Individual Project Report:

Implementation & Analysis of Selection Algorithms

Cameron Wright

February 19th, 2021

CSE4081 Analysis of Algorithms

Dr. William Shoaff

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Preface & Background

This report was assembled as an assignment through January and March of 2021 for CSE4081, Analysis of Algorithms at Florida Tech. Aimed at those interested in selection algorithms, the analysis of them, or anything in between. This report's sole goal is to analyze a selection algorithm, something that after many sleepless nights would be accomplished. This report's literary structure is reminiscent of a journey, giving a linear account of the entire process. Python 3 is used to implement two different, comparable algorithms, one of which could be chosen, the other being the selection algorithm described by Bentley. Both of which find and output the K^{th} smallest element in an array composed of unique elements. The former of the algorithms deemed the naïve-solution, performs a recursive Quicksort on the list using Hoare's Partition scheme, sorting the entire list in ascending order, then outputting the K^{th} element. However, the latter, deemed the smarter-solution, takes in K and uses a recursive select algorithm that selects a random pivot, continuing in the partition containing K^{th} smallest element until found, returning it to be output. Comparing these algorithms, how they are similar and different, in their underlying logic and technical implementation, in correlation with the similarities and differences in their analysis, to gain insight into what makes a good algorithm and the practical difference that can make.

Introduction

In this report, we will be solving a unique, at least in the 1985s, selection problem presented by Jon Bentley using two different algorithms, one to be chosen, the other predetermined to be a selection algorithm described by Bentley (1988). The premise is selecting an element out of a list; however, as Bentley pointed out if the element is the largest or smallest value in a list, the solution would be trivial (1985). As such values could be found traversing the list once, linear time, instead, Bentley detailed a problem with a not-so-apparent optimal solution. Finding an intermediate value to other elements in the list is a real selection problem. Even giving a practical example of such a problem, "how would you find the person on the list who is taller than the 50 shortest people and shorter than the 50 tallest?" (1985, p. 1121). However, programs cannot be run on people, at least not yet. Thus, the problem must be refined to a more variable-centric form. That form being "selecting the Kth-smallest member in a set of N elements" (Bentley, 1985, p. 1121).

However, Bentley allows for duplicate elements in the input list, as seen in the example page 161 (1988). Such a herculean task can be wholly left to him, and instead, to make solving such problems easier and more optimal for the algorithms to be implemented. Input lists, or arrays, as they will be referred to from here on, will consist of unique integers. Additionally, the size of these arrays is also predetermined; however, to better suit the algorithms to be implemented and further illustrate the extremes and,

in turn, allow for a more obvious correlation between the analysis and the practical runtime of the algorithms. Five array sizes will be used instead of the recommended three, those being 100, 1,000, 100,000, 1,000,000, and 10,000,000 elements. Moreover, these arrays' contents will be all unique pseudorandom numbers, *0* to *(N – 1)*, where *N* is the number of elements in the array. (Of note, truly random numbers cannot be produced by current computers, thus for all practical purposes, the term random will mean pseudorandom for the remainder of this report.)

Finally, what technology will be used to conquer the problem of finding the K^{th} smallest element in an array of unique integers? Well, thanks to Bentley, "The ideas in this book aren't limited to any one programming language. Programmers should learn to think in a convenient pseudocode, and then express their ideas in the implementation language" (1986, p. 152). Thus, a choice was made, which is perhaps a retrospectively, the wrong one, but Python is used, as described by its creator Van and Drake, as a language with clear syntax (2009). Furthermore, his Python will be run in a Jupyter Notebook environment. Those unfamiliar with Jupyter, formally known as IPython, described by Thomas Kluyver at a conference accompanied by the rest of the Jupyter Development Team, as "a document format for publishing code, results and explanations in a form that is both readable and executable" (2016, p. 87). This technology-enabled all coding for this report to be written, built, and executed in the same place, displaying code and output in a form as readable as a notebook.

Given that, although the relevant sections of the notebook have been included directly in this report. A full rendering of the notebook containing additional code, comments, and output can be found here hosted in a GitHub repository, in addition to References; be careful of spoilers. (For those interested, additional information about running the code can be found in Additional Remarks.) Alas, now, with a lengthy overview of everything that will be covered in the report, the fun part begins.

Problem Analysis

Unlike other professions and practices, there is no entry bar for starting to code; quality anyone can do it. Furthermore, for those coders that make it so far as to produce something with their code, the allure of immediately starting to code given a problem is ever-present. Conversely, those of us with greater experience coding know that to do so on any non-trivial problem can result in hours, days, or even weeks of wasted time. Thus, the report contains a Problem Analysis section, in which the problem is solved before any practical coding even beings. However, more importantly, allowing for efficient implementation to lay the path for deeper analysis and understanding in the section to come. Before starting with the first possible solution, two things about the solution are pre-determined, as stated in the Introduction. Input will be an array of N containing the unique integers, 0 to $(N - 1)$ in random order, and output will be the K^{th} smallest element in the array. Now the task of coming up with solutions to fit that gap between input and output.

First, the naïve-solution, the most obvious solution being to sort the array in ascending order, simply outputting the K^{th} element. However, this being a classic programming problem, there are many different potential sorting algorithms. Arguably the most basic of these being bubble sort, which traverses the array, $j \times (i-1)$ times, and swaps adjacent elements, *A[i]* and *A[i+1]*, which are not in order until the Array is sorted. Thus, inspired by Cormen's implementation, pseudocode for a downward bubble sort (2009, p.40); this pseudocode for the exact opposite, an upward bubble sort, Figure 1.

```
Bubble-Sort(A)
1     for j in length of A
2         j = j + 1
3         for I in length of A -1
4             i =  i + 1
5       if A[i] > A[i+1]
6                 swap A[i], A[i+1]
7     Bubble-Sort(A)
8     Display A[k]
```

Figure 1. Too-Naïve-Solution Pseudocode

However, Cormen also states that such an algorithm is "inefficient" (2009, p.40). Moreover, attempting to compare this solution to Bentley's selection sort may not be as fruitful as a similar, more efficient algorithm. Thus, this solution will be put on the back burner, and an additional naïve-solution should be found. It would be more beneficial to compare similar solutions, and Bentley's selection sort is very similar to quicksort. A quicksort algorithm would be a good solution, though there are many quicksort flavors, the differentiating factor being how the array is partitioned. This solution will be using

the original portioning scheme created by the creator of Quicksort himself, Hoare. More specifically, based on Cormen's pseudocode of Hoare's Partition (2009, p.185). Along with the standard recursive quicksort function, it was altered to use Hoare's partitioning scheme.

```
Hoares-Partition (A, l, u)
1   p = A[l]
2   i = l - 1
3   j = u + 1
4   while True
5       j = j - 1
6       while A[j] > p
7           j = j – 1
8       i = i + 1
9       while A[i] < p
10          i = i - 1
11  if i < j
12      swap A[j] and A[i]
13  else
14      return j

Quick-Sort  (A, l, u)
15  if l < u
16      p = Hoares-Partition (A, l, u)
17      Quick-Sort  (A, l, p)
18      Quick-Sort  (A, p+1, u)
19   Quick-Sort  (A, 0, length A - 1)
20   Display A[k]
```

Figure 2. Naïve-Solution Pseudocode

This solution works by doing the following, based upon Figure 2, starting with the pivot, selects the first element in the given partition, *l* to *u*, as the pivot, then, on either side of the partition, not including pivot, *l-1*, *i* and *j* converge toward each other, either stopping if an element is found to be out of place compared to pivot. Continuing until the pivot is sorted, meaning that elements below are less and above are greater.

Occurring when *i* and *j* meet, the index at which they met is returned. This partition function is then called within the Quicksort function, which calls itself on two partitions, *upper* and *lower*, on either side of the index returned by the pivot function, *quick-sort (A, p+1, u)* and *quick-sort (A, l, p)*, respectively. Continuing, dividing the partition until the base case, where every partition consists of one element: $l \geq u$, meaning each is a pivot in the correct place; thus, the array is sorted.

The smarter-solution, which is predetermined to be quickselect; this algorithm takes the same approach as the Quicksort, except instead of recursing on both sides of the pivot, it chooses only the side with the K^{th} value. This implementation follows Bentley's pseudocode and description (1988, .p 160 – 163).

```
Quick-Select (A, l, u, k)
1   if lower = upper
2       return A[l]
3   p = a random integer l – u
4   v = A[p]
5   swap A[u] and A[p]
6   p = l
7   for x from l to u
8       if A[x] ≤ v
9          swap A[x] and A[p]
10         p = p + 1
11  swap A[u] and A[p]
12  if k < p
13      Quick-Select (A, l, p-1, k)
14  else if k > p
15      Quick-Select (A, p+1, u, k)
16  else k = p
17      return A[k]
18  Display Quick-Select (A, 0, length A - 1, k)
```

Figure 3. Smarter-Solution Pseudocode

The following happens when Quick-Select, as seen in Figure 3, is called, first check if only one element left in partition, $l = u$, if so return that value, *A[l]* or *A[u]*. Randomly select a pivot, p, within the partition, l to u, also store the value of the pivot, $v = A[p]$, moving pivot out of the way, to the end of the partition, swap *A[u]* and *A[p]*. Then traverse the partition indexes using x; if an element is found less than the pivot, $A[x] \leq v$, element moved below pivot index, *swap A[x] and A[p]*, and move the pivot index up, $p = p + 1$. After reaching partition end, u, move pivot value back to pivot index, *swap A[u] and A[p]*, now pivot is sorted. Now it decides which side of the pivot has the K^{th} element, if K^{th} element is less than the pivot, $k < p$, call self with partition below the pivot, *Quick-Select (A, l, p-1, k)*. Else if the K^{th} element is greater than the pivot, $k > p$, then call self on partition above the pivot. Else the K^{th} element is the pivot, $k = p$, return K^{th} element, *A[k]*. Now with chosen solutions to the problem that contains logic that has been traced through execution, assuming no logical errors, have been proven to be valid solutions. Moreover, those solutions' implementation in pseudocode further cements their validity and proves they can be implemented practically in code.

