Investigating the Selection Problem by Implementing Randomized Select, Bubble Sort, & Gnome Sort

Abstract

This miniproject took place to investigate the selection problem and different types of sorting algorithms to sort a generated dataset with distinct elements. Three sorting algorithms were implemented where running times and memory usage was measured to get a better idea of each algorithm's performance. A dataset of 100.000 unique integers number were generated. The sorting algorithms were each given this dataset, but more precisely, they were implemented to take in 10000, 50000, and 85000 integers to see the differences of running times and memory usage depending on the size of the dataset when sorting. The performance from each algorithm was expected based on their asymptotic running times, as Randomized Select outperformed Bubble Sort & Gnome Sort, while those two performed somewhat similar.

Introduction

The selection problem concerns itself with lists or arrays where it will find the *ith* smallest element in a set of *n* elements. The *ith* smallest element in *n* elements is known as the *ith order statistic,* where the minimum element is the *first order statistic* and maximum is the *nth order statistic*. With this in mind, we can describe the selection problem as:

- Input: Given a set, A, of n (distinct) numbers and an integer, i, with 1 <= i <= n.
- Output: The element, x, in A that is larger than exactly i – 1 other elements of A.

Thus, the selection problem can be solved by sorting a dataset in ascending order and then choosing the ith element from the sorted dataset list as (Cormen et al., 2009, pp. 213-217). There are multiple sorting algorithms which can be implemented to solve said issue. For this miniproject, I decided to develop three sorting algorithms to compare their running times and memory usage against each other. The three chosen algorithms are Randomized Selection, Bubble Sort, And Gnome sort. This miniproject will present pseudocode of the developed sorting algorithms followed by results on the running times and memory usage of each algorithm for three different sizes of the dataset. The results will then be reported and discussed in detail followed by a conclusion which sums up the findings of this miniproject.

Pseudocode

This section will present pseudocode of each of the three implemented sorting algorithms. After each set of pseudocode there will be an explanation of what it does.

The pseudocode is developed out of the original implementation which can be found here:

https://github.com/CeeGeeArt/ADSSE

Pseudocode – Randomized Select and Partition

This subsection will present pseudocode of the randomized select algorithm. It also includes partition and randomized partition pseudocode which is used for the randomized select.

```
1   function Partition(PArray[], Pstart, Pend)
2       x := PArray[Pend]
3       i := Pstart - 1
4       for j := Pstart to Pend - 1
5           if PArray[j] <= x do
6               i = i + 1
7               temp := PArray[i]
8               PArray[i] = PArray[j]
9               PArray[j] = temp
10          end if
11      end for
12      temp := PArray[i + 1]
13      PArray[i + 1] = PArray[Pend]
14      PArray[Pend] = temp
15      return i + 1
16  end function
```

Figure 1 - Pseudocode of Partition

Figure 1 presents the first of the three parts of the randomized select sorting algorithm implemented in this miniproject. At line (1) the function Partition is initialized with the array PArray[] and integers Pstart and Pend. Line (2) and (3) defines two new sets of integers, x and i. Then, from line (4) to (11) a for loop is computed ranging from Pstart to Pend – 1. While this for loop runs, an if-statement from line (5) to (10) is running which states that if PArray[j] is smaller or equal to value x, then we swap the contents of PArray[i] and PArray[j] by initially storing PArray[i] in a temporary variable (line (7) to (9)). Line (10) and (11) ends the if-statement and for loop. From line (12) to (14) we do the same as in line (7) to (9) where we swap the contents of PArray[i] and PArray[j], except that we do it for PArray[Pend] instead and also increase the value of PArray[i] by 1. The partition algorithm is implemented based on the partition procedure of how the quicksort algorithm also would solve this problem (Cormen et al., 2009, pp. 171-172).

```
1   function RandomizedPartition(RPArray[], RPstart, RPend)
2       i := rand()
3       temp = RPArray[RPend]
4       RPArray[RPend] = RPArray[i]
5       RPArray[i] = temp
6       return Partition(RPArray, RPstart, RPend)
7   end function
```

