**SECTION 6. BASIC SORTING ALGORITHMS**

6.1. General characteristics of sorting algorithms

Sorting operations are used in almost all areas of human activity. Sorting algorithms are among the most common in IT data processing technologies. In general, the sorting task can be formulated as follows: there is a sequence of similar records, one of which is selected as a key (sorting key). You want to convert the original sequence to a sequence containing the same entries, but in the order in which the key values are increasing (or decreasing). The purpose of sorting is to facilitate a subsequent search of items in a sorted set.

At present, a sufficient number of sorting algorithms have been developed, with different possibilities of application depending on the number of items in the collection. Some algorithms are simple to implement and are suitable for small input datasets, but longer time is required for large datasets. Other algorithms are effective for sorting large datasets, but their use for small datasets is simply not effective. In addition, the algorithms differ in complexity of implementation. Understanding some of them requires visual accompaniment, in particular by using DRAKON- diagrams to represent logic of algorithm and explanatory illustrations. Data sets in the form of a slice almost cover arrays, so this section considers the slice sorting.

6.2. Bubble sorting

Sorting by "bubble" is the simplest algorithm, easy to implement for sets with few items. The algorithm got its name because the larger values gradually "pop up" at the end of the set. The DRAKON-diagram of the algorithm is represented in Figure 6.1. The implementation of the "bubble" sorting algorithm consists of two modules: main() and bubblesort (ar [] int).

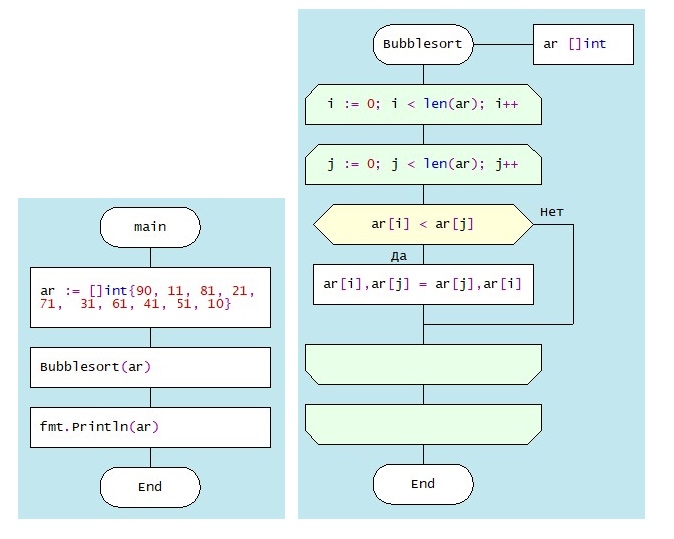


Figure. 6.1. Drakon-diagram of "Bubble" sorting algorithm

In this algorithm, the set is traversed by index (j) for each index (i). In this case, each pair of values ar[i] and ar[j] are compared. If you want to sort values in ascending order, then the two items are swapped if the value of ar[j] is less than the value of ar[i]. Otherwise, there is a transition to the next pass on the index (i). Thus, the largest values appear at the end of the set. Figure6.2. presents a fragment of the algorithm at i = 8, and Figure6.3. displays only strings in which the exchange of items took place.

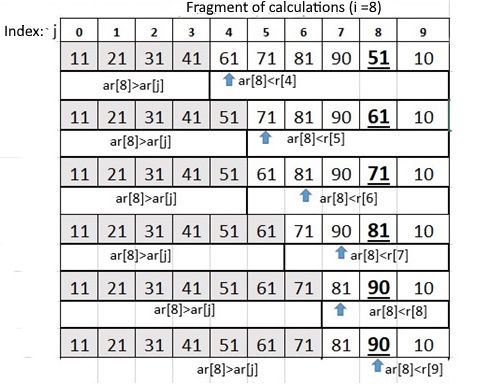


Figure 6.2. Fragment of the sorting process "Bubble”

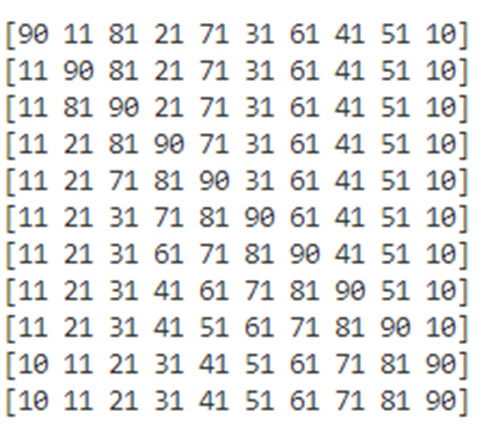


Figure 6.3. The rows in which the items were exchanged

Obviously, the time complexity of this algorithm is quite high O(n2), since it is determined by the number of condition checks ar[i]<ar[j] and the number of exchanges ar[i] ar[j]. At the same time, the space complexity of the "bubble" algorithm is O(1), since it does not require additional memory to organize the computational process. The algorithm has a high level of stability.

