Common Algorithms

Todd Resudek
supersimple

What is the point of all this?

Time Complexity

In computer science, the time complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the string representing the input. The time complexity of an algorithm is commonly expressed using big O notation, which excludes coefficients and lower order terms. When expressed this way, the time complexity is said to be described asymptotically, i.e., as the input size goes to infinity. For example, if the time required by an algorithm on all inputs of size n is at most $5n^3 + 3n$ for any n (bigger than some n0), the asymptotic time complexity is $O(n^3)$.

Time complexity is commonly estimated by counting the number of elementary operations performed by the algorithm, where an elementary operation takes a fixed amount of time to perform. Thus the amount of time taken and the number of elementary operations performed by the algorithm differ by at most a constant factor.

Time Complexity

Name	Big O	Explanation
Constant	$O(1)$	It will always take the same amount of time to run.
Logarithmic	$O(\log n)$	The time to run increases at a declining rate as the size of the array increases
Linear	$O(n)$	Time increases at the same rate as size of array increases
Quadratic	$O(n^2)$	Exponential growth of time as array increases

Constant Time

```
def last_ten(arr)
 arr.last(10)
end
```

Linear Time

```
def linear(arr)
  arr.each do |val|
   puts val ** 2
  end
end
```

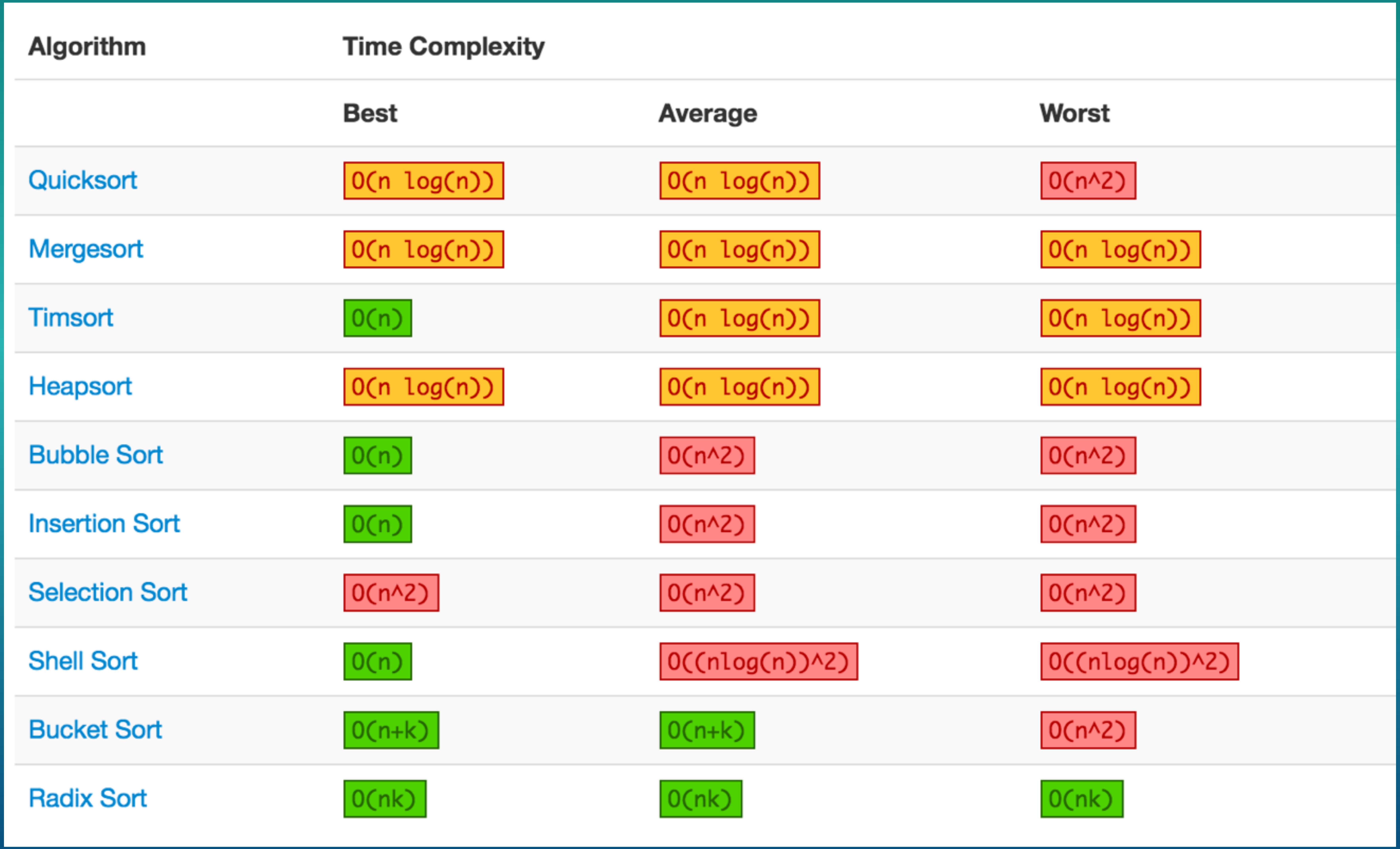
Logarithmic Time

```
def loga(str)
 if str.length > 1
  str = str[str.length/2, str.length]
   puts str
   loga(str)
 end
end
```

