**Sorting and Searching**

## 1 Sorting

When dealing with large data sets we often sort them for ease of use. MATLAB itself has a very powerful and well written sorting utility called `sort`. To develop our appreciation for the concept of sorting we will look at two basic sorting algorithms. MATLAB itself uses the second of these ideas.

To fix ideas let us consider the problem of sorting numerical data in an array. We say that an array `A` is sorted in ascending order if

$$\mathtt{A}(1) \leq \mathtt{A}(2) \leq \mathtt{A}(3) \leq \cdots \leq \mathtt{A}(\mathtt{n})\,, \tag{1}$$

while an array sorted in descending order will satisfy

$$\mathtt{A}(1) \geq \mathtt{A}(2) \geq \mathtt{A}(3) \geq \cdots \geq \mathtt{A}(\mathtt{n})\,. \tag{2}$$

Given an array with elements in any order, sorting is the process of rearranging the elements within the array to produce a sorted array. Sorting requires that the elements have an order. That is, the $<$ operator must be defined.

### 1.1 Bubble Sort

The bubble sort is a simple sorting algorithm that is relatively easy to understand. (It is also rather inefficient so it is not really of any great utility; nonetheless it is a good place to start when learning about sorting.) The bubble sort makes use of a pointer, which in this case, is an index. The pointer moves through the data array element by element. This pointer "catches" the largest element it has seen so far and drags it along. Thus the largest element "bubbles" through the other elements and eventually ends up at the right end of the array – in the position `A(j)`. The pointer then starts at the left again and bubbles through until it places the second largest element in the array at the right hand side in position `A(j-1)`. Continuing this way, the pointer sorts the array. Here is the code to perform a bubble sort:

```
function [A] = bubble(A)
% Usage: [A] = bubble(A)
% Purpose: Performs a bubble sort on array A
% Inputs: A -- array
% Output: A -- sorted version of input array

```

```
n=length(A);

for j = n-1 : -1 : 1
  for pointer = 1 : j
   if (A(pointer) > A(pointer+1))  % Swap enties if needed
     temp = A(pointer);
     A(pointer) = A(pointer+1);
     A(pointer+1) = temp;
   end
  end
end
end
```

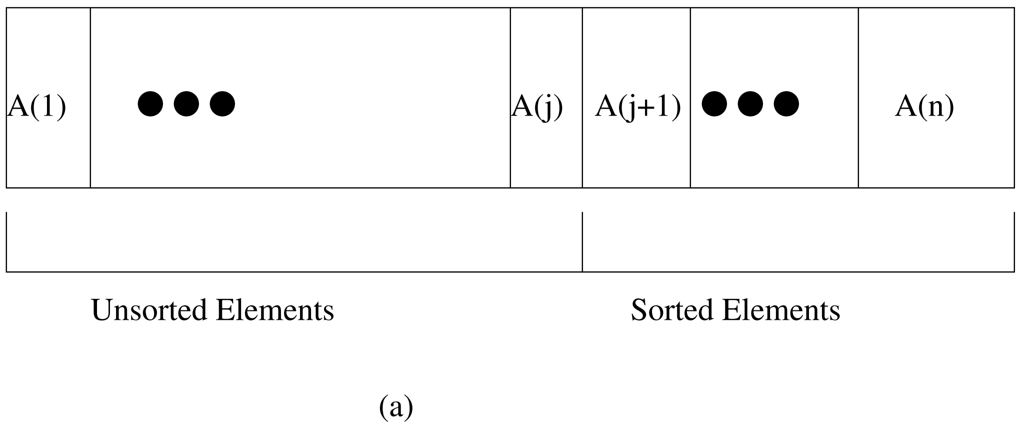
To understand this program, start with the inner for-loop. The "pointer" is (conveniently) named `pointer`. At the beginning of each execution of the inner loop, `A(pointer)` is larger or equal than `A(1),...,A(pointer-1)`. When `pointer = 1`, this statement is automatically true. `A(1)>=A(1)`. To go from `pointer` to `pointer+1`, we execute the inner loop once. That is, we compare `A(pointer)` with `A(pointer+1)`: If `A(pointer)` is smaller than or equal to `A(pointer+1)`, we do not swap the entries, because `A(pointer+1)` is already larger than `A(pointer)` and also `A(1),...,A(pointer-1)` by induction. On the other hand, if `A(pointer)` is larger than `A(pointer+1)`, we swap the entries. This ensures that after the swap `A(pointer+1)` is larger than `A(pointer)` and also `A(1),...,A(pointer-1)` (by induction). After the inner loop has completely finished executing, `A(j+1)` is larger than all of `A(1),...,A(j)`, because the last value of pointer in the loop was `pointer = j`. Now the first time through the outer loop, we have `j = n-1`. After the inner loop executes once, we will have `A(n)` larger or equal than `A(1),...,A(n-1)`. That means that the largest element in the array is now in the right-most position. That is where it belongs in the sorted array, so we leave it there. The next time through the outer loop, `j = n - 2`. This means the largest element from `A(1),...,A(n-1)` is found and placed in `A(n-1)`. That too is where it belongs. Continuing in this way shows that all the elements to the left of `A(n)` will eventually be sorted. We can summarize this with a picture as shown in Fig. 1.

