Time and space complexity in sorting algorithms

You previously learned that time and space complexity are a means of evaluating code efficiency. In this reading, you will explore time and space complexity in both the selection sort and quicksort algorithms. These are common algorithms that are used to sort data in an array.

Selection sort

Selection sort is a sorting algorithm that works from a very simple principle. Take an array to items and iterate from left to right. Starting with the first place on the index, iterate over the entire array and swap this value with the lowest value found to the right of this item. Repeat until the entire array is sorted.

Selection sort has:

- Worst case time complexity is `O(N^2)`
- Average case time complexity is `O(N^2)`
- Best case time complexity is `O(N^2)`
- Space complexity: `O(1)` *Auxiliary.*

To perform selection sort, take the following steps:

- Find the smallest value and swap it with the first value of the array
- Find the second smallest value and swap it with the second place in the array
- Repeat until all items are changed from ordered from smallest to largest

Time complexity is determined in relation to the number of transactions enacted. Given a list of size `n`, the compiler must search each entry in the list to identify the smallest item, then perform a swap to index location 0. The pseudocode for the algorithm is as below.

```
1   for(i = 0; i < n-1; i++)
2   int min_index=List[i]
3       for(j=i+1; j<n;j++)
4           if(List[j] < List[min_index])
5               min_index=j
6       swap(List[i], List[min_index])
```

Line 1 says that the length of the list `List` must be searched `n-1` times. Line 2 sets a temporary variable to hold the lowest value. Line 3 is an inner loop that must iterate through the loop `n-1` times. Line 4 checks if the value found in position `List[j]` is smaller than the current lowest value. If so, the position of that element is recorded. At the end of each inner loop, the value found to be the lowest is swapped with position `i` in the index, `i` is incremented and the procedure begins again. Always check the next item in the list until every item has been checked.

There are four considerations to be made when evaluating this algorithm.

1. Worst case scenario: Given a list sorted in reverse order, how many comparisons are made? The inner and outer loop will have to run `n` times so it can be determined that worst case `= O(n^2)`.
2. Average comparison: Regardless of the order of the list, every item must be checked against average case `= O(n^2)`.
3. Best comparison: Given a sorted list of how many comparisons must be made. Again, regardless of the items in

the list, every item must be checked, so best case `= O(n^2)`.

4. Finally, what is the space complexity of this approach? Because an in-place swap is being performed, no temporary array is required. There are three temporary variables **`i`**, **`j`** and **`min_index`**; however, these are not dependent on the list size. So, the image doubles the list, and the space complexity does not increase accordingly. Therefore, space complexity `= O(1)`.

When evaluating time complexity, a good rule of thumb is to consider what will happen if the list is doubled. Naturally, the inner and outer loops will have to increase by no iterations to match the additional elements in the list. Therefore, it can be concluded that the time complexity increases with the size.

Quicksort

Quicksort is a sorting approach that uses a divide-and-conquer methodology. Given an array of items, a place is determined on the array on which to split the array and this is called the pivot point. All values greater than this point go to the right and all values less than this point go to the left. In this step, you have two arrays. The same process is applied to these arrays until there are no elements left to sort.

Quicksort has:

- Worst case time complexity **`O(n^2)`**
- Average case time complexity **`O(n log n)`**
- Best case time complexity **`O(n log n)`**
- Space complexity **`O(n)`**

To perform quicksort, take the following steps:

- Select a point on the list to pivot on.
- Split the list into two lists, items to the left of the pivot and items to the right.
- Set variables **`i`** to iterate from left to right on the left of the pivot. Set variable **`j`** to repeat from right to left on the left side of the pivot.
- The variables **`i`** on the left look for a value greater than or equal to the pivot. Variables **`j`** on the right look for a value less than or equal to the pivot.
- When **`j < i`**, the values at these index locations are swapped, this is repeated until **`i`** and **`j`** meet at the pivot point.
- Partition the list values into two lists, one to the left and one to the right of the pivot. Repeat the process on each of the resulting arrays.
- Recursively apply the algorithm.

The pseudocode below for quicksort is done recursively.

- Starting at the leftmost element, each subsequent element is checked, and if it is found to be less, it is swapped.
- Line 3 calls the partition method, which begins on line 8.
- Line 10 determines the more significant element to be placed on the right side of the list. Line 10 sets a variable **`i`**

to be assigned to the index of the smaller element. The variable `j` is then used to check the elements to the right from which to make a comparison with the current smallest element.

- Line 12 determines if there is to be a swap, a smaller element on the right will require moving to the current index position. Line 4 is for sorting the left array.
- Line 5 is for sorting the right array. At each iteration, the size of the array to be sorted is halved. The arrays will continually break down until only one element is left in the subarrays. The result of calling partition will determine the location of the current element. This location is incremented and repeated until every element rests in its naturally ordered position.

```
1   QuickSort(List, low, high)
2           if(low<high)
3           pivot=partition(List, high, low)
4           QuickSort(List, how, pivot-1)
5           QuickSort(List, pivot+1, high)
6
7   Partition(List,high,low)
8           pivot=arr[high]
9           i=(low-1)
10          for j = low; j <= high-1; j++)
11          if(List[j] < pivot)
12                  i++
13                  swap(List[i], List[j])
14          swap(arr[i+1], List[j])
15          return I + 1
16
```

Things to consider when evaluating this algorithm:

1. Worst case scenario: this happens when the most significant element is consistently chosen as a pivot point. This will cause a loop to iterate over every element n from the left. The split will cause a search of every element on the right with none on the left, `O(n^2)`.
2. Average case scenario: an average pivot point is selected at every call. This will reduce the number of additional iterations required. So, there will be n iterations and an ever-decreasing `logn` iterative calls, `O(n*logn)`.
3. Best case scenario: The middle value is always selected, and the iteration space is halved at every iteration, `O(n*logn)`.
4. The iterative nature of the algorithm will impact the space complexity because the function call and variables are retained on the stack while the calculations are performed. However, the decision to use an in-place swap means no new array needs to be created, `O(log n)`.

Conclusion

Two different sorting approaches have been broken down and analyzed through the lens of Big-O space and complexity. It has been shown that quicksort is more complex in implementation but returns overall quicker solutions. Selection sort is more simplistic and less code-heavy and requires less space, but will not generate results as effectively.

In this reading, you explored time and space complexity in both the selection sort and quicksort algorithms.