Implementation

Up to this point, everything has been abstract and independent of specific technologies. However, simply talking about solutions to a problem and putting them into practice are two separate entities entirely, and this is where the latter begins. As mentioned in the Preface and Introduction, the implementation of these solutions will

be done in Python, in a Juypter notebook environment. For reproducibility and readability of solution implementations, the specific scaffolding needed to implement Python solutions must be briefly covered.

```
1  from sys import stdout
2  from copy import deepcopy
3  from random import sample, randint
4  %load_ext memory_profiler
5  %load_ext watermark
6  %watermark -v -iv
```

```
Python implementation: CPython
Python version         : 3.8.5
IPython version        : 7.20.0
```

Figure 4. Implementation Dependencies

The first part of this scaffolding being the specific Python environment; see the output for Figure 4. This implementation uses the current standard Python distribution, CPython 3.8.5, in addition to IPython 7.20.0, which powers the Juypter Notebook. Next, the dependencies, starting with the libraries, *sys*, *copy*, and *random*, are contained in the Python Standard Library, meaning they come with CPython; this is important as updates to CPython come potential updates to these libraries. Furthermore, only the needed modules from each library have been imported and do the following, from *sys*, *stdout* is used to write to the standard output stream. From *copy*, *deepcopy* creates a new, separate duplicate of an object in memory space instead of pointing to the original copy. Finally, from *random*, *sample* returns an array containing unique elements from the population set of size *k*, i.e., *sample(<set>, k)*, and *randint* returns random integers in the inclusive range *a – b*, i.e., *randint(a, b)*.

Next to the extensions, these are packages not included in CPython and instead must be downloaded before use in a program. In addition, when using IPython, they are not imported like libraries but instead use *%load_ext* prefix; see Figure 4. The two-extension used are *memory_profiler* version 2.2.0, which is used to measure memory usage of functions, and *watermark* version 0.58.0, which is used to display the current version of all Python technologies. Downloading these can be done easily using pip, which is included in CPython and is used in the command line. To install the latest version of *memory_profiler*, enter "pip install memory-profiler" (Pedregosa & Gervais, 2020). Furthermore, to download the latest *watermark* version, enter "pip install watermark" (Raschka, 2021).

```
1   def randomize(hundred, thous, hunderedThous, mill, tenMill):
2       hundred = sample(range(0, 100), 100)
3       thous = sample(range(0, 1000), 1000)
4       hunderedThous = sample(range(0, 100000), 100000)
5       mill = sample(range(0, 1000000), 1000000)
6       tenMill = sample(range(0, 10000000), 10000000)
7       return hundred, thous, hunderedThous, mill, tenMill
8
9
10  hundred, thous, hunderedThous, mill, tenMill = [], [], [], [], []
11  arrays = [hundred, thous, hunderedThous, mill, tenMill]
12  randArrays = randomize(*arrays)
13  hundred, thous, hunderedThous, mill, tenMill = deepcopy(randArrays)
14
15  k = randint(1, 100)
```

Figure 5. Input Generation Implementation

Next is how input will be generated; see Figure 5, which will be composed of random integers. All these random integers are generated using modules from the

random library, dependent on the base *random* function. This function generates pseudorandom numbers using the Mersenne Twister algorithm, which is used for "generating uniform pseudorandom numbers" (Matsumoto & Nishimura, 1998). Looking back at implementation, Figure 5, the *randomize* function is responsible for filling the input arrays with unique random integers using the sample module. To ensure consistency, each array contains the same integers in terms of *N*, 0 to *N*-1. Meaning that the K^{th} smallest element will always be *K*-1 for any input; this relationship, in addition to an example array created with *sample*, can be seen in output blow in Figure 6.

```
1  ten = sample(range(0, 10), 10)
2  stdout.write(" ".join(str(x) for x in ten))
3  bubbleSort(ten)
4  stdout.write("\n"+" ".join(str(x) for x in ten))
```

```
3 6 7 5 8 0 9 2 4 1
0 1 2 3 4 5 6 7 8 9
```

Figure 6. Example Array Contents

However, as illustrated in Figure 6, the solutions will alter the input arrays' contents, meaning the original random input arrays must be saved to allow the remaining solution to execute on the same arrays. This process of saving the original input array returned by *randomize* in *randArrays*, before assigning them temporary variables is done on lines 10 - 13, in Figure 5. The reason for this is so the original input arrays can be maintained and reassigned before each solution, ensuring each solution runs on the same set of input arrays. Finally, the K^{th} element randomly generated integer 1 through 100 inclusive, using the *randint* function. The K^{th} element must be within 1

and 100 to fit within all input arrays and excludes 0, as there is no 0^{th} smallest element. Thus, this input implementation allows for each run, in summation, generating a set of random input arrays. Of lengths, 100, 1,000, 100,000, 1,000,000, and 10,000,000, in addition to a random K^{th}, contained within each. Each of these can be maintained through the consecutive execution of each solution. Onto what will be held up by the scaffolding, the solutions; firstly, the naïve-solution, which as discussed in Problem Analysis, is to be Quicksort using Hoare's Partition.

```
1   def hoaresPartition(array, lower, upper):
2       pivot = array[lower]
3       i = lower - 1
4       j = upper + 1
5
6       while True:
7           j -= 1
8           while (array[j] > pivot):
9               j -= 1
10
11          i += 1
12          while (array[i] < pivot):
13              i += 1
14
15          if i < j:
16              array[i], array[j] = array[j], array[i]
17          else:
18              return j
19
20
21  def quickSort(array, lower, upper):
22      if lower < upper:
23
24          pivot = hoaresPartition(array, lower, upper)
25
26          quickSort(array, lower, pivot)
27          quickSort(array, pivot + 1, upper)
28
29
30  def naiveSolution(array, k):
31      quickSort(array, 0, len(array)-1)
32      stdout.write("%dth smallest element in array is %d." %(k, array[k-1]))
```

Figure 7. Naive-Solution implementation

Following the pseudocode in Figure 2, with only minor differences, the conversion of plain English instructions like swap to Python, see line 16 in Figure 7. The expansion of variable names and addition of a pilot function *naiveSolution*, to pass the correct parameters and display the resulting K^{th} smallest element.

Finally, the smarter solution was pre-determined to be Hoare's selection algorithm, aka quicksort. Again, closely following the pseudocode, Figure 3, in its implementation, mirroring those changes in the implementation of the naive solution.

```
1   def quickSelect(array, lower, upper, k):
2
3       if lower == upper:
4           return array[lower]
5
6       pivotIndex = randint(lower, upper)
7       pivotValue = array[pivotIndex]
8       array[upper], array[pivotIndex] = array[pivotIndex], array[upper]
9       pivotIndex = lower
10
11      for index in range(lower, upper):
12          if array[index] <= pivotValue:
13              array[index], array[pivotIndex] = array[pivotIndex], array[index]
14              pivotIndex = pivotIndex + 1
15
16      array[upper], array[pivotIndex] = array[pivotIndex], array[upper]
17
18      if k < pivotIndex:
19          return quickSelect(array, lower, pivotIndex - 1, k)
20      elif k > pivotIndex:
21          return quickSelect(array, pivotIndex + 1, upper, k)
22      else:
23          return array[k]
24
25
26  def smarterSolution(array, k):
27      stdout.write("%dth smallest element in array is %d." %(k, quickSelect(array, 0, len(array)-1, k-1))
```

Figure 8. Smarter-Solution Implementation

Mathematical Analysis

Given the solution now implemented in code, mathematical analysis can be done of time and space complexity. All code snippets used in this section are from their

respective solution Implementation, Figure 7 for the naïve-solution, Quicksort, and Figure 8 for the smarter-solution, Hoare's selection algorithm, quickselect. Before beginning any practical analysis, it is important to take a step back and get an overview of the type of algorithm or algorithms being analyzed. Thankfully, this has already been done during Problem Analysis, and the essential thing to keep in mind about the solutions is, they are recursive. Meaning they call themselves until their base case is reached, which means that after doing a line-by-line trace of the solution and determining complexity, a mathematic pattern or relation that will occur over the recursive calls must be found. Meaning this is the recurrence relation, which can further be used to find time and space complexity.

