This lecture and several more ones cover the sorting problem.

For the beginning, we need to give a definition to this problem.

The input for this problem is a one-dimensional set of items (, so that we can say which item is located after which.

Also, the items must possess the property of total ordering, described in the lecture 1a.

The output is the permutation of original input items in ascending order, that is

Usually, this set is represented as an array, and the items are indexed from to.

Keep in mind that the relation can be applied for numerical values, but the idea of sorting is valid for other non-numerical types as long as the property of total ordering is established.

Let’s start with one of the easiest sorting algorithms.

**Insertion Sort.**

For sorting in this manner, we observe that a sequence of *one* element is sorted.

Then we look at the second element. If it is greater than or equal to the first one, leave it at its place; otherwise move it to the left of the first one.

Second is smaller: move before first

Second is greater: stay at place

Then look at the third element. If it is greater than or equal to the second one, leave it at its place; If it is less than the second but greater than or equal to the first one, move it to the left by one position. If it is less than the first one, move it to the left by two positions, making it the first one.

Third is smaller than first: move by two positions to the left

Third is smaller than second but greater than first: move by one position to the left

Third is greatest: stay at place

The same approach is applied to all remaining elements. In fact, we scan from left to right and insert every element in its proper position.

* Start: the first (leftmost) element is considered on the proper place.
* Moving from left to right, beginning from the second element, insert the current element in the proper position among first sorted elements.
* The second element is placed before the first one, if it is smaller, otherwise it stays on place.
* The third element is placed either before the first one, or before the second one, or stays on place.
* Proceed until the last element is placed correctly.

The following picture visualizes the process.

5

2

4

6

1

3

a) 2 < 5; move 2 before 5.

2 and 5 become sorted.

2

5

4

6

1

3

b) 4 < 5, and 4 > 2; move 4 before 5.

2, 4, and 5 become sorted

2

4

5

6

1

3

c) 6 > 5; stays on place.

2, 4, 5, 6 are sorted

d) 1 < 6, 5, 5, 2; move 1 before 2.

2

4

5

6

1

3

1

2

4

5

6

3

e) 3 < 6, 5, 4, but 3 > 2.

Move 3 before 4

f)

1

2

3

4

5

6

The process of sorting finished.

Once we realized how the process works, we can write the algorithm in the pseudocode.

First, following the approach used in the textbook, we assume that the array is indexed from 1 to .

1.

2.

3. // Insert into the sorted sequence

4.



Pay attention to the indentations: the lined 2 to 5 and the line 8 are indented from the line 1, meaning that they represent the body of the loop. Similarly, the lines 6 and 7 are indented from the line 5: they represent the body of the loop.

Recall, that after an algorithm is represented in any form, we may want to implement it in a programming language. We will use Java in this course, and Java, as most of the languages, assume that array indices are in the range from 0 to , where is the length of the array.

The pseudocode reflects this convention.

1.

2.

3. // Insert into the sorted sequence

4.



8

Now you may be ready to write the Java code, that is translate pseudocode to the Java code.

This process it mostly straightforward; just follow the specific language requirements: braces, parentheses, variable declarations etc.

public static void insertionSort (int [] list) {  
 int n = list.length;  
 for (int j = 1; j < n; j ++) {  
 int key = list[j];  
 int i = j-1;  
 while (i >= 0 && list[i]> key ) {  
 list [i+1] = list[i];  
 i --;  
 }  
 list[i+1] = key;  
 }  
}

Also consider insertion sort procedure for the data organized as a linked list.

You should understand that since the procedure works with traveling along the list both forward and backward, the list must be doubly linked: with references to the next and to the previous nodes.

**Algorithm**

1.

2.

3. // Insert into the sorted s ended by

5.

**6.**

7.

8.

9.

Another sorting algorithm is

**Selection Sort.**

Here is the description in common language:

* Find the smallest element and exchange it (swap) with the element in
* Find the second smallest element and swap it with the element in
* Continue in this manner for the first elements of

The following picture visualizes the process.

1. 1 is smallest

5

2

4

6

1

3

element; swap it

with 5 at the

first position.

1

2

4

6

5

3

1. 2 is the smallest; it

is at proper position.

1

2

4

6

5

3

1. 3 is the smallest

element; swap it

with 4 at the third

position.

1

2

3

6

5

4

1

2

3

4

5

6

1

2

3

4

5

6



The process of sorting finished.

Now we are ready to write a pseudocode for Selection Sort.

Pay attention: in the InsertionSort, we scan the array starting from the second element, because one-element array is considered sorted.

In the SelectionSort, however, we start with the first position, putting there the smallest element, and finish in the second-to-last position, because if they are sorted, the last one is automatically the smallest between this one element.



2. // then find the smallest key among those after and its position

*7.*

8.

9.

10.

You can implement this algorithm in Java code the same way as it was done with insertion sort.

There is an issue here related to the fact that Java language is strictly typed language, meaning that variables must be defined with specific types. So, even if you have the method insertionSort (int [] list)

you cannot use it for sorting array of doubles or array of strings. You would need to create corresponding methods insertionSort (double [] list) or insertionSort (String [] list).

Fortunately, Java has an opportunity to generalize such creation using so called generic types.

The header of the method may look like

public static <T> void insertionSort (T [] list)

where T is a name for any *reference* type.

Then, when you have an array of strings, like String[] words, you can call

insertionSort (words),

and Java would substitute String for T.

Two things must be noted.

1. As was mentioned, T is a reference type, so you would not be able to sort an array of int, but Java has wrappers for all primitive types: Integer for int, Double for double etc.
2. The relation “<” is not defined for non-numerical types. Instead, for some types (actually, classes) the method compareTo is defined, which works similarly:

o1.compareTo(o2) < 0 can be perceived as o1 < o2.

If a class defines the method compareTo, it implements the interface Comparable. The sorting method may work only with comparable items, and it must be reflected in the header.

Also, in the method body you should use calls to compareTo instead of <.

With this understanding, the method insertionSort looks as follows:

public static <T extends Comparable <? super T>>

void insertionSort (T [] list) {  
 for (int j = 1; j < list.length; j ++) {  
 T key = list[j];  
 int i = j-1;  
 while (i >= 0 && list[i].compareTo (key) > 0 ) {  
 list [i+1] = list[i];  
 i --;  
 }  
 list[i+1] = key;  
 }  
}

As you see, the expression list[i] > key is replaced with list[i].compareTo (key) > 0 ; the rest stays the same.

The final part of this lecture is estimating the efficiency of these sorting algorithm.

For this purpose, we select the typical operation and count the number of time it is performed.

There are arithmetic operations (like j++, i---, i+1) , assignments (like j=1, key = list[j]) and comparisons.

A close look suggests that the number of comparisons is approximately the same as number of assignments, and the running time of a comparison may be much longer than that of assignment.

Therefore, we count the number of comparisons.

Denoting , we see iterations of the outer loop

The inner loop performs different number of iterations: from 1 in the best case to in the worst case.

In the best case we have comparisons;

It means

In the worst case we have comparisons.

It means

As always, it is not clear how to define average case. The simplest approach is to assume that in every iteration if the outer loop, there are approximately comparisons. Then we come out to the estimation

So, the conclusion is: insertion sort has quadratic time complexity.

Compute now the number of comparisons in the selection sort:



1. // iterations
2. // then find the smallest key among those after and its position

5. // iterations

*7.*

8.

9.

10.

We are concerned with the comparison which being applied to generics, will look like A[i].compareTo(key) < 0

Counting comparisons ( one per iteration ), we have

,

and again ;

the selection sort procedure is also quadratic in time complexity.