UNIVERSITY OF CALGARY
FACULTY OF SCIENCE

University of Calgary: Faculty of Science

DATA 311: Data Processing and Storage, Fall 2022 L01

Assignment 2

Computational Complexity of Sorting Algorithms

Ivan Law

October 16, 2022

Overview

The goal of this report is to analyze the computational complexity of various algorithms used to sort lists of integers in ascending order. The algorithms analyzed are bubble sort, insertion Sort, merge sort and quick sort. Variables of this analysis include the size of the list (n) and the order of the list prior to sorting. The size of the list we will try are n = 10, n = 100, n = 1000, n = 10000, n = 100000, and n = 1000000. The order of the list can be random, sorted ascending or sorting descending prior to running the sorting algorithm. For each algorithm, we determine the time complexity of each algorithm for every combination of variables. Time complexity is determined by checking the time it takes for the sorting algorithm to run in code. The runtime data for each algorithm have been tabulated and analyzed. A computation complexity analysis is conducted on for the four algorithms by counting the number of operations in the worse case scenario, using the big-O notation. The results of the time and operational complexity are interpreted. The most appropriate algorithm for sorting small versus large list sizes is determined and algorithm limitations are discussed.

Analysis and interpretation of Time Complexity Data

The bubble sort algorithm appears to have the highest runtime across all scenarios followed closely by insertion sort. This conforms with expectations since both of the algorithms are quadratic algorithms and as you will see later in the analysis, has a computational complexity of O(n^2). So naturally as the input size increases to a large number, the computational complexity is approximately the square of that large number. Both algorithms faster than merge sort and quicksort in the best case scenario, suggesting that they can take better advantage of pre-sorted lists. However, they take longer than 10 minutes to run for list sizes greater than 100000 in the worst case scenario, while the merge sort and quick sort run much faster. It appears though that Quick Sort was unable to run for list sizes of 100000 and above. This is likely because recursion is a memory intensive task. In the worst case of quicksort, where the first integer of the list is selected as the pivot, it requires n-1 recursions to complete the task (where n is the list size). It is also observed that QuickSort performs better than merge sort at very large values, particularly at list sizes of 100000 and larger. As for merge sort, it appears to perform almost the same across worst, best and average cases. This suggests that the order of the list does not matter.

Analysis and Interpretation of Complexity Analysis

When comparing the big-O value for each algorithm, bubble sort, insertion and quick sort are O(n^2) while merge sort is O(n lg n). Note that quick sort would also be an n lg n algorithm in the average and best case. In the best case, the middle value of the list is selected as the pivot instead of the first. We will treat quick sort as an O(n lg n) for this analysis. The O(n lg n) sorting algorithms generally run much faster than the O(n^2) algorithms for lists of large size. The time complexity data confirms this is the case. The O(n lg n) algorithms are also recursive algorithms, which suggest that recursive methods of sorting have comparatively lower complexity than the other sorting algorithms. Note that quick sort did have very large space complexity in the worse case scenario. When comparing the O(n lg n) algorithms to each other, we can see that merge sort uses list slicing method that is O(n) complexity twice. Whereas, quick sort uses an assignment method that is O(1) complexity. This could contribute to better time efficiency of

quick sort over merge sort. For the O(n^2) algorithms, insertion sort would take less steps to sort an already partially sorted list compared to bubble sort. Order appears to matter for bubble sort, insertion sort and quick sort, but not merge sort. This appears to be related to how to merge sort slices lists without checking location.

Time Complexity Data

Bubble Sort			
	Scenario		
Data Size	Worst Case	Best Case	Average Case
10	0	0	0
100	0	0	0
1000	0.203125	0	0.171875
10000	15.703125	0.015625	11.953125
100000	>10 minutes	0.015625	>10 minutes
1000000	>10 minutes	0.25	>10 minutes

Insertion Sort			
	Scenario		
Data Size	Worst Case	Best Case	Average Case
10	0	0	0
100	0	0	0
1000	0.109375	0	0.078125
10000	9.140625	0	5.140625
100000	>10 minutes	0.015625	>10 minutes
1000000	>10 minutes	0.359375	>10 minutes

Merge Sort			
	Scenario		
Data Size	Worst Case	Best Case	Average Case
10	0	0	0
100	0	0	0
1000	0	0	0
10000	0.046875	0.046875	0.078125
100000	0.4375	0.40625	0.5
1000000	6.46875	6.25	6.421875