Quadratic Time

```
def inner_loops(two_dim_arr)
 two_dim_arr.each do |parent_arr|
  parent_arr.each do |child_arr|
   puts child_arr.attr
  end
 end
end
```

Time Complexity



Algorithm	Time Complexity		
	Best	Average	Worst
Quicksort	O(n log(n))	O(n log(n))	O(n^2)
Mergesort	O(n log(n))	O(n log(n))	O(n log(n))
Timsort	O(n)	O(n log(n))	O(n log(n))
Heapsort	O(n log(n))	O(n log(n))	O(n log(n))
Bubble Sort	O(n)	O(n^2)	O(n^2)
Insertion Sort	O(n)	O(n^2)	O(n^2)
Selection Sort	O(n^2)	O(n^2)	O(n^2)
Shell Sort	O(n)	O((nlog(n))^2)	O((nlog(n))^2)
Bucket Sort	O(n+k)	O(n+k)	O(n^2)
Radix Sort	O(nk)	O(nk)	O(nk)

Kadane’s Algorithm

Used to solve the maximum subarray problem.

The maximum subarray problem is the task of finding the contiguous subarray within a one-dimensional array of numbers which has the largest sum.

Ruby Implementation

```ruby
def max_subarray(a)
   max_so_far = max_ending_here = -1.0 / 0
   a.each do |i|
     max_ending_here = [i, max_ending_here + i].max
     max_so_far = [max_so_far, max_ending_here].max
   end
   max_so_far
 end
```

Fisher-Yates Shuffle

Starting with an initial list of values: 1 2 3 4 5 6 7 8

Choose a random index and move it’s value to the end:

