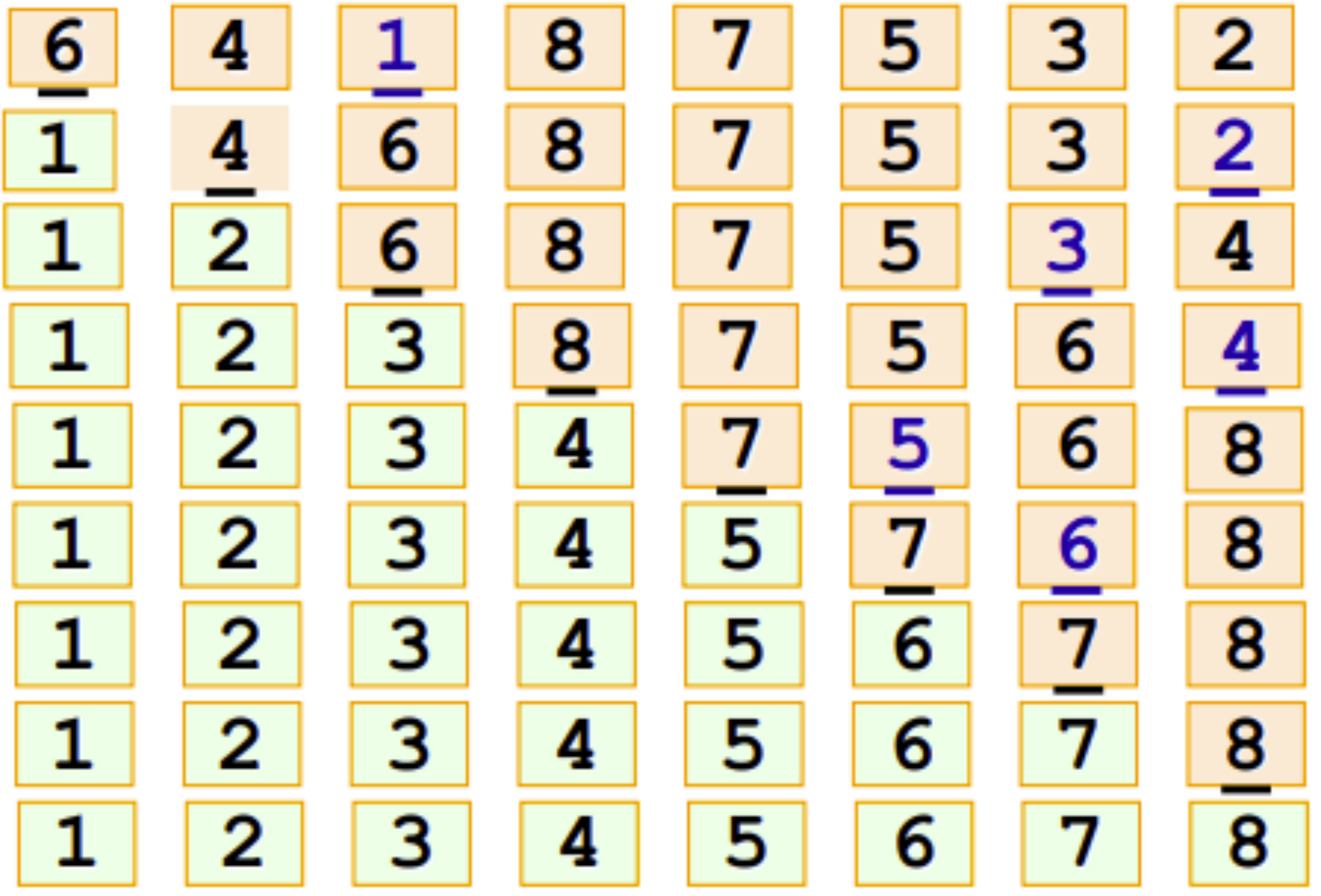
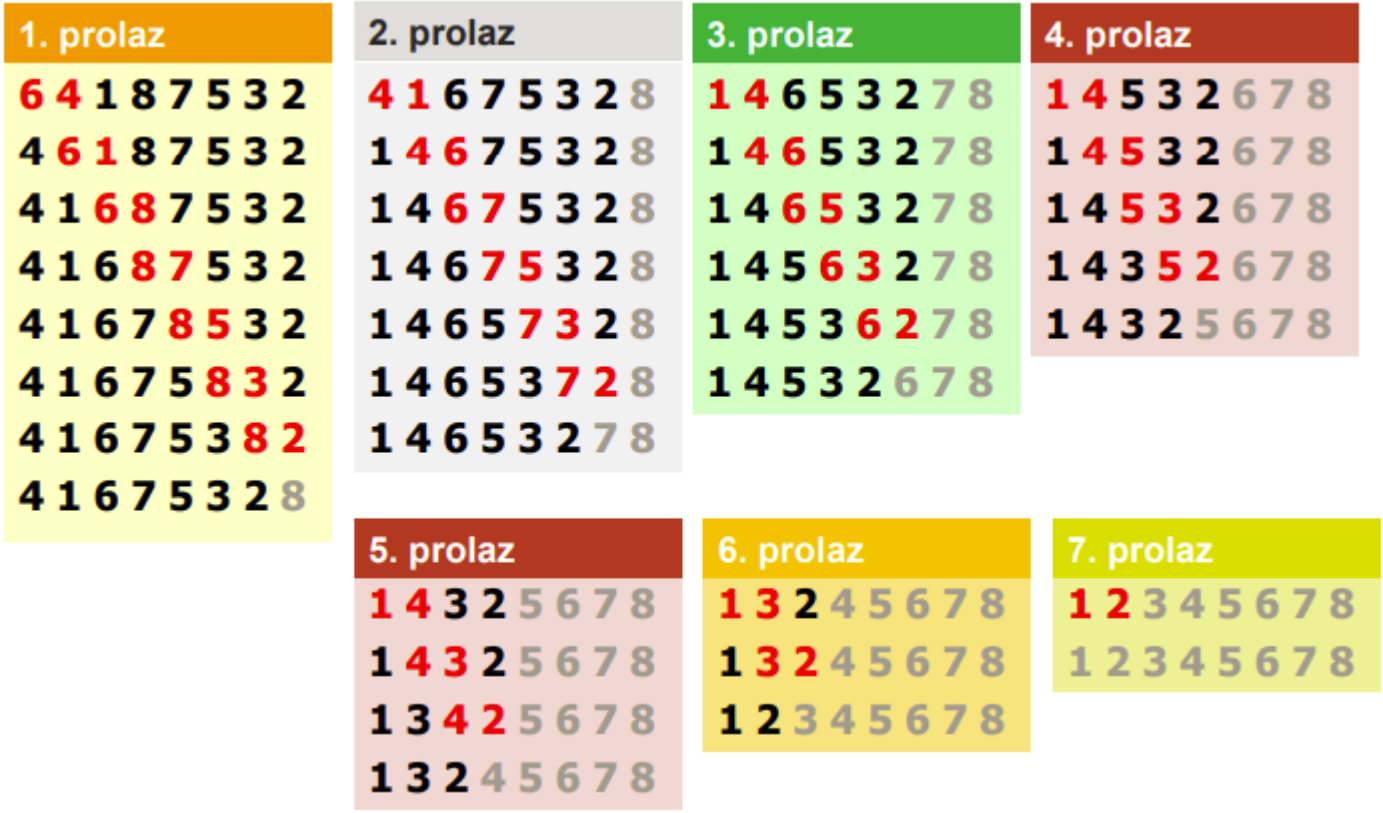
naziv	najbolje	prosječno	najlošije	stabilan
selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	ne
insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$	da
bubble sort	$O(n)$	-	$O(n^2)$	da
shell sort	-	-	$O(n^{3/2})$	ne
merge sort	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	da
quick sort	$O(n\log n)$	$O(n\log n)$	$O(n^2)$	ne
heap sort	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	ne