Starting with the naive-solution, which begins with a call to the aptly named *naiveSolution* function, passing an array and desired K^{th} element, which it then uses to make the initial call to quickSort, passing the array and its bounds.

```
30  def naiveSolution(array, k):
31      quickSort(array, 0, len(array)-1)
```

Here is the first line, which must be assigned runtime. To categorize runtime, each function will be denoted by a letter. For this solution, let $N(N)$ be the runtime of *naiveSolution*, $Q(N)$ the runtime of *quickSort*, and $P(N)$ the runtime of *hoaresPartition*. Now a runtime can be assigned, which would be the runtime of the *quickSort* in addition

to the action of just calling a function, that is $Q(N)$ and 1, as a function call takes constant time, resulting in $N(N) = Q(N) + 1$.

```
21  def quickSort(array, lower, upper):
22      if lower < upper:
23
24          pivot = hoarsPartition(array, lower, upper)
```

Continuing into *quickSort* starting with a check for the base case, *lower* ≥ *upper*, using an if statement, which takes constant time, $Q(N) = 1$. Next is the declaration and an assignment of *pivot,* which both take constant time, now $Q(N) = 3$, and call to *hoaresPartition* taking $P(N)$ time plus the constant time of calling a function, all this resulting in $Q(N) = P(N) + 4$.

```
1   def hoaresPartition(array, lower, upper):
2       pivot = array[lower]
3       i = lower - 1
4       j = upper + 1
```

Within *hoaresPartition,* the first 3 lines, 2 – 4, are all assignment of variables, the first declaring and assigning pivot to *array[lower]*, takes the constant time to declare, assign, and then access array index, $P(N) = 3$. The remaining two lines, both *i* and *j* are declared then assigned, each taking constant time, all together $P(N) = 7$.

```
6       while True:
7           j -= 1
8           while (array[j] > pivot):
9               j -= 1
10
11          i += 1
12          while (array[i] < pivot):
13              i += 1
14
15          if i < j:
16              array[i], array[j] = array[j], array[i]
17          else:
18              return j
```

Continuing, here is the partitioning loop responsible for sorting around and returning the final pivot location once the loop terminates at $i \geq j$. Let the runtime of this outer loop be $O(N)$, continuing line 7, j is assigned to $j - 1$, which takes constant time, $O(N) = 1$. Now the first inner loop, which traverses the array starting from *upper* + 1 to *pivot*, stopping at any element smaller than *pivot*, let the runtime of this first inner loop be $I_1(N), O(N) = I_1(N) + 1$. The only operation in this loop as above takes constant time, $I_1(N) = 1$. Line 11, again being an assignment, takes constant time, $O(N) = I_1(N) + 2$. Reaching the second inner loop, which traverses the array from *lower* – 1 to *pivot*, stopping at any element greater than *pivot*, letting its runtime be $I_2(N)$, having the same constant runtime as $I_1(N), I_2(N) = 1$, mean now $O(N) = I_1(N) + I_2(N) + 2$. Finally, on remaining lines 15 – 18, it determines if pivot location reached or if elements should be swapped. In the latter case, which passes if statement on line 15, taking constant time, $O(N) = I_1(N) + I_2(N) + 3$. The element on either side of the pivot is swapped, line

16, although masked by Python syntax, this requires three assignments and the utilization of a temp variable, resulting in constant time for each, $O(N) = I_1(N) + I_2(N) + 6$. Else the pivot is reached, and pivot is returned, line 18; however, this only occurs during the final outer loop. Bring about the question of how many times this loops.

```
31      quickSort(array, 0, len(array)-1)
```

```
3       i = lower - 1
4       j = upper + 1
```

Determined, given that the array is of size *N*, and as seen in the call on line 31, the upper and lower bound *N* total element on the initial call. Following these bounds further on lines 3,4, they are expanded, now contain $N + 2$. However, this is corrected during the first loop, on lines 7,11, where the bound are reduced, now again containing only N, which is essential because although the pivot's final location is not constant, the number of elements traversed in reaching it is. For example, with a *lower* and *upper* of 0 and 9, even if the pivot is 1, although the pivot is only 1 element from *lower* it is 9 from *upper*, all 10 elements will be covered. Thus, applying this logic, the outer loop, line 6 will run *N* times before i and j meet, finding the final pivot location. Meaning $O(N) = N(I_1(N) + I_2(N) + 6) + 1$, the additional +1 is for the previously mentioned return statement, which will only occur on the final loop. Now plug in the values for the inner loops, $O(N) = N + N + 6N + 1$. Stepping back even further, $P(N) = O(N) + 7$, plugging

what was found for O(n), P(n) = n + n + 6n + 8, converted into big-O, $O(N + N + 6N + 8) = O(N)$.

```
26      quickSort(array, lower, pivot)
27      quickSort(array, pivot + 1, upper)
```

Returning from *hoaresPartition* function back to *quickSort*, the next lines are the recursive calls, one passing the lower section and the other the upper section excluding the previous pivot. However, although when finding the time complexity of *hoaresPartition,* the location of the pivot was of minute importance, it will directly be correlated to this recurrence relation. For example, if every pivot is chosen happens to be the largest element in the array that is not already sorted (previous pivots), the first recursive call on the lower section, line 26, would contain all remaining elements excluding the partition, *N* - 1, while the second call, line 27, would contain nothing. This pattern will repeat, forming an arithmetic series for the input of the first call, $[(N-1), (N-2), (N-3) \ldots]$, while the second call returns. Logically, such a situation will take longer as although both recursive calls are being made, the first call is doing all the work. Another, perhaps easier way to visualize a recursive function is as a tree. The initial call is the root, and each time a recursive call is made, a child node is spawned until the base case is reached. In this instance time, complexity is important, so that each node will contain its big-O runtime. Furthermore, to get the total runtime of the function, take the cost of every level of the tree.

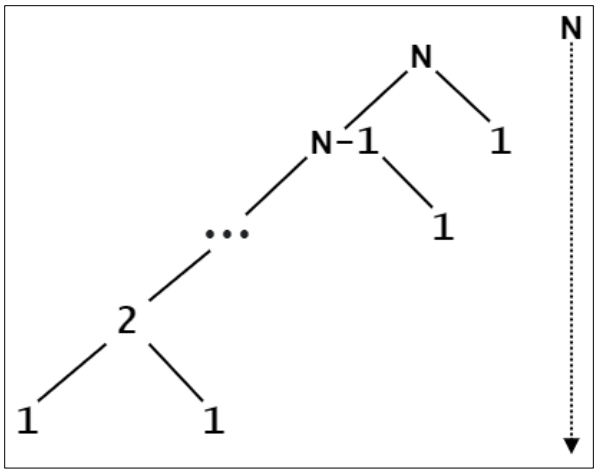

Figure 9. Worst-Case Recursive Tree Quicksort

Here is the resulting tree for one of the worst-cases, as described above, taking the total cost of each level, which is the previously mentioned arithmetic sequence +1 for the other call's constant time at every level except the root, $\{N, N, N-1, \dots, 2\}$. This arithmetic sequence sum can be expressed as $\frac{N(N+2)}{2}$, which in big-O is, $O(N^2)$. Thus, the pivot location matters, as this *naiveSolution* should have a better time-complexity than $O(N^2)$, a time complexity akin to bubble sort, which was already deemed too naïve. However, such a situation is very unlikely, as the input array contains unique random values making it equally likely that any value is selected as the pivot. Moreover, in getting an accurate measurement of runtime following this assumption, each case and its average must be found. Thus, there would be *N* different possible pivot locations, $\{0,1,2, \dots, N-1\}$, creating N different possible partition sets, from $[Q(0) + Q(N-1)]$ to $[Q(N-1) + Q(0)]$, thus, $Q(N) = Q(0) + Q(N-1), Q(1) + Q(N-2), \dots, Q(N-1) + Q(0)$, however as shown above *quickSort* must check for the base case and be partitioned, $Q(N) = N + 4$, in big-O, $O(N)$, so the set is now

$Q(n) = [Q(0) + Q(N-1) + N], [Q(1) + Q(N-2) + N], \ldots, [Q(N-1) + Q(0) + N]$. Given that it is equally likely that any element of set occurs, multiply the set by $\frac{1}{N}$, resulting in, $\frac{1}{N}Q(n) = \frac{1}{N}\{Q(0) + \cdots Q(N-1) + N\} + \{Q(N-1) + \cdots Q(0) + N\}$, and multiply both sides by *N* to cancel out and get a more refined set, $N[\frac{1}{N}\ Q(N)] = \frac{1}{N}[\{Q(0) + \cdots Q(N-1)\} + \{Q(N-1) + \cdots Q(0)\} + N^2\ ]$, and simplify, $Q(N) = \frac{2}{N}\{Q(0) + \cdots Q(N-1)\} + N\}$. Again, multiplying by *N*, duplicating the resulting set, and substituting *N* – 1 for *N*, taking the difference between these resulting sets $\{NQ(N) = 2[Q(0) + \cdots + Q(N-1) + N^2]\} - \{(N-1)Q(N-1) = 2(Q(0) + \cdots + Q(N-2) + N^2 - 2N + 1\} = \ NQ(N) - (N-1)Q(N-1) = 2Q(N-1) + 2N - 1$. Now refined result by taking big-O, $O(NQ(N) - (N-1)Q(N-1) = 2Q(N-1) + 2N)$, rearranging and simplifying, $Q(N) = Q(N-1) + 2N$, now to telescope and iterate, by first dividing everything by $N(N+1)$, $\frac{Q(N)}{N+1} = \frac{Q(N-1)}{N+1} + \frac{2N}{N+1}$. Then iterate negatively along all *N*'s, $\left[\frac{Q(n)}{N+1} + \frac{Q(N-1)}{N-1} + \cdots + \frac{Q(1)}{2}\right] = \left[\frac{Q(N-1)}{N} + \frac{Q(N-2)}{N-1} + \cdots + \frac{Q(0)}{1}\right] + \left[\frac{2N}{N+1} + \frac{2N}{N} + \cdots + \frac{2N}{2}\right]$, add all terms, $\frac{Q(N)}{N+1} = \frac{Q(0)}{1} + 2N(\frac{1}{2} + \frac{1}{3} + \cdots \frac{1}{N+1})$, and now there is a harmonic series, $2N(\frac{1}{2} + \frac{1}{3} + \cdots \frac{1}{N+1})$, which is approximately equal to log, replace with log, then multiply both sides by $N+1$, $Q(N) = 2(N+1)log(N+1)$, and finally take big-O, $Q(N)\ =\ O(N\ lg(N))$. Thus, the time complexity of quickSort is *O(N lg(N)*, assuming all pivots are equally likely. Given the

fractured nature of math in text, the mathematical analysis in its entirety can be found below.

