S O R T I N G

Sorting is interpreted as arranging elements of a list in some particular order. In this handout we discuss different sorting techniques for **a list implemented as an array.**

In all algorithms of this handout the sorting of elements is in ascending order (to sort in descending order you just need to change < and <= operations, when comparing elements, with > and >= correspondingly).

In all algorithms of this handout the indexing is consistent with Java which means that a list of N elements is saved in *list[0..N-1]* array segment (i.e. the list occupies array cells 0,1,…,N-1).

Attention: In all presented algorithms the only two operations performed on data are assignment and comparison. Such sorting algorithms are known as ***comparison-based*** algorithms. When analyzing this type of algorithms, for the abstract measure of running time it is customary to take the **number of element-comparisons**.

For simplicity, all algorithms are implemented for a list of *int* values.
Everywhere in this handout we assume that prior to calling an algorithm:

-- the *N* variable is declared and assigned some value

-- the *list* array is declared, defined (allocated memory for **at least** N elements), and its first N cells are filled with values

Attention: *N* and *list.length* are two different things. The variable *N* is the length of the list (i.e. the number of elements in the list) which changes with every added/deleted element, while *list.length* is the size of the array – a constant value that is set at the creation of the array.

1. Selection Sort Algorithm

To sort a list of N elements, this algorithm makes N-1 passes through the list, rearranging elements as follows: the smallest element is found and put into the 1-st position (index 0), then the 2-nd smallest is found and put into the 2-nd position, then the 3-rd smallest is found and put into the 3-rd position, etc.
Note: "an element is put into *i*-th position" means it is swapped with the element at *i*-th position.

Example:

The original list	(7)	4	2	3	10	9	1	8
After 1-st iteration of *for*-loop	1	(4)	2	3	10	9	7	8
After 2-nd iteration of *for*-loop	1	2	(4)	3	10	9	7	8
After 3-rd iteration of *for*-loop	1	2	3	(4)	10	9	7	8
After 4-th iteration of *for*-loop	1	2	3	4	(10)	9	7	8
After 5-th iteration of *for*-loop	1	2	3	4	7	(9)	10	8
After 6-th iteration of *for*-loop	1	2	3	4	7	8	(10)	9
After 7-th iteration of *for*-loop	1	2	3	4	7	8	9	10

Notes: The *oval* shows the *i*-th position that should be considered (arranged) in the **next** iteration. The *red* highlights the two elements that were swapped at the end of the given iteration.

Here is a description of the work of the selection sort algorithm:

FOR every *i* starting with first and ending with pre-last position of the list
  Find the minimum value in *list[i .. length]* and save its index in *minIndex*
  Swap the two elements at *minIndex* and *i* positions

The running time of this routine is $\Theta(N^2)$.

2. Mergesort Algorithm

Mergesort is one of the faster algorithms for sorting a list of elements implemented as an array. **The running time of this routine is Θ(NlogN).**

The idea implemented in mergesort is simple: cut the list into half, sort the left and right halves separately, and then merge the two sorted halves into one sorted list. The same strategy is used for sorting each of the halves. It is easy to see that **mergesort is a recursive routine**. The cutting of the list into halves continues until there is only one element left in the portion of the list to be sorted (the base case). Here is the description of the work of mergesort:

IF the list contains more than one element
    Cut the list into half //This simply means get the midpoint of the list
    Mergesort the first half
    Mergesort the second half
    Merge the two sorted halves into one sorted list

Mergesort is a great example of **"divide-and conquer" strategy** according to which the problem is split into roughly equal sub-problems (the "divide" part), then these sub-problems are solved recursively and their solutions are patched together to obtain the solution of the whole problem (the "conquer" part).

Since mergesort is a recursive algorithm, its Java-implementation will include two methods: one is a **public** method that the client invokes to sort the given list, and second is a **private** recursive method that is there to do the "behind-the-scenes" work to sort a list **segment** (let's call both methods *mergeSort*). Thus, its Java implementation will include this code:

```
public static void mergeSort (int[] list, int N) {  mergeSort(list, 0, N-1);  }

private static void mergeSort (int[] list, int first, int last)   //this method sorts list[first..last] segment
{       if (first < last)       //checking if there is more than one element in list[first..last] segment
        {   int middle = (first + last)/2;
            mergeSort(list, first, middle);
            mergeSort(list, middle+1, last);
            mergeSortedHalves (list, first, middle, last);  //call a supporting method to merge 2 halves
 }      }
```

The *mergeSortedHalves* supporting method will be private and will have the following signature:

```
private static void mergeSortedHalves (int[] arr, int left, int middle, int right)
                                    //Merges two sorted halves of the array segment arr[left..right]
                                    //Precondition: arr[left..middle] is sorted; arr[(middle+1)..right] is sorted
                                    //Postcondition: arr[left..right] is sorted.
```

Here is a description of the work of the *mergeSortedHalves* method:

Create a temporary array *temp* of length ***right-left+1***
SET *index1* to first cell of the 1-st half //it is used to scan through elements of the 1-st half
SET *index2* to first cell of the 2-nd half //it is used to scan through elements of the 2-nd half
SET *index* to the first cell of *temp* array //it is used to move through cells in *temp* array
WHILE there are elements in both halves
    Save the smaller of two halves' indicated elements into the indicated cell of *temp*
    Increment the index of appropriate half (the one that had smaller value)
    Increment index of *temp* to point to the next cell
Copy all remaining elements of the **un-finished** half into the remaining cells of *temp* array
Copy all elements from *temp* array back into *arr[**left..right**]*

Note: drawback of mergesort is the need to use a temp array (takes extra time to copy back).

