Binary search: o(logn)

The advantage of a binary search over a linear search is astounding for large numbers. For an array of a million elements, binary search, O(log N), will find the target element with a worst case of only 20 comparisons. Linear search, O(N), on average will take 500,000 comparisons to find the element. Probably the only faster kind of search uses *hashing*, a topic that isn't covered in these notes.

This performance comes at a price - the array must be sorted first. Because sorting isn't a fast operation, O(N log N), it may not be worth the effort to sort when there are only a few searches.

```
//============================================================ binarySearch
/** Binary search of sorted array.  Negative value on search failure.
 *  The upperbound index is not included in the search.
 *  This is to be consistent with the way Java in general expresses ranges.
 *  The performance is O(log N).
 *  @param sorted Array of sorted values to be searched.
 *  @param first Index of first element to serach, sorted[first].
 *  @param upto  Index of last element to search, sorted[upto-1].
 *  @param key   Value that is being looked for.
 *  @return      Returns index of the first match, or or -insertion_position
 *               -1 if key is not in the array. This value can easily be
 *               transformed into the position to insert it.
 */
public static int binarySearch(int[] sorted, int first, int upto, int key) {

   while (first < upto) {
      int mid = (first + upto) / 2;  // Compute mid point.
      if (key < sorted[mid]) {
         upto = mid;     // repeat search in bottom half.
      } else if (key > sorted[mid]) {
         first = mid + 1;  // Repeat search in top half.
      } else {
         return mid;     // Found it. return position
      }
   }
   return -(first + 1);    // Failed to find key
}
```

Recursive Binary Search

Iterative algorithms, ie those with a loop, can usually be easily rewritten to use recursive method calls instead of loops. However the iterative version is usually simpler, faster, and uses less memory.

Some problems, eg traversing a tree, are better solved recursively because the recursive solution is so clear (eg, binary tree traversal). Iterative binary search is more efficient than the recursive form, but it is one of the more plausible algorithms to use as an illustration of recursion.

This recursive version checks to see if we're at the key (in which case it can return), otherwise it calls itself to solve a smaller problem, ie, either the upper or lower half of the array.

```
/** Recursive binary search of sorted array.    Negative value on failure.
 *  The upperbound index is not included in the search.
 *  This is to be consistent with the way Java in general expresses ranges.
 *  The performance is O(log N).
 *  @param sorted Array of sorted values to be searched.
 *  @param first Index of beginning of range.  First element is a[first].
 *  @param upto  Last element that is searched is sorted[upto-1].
 *  @param key   Value that is being looked for.
 *  @return      Returns index of the first match, or or -insertion_position
 *               -1 if key is not in the array. This value can easily be
 *               transformed into the position to insert it.
 */
public static int rBinarySearch(int[] sorted, int first, int upto, int key) {

    if (first < upto) {
        int mid = first + (upto - first) / 2;  // Compute mid point.
        if (key < sorted[mid]) {
            return rBinarySearch(sorted, first, mid, key);

        } else if (key > sorted[mid]) {
            return rBinarySearch(sorted, mid+1, upto , key);

        } else {
            return mid;   // Found it.
        }
    }
    return -(first + 1);  // Failed to find key
}
```

Comparsion: iterative VS recursive

generally iterative is much faster becuase you don't need to set a stack frame, you don't need to keep re-allocating variables on the stack, and you don't need to pass any arguments, it also uses up less memory on top of all that.

Notes: stack, call stack

Call Stacks, Recursion, and Stack Frames

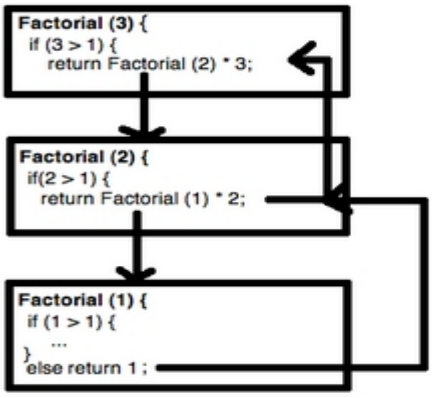
A call stack is a data structure used by the program to store information about the active subroutines (like functions in C++ or methods in Java) in a program. The main reason for having a call stack is so that the program can keep track of where a subroutine should return control to once it finishes executing. For example, suppose we have a method “CreateBox” which calls another method “CreateLine” in 4 different places. If the program has finished executing the method CreateLine, then it needs to know where in the CreateBox method it needs to return to. This is why the program uses a call stack – so that it can keep track of these details.

