T2A1-B WORKBOOK

Timothy Long CAB022105

.md and .rb files at https://github.com/tfxl/workbook_T2A1--B.git

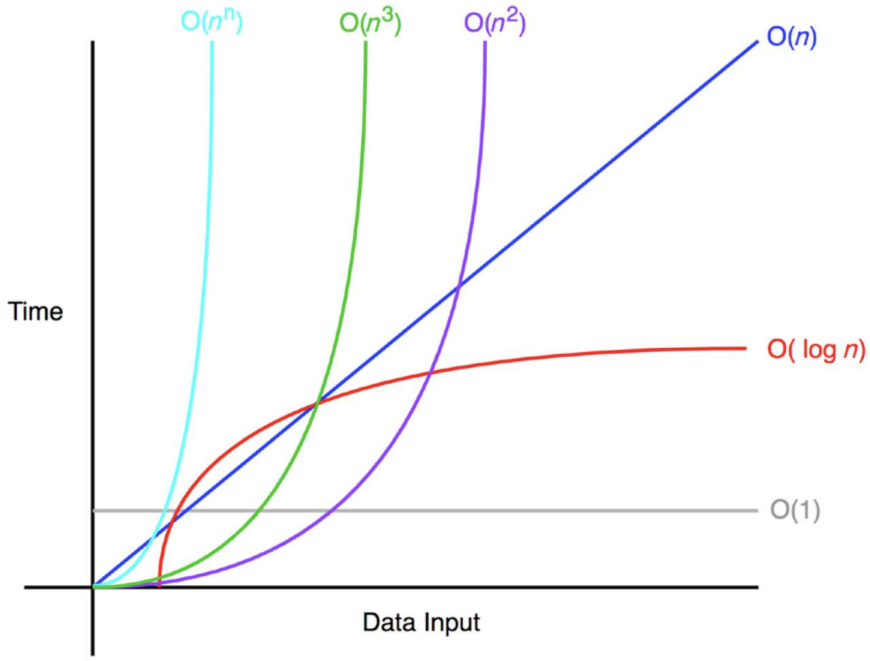
- **Big O Notation**
- **Sorting Algorithms**
  - Bubble
  - Merge
- **Search Algorithms**
  - Linear
  - Binary

Big O Notation

When discussing the performance of algorithms, one particular measure is using Big O notation, which can describe the efficiency regarding time complexity or memory required (1). Big O notation can be categorised as one of the asymptotic notations, which is a notation that takes into account the various type of input, such as the size of the input or whether that input is sorted or not (2), making it useful for comparing not only general efficiencies of various algorithms, but also specific efficiency for assessing one algorithm's performance with different inputs.

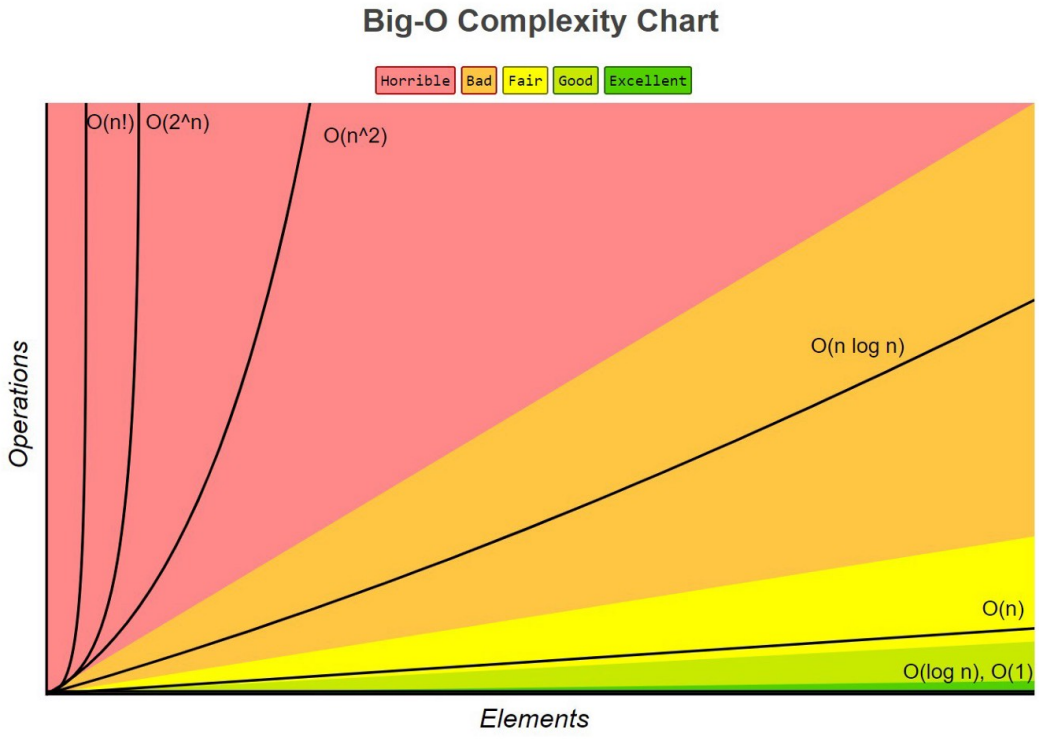
It is important to realise that Big O notation provides an upper bound representation of an algorithm's performance (1) via the worst-case scenario, or longest possible running time (3). This provides a generally more useful comparison between the various algorithms than a best-case scenario (3). For example, in a best-case scenario a search algorithm with poor efficiency might find the value in a single step which may replicate the same number of passes a highly efficient algorithm also takes in that same situation. Low bound representations, dealing in best case scenarios, make it difficult to compare the efficiencies, thus in many circumstances the upper bound Big O Notation is better for comparisons.