Because of its simplicity, Bubble sorting is often used, for example, in computer graphics, where it is popular for its ability to detect minor errors in almost sorted arrays and correct them with linear complexity (2n). However, the "bubble" algorithm is extremely inefficient for sorting large datasets.

6.3. Selection Sort

The choice sorting algorithm is based on the comparison operations, in which the data set is divided into two parts: the sorted left part and the unordered one in the right part. The selection DRAKON-diagram of the sorting algorithm is shown in Figure 6.4.

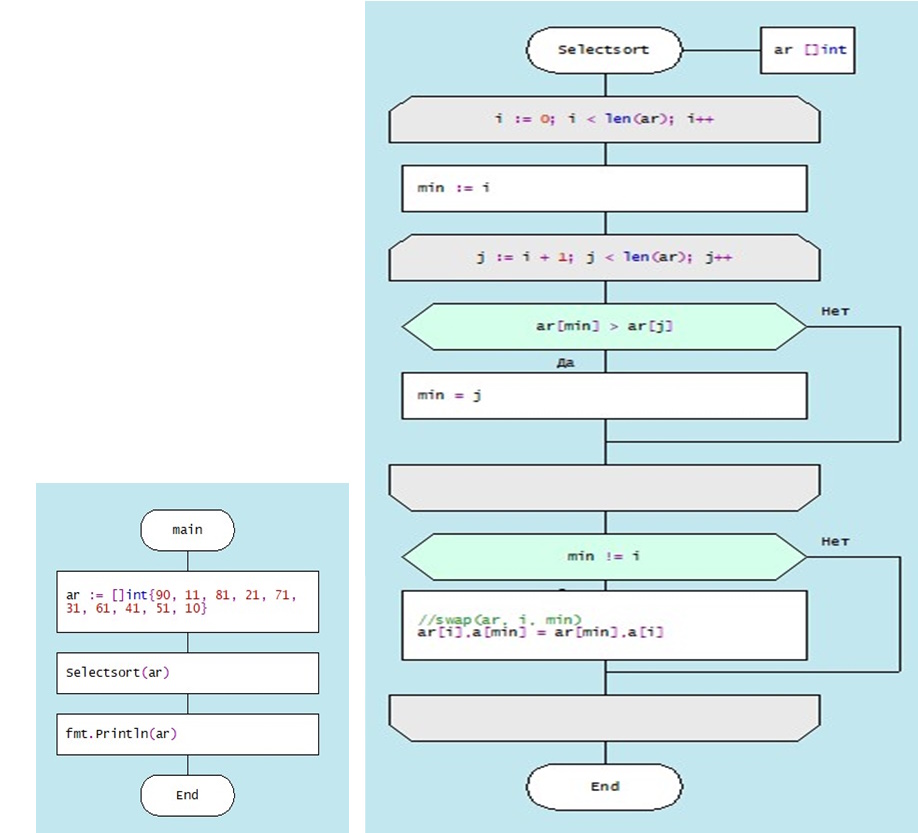


Figure 6.4. DRAKON-diagram of Selection sort algorithm

Selection sort algorithm is performed by obtaining the smallest value in each iteration and then replacing it with the current index. And the sorted part is empty, and the unsaved part is the whole set. The smallest item is selected from an unsorted array and replaced with the leftmost item, and this item becomes part of the sorted array. This process continues to move the undistributed edge of the array to one item to the right. For example, given a set of integers [90, 12, 83, 24, 75, 38, 62, 41, 59, 10]. In the first position, where 90 is currently stored, the algorithm passes the whole set and finds the smallest value - 10, after which the two values are reversed. This process shall then be applied to the remaining items in the slice (Figure 6.5.):

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 90 | 75 | 83 | 24 | 12 | 38 | 62 | 41 | 59 | 10 |
| 10 | 75 | 83 | 24 | 12 | 38 | 62 | 41 | 59 | 90 |
| 10 | 12 | 83 | 24 | 75 | 38 | 62 | 41 | 59 | 90 |
| 10 | 12 | 24 | 83 | 75 | 38 | 62 | 41 | 59 | 90 |
| 10 | 12 | 24 | 38 | 75 | 83 | 62 | 41 | 59 | 90 |
| 10 | 12 | 24 | 38 | 41 | 83 | 62 | 75 | 59 | 90 |
| 10 | 12 | 24 | 38 | 41 | 59 | 62 | 75 | 83 | 90 |
| 10 | 12 | 24 | 38 | 41 | 59 | 62 | 75 | 83 | 90 |
| 10 | 12 | 24 | 38 | 41 | 59 | 62 | 75 | 83 | 90 |
| 10 | 12 | 24 | 38 | 41 | 59 | 62 | 75 | 83 | 90 |