A call stack is composed of stack frames

A stack frame is a part of the call stack, and a new stack frame is created every time a subroutine is called. So, in our recursive Factorial method above, a new stack frame is created every time the method is called. The stack frame is used to store all of the variables for one invocation of a routine.

Stack Frames in Recursion

A diagram of how the stack frames work in recursion will really help to clarify things – so let’s take a look at one. Lets suppose that we try to find the factorial of “3″ using the function that we created above (so “x” is equal to 3), this is how it would look with the stack frames:



You can see that the first stack frame is created with x equal to 3. And then a call to Factorial(2) is made – so the 1st call to “Factorial” does not run to completion because another call is made before the current call to Factorial can run to completion. A stack frame is used to “hold” the “state” of the first call to Factorial – it will store the variables (and their values) of the current invocation of Factorial, and it will also store the return address of the method that called it. This way, it knows where to return to when it finishes running.

Finally, in the 3rd stack frame, we run into our base case, which means the recursive calls are finished and then control is returned to the 2nd stack frame, where Factorial(1) * 2 is calculated to be 2, and then control is returned to the very first stack frame. Finally, our result of “6″ is returned.

Sorting algorithms:

Bubble sort

Bubble Sort

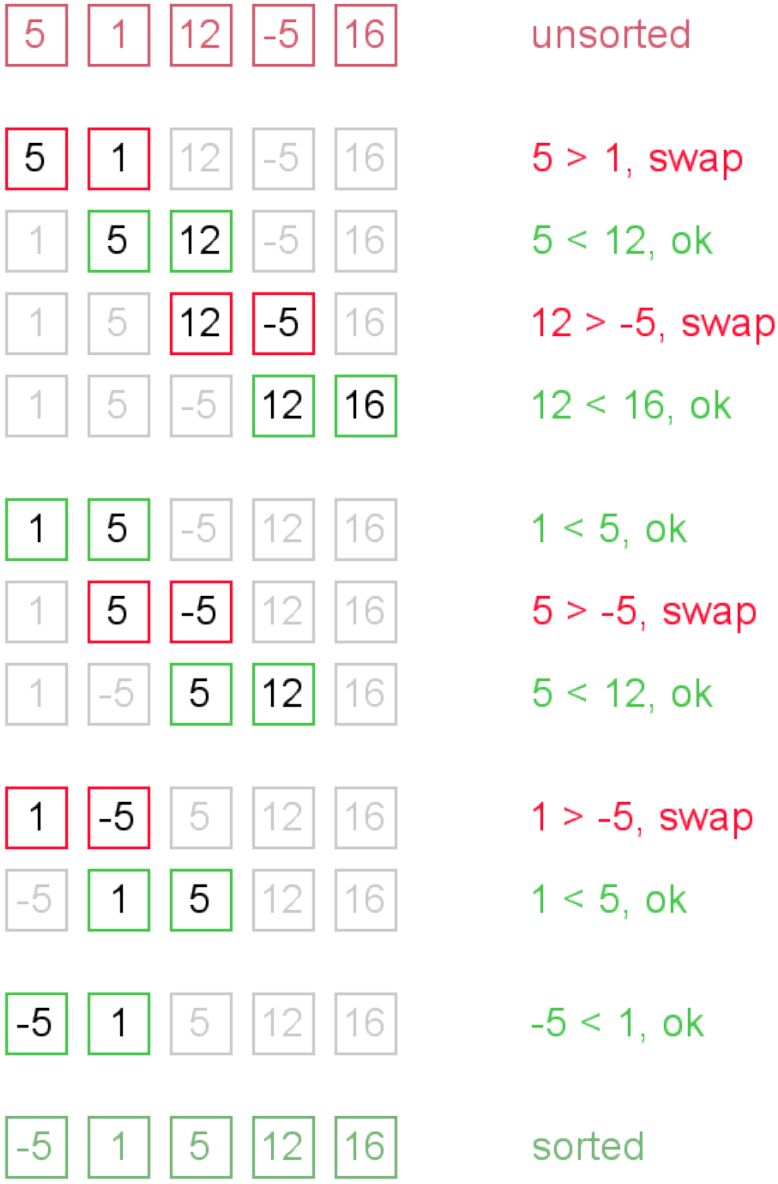
Bubble sort is a simple and well-known sorting algorithm. It is used in practice once in a blue moon and its main application is to make an introduction to the sorting algorithms. Bubble sort belongs to $O(n^2)$ sorting algorithms, which makes it quite inefficient for sorting large data volumes. Bubble sort is stable and adaptive.