Quick Sort			
	Scenario		
Data Size	Worst Case	Best Case	Average Case
10	0	0	0
100	0	0	0
1000	0.078125	0	0
10000	2.160927	0.015625	0.046875
100000	Insufficient memory	0.171875	0.265625
1000000	Insufficient memory	3.546875	3.9375

Worse Case Computational Complexity

Bubble Sort Algorithm:

In the worse case scenario, the bubble sort algorithm is given a large list of descending values. This will maximize the number of moves and comparisons needed to sort the list since the combined distance of all the integers from its sorted position is the largest. The algorithm is shown below.

```
def bubbleSort(array):
    while(True):
        swapDetected = False
        for i in range(0, len(array)-1):
            if array[i] > array[i+1]:
                array[i], array[i+1] = array[i+1], array[i]
                swapDetected = True
        if not swapDetected:
            break
```

The outer while loop runs for each integer out of position (in which case a swap is required). In the case of descending order (worst case), every integer in the list is out of position. The while Loop will run for n times for each integer in the list and will break on iteration n since no Swap will be detected. Therefore, the while loop has a computational complexity of O(1) * O(n). Each operation within in while loop is multiplied by the O(n). The break statement is O(1) as it will only execute once when the list is fully sorted. Note that the operations in the for loop is also be multiplied by O(n) and will be shown next.

```
    while(True):             <- O(1) * O(n)
      …
        if not swapDetected: <- O(1) * O(n)
            break            <- O(1)
```

The inner for loop runs once for each integer of the array minus the last, meaning it runs n-1 times within each while loop iteration. Note that the inner for loop is nested within the outer while loop, so the operations of the for loop are multiplied by O(n). This means the inner for loop will run n-1 * n times. Therefore, the for loop has a computational complexity of O(1) * O(n) * O(n). Each operation in the for loop is multiplied by O(n) * O(n). The if statement will run for every iteration of the for loop, so it is O(1) * O(n) * O(n) . The code block within the if statement will run at n, n-1, n-2…times until it reaches 1 time for each while loop iteration. This is because it takes one less move and comparison after each while loop iteration to move the integer to the right position. It can be simplified to O(n) times for each while loop iteration.

```
   for i in range(0, len(array)-1):                    <- O(1) * O(n) * O(n)
      if array[i] > array[i+1]:                        <- O(1) * O(n) * O(n)
         array[i], array[i+1] = array[i+1], array[i] <- O(1) + O(1) * O(n) * O(n)
         swapDetected = True                           <- O(1) * O(n) * O(n)
```

If we sum up the computational complexity of each line while omitting O(1), which are negligible to the overall algorithm complexity, we get the worst case computational complexity of the bubble sort algorithm. The calculation is as follows:

4(O(n) * O(n)) + 2O(n) = 4(O(n^2)) + 2O(n) = 4(O(n^2) = **O(n^2)**

Insertion Sort Algorithm:

In the worst case scenario, the insertion sort algorithm is given a large list of descending values. This will maximize the number of moves and comparisons needed to sort the list since the combined distance of all the integers from its sorted position is the largest. The algorithm is shown below.

```
def insertionSort(array):
    for i in range(1,len(array)):
        val, pos = array[i], i
        while pos>0 and array[pos-1]>val:
            array[pos] = array[pos-1]
            pos = pos-1
        array[pos]=val
```

The outer for loop runs once for every integer of the array except the first, meaning it would run for n-1 number of times. Therefore, the outer loop has a computational complexity of O(1) * O(n). The operations in the for loop are multiplied by O(n). Note the operations of the while loop is also multiplied by O(n) and is shown next.

```
    for i in range(1,len(array)): <- O(1) * O(n)
        val, pos = array[i], i    <- O(1) * O(n)
      …
    array[pos]=val                <- O(1) * O(n)
```

The inner while loop runs once 1, n-(n-2), n-(n-3)…times in each for loop iteration until it reaches n-1 number of time in the last for loop iteration. This is because it takes one more move and comparison for every iteration of the outer for loop. It can be simplified to O(n) times in each for loop iteration. Note that the operations of the inner while loop are nested within the outer for loop, so the operations of the while loop are multiplied O(n) times. Therefore, the computation complexity of the while loop is (O(1) + O(1)) * O(n) * O(n). Each operation in the for loop is multiplied by O(n) * O(n).

```
    while pos>0 and array[pos-1]>val:  <- (O(1) + O(1)) * O(n) * O(n)
       array[pos] = array[pos-1]       <- O(1) * O(n) * O(n)
       pos = pos-1                     <- O(1) * O(n) * O(n)
```