When learning more complicated algorithms, this is the type of picture you should get used to drawing in order to better understand them. Rather than focusing on the code itself, a graphical description like this concentrates on what physically happens to the data. This makes for a simpler description, and it also makes it easier to see why the code works (or why it doesn't if it still has bugs).

Bubble sort unfortunately is rather slow and behaves rather poorly as the size of the array gets larger. The amount of work required to perform a bubble sort depends upon the number of comparisons that have to be performed. If the array to be sorted has length `n`, then from the code we see that on the first pass there are basically `n` comparisons. On the second pass, there are $n-1$, etc. down to 1. Thus the number of compares, i.e. the cost, is

$$\text{cost}(\mathtt{n}) = \sum_{k=1}^{\mathtt{n}} k = \frac{\mathtt{n}^2 + \mathtt{n}}{2}\,. \tag{3}$$

For large values of `n` the cost grows quadratically with `n` i.e. $\text{cost}(\mathtt{n}) = O(\mathtt{n}^2)$.



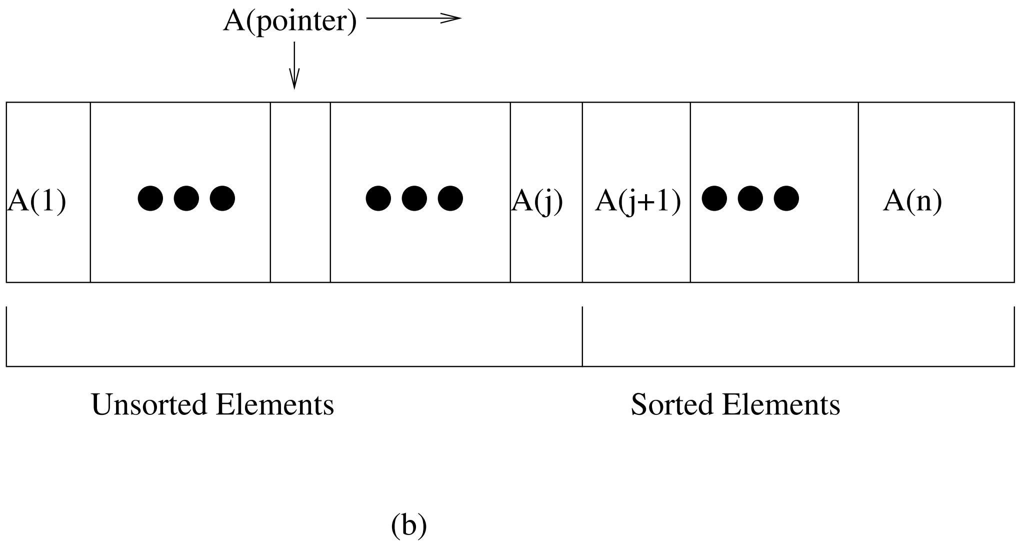


Figure 1: Bubble sort: (a) Current state of array for arbitrary value of `j`. (b) Current `pointer` position (`A(pointer) > A(k)` for all `k < pointer`) moving through array percolating the largest element up to position `n`.