Algorithm

1. Compare each pair of adjacent elements from the beginning of an array and, if they are in reversed order, swap them.
2. If at least one swap has been done, repeat step 1.

You can imagine that on every step big bubbles float to the surface and stay there. At the step, when no bubble moves, sorting stops. Let us see an example of sorting an array to make the idea of bubble sort clearer.

Example. Sort {5, 1, 12, -5, 16} using bubble sort.



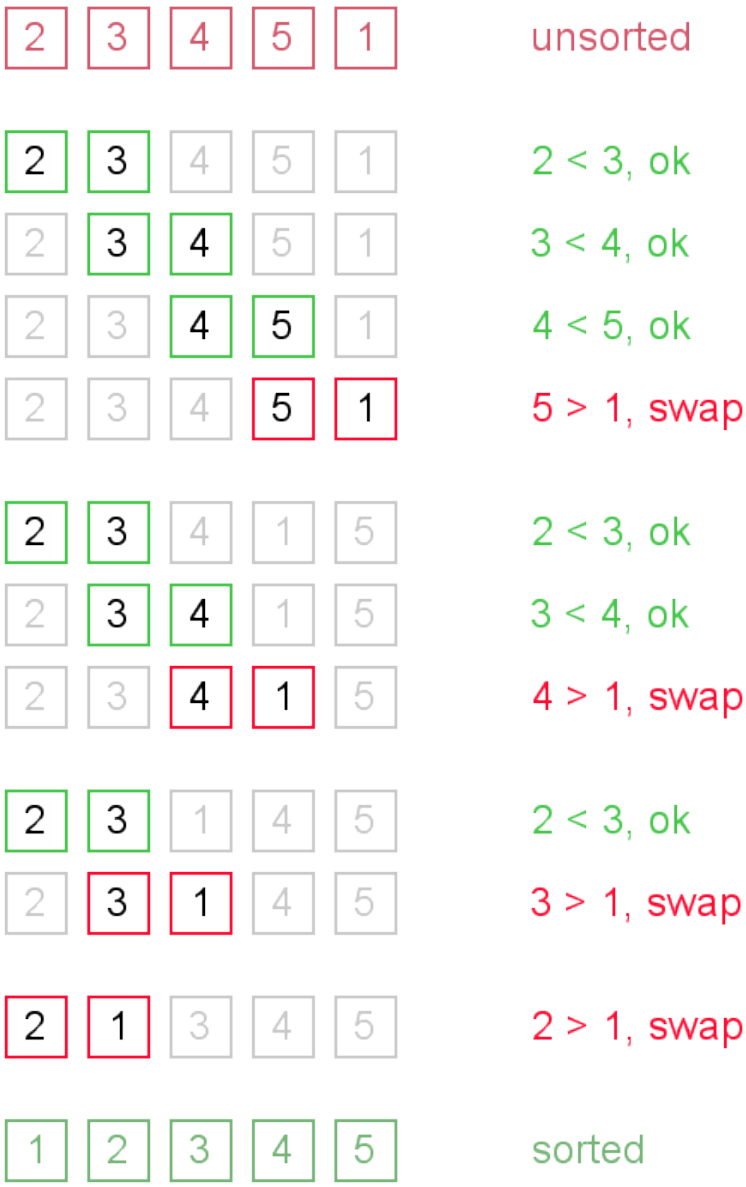
Complexity analysis

Average and worst case complexity of bubble sort is $O(n^2)$. Also, it makes $O(n^2)$ swaps in the worst case. Bubble sort is adaptive. It means that for almost sorted array it gives O(n) estimation. Avoid implementations, which don't check if the array is already sorted on every step (any swaps made). This check is necessary, in order to preserve adaptive property.

