ada computer science

Computer Science

STEM SMART - Week 29 - Big O notation

UNIVERSITY OF CAMBRIDGE

Raspberry Pi Foundation

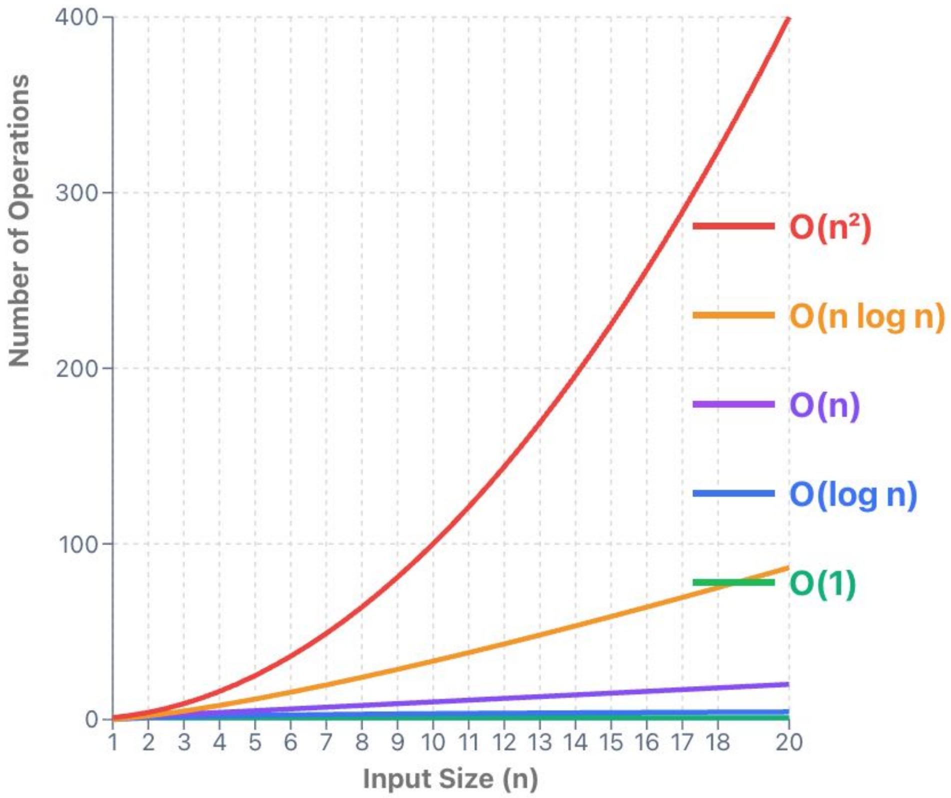
This week’s topic is complexity of searching and sorting algorithms

- Understand that some algorithms are more efficient time-wise than other algorithms.
- Understand that some algorithms are more efficient space-wise than other algorithms.
- Understand that algorithms can be compared by expressing their complexity as a function relative to the size of the problem.
- Show understanding that different algorithms which perform the same task can be compared by using criteria (e.g. time taken to complete the task and memory used).
- Be familiar with Big-O notation to express time and space complexity.
- Interpret the time complexity of an algorithm.

STEM SMART Computer Science Week 29

1. Linear search: time complexity
2. Binary search: time complexity
3. Search complexity
4. Bubble sort: time complexity 1
5. Insertion sort: time complexity
6. Merge sort: time complexity
7. Quick sort: time complexity
8. Compare sorting algorithms
9. Quick vs bubble sort
10. Quick vs merge sort

Big O Complexity Growth Comparison



Question 1 – Linear search: time complexity

Which of the following is the worst-case time complexity of a linear search, expressed in Big O notation?

