Selection Algorithms

A Comparison between Randomized Selection, Selection by Sorting, and Partial Selection Sort

Abstract:

Different algorithms can be used to tackle the Selection Problem, a problem which aims to find the *i*th-smallest element in an array of *n* elements. This report introduces, analyzes and compares three different algorithms which tackle the Selection Problem. These three algorithms are Bubble Sort, Randomized Selection, and Partial Selection Sort, which all have different expected running times and potential use cases, whether you want a sorted list, a partially sorted list or not. All algorithms performed as expected from their asymptotic running times, meaning that Randomized Select outperformed both Partial Selection Sort and Bubble Sort in the Selection Problem. In cases where a sorted list is needed, algorithms such as Mergesort or Quicksort would be better performing candidates than Bubble Sort. For cases where only a partial sorting of the array is required, Partial Selection Sort can be effective, as long as the target element *k* does not get too large compared to the size of the entire data set.

Introduction - The Selection Problem:

A Selection Algorithm is an algorithm which, over a list or an array, is concerned with finding the *i*th smallest element. Given a set of *n* elements, the *i*th smallest element in the set, is known as the *ith order statistic*. Thus, the minimum element of the set is the *first order statistic*, and the maximum is the *nth order statistic*.
The ***Selection Problem***, can therefore be described as:
Input: Given a set A of n (distinct) numbers and an integer i, $1 \leq i \leq n$.
Output: The element $x \in A$ that is larger than exactly $i - 1$ other elements of A.
As can be seen, the selection problem can be solved as a simple issue of sorting the set in ascending order, and then choosing the *i*th element from the index of the sorted array or list (Cormen et al., 2009, pp. 213-227). I have therefore decided to implement and explain the following solutions: *Bubble Sort* for a *selection by sorting* approach, *Randomized Selection* as discussed in class, and *Partial Selection Sort*, as an alternative solution when some sorting is preferred.

The