$$Q(n) = [Q(0) + Q(N-1) + N], [Q(1) + Q(N-2) + N], \dots, [Q(N-1) + Q(0) + N]$$

$$\frac{1}{N}Q(n) = \frac{1}{N}\{Q(0) + \cdots Q(N-1) + N\} + \{Q(N-1) + \cdots Q(0) + N\}$$

$$N[\tfrac{1}{N}\ Q(N)] = \tfrac{1}{N}[\{Q(0) + \cdots Q(N-1)\} + \{Q(N-1) + \cdots Q(0)\} + N^2\ ]$$

$$Q(n) = \frac{2}{N}\sum_{x=0}^{N-1} Q(x) + N$$

$$(N)Q(n) = 2\sum_{x=0}^{N-1} Q(x) + N^2$$

$$(N-1)Q(N-1) = 2\sum_{x=0}^{N-2} Q(x) + (N-1)^2$$

$$[(N)Q(n) = 2\sum_{x=0}^{N-1} Q(x) + N^2\ ] - [\ (N-1)Q(N-1) = 2\sum_{x=0}^{N-2} Q(x) + (N-1)^2\ ]$$

$$\{NQ(N) = 2[Q(0) + \cdots + Q(N-1) + N^2]\} - \{(N-1)Q(N-1) = 2(Q(0) + \cdots + Q(N-2) + N^2 - 2N + 1\}$$

$$= NQ(N) - (N-1)Q(N-1) = 2Q(N-1) + 2N - 1$$

$$O(NQ(N) - (N-1)Q(N-1) = 2Q(N-1) + 2N)$$

$$\frac{Q(N)}{N+1} = \frac{Q(N-1)}{N+1} + \frac{2N}{N+1}$$

$$\left[\frac{Q(n)}{N+1} + \frac{Q(N-1)}{N-1} + \cdots + \frac{Q(1)}{2}\right] = \left[\frac{Q(N-1)}{N} + \frac{Q(N-2)}{N-1} + \cdots + \frac{Q(0)}{1}\right] + \left[\frac{2N}{N+1} + \frac{2N}{N} + \cdots + \frac{2N}{2}\right]$$

$$\frac{Q(N)}{N+1} = \frac{Q(0)}{1} + 2N\left(\frac{1}{2} + \frac{1}{3} + \cdots \frac{1}{N+1}\right)$$

$$Q(N) = 2(N+1)\lg(N+1)$$

$$\therefore Q(N) = O[N\lg(N)]$$

Figure 10. Time Complexity Analysis of Quicksort

```
32      stdout.write("%dth smallest element in array is %d." %(k, array[k-1]))
```

However, the naïve-solution time analysis is not complete, now exiting *quickSort* into *naiveSolution* after $O(N\ log\ N)$ time, the K^{th} smallest element in the array at index K-1 is written to standard output. Requiring retrieving the value of *K*, index K-1, and then writing both in a format string to standard out, each taking constant time, $N(N) = Q(N)\ +\ 4$. Now plugging in quickSort time, $N(N) = O(N\ log\ N) + 4$, and taking the big-O, we get the final time complexity, $N(N)\ =\ O(N\ log\ N)$, this the naïve-solution takes linearithmic time.

```
2       pivot = array[lower]
3       i = lower - 1
4       j = upper + 1
```

```
24          pivot = hoaresPartition(array, lower, upper)
```

That still leaves the space used by the naïve-solution; for this, the space of items input will be ignored, as those are declared outside and never returned as the array has a global pointer. Thus, let $S(N)$ be the space of *naiveSolution* excluding input; there are four local variable declarations within this solution. The first three all within *hoaresPartition*, lines 2 - 4, each being an integer taking constant space, $S(N) = 3$. The other in quickSort, line 24, taking in the integer value return by *hoaresPartition* in *quickSort*, but only takes up space for as long as that function is active, meaning constant space, $S(N) = 4$. Something not apparent in the code is the number of times quickSort will return or where all the recursive call information will be stored. The call

stack determines this. Every call to *quickSort* will result in all its relevant information getting stored on this recursive call stack, which means that space can be found by multiplying each function's space by the stack's max depth. Once a base case is met in one of the leaves, that node returns to its parent, all the way back to the root, where the main function then returns. In the worst-case, Figure 9, time complexity, the call stack would occupy *N* space.

However, now having a more accurate, lesser time complexity, $O(N\ lg\ N)$, a corresponding lesser space complexity must be found. Given that the upper bound of space has already been found, finding the lower bound will give insight into the average depth. Given the best possible case, the pivot divides the partitions in half every call, creating two even subproblems, resulting in the recursive tree below, in Figure 11.

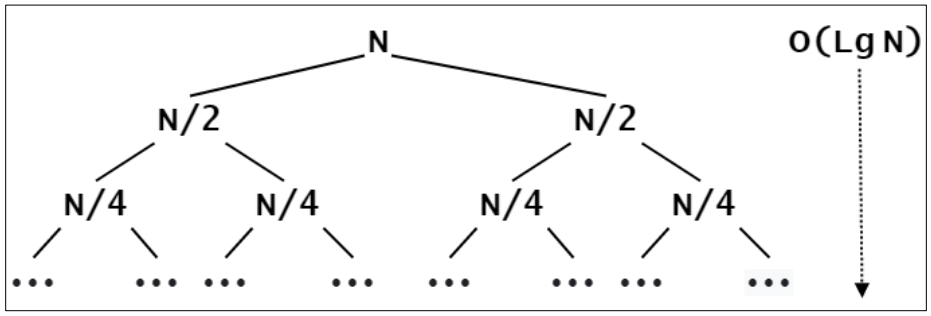

Figure 11. Best-Case Recursive Tree Quicksort

Resulting in this tree of depth $lg(N)$, furthermore knowing that our tree will be better than the worst-case $O(N)$ and be no better than the best-case $O(lg\ N)$, it can be concluded that our average tree's depth is contained in $O(lg\ N)$. Thus, average space complexity would be $S(N) = 4 \cdot O(lg\ N)$, taking big-O once more, we get the final

average space complexity for *naiveSolution*, $S(N) = O(lg\ N)$. Thus, it can be concluded that the naïve-solution takes logarithmic space.

```
26  def smaterSolution(array, k):
27      stdout.write("%dth smallest element in array is %d." %(k, quickSelect(array, 0, len(array)-1, k-1))
```

The analysis of time and space complexity for the smarter-solution will be done in the same way as the naïve-solution. Starting in the smarter-solution driver function, letting the runtime of *smarterSolution* be $S(N)$, the driver function retrieves, *k* and calls *quickSelect* passing an array, 0, *len(array)*-1 and *k*-1, this action, including the *len* function, takes constant time, $S(N) = 2$.

```
1   def quickSelect(array, lower, upper, k):
2
3       if lower == upper:
4           return array[lower]
```

Now into the *quickSelect* function, which first checks one of the two base cases, this one is the same as *quickSort*, if the lower and upper partition meet. Let the runtime of *quickSelect* be $Q(N)$. The if statement and comparison on line 3 take constant time; on line 4, returning an element in an array will also take constant time but will only occur if this base case is reached. Given that this base case is not always reached, line 4 will not be factored into $Q(N)$, but line 3 will, $Q(N) = 1$.

```
6       pivotIndex = randint(lower, upper)
7       pivotValue = array[pivotIndex]
8       array[upper], array[pivotIndex] = array[pivotIndex], array[upper]
9       pivotIndex = lower
```

Further on, line 6, a variable *pivotIndex* is declared and assigned a random integer between the lower and upper bound, generated by *randint*, both taking constant time, $Q(N) = 3$. On the next line, *pivotValue* is declared and set equal to the element at *pivotIndex* in the array, declaration and retrieving element both take constant time, $Q(N) = 5$. Next line, line 8, the elements at the indexes *upper* and *pivotIndex* are swapped, as mentioned in the analysis of the naïve-solution, which is done using a temporary variable and takes constant 3, $Q(N) = 8$. On the next line, line 9, *pivotIndex* is assigned the value of *lower*, taking constant time, $Q(N) = 9$.

```
11      for index in range(lower, upper):
12          if array[index] <= pivotValue:
13              array[index], array[pivotIndex] = array[pivotIndex], array[index]
14              pivotIndex = pivotIndex + 1
15
16      array[upper], array[pivotIndex] = array[pivotIndex], array[upper]
```