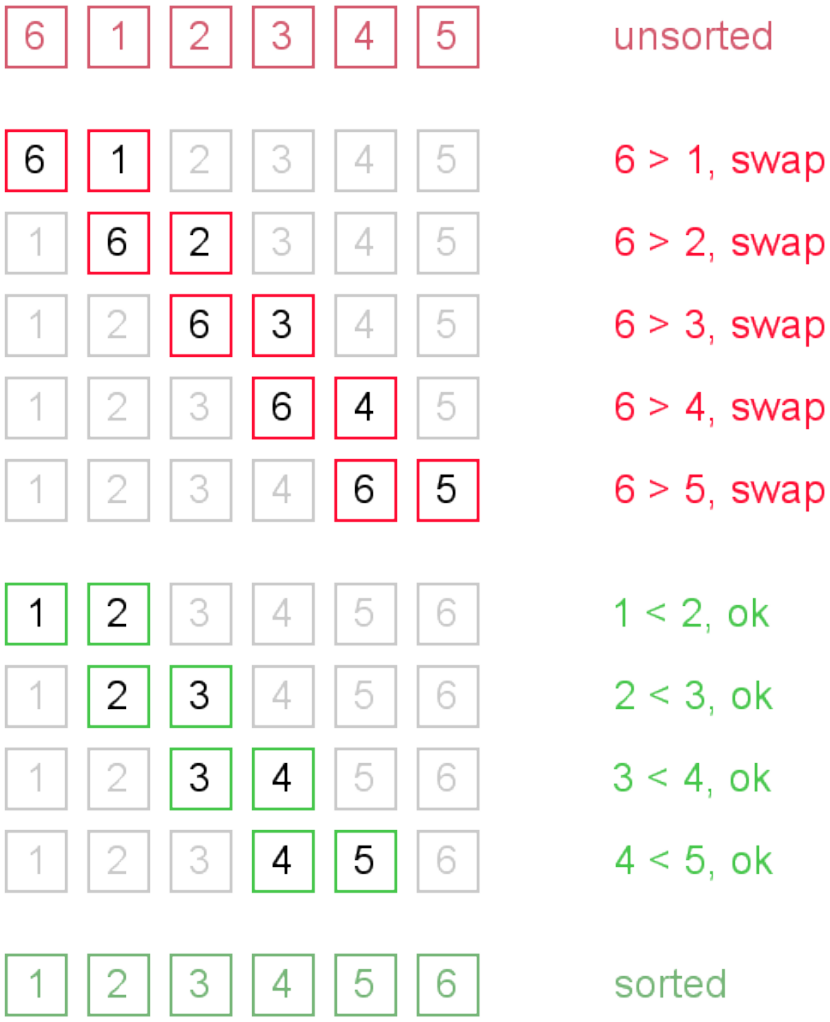
Turtles and rabbits

One more problem of bubble sort is that its running time badly depends on the initial order of the elements. Big elements (rabbits) go up fast, while small ones (turtles) go down very slow. This problem is solved in the Cocktailsort.

Turtle example. Thought, array {2, 3, 4, 5, 1} is almost sorted, it takes $O(n^2)$ iterations to sort an array. Element {1} is a turtle.



Rabbit example. Array {6, 1, 2, 3, 4, 5} is almost sorted too, but it takes O(n) iterations to sort it. Element {6} is a rabbit. This example demonstrates adaptive property of the bubble sort.



Code snippets

There are several ways to implement the bubble sort. Notice, that "swaps" check is absolutely necessary, in order to preserve adaptive property.