The graph below represents Big O Notation of various, but not all, notation types. The "O" represents the order-of-magnitude while the "n" is essentially the input being compared to the time complexity of the task (4) that the algorithm is solving. For instance, Big O notation of the linear O(n) relationship indicates that the order of magnitude is related to, at worst case, every increase of "n" units of input increases "n" units of time (or memory). Likewise, Big O notation of the quadratic $O(n^2)$ means that, at worst case, every increase of "n" units of input increases n^2 units of time (or memory).



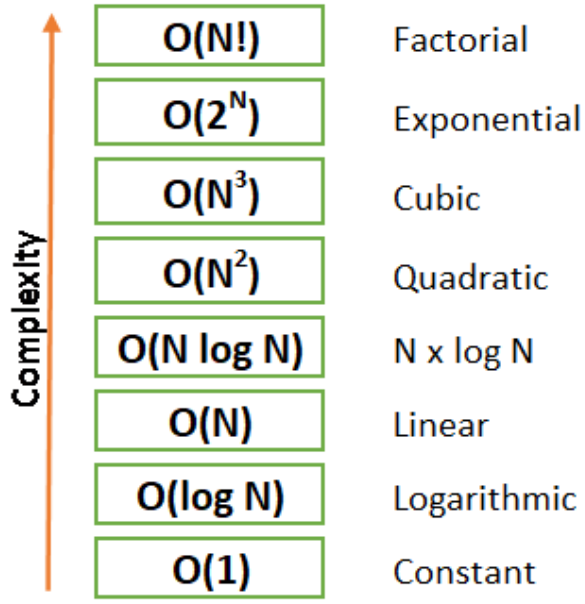
Source (5)

Additionally, it can be seen that Big O Notation is not designed to represent exact algorithmic equations, but the orders of growth (1). For instance, $O(n^2)$ might represent an algorithm such that $f(n) = 2n^2 + n + 3$ where non-dominant terms (1) and constants are removed (6). This does not affect the general rationale of Big O notation, noting that as "n" becomes significantly larger, the constants and non-dominant terms has less impact on the comparative algorithmic runtime (6). This leaves the focus on the aforementioned growth and allows useful comparisons between algorithmic efficiencies based on increased data inputs.



Source (7)

With this in mind, and prior exploring our sorting and searching algorithms, the Big O performances or efficiency of algorithms can ordered as such, with more complexity resulting in slower or more memory demanding algorithms.