1 2 4 5 6 7 8 -> 3

1 2 4 5 7 8 -> 3 6

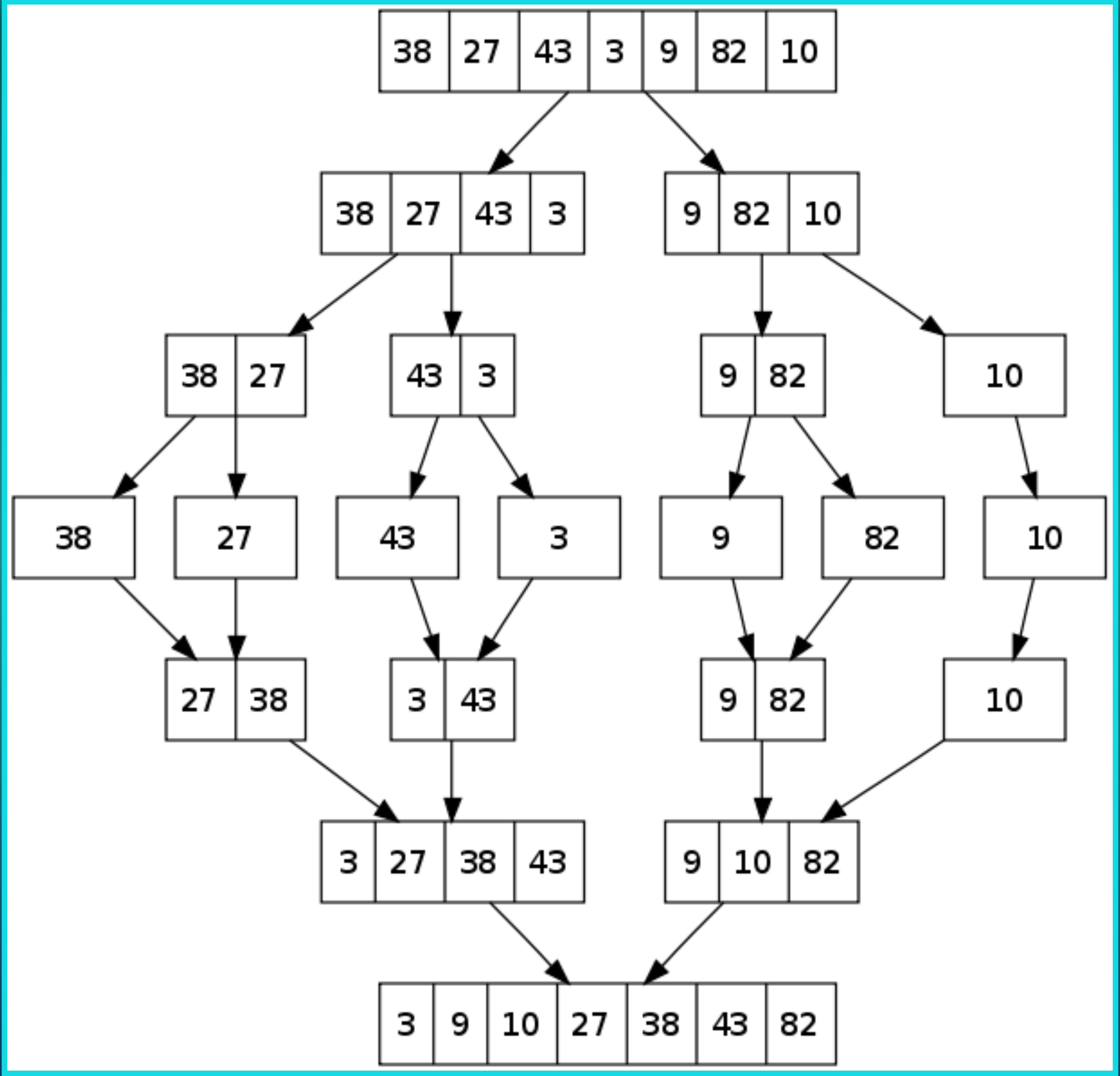
1 2 5 7 8 -> 3 6 4

Ruby Implementation

```
(length - 1).downto(1) do |i|
 j = rand(i + 1)
 self[i], self[j] = self[j], self[i]
end
```

Merge Sort

Break down array into smallest unit and merge incrementally, ordering by values in that limited group.



