Sorting in MATLAB

This is outside of textbook (except for SelectionSort/p216-221)

Disclaimer: *These abbreviated notes DO NOT substitute the textbook for this class. They should be used IN CONJUNCTION with the textbook and the material presented in class. If there is a discrepancy between these notes and the textbook, ALWAYS consider the textbook to be correct. Report such a discrepancy to the instructor so that he resolves it. These notes are only distributed to the students taking this class with A. Gerbessiotis in Fall 2011 ; distribution outside this group of students is NOT allowed.*

(c) Copyright A. Gerbessiotis. CS 101-005 : Fall 2011 . All rights reserved.

S1. Sorting n keys.

The sorting problem can be defined the following way.

Input. A sequence of n keys $\langle a_1, a_2, \ldots, a_n \rangle$.

Output. A reordering (i.e. permutation) $\langle a_{j_1}, a_{j_2}, \ldots, a_{j_n} \rangle$ of the input sequence so that $a_{j_1} \leq a_{j_2} \leq \ldots \leq a_{j_n}$.

Example If the input is $\langle 30, 20, 40, 10 \rangle$, i.e. $a_1 = 30, a_2 = 20, a_3 = 40, a_4 = 10$, then the output is the permutation $\langle a_4, a_2, a_1, a_3 \rangle$ because then $a_4 = 10 \leq a_2 = 20 \leq a_1 = 30 \leq a_3 = 40$.

Note that the output for the problem of sorting is in what we call in Mathematics non-decreasing or monotonically increasing order i.e. one that looks like $\leq, \leq, \ldots, \leq$. If the keys were distinct we could have said that the output keys were in increasing order i.e. $<, <, \ldots, <$: this is also called the **sorted** order, as implied by the output for the problem of sorting.

If the keys were in non-increasing or monotonically decreasing order (ie. $\geq, \geq, \ldots, \geq$) we would have said that the input was in **reverse** sorted order instead.

Sorting in-place. We say that a sorting algorithm sorts **in-place** if the same space (eg array,matrix,vector) is used for input and output and the extra memory used by the algorithm is constant.

Stable sorting algorithm. A sorting algorithm is **stable** if the relative order of same valued keys remains the same in the input and output sequence. Thus if the input is $\langle 1_1, 1_2 \rangle$ (in this input the key value is the 1 and the subscripts $_1, _2$ are just to tell us which one is which) the output remains the same $\langle 1_1, 1_2 \rangle$ for a stable sorting algorithm and does not become $\langle 1_2, 1_1 \rangle$ since in the latter case the relative order of the ones would have changed!

Counting operations and comparing sorting algorithms. For a sorting algorithm, the most important operation is that of comparing key values. Although in MATLAB-based sorting most comparisons involve integers or numbers in general, it is also the case that other key values can be compared (e.g. strings). Thus it is important to consider how many comparisons a sorting algorithm requires.

(c) Copyright A. Gerbessiotis. All rights reserved.

```
1  % Find minimum of n keys A(1) ... A(n)
2  % key    is the minimum element
3  % index is the location in A storing the min key
4  % i.e. A[index]=key
5  function [key index] =findmin(A)
6    n=length(A);
7    index=1;
8    key=A(1);
9    for jj= 2:n
10     if A(jj) < key
11       index=jj;
12       key  =A(jj);
13     end
14   end
15 end
```

SelectionSort. The `SelectionSort` algorithm is a instance of an `incremental` algorithm: the output is formed one element at a time. Another incremental algorithm that we have seen is the one for finding the n-th Fibonacci number. There are two variants of SelectionSort: (a) the MIN variant that involves finding the MINimum of the remaining keys to be sorted, and (b) the MAX variant that involves finding the MAXimum of the remaining keys.

Idea of SelectionSort. Starting with n keys, perform i iterations, $i = 1, \ldots, n$, of the following algorithm: in the i-th iteration, find the minimum of the $n - i + 1$ remaining keys in that iteration, thus filling the i-th left-most position of the output sequence and then discarding this key from consideration in subsequent iterations.