3. Quick Sort Algorithm

The last algorithm we'll discuss for sorting a list implemented as an array is quicksort – another good example of **"divide-and conquer"** strategy.

The **average** running time of this algorithm is $\Theta(N\log N)$. In its worst case it performs in $\Theta(N^2)$ time so it is **technically an $O(N^2)$ algorithm**, however it is still considered one of the fastest algorithms in practice: with some effort the worst case is made exponentially unlikely, so **this algorithm can be practically classified as an O(NlogN) algorithm based on its average case.**

For clarity of explanations, **let's assume for now that elements in the list are distinct**; it will be noted later that everything works exactly the same way if duplicates are present.

The **idea** implemented in the quicksort is the following: select some value in the list to be the split value (usually called **pivot**) and split the list into two sub-lists so that the first one contains all elements smaller than the pivot, and the second one contains all elements greater than the pivot. To implement this idea, we rearrange the list to obtain the following order of elements: moving from left to right we should see all elements of the first sub-list (i.e. elements smaller than the pivot), then the pivot, and then all elements of the second sub-list (i.e. elements greater than the pivot). Once the list is rearranged, **we sort each sub-list using the same strategy**.

Easy to see, that **quicksort is a recursive routine**. Selecting a pivot and splitting the list (i.e. rearranging) continues until there is no more than one element left (the base case).
Here is the description of the work of quicksort:

IF the list contains more than one element
  Set the pivot // pick a pivot and save it at the **end** of the list
  Split the list //Rearrange the list into [*first sub-list*] [*pivot*] [*second sub-list*]
  Quicksort the first sub-list //elements smaller than pivot are sorted
  Quicksort the second sub-list //elements greater than pivot are sorted

Because quicksort is recursive, its Java implementation, just as mergesort's implementation, will include two methods: one is a **public** method that the client invokes to sort the given list, and the second is a **private** recursive method that does the "behind-the-scenes" work to sort a list **segment**. Thus, the Java implementation of quicksort will include this code:

```
public static void quickSort (int[] list, int N) {   quickSort(list, 0, N-1);   }

private static void quickSort (int[] list, int first, int last)  //sorts list[first..last] segment
{       if (first < last)    //checking if there is more than one element in list[first..last] segment
        {   setPivotToEnd(list, first, last);    //call to a supporting method
            int pivotIndex = splitList (list, first, last);   //call to a supporting method
            quickSort(list, first, pivotIndex-1);
            quickSort(list, pivotIndex+1, last);
 }      }
```

There are two **private** supporting methods: *setPivotToEnd* and *splitList* that need to be defined.

1) The *setPivotToEnd* method chooses the pivot-value and places it as the last element of the list-segment, so after this method is done, we guarantee that the pivot-value is the last element in the array-segment given as a parameter. The signature of this method is this:

```
private static void setPivotToEnd (int[] arr, int left, int right)
                                    //Chooses a pivot in arr[left..right] and place it at the end of the segment
                                    //Precondition: none
                                    //Postcondition: arr[right] is the pivot.
```

Strategy for choosing the pivot: The pivot is one of the list-elements. There are different ways to choose it (some good, some bad). If we choose a specific element in the list (bad idea) – e.g. the first or the last element– the algorithm may take $O(N^2)$ time since that may trigger the worst case scenario. Choosing a random element in the list doesn't give the best choice either.

We'll use one of the better strategies: as a pivot we'll take the **median of three elements,** namely, **the median of first, last, and middle/center elements** of the list segment.
Note: a median of N elements is the $\lceil N/2 \rceil$-th largest element.

Example: Given the list { 7, 8, 1, 5, 2, 9, 12, 10 } //first, last, center elements are in red
The pivot will be chosen as the median of 7, 5, 10 which is 7 (second largest).

In addition to picking the pivot as the "median of 3", **we will apply a special trick which will make the worst case scenario of this algorithm very unlikely**: we will rearrange those three elements – *arr[left]*, *arr[center]*, and *arr[right]* – so that the smallest of the three is at the *left* index, the largest of the three is at the *center* index, and the median of the three is at the *right* index. As a result we'll get the pivot (the median of the three) selected and placed at the end of the list segment. Reminder: the *center* index is calculated as: *center = (left+right)/2*).