Figure 2 - Pseudocode of Randomized Partition

Figure 2 presents the second part of the Randomized Select sorting algorithm implemented in this miniproject. At line (1) we initialize RandomizePartition with array RPArray[] and integers RPstart and RPend as input arguments. Line (2) generates a set of random integers which through line (3) to (5) gets tored into the RPArray[i]. At line (6) we state what the function should return which in this case is the RPArray, RPstart and RPend values obtained in the function. The RandomizedPartition pseudocode is, like the Partition, developed based on the same procedure of development for quicksort (Cormen et al., 2009, pp. 179).

```
1  function RandomizedSelect(Array[], start, end, target)
2      if start == end do
3          return Array[start]
4      end if
5      pivot := RandomizedPartition(Array, start, end)
6      k := pivot - start + 1
7      if target == k do
8          return Array[pivot]
9      end if
10     else if target < k do
11         return RandomizedSelect(Array, start, pivot - 1, target)
12     end else if
13     else
14         return RandomizedSelect(Array, pivot +1, end, target -k)
15     end else
16 end function
```

Figure 3 - Pseudocode of Randomized Select

Figure 3 presents the pseudocode for the randomized select algorithm. At line 1 we define the function with one array and three integers as input arguments. From line (2) to (4) we check if integers ‘start’ and ‘end’ have the same value, and if they do, we return the Array[start]. From line (5) to (9) we define a pivot integer and from that define integer k which is used in line (7) to check if it’s equal to our target integer and if that is the case, we return the Array[pivot]. From line (10) to (12) we check the k value again, but this is if it is higher than our target integer and if that is the case, we instead return a set of values, including our pivot-1. Finally, line (13) to (15) is an else statement which is used to return the RandomizedSelect if none of the above statements are met. If that’s the case, we return the same values as before, but with pivot integer increased and a decreased target integer. The target integer is decreased by our k value. (Cormen et al., 2009, pp. 213-217).

Pseudocode – Bubble Sort

```
1   function bubbleSort(A[],n)
2       swapped := true
3       while swapped do
4           swapped := false
5           for i := 1 to n - 1 do
6               if A[i - 1] > A[i] then
7                   swap(A[i-1], A[i])
8                   swapped = true
9               end if
10          end for
11          n := n - 1
12      until not swapped
13  end function
```

Figure 4 - Pseudocode of Bubble Sort

Figure 4 present the pseudocode of the bubble sort algorithm used in this miniproject. At line (1) the function named bubbleSort is initialised along with the array A[] and integer n. At line (2) we initialize a Boolean called swapped and set it as true. Then from line (3) to (12) a while loop is computed which requires the Boolean swapped to be true to run. Then the first thing that happens at line (4) is the Boolean swapped being set to false. After this, at line (5) to (10) a for loop is computed where the initialized value, i = 1 has to be smaller than the value of n to run. For each iteration we increase the value of i by 1 until we reach the point where i is larger than n. Inside this for loop there an if-statement from line (6) to (9). Here we check if the content of array A[i -1] is larger than the content of array A. If that is the case, then line (7) and (8) runs. Here, we swap the content of array A[i-1] and array[i] followed by setting the Boolean swapped to be true. At line (9) and (10) we end the if and for statement. At line (11) we decrease the value of n by 1 each time the code reach this state. This is to optimize the running time of the bubble sort so the loop avoids looking at the n-th largest item when running for the n-th time. After this, line (12) ends the while loop while line (13) ends the function bubbleSort (Wikipedia, 2021, Bubble Sort, https://en.wikipedia.org/wiki/Bubble_sort).

Pseudocode – Gnome Sort

```
1  function gnomeSort(a[],dataSize)
2      pos := 0
3      while pos < dataSize do
4          if pos := 0 or a[pos] >= a[pos -1] do
5              pos := pos + 1
6          end if
7          else
8              swap(a[pos], a[pos -1])
9              pos := pos -1
10         end else
11     end while
12 end function
```

Figure 5 - Pseudocode of Gnome Sort

Figure 5 presents the pseudocode of the gnome sort algorithm used in this miniproject. At line (1) the function named gnomeSort is initialized along with a couple of values: the array a[] and the integer dataSize. At line (2) the integer pos is initialized and set to 0. From line (3) to (11) a while loop is computed. Starting from line (3), we check if the value of pos is lower than the value of dataSize, and while that is the case, the code within our while loop runs. At line (4) we have an if-statement. It checks if the value pos is equal to 0 or if the value a[pos] is higher or equal to a[pos -1]. If either statements are true, line (5) will run which increases the value of pos by 1. At line (6) we end this statement followed by and else statement at line (7). This code (line (8) and (9) will run if the if-statement at line (4) does not. Here, in line (8) we swap the contents of a[pos] and a[pos-1] followed by decreasing the value of pos by 1 in line (9). Line (10), (11), and (12) ends the else-statement, while loop, and the gnomeSort function, respectively (Wikipedia, 2021, https://en.wikipedia.org/wiki/Gnome_sort).