Thus in iteration 1 of SelectionSort the MINimum of n keys needs to be found; this requires $n-1$ comparisons. In the following iteration 2, the minimum of the remaining $n-1$ keys needs to be found by performing $n-2$ comparisons. Thus in iteration i we need to perform $n-i$ comparisons to determine the minimum of the $n-i+1$ remaining keys for that iteration. The total number of comparisons performed is thus $(n-1)+(n-2)+\ldots+2+1$ which is $n(n-1)/2$, i.e quadratic to the number of keys.

We can find the MINimum of n keys in $n-1$ comparisons by algorithm `findmin` shown at the top. Initially we consider as a candidate for a minimum the first key $A(1)$. We then search each one of the remaining keys to determine whether the previously considered candidate is still the MINimum or not; If a new minimum is located (line 10) we update accordingly.

(c) Copyright A. Gerbessiotis. All rights reserved.

S2. SelectionSort : An example

```
Input  < 40 10 20 30 50 >          Output :   <               >       Separate input+output
Iteration 1:                                     1  2  3  4  5        (preliminary version)
Input  < 40 10 20 30 50 >          Output :   < 10            >       for SelectionSort
           ^^
Iteration 2:                                     1  2  3  4  5
Input  < 40 20 30 50    >          Output :   < 10 20         >
           ^^
Iteration 3:                                     1  2  3  4  5
Input  < 40 30 50    >             Output :   < 10 20 30      >
           ^^
Iteration 4:                                     1  2  3  4  5
Input  < 40  50  >                 Output :   < 10 20 30 40   >
         ^^
Iteration 5:                                     1  2  3  4  5
Input  < 50  >                     Output :   < 10 20 30 40 50 >
         ^^
```

SelectionSort is an in-place sorting algorithm. We need to keep track in each iteration the output position that we will be filling (at iteration i, it is i), and the index of the input array storing the minimum key. At iteration i the first $i-1$ positions are considered output and the remaining $i, i+1, \ldots, n$ positions are still considered part of the input. Thus the same array is both input and output, and at every iteration input size decreases by one and output size increases by one.

```
                  Selection Sort example (in-place final version)
Input  <|40 10 20 30 50 >        -- : Shows the output position to be filled at iteration i
Iteration 1:                     ^^ : Shows the location  of the minimum    at iteration i
Input  <|40 10 20 30 50 >             Swap -- and ^^ keys to complete          iteration i
         -- ^^
Iteration 2:                      | : On the left of | the keys are sorted and considered
Input  < 10|40 20 30 50 >             part of the output. The key immediately to the right
            -- ^^                      of | is the key considered in that iteration. At the
Iteration 3:                          end of the iteration the | will move to the right one
Input  < 10 20|40 30 50 >             position
               -- ^^
Iteration 4:
Input  < 10 20 30|40 50 >          ^^ and -- point to the same position: no swap needed!
                  --
                  ^^                  No need for iteration 5; there is only one
                                      key left; and it's then minimum of itself
```

(c) Copyright A. Gerbessiotis. All rights reserved.

S2. SelectionSort : MATLAB implementation

```
1  function A=selsort(A)
2    n=length(A);
3    for ii=1:(n-1)
4       index=ii;
5       key = A(ii);
6       for jj = ii+1:n
7         if A(jj) < key
8           index= jj;
9           key=A(jj);
10        end
11      end
12      if (index ~= ii)
13          A(index)=A(ii);
14          A(ii)=key;
15      end
16    end
17 end
```

The implementation shown for function `selsort` is an optimized form of the textbook code shown in pages 216-221. We use `ii` indexing instead of `i` used in the discussion of the previous two pages. Thus, iteration `i` becomes iteration `ii`.

Lines 4,5, and 6-10 are reminiscent of the `findmin` code. The minimum of the keys $A(\texttt{ii}), \ldots, A(\texttt{n})$ is to be found at iteration `ii`.