Source (3)

Two alternative **sorting** algorithms and a comparison of their performance/efficiency

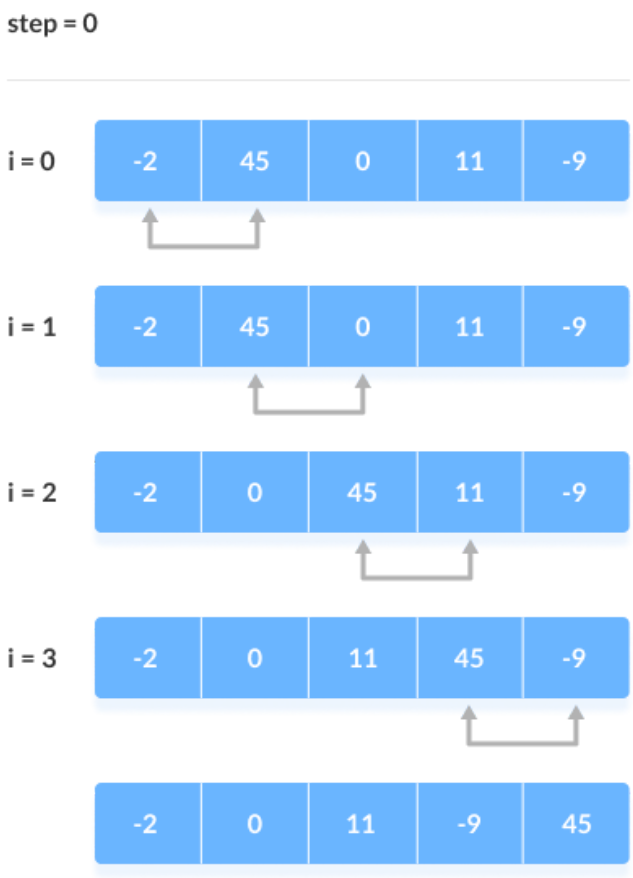
---

Bubble Sort

This stable algorithm employs a relatively simple solving process, but is not particularly time efficient, especially when acting on mostly unsorted data. Using Big O notation we can acknowledge this algorithm to be $O(n^2)$ for time complexity, although it has a memory or space complexity of only O(1) (8)

The process that Bubble Sort employs is that, given an array, it compares the first term (index 0) with the second term (index 1). If the value at index 0 is smaller than index 1, the algorithm will then compare values at index 1 and index 2. However, if the value at index 0 is larger than index 1, the algorithm will switch the values (index 0 value becomes index 1 value, and vice versa).

This process is repeated through the array, comparing (index i) with (index i+1) as illustrated :



Source (9)

It can be noted that once through the array, some sorting has been completed, however the array is not entirely sorted. The process needs to be repeated, potentially with many iterations, until the array is finally ordered.

```
def bubble_sort(array)
  return array if array.length < 2 # instantly returns any empty array, or
any single-element array

  max_index = (array.length - 2) # max_index is the upper range of the
index, i, required for comparison (recalling that i compares with i+1)

  loop do
    swap_occurred = false

    0.upto(max_index) do |i|

      if array[i] > array[i+1]
        array[i], array[i+1] = array[i+1], array[i] # single line to swap
values
        swap_occurred = true
      end
```