## 1.2 Quick Sort

Quick sort is another sorting algorithm for arrays. As the name implies it is quick or at least usually it is quick. Amazingly, it is at its slowest when the array to be sorted is already sorted. This happens to be a minor point which can be gotten around. In fact, MATLAB's internal sort function is a quick sort.

The basic idea of quick sort is as follows.

1. Take an array `A` of length `n`. Choose a random element, say `A(q)`.
2. Rearrange the array so that
$$\mathtt{A}(\mathtt{1}:\mathtt{q}-\mathtt{1}) \leq \mathtt{A}(\mathtt{q}) \leq \mathtt{A}(\mathtt{q}+\mathtt{1}:\mathtt{n})\,. \tag{4}$$
3. Perform the same operation on the sub-arrays `A(1:q-1)` and `A(q+1:n)`.

As you can see this is a recursive algorithm. In fact it is a type of tree recursion. The cost of the algorithm can also be understood fairly easily. At each step of the algorithm, one needs to break the current array into two pieces; this partitioning process involves performing essentially $n$ compares, where $n$ is the length of the array that one is trying to partition. The number of times, $k$, we have to break the array up until we get to length 1 arrays is given by $n/2^k = 1$. Thus, $k = \log_2(n)$. So the total cost is
$$\text{cost}(n) = O(n\log_2(n))\,. \tag{5}$$
This estimate is much better than the cost of the bubble sort for large $n$. There are some further technicalities with this analysis; we have assumed that the sub-arrays are evenly sized but it is possible that they are not and this slows the algorithm down as the number of array splits can grow to be as large as $n$ which in turn can make the cost of the algorithm as high as $O(n^2)$ (i.e. just as bad as the bubble sort). Nonetheless, on average quick sort is very much faster and can be adjusted so that it does not fall into the worst case.

To program quick sort requires two routines: (1) the quicksort itself and (2) a routine to partition the array. The two can be combined but it makes for cleaner coding to separate them. The quicksort itself is given by:

```
function A = quicksort(A,left,right)
% Usage: A = quicksort(A,left,right)
% Purpose: Quicksort the sub—array A(left:right)
% Inputs:
%       A —— Array to be sorted
%       left —— left index of part to be sorted
%       right —— right index of part to be sorted
% Outputs:
%       A —— sorted between left and right
%
```

```
if left < right
   % split the array A(left:q-1) < A(q) < A(q+1:right)
   [A,q] = partition(A,left,right);
   A = quicksort(A,left,q-1);     % sort the left half
   A = quicksort(A,q+1,right);    % sort the right half
end
end
```

The real work happens in the partitioner. The version given here is the most straightforward version but it can have trouble if the array to be sorted is already sorted.

```
function [A,q] = partition(A,left,right)
% Usage: [A,q] = partition(A,left,right)
% Purpose: partitioner for quicksort
% Inputs:  A -- array for partitioning between A(left:right)
%          left -- left index
%          right -- right index
% Outputs: A,q -- rearranged such that A(left:q-1) < A(q) < A(q+1:right)

% choose the last term as the pivot
pivot = A(right);

% Current location for dividing terms < pivot
pointer_div = left-1;

% loop through the array finding elements less than pivot
% and placing them upfront
for pointer = left:(right-1)

      % check the element value against the pivot and place
      % in front part of array if less than or equal to pivot
      if A(pointer) <= pivot
            pointer_div = pointer_div + 1;
            tmp = A(pointer_div);
            A(pointer_div) = A(pointer);
            A(pointer) = tmp;
      end
end

% place pivot in the middle
tmp = A(right);
A(right) = A(pointer_div+1);
A(pointer_div+1) = tmp;

% return the pivot index
q = pointer_div+1;
end
```

To test the routine on can type something like:

```
>> A = rand(1,500); plot(A); hold on;
>> A = quicksort(A,1,500); plot(A)
```

The output of which will be a sorted `A` array and a plot of the unsorted and sorted arrays.