In line 12 we determine whether there is a need to swap the keys in position `index` and `ii`. If the two positions are the same no swapping takes place, otherwise we swap $A(\texttt{index})$ and $A(\texttt{ii})$. Note that we do not need a `temp` variable for the swap that the textbook is using, because the role of `temp` is played by `key` and `key` already stores the value of $A(\texttt{index})$.

That line 7 has a $<$ rather than $\leq$ is what makes `SelectionSort` a stable sorting algorithm. For example if the input was two ones i.e. $\langle 1, 1\rangle$ to be shown as $\langle 1_1, 1_2\rangle$ so that we can talk about the first vs second, line 7 as is determines that the minimum is 1_1 and not 1_2. Thus the output will remain $\langle 1_1, 1_2\rangle$ and this makes `selsort` stable. Had the output been $\langle 1_2, 1_1\rangle$, say because we had determined that the minimum is the second 1 (and this could have resulted by the use of a $\leq$ in line 7), the relative order of same valued keys would have been changed from the input $\langle 1_1, 1_2\rangle$ to the newly formed unstable output $\langle 1_2, 1_1\rangle$. The $<$ in line 7 guarantees that this would never occur!

(c) Copyright A. Gerbessiotis. All rights reserved.

InsertionSort. The `InsertionSort` algorithm is a form of an `incremental` algorithm. The output is formed one element at a time similarly but differently from `SelectionSort`. `InsertionSort` considers each input key in left-to-right order one element at a time, and **inserts** that key in its proper (correct) position in the forming output sequence.

The first iteration of `InsertionSort` is trivial and can be ignored (not run) by going directly to the second iteration. The reason for this is that the first key needs to be inserted into an originally empty sequence. This step is straightforward: nothing needs to be done.

Idea of InsertionSort. Starting with n keys $\langle a_1, a_2, \ldots, a_n \rangle$, , perform i iterations, $i = 1, \ldots, n$, of the following algorithm: in the i-th iteration, consider key a_i and insert it into the output sequence already formed by the sorting of the previous $a_1, a_2, \ldots, a_{i-1}$ input keys.

Thus in iteration $i = 2, 3, \ldots, n$, if the $i-1$ input keys $a_1, a_2, \ldots, a_{i-1}$ are sorted so that their sorted order is $a_{j_1}, a_{j_2}, \ldots, a_{j_{i-1}}$, where $j_1, j_2, \ldots, j_{i-1}$ is a reordering (called permutation) of $1, 2, \ldots, i-1$, what we are interested in doing is inserting the i-th key a_i into the sorted sequence

$$\langle a_{j_1}, \quad a_{j_2}, \quad \ldots, \quad a_{j_{i-1}} \rangle$$

For that we scan the already formed output right to left starting with $a_{j_{i-1}}$. If a_i is greater than or equal to $a_{j_{i-1}}$, we are done, the newly considered input key a_i should appear after the previously considered $i-1$ keys and thus the output would become $\langle a_{j_1}, \quad a_{j_2}, \quad \ldots, \quad a_{j_{i-1}}, a_i \rangle$. If however, $a_i < a_{j_{i-1}}$, then we go on comparing a_i to the key to the left of $a_{j_{i-1}}$, that is $a_{j_{i-2}}$ and go on doing so until we find the **insertion** position for a_i. Because we are to determine for every key a_i its `insertion position` in the output formed by $a_1, \ldots, a_{i-1}$, the algorithm became known as `InsertionSort`.