38 27 43 3 9 82 10
38 27 43 3
9 82 10
38 27
43 3
9 82
10
38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10 82
3 9 10 27 38 43 82

6 5 3 1 8 7 2 4

Ruby Implementation

```ruby
def merge_sort(lst)
  if lst.length <= 1
    lst
  else
    mid = (lst.length / 2).floor
    left = merge_sort(lst[0..mid - 1])
    right = merge_sort(lst[mid..lst.length])
    merge(left, right)
  end
end

def merge(left, right)
  if left.empty?
    right
  elsif right.empty?
    left
  elsif left.first < right.first
    [left.first] + merge(left[1..left.length], right)
  else
    [right.first] + merge(left, right[1..right.length])
  end
end
```

Quick Sort

Break down array into smallest unit and merge incrementally, ordering by values in that limited group.

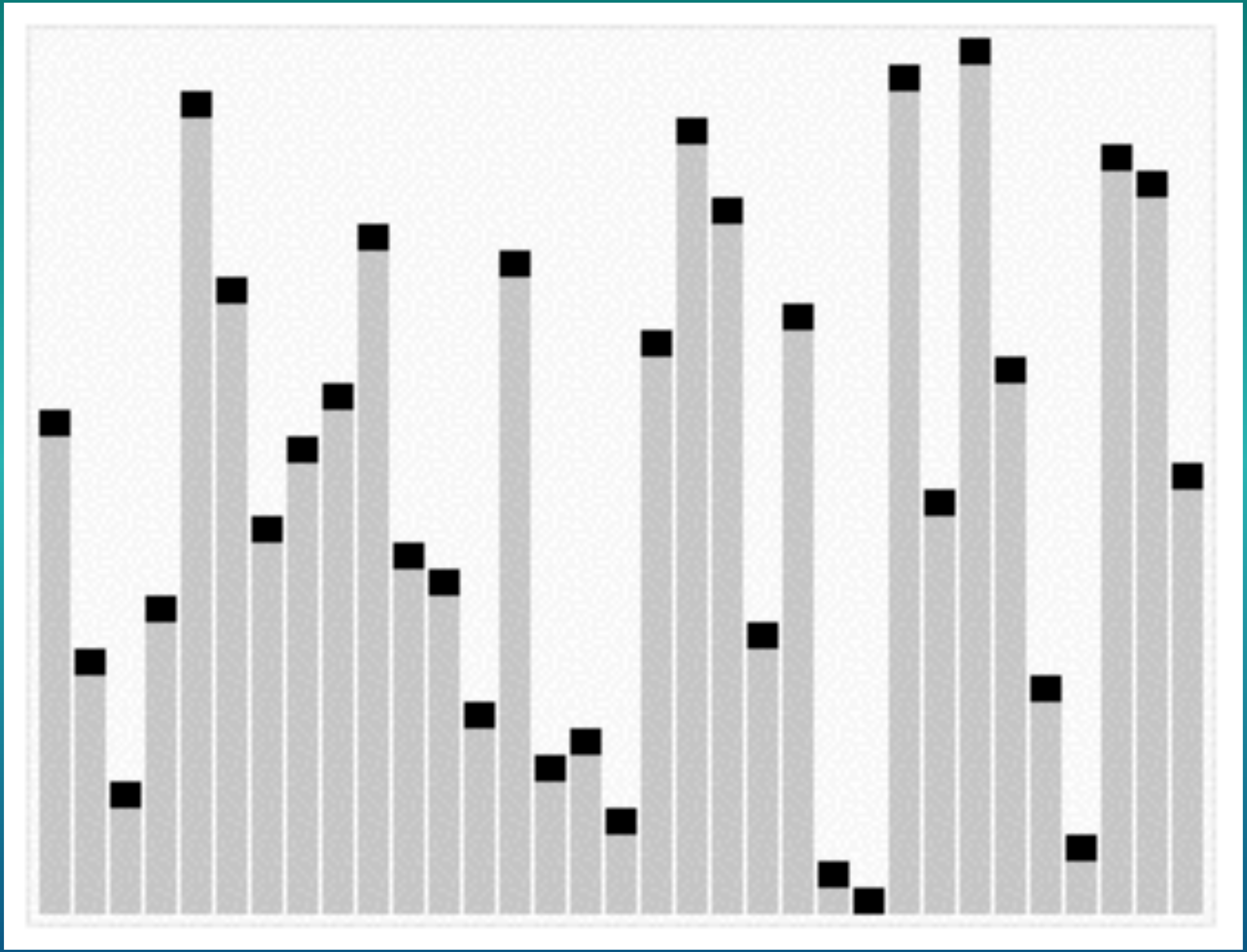
Quick Sort

Pivots

The right most item is generally the initial pivot. Other methods are to use a random pivot or the median of the first, middle, and last elements