Java

```
public void bubbleSort(int[] arr) {
    boolean swapped = true;
    int j = 0;
    int tmp;
    while (swapped) {
        swapped = false;
        j++;
        for (int i = 0; i < arr.length - j; i++) {
            if (arr[i] > arr[i + 1]) {
                tmp = arr[i];
                arr[i] = arr[i + 1];
                arr[i + 1] = tmp;
```

```
                swapped = true;
            }
        }
    }
}
```

Selection Sort

Selection sort is one of the $O(n^2)$ sorting algorithms, which makes it quite inefficient for sorting large data volumes. Selection sort is notable for its programming simplicity and it can over perform other sorts in certain situations (see complexity analysis for more details).

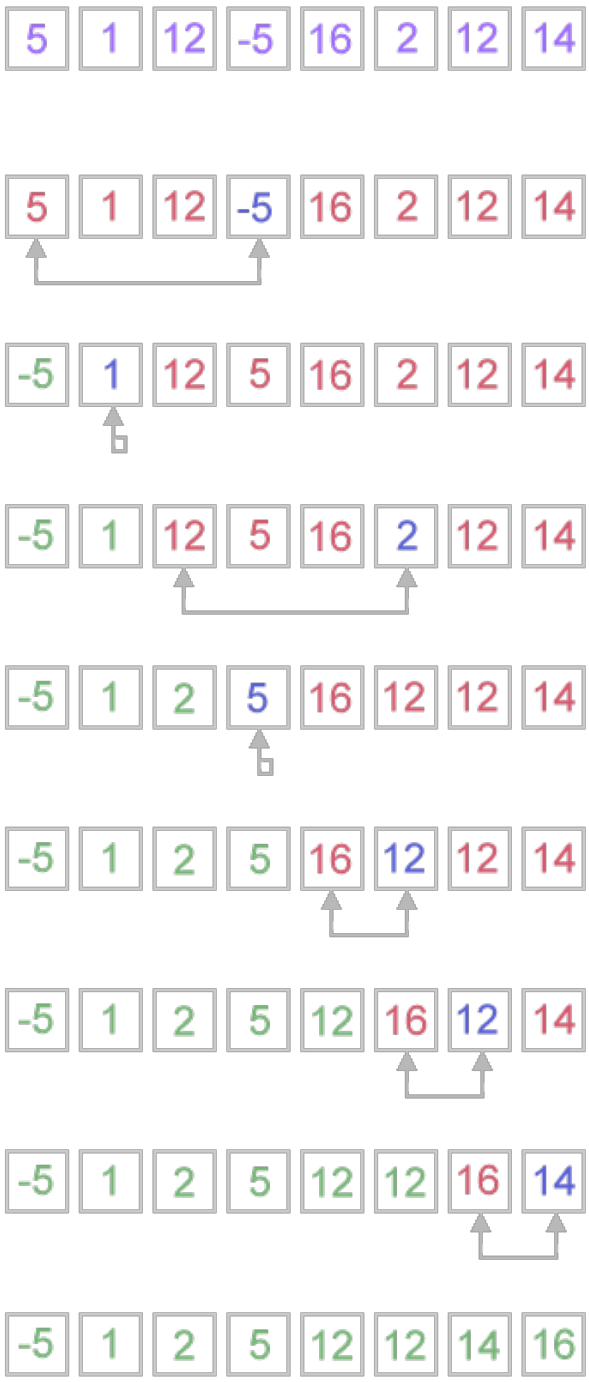
Algorithm

The idea of algorithm is quite simple. Array is imaginary divided into two parts - **sorted one** and **unsorted one**. At the beginning, **sorted part** is empty, while **unsorted one** contains whole array. At every step, algorithm finds minimal element in the **unsorted part** and adds it to the end of the **sorted one**. When **unsorted part** becomes empty, algorithm stops.

When algorithm sorts an array, it swaps first element of unsorted part with minimal element and then it is included to the sorted part. This implementation of selection sort in not stable. In case of linked list is sorted, and, instead of swaps, minimal element is linked to the unsorted part, selection sort is stable.