Figure 6.5. Selection and replacement of slice items

|  |  |
| --- | --- |
| Time complexity: |  |
| Worst case | O(n2) |
| Average case | O(n2) |
| Best case | O(n2) |
| Space complexity: | О(1) |
| Selection sort algorithm is unstable |  |

The evaluation of the complexity of the selection sort algorithm is presented in the table.

The reason why time complexity is the same for all three cases is that the sorting algorithm by choice uses two nested cycles. The outer cycle is executed n times, where n is the number of elements in the array. In each iteration of the outer cycle, the inner cycle is executed (n-1) times. Thus, the total number of comparison and permutation operations is n\*(n-1), giving the time complexity of O(n 2) for all three cases 14.

6.4. Insertion Sort

The insertion sort is performed by repeatedly extracting the item from the unordered part of the set and then inserting it into the sorted part of the set until all items are inserted. This algorithm is usually used by people when sorting stacks of papers. The DRAKON-diagram of the insertion sorting algorithm is presented in Figure 6.6.

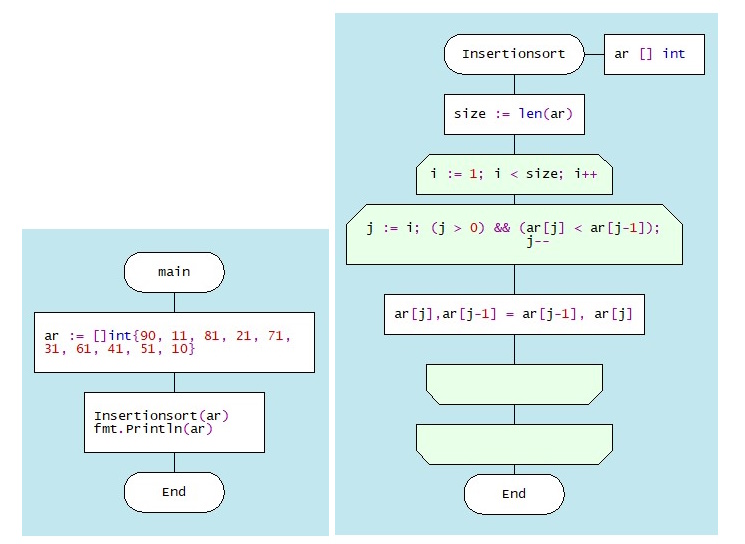


Figure 6.6. DRAKON-diagram of Insertion sort algorithm

In this example, the items with the maximum value "advance" to the right, then a loop is executed, in which the neighboring items are compared and, if necessary, swap places (marked with "\_").

|  |  |  |
| --- | --- | --- |
| i = 1 | i = 4 | i = 8 |
| [11 **90** 81 21 71 31 61 41 51 10] | [11 21 71 81 **90** 31 61 41 51 10] | [11 21 31 41 51 61 71 81 90 **10**] |
| [11 81 **90** 21 71 31 61 41 51 10] | [11 21 71 81 31 **90** 61 41 51 10] | [11 21 31 41 51 61 71 81 **10** 90] |
| i = 2 | [11 21 *71* 31 81 **90** 61 41 51 10] | [11 21 31 41 51 61 71 **10** 81 90] |
| [11 81 **90** 21 71 31 61 41 51 10] | [11 21 31 *71* 81 **90** 61 41 51 10] | [11 21 31 41 51 61 **10** 71 81 90] |
| [11 81 21 **90** 71 31 61 41 51 10] | i = 5 | [11 21 31 41 51 **10** 61 71 81 90] |
| [11 21 81 **90** 71 31 61 41 51 10] | [11 21 31 71 81 **90** 61 41 51 10] | [11 21 31 41 **10** 51 61 71 81 90] |
| i = 3 | i = 7 | [11 21 31 **10** 41 51 61 71 81 90] |
| [11 21 81 **90** 71 31 61 41 51 10] | [11 21 31 41 61 71 81 **90** 51 10] | [11 21 **10** 31 41 51 61 71 81 90] |
| [11 21 81 71 **90** 31 61 41 51 10] | [11 21 31 41 61 71 81 51 **90** 10] | [11 **10** 21 31 41 51 61 71 81 90] |
| [11 21 71 81 **90** 31 61 41 51 10] | [11 21 31 41 61 71 51 81 90 10] | [**10** 11 21 31 41 51 61 71 81 90] |
|  | [11 21 31 41 61 51 71 81 90 10] | i = 9 |
|  | [11 21 31 41 51 61 71 81 90 10] | [10 11 21 31 41 51 61 71 81 90] |

Figure 6.7. Insertion and replacement of slice items

The complexity of the insertion sort algorithm is determined by the number of comparisons and movements of elements that need to be performed to order the array. It depends on how the array is originally sorted.