InsertionSort is both stable and sorts in-place. The example of the following case will convince us of the latter. The proper structure and introduction of a $<$ will convince us of the former. The $<$ relates to the following question: `Is` a_i `less than` $a_{j_{i-1}}$ `or not`? If it is so then and only then do we move a_i further to the left in the output sequence. If a_i is equal to $a_{j_{i-1}}$, a_i will remain to the right of $a_{j_{i-1}}$ i.e. their relative order will be maintained. Consider the following case where a key can be a number or a name and keys are first sorted based on name and then sorted sorted based on number; in a stable algorithm the previous order is still available.

```
Input:
 <<2,Pat>,<2,Tim>,<2,Tom>,<1,Vic>>        <<2,Pat>,<2,Tim>,<2,Tom>,<1,Vic>>
      Stable Algorith gives                  Non-stable algorithm may give
 <<1,Vic>,<2,Pat>,<2,Tim>,<2,Tom>>        <<1,Vic>,<2,Tom>,<2,Tim>,<2,Pat>>
```

```
Iteration 1:                                    1  2  3  4  5
Input  < 40 10 20 30 50 >          Output :   <  40            >    Iteration 1 can be skipped!
        ---------------------------------------->
Iteration 2:
Input  <    10 20 30 50 >          Output :   < 10 40          >
            ---------------------------------->
Iteration 3:
Input  <       20 30 50 >          Output :   < 10 20 40       >
               ------------------------------------>
Iteration 4:
Input  <          30 50 >          Output :   < 10 20 30 40    >
                  --------------------------------->
Iteration 5:
Input  <             50 >          Output :   < 10 20 30 40 50 >
                     ------------------------------------>
```

Although it seems that `InsertionSort` is not an in-place algorithm, this is how its in-place form looks like.

```
Input  <|40 10 20 30 50 >        ^^ : Shows the key  considered          in  iteration  i
                                 -- : Show  the keys swapped with the key of  iteration i
Iteration 1:                      | : Separates input from output
Input  < 40|10 20 30 50 >             On the left of | keys are sorted on the right of |
         ^^                           keys have not been considered yet
Iteration 2:                          Input is shown at then end of iteration i.
Input  < 10 40|20 30 50 >            1st swap : 40 and 10 get swapped
         ^^ --
Iteration 3:
Input  < 10 20|40|30 50 >             2nd swap : 20 and 40 get swapped.
            ^^ --
Iteration 4:
Input  < 10 20 30 40|50 >             3rd swap : 30 and 40 get swapped.
               ^^ --
Iteration 5:
Input  < 10 20 30 40 50|>
                  ^^
```

(c) Copyright A. Gerbessiotis. All rights reserved.

S3. InsertionSort : MATLAB implementation

```
1  function A=inssort(A)
2    n=length(A);
3    for ii= 2:n
4      key=A(ii);
5      jj=ii-1;
6      while ((jj>0) && (A(jj) > key))
7          A(jj+1) =A(jj);
8          jj=jj-1;
9      end
10     A(jj+1)=key;
11   end
12 end
```

The implementation shown for function `inssort` is optimized in the sense that most of the swaps related to a_i (shown as $A($`ii`$)$) are half-swaps except for the final insertion position that is a full swap. The optimization is the same as that of SelectionSort. The use of `key` eliminates the use of a `temp` variable that the textbook code used for `SelSort`. Using the same idea, we store a_i (or $A($`ii`$)$ in MATLAB) into `key`, and when we are to swap a_i (i.e.$A($`ii`$)$) with say $a_{j_{i-1}}$ (this is $A($`jj`$)$ since `jj` $=$ `ii` $-$ `1`), we only move $A($`jj`$)$ to the right but leave $A($`ii`$)$ in `key`.

Lines 3-11 consider each key other than the first one in turn. The i-th key becomes `ii`-th in MATLAB and is stored in variable `key`. This key is to be inserted into the sorted sequence consisting of keys $A(1), \ldots, A($`ii` $-$ `1`$)$. The rightmost position is thus $ii-1$, and we record it in `jj` in line 5.

Lines 6-10, consider for key `key` (i.e. $A($`ii`$)$) where it is to be inserted in the output formed by the previous `ii` $-$ `1` keys. Key `key` moves leftwards as long as `key` is less than $A($`jj`$)$ for decreasing values of `jj`. This is equivalent to checking whether $A($`jj`$) >$ `key` which is the second expression of the `while` statement in line 6. The condition `jj` > 0 checks that `key` does not try to insert itself into the left of the left-most position.