Let us see an example of sorting an array to make the idea of selection sort clearer.

Example. Sort {5, 1, 12, -5, 16, 2, 12, 14} using selection sort.



Complexity analysis

Selection sort stops, when unsorted part becomes empty. As we know, on every step number of unsorted elements decreased by one. Therefore, selection sort makes n steps (n is number of elements in array) of outer loop, before stop. Every step of outer loop requires finding minimum in unsorted part. Summing up, n + (n - 1) + (n - 2) + ... + 1, results in $O(n^2)$ number of comparisons. Number of swaps may vary from zero (in case of sorted array) to n - 1 (in case array was sorted in reversed order), which results in O(n) number of swaps. Overall algorithm complexity is $O(n^2)$.

Fact, that selection sort requires n - 1 number of swaps at most, makes it very efficient in situations, when write operation is significantly more expensive, than read operation.

Code snippets

Java

```
public void selectionSort(int[] arr) {
    int i, j, minIndex, tmp;
    int n = arr.length;
    for (i = 0; i < n - 1; i++) {
        minIndex = i;
        for (j = i + 1; j < n; j++)
            if (arr[j] < arr[minIndex])
                minIndex = j;
        if (minIndex != i) {
            tmp = arr[i];
            arr[i] = arr[minIndex];
            arr[minIndex] = tmp;
        }
    }
}
```

Insertion Sort

Insertion sort belongs to the O(n^2) sorting algorithms. Unlike many sorting algorithms with quadratic complexity, it is actually applied in practice for sorting small arrays of data. For instance, it is used to improve quicksort routine. Some sources notice, that people use same algorithm ordering items, for example, hand of cards.

Algorithm

Insertion sort algorithm somewhat resembles selection sort. Array is imaginary divided into two parts - **sorted one**and **unsorted one**. At the beginning, **sorted part** contains first element of the array and **unsorted one** contains the rest. At every step, algorithm takes first element in the **unsorted part** and inserts it to the right place of the **sorted one.** When **unsorted part** becomes empty, algorithm stops. Sketchy, insertion sort algorithm step looks like this:

becomes

The idea of the sketch was originaly posted here.

Let us see an example of insertion sort routine to make the idea of algorithm clearer.

Example. Sort {7, -5, 2, 16, 4} using insertion sort.

The ideas of insertion

The main operation of the algorithm is insertion. The task is to insert a value into the sorted part of the array. Let us see the variants of how we can do it.

"Sifting down" using swaps

The simplest way to insert next element into the sorted part is to sift it down, until it occupies correct position. Initiallythe element stays right after the sorted part. At each step algorithm compares the element with one before it and, if they stay in reversed order, swap them. Let us see an illustration.

This approach writes sifted element to temporary position many times. Next implementation eliminates those unnecessary writes.

Shifting instead of swapping

We can modify previous algorithm, so it will write sifted element only to the final correct position. Let us see an illustration.

It is the most commonly used modification of the insertion sort.

Using binary search

It is reasonable to use binary search algorithm to find a proper place for insertion. This variant of the insertion sort is called binary insertion sort. After position for insertion is found, algorithm shifts the part of the array and inserts the element. This version has lower number of comparisons, but overall average complexity remains $O(n^2)$. From a practical point of view this improvement is not very important, because insertion sort is used on quite small data sets.

Complexity analysis

Insertion sort's overall complexity is $O(n^2)$ on average, regardless of the method of insertion. On the almost sorted arrays insertion sort shows better performance, up to O(n) in case of applying insertion sort to a sorted array. Number of writes is $O(n^2)$ on average, but number of comparisons may vary depending on the insertion algorithm. It is $O(n^2)$ when shifting or swapping methods are used and O(n log n) for binary insertion sort.

From the point of view of practical application, an average complexity of the insertion sort is not so important. As it was mentioned above, insertion sort is applied to quite small data sets (from 8 to 12 elements). Therefore, first of all, a "practical performance" should be considered. In practice insertion sort outperforms most of the quadratic sorting algorithms, like selection sort or bubble sort.