The worst case scenario is when the array is sorted backwards. In this case, each element should be compared with all the previous elements and moved to the beginning of the array. The number of comparisons and moves is equal to

(n(n-1)/2,

it leads to O(n2)

Average case: when the array is partially sorted. In this case, each element should be compared on average with half of the previous elements and moved to the appropriate position. The number of comparisons and movements is equal to

n2/4,

it leads to O(n2).

The best case is when the array is already sorted. In this case, each element should be compared with only one previous element and left in its place. The number of comparisons and moves is equal to

(n – 1),

it leads to O(n).

The evaluation of the complexity of the insertion sort algorithm is presented in the table.

|  |  |
| --- | --- |
| Time complexity: |  |
| Worst case | O(n2) |
| Average case | O(n2) |
| Best case | O(n) |
| Space complexity: | О(1) |
| Insertion sort algorithm is stable |  |

6.5. Quick Sorting

Quicksort is a highly efficient sorting algorithm whose general scheme consists of the following steps:

1. Selecting a reference item from a slice.

2. Redistributing items in a slice in such a way that items smaller than the reference one are placed in front of it, and those greater or equal - after it.

3. Recursively applying the first previous steps to slice fragments to the left and right of the reference item.

4. As a result, a fully sorted array is formed.

The DRAKON-diagram of the quicksort algorithm is shown in Figure 6.8.

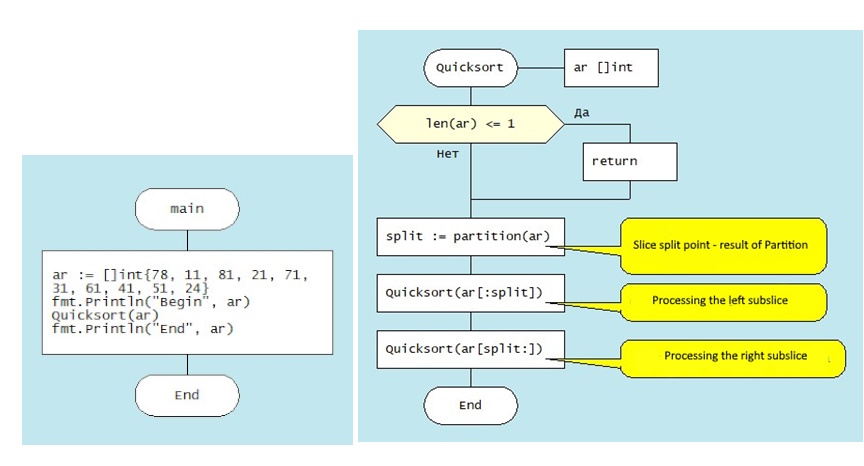
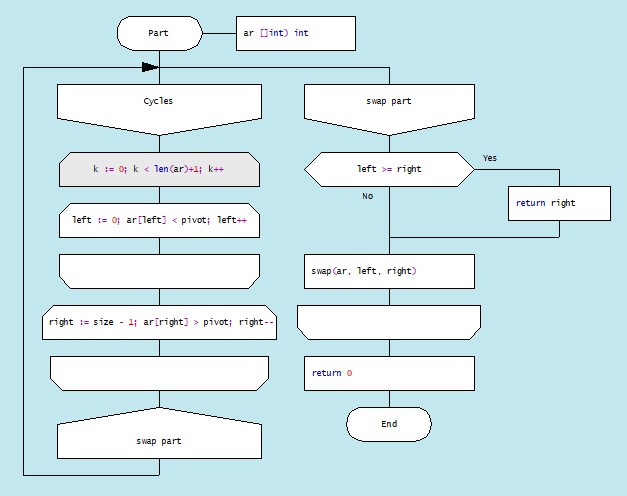
 

Figure 6.8. DRAKON-diagram quicksorting algorithm

a) function main() b) function Quicksort c) function Partition

Consider this algorithm in depth. The main function presents a collection of integer data. The pivot support item is selected with the slice item of 31.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 78 | 11 | 81 | 21 | 71 | 31 | 61 | 41 | 51 | 28 |