Now, the loop responsible for partitioning the elements around the pivot is reached; on line 11, *index* will iterate from the value of *lower* to *upper*, given that *quickSelect* is initially called with 0 as *lower* and *N-1* as *upper*, this will loop N times, let the runtime of this loop be $I(N)$. On line 12, an if statement checks if the element in the current index is less than or equal to *pivot*, taking constant time, $I(N) = 1$, if the condition is true, the value at *index* is swapped with *pivotIndex*, which start at *lower*,

taking a constant time of 3. After that, on line 14, *pivotIndex* is moved forward as that index now contains an element less than or equal to the pivot, taking constant time of 1. Now reaching the end of the loop, it still is at a constant time of 1 per loop, as lines 13 and 14 are not may never be reached; thus, only the if statement, line 12, will run every loop. Given that $I(N) = 1$, and the loop will run *N* times, the runtime must be multiplied by *N*, therefore $I(N) = N$. Before moving on, this loop's runtime must be included in the runtime for *quickSelect*, $Q(N) = I(N) + 9$, plug-in loop runtime,$Q(N) = N + 9$.

```
18    if k < pivotIndex:
19        return quickSelect(array, lower, pivotIndex - 1, k)
20    elif k > pivotIndex:
21        return quickSelect(array, pivotIndex + 1, upper, k)
22    else:
23        return array[k]
```

Here is where the smarter-solution diverges from the naïve-solution, starting on line 18; if the K^{th} element is less than *pivotIndex*, the K^{th} element is in the lower partition, return it excluding the pivot. Else if the K^{th} element is greater than *pivotIndex*, the K^{th} element is in the upper partition, return it, excluding the pivot. Finally, else, the second base case, if the K^{th} element is not above or below *pivotIndex*, it must be the K^{th} element, return K^{th} element. These checks will take constant time; however, they do not occur unless this second base case is met. However, the minimum amount of time can be chosen as it will take at least that time in all instances, which occurs when the first if statement, line 18, is true, resulting in constant time, $Q(N) = Q(N) = N + 10$. Reaching

the end of *quickSelect*, before moving forward, the runtime should be refined, taking big-O, $Q(N) = O(N)$.

Given that it has already been established that pivot location is important for the time in such partitioning algorithms, it may still be beneficial to find the worst-case time complexity. With the knowledge of quickSort and that it has a worst-case of $O(N^2)$, the worst-case of *quickSelect* can be found easily. Although *quickSelect* only recurs on a single partition, if every pivot were the greatest remaining element, and the final pivot was the K^{th} element, it would run similarly to *quicksort*'s worst-case. As the partition would contain all element – pivot, creating an arithmetic series identical to *quickSort*, the only difference being there would be no recurrence on the empty set.

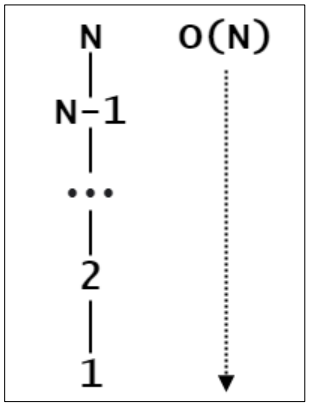

Figure 12. Worst-Case Recursive Tree Quickselect

Figure 12 is the resulting recursive tree of the worst-cases for *quickSelect*, which would have the same big-O as the tree in Figure 9, $O(N^2)$. Thus, the worst-case for *quickSelect* has the same big-O time complexity as quickSort. However, one important distinction between these implementations of *quickSort* and *quickSelect* is the random

pivot. Although these input arrays are random, having a sorted array may be more likely than any given configuration under normal circumstances; This is important as the worst-case in *quickSort* is on a sorted array, while the worst-case would be some array where the greatest element just happened to be at the random pivot location for every pivot.

Now onto the average case, looking back at *quickSort*, this should be similar, except instead of having both sides of the pivot, there will only be one. Thus, there are *N* possible pivots, 0 to *N*-1, with *N* corresponding partitions, *N*-1 to 0, each of which is possible during *quickSelect*, $Q(N) = Q(0), Q(1), \dots, Q(N-1)$. Additionally, the cost of looping to find the pivot must be added to each call. Which was found to be $O(N)$, therefore $Q(N) = Q(0) + N, Q(1) + N, \dots, Q(N-1) + N$. Again, following what was done in find time complexity in the naïve-solution, divide both sides by $\frac{1}{N}$, $\frac{1}{N}Q(N) = \frac{1}{N}\{Q(0) + N, Q(1) + N, \dots, Q(N-1) + N\}$, then multiply both sides by *N*, $N[\frac{1}{N}Q(N)] = \frac{1}{N}[\{Q(0), Q(1), \dots, Q(N-1)\} + N^2]$, simplifying into the following relation, $Q(N) = \frac{1}{N}\sum_{x=0}^{N-1} Q(x) + N$ an equation identical to *quickSelect* except $\frac{1}{N}$ instead of $\frac{2}{N}$. As only one recursive call is made. Then still following what was done in the naïve-solution, multiply both sides by *N*, $N\ Q(N) = \sum_{x=0}^{N-1} Q(x) + N^2$ and then substitute *N* with *N*-1, $(N-1)Q(N) = \sum_{x=0}^{N-2} Q(x) + (N-1)^2$, taking the difference of these equations, $NQ(N) - (N-1)Q(N-1) = Q(N-1) + 2(N-1)$. Solve for *Q(N)* by dividing both sides by *N*, $Q(N) = Q(N-1) + 2(1 - \frac{1}{N})$, then simplify further using the summation formula,

$Q(N) = Q(1) + 2\sum_{x=1}^{N}(1 - \frac{1}{x})$. Identifying the base case $Q(1) = 0$, and the harmonic series, $\frac{1}{x}$, then expanding the summation, $Q(N) = 2(N - \left[\frac{1}{1}, \frac{1}{2}, \ldots \frac{1}{N}\right])$, replacing the harmonic series with lg as before, $Q(N) = 2(N - \lg N)$, and finally take big-O, $Q(N) = O(N)$. Again, as with the *quickSort*, the mathematical analysis in its entirety can be found below, Figure 12.

$$Q(N) = Q(0), Q(1), \dots, Q(N-1)$$

$$Q(N) = Q(0) + N, Q(1) + N, \dots, Q(N-1) + N$$

$$\frac{1}{N}Q(N) = \frac{1}{N}\{Q(0) + N, Q(1) + N, \dots, Q(N-1) + N\}$$

$$N\left[\frac{1}{N}Q(N)\right] = \frac{1}{N}[\{Q(0), Q(1), \dots, Q(N-1)\} + \{N^2\}]$$

$$Q(N) = \frac{1}{N}\sum_{x=0}^{N-1} Q(x) + N$$

$$NQ(N) = \sum_{x=0}^{N-1} Q(x) + N^2$$

$$(N-1)Q(N) = \sum_{x=0}^{N-2} Q(x) + (N-1)^2$$

$$[NQ(N) = \sum_{x=0}^{N-1} Q(x) + N^2] \; - \; [\,(N-1)Q(N) = \sum_{x=0}^{N-2} Q(x) + (N-1)^2]$$

$$= NQ(N) - (N-1)Q(N-1) = Q(N-1) + 2(N-1)$$

$$NQ(N) = NQ(N-1) + 2(N-1)$$

$$Q(N) = Q(N-1) + 2(1 - \frac{1}{N})$$

$$Q(N) = Q(1) + 2\sum_{x=1}^{N}(1 - \frac{1}{x})$$

$$Q(N) = 2(N - \left[\frac{1}{1}, \frac{1}{2}, \dots \frac{1}{N}\right])$$

$$Q(N) = 2(N - \lg N)$$

$$\therefore Q(N) = O(N)$$

Figure 13. Time Complexity Analysis Quickselect

```
26  def smaterSolution(array, k):
27      stdout.write("%dth smallest element in array is %d." %(k, quickSelect(array, 0, len(array)-1, k-1))
```

As before, the time analysis of the smarterSolution is not complete, as once the base case of *quickSelect* is reached, it returns the Kth integer, *k*, to the driver function. This function then writes *k* and returns from *quickSelect* to standard output, these actions taking constant time, meaning $S(N) = Q(N) + 3$, plug in the average time found for *quickSelect*, $S(N) = O(N) + 3$, again taking big-O to find the final time complexity, $S(N) = O(N)$; thus, the smarter-solution takes linear time.

```
6       pivotIndex = randint(lower, upper)
7       pivotValue = array[pivotIndex]
```

Now into space complexity for *smarterSolution*, as before, this will not include the input items' space. Let the space taken by *smarterSolution* be $S(N)$. There are two local variables within *quickSelect*, lines 6,7, *pivotIndex* and *pivotValue*, both taking up constant space, $S(N) = 2$. However, as in *quickSort*, most of the space used is due to recursion. As each call must be stored on the recursive call stack, all of which will return when the base case is reached. Meaning space complexity can be found based on the depth of the recursion tree. Unlike in *quickSort,* the best-case for *quickSelect* is when the first pivot is chosen is the Kth element, as it would only partition the array once before returning, with constant space complexity, $O(1)$. Conversely, in the worst-case, see Figure 12, the space complexity is $O(N)$. Looking at a more average case, where the

pivot is always around the list's median value and, but the Kth element is not found until the entire list is or is almost sorted.

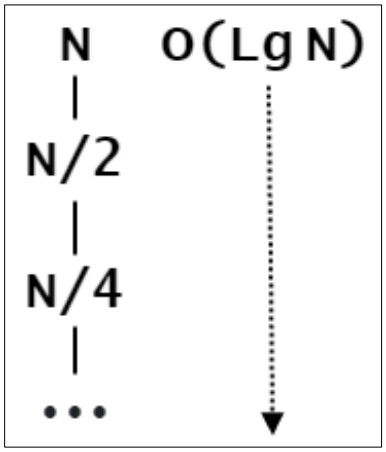

Figure 14. Average Recursive Tree Quickselect

Above is the recursive tree for that case, Figure 14, given that the best-case $O(1)$ and the worst-case $O(N)$, bound the average space complexity, and some case around the average case has *O(lg N)*. It can be concluded that the recursive stack in the average takes up logarithmic space. Updating the space complexity for *smarterSolution*, $S(N) = 4 \cdot O(lg\ N)$, again taking big-O to get the final space complexity, $S(N) = O(lg\ N)$. Thus, it can be concluded that the smarter-solution takes logarithmic space.

