# Sorting

Data structures and Algorithms

---

**Outline**

- Bubble Sort
- Insertion Sort
- Merge Sort
- Quick Sort

---

## Bubble Sort

**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.

---

<table>
<thead>
<tr>
<th>1st pass</th>
<th>2nd pass</th>
<th>3rd pass</th>
<th>4th pass</th>
<th>unsorted</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 1 12 -5 16</td>
<td>5 1 12 -5 18</td>
<td>1 5 12 -5 18</td>
<td>1 5 -5 12 16</td>
<td>5 &gt; 1, swap</td>
</tr>
<tr>
<td></td>
<td>6 &lt; 12, ok</td>
<td>12 &gt; -5, swap</td>
<td>12 &lt; 16, ok</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 &lt; 5, ok</td>
<td>6 &gt; -5, swap</td>
<td>5 &lt; 12, ok</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 &gt; -5, swap</td>
<td>1 &lt; 5, ok</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-5 &lt; 1, ok</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>sorted</td>
<td></td>
</tr>
</tbody>
</table>

Reference: [http://www.algolist.net/Algorithms/Sorting/Bubble_sort](http://www.algolist.net/Algorithms/Sorting/Bubble_sort)
Bubble Sort pseudocode

Algorithm `bubbleSort(S, C)`
Input sequence `S` with `n` elements, comparator `C`
Output sequence `S` sorted according to `C`
do
swapped ← false
for each `i` in 1 to length(`S`) – 1 inclusive do:
  if `S[i - 1]` > `S[i]` according to `C` then
    `swap(S[i - 1], S[i])`
    swapped ← true
while swapped

Bubble Sort performance
- Worst-case & average-case: O(n^2)
- Best-case: (over an already-sorted list) :O(n)

Insertion Sort

For pass 1 ... n-1:
  for `j` in 1..n-pass
    if `S[j-1]` > `S[j]`:
      `swap(s[j-1], s[j])`

For i in 1 ... n-1:
  for `j` in 1..n-i
    if `S[j-1]` > `S[j]`:
      `swap(s[j-1], s[j])`

Reference: [http://www.algolist.net/Algorithms/Sorting/Insertion_sort](http://www.algolist.net/Algorithms/Sorting/Insertion_sort)
**Insertion Sort pseudocode**

Algorithm `insertionSort(S, C)`
- **Input** sequence `S` with `n` elements, comparator `C`
- **Output** sequence `S` sorted according to `C`

```
for i from 1 to length(S) do
    j ← i
    while j > 0 && S[j - 1] > S[j] then
        swap(S[j - 1], S[j])
        j--

sorted
```

**Insertion Sort performance**
- Worst-case & average-case: $O(n^2)$
- Best-case: (over an already-sorted list) :$O(n)$
- 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).

**Divide-and-Conquer**
- Divide-and-conquer is a general algorithm design paradigm:
  - **Divide:** divide the input data `S` in two or more disjoint subsets `S_1, S_2, ...`
  - **Recur:** solve the subproblems recursively
  - **Conquer:** combine the solutions for `S_1, S_2, ...`, into a solution for `S`
- The base case for the recursion are subproblems of constant size
- Analysis can be done using recurrence equations
Merge Sort

Merge-Sort

- Merge-sort on an input sequence $S$ with $n$ elements consists of three steps:
  - **Divide:** partition $S$ into two sequences $S_1$ and $S_2$ of about $n/2$ elements each.
  - **Recur:** recursively sort $S_1$ and $S_2$.
  - **Conquer:** merge $S_1$ and $S_2$ into a unique sorted sequence.

Merging Two Sorted Sequences

- The conquer step of merge-sort consists of merging two sorted sequences $A$ and $B$ into a sorted sequence $S$ containing the union of the elements of $A$ and $B$.
- Merging two sorted sequences, each of size $n/2$ elements and implemented by means of a doubly linked list, takes $O(n)$ time.

Merge-Sort Tree

- An execution of merge-sort is depicted by a binary tree:
  - each node represents a recursive call of merge-sort and stores:
  - the unsorted sequence before the execution and its partition.
  - the sorted sequence at the end of the execution.
  - the root is the initial call.
  - the leaves are calls on subsequences of size 0 or 1.
Execution Example (cont.)