## 2 Searching

One of the main uses for sorted data is to search it for various values. As with sorting, MATLAB itself has a very powerful and well written sorting utility called `find`. To gain a better appreciation of the problem of searching we will examine one of the main methodologies for searching known as *binary search*.

### 2.1 Binary Search

The binary search algorithm is a simple and very fast way to find the index of a particular datum, or key, in the array once it has been sorted. A binary search uses a divide and conquer approach. First it divides the array in half. Then it looks to see if the key is in the left or right half (or directly on the division). Then it divides that half in half again, and that half in half, and so on. It is most naturally written as a recursive algorithm. Here is an example:

```
function indx = bsearch(A, key, left, right)
% Usage: indx = bsearch(A, key, left, right)
% Purpose: Recursive binary search of an array for a key
%
% Inputs:
%       A -- a sorted array
%       key -- desired value
%       left -- left index of subarray to search
%       right -- right index of subarray to search
%
% Output:
%       indx -- 'null' if key is not in array, else the index of the key

if left > right
        indx = 'null';
elseif (left == right)
        if (A(left) == key)
                indx = left;
        else
                indx = 'null';
        end
else
        mid = floor((left + right)/2);
        if A(mid) ==  key
                indx = mid;
        elseif A(mid) < key
                indx = bsearch(A,key,mid+1,right);
        else
                indx = bsearch(A,key,left,mid−1);
        end
end
```

Let us look at how this function works. First of all, left and right describe the limits of the subarray within which we will search. If they are equal, then we are looking at a subarray that consists of a single element. Either the element `A(left)=A(right)` is the element we are looking for, or there is no element matching the key. This test is taken care of by the first 8 lines of the function. In the general case where `left < right`, we are looking in an array with at least two elements. Binary search divides the array approximately in two by computing a midpoint, which is `mid = floor((left + right)/2)`. Then it compares the element `A(mid)` with the key. If they are equal, then it returns `mid` as the index. If not it decides if the key is to be found to the right or the left of `mid`. Since the array is sorted, all the elements to the left of `A(mid)` are less than or equal to `A(mid)`, while all the elements to the right of `A(mid)` are greater than or equal to `A(mid)`. So if `key` is larger than `A(mid)`, it must be to the right of `A(mid)`, if it is in the array at all. That is, it must be in the subarray from `A(mid+1)` to `A(right)`. The recursive call to `bsearch` looks for it there. If `key` is less than `A(mid)`, then it must be in the subarray from `A(left)` to `A(mid-1)`. The other recursive call looks for it there. The binary search will correctly choose the correct half to search in the recursive call. Note we also check for the exceptional case of `left` > `right` and return `'null'`.

To show that the algorithm works, we need to make sure that it stops. Hopefully this is self-evident. It should also be clear that if the array size, `(right-left+1)`, is large enough, then `A(mid)` will be a good midpoint, and the recursive calls will work on arrays that are only about half as big as the input array. Thus the cost of the algorithm is roughly $\text{cost}(n) = O(\log_2(n))$ as this is the number of divides needed to get down to an array of length 1 where the algorithm terminates.

## 2.2 Binary Search Trees

Another useful data structure for performing searches is a binary search tree (BST). In a binary search tree the data is not stored in a sorted array as in the last example. It is stored in a tree data structure. At each node of the tree we store the data. Each node has a left and right (child). The children to the left all have keys which are less than the parent and those on the right all have keys that are larger than the parent. Figure 2 shows an example of a valid BST.