Insertion sort properties

- adaptive (performance adapts to the initial order of elements);
- stable (insertion sort retains relative order of the same elements);
- in-place (requires constant amount of additional space);
- online (new elements can be added during the sort).

Code snippets

We show the idea of insertion with shifts in Java implementation and the idea of insertion using swaps in the C++ code snippet.

Java implementation

```
void insertionSort(int[] arr) {
    int i, j, newValue;
    for (i = 1; i < arr.length; i++) {
        newValue = arr[i];
        j = i;
        while (j > 0 && arr[j - 1] > newValue) {
            arr[j] = arr[j - 1];
            j--;
        }
        arr[j] = newValue;
    }
}
```

Quicksort

Quicksort is a fast sorting algorithm, which is used not only for educational purposes, but widely applied in practice. On the average, it has **O(n log n**) complexity, making quicksort suitable for sorting big data volumes. The idea of the algorithm is quite simple and once you realize it, you can write quicksort as fast as bubble sort.

Algorithm

Choosing the pivot is an essential step.

Depending on the pivot the algorithm may run very fast, or in quadric time. **:**

1. Some fixed element: e.g. the first, the last, the one in the middle

   This is a bad choice - the pivot may turn to be the smallest or the largest element,
   then one of the partitions will be empty.

2. Randomly chosen (by random generator) - still a bad choice.
3. The median of the array (if the array has N numbers, the median is the [N/2] largest number. This is difficult to compute - increases the complexity.
4. The median-of-three choice: take the first, the last and the middle element.
   Choose the median of these three elements.

Example:

8, 3, 25, 6, 10, 17, 1, 2, 18, 5

The first element is 8, the middle - 10, the last - 5.
The median of [8, 10, 5] is **8**

The divide-and-conquer strategy is used in quicksort. Below the recursion step is described:

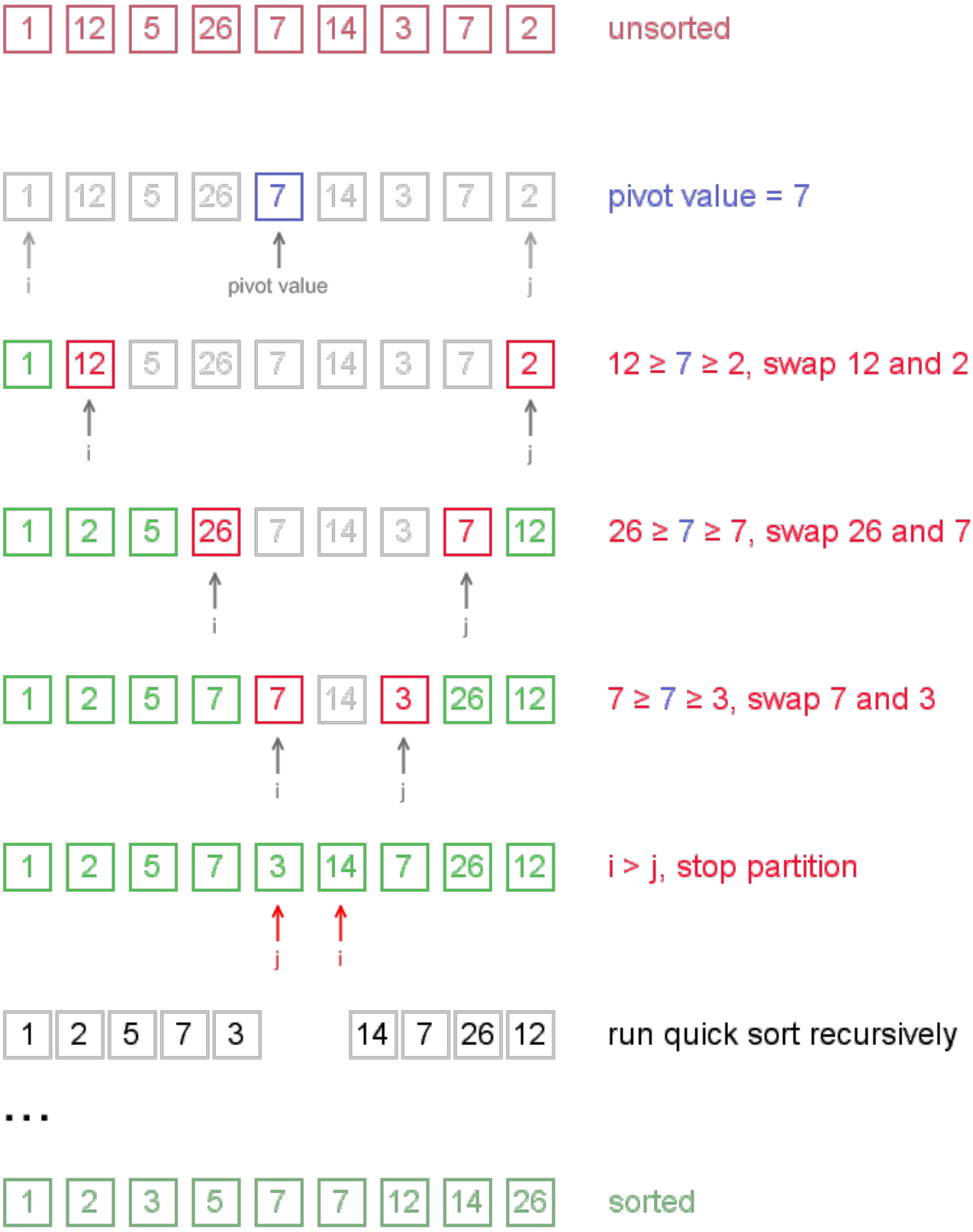
1. Choose a pivot value. We take the value of the middle element as pivot value, but it can be any value, which is in range of sorted values, even if it doesn't present in the array.
2. Partition. Rearrange elements in such a way, that all elements which are lesser than the pivot go to the left part of the array and all elements greater than the pivot, go to the right part of the array. Values equal to the pivot can stay in any part of the array. Notice, that array may be divided in non-equal parts.
3. Sort both parts. Apply quicksort algorithm recursively to the left and the right parts.