Recursive call, partition

Execution Example (cont.)

Recursive call, base case
Execution Example (cont.)

- Recursive call, base case

```
7 2 9 4 | 3 8 6 1
```

```
7 2 9 4
7 2
7 → 7 2 → 2
```

```
Sorting
21
```

Execution Example (cont.)

- Merge

```
7 2 9 4 | 3 8 6 1
```

```
7 2 9 4
7 2
7 → 7 2 → 2
```

```
Sorting
22
```

Execution Example (cont.)

- Recursive call, ..., base case, merge

```
7 2 9 4 | 3 8 6 1
```

```
7 2 9 4
7 2 2 7
9 4 4 9
```

```
Sorting
23
```

Execution Example (cont.)

- Merge

```
7 2 9 4 | 3 8 6 1
```

```
7 2 9 4
7 2 2 7
9 4 4 9
```

```
Sorting
24
```
Sorting 28

Merge(S1, S2) takes time O(n), where n is the size of S1 and S2.

Analysis of Merge-Sort using Recurrence Relations

- Merge(S1, S2) takes time O(n), where n is the size of S1 and S2.
- T(n) = 2T(n/2) + O(n)
- Solving, get T(n) = O(n log n)

Algorithm mergeSort(S, C)
Input sequence S with n elements, comparator C
Output sequence S sorted according to C
if S.size() > 1
(S1, S2) ← partition(S, n/2)
mergeSort(S1, C)
mergeSort(S2, C)
S ← merge(S1, S2)

Execution Example (cont.)
Recursive call, ..., merge, merge

Execution Example (cont.)
Merge

Analysis of Merge-Sort
- The height h of the merge-sort tree is O(log n)
  - at each recursive call we divide in half the sequence,
  - The overall amount or work done at the nodes of depth i is O(n)
    - we partition and merge 2^i sequences of size n/2^i
    - we make 2^i-1 recursive calls
  - Thus, the total running time of merge-sort is O(n log n)
Quick-Sort

Quick-Sort is a randomized sorting algorithm based on the divide-and-conquer paradigm:
- **Divide**: pick a random element \( x \) (called pivot) and partition \( S \) into
  - \( L \) elements less than \( x \)
  - \( E \) elements equal to \( x \)
  - \( G \) elements greater than \( x \)
- **Recur**: sort \( L \) and \( G \)
- **Conquer**: join \( L \), \( E \) and \( G \)

**Partition**

- We partition an input sequence as follows:
  - We remove, in turn, each element \( y \) from \( S \) and
  - We insert \( y \) into \( L \), \( E \) or \( G \), depending on the result of the comparison with the pivot \( x \)
- Each insertion and removal is at the beginning or at the end of a sequence, and hence takes \( O(1) \) time
- Thus, the partition step of quick-sort takes \( O(n) \) time

**Algorithm** `partition(S, p)`

```
Input sequence \( S \), position \( p \) of pivot
Output subsequences \( L, E, G \) of the elements of \( S \) less than, equal to, or greater than the pivot, resp.

\( L, E, G \leftarrow \) empty sequences
\( x \leftarrow S.remove(p) \)

while \(!S.isEmpty()\)
  \( y \leftarrow S.remove(S.first()) \)
  if \( y < x \)
    \( L.insertLast(y) \)
  else if \( y = x \)
    \( E.insertLast(y) \)
  else \( y > x \)
    \( G.insertLast(y) \)

return \( L, E, G \)
```

Quick-Sort Tree

An execution of quick-sort is depicted by a binary tree:
- Each node represents a recursive call of quick-sort and stores
  - Unsorted sequence before the execution and its pivot
  - Sorted sequence at the end of the execution
- The root is the initial call
- The leaves are calls on subsequences of size 0 or 1
Execution Example

- Pivot selection

```
7 2 9 4 3 7 6 1
```

Execution Example (cont.)

- Partition, recursive call, pivot selection

```
7 2 9 4 3 7 6 1
```

Execution Example (cont.)

- Partition, recursive call, base case

```
7 2 9 4 3 7 6 1
```

Execution Example (cont.)