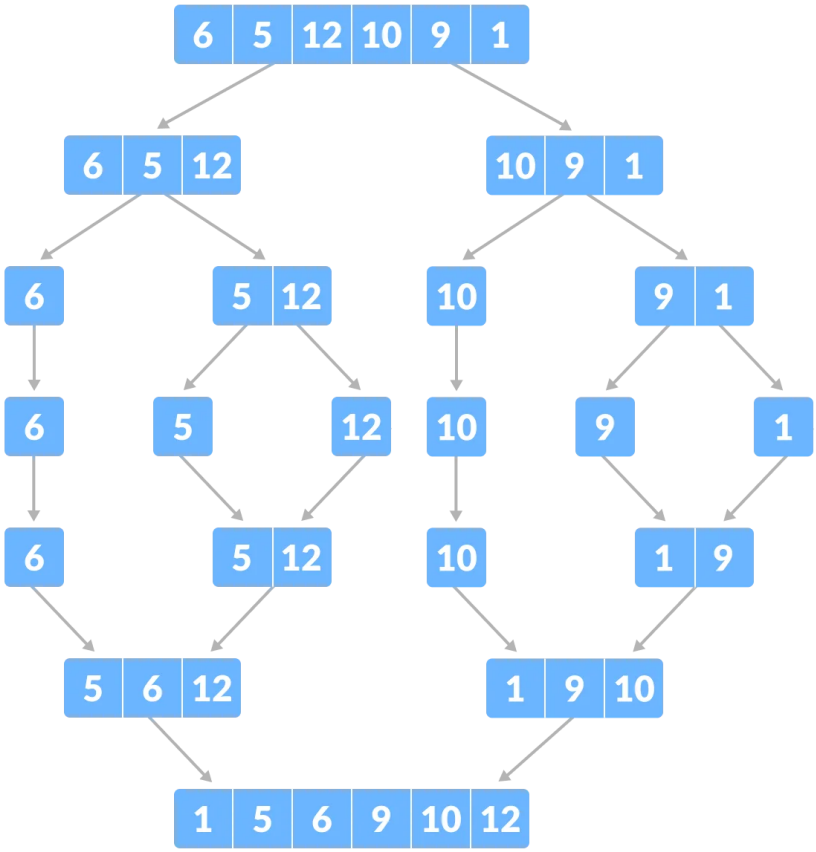
```
    end

  break unless swap_occurred # if no swap occurred, array is sorted, can
break loop and return array
  p array # added for output display purposes only

  end
  array
end

unsorted_array = [20, 21, 4, 8, 21, 3, 1, 70, 10]
sorted_array = bubble_sort(unsorted_array)
puts "Sorted array is #{sorted_array}"
```

```
workbook_T2A1-B % ruby bubble_sort.rb
[20, 4, 8, 21, 3, 1, 21, 10, 70]
[4, 8, 20, 3, 1, 21, 10, 21, 70]
[4, 8, 3, 1, 20, 10, 21, 21, 70]
[4, 3, 1, 8, 10, 20, 21, 21, 70]
[3, 1, 4, 8, 10, 20, 21, 21, 70]
[1, 3, 4, 8, 10, 20, 21, 21, 70]
Sorted array is [1, 3, 4, 8, 10, 20, 21, 21, 70]
```

Thus, while Bubble Sort could be useful in shorter arrays, or ones that are mostly sorted already, it is not as efficient as some other options, such as Merge Sort.

Merge Sort

Merge Sort is a stable sorting algorithm that is more time efficient compared with Bubble Sort, with O(n log n) time complexity (8), although it has a slightly decreased memory efficiency of O(n) (8) meaning that it requires an additional unit of memory per increased input unit, due to a series of recursive method calls (10). These recursive calls allow for the initial input, an array, to separate into smaller and smaller sub-arrays, until each sub-array is composed of only one element (10). This can be seen in the image below:



Source (11)

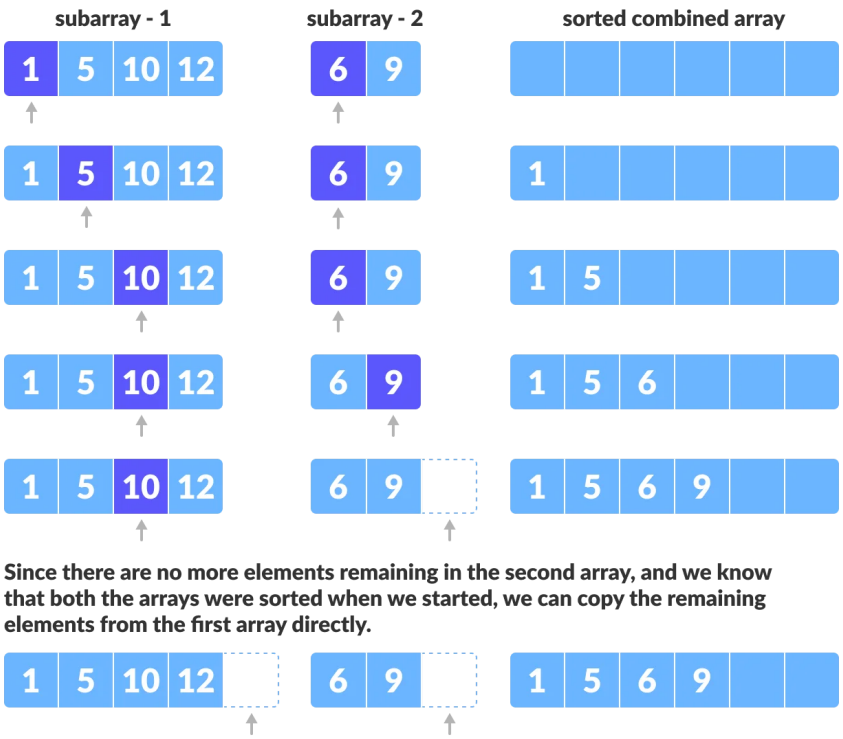
Once there is only one element in each sub-array, they can start to be reassembled, or "merged", but this time into their sorted forms. Importantly, before each sub-array is merged, it can be noted that they are already individually in order, starting from that single element array.