For the above example, when *setPivotToEnd* finishes its work, the content of the list is as follows: { 5, 8, 1, 10, 2, 9, 12, 7 }

Implementing the *setPivotToEnd* method. This method is very simple and consists of only 3 independent if-statements (no else-cases, no nesting) that compare two of the mentioned three elements at a time, swapping them properly (if needed) to achieve the above-mentioned placing of those three. Here is the description of the steps that accomplish the task efficiently:

1. Compare fist and center elements and swap them if needed to place the smaller of the two values at *left* index.
2. Compare first (now it is the smaller of previously first and center elements) and last elements and swap them if needed to place the smaller of the two values at *left* index. After this, the smallest of the three elements will be located at the *left* index.
3. Now there are only 2 elements left to re-arrange. Compare the center and last elements and swap them if needed to place the larger of these two values at *center* location (this is the largest of the three elements). As a result, the median of the three values will end up being at *right* index.

2) In the *splitList* method we rearrange elements of the list segment so that the pivot-value is preceded by smaller values and followed by greater values of the segment. This method must be called **after** the pivot has been selected and placed at the end of the segment (it's a precondition):

```
private static int  splitList (int[]  arr, int left, int right)
                    //Rearranges the list by placing the pivot so that it is preceded by smaller
                    //values and followed by greater values. Returns pivot's index.
                    //Precondition: arr[right] contains the pivot
                    //Postcondition: the pivot is properly placed and its index is returned.
                    //      Elements in the list are arranged so that arr[i]<pivot for all arr[i]
                    //      located to the left of pivot, and arr[i]>pivot for all arr[i] located to
                    //      the right of the pivot.
```

Let's see how the splitting of the list segment *arr*[*left..right*] is done (remember, the pivot is the last element in this list segment).

We define two indexes: *indexL* (initially is set to the first index of the list-segment and gets incremented by 1) and *indexR* (initially is set to the second-to-last element, the one in front of the pivot, and gets decreased by 1). Note that these two indexes are for scanning through elements of the segment, starting with edges and moving toward the middle of the list-segment.

We start out by moving *indexL* to the right for **as long as** the value at it is smaller than the pivot-value. Once *indexL* stops, we start moving *indexR* to the left for **as long as** it didn't "cross over" *indexL* (i.e. it didn't become **smaller** than *indexL*) **and** the value at it is larger than pivot-value. Once *indexR* stops, we check the two indexes: if they didn't "cross over", then values at those indexes are on the wrong sides of the list-segment, so we need to swap *arr[indexL]* and *arr[indexR*. We repeat actions defined in this paragraph all over again.

The process stops when *indexL* and *index R* "cross over" (i.e. *indexL>indexR*).

Note that at any given moment in the process the following is true for *arr*[*left..right-1*] segment:
- all elements to the left of *indexL* are smaller than the pivot (although they are not sorted)
- all elements to the right of *indexR* are greater than the pivot (although they are not sorted).

When the whole process is finished, *indexL'*s value is by 1 greater than *indexR'*s, thus the element at *indexL* is the first value greater than pivot. To complete the work, the element at the indexL index is swapped with the pivot element located at the end of the list segment. Now **the pivot is located at *indexL* position** and *indexL* is returned by the method.

Example1: The list segment is the following: {1, 5, 15, 4, 2, 17, 7, 12, 0, 13, 3, 9, 8, 6}.
When the indexes stop moving, the list segment has the following content:
{1, 5, 3, 4, 2, 0, 7, 12, 17, 13, 15, 9, 8, 6}
Note: underscored elements have been moved, each color (other than red) is indicating a pair of elements that have been swapped during the execution of the method.

At this time *indexL* is pointing at 7. As a last step, the pivot is swapped with 7.
The final content of the list segment is: {1, 5, 3, 4, 2, 0, 6, 12, 17, 13, 15, 9, 8, 7}

Example2: The list segment is the following: { 1, 8, 2, 10, 4, 5, 9, 11, 3, 12, 15, 7}
When the indexes stop moving, the list segment has the following content:
{ 1, 3, 2, 5, 4, 10, 9, 11, 8, 12, 15, 7}
Note: underscored elements have been moved, each color (other than red) is indicating a pair of elements that have been swapped during the execution of the method.

At this time *indexL* is pointing at 10. As a last step, the pivot is swapped with 10.
The final content of the list segment is: { 1, 3, 2, 5, 4, 7, 9, 11, 8, 12, 15, 10}

Here is the description of the work done by the *splitList* method:

```
SET indexL to the first cell of the list-segment
SET indexR to the second-from-last cell of the list-segment
SET pivot to the last element of the list-segment   //for clarity, save pivot-value in variable pivot
WHILE the two indexes didn't "cross over"
    Move indexL right as long as elements are smaller than pivot
    Move indexR left as long as it didn't cross over indexL & the element is greater than pivot
    IF the two indexes didn't "cross over"
        Swap elements at these two index positions
        Move indexL and indexR one cell to right and left respectively
Swap the element at indexL position with the pivot  //pivot is at the last cell of the list-segment
RETURN indexL
```

Attention: This routine **as is** works correctly if duplicates are allowed in the list (no changes should be made anywhere).