If $A($`jj`$) >$ `key` is indeed so, `key` needs to get swapped with $A($`jj`$)$ i.e. $A($`jj`$)$ need to move to $AA($`jj` $+$ `1`$)$ and perhaps `key` to $A($`jj`$)$. The former is done in line 7, the latter is delayed as much as possible. We continue this by checking `key` against the key to the left of the key at position `jj` i.e. the one at position `jj` $-$ `1`. This is the effect of line 8.

As soon as we decide that `key` is to move indeed to position $A(($`jj`$)$, we do so in line 10. Note that in line 8, `jj` was decremented by one, and we thus need to use `jj+1` in line 10 to effect this.

(c) Copyright A. Gerbessiotis. All rights reserved.

```
1. function A=inssort(A)                          Key comparisons      Swaps/Half-swaps
2.   n=length(A);                                  (total)              (total)
3.   for ii= 2:n                                   -                    -
4.     key=A(ii);                                  -                    -
5.     jj=ii-1;                                    -                    -
6.     while ((jj>0) && (A(jj) > key))             t2+t3+...+tn         -
7.        A(jj+1) =A(jj);                          -                    (t2-1)+(t3-1)+...+(tn-1)
8.        jj=jj-1;                                 -                    -
9.     end                                         -                    -
10.    A(jj+1)=key;                                -                    n-1
11.  end                                           -                    -
12. end                                            -                    -
```

We note that in `InsertionSort` line 6 is executed a variable number of times for key `key` (which is key $A($(`ii`)$)$ based on the line 4 assignment). Thus for key `A(2)` let us call this number t_2, for key `A(3)` let us call it t_3 and so on. If line 6 is executed t times, the body of the while loop is executed once less (or in other words the while loop is executed as many times as the body plus once more to determine that it's time to exit the while loop and move to line 10). This determines how many times the key comparison is performed in line 6 and also the half-swap in line 7. The other half of the swap in line 10 is being performed once per iteration i.e. few times.

Thus the running time $T(n)$ of algorithm `inssort` as depicted is

$$T(n) = (t_2 + t_3 + \ldots + t_n) + ((t_2 - 1) + (t_3 - 1) + \ldots + (t_n - 1)) + n - 1$$

The $n-1$ get cancelled out with the $n-1$ -1s. Thus we get that

$$T(n) = 2\,(t_2 + t_3 + \ldots + t_n) = 2\sum_{i=2}^{n} t_i$$

How big or how small can t_i be? The smallest value for t_i is one, i.e. line 6 is executed only once to determine that $A($`ii`$) >$ `key` is false i.e. `key` is greater than or equal to $A($`ii`$)$.

The largest value of t_i is i, which means that $A($`ii`$)$ is larger than the previous `jj=ii-1` keys.

Thus we distinguish the following cases

(c) Copyright A. Gerbessiotis. All rights reserved.

A. Best case of `InsertionSort`. This is the case for which $t_i = 1$ for all values of i. This is the case where the second key is larger than the first, the third is larger than the second, the fourth larger than the third, and so on, i.e. the input is already sorted. Then

$$T(n) = 2\left(t_2 + t_3 + \ldots + t_n\right) = 2\sum_{i=2}^{n} t_i = 2\sum_{i=2}^{n} 1 = 2(n-1)$$

We then conclude that the running time $T(n)$ is a linear function of n for the best case when the input is already sorted.

B. Worst case of `InsertionSort`. This is the case for which $t_i = i$ for all values of i. This is the case where the second key is smaller than the first, the third is smaller than the second and first, the fourth smaller than the third, second and first, and so on, i.e. the input is reverse sorted. Then

$$T(n) = 2\left(t_2 + t_3 + \ldots + t_n\right) = 2\sum_{i=2}^{n} t_i = 2\sum_{i=2}^{n} i = 2\left(2 + 3 + \ldots + (n-1)\right) = -2 + 2\left(1 + 2 + 3 + \ldots + (n-1)\right) = -2 + (n-1)n$$

Here we used the fact from Problem Set 3 and also the analysis of `SelectionSort` that $1 + 2 + 3 + \ldots + n = n(n+1)/2$ or equivalently that $1 + 2 + 3 + \ldots + (n-1) = n(n-1)/2$. The conclusion is that $T(n)$ is quadratic in n, the number of keys.