- ○ $\mathcal{O}(1)$
- ○ $\mathcal{O}(\log n)$
- ○ $\mathcal{O}(n)$
- ○ $\mathcal{O}(n^2)$

Pseudocode | Python | C# | Visual Basic | Java

```
FUNCTION linear_search_version_2(items, search_item)

    // Initialise the variables
    found_index = -1
    current = 0
    found = False

    // Repeat while the end of the list has not been reached
    // and the search item has not been found
    WHILE current < LEN(items) AND found == False

        // Compare the item at the current index to the search item
        IF items[current] == search_item THEN
            // If the item has been found, store the current index
            found_index = current
            found = True // Raise the flag to stop the loop
        ENDIF
        current = current + 1 // Go to the next index in the list
    ENDWHILE

    // Return the index of the search_item or -1 if not found
    RETURN found_index

ENDFUNCTION
```

Question 2 – Binary search: time complexity

Which of the following is the worst-case time complexity of a binary search, expressed in Big O notation?

- $\mathcal{O}(\log n)$
- $\mathcal{O}(n)$
- $\mathcal{O}(n^2)$
- $\mathcal{O}(n \log n)$

```python
def binary_search(items, search_item):

    # Initialise the variables
    found = False
    found_index = -1
    first = 0
    last = len(items) - 1

    # Repeat while there are still items between first and last
    # and the search item has not been found
    while first <= last and found == False:

        # Find the midpoint position (in the middle of the range)
        midpoint = (first + last) // 2

        # Compare the item at the midpoint to the search item
        if items[midpoint] == search_item:
            # If the item has been found, store the midpoint position
            found_index = midpoint
            found = True # Raise the flag to stop the loop

        # Check if the item at the midpoint is less than the search item
        elif items[midpoint] < search_item:
            # Focus on the items after the midpoint
            first = midpoint + 1

        # Otherwise the item at the midpoint is greater than the search i
        else:
            # Focus on the items before the midpoint
            last = midpoint - 1

    # Return the position of the search_item or -1 if not found
    return found_index
```

Question 3 – Bubble sort: time complexity

Which of the following is the worst-case time complexity of an bubble sort, expressed in Big O notation?

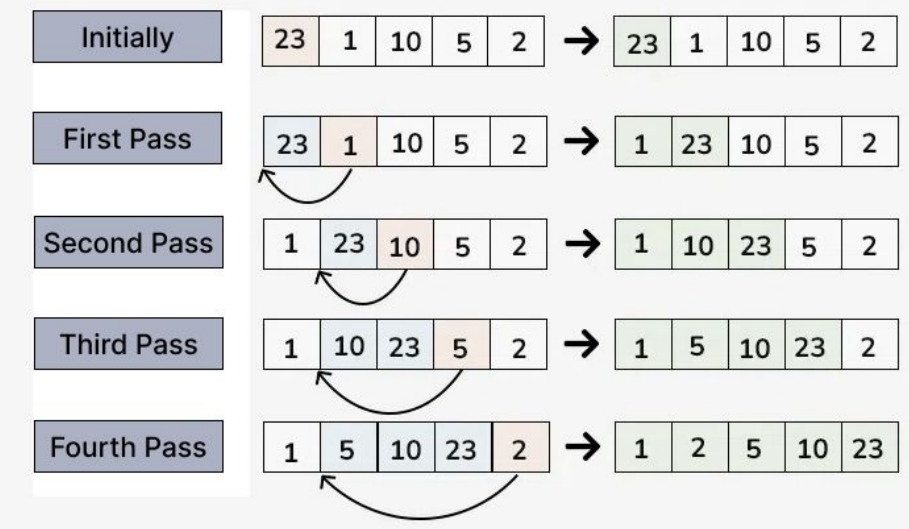
- $\mathcal{O}(n)$
- $\mathcal{O}(\log n)$
- $\mathcal{O}(n^2)$
- $\mathcal{O}(n \log n)$

```
PROCEDURE bubble_sort_version_3(items)

    // Initialise the variables
    num_items = LEN(items)
    swapped = True
    pass_num = 1

    // Repeat while one or more swaps have been made
    WHILE swapped == True
        swapped = False
        // Perform a pass, reducing the number of comparisons each time
        FOR index = 0 TO num_items - 1 - pass_num
            // Compare items to check if they are out of order
            IF items[index] > items[index + 1] THEN
                // Swap the items
                temp = items[index]
                items[index] = items[index + 1]
                items[index + 1] = temp
                swapped = True
            ENDIF
        NEXT index
        pass_num = pass_num + 1
    ENDWHILE
ENDPROCEDURE
```

Question 4 – Insertion sort: time complexity

Which of the following is the worst-case time complexity of an insertion sort, expressed in Big O notation?