6 4 1 8 7 5 3 2
1 4 6 8 7 5 3 2
1 2 6 8 7 5 3 4
1 2 3 8 7 5 6 4
1 2 3 4 7 5 6 8
1 2 3 4 5 7 6 8
1 2 3 4 5 6 7 8
1 2 3 4 5 6 7 8
1 2 3 4 5 6 7 8

```
template <typename T> static void SelectionSort(T A[], size_t n) {
    size_t i, j, min;

    for (i = 0; i < n; i++) {
        min = i;

        for (j = i + 1; j < n; j++) {
            if (A[j] < A[min])
                min = j;
        }

        Swap(&A[i], &A[min]);
    }
}
```

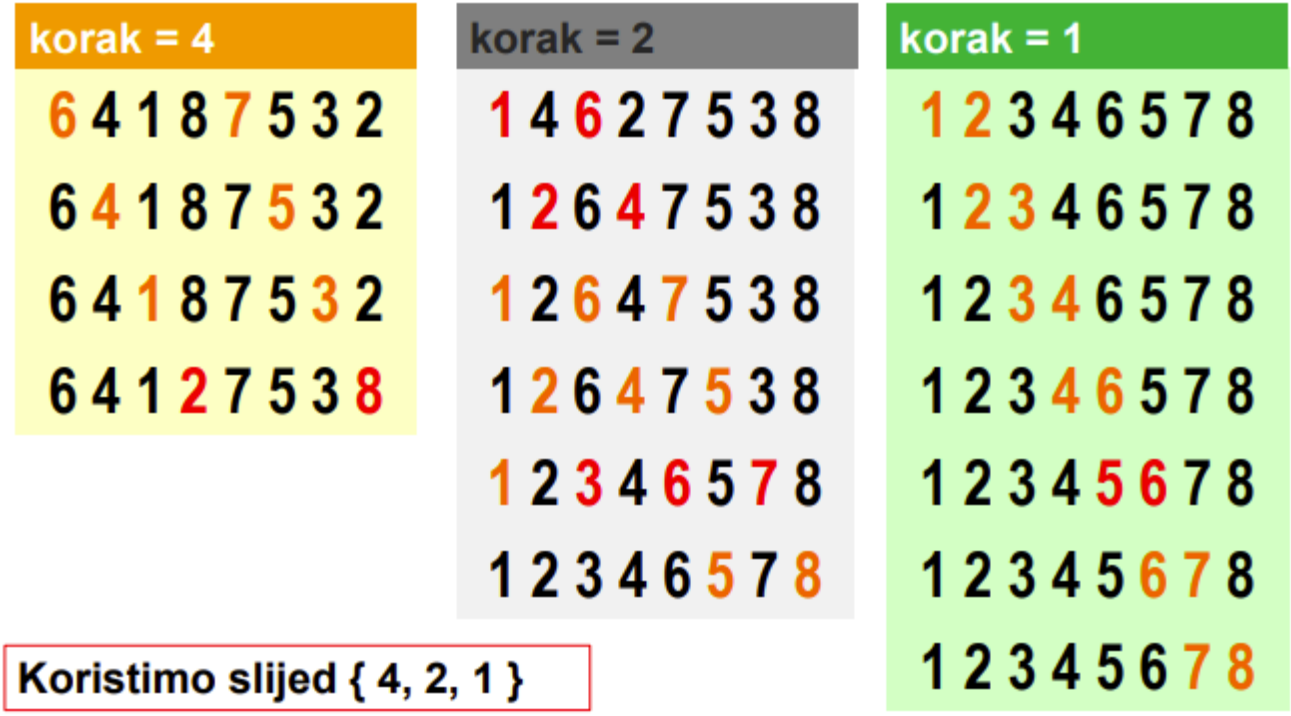

1. prolaz
6 4 1 8 7 5 3 2
4 6 1 8 7 5 3 2
4 1 6 8 7 5 3 2
4 1 6 8 7 5 3 2
4 1 6 7 8 5 3 2
4 1 6 7 5 8 3 2
4 1 6 7 5 3 8 2
4 1 6 7 5 3 2 8
2. prolaz
4 1 6 7 5 3 2 8
1 4 6 7 5 3 2 8
1 4 6 7 5 3 2 8
1 4 6 7 5 3 2 8
1 4 6 5 7 3 2 8
1 4 6 5 3 7 2 8
1 4 6 5 3 2 7 8
3. prolaz
1 4 6 5 3 2 7 8
1 4 6 5 3 2 7 8
1 4 6 5 3 2 7 8
1 4 5 6 3 2 7 8
1 4 5 3 6 2 7 8
1 4 5 3 2 6 7 8
4. prolaz
1 4 5 3 2 6 7 8
1 4 5 3 2 6 7 8
1 4 5 3 2 6 7 8
1 4 3 5 2 6 7 8
1 4 3 2 5 6 7 8
5. prolaz
1 4 3 2 5 6 7 8
1 4 3 2 5 6 7 8
1 3 4 2 5 6 7 8
1 3 2 4 5 6 7 8
6. prolaz
1 3 2 4 5 6 7 8
1 3 2 4 5 6 7 8
1 2 3 4 5 6 7 8
7. prolaz
1 2 3 4 5 6 7 8
1 2 3 4 5 6 7 8

```
template <typename T> static void BubbleSort(T A[], size_t n) {
    for (size_t i = 0; i < n - 1; i++)
        for (size_t j = 0; j < n - 1 - i; j++)
            if (A[j + 1] < A[j])
                Swap(&A[j], &A[j + 1]);
}
```