Report of Running Times & Memory Usage

Randomized Select

Randomized Select is a divide-and-conquer algorithm which works on one side of the partition to find out if the *ith* element is larger or smaller than the given pivot. The partition running time is $\theta(n)$ due to it having to sort element either larger or smaller than the given pivot. Randomized Selects worst case running time is due to its randomness, i.e., should it in each iteration choose the smallest or the largest element in the dataset as the pivot, the running time would end up being $\theta(n^2)$ since the partition would be reduced by one element per iteration. The space complexity for Randomized Selection is $O(\log n)$, and since it also takes in the input array, the total space complexity of Randomized Select is $O(n \log n)$. Randomized Select is an in-place sorting algorithm, i.e. in order to control start, ends, and pivot for a subarray, Randomized Select will only require some number of constant time integer initialization. (Cormen et al., 2009, pp. 216-217).

Should the case be where the elements are non-distinct, the probability of choosing some random value as the pivot will be larger due to the chance of picking a number occurring multiple times being larger. In our

case we wish to find the *ith* smallest element, so a dataset with non-distinct elements should not be used for this scenario. However, if we did not have another dataset besides this example, a solution to avoid having a skewed pivot could be implemented by simply moving duplicated elements out of the picking order when selecting the pivot.

Bubble Sort

Bubble Sort is a simple sorting algorithm but at the same time not one of the more practical ones, especially in larger datasets. This is due to the amount of time it takes the algorithm to sort large datasets where it will continuously have to compare and swap elements with each other until the dataset is finally sorted. Its average running time is $\theta(n^2)$ while its worst running time is $O(n^2)$ if the dataset is sorted in the exact reverse of the desired order. As earlier mentioned, this Bubble Sort is an optimized version of Bubble Sort. Should the case be that the data is already sorted, the running time would thus be $\Omega(n)$. Like Randomized Select, Bubble Sort sorts in place and therefore has a space complexity of O(1). Should the case be that our dataset already is sorted, the total space complexity would be O(n).

Unlike Randomized Select, Bubble Sort is computed in a way that non-distinct elements will not influence the running time due to how it will compare the elements with each other and hence won't swap contents next to each other if they are equal.

Gnome sort

Just like Bubble Sort, Gnome Sorts average running time is $\theta(n^2)$ as it will check if the current element is out of order and then find its correct location in the dataset. Meanwhile, like the case where Bubble Sort works on sorted data, Gnome Sort has a running time of $\Omega(n)$. Its worst running time would also like Bubble Sort be $O(n^2)$ in the same case where the dataset is completely the opposite of how it should be sorted. Its worst space complexity is also O(1) as it sorts in place (GeeksForGeeks, 2021, https://www.geeksforgeeks.org/gnome-sort-a-stupid-one/)

Analysis of running time & memory usage

This section will present an analysis of the running times of the different algorithms. Each algorithm has been given the same, unordered dataset. Multiple running time tests were made, but in the end I decided to stick to three lengths of the dataset for comparisons. The generated dataset has been used in the algorithms with 10000, 50000, and 85000 unique integers.

To develop an analysis of the running time and memory usage, I took use of the Chrono library in Visual Studio. I also used a Memory Usage diagnostic tool from Visual Studio to measure the memory usage in different stages of the algorithm. Each will be presented here and discussed in 'Report of Running Times & Memory Usage' (GeeksForGeeks, 2017, https://www.geeksforgeeks.org/chrono-in-c/#:~:text=Chrono), (Microsoft, 2018, https://docs.microsoft.com/en-us/visualstudio/profiling/memory-usage).

