch them
Compare a(4) and a(5)
If a(4)<a(5)
Switch them

Compare a(1) and a(2)

If a(1)<a(2)
Switch them
Compare a(2) and a(3)
If a(2)<a(3)
Switch them
Compare a(3) and a(4)
If a(3)<a(4)
Switch them

Compare a(1) and a(2)
If a(1)<a(2)
Switch them
Compare a(2) and a(3)
If a(2)<a(3)
Switch them

Compare a(1) and a(2)
If a(1)<a(2)
Switch them

Output the sorted array

The iterations have been separated by spaces deliberately. Notice how each subsequent iteration has fewer steps than the previous, because the elements near the end are already sorted.

The lab exercise will include writing a bubble sort function. The problem is a little bit more difficult than it may appear. This is due to the tricks mentioned in the very first paragraph of this tutorial.

Here are a few hints for this program. The first hint is that nested loops are required for this program. Inside the inner *for* loop is an *if* statement to determine if switching is needed. The way this pseudocode is written, the index of the inner loop should actually decrease!

The second hint involves reassigning the value. As an example, consider the second step in the sorting of the array, where none of the values have been sorted. When the program finds that *a(2)* < *a(3)* or 4 < 8, it must switch them. The temptation is to write the following code inside the for loop:

```
a(ii)=a(ii+1);
a(ii+1)=a(i);
```

If this is carried out for the example, *a(2)* will be set equal to 8 and then *a(3)* will be set equal to the new value of *a(2)* which is 8. This is disastrous for the sorting algorithm. An alternative method is to use a place holder in the following way:

```
ph=a(ii); %ph – is the place holder
a(ii)=a(ii+1);
a(ii+1)=ph;
```

Now, the placeholder will hold the value of *a(2)*, so it is not lost when *a(2)* is reassigned.

These tips are minimalistic on purpose, so the student will think about how they can help if problems are found in writing this program. Also keep in mind that there are many ways to write this code, so the author cannot guarantee that these skills will be absolutely necessary.

The bubble sort has the advantage that it does not store two vectors to do the sorting. It is more difficult to program. In lab, students will be asked to clock the two methods and see which one is faster.

Random Number Generation

In writing and testing sorting algorithms, a time consuming step is simply entering arrays to be sorted. In most cases, it is a good idea to start with a vector of <10 elements and once that appears to work, try the algorithm on a larger array. But, who wants to type in all of those numbers?

A solution to this problem lies in the random number generator. (Certain people will argue that the numbers are not truly random. They are correct, but for this tutorial that is not important.) To generate an array of random numbers use the rand command like so:

```
>> a=rand(n,1);
```

The array *a* will have *n* elements in a row array. If the *n* and 1 are reversed, there will be *n* elements in a column array. If the 1 is left out, an *n x n* matrix of random numbers will be generated. The student is encouraged to play around with *rand* to see what else can be done. Also note that there are multiple random number generators in MATLAB with various uses. Using *rand* will save time in entering data if the user would like to sort an array of 100 elements or 1000 elements.

Finally, at times it is useful to clock sorting algorithms with very large arrays. Using *rand* will ease the pain when it comes to entering 1000 numbers, while leaving it possible to sort very large arrays.