Enter two pointers: left and right. At the beginning of the algorithm, they indicate the left and right end of the set respectively. In the left pointer algorithm, the left pointer is moved in 1 step towards the end of the slice until the current item value is less than the reference item value. The index of the first item, whose value is greater than pivot (ar[left] > pivot), is fixed in the variable left. In the algorithm, the right pointer then moves from the end of the slice to the beginning until an item for which the condition ar[right] <= pivot is found. This fragment of the algorithm is executed until an item whose value exceeds that of the right item on the right is found on the left. In this case, these items are reversed. In this example, the first (left) item is larger than the supporting one (78 >31) and the last (right) item is smaller than the supporting one (24 < 31). In the second line of the table they switched. The process continues (Figure 6.7.):

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |  |
| Left |  |  |  |  |  |  |  |  |  |  | Right |
|  | 78 | 11 | 81 | 21 | 71 | 31 | 61 | 41 | 51 | 24 |  |
|  |  |  |  |  |  |  |  |  |  |  |  |
|  | 24 | 11 | 81 | 21 | 71 | 31 | 61 | 41 | 51 | 78 |  |
|  |  |  |  |  |  |  |  |  |  |  |  |
|  | 24 | 11 | 31 | 21 | 71 | 81 | 61 | 41 | 51 | 78 |  |
|  |  |  |  |  |  |  |  |  |  |  |  |
|  | 24 | 11 | 21 | 31 | 71 | 81 | 61 | 41 | 51 | 78 |  |

Figure 6.9. QuickSort algorithm runtime

A new fragment of the source set is then formed in which the support item is overridden. In this fragment, pivot = 11.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  | | --- | |  | | |  | | --- | |  | |  |
| 24 | 11 | 21 |
|  | |  | | --- | |  | |  |
| 11 | 24 | 21 |
|  |  |  |
| 11 | 21 | 24 |

After processing the left part of the set, the right part is processed in the same way. Eventually, the slice becomes sorted:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 11 | 21 | 24 | 31 | 41 | 51 | 61 | 71 | 78 | 81 |

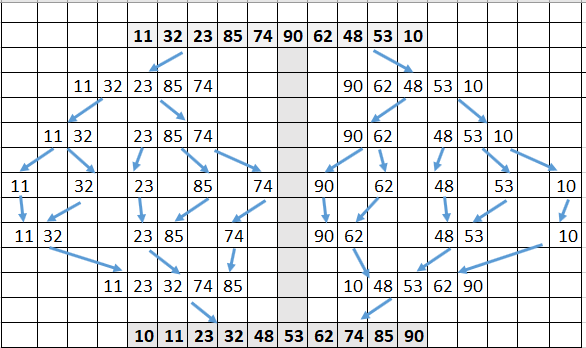
This algorithm is quite effective for large datasets as its average and worst complexity is O(n2), respectively.

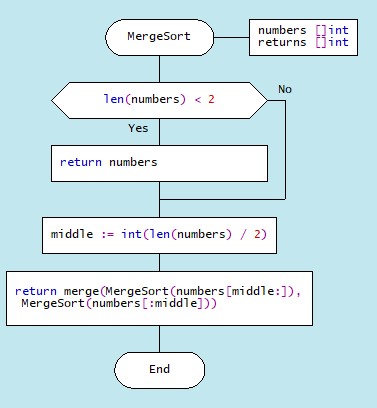
|  |  |
| --- | --- |
| Time complexity: |  |
| Worst case | O(n2) |
| Average case | O(nlogn) |
| Best case | O(nlogn) |
| Space complexity: | O(nlogn) |
| Insertion sort algorithm is unstable |  |

The evaluation of the complexity of the Quiсk sort algorithm is presented in the table.