If we sum up the computational complexity of each line while omitting O(1), which are negligible to the overall algorithm complexity, we get the worst case computational complexity of the insertion sort algorithm. The calculation is as follows:

3O(n^2) * 3O(n) = 3O(n^2) = **O(n^2)**

Merge Sort Algorithm:

In the worst case scenario, the merge sort algorithm is given a list which splits into interleaved lists. This guarantees that the algorithm has to make n-1 number of comparisons. The algorithm can do about half the number of comparisons if all integers in one list are smaller than the other in the same split. The algorithm has two components, the main function (mergeSort) and the function called by mergeSort to merge the split lists (merge). The algorithm is shown below.

```
def merge(dest, a1, a2):
    i, j, k = 0, 0, 0
    while i < len(a1) and j < len(a2):
        if a1[i] < a2[j]:
            dest[k]=a1[i]
            i=i+1
        else:
            dest[k]=a2[j]
            j=j+1
        k=k+1
    while i < len(a1):
        dest[k]=a1[i]
        i=i+1
        k=k+1
    while j < len(a2):
        dest[k]=a2[j]
        j=j+1
        k=k+1
```

```
def mergeSort(array):
    if len(array) <2:
        return
    mid = len(array) // 2
    left = array[:mid]
    right = array[mid:]
    mergeSort(left)
    mergeSort(right)
    merge(array, left, right)
```

The merge sort function recursively runs for the number of times the list must be halved until it reaches the case base (represented by variable k) where the length of each list is 1. At the base case, the length of the list n/2^k is equal to 1. This simplifies to k = 2^k or k = lg n. However, can the mergeSort function is called twice within itself, which means there are two recursive runs in parallel (2lg n). Therefore, the computational complexity of the recursive runs of mergeSort is O(lg n). Each operation in the mergeSort function is multiplied by lg n due to the recursive runs. Copying half the list using variable "right" or variable "left" has a computational complexity of n/2 simplifies to O(n). Each operation, including the merge is multiplied by O(lg n) times.

```
def mergeSort(array):           <- O(1) * lg n
    if len(array) <2:           <- O(1) * lg n
        return                  <- O(1) * lg n
    mid = len(array) // 2       <- O(1) * lg n
    left = array[:mid]          <- O(1) * O(n) * lg n
    right = array[mid:]         <- O(1) * O(n) * lg n
    mergeSort(left)             <- O(1) * lg n
    mergeSort(right)            <- O(1) * lg n
    merge(array, left, right)   <- O(1) * lg n * operations in merge function
```

The first while loop runs at maximum 2^k*(n/2^k-1) number of times for each recursive run of mergeSort. Note that the 2^k represents the number of lists in the final iteration, n/2^k represents the number of integers in the list and minus one for excluding i or j equal to length of the list. Therefore, the computational complexity the first while loop can be simplified to O(2^k) * O(n/2^k) or O(n) for each recursive run. Each operation in the first while loop is multiplied by O(n) for each iteration of the recursive. Either of the second or third while loops will execute the deplete the remaining integer. Note that all operations with the merge will be multiplied by each recursive run, meaning it is multiplied by O(lg n).

```
def merge(dest, a1, a2):
    i, j, k = 0, 0, 0                       <- O(1) * O(lg n)
    while i < len(a1) and j < len(a2):      <- O(1) * O(n)* O(lg n)
        if a1[i] < a2[j]:                   <- O(1) * O(n)* O(lg n)
            dest[k]=a1[i]                   <- O(1) * O(n)* O(lg n)
            i=i+1                           <- O(1) * O(n)* O(lg n)
        else:
            dest[k]=a2[j]                   <- O(1) * O(n)* O(lg n)
            j=j+1                           <- O(1) * O(n)* O(lg n)
        k=k+1                               <- O(1) * O(n)* O(lg n)
    while i < len(a1):                      <- O(1) * O(lg n)
        dest[k]=a1[i]                       <- O(1) * O(lg n)
        i=i+1                               <- O(1) * O(lg n)
        k=k+1                               <- O(1) * O(lg n)
    while j < len(a2):                      <- O(1) * O(lg n)
        dest[k]=a2[j]                       <- O(1) * O(lg n)
        j=j+1                               <- O(1) * O(lg n)
        k=k+1                               <- O(1) * O(lg n)
```