The actual BST is stored in a structure array with a root index. A possible way of creating the BST for the example would be to explicitly type:

```
>> bst.root = 1;
>> bst.node(1).key = 10; bst.node(1).left = 2; bst.node(1).right = 3;
```

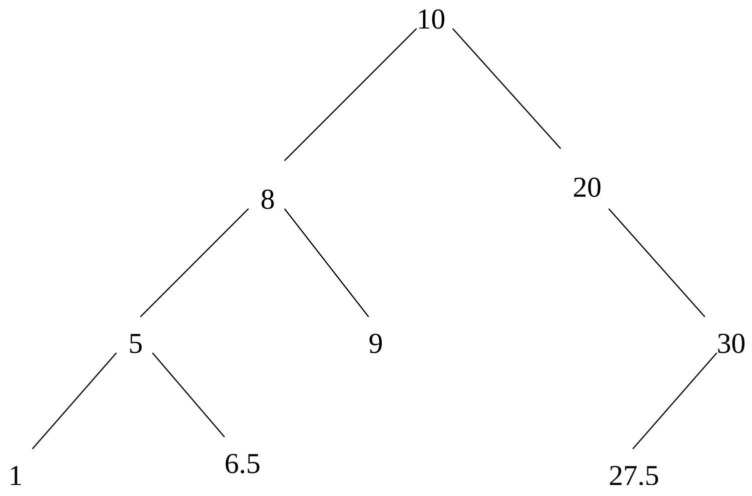


Figure 2: Valid binary search tree.

```
>> bst.node(2).key = 8 ; bst.node(2).left = 4; bst.node(2).right = 5;
>> bst.node(3).key = 20; bst.node(3).left = 0; bst.node(3).right = 6;
>> bst.node(4).key = 5 ; bst.node(4).left = 7; bst.node(4).right = 8;
>> bst.node(5).key = 9 ; bst.node(5).left = 0; bst.node(5).right = 0;
>> bst.node(6).key = 30; bst.node(6).left = 9; bst.node(6).right = 0;
>> bst.node(7).key = 1 ; bst.node(7).left = 0; bst.node(7).right = 0;
>> bst.node(8).key = 6.5; bst.node(8).left = 0; bst.node(8).right = 0;
>> bst.node(9).key = 27.5; bst.node(9).left = 0; bst.node(9).right = 0;
```

This is of course a rather painful way of setting up such a structure. A more automated way is to have an insert function that will take an arbitrary key and insert it into the BST in a proper position. Such a routine can be easily written using recursion. Shown below is such a routine which uses the helper functions `addleaf` and `countnodes`. The overall scheme is to recursively descend the tree looking for a node upon which the key can be attached. Being able to attach to a node requires that a node not have a child on the correct side. Once such a node is found a leaf is added with the `addleaf` function. `addleaf` works by first counting the number of nodes in the BST using the `countnodes` function and then it adds the new leaf in the data position one beyond the current end. `countnodes` itself works by recursively descending the BST and counting the nodes. The routines presented are simple versions and only work when every node in the tree data structure is referenced.

To use the routines to set up our example BST one could type

```
>> bst.node(1).key   = 10;
>> bst.node(1).left  = 0;
>> bst.node(1).right = 0;
>> bst.root = 1;
>> bst = binsert(bst,bst.root,8);
>> bst = binsert(bst,bst.root,5);
>> bst = binsert(bst,bst.root,1);
>> bst = binsert(bst,bst.root,9);
>> bst = binsert(bst,bst.root,6.5);
>> bst = binsert(bst,bst.root,20);
>> bst = binsert(bst,bst.root,30);
>> bst = binsert(bst,bst.root,27.5);
```