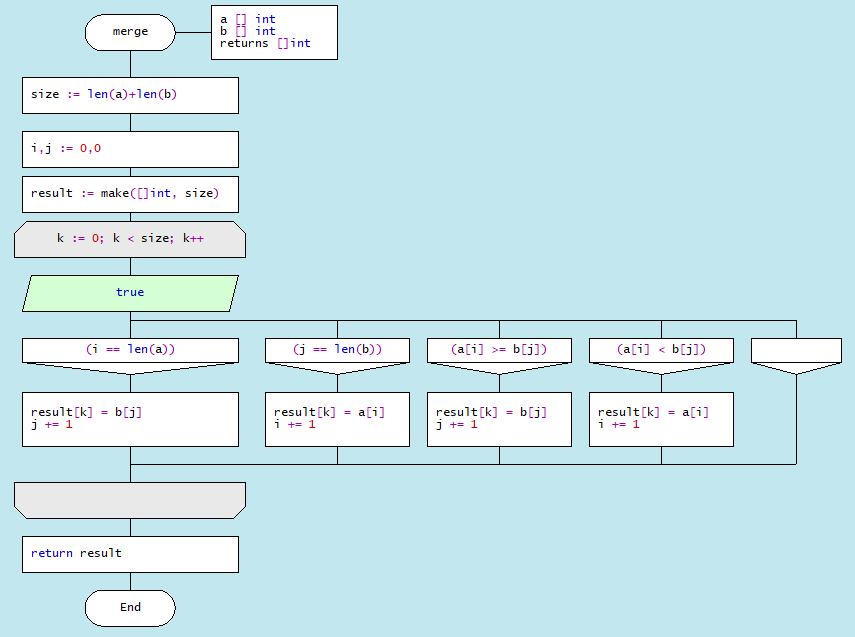
6.5. Mergesort

Merge sorting is performed by recursively splitting the collection into fragments until there is a fragment consisting of two items. The two items are easily compared and ordered according to the requirement: ascending or descending. The split is followed by an inverse merge, in which at one point (or in a loop) one item from each slice fragment is selected and compared. The smallest (or largest) item is stored in the result set, the remaining item remains valid for comparison with an item from another fragment in the following step (Figure 6.8.):

 Figure 6.8. Visualizing the sort algorithm

The DRAKON-diagram of the merge sorting algorithm is presented in Figure 6.9. 

a)



b)

Figure 6.9. The DRAKON-diagram of the Merge sorting algorithm:

a) function MergeSort; b) function merge

The evaluation of the complexity of the Merge sorting algorithm is presented in the table.

|  |  |
| --- | --- |
| Time complexity: |  |
| Worst case | O(nlogn) |
| Average case | O(nlogn) |
| Best case | O(nlogn) |
| Space complexity: | O(n) |
| Insertion sort algorithm is unstable |  |

6.6. Shellsort

ShellSort is a sort of insertion sort. When sorting out, Shellsort first compares and sorts between values that are separated from each other at some distance d. After that, the procedure is repeated for some smaller values d until the distance becomes d=1 (that is, the usual sorting of inserts). The DRAKON-diagram of the algorithm is presented in Figure 6.10.

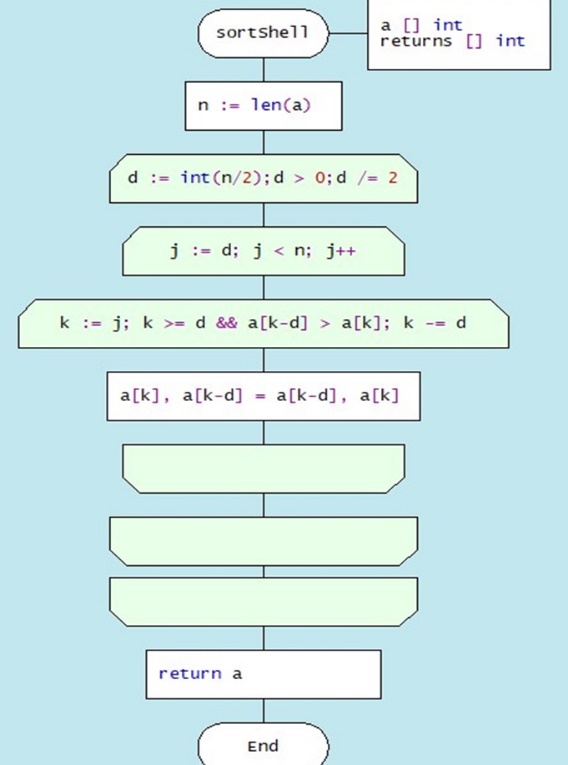


Figure 6.10. DRAKON-diagram of Shellsort algorithm

Shellsort uses a process that underlies many of the sorts presented: sequential segmentation, sorting of these segments, and finally grouping them into a sorted set. The process of partitioning occurs so that each item in the segment represents a fixed number of positions from each other. This creates uncertainty in the choice of this number of positions, in other words, the distance between the items in the segment (d). The simplest example is d = n / 2, d2 = d/2 ... dn =1.