Practical Results

With a complete analysis of space and time complexity for both implemented solutions, their validity can be elicited using definite space and time values.

```
1  %%timeit -r 1 -n 1
2  solution(array,k)
```

Kth smallest element in array is K-1.

<measurement> (<unit>) ± 0 <unit> per loop (mean ± std. dev. of 1 run, 1 loop each)

Figure 15. Runtime in Implementation

Just before Figure 15, it shows how runtime will be measured using the *timeit* module from the *memory_profiler* extension. The function is preceded by *%%*, which indicates that it will time the whole cell. Also, two options are used *-r* and *-n*, indicating the number of runs and loops, respectively.

Array size	Unit	Naïve-Solution	Unit	Smarter-Solution
100	*µs*	226, 534, 206, 139, 160, 468, 141, 311, 150, 143, 150, 564, 140, 140, 342, 142, 149, 294, 139, 150	*µs*	59.1, 80.5, 72.7, 141, 151, 83.5, 52.2, 95.6, 67.2, 128, 61.2, 171, 72.5, 63.5, 103, 72.2, 46.5, 73.9, 75.9, 64.7
1000	*ms*	1.78, 1.99, 1.71, 1.69, 2, 1.65, 1.78, 2.72, 1.69, 1.82, 1.65, 1.83, 1.72, 1.63, 1.81, 1.77, 1.65, 1.83, 1.95, 1.67	*µs*	369, 479, 368, 577, 453, 301, 261, 174, 441, 265, 252, 231, 310, 461, 168, 136, 262, 256, 329, 482
100000	*ms*	269, 263, 258, 257, 276, 256, 255, 266, 258, 255, 259, 270, 254, 265, 255, 279, 256, 257, 253, 271	*ms*	10.4, 6.42, 25.6, 24.9, 21, 37.5, 17.4, 32.1, 12.1, 7.64, 66.1, 43.3, 39.4, 36.3, 33.6, 32.5, 35.7, 12.2, 38.2, 7.81
1000000	*s*	3.44, 3.47, 3.22, 3.31, 3.28, 3.23, 3.33, 3.53, 3.24, 3.19, 3.34, 3.21, 3.2, 3.17, 3.29, 3.25, 3.38, 3.31, 3.33, 3.22	*ms*	449, 303, 471, 398, 467, 320, 411, 562, 224, 210, 292, 231, 121, 357, 395, 152, 443, 273, 110, 329
10000000	*s*	39.5, 42.4, 40.1, 40.2, 40.8, 42.6, 40.4, 41.6, 39.7, 41.8, 42.2, 39.9, 41.8, 42.5, 42.1, 40.8, 41.2, 40.9, 41.3, 40.6	*s*	5.9, 6.58, 4.83, 5.62, 3.44, 7.64, 7.98, 5.19, 7.38, 2.91, 1.66, 4.08, 4.09, 2.31, 2.55, 10, 3.12, 6.91, 1.35, 7.43

Table 1. Runtime Data Over 20 Runs

$$\left\{\frac{(RT_1 \ldots RT_n)}{n}\right\} \Big/ Ratio\ w.r.t\ Seconds$$

Figure 16. Equation for Average Runtime in Seconds

Array size	Naïve-Solution (s)	Smarter-Solution (s)
100	0.00023435	0.000086755
1000	0.0018165	0.00032875
100000	0.26155	0.027009
1000000	3.2965	0.3259
10000000	41.115	5.0485

Table 2. Seconds Homogenized Average Runtime Data

The runtimes of 20 runs, each having an initial randomized K and arrays that each solution would solve, are in Table 1. However, some runtimes have different time units; to correct this, the equation in Figure 16 takes the average converting into seconds. The resulting values can be seen above in Table 2.