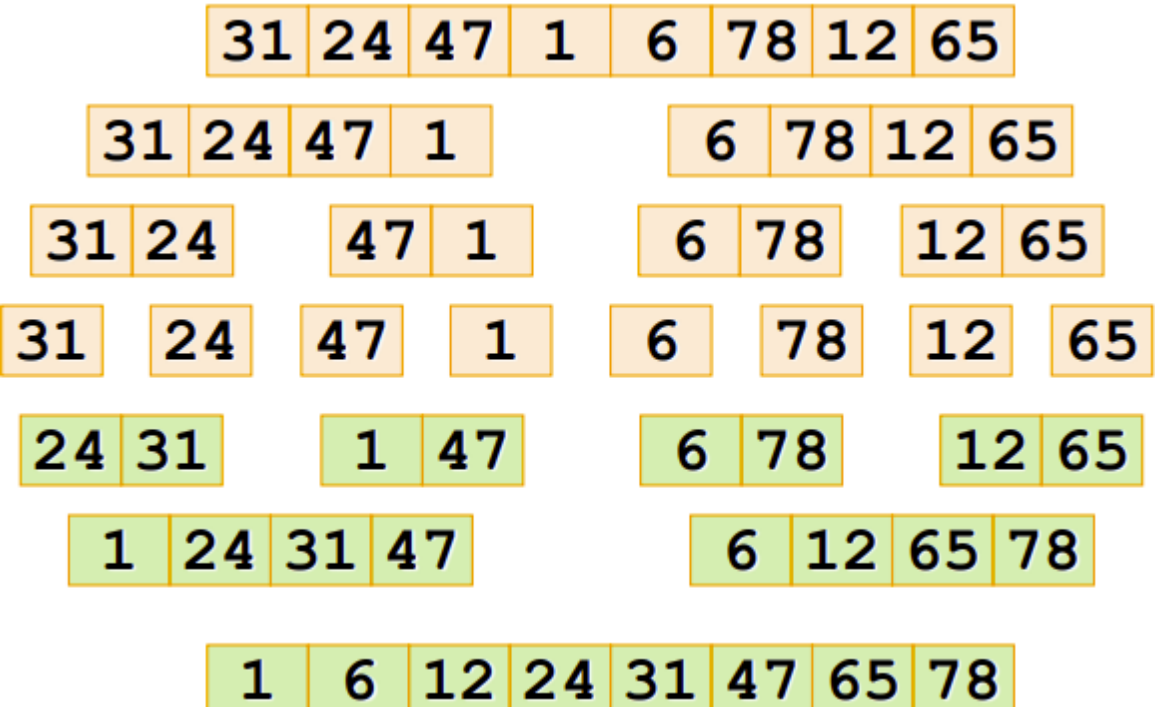
```
template <typename T> static void BubbleSortEnhanced(T A[], size_t n) {
    bool swapHappened = true;

    for (size_t i = 0; swapHappened; i++) {
        swapHappened = false;

        for (size_t j = 0; j < n - 1 - i; j++) {
            if (A[j + 1] < A[j]) {
                Swap(&A[j], &A[j + 1]);

                swapHappened = true;
            }
        }
    }
}
```

6	4	1	8	7	5	3	2
6	4	1	8	7	5	3	2
4	6	1	8	7	5	3	2
1	4	6	8	7	5	3	2
1	4	6	8	7	5	3	2
1	4	6	7	8	5	3	2
1	4	5	6	7	8	3	2
1	3	4	5	6	7	8	2
1	2	3	4	5	6	7	8

```
template <typename T> static void InsertionSort(T A[], size_t n) {
    size_t i, j;
    T temp;

    for (i = 1; i < n; i++) {
        temp = A[i];

        for (j = i; j >= 1 && A[j - 1] > temp; j--)
            A[j] = A[j - 1];

        A[j] = temp;
    }
}
```


korak = 4
6 4 1 8 7 5 3 2
6 4 1 8 7 5 3 2
6 4 1 8 7 5 3 2
6 4 1 2 7 5 3 8
korak = 2
1 4 6 2 7 5 3 8
1 2 6 4 7 5 3 8
1 2 6 4 7 5 3 8
1 2 6 4 7 5 3 8
1 2 3 4 6 5 7 8
1 2 3 4 6 5 7 8
korak = 1
1 2 3 4 6 5 7 8
1 2 3 4 6 5 7 8
1 2 3 4 6 5 7 8
1 2 3 4 6 5 7 8
1 2 3 4 5 6 7 8
1 2 3 4 5 6 7 8
1 2 3 4 5 6 7 8
Koristimo slijed { 4, 2, 1 }

```
template <typename T> static void ShellSort(T A[], size_t n) {
    size_t i, j, step;
    T temp;

    for (step = n / 2; step > 0; step /= 2) {

        for (i = step; i < n; i++) {
            temp = A[i];

            for (j = i; j >= step && A[j - step] > temp; j -= step)
                A[j] = A[j - step];

            A[j] = temp;
        }

    }
}
```