Repeated Values

In a list with many repeated values, it is common to have a very lopsided pivot. If all values were equal, for instance, the left side would be empty and the right side would only decrease by 1 each iteration. To solve this, there is another routine that breaks the list into values lower, values higher, and values equal to the pivot. Values that are equal to the pivot are considered to be sorted already.



Ruby Implementation

```ruby
def quicksort(array)
  return array if array.length <= 1

  pivot_index = (array.length / 2).to_i
  pivot_value = array[pivot_index]
  array.delete_at(pivot_index)

  lesser = Array.new
  greater = Array.new

  array.each do |x|
    if x <= pivot_value
      lesser << x
    else
      greater << x
    end
  end

  return quicksort(lesser) + [pivot_value] + quicksort(greater)
end
```

Dijkstra’s Algorithm

Dijkstra's algorithm is an algorithm for finding the shortest paths between nodes in a graph, which may represent, for example, road networks. It was conceived by computer scientist Edsger W. Dijkstra in 1956 and published three years later.

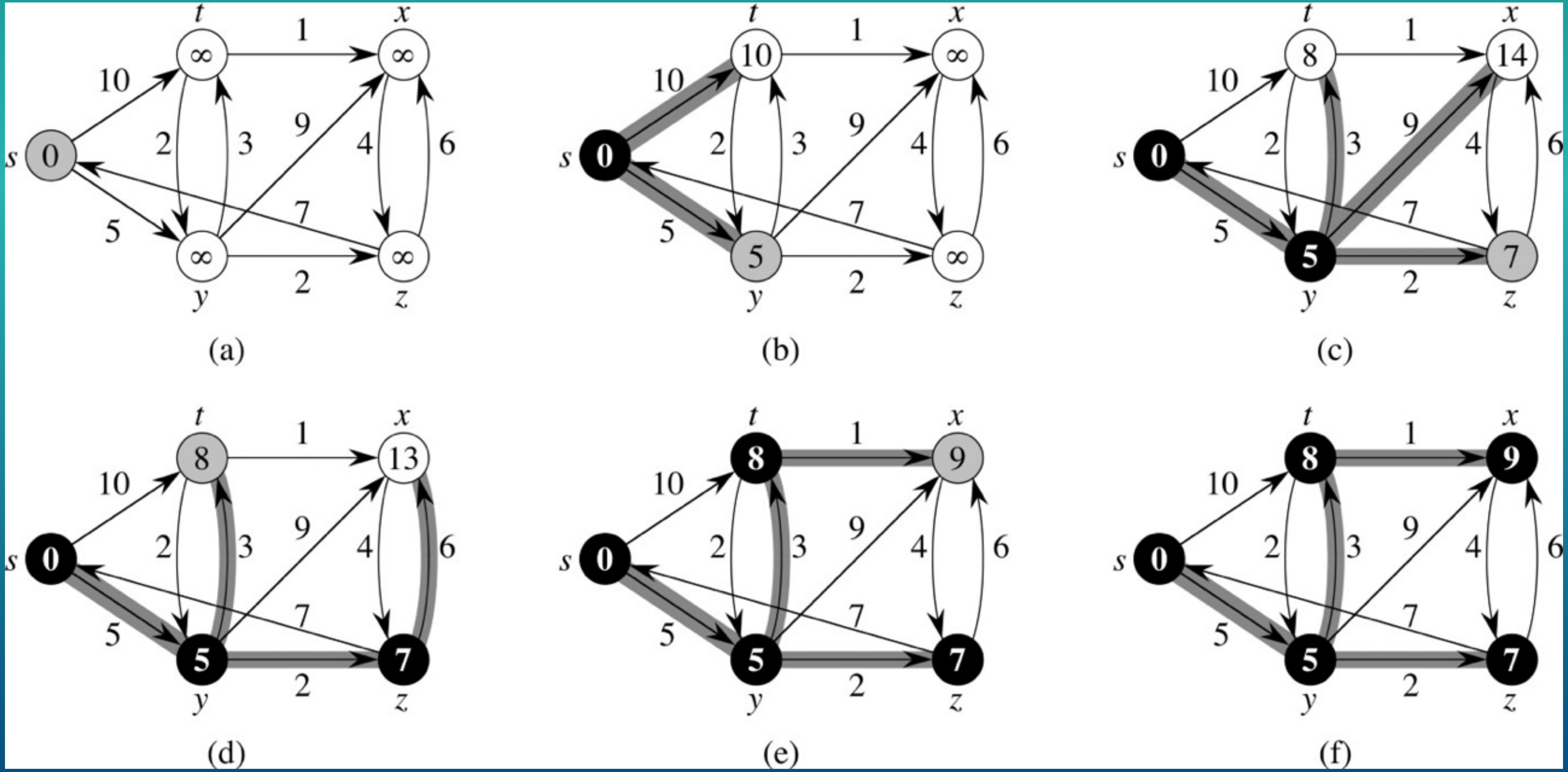
Dijkstra’s Algorithm

- Assign to every node a tentative distance value: set it to zero for our initial node and to infinity for all other nodes.
- Set the initial node as current. Mark all other nodes unvisited. Create a set of all the unvisited nodes called the unvisited set.
- For the current node, consider all of its unvisited neighbors and calculate their tentative distances. Compare the newly calculated tentative distance to the current assigned value and assign the smaller one. For example, if the current node A is marked with a distance of 6, and the edge connecting it with a neighbor B has length 2, then the distance to B (through A) will be 6 + 2 = 8. If B was previously marked with a distance greater than 8 then change it to 8. Otherwise, keep the current value.