To merge each sub-array, the first value (index 0) of the first sub-array is compared with the first value (index 0) of the second sub-array. Whichever is smaller will be the first value (index 0) of the merged array.

For the second value of the merged array (now at index 1) the sub-arrays are compared again.

Assuming that the first value of the first sub-array was the smallest prior, then the second element in that sub-array will be compared with the first element of the second sub-array. Again, the smaller of these values will occupy the next position (index 1) in the merged array, and so on.

Repeating this process can be better shown graphically as follows :



Source (11)

Once one sub-array is exhausted, the remaining values of the other sub-array can be instantly appended to the merged array. An efficiently written piece of code can be utilised from Vladislav Trotsenko (12), although the comments have been added by this author, as have some small additions as noted. It is important to note that the "merge_sort" method is recursively called.

```
def merge_sort(arr)
  return arr unless arr.size > 1 # an empty array or an array with only
one element will be returned, otherwise method continues

  mid = arr.size/2 # this is the index where the first division will
occur, being half of the array length

  a, b, sorted = merge_sort(arr[0...mid]), merge_sort(arr[mid..-1]), []

      # this is where the recursive method calls are implemented, with "a"
starting on the first half, and "b" on the second half
      # as the call is recursive, it means that the initial "a" will be
split again into two parts, and repeated until a single element array
remains (each arr.size = 1)
      # an empty array is assigned to "sorted" to be used once the merging
begins

  sorted << (a[0] < b[0] ? a.shift : b.shift) while [a,b].none?(&:empty?)

  p sorted # added by author of comments for output display purposes only

    # these sub-arrays can be re-assembled (merged) by comparing their
```

```
first elements, and moving the smaller value into the sorted array
    # this will continue while it is true that a or b still has elements
that need to be merged

  sorted + a + b

  # once one of the sub-arrays no longer has elements then the sorted
array (currently merged values) can be concatenated with the arrays "a"
and "b"
  # It should be noted that "a" or "b" will be an empty array, and the
other will be the sorted values that are higher than those in "sorted"
  # this means that concatenating the arrays at this stage will still
retain the sorted order

end

unsorted_array = [20, 21, 4, 8, 21, 3, 1, 70, 10]
sorted_array = merge_sort(unsorted_array)
puts "Sorted array is #{sorted_array}"
```

```
workbook_T2A1-B % ruby merge_sort.rb
[20]
[4]
[4, 8]
[3]
[10]
[1]
[1, 3, 10, 21]
[1, 3, 4, 8, 10, 20, 21, 21]
Sorted array is [1, 3, 4, 8, 10, 20, 21, 21, 70]
```

Now that two **sorting** algorithms have been examined, two **searching** algorithms can be discussed.

Two alternative **searching** algorithms and a comparison of their performance/efficiency

Here, we can examine two fairly straightforward searching algorithms which can have dramatically different efficiencies. These are the Linear Search and Binary Search algorithms, and the below image gives a clear illustration of how starkly different the number of operations is for the number of inputs.

Input Size	Linear Search	Binary Search
10	10	4
50	50	6
100	100	7
200	200	8
500	500	9
1000	1000	10
5000	5000	13
10000	10000	14
50000	50000	16
100000	100000	17
1000000	1000000	20
10000000	10000000	24
100000000	100000000	27

Source (13)

It should be noted that these search algorithms are concerned with finding the first case of the searched value. It should also be noted that as our compared complexity relates to Big O Notation, the representations are upper bound, thus represent the worst case scenarios.