Input Target: 500	**10000**	**50000**	**85000**
Randomized Selection Sort	1,2978 ms	0,6837 ms	0,777 ms
Bubble Sort	1360,97 ms	34047,2 ms	97683,5 ms
Gnome Sort	1257,76 ms	31417,8 ms	90739,9 ms

Table 1 - Overview of Running Times

Table 1 presents and overview of the running times obtained after running each specific set of data points for each sorting with an input target of 500. **Note**: A dataset of 100,000 unique integers was desired as the biggest dataset, but an internal issue with Visual Studio resulted in the algorithm not running, and after trying several other maximums for the size of the data, 85000 was chosen. The figure makes it easy to see that Randomzied Seleciton Sort performs much better than Bubble and Gnome Sort which both have somewhat similar results. The results will be discussed further in the section below. The dataset was generated using OnlineNumberTools (OnlineNumberTools, https://onlinenumbertools.com/generate-integers).

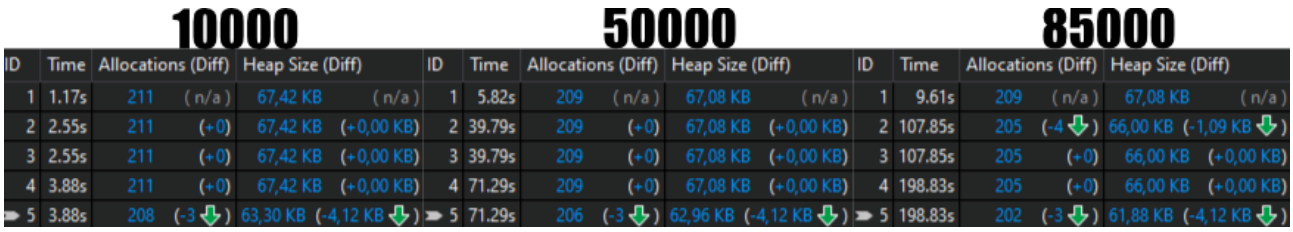

Figure 6 is a collection of screenshot images of memory usage information obtained from using the aforementioned Memory Usage diagnostic tool. Each number (1 to 5) presents a specific line of code which I chose as breakpoints to find out how long it took the algorithm to reach this specific line as well as how much memory was used. The numbers in the top, i.e. 10000, 50000, and 85000 represent the data size for each of the three iteration that memory usage information was obtained. The IDs specific lines of code are:

- ID 1: Line 114. This is before running any of the three sorting algorithms.
- ID 2: Line 122. This is after running the bubble sort algorithm.
- ID 3: Line 130. This is after running the Randomized Select algorithm.
- ID 4: Line 138. This is after running the Gnome Sort algorithm.
- ID 5: Line 140. Final line of code.

Conclusion

This miniproject found different running times and memory usage for the three described sorting algorithms and compared the results to each other. Based on research the results make sense, but could be improved by running the sorting algorithms on a system not being bottlenecked by RAM as previously mentioned. As expressed earlier, the Randomized Select prove to be the superior sorting algorithm of the three. This came as no surprise as the running time of Bubble and Gnome sort are similar in their worst, average and bests cases, so their overall similarities in this miniproject was to be expected. In the future, developing sorting algorithms with different running times would be interesting to see how their performance compare to those implemented in the miniproject.

Bibliography

[1] Cormen, T.H., Leirserson, C, Rivest, R., & Stein, C (2009). Introduction to Algorithms, 3rd Edition

[2] https://en.wikipedia.org/wiki/Bubble_sort – (2021) – “Bubble Sort” – retrieved from Wikipedia

[3] https://en.wikipedia.org/wiki/Gnome_sort - (2021) - “Gnome Sort” – retrieved from Wikipedia

[4] https://www.geeksforgeeks.org/gnome-sort-a-stupid-one/ - (2021) – “Gnome Sort” – by GeeksForGeeks

[5] https://www.geeksforgeeks.org/rand-and-srand-in-ccpp/ - (2020) – “rand() and srand() in C/C++” – by GeeksForGeeks

[6] https://www.geeksforgeeks.org/chrono-in-c/#:~:text=Chrono – (2017) – “Chrono in C++” – by GeeksForGeeks

[7] https://docs.microsoft.com/en-us/visualstudio/profiling/memory-usage – (2018) – “Measure memory usage in Visual Studio” – by Microsoft

[8] https://onlinenumbertools.com/generate-integers – (unknown year) – by OnlineNumberTools