Partition algorithm in detail

There are two indices i and j and at the very beginning of the partition algorithm i points to the first element in the array and j points to the last one. Then algorithm moves i forward, until an element with value greater or equal to the pivot is found. Index j is moved backward, until an element with value lesser or equal to the pivot is found. If $i \leq j$then they are swapped and i steps to the next position (i + 1), j steps to the previous one (j - 1). Algorithm stops, when i becomes greater than j.

After partition, all values before i-th element are less or equal than the pivot and all values after j-th element are greater or equal to the pivot.

Example. Sort {1, 12, 5, 26, 7, 14, 3, 7, 2} using quicksort.



Notice, that we show here only the first recursion step, in order not to make example too long. But, in fact, {1, 2, 5, 7, 3} and {14, 7, 26, 12} are sorted then recursively.

Why does it work?

On the partition step algorithm divides the array into two parts and every element **a** from the left part is less or equal than every element **b** from the right part. Also **a** and **b** satisfy **a ≤ pivot ≤ b** inequality. After completion of the recursion calls both of the parts become sorted and, taking into account arguments stated above, the whole array is sorted.

Complexity analysis

On the average quicksort has O(n log n) complexity, but strong proof of this fact is not trivial and not presented here. Still, you can find the proof in [1]. In worst case, quicksort runs $O(n^2)$ time, but on the most "practical" data it works just fine and outperforms other O(n log n) sorting algorithms.

```
int partition(int arr[], int left, int right)
{
    int i = left, j = right;
    int tmp;
    int pivot = arr[(left + right) / 2];

    while (i <= j) {
        while (arr[i] < pivot)
            i++;
        while (arr[j] > pivot)
            j--;
        if (i <= j) {
            tmp = arr[i];
            arr[i] = arr[j];
            arr[j] = tmp;
            i++;
            j--;
        }
    };

    return i;
}

void quickSort(int arr[], int left, int right) {
    int index = partition(arr, left, right);
    if (left < index - 1)
```

```
        quickSort(arr, left, index - 1);
    if (index < right)
        quickSort(arr, index, right);
}
```