Every other case of input has t_i that fluctuates between 1 (best) and i (worst). Thus the running time of `InsertionSort` is at worst the worst-case running time which is quadratic.

Thus `InsertionSort` exhibits the quadratic performance of `SelectionSort.`

Are there sorting algorithms that have worst case performance less than quadratic? The answer is yes and `MergeSort` is such an algorithm (and `HeapSort` is another one). Both of these algorithm have log-linear performance in the worst-case i.e. their running time is at most $n \lg n$ where $\lg n$ is the base two logarithm of n.

(c) Copyright A. Gerbessiotis. All rights reserved.

S5. MergeSort : Introduction

MergeSort. The `MergeSort` algorithm is a form of a `divide-and-conquer` algorithm. The elegance of a `divide-and-conquer` approach was exhibited in the context of the Fibonacci sequence. The pitfalls of such an approach were also made apparent. The case of a `divide-and-conquer` approach to `MergeSort` is more involved. In the `Divide` phase we split the problem into two subproblems of about the same size $n/2$ (plus or minus one). This division is obvious if $A(1:n)$ is the array of n keys one subproblems consists of $A(1:m)$ and the other of $A(m+1:n)$, where $m = \lfloor (1+n)/2 \rfloor$, with $\lfloor . \rfloor$ representing the MATLAB `floor` function. We then run `MergeSort` recursively in the two subproblems (also known as halves). After each recursive call each of the two subproblems is available in sorted order. We thus need to merge two sorted sequences of about the same size. The problem of merging is easier than the problem of sorting: the merging problem involves two already sorted sequences!

If we have two sorted sequence of s and t number of keys we can merge them quickly by performing $s+t-1$ comparisons in the worst-case. The method is quite simple. We make the keys become players participating in a tournament of games with one player from one sequence and another from the other sequence. Players from a given sequence participate left-to-right i.e. we pick the one with the smallest key value first. Given that the two sequences are sorted this is implied by the left-to-right ordering. A game is won by the player with the lower key value; the winner fills the next position in the output; the output is also filled left-to-right in a smaller-to-larger key fashion. If a game cannot be played because one of the two sequences runs out of players (keys) the other team's player wins by default and goes to the output.

Figure 1 of the following page shows this process.

In the example of Figure 1, the first games was played by `30` and `10` and won by `10`. The next game was played by `30` (again) and the next player of the second sequence i.e. `20` and won by `20`. The last game played was between `70` and `80` and `70` won. There were no games left thus `80,90` went into the output with out playing any more games; if fact `90` never played a game.

The number of games played is the number of comparisons performed; in our case this was 7 for the first seven keys.

(c) Copyright A. Gerbessiotis. All rights reserved.

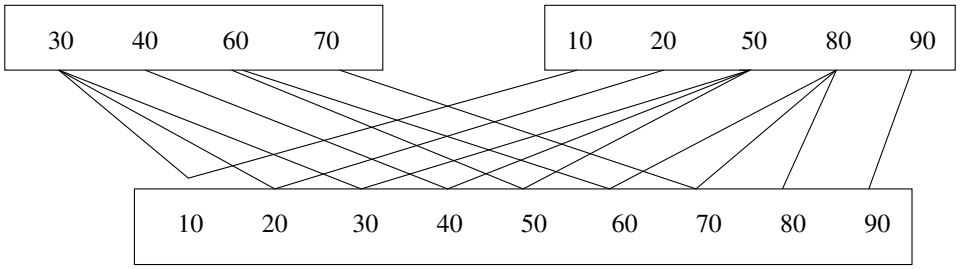


Figure 1: Merging two sorted sequence of length $s = 4$ and $t = 5$ respectively

Implementing `Merge`. For an implementation of `Merge` we need two pointers/indexers running through the left and the right sequences. Let them be `S, T` respectively. Both are initially pointing to the first element of each sequence. Thus `S=1, T=1`. Let the sizes of the left and right sequences be `s and t` respectively. In addition we keep track a pointer/indexer `P` that is used to fill the output. Initially `P=1`. At any iteration we compare the element pointed by `S, T` respectively and let them be L(S) and R(T), if the left array is `L(1:s)` and the right one `R(1:t)`. Let the output array be `A(1:s+t)` (and of length $s + t$ to accommodate all elements of `L,R`). We have four cases (including two pathological cases) to consider.