If we sum up the computational complexity of each line while omitting O(1), which are negligible to the overall algorithm complexity, we get the worst case computational complexity of the merge sort algorithm. The calculation is as follows:

9(O(n) * O(lg n)) + 15(lg n) = 9(O(n) * O(lg n)) = O(n) * O(lg n) = **O(n lg n)**

Quick Sort Algorithm:

In the worse case scenario, the quick sort algorithm is given a large list of descending values. This will maximize the number of recursive runs required to reach the last iteration, since the list will partition off 1 integer at a time. Note that the first integer is always set as the pivot in the partition function, which leads to the smallest integer in the list becoming the pivot each time. The algorithm is shown below.

```
def partition(a, lo, hi):
    pivot = a[lo]
    i, j = lo, hi
    while True:
        while a[i] < pivot: i = i + 1
        while a[j] > pivot: j = j - 1
        if i >= j: return j
        a[i], a[j] = a[j], a[i]
        i, j = i + 1, j - 1
```

```
def quicksortHelper(a, lo, hi):
    if lo >= hi: return
    p = partition(a, lo, hi)
    quicksortHelper(a, lo, p)
    quicksortHelper(a, p + 1, hi)
```

```
def quickSort(array):
    quicksortHelper(array, 0, len(array)-1)
```

The quicksort function is used to run the quicksortHelper function. The purpose of this is just to ensure that the partitioning process starts from the full list of integers. It runs O(1)*all operations in quicksortHelper.

```
def quickSort(array):
    quicksortHelper(array, 0, len(array)-1)<- O(1) * quicksortHelper operations
```

The quicksortHelper function is used to recursively perform the partitioning process. Since the partitioning only splits one integer off the larger list for each recursive iteration, then the recursive function runs for n-1 number of times. Each operation in the function will also be multiplied performed n-1 number of times. So the computational complexity for each operation is O(1) * O(n). An extra O(1) added for the return function.

```
def quicksortHelper(a, lo, hi):
    if lo >= hi: return              <- O(1) + O(1) * O(n)
    p = partition(a, lo, hi)         <- O(1) * O(n)
    quicksortHelper(a, lo, p)        <- O(1) * O(n)
    quicksortHelper(a, p + 1, hi)    <- O(1) * O(n)
```

Since the partition function is run n-1 number of times, so we know that each operation in the function will also be multiplied by n-1 times. In the first while loop runs for n, n-1, n-2…times until 1 for each recursive iteration. This can be simplified to O(n) for each recursive iteration. This means the first while loop has a computational complexity of O(n) * O(n). Each operation within the first while loop will be multiplied by O(n) * O(n) as a result. Note that due to the second while loop only runs twice for each partition, once in the first iteration of the first while loop and once after integer swap occurs. Therefore in the worst case scenario, the second while loop runs at O(1) * O(n).

```
def partition(a, lo, hi):
    pivot = a[lo]                          <- O(1) * O(n)
    i, j = lo, hi                          <- O(1) * O(n)
    while True:                            <- O(1) * O(n) * O(n)
        while a[i] < pivot: i = i + 1      <- O(1) * O(n)
        while a[j] > pivot: j = j - 1      <- O(1) * O(n) * O(n)
        if i >= j: return j                <- (O(1) + O(1)) * O(n) * O(n)
        a[i], a[j] = a[j], a[i]            <- (O(1) + O(1)) * O(n) * O(n)
        i, j = i + 1, j - 1                <- (O(1) + O(1)) * O(n) * O(n)
```

If we sum up the computational complexity of each line while omitting O(1), which are negligible to the overall algorithm complexity, we get the worst case computational complexity of the quick sort algorithm. The calculation is as follows:

5(O(n) * O(n)) + 7O(n) = 5(O(n) * O(n)) = O(n) * O(n) = **O(n^2)**

Conclusion

Although not shown in our data, it is well known that insertion sort performs better on smaller data sizes while quick sort and merge sort perform better for large data sizes. Limitations of algorithms include memory limitations of recursive algorithms for large data sizes and long runtime for insertion sort and bubble sort. As well, note that merge sort cannot take advantage of partially sorted lists.