- Recursive call, ..., base case, join

```
7 2 9 4 3 7 6 1
```
Execution Example (cont.)

- **Recursive call, pivot selection**
  - Initial array: \(72943761\)
  - Pivot selection:
    - \(2431 \rightarrow 1234\)
    - \(797\)
    - \(43 \rightarrow 34\)
    - \(4 \rightarrow 4\)

Execution Example (cont.)

- **Partition, ..., recursive call, base case**
  - Initial array: \(72943761\)
  - Pivot selection:
    - \(2431 \rightarrow 1234\)
    - \(797\)
    - \(1 \rightarrow 1\)
    - \(43 \rightarrow 34\)
    - \(4 \rightarrow 4\)

Execution Example (cont.)

- **Join, join**
  - Initial array: \(72943761 \rightarrow 12346779\)
  - Joining:
    - \(2431 \rightarrow 1234\)
    - \(797 \rightarrow 779\)
    - \(43 \rightarrow 34\)
    - \(4 \rightarrow 4\)

Worst-case Running Time

- The worst case for quick-sort occurs when the pivot is the unique minimum or maximum element.
- One of \(L\) and \(G\) has size \(n - 1\) and the other has size 0.
- The running time is proportional to the sum \(n + (n - 1) + \ldots + 2 + 1\).
- Thus, the worst-case running time of quick-sort is \(O(n^2)\).

Worst-case Running Time

- The worst case for quick-sort occurs when the pivot is the unique minimum or maximum element.
- One of \(L\) and \(G\) has size \(n - 1\) and the other has size 0.
- The running time is proportional to the sum \(n + (n - 1) + \ldots + 2 + 1\).
- Thus, the worst-case running time of quick-sort is \(O(n^2)\).
Expected Running Time

- Consider a recursive call of quick-sort on a sequence of size \( n \)
  - **Good call**: the sizes of \( L \) and \( G \) are each less than \( 3n/4 \)
  - **Bad call**: one of \( L \) and \( G \) has size greater than \( 3n/4 \)

<table>
<thead>
<tr>
<th>1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Good call</strong></td>
</tr>
<tr>
<td>1/2 of the possible pivots cause good calls:</td>
</tr>
<tr>
<td>1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16</td>
</tr>
<tr>
<td><strong>Bad call</strong></td>
</tr>
</tbody>
</table>

In-Place Quick-Sort

- Quick-sort can be implemented to run in-place.
- In the partition step, we use replace operations to rearrange the elements of the input sequence such that
  - the elements less than the pivot have rank less than \( h \)
  - the elements equal to the pivot have rank between \( h \) and \( k \)
  - the elements greater than the pivot have rank greater than \( k \)
- The recursive calls consider
  - elements with rank less than \( h \)
  - elements with rank greater than \( k \)

In-Place Partitioning

- Perform the partition using two indices to split \( S \) into \( L \) and \( EYG \) (a similar method can split \( EYG \) into \( E \) and \( G \)).

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

(pivot = 6)

Repeat until \( j \) and \( k \) cross:
- Scan \( j \) to the right until finding an element \( \geq x \).
- Scan \( k \) to the left until finding an element \( < x \).
- Swap elements at indices \( j \) and \( k \)

```
3 2 5 1 0 7 3 5 9 2 7 9 8 9 7 6 9  
```
## Summary of Sorting Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>bubble-sort</td>
<td>$O(n^2)$</td>
<td>in-place, slow (good for small inputs)</td>
</tr>
<tr>
<td>insertion-sort</td>
<td>$O(n^2)$</td>
<td>in-place, slow (good for small inputs)</td>
</tr>
<tr>
<td>quick-sort</td>
<td>$O(n \log n)$ expected</td>
<td>in-place, randomized, fastest (good for large inputs)</td>
</tr>
<tr>
<td>heap-sort</td>
<td>$O(n \log n)$</td>
<td>in-place, fast (good for large inputs)</td>
</tr>
<tr>
<td>merge-sort</td>
<td>$O(n \log n)$</td>
<td>sequential data access, fast (good for huge inputs)</td>
</tr>
</tbody>
</table>