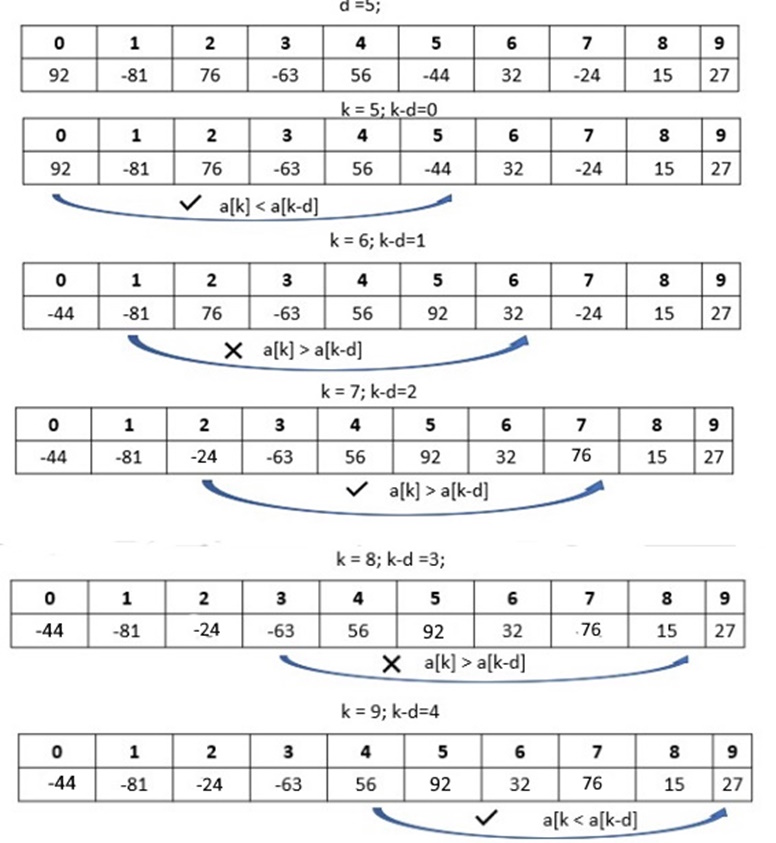
Figure 6.11 shows the exchange of items under a[k] < a[k-d], where d=5 at the first pass of the collection. 

Figure 6.11. Fragment of algorithm of exchange of items of set

The evaluation of the complexity of the shell sorting algorithm is presented in the table.

|  |  |
| --- | --- |
| Time complexity: |  |
| Worst case | O(nlogn)2 |
| Average case | O(nlogn2) |
| Best case | O(n) |
| Space complexity: | O(1) |
| Insertion sort algorithm is unstable |  |

6.7. Pyramid sorting (Heapsort)

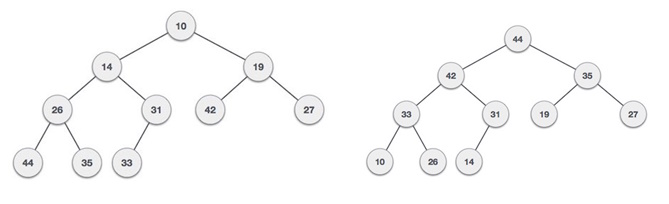
The pyramid sorting algorithm can be seen as an improved version of the choice sorting algorithm (Select Sort): it divides the input data into sorted and unreported areas, and then successively reduces the unreported area, removing the largest item and moving it to the sorted area. An improvement is that the binary pile is used to find the highest value, not the linear search algorithm. This algorithm is executed using the notion of heap, which is a complete binary tree (see sub-section 1.3.). All nodes of a heap are either larger than its child items or smaller than its child items. A heap binary tree can be of two types: a minimum heap (MinHeap), in which the parent node is always smaller than the child nodes, and a maximum heap (MaxHeap), in which the parent node is always greater than or equal to the child nodes Figure 6.12).

Figure 6.12. Binary tree examples

Construction of a binary tree from an array is simplified by the fact that the original array is actually an unordered heap (Figure 6.13.):

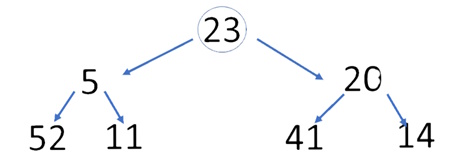


Figure 6.13. Binary tree example

The tree node sequence, starting with the root node, is performed by the formula: iн = (array size / 2) – 1;

First, the algorithm swaps the nodes (20) and (41), then the nodes (5) and (52), then we present this process in the table:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 23 | 5 | 20 | 52 | 11 | 41 | 14 |
| 23 | 5 | 41 | 52 | 11 | 20 | 14 |
| 23 | 52 | 41 | 5 | 11 | 20 | 14 |
| 52 | 23 | 41 | 5 | 11 | 20 | 14 |
| 14 | 23 | 41 | 5 | 11 | 20 | 52 |
| 41 | 23 | 14 | 5 | 11 | 20 | 52 |
| 20 | 23 | 14 | 5 | 11 | 41 | 52 |
| 23 | 20 | 14 | 5 | 11 | 41 | 52 |
| 11 | 20 | 14 | 5 | 23 | 41 | 52 |
| 20 | 11 | 14 | 5 | 23 | 41 | 52 |
| 5 | 11 | 14 | 20 | 23 | 41 | 52 |
| 11 | 5 | 14 | 20 | 23 | 41 | 52 |
| 5 | 11 | 14 | 20 | 23 | 41 | 52 |