Linear Search

In this instance, assume an array containing 1000 elements. The linear search requires sequentially iterating over every element in a worst-case scenario, from index 0 to index 999 in order to find the value (or even to show that the value is not in the array). For an array of 10,000 elements it likewise has to sequentially iterate through all 10,000 elements to find the sought-after value, or show it to be missing. The increase in input correlates linearly with the increase in operations/time, thus Big O notation is **O(n)** for time complexity. This means that a Linear Search algorithm can be utilised for smaller inputs but will quickly suffer in performance with larger inputs. The Linear Search is noted to be memory efficient however, with a space complexity of 0(1) (14)

```
def linear_search(array, searched_value)

  i = 0
  while i < array.length

    if array[i] == searched_value
      puts "Index being checked is #{i}" # added for output display
purposes only
```

```
      return "Searched value '#{searched_value}' first located at index #
{i}"
    else
      p "Index being checked is #{mid_index}" # added for output display
purposes only
      i += 1 # if the value is not found, then iterate to the next index
in the array
    end
  end
  return "Searched value not found in provided array" # return once every
element in the array is exhausted in the search
end

array = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
searched_value = 10

result = linear_search(array, searched_value)
puts result
```

```
workbook_T2A1-B % ruby linear_search.rb
Index being checked is 0
Index being checked is 1
Index being checked is 2
Index being checked is 3
Index being checked is 4
Index being checked is 5
Index being checked is 6
Index being checked is 7
Index being checked is 8
Index being checked is 9
Searched value '10' first located at index 9
```

Note that although the array in the provided example is sorted, it **does not have to be** for a linear search to be performed, which can be considered an advantage of this search algorithm.

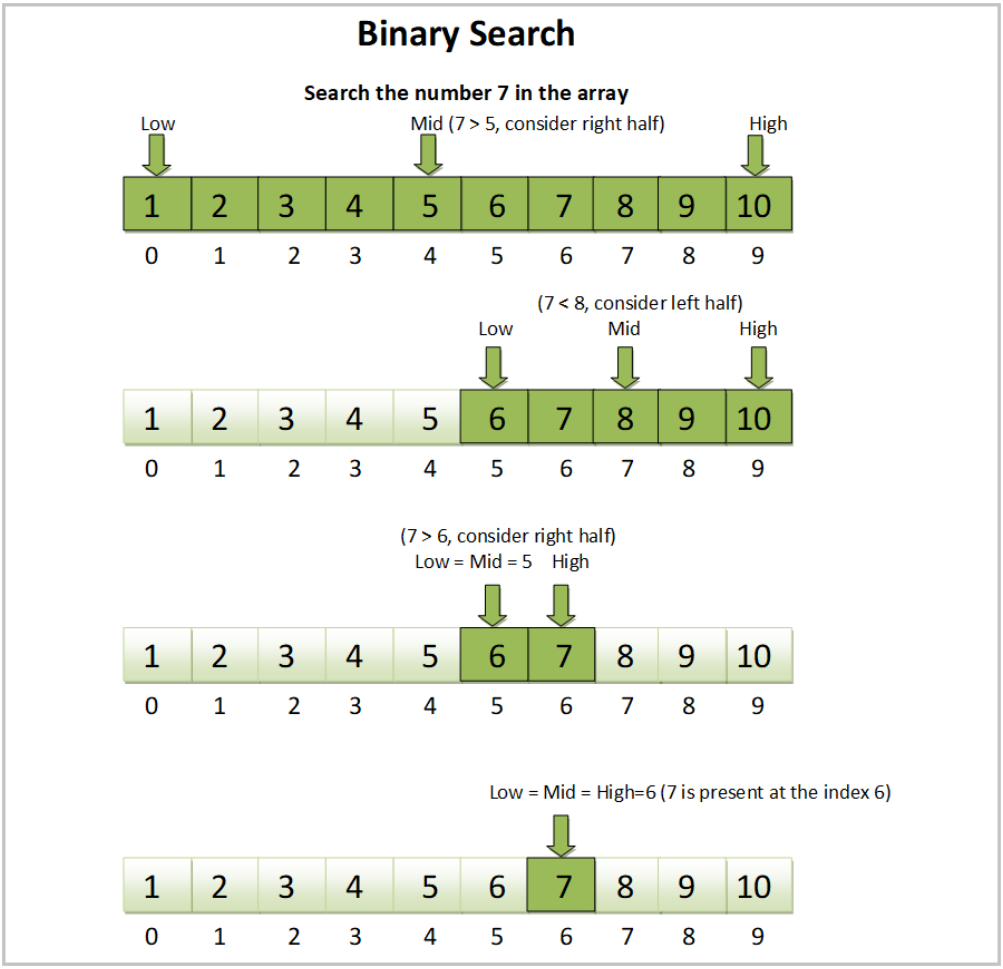
Binary Search

Conversely, a Binary search would have to be performed on a sorted array, so the Big O Notation of O(log n) has to take this into account.