- $\mathcal{O}(n)$
- $\mathcal{O}(\log n)$
- $\mathcal{O}(2^n)$
- $\mathcal{O}(n^2)$

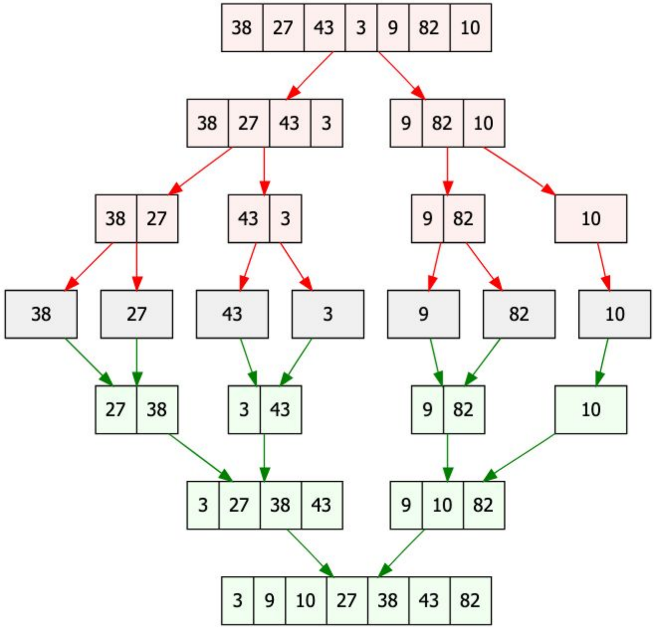


```
PROCEDURE insertion_sort(items)

    // Initialise the variables
    num_items = LEN(items)

    // Repeat for each item in the list, starting at the second item
    FOR index = 1 TO num_items - 1
        // Get the value of the next item to insert
        item_to_insert = items[index]

        // Get the current position of the last sorted item
        position = index - 1

        // Repeat while there are still items in the list to check
        // and the current sorted item is greater than the item to ins
        WHILE position >= 0 AND items[position] > item_to_insert

            // Copy the value of the sorted item up one place
            items[position + 1] = items[position]

            // Get the position of the next sorted item
            position = position - 1
        ENDWHILE

        // Copy the value of the item to insert into the correct posit
        items[position + 1] = item_to_insert
    NEXT index
ENDPROCEDURE
```

Question 5 - Merge sort: time complexity

Which of the following is the worst-case time complexity of a merge sort, expressed in Big O notation?

- $\mathcal{O}(\log n)$
- $\mathcal{O}(n)$
- $\mathcal{O}(n^2)$
- $\mathcal{O}(n \log n)$