The heap sorting algorithm uses three functions: heap\_Sort, which performs node overwriting, heapify, which compares adjacent nodes, and swap, which swap two nodes. The sequence of the nodes in the heap is shown in Figure 6.14:

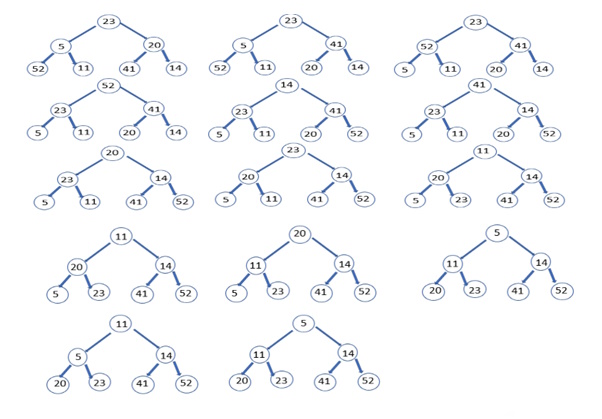
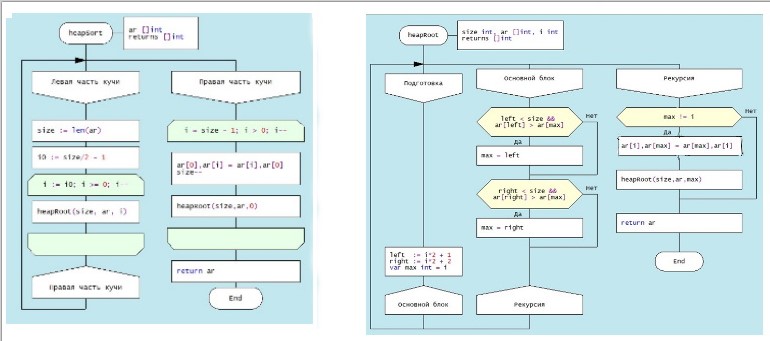


Figure 6.14. Sequence of node movement in heap

DRAKON-diagram of heap algorithm is presented in Figure 6.15:





a) function heapsort b) function heapRoot

Figure 6.15. DRAKON-diagram heap sorting algorithm

The evaluation of the complexity of the heap sorting algorithm presented in the table

|  |  |
| --- | --- |
| Time complexity: |  |
| Worst case | O(nlogn). |
| Average case | O(nlogn). |
| Best case | O(nlogn). |
| Space complexity: | О(1) |
| Insertion sort algorithm is unstable |  |

6.8. Sorting comparison

The selection of a sorting algorithm is determined by the following factors:

• Time complexity;

• Spatial complexity;

• Stability/instability.

Knowing the strengths and weaknesses of each of the algorithms considered allows you to make a choice in favor of a particular sort. Each algorithm is unique and works best under certain conditions.

|  |  |  |  |
| --- | --- | --- | --- |
| Sorting algorithm | Average | Best | Worst |
| [Bubble Sort](https://coderslegacy.com/python/bubblesort-algorithm/) | O(n2) | O(n) | O(n2) |
| [Selection Sort](https://coderslegacy.com/python/selection-sort-algorithm/) | O(n2) | O(n2) | O(n2) |
| [Insertion Sort](https://coderslegacy.com/python/insertion-sort-algorithm/) | O(n2) | O(n) | O(n2) |
| [Quick Sort](https://coderslegacy.com/python/quicksort-algorithm/) | O(n.log(n)) | O(n.log(n)) | O(n2) |
| [Merge Sort](https://coderslegacy.com/python/merge-sort-algorithm/) | O(n.log(n)) | O(n.log(n)) | O(n.log(n)) |
| Shell Sort | n(log(n)2 | O(n) | n (log n)2 |
| [Heap Sort](https://coderslegacy.com/python/heap-sort-algorithm/) | O(n.log(n)) | O(n.log(n)) | O(n.log(n)) |

Some common sorting algorithms are inherently stable, such as merge sort, count sort, insertion sort, and bubble sort. Others, such as Quicksort, Heapsort, and Selection Sort, are unstable. For example, we can use the extra space to maintain stability in Quicksort.