> **Side Note:**
>
> If the provided array was unsorted, then comparing efficiencies would need to be "apples-to-apples" by comparing their Big O Notations on unsorted arrays, not sorted arrays.
> For Linear Search algorithms, this simply remains the same O(n), but it is not immediately possible with Binary Searches, given they require sorted arrays.

> In this case, the search algorithm can only be performed after applying a sorting algorithm, such as the previously discussed Merge Sort. Only then is there an "apples-to-apples" comparison, where Linear Search is now compared with a combined (Merge-Sort + Binary Search) performance. While the Binary Search algorithm is only at O(log n) complexity, the Merge Sort is O(n log n). In these cases, the overall Big O Notation is attributed based on the slower of the two; being O(n log n) in this case. Still, even here the (Merge Sort + Binary Search) is still considered much faster than a Linear Search with larger inputs.

The method that Binary Search employs is to divide the sorted array into two evenly split sub-arrays, and examining the element at the point of the divide. If the element is lower than the value searched, then given that the original array is sorted, the first sub-array can effectively be discarded, and the second sub-array can be examined. At this point, half of the values have effectively been assessed, and the same action is then applied to the second sub-array. At any point if the examined mid-point element matches the searched value, then the search is completed, otherwise it will continue in this pattern until the value is found, or shown to not be in the input.



Source (15)

It can be noted here that the data is effectively halved after every operation, so even in the worst case scenario, a sorted array of 10,000 elements would require 14 operations in the worst-case-scenario. With only an additional 10 operations in the worst-case-scenario, the Binary Search algorithm could find a value

within 10,000,000 elements in a sorted array. While the input is dramatically increased, the operations and time is not. This is representative of its time complexity O(log n), and it can be noted too that the memory or space complexity is the same as Linear Search, at O(1) (16)

An example of a Binary Search algorithm could be as follows:

```
def binary_search(array, searched_value)

  low_index = 0
  high_index = (array.length - 1)

  while low_index <= high_index

    mid_index = (low_index + high_index) / 2 # this returns the index that
is positioned midway between the lower end and higher end
    puts "Index being checked is #{mid_index}" # added for output display
purposes only

    if array[mid_index] == searched_value
        return "Searched value '#{searched_value}' first located at index
#{mid_index}"

    elsif array[mid_index] < searched_value
        low_index = mid_index + 1 # this addition is because the value at
the mid-index has already been checked

    else
        high_index = mid_index - 1 # this subtraction is because the value
at the mid-index has already been checked
    end
  end
  return "Searched value not found in provided array"
end


array = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
searched_value = 10
result = binary_search(array, searched_value)
puts result
```

```
workbook_T2A1-B % ruby binary_search.rb
Index being checked is 4
Index being checked is 7
Index being checked is 8
Index being checked is 9
Searched value '10' first located at index 9
```

It can also be noted that the Binary Search algorithm can utilise recursion, however not only does this not improve the time complexity, but it increases the memory or space complexity in Big O from a constant,

O(1) to O(log n) (16) making it less efficient in that regard.

Now, assuming the input is doubled, but still ordered, and a 'worst-case' scenario search is performed for the value "14", then it can be noted that even with double the input, only one extra step is required.

```
array = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18,
19, 20]
searched_value = 14

result = binary_search(array, searched_value)
puts result
```

```
workbook_T2A1-B % ruby binary_search.rb
Index being checked is 9
Index being checked is 14
Index being checked is 11
Index being checked is 12
Index being checked is 13
Searched value '14' first located at index 13
```

Indeed, this is predictable based on applying a logarithmic equation, given the Binary Search algorithm time complexity is O(log n).

- For instance, $\log_2(8) = 3$, indicating that for any array of up to but not including 8 elements, a Binary Search algorithm can find, or identify as absent, any element within 3 iterations.
- Applying $\log_2(16) = 4$ indicates that for any array up to but not including 16 elements in length, a maximum 4 operations will be required
- Thus an array of up to 10 elements will require, at most, 4 iterations, as exampled above.

