S O R T I N G

Sorting is interpreted as arranging data in some particular order. In this handout we discuss different sorting techniques for **a list of elements 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 that are performed on data are assignment and comparison. Therefore they 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 the following definitions:

```
final int  MAX_LENGTH =  someInteger;    // Maximum possible length of the list
int[]  list  = new int [MAX_LENGTH] ; //Definition of the list (allocate memory)
int N;        // number of elements in the list (not array)
```

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.

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 .. N-1] and save its index in minIndex
    Swap the two elements at minIndex and i positions
```

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

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.

2. Bubble Sort Algorithm

To sort a list of N elements, this algorithm makes passes through the list until it is perfectly ordered. During each pass, every pair of *neighboring* elements is checked (i.e. elements at 1-st and 2-nd positions, then elements of 2-nd and 3-rd positions, etc.) – if the two elements of the pair are not sorted, they are swapped. If during a pass through the list even one pair is corrected, another pass has to be made (the process stops if no corrections were made during the pass).

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

```
SET  flag UP //flag is a Boolean variable used to determine if the list needs another pass
WHILE  flag is UP
    SET flag DOWN
    FOR every i starting from 1-st and ending with pre-last position of the list
        Compare i-th element and its neighbor; if "bad" pair, swap elements and set flag up
```

Note: maximum number of passes (*while*-loop iterations) through the list is N.

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

Example:

The original list		7	4	2	8	1	10	9	6	flag↑
After 1-st iteration of *while*-loop*	flag↓	4	2	7	1	8	9	6	10	flag↑
After 2-nd iteration of *while*-loop	flag↓	2	4	1	7	8	6	9	10	flag↑
After 3-rd iteration of *while*-loop	flag↓	2	1	4	7	6	8	9	10	flag↑
After 4-th iteration of *while*-loop	flag↓	1	2	4	6	7	8	9	10	flag↑
After 5-th iteration of *while*-loop	flag↓	1	2	4	6	7	8	9	10	flag↓

***Clarification:** To show the work during one pass, here is the detailed description of the first pass through the list (the first while-iteration) – the red shows that elements were swapped, while blue indicates no change.*

The original list		*7*	*4*	*2*	*8*	*1*	*10*	*9*	*6*	*flag↑*
During 1-st iteration of while-loop	*flag↓*	*4*	*7*	*2*	*8*	*1*	*10*	*9*	*6*	*flag↑*
		4	*2*	*7*	*8*	*1*	*10*	*9*	*6*	*flag↑*
		4	*2*	*7*	*8*	*1*	*10*	*9*	*6*	*flag↑*
		4	*2*	*7*	*1*	*8*	*10*	*9*	*6*	*flag↑*
		4	*2*	*7*	*1*	*8*	*10*	*9*	*6*	*flag↑*
		4	*2*	*7*	*1*	*8*	*9*	*10*	*6*	*flag↑*
		4	*2*	*7*	*1*	*8*	*9*	*6*	*10*	*flag↑*

Attention: After the 1-st pass the max element is at the end (in its proper place), after 2-nd pass the 2-nd largest element is in the 2-nd position from the end (in its proper place), etc.

IMPROVEMENT: the performance of bubble sort algorithm can be slightly improved if during each *n*-th pass through the list we compare/check **`only`** the first N-*n* elements with their right neighbor (i.e. the 1-st pass checks elements of the segment *list[0..N-2]*, the 2-nd pass checks elements of the segment *list[0..N-3]*, the 3-rd pass checks elements of *list[0..N-4]*, etc.).

Here is a description of the work of the **improved bubble sort** algorithm:

```
SET  flag UP   //flag is a Boolean variable used to determine if the list needs another pass
SET  last  to the last position    // last is an index (set/point it to the last position in the list)
WHILE  flag is UP
    SET flag DOWN
    FOR every i starting from 1-st and ending with pre-last position of the list
        Compare i-th element and its neighbor; if "bad" pair, swap elements and set flag up
    Decrease last by 1
```

3. 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. We can 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 the second is a **private** method (a recursive method) that is there to do the "behind-the-scenes" work on list *segments* (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);  //supporting method for merging two 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 elements at *index1* and *index2* into the "first" available cell of *temp*
    Increment the appropriate index (the one that had the smaller value)
    Increment the 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: the drawback of mergesort is the necessity to use a temporary array.

4. 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**), 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. Rearrange the list to obtain the following order of elements: moving from left to right in the list we see all elements of the first sublist, then the pivot, and then all elements of the second sublist. Once the list is rearranged, sort each sub-list using the same strategy.

Easy to see, that **quicksort is a recursive routine**. Selecting a pivot and splitting the list (with 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 sublist*] [*pivot*] [*second sublist*]
    Quicksort the first sublist //elements smaller than pivot are sorted
    Quicksort the second sublist //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** method (a recursive method) to do the "behind-the-scenes" work on list segments. Thus, its 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);     //supporting method
                 int pivotIndex = splitList (list, first, last);   //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, 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.
```

One thing to discuss is the strategy for choosing the pivot. There are different ways to do it (some good, some bad). If we choose a particular element in the list – e.g. the first or last element (bad idea) – 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 – the median of first, last, and center elements** of the list segment.
Note: a median of N elements is the $\lceil N/2 \rceil$-th largest element.

> Example: For the following list { 7, 8, 1, 5, 2, 9, 12, 10 }
> The pivot will be chosen to be the median of 7, 5, 10 which is 7.

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. Remember that 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 }

Implement *setPivotToEnd* method on your own. Its work is very simple and consists of only 3 comparisons and swaps. You should compare 3 elements (first, last, and center elements of the list segment) and swap them properly to get the smallest of the three to be at *left* index, largest of the three to be at *center* index, and the median of the three to be at *right* index. As a result we get the pivot chosen (it is the median of the three) and placed at the end of the list segment.

Important: you **must** accomplish the task with **just 3 comparisons** (3 *if*-statements, no *else*'s):

1. Compare the fist and center elements and get the smaller of the two to be at *left* index
2. Compare the first (it is the smaller of previously first and center elements) and last elements and get the smaller of the two to be at *left* index.
   After this step, the smallest of the three elements will be at the *left* index.
3. Now there are only 2 elements left to re-arrange. Compare the center and last elements and get the larger of these two to be at *center* location (this is the largest of the three elements). Consequently, the median of the three will end up being at *right* index.

2) In the *splitList* method we rearrange the 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 list (it has 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 the list segment).

We define two indexes: *indexL* (starts with the first index of the segment and gets incremented by 1) and *indexR* (starts with 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 elements of the segment, starting with edges and moving towards 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. Note that at any moment in the process all elements to the left of *indexL* are smaller than the pivot (although they are not sorted), and all elements to the right of *indexR* are greater than the pivot (although they are not sorted). Once *indexR* stops, if the two indexes didn't "cross over", it means that *arr[indexL]* and *arr[indexR]* elements are on the wrong sides of the list segment, so we need to swap them. We repeat actions defined in this paragraph all over again.
The process stops when *indexL* and *index R* "cross over" (i.e. *indexL>indexR*).
At this point the element at *indexL*, which will be the first element greater than pivot, is swapped with the pivot element located at the end of the list segment. Now **the pivot is located at *indexL* position,** thus *indexL* should be 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's not crossed over indexL & elements are greater than pivot
    IF the two indexes aren't "crossed 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).