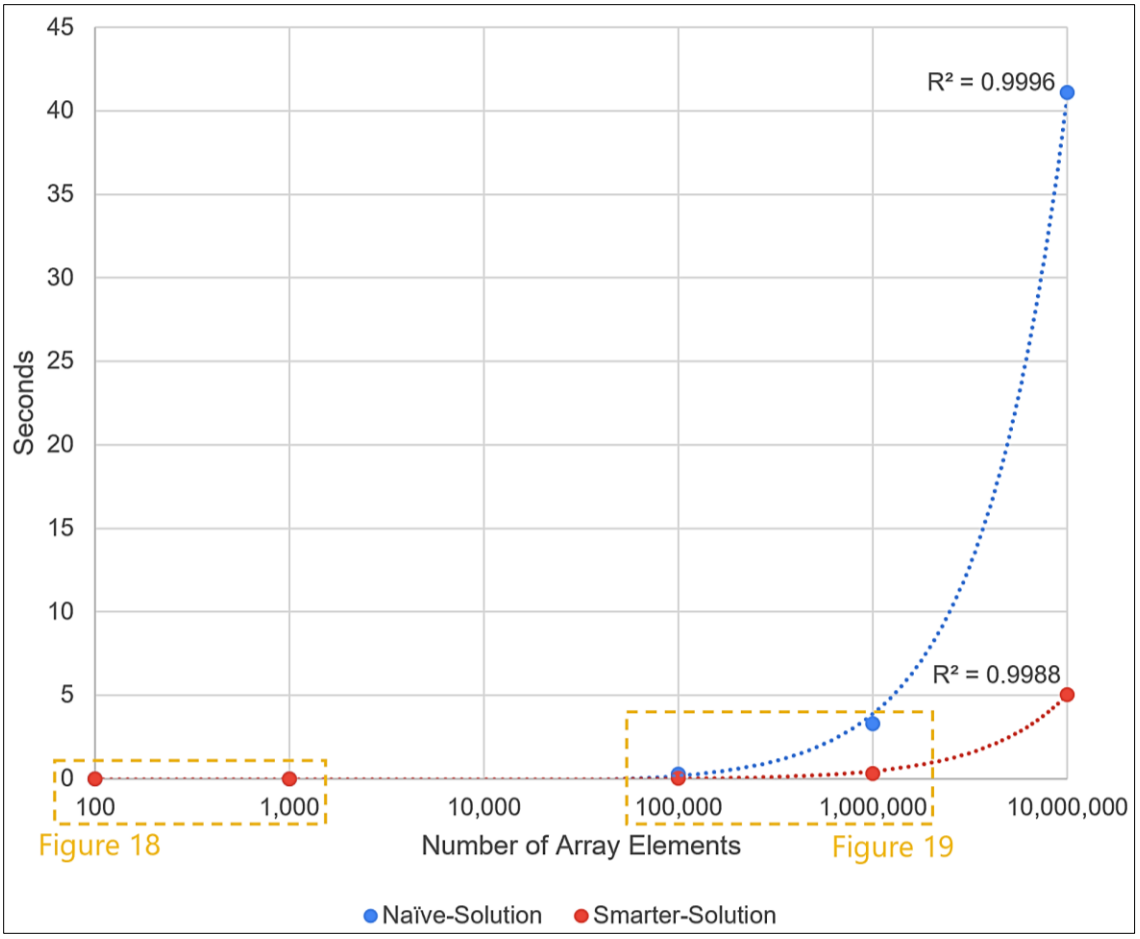

Figure 17. Average Runtime with Trendlines & R^2

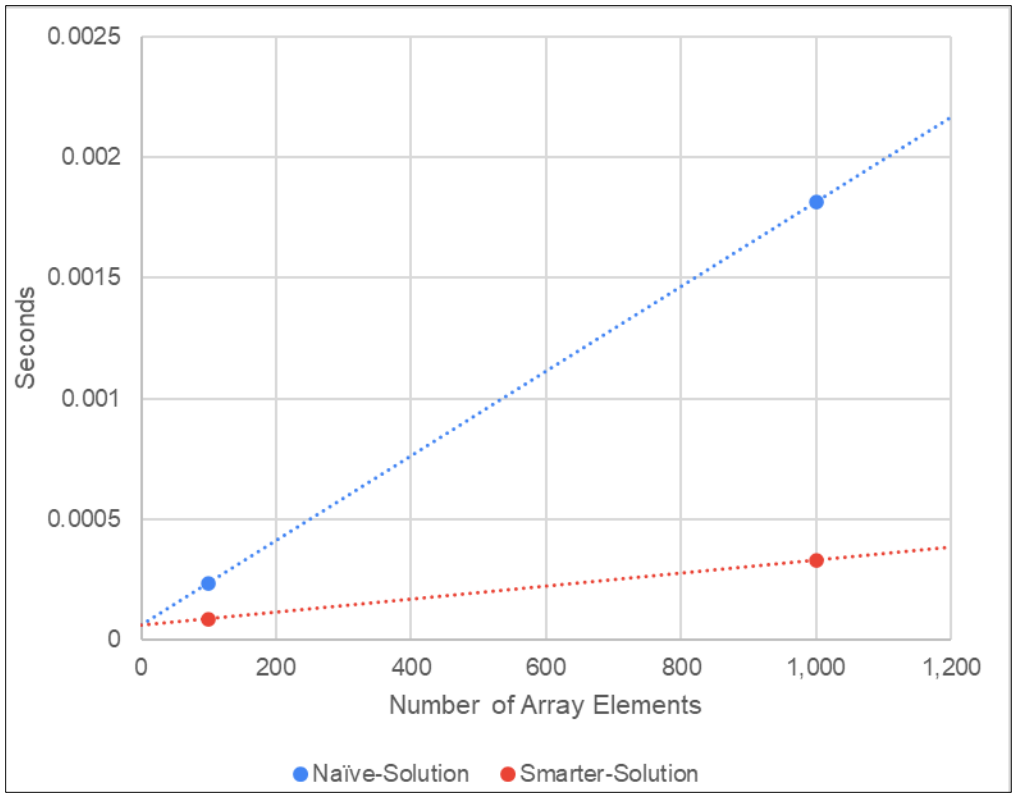

Figure 18. Average Runtime of 100 & 1,000 Elements

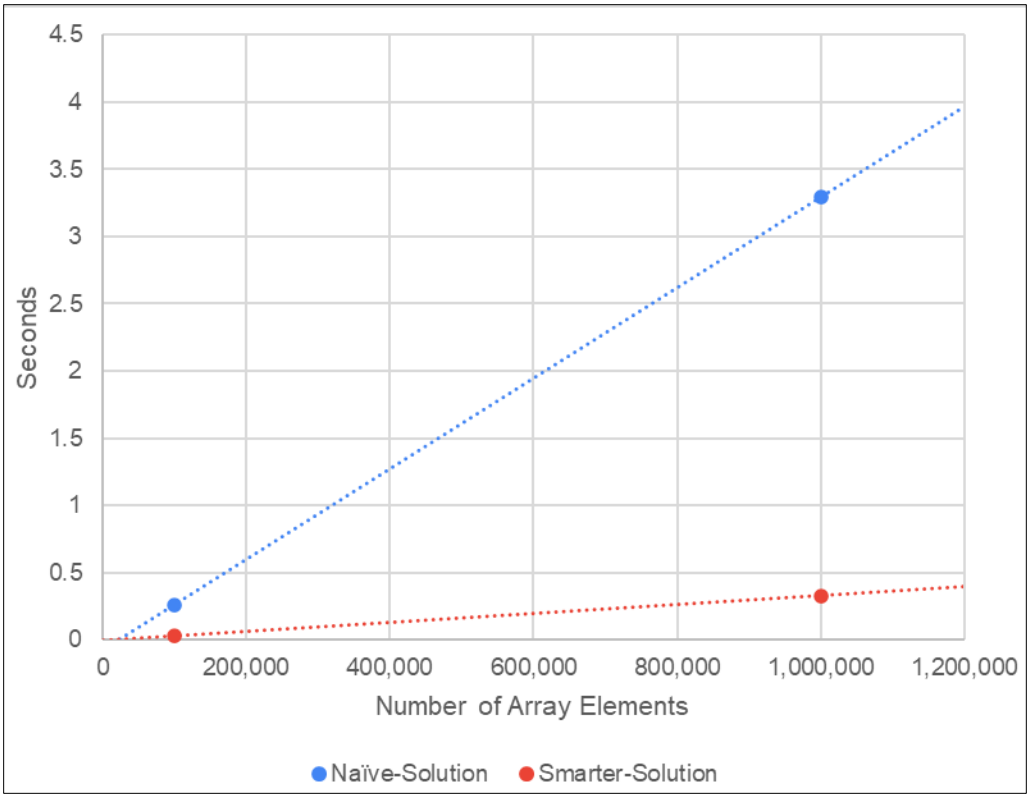

Figure 19. Average Runtime of 100,000 & 1,000,000 Elements

These values for average runtime in Table 2 are used in the scatter plot graph seen in Figure 17, showing the correlation between the input array size, meaning *N*, and runtime for both solutions. The large disparity in the smallest and largest array caused a related disparity in the smallest and largest runtimes. The relation and information for lesser input regions of Figure 17 are unclear; thus, those regions are expanded in Figures 18 and 19.

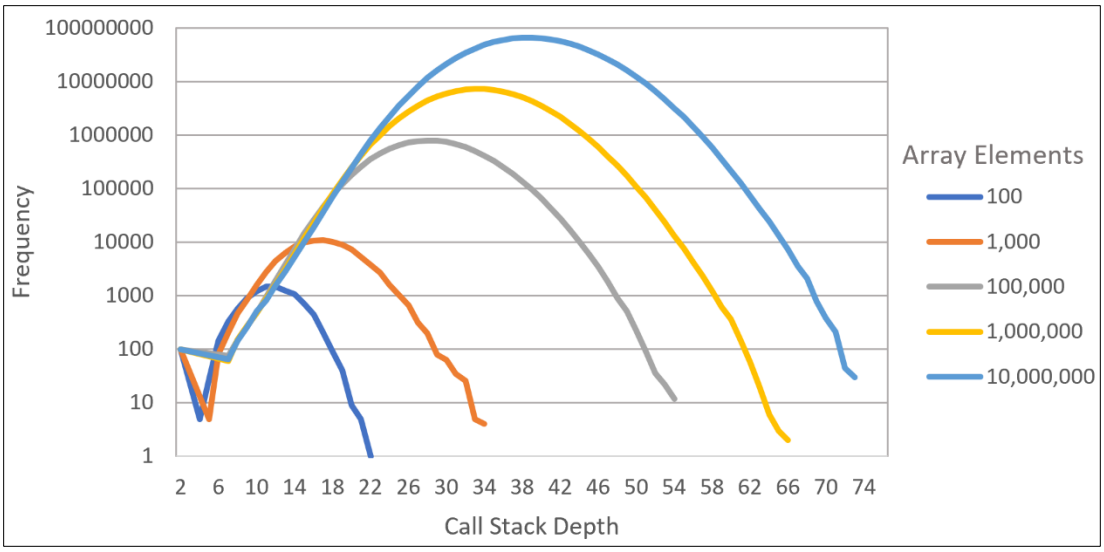

Figure 20. Naïve-Solution Recursion Pattern & Depth

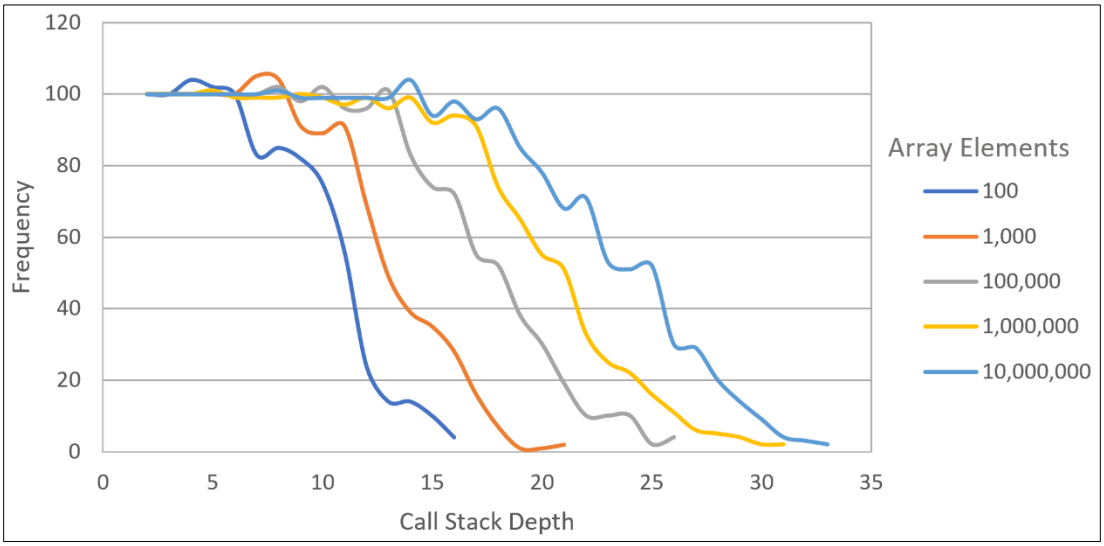

Figure 21. Smarter-Solution Recursion Pattern & Depth

Lastly, the above figures, Figures 20 and 21, show the C call stack's depth relating to the number of times that depth occurs. This data was collected utilizing the *_getframe()* function from *sys*, which was inserted before, on the first line, and in each solution's base cases. Each solution was run 100 times for each array size, using a randomized K and array every run. Withstanding, the data for the naïve-solution with 10,000,000 element input array uses a dataset of 20 runs that is multiplied by 5 due to the amount of output per run (10 million depths). Additionally, this was not included in the notebook due also to the size of the output.

Discussion

The results reflect the relationship expected of both complexity's, time Figures 17–19, and space Figures 20 and 21. However, as demonstrated in the difference of the call stack depth, Figures 20 and 21, there are extreme limitations when comparing big-O complexity to practical results. Due to both the refined nature of big-O and the nuances of practical computation, they cannot be compared in their given states, and thus many assumptions must be made when attempting to compare them. Primarily that there exists some constant k, such that $k \cdot complexity = practical\ results$, where k is the computational time. Moreover, such a comparison is impractical given the nature of big-O. Instead, a better comparison is that of other big-O complexities.