Dijkstra’s Algorithm

- When we are done considering all of the neighbors of the current node, mark the current node as visited and remove it from the unvisited set. A visited node will never be checked again.
- If the destination node has been marked visited (when planning a route between two specific nodes) or if the smallest tentative distance among the nodes in the unvisited set is infinity (when planning a complete traversal; occurs when there is no connection between the initial node and remaining unvisited nodes), then stop. The algorithm has finished.
- Otherwise, select the unvisited node that is marked with the smallest tentative distance, set it as the new "current node", and go back to step 3.

Dijkstra’s Algorithm



Ruby Implementation

```ruby
def shortest_path(start, finish)
    maxint = (2**(0.size * 8 -2) -1)
    distances = {}
    previous = {}
    nodes = PriorityQueue.new

    @vertices.each do | vertex, value |
        if vertex == start
            distances[vertex] = 0
            nodes[vertex] = 0
        else
            distances[vertex] = maxint
            nodes[vertex] = maxint
        end
        previous[vertex] = nil
    end

    while nodes
        smallest = nodes.delete_min_return_key

        if smallest == finish
            path = []
            while previous[smallest]
                path.push(smallest)
                smallest = previous[smallest]
            end
            return path
        end

        if smallest == nil or distances[smallest] == maxint
            break
        end

        @vertices[smallest].each do | neighbor, value |
            alt = distances[smallest] + @vertices[smallest][neighbor]
            if alt < distances[neighbor]
                distances[neighbor] = alt
                previous[neighbor] = smallest
                nodes[neighbor] = alt
            end
        end
    end
    return distances.inspect
end
```