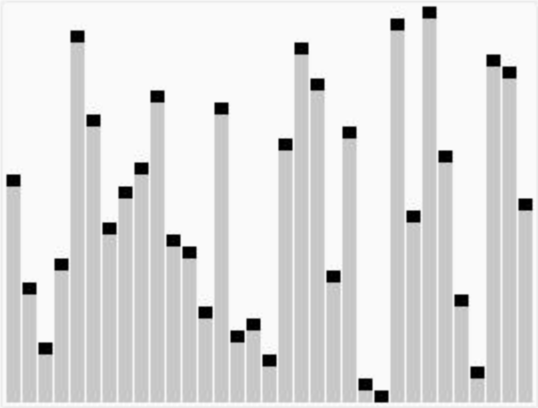
```python
def merge(left, right):

    merged = [] # New list for merging the items
    index_left = 0 # left current position
    index_right = 0 # right current position

    # While there are still items to merge
    while index_left < len(left) and index_right < len(right):

        # Find the lowest of the two items being compared
        # and add it to the new list
        if left[index_left] < right[index_right]:
            merged.append(left[index_left])
            index_left += 1
        else:
            merged.append(right[index_right])
            index_right += 1

    # Add to the merged list any remaining data from left list
    while index_left < len(left):
        merged.append(left[index_left])
        index_left += 1

    # Add to the merged list any remaining data from right list
    while index_right < len(right):
        merged.append(right[index_right])
        index_right += 1

    return merged
```

```python
def merge_sort(items):

    # Base case for recursion:
    # The recursion will stop when the list has been divided into single
    if len(items) <= 1:
        return items
    else:
        midpoint = (len(items)-1) // 2 # Calculate the midpoint index
        left_half = items[0:midpoint+1] # Create left half list
        right_half = items[midpoint+1:len(items)] # Create right half li

        left_half = merge_sort(left_half) # Recursive call on left half
        right_half = merge_sort(right_half) # Recursive call on right ha

        # Call procedure to merge both halves
        merged_items = merge(left_half, right_half) # Call function to m

        return merged_items
```

Question 6 – Quick sort: time complexity

Which of the following is the worst-case time complexity of a quick sort, expressed in Big O notation?

- $\mathcal{O}(n^2)$
- $\mathcal{O}(\log n)$
- $\mathcal{O}(n \log n)$
- $\mathcal{O}(n)$



```python
def quick_sort(items, start, end):

    # Base case for recursion:
    # The recursion will stop when the partition contains a single item
    if start >= end:
        return

    # Otherwise recursively call the function
    else:
        pivot_value = items[start] # Set to first item in the partition
        low_mark = start + 1 # Set to second position in the partition
        high_mark = end # Set to last position in the partition
        finished = False

        # Repeat until low and high values have been swapped as needed
        while finished == False:

            # Move the left pivot
            while low_mark <= high_mark and items[low_mark] <= pivot_valu
                low_mark = low_mark + 1 # Increment low_mark

             # Move the right pivot
            while items[high_mark] >= pivot_value and high_mark >= low_ma
                high_mark = high_mark - 1 # Decrement high_mark

            # Check that the low mark doesn't overlap with the high mark
            if low_mark < high_mark:
                # Swap the values at low_mark and high_mark
                temp = items[low_mark]
                items[low_mark] = items[high_mark]
                items[high_mark] = temp

            # Otherwise end the loop
            else:
                finished = True

        # Swap the pivot value and the value at high_mark
        temp = items[start]
        items[start] = items[high_mark]
        items[high_mark] = temp

        # Recursive call on the left partition
        quick_sort(items, start, high_mark - 1)

        # Recursive call on the right partition
        quick_sort(items, high_mark + 1, end)
    return items
```

Time complexity

If you are asked only to state the "time complexity" of the quick sort algorithm, you should give the worst case, which is $\mathcal{O}(n^2)$. This is called polynomial time efficiency.

Best, average, and worst-case time complexity

The algorithm is recursive and the number of calls (i.e. number of times that the list is partitioned) will depend on the value chosen for the pivot. The choice of pivot value can make a big difference to the time taken for the algorithm to run.

- In the **best case**, each time the list is partitioned, it is divided neatly into two sections of equal size. This means $\log_2 n$ nested calls will be needed before the partition size is 1 and the base case is reached. Therefore, the best-case time complexity is $\mathcal{O}(n \log n)$.

- The **worst case** is encountered when the pivot value is in the first or last position of the current partition as this will result in one partition with zero elements (for example, no elements to the left of the pivot value) and another partition having all the remaining elements; starting with $n - 1$ elements. If this happens repeatedly (every time the list is partitioned), the next step will result in a partition with zero elements and another one with $n - 2$ elements and so on. As a result of this, the time complexity is classified as $\mathcal{O}(n^2)$.