Case 1. `L(S) <= R(T)`. Key `L(S)` fills the output i.e. `A(P)=L(S)`. We prepare for the next iteration by incrementing `P,S` i.e. `P=P+1, S=S+1`.

Case 2. `L(S) > R(T)`. Key `R(T)` fills the output i.e. `A(P)=R(T)`. We prepare for the next iteration by incrementing `P,T` i.e. `P=P+1, T=T+1`.

Case 3. `S > s` i.e. we run out of elements of `L`. In this case we fill `A(P:s+t)` with `R(T:t)`.

Case 4. `T > t` i.e. we run out of elements of `R`. In this case we fill `A(P:s+t)` with `L(S:s)`.

(c) Copyright A. Gerbessiotis. All rights reserved.

S5. MergeSort : Implementation

A recursive (but not optimized) implementation of `MergeSort` is `rmergesort.m`

```
1 function A=rmergesort(A)
2   n=length(A);
3   if (n>1)
4     m=floor((1+n)/2);
5     L=rmergesort(A(1:m));
6     R=rmergesort(A(m+1:n));
7     A=merge(L,R);
8   end
9 end
```

Function `rmergesort` is a wrapper function. In `rmergesort` we only perform some work for `n >1`. We split the input array A into two halves as stated earlier `A(1:m)` and `A(m+1:n)`, where $r = \lfloor (1+n)/2 \rfloor$. The result of `MergeSort` of the first half is `L` and that of the second half is `R`. Thus `L,R` are sorted when line 7 is executed. The merging of line 7 returns a new array containing the merging of `L,R` back into A. And this new A is returned to the calling program. (We thus reuse memory in this last step.)

(c) Copyright A. Gerbessiotis. All rights reserved.

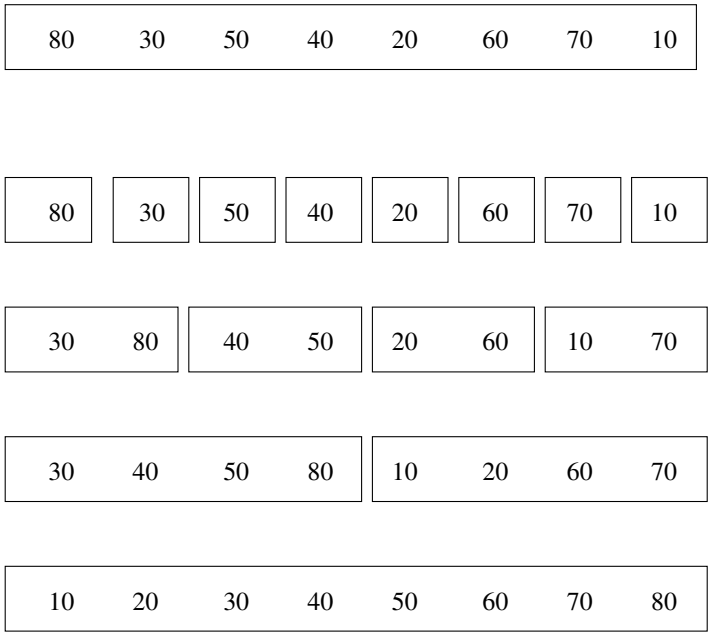
S5. MergeSort : Implementation continued

