Visualizing Sorting Algorithms

Leon Rische

[2017-12-10 Sun 12:00]

Contents

1 Bubble Sort **1**

2 Selection Sort **2**

3 Insertion Sort **3**

4 Quick Sort **4**

5 Heap Sort **5**

- Sorting Algorithms

Mike Bostock created some great visualizations of sorting algorithms with D3.js and I wondered how hard it would be to implement something similar from scratch.

I've generated images for five different algorithms, each sorting an array of 15 numbers in sorted, reversed and randomized order.

1 Bubble Sort

language=Ruby,label= ,caption= ,captionpos=b,numbers=none class BubbleSort < Sort def sort swapped = true

while swapped swapped = false (1...@array.size).each do |i| if @array[i - 1] > @array[i] swap(i - 1, i) swapped = true end end end end end

- Best case $\mathcal{O}(n)$
- Average case $\mathcal{O}(n^2)$
- Worst case $\mathcal{O}(n^2)$

1.1 Sorted

If the input array is already sorted, the algorithm is done after one iteration and needs so swaps.

[width=.9]images/sorting/BubbleSort-sorted

1.2 Reversed

The extreme opposite happens when the array is in reversed order, the first element has to "bubble up" to the last index of the array, the next one to next-to-last, . . .

[width=.9]images/sorting/BubbleSort-reversed

1.3 Random

[width=.9]images/sorting/BubbleSort-random

2 Selection Sort

language=Ruby,label= ,caption= ,captionpos=b,numbers=none class SelectionSort < Sort def sort (0...(@array.size - 1)).each do |j| i$_min = j((j + 1)...@array.size).eachdo|i|i_min = iif@array[i] < @array[i_min]end$
swap(j, i$_min)ifi_min! = jendendend$

- Best case $\mathcal{O}(n^2)$
- Average case $\mathcal{O}(n^2)$
- Worst case $\mathcal{O}(n^2)$

Selection sort iterates over the array, looks for the smallest element in the rest of the array and swaps it to the current position.

2.1 Sorted

Just like bubble sort, no swaps are used for a sorted array.

[width=.9]images/sorting/SelectionSort-sorted

2.2 Reversed

If the array is in reversed order, the last element is swapped to the first position, the next-to-last to the second position, . . . , leading to a nice triangle shape.

[width=.9]images/sorting/SelectionSort-reversed

2.3 Random

Just from looking at these images, it might seem like selection sort is way better than bubble sort because the images are smaller. In the real world, the performance of a sorting algorithms depends on the number of comparisons made, too, while the size of these images depends only on the number of swaps.

[width=.9]images/sorting/SelectionSort-random

3 Insertion Sort

language=Ruby,label= ,caption= ,captionpos=b,numbers=none class InsertionSort < Sort def sort i = 1 while i < @array.size j = i while j > 0 @array[j - 1] > @array[j] swap(j, j - 1) j -= 1 end i += 1 end end end

- Best case $\mathcal{O}(n)$
- Average case $\mathcal{O}(n^2)$
- Worst case $\mathcal{O}(n^2)$

3.1 Sorted

[width=.9]images/sorting/InsertionSort-sorted

3.2 Reversed

[width=.9]images/sorting/InsertionSort-reversed

3.3 Random

[width=.9]images/sorting/InsertionSort-random

4 Quick Sort

language=Ruby,label= ,caption= ,captionpos=b,numbers=none class Quick-Sort < Sort def median$_{o}f_{t}hree(left, right)center = (left+right)/2[left, right, center].sort_{b}y|index|@a$

def sort(left = 0, right = @array.size - 1) return unless left < right

pivot = median$_{o}f_{t}hree(left, right)center = partition(left, right, pivot)$

sort(left, center - 1) sort(center + 1, right) end

Reorder the elements in the array so that the elements less that the pivot element are to its left and the other ones are to its right, then return the new index of the pivot element. def partition(left, right, pivot) pivot$_{v}alue = @array[pivot]swap(pivot, right)i = left - 1$

(left...right).each do |j| if @array[j] <= pivot$_{v}aluei+ = 1swap(i, j)endend$

swap(i + 1, right) i + 1 end end

- Best case $\mathcal{O}(n \log n)$
- Average case $\mathcal{O}(n \log n)$
- Worst case $\mathcal{O}(n^2)$

This is not your garden-variety quicksort, it swaps the elements in-place instead of splitting the input array into two smaller ones, recursively sorting both and merging them again into a big sorted array.

The pivot element is selected by calculating the median of the first, last and center element.

4.1 Sorted

Here you can see how the pivot elements are selected and swapped to the end of the array. Because the elements are already sorted, each one is only swapped with itself, leading to a long section of straight lines. Then the pivot element is swapped to its correct position and stays there for the rest of the steps.

If you look closely, you can see how after that the right half of the array is sorted (recursively), then the left.

[width=.9]images/sorting/QuickSort-sorted

4.2 Reversed

If the array is reversed, a lot of swaps are needed to partition the array each time.

[width=.9]images/sorting/QuickSort-reversed

4.3 Random

In the average case quick sort needs fewer steps that the previous algorithms. The worst case happens if each step the pivot element the pivot element is the biggest or smallest element of the subarray, so that one of the partitions is empty.

[width=.9]images/sorting/QuickSort-random

5 Heap Sort

This is the most complicated one of the algorithms presented here, but the one with the best worst-case complexity.

Heap sort works by reordering the elements of the array so that they form a heap (a way to store binary trees in arrays) and then searching for the smallest element in logarithmic time.

language=Ruby,label= ,caption= ,captionpos=b,numbers=none class HeapSort < Sort def sort heapify

to = @array.size - 1 while to > 0 swap(to, 0) to -= 1 sift$_down(0, to) end end$

def heapify from = index$_parent(@array.size-1) while from >= 0 sift_down(from, @array.size-1) from -= 1 end end$

def sift$_down(from, to) root = from$

while index$_left_child(root) <= to child = index_left_child(root) swap = @array[root] < @array[child] ? child : root swap = child + 1 if child + 1 <= to @array[swap] < @array[child + 1]$

return if swap == root

swap(root, swap) root = swap end end

def index$_parent(i) (i - 1)/2 end$

def index$_left_child(i) 2 * i + 1 end end$

- Best case $\mathcal{O}(n \log n)$
- Average case $\mathcal{O}(n \log n)$
- Worst case $\mathcal{O}(n \log n)$

5.1 Sorted

[width=.9]images/sorting/HeapSort-sorted

5.2 Reversed

[width=.9]images/sorting/HeapSort-reversed

5.3 Random

[width=.9]images/sorting/HeapSort-random