The order of entry matters not. The routines take care of everything. Note that the resulting tree structure will not be exactly the same as the one we entered manual but will represent the same sorted data. Likewise if we change the order of the `binsert` calls we will represent the same sorted data but the layout of the data in the tree will be different.

```
function [bst] = binsert(bst, idx, key)
% Usage: [bst] = binsert(bst, idx, key)
%
% Purpose: Insert a key into a subtree of a BST which starts at index == idx
%
% Inputs: bst     -- structure array containing the BST
%         idx     -- index of subtree to receive the key
%         key     -- key value to be inserted
%
% Outputs: bst    -- the new BST

% Copy the current node for convenience
thisnode = bst.node(idx);

% If key is greater than current key add to right subtree
if key > thisnode.key
         % If the right subtree is empty add here else descend recursively
         if thisnode.right == 0
                 bst = addleaf(bst, idx, 'r', key);
         else
                 bst = binsert(bst, thisnode.right, key);
         end
% If key is less than current key add to left subtree
elseif key < thisnode.key
         % If the left subtree is empty add here else descend recursively
         if thisnode.left == 0
                 bst = addleaf(bst, idx, 'l', key);
         else
                 bst = binsert(bst, thisnode.left, key);
         end
% If key is present
else
         % Print message; do nothing since key is already present
         disp('Key already present');
end
end
```

```
function [bst] = addleaf(bst, idx, child, key)
% Usage: [bst] = addleaf(bst, idx, child, key)
%
%  Purpose: Add a leaf to a BST with given key as a left or right child of a given node
%
% Inputs: bst     -- structure array containing the BST
%         idx     -- index of node to receive the new leaf
%         child   -- string indicating a new left or right child
%         key     -- key value of the new leaf
%
% Outputs: bst  -- the new BST

% Count the total number of nodes
numnodes = countnodes(bst,bst.root);

% New node will go at end
newnode = numnodes + 1;

% Set up data for new node which will be a leaf
bst.node(newnode).key   = key;
bst.node(newnode).right = 0;
bst.node(newnode).left  = 0;

% Set the node at idx to point to the new node
if child == 'r'
         bst.node(idx).right = newnode;
elseif child == 'l'
         bst.node(idx).left = newnode;
else
         disp('Error new leaf needs to be either a left or right child');
end
end
```

```
function [cnt] = countnodes(bst,idx)
% Usage: [cnt] = countnodes(bst,idx)
%
% Purpose: Count the number of nodes in a BST starting at the subtree with
%          index == idx
%
% Inputs: bst -- structure array with the BST
%         idx -- index of the subtree to be counted
%
% Outputs: cnt  -- number of nodes in the subtree beginning at idx

% Base case == empty tree
if idx == 0
         cnt = 0;
% Recursively count the left and right subtrees and add to current node
else
         cnt = countnodes(bst,bst.node(idx).left) + ...
```

```
               1 + ...
               countnodes(bst,bst.node(idx).right);
end
```

The issue of finding keys in a BST can be easily handled recursively as is shown below in the `bfind` routine. The basic idea is to look to see what side of the current node the key must lie on (if not equal to the current node's key). Then to search down that side of the tree. The routine is quite efficient. To use the routine one would for example type

```
>> bfind(bst,bst.root,6.5)
```

The return value will be the index in `bst` which corresponds to the data value of 6.5.

```
function [v] = bfind(bst,idx,key)
% Usage: [v] = bfind(bst,idx,key)
%
% Purpose: Find the index of a key in a BST starting with the subtree
%          located at index == idx
%
% Inputs: bst -- structure array with the BST
%         idx -- index of the start of the subtree to search
%         key -- value to search for in the subtree
%
% Outputs: v  -- index of the key in the BST ('null' if not found)

if idx == 0
         % Return null if not found
         v = 'null';
else
         % If key is found
         if bst.node(idx).key == key
                 v = idx;
         % If key is in the right subtree search recursively
         elseif bst.node(idx).key < key
                 v = bfind(bst, bst.node(idx).right, key);
         % If key is in the left subtree search recursively
         else
                 v = bfind(bst, bst.node(idx).left, key);
         end
end
end
```