As such, when examining the results of time complexity, it is apparent that the local relationship between the solutions reflects one of *O(N lg N)* and *O(N)*, that is, the

naïve-solutions *O(N lg N)* bounds the smarter-solution *O(N)*. They could, however, be some other set of big-O functions that bound. Thus, a solution is to bound both solutions' practical runtime with algorithms' practical runtime with known complexities.

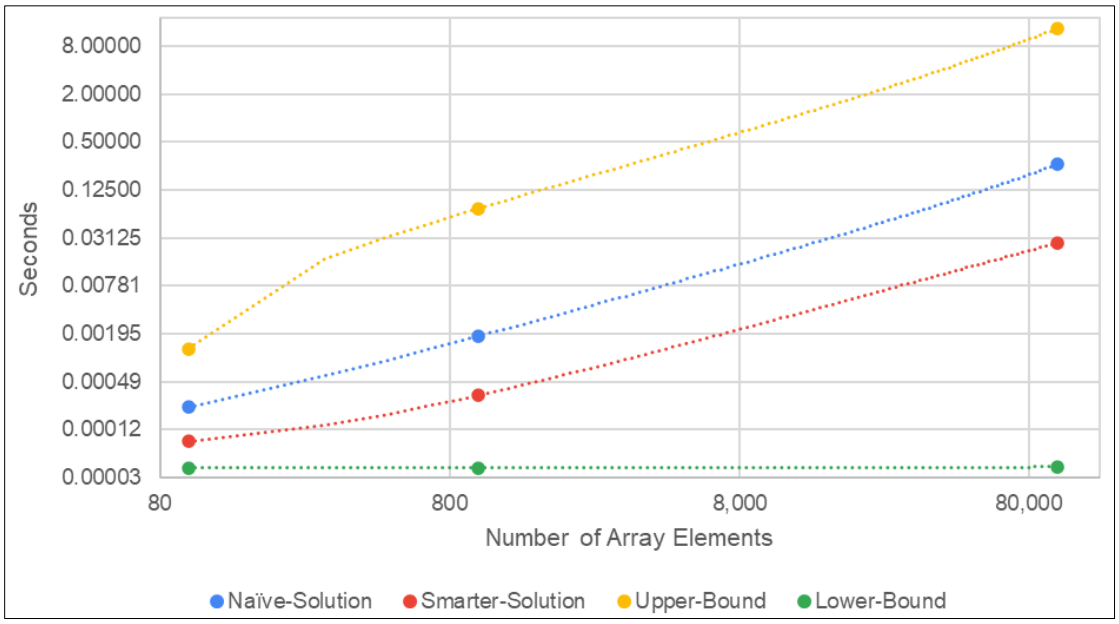

Figure 22. Bounded Solution Results

That is what was done as seen above, Figure 22, using arrays of size 100, 1,000, and 100,000. As with the solutions, 20 runtime tests were conducted on two additional algorithms. The upper-bound algorithm is bubble sort which, according to Verma and Singh, has a "time complexity of bubble sort is O (N^2)" (2015, pp. 381). What is more, the lower-bound algorithm was a binary search which Vuyyuru points out "works with time complexity of *O(log_2 N)* time" (2019, pp. 185). Thus, given these bound the solutions as expected, meaning lower-bound *O(lg N) < smarter-solution O(N) < naïve-solution O(N lg N) < upper-bound O(lg N)*. Confirming the results of the analysis of runtime for both solutions. The implementation of these bounding algorithms Now into space

complexity, the analysis shows that both solutions have an expected complexity of *O(lg N)*, although, in the practical results, there is variance between the solutions. This is most likely due to the naive nature of big-O. As before, bounding the complexity by the contiguous big-O terms will prove the analysis. This time simply looking at the max call stack depth, the big-O can be proved as they are neither constant nor exponential. That is $O(1) < space\ used < O(N^2)$, implying that $space\ used = O(lg\ N)$. Thus confirming the conclusion that each solution has a logarithmic space complexity.

Conclusion

The average case for the time and space complexity of two solutions to the 1985 Bentley selection problem asked to find the K^{th} element in a list. First, a naive-solution using *quickSort* and secondly, a smarter-solution using a *quickSelect* were found and proven. Through the line-by-line analysis finding base complexity followed by mathematical analysis, the average case time complexity was found, then profiled the results bounded proving their validity. Those results showed that, on average, the naïve-solution took *O(N lg N)*, and the smarter-solution took *O(N)*. Furthermore, through a similar analysis, using recursion trees, the average space complexity was found and then bounded, proving the validity; those complexities were on average, that both the naïve-solution and smart-solution took *O(N lg N)* space.

These findings illustrate how a problem is viewed, and in turn, the approach is integral to the efficiency of the resulting solution and implementation. In the same vein,

a re-examination of a solution by others is integral to improve its implementation further. The disparity in the two views to solutions of the selection problem, the naïve-solution stemming from the view that the solution was to sort the list and, the smarter-solution stemmed from Hoare's view that the solution was to search the list. This disparity in views caused a fundamental difference in approach during implementation, which would result in a difference of runtime in favor of the smarter solution. However, the difference would further grow as through re-examination of Hoare's quicksort by Bentley differing views enable further improvement resulting in this report's implementation.

Thus, with both the differing views and re-examination, the two solutions that utilize recursion on a pivot have differing significant terms for average time complexity. In practical terms, while testing on a list of 100,000,000 elements, the naïve-solution took on average ~41 seconds while the smarter-solution took ~5 seconds. In summation, as emphasized in Problem Analysis, it is essential to do a complete assessment of all different theorized solutions. Furthermore, the careful and complete culmination and analysis of different views is essential to an effective approach to implementing and further improving a solution.

Reflection

Firstly, choosing algorithms to compare, the complexity of algorithms, if possible, choose similar algorithms, the initial naïve-solution for this report was to be bubble sort.

However, when comparing the practical results of *bubbleSort* to the smarter-solution, *quickSelect*, the literal exponential runtime of *bubbleSort* meant it could not reasonably be run on any array over 100,000 elements.

```
1  %%timeit -r 1 -n 1
2  tooNaiveSolution(hunderedThous,k)
```

```
18th smallest element in array is 17

18 min 31s ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

Figure 23. Too-Naïve-Solution Runtime

Figure 23 demonstrates the abysmal run times on the array of 100,000 elements; additionally, the Y-axis in Figure 22 had to be scaled logarithmically to keep *bubbleSort's* runtime within the graph. Furthermore, comparing two vastly different algorithms as bubble sort and quickselect gives little insight as, upon cursory inspection, the complexity difference and why it exists is clear.

Secondly, the technology to use, Python, due to its syntax, is an excellent visual language that an IPython Notebook further complements. What Python is not is a language to convey detailed time complexity. Being a high-level language built on top of C, often many instructions are condensed into one. Furthermore, when comparing algorithms like those in this report, all sub 40 lines, a lower-level language such as C should be used to do a better analysis of complexity.

Recommendation

When attempting to solve a problem where efficiency is integral, as observed from the differences in methods and results of the naïve-solution and smart-solution, first, come up with different ways to view the problem. Attempt to theorize possible solutions from each of those views, analyze those, and pick the best after implementing, repeatedly coming up with different views, critically looking at the implemented solution for improvements. Additionally, have others do the same until no further improvements can be found. The time put into the solution of a problem, especially when taking differing approaches, correlates with the resulting solution's quality.

Additional Remarks

For those interested, all relevant code and output can be found in the GitHub repository in References. Furthermore, this code can be run on a machine in two ways. Using an IPython notebook, download the *.ipynb* file else using traditional Python 3, the *.py* file. Note that the *memory-profiler* and *watermark* extensions must be downloaded as detailed in the Implementation section for the full functionality.

References

Bentley, J. L. (1985). *Programming Pearls: Selection*. Communications of the ACM, 28(11), 1121–1127. DOI: 10.1145/4547.315131

Bentley, J. L. (1986). *Programming Pearls*. Boston, MA: Addison Wesley.

Bentley, J. L. (1988). *More programming pearls: Confessions of a coder*. Boston, MA: Addison Wesley.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). London, England: MIT Press.

Matsumoto, M., & Nishimura, T. (1998). *Mersenne Twister*. ACM Transactions on Modeling and Computer Simulation, 8(1), 3-30. DOI:10.1145/272991.272995

Pedregosa, F., & Gervais, P. (2020, October 19th). *Memory-profiler 0.58.0*. A module for monitoring memory usage of a python program.

Raschka, S. (2021, February 18th). *Watermark 2.2.0*. IPython magic function to print date/time stamps and various system information.

Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn and Kurt Smith. *Cython: The Best of Both Worlds, Computing in Science and Engineering*, 13, 31-39 (2011), DOI:10.1109/MCSE.2010.118

Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, Carol Willing, & Jupyter development team (2016). *Jupyter Notebooks - a publishing format for reproducible computational workflows*. IOS Press.

Wright, E. Cameron. (2021). A pre-executed Jupyter (IPython) Notebook containing all code and profiling tools used in this report. *Individual-Project-Notebook*. DOI:10.5281/zenodo.4635665

Van Rossum, G., & Drake, F. L. (2009). *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace.

Verma, R., & Singh, J. (2015). *A comparative analysis of deterministic sorting algorithms based on runtime and count of various operations*. International Journal of Advanced Computer Research, *5*(21), 380-385.

Vuyyuru, G. M. (2019). *KLP's Search Algorithm - A New Approach to Reduce the Average Search Time in Binary Search*. Communication, Computer Technologies and Optimization Techniques (ICEECCOT), pp. 185-190, DOI:10.1109/ICEECCOT46775.2019.9114690.