Question 7 - Compare sorting algorithms

Your classmate asks you for help in picking an efficient sorting method to use in his zoo application.

He wants to sort the list of animals below into alphabetical order:

```
animals = ['aoudad', 'camel',
'cheetah', 'crow', 'baboon',
'deer', 'hare', 'leopard', 'mink',
'peccary', 'moose', 'mule',
'parrot']
```

Which of the below statements is correct?

- Insertion sort will make the fewest comparisons because the list is almost sorted.
- Bubble and insertion sort will take the same amount of time to sort the list because they both have a time complexity of $O(n^2)$.
- Merge sort will be the most efficient choice because it has the lowest time and space complexity.
- Bubble sort will make the fewest comparisons because it stops as soon as the list is sorted.

Question 7 – working out

aoudad, camel, cheetah, crow, baboon, deer, hare, leopard, mink, peccary, moose, mule, parrot

Question 8 – Quick vs bubble sort

Two sorting algorithms are quick sort and bubble sort. Two students have been arguing about which is the best to use. Barack favours the quick sort algorithm whereas Salome is championing the bubble sort.

Both students have written two statements in support of their favoured algorithm but only one of the statements is correct. Which one is it?

- On average, the bubble sort algorithm is faster at sorting data than the quick sort algorithm.
- The quick sort algorithm is more suitable for sorting large data sets than the bubble sort algorithm.
- The quick sort algorithm is better than the bubble sort algorithm for sorting a set of data that is already substantially in order.
- The bubble sort algorithm has lower space complexity than quick sort.

Question 9 – Quick vs merge sort

The quick sort algorithm is a very efficient sorting algorithm. On average, time complexity is $O(n\ log\ n)$ which is the same as a merge sort. However, in the best-case time complexity merge sort outperforms quick sort. Despite this, the quick sort is often favoured over merge sort.

Which of the following statements provides the best reason for choosing quick sort to sort your data?

- Quick sort has better space complexity than merge sort.
- Quick sort will out perform merge sort if an optimal pivot value is chosen.
- Quick sort performs better than merge sort in the worst-case time complexity.
- Quick sort is easier to code than merge sort.