The only missing item is the `merge.m` function shown below. The code is self-explanatory following the discussion related to Figure 1.

```
function A=merge(L,R)
  s=length(L);        % length of left  sequene L is s
  t=length(R);        % length of right sequence R is t
  S=1;                % Runs through L
  T=1;                % Runs through R
  A=zeros(1,s+t);     % Merging is not in-place

  for  P= 1:(s+t)     % Build output one element at a time
    if  S >  s                                % This is Case 3
      A(P) = R(T); T=T+1;          % copy R(T:t) to A...
    elseif  T > t                             % This is Case 4
          A(P) = L(S); S=S+1;  % copy L(S:s) to A
    elseif L(S) <= R(T)                       % This is Case 1
          A(P)  = L(S); S=S+1; % copy element
    else                                      % This is Case 2
          A(P)  = R(T); T=T+1; % copy element
    end
  end
end
```

(c) Copyright A. Gerbessiotis. All rights reserved.



STEP 1	INPUT n=8
STEP 2:	Divide phase (split into smaller and smaller sequences)
STEP 2a	Base case is a sequence of size 1
STEP 3.1	Start conquering (i.e. merging sequences of size 1 into sequences of size 2)
STEP 3.2	Continue merging sequences of size 2 into sequences of size 4
STEP 3.3	Continue merging sequences of size 4 into sequence(s) of size 8 until you get only one sequence, the OUTPUT

Figure 2: MergeSort: working out bottom up after splitting the input

If we start with n input keys in the input, in STEP 2a above we have n sequences each of size 1. A pair of such sequences get merged into a sequence of size 2. There are $n/2$ pairs and this will cost us $n/2$ comparisons. (This is at most n, by overcounting.) The resulting $n/2$ sequences formed in Step 3.1 will be paired and merged into $n/4$ sequences of size 4 in Step 3.2. Two sequences of size 2 and 2 can be merged by performing $2 + 2 - 1 = 3$ comparisons. Thus the $n/4$ pairs will require at most $3 \cdot n/4 = 3n/4 < n$ comparisons as well. We observe that also in Step 3.3 we end up doing at most n comparisons if we merge two sorted sequences of size $n/2$ and $n/2$ (since $n/2 + n/2 - 1 < n$ comparisons will be needed for merging). Thus if each step such as Step 3.1, 3.2, 3.3 takes at most n comparisons, we need to figure out how many such steps we have. The answer is $\lceil \lg n \rceil$, where $\lg n$ is the base two logarithm of n, since the size of a sequence doubles every time.

(c) Copyright A. Gerbessiotis. All rights reserved.

S6. Odd-even transposition sort. We present a very simple, naive, and highly inefficient sorting algorithm that performs close to n^2 comparisons to sort n keys arranged as vector elements. Even if the algorithm is not as efficient as mergesort, it is however very elegant and simple. It consists of n rounds: the "odd" rounds (i.e. indexed $1, 3, 5, 7$) and the "even" rounds (i.e. indexed $2, 4, 6, 7$). In an odd-round an odd indexed key is compared to the key on its right (which is an even indexed one); if the former is greater than the latter they get swapped. An even round works similarly. In each one of the n rounds, roughly $n/2$ comparisons are performed. Thus overall approximately $n \times n/2$ comparisons are performed. We call this algorithm a `quadratic` running time algorithm for that. Because of the way it works the algorithm is known as odd-even transposition sort: a refinement of it is know as `bubblesort`. Odd-even transposition sort is an `oblivious` sorting algorithm: the schedule of comparisons is fixed in advance. (Thus in the first round, the 1st vs 2nd, 3rd vs 4th, etc comparisons are performed no matter what the input looks like, and similarly the same schedule is followed in the third, fifth etc rounds.)

```
function  x= sub8oddeven ( x)
 n = length(x);
 for ii= 1:n ;
  if mod(ii,2)==1
    for jj=1:2:n-1
       if (x(jj) > x(jj+1))
          temp=x(jj);
          x(jj)=x(jj+1);
          x(jj+1)=temp;
       end
    end
  else
    for jj=2:2:n-1
       if (x(jj) > x(jj+1))
          temp=x(jj);
          x(jj)=x(jj+1);
          x(jj+1)=temp;
       end
    end
  end
end
```

(c) Copyright A. Gerbessiotis. All rights reserved.