A shorter method is to directly apply $\log_2(10)$ which gives 3.32 as the answer. By taking the floor (3) and adding one (3+1) means that at most, 4 operations will find any element, or show to be absent, in an array of 10 elements.

- Thus, predictably, $\log_2(20) = 4.32$, means that in the worst case scenario, 5 steps are required to find any value, or prove it absent, from an array with 20 elements, which is also exampled above.

Using the conversion from logarithms to exponents is also possible, showing that with up to 5 steps, a Binary Search can find, or identify as absent, any element of up to but not including 32 elements, as follows: $2^5 = 32$

And for larger values, $\log_2(1000000) = 19.93$ calculates that for an array of up to 1 million elements in length, any element can be found, or shown as missing, using a Binary Search algorithm, in only 20 steps.

Again, adjusting from logarithms to exponents, it can be shown that with 20 steps any element can be found, or identified as missing, by a Binary Search algorithm in an array of up to but not including 1048576 elements in size, by the equation $2^{20} = 1048576$

---

1. 2020, Ferreira, C., Towards Data Science, *The Math Behind "Big O" and Other Asymptotic Notations, viewed 20/11/2021, https://towardsdatascience.com/the-math-behind-big-o-and-other-asymptotic-notations-64487889f33f
2. Programiz, *Asymptotic Analysis: Big-O Notation and MOre*, viewed 20/11/2021, https://www.programiz.com/dsa/asymptotic-notations
3. Jamieson, A., 2021, Towards Data Science, *Introduction to Big O Notation*, viewed 20/11/2021, https://towardsdatascience.com/introduction-to-big-o-notation-820d2e25d3fd
4. Kuredjian, S., 2017, Medium, *Algorithm Time Complexity and Big O Notation*, viewed 20/11/2021, https://medium.com/@StueyGK/algorithm-time-complexity-and-big-o-notation-51502e612b4d
5. Ansari, G., 2020, Droid Tech Know, Big-O-Notation, viewed 20/11/2021, https://droidtechknow.com/programming/algorithms/big-o-notation/images/big-o-notation.jpg
6. Ryu, B., 2019, Medium, *Dropping conBig O Notation Cont.*, viewed 20/11/2021, https://medium.com/@s.brianryu/big-o-notation-cont-6b169c44235f
7. Saloton do Prado, K., 2019, Towards Data Science, *Understanding time complexity with Python examples*, viewed 20/11/2021, https://towardsdatascience.com/understanding-time-complexity-with-python-examples-2bda6e8158a7
8. Galler, L., Kimura, M., 2019, LAMFO, *Sorting Algorithms*, viewed 20/11/2021, https://lamfo-unb.github.io/2019/04/21/Sorting-algorithms/
9. Kent, J., 2020, Honey Badger, *Understanding and Implementing Bubble Sort in Ruby*, viewed 20/11/2021, https://www.honeybadger.io/blog/ruby-bubble-sort/
10. Parekh, D., 2020, Java Revisited, *Sorting Algorithms: Slowest to Fastest!*, viewed 20/11/2021, https://medium.com/javarevisited/sorting-algorithms-slowest-to-fastest-a9f0e30937b9
11. Programiz, *Merge Sort Algorithm*, viewed 20/11/2021, https://www.programiz.com/dsa/merge-sort
12. GitHub Gist, *Merge Sort Example in Ruby*, viewed 20/11/2021, https://gist.github.com/bih/9726300
13. 2020, Angelidakis, H., 2020, Cantors Paradise, *The Power of Binary Search*, viewed 20/11/2021, https://www.cantorsparadise.com/the-power-of-binary-search-86218176d08e
14. IQ, *Time & Space Complexity of Linear Search [Mathematical Analysis]*, viewed 20/11/2021, https://iq.opengenus.org/time-complexity-of-linear-search/
15. Praveen, H., 2019, Java Code Korner, *Java Program For Binary Search*, viewed 20/11/2021, https://javacodekorner.blogspot.com/2019/03/binary-search-algorithm-java.html
16. IQ, *Time & Space Complexity of Linear Search [Mathematical Analysis]*, viewed 20/11/2021, https://iq.opengenus.org/time-complexity-of-binary-search/