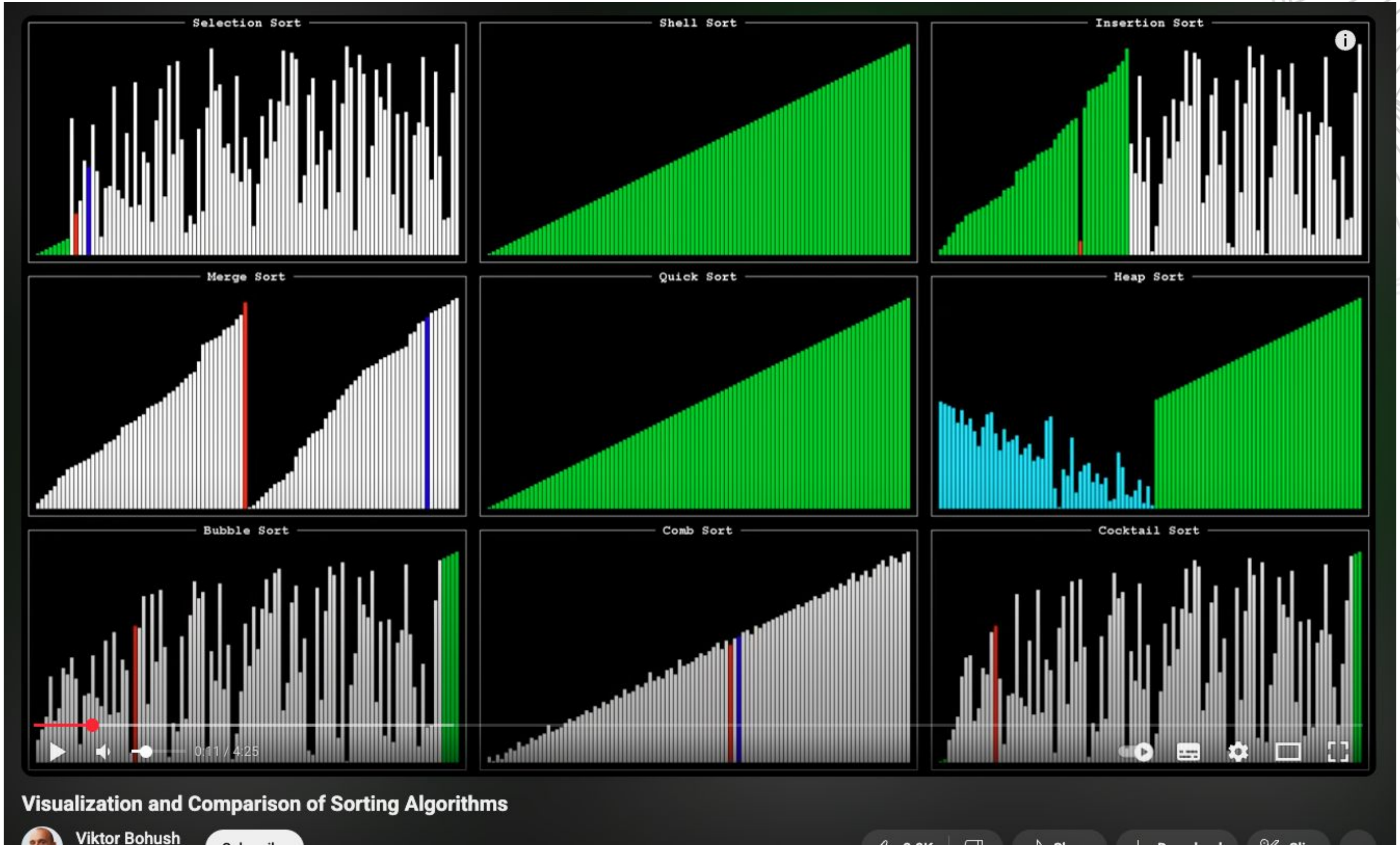
Big O Notation Algorithm Comparison

Key Notes:

- n represents the number of elements in the dataset
- $O(1)$ = Constant time (fastest)
- $O(\log n)$ = Logarithmic time (very fast)
- $O(n)$ = Linear time (moderate)
- $O(n \log n)$ = Linearithmic time (efficient for sorting)
- $O(n^2)$ = Quadratic time (slow for large datasets but still acceptable)

- **Stable sort** means equal elements maintain their relative order
- **In-place** means minimal extra memory is used

Algorithm	Worst Case Time	Best Case Time	Space Complexity	Pros	Cons
Linear Search	$O(n)$	$O(1)$	$O(1)$	Simple to implement Works on unsorted data	Slow for large datasets
Binary Search	$O(\log n)$	$O(1)$	$O(1)$	Very fast for large datasets Predictable performance	Requires sorted data
Bubble Sort	$O(n^2)$	$O(n)$	$O(1)$	Simple to understand and implement In-place sorting	Very slow for large datasets Inefficient - many comparisons
Insertion Sort	$O(n^2)$	$O(n)$	$O(1)$	Efficient for small datasets Good for nearly sorted data In-place sorting	Slow for large datasets Many shifts required
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n)$	Guaranteed $O(n \log n)$ time Predictable performance Good for large datasets	Requires extra memory – Not in-place Slower than Quick Sort in practice
Quick Sort	$O(n^2)$	$O(n \log n)$	$O(\log n)$ *recursion stack*	Very fast average case In-place sorting Widely used in practice	Unstable sort Worst case is $O(n^2)$ Performance depends on pivot choice



https://www.youtube.com/watch?v=ZZuD6iUe3Pc