31 24 47 1 6 78 12 65
31 24 47 1
6 78 12 65
31 24
47 1
6 78
12 65
31
24
47
1
6
78
12
65
24 31
1 47
6 78
12 65
1 24 31 47
6 12 65 78
1 6 12 24 31 47 65 78

```
template <typename T> static void MergeArrays(T A[], T helperArray[],
      size_t leftPosition, size_t rightPosition, size_t rightEnd) {

   size_t i, leftEnd, numOfElements, helperPosition;

   leftEnd = rightPosition - 1;
   helperPosition = leftPosition;
   numOfElements = rightEnd - leftPosition + 1;

   while (leftPosition <= leftEnd && rightPosition <= rightEnd) {
      if (A[leftPosition] <= A[rightPosition])
         helperArray[helperPosition++] = A[leftPosition++];
      else
         helperArray[helperPosition++] = A[rightPosition++];
   }

   while (leftPosition <= leftEnd)
      // copy the remainder of the first half
      helperArray[helperPosition++] = A[leftPosition++];
   while (rightPosition <= rightEnd)
      // copy the remainder of the second half
      helperArray[helperPosition++] = A[rightPosition++];

   for (i = 0; i < numOfElements; i++, rightEnd--)
      A[rightEnd] = helperArray[rightEnd]; // copy temp array back to the
                                           // original array
}

template <typename T>   static void MergeRecursive(T A[], T helperArray[],
      size_t left, size_t right) {
   size_t middle;

   if (left < right) {
      middle = (left + right) / 2;

      MergeRecursive(A, helperArray, left, middle);
      MergeRecursive(A, helperArray, middle + 1, right);
      MergeArrays(A, helperArray, left, middle + 1, right);
   }
}

ublic:
template <typename T> static void MergeSort(T A[], size_t n) {
   T *helperArray;
   helperArray = new (nothrow) T[n];

   if (helperArray != nullptr) {
      MergeRecursive(A, helperArray, 0, n - 1);
      delete[] helperArray;
   } else
      throw bad_alloc();
}
```

izbor stožera

```
8 1 4 9 6 3 5 2 7 0
^       ^         ^
0 1 4 9 6 3 5 2 7 8
```

i i j se mimoilaze

```
0 1 4 2 5 3 7 9 6 8
        j i
0 1 4 2 5 3 7 9 6 8
```

vraćamo stožer na i

```
0 1 4 2 5 3 7 9 6 8
          i
0 1 4 2 5 3 6 9 7 8
```

```
0 1 4 9 7 3 5 2 6 8
i->           <-j
0 1 4 9 7 3 5 2 6 8
      i       j
0 1 4 2 7 3 5 9 6 8
        i   j
0 1 4 2 5 3 7 9 6 8
```

izbor stožera

```
0 1 4 2 5 3   6
```

izbor stožera

```
  9 7 8
```

```
// QuickSort Pivot=Median
template <typename T> static T medianOf3(T A[], size_t left, size_t right) {
    size_t middle = (left + right) / 2;

    if (A[left] > A[middle])
        Swap(&A[left], &A[middle]);
    if (A[left] > A[right])
        Swap(&A[left], &A[right]);
    if (A[middle] > A[right])
        Swap(&A[middle], &A[right]);

    // Now we have: A[left] <= A[middle] <= A[right]
    Swap(&A[middle], &A[right - 1]); // Hide pivot
    return A[right - 1];             // Return pivot
}

template <typename T> static void QSortMedianOf3(T A[], size_t left, size_t right) {
    size_t i, j;
    T pivot;

    if (left + Cutoff <= right) {
        pivot = medianOf3(A, left, right);
        i = left;
        j = right - 1;

        while (1) {
            while (A[++i] < pivot);
            while (A[--j] > pivot);

            if (i < j)
                Swap(&A[i], &A[j]);
            else
                break;
        }

        Swap(&A[i], &A[right - 1]); // Renew pivot
        QSortMedianOf3(A, left, i - 1);
        QSortMedianOf3(A, i + 1, right);
    } else {
        // If subarray is small sort it via Insertion sort
        InsertionSort(A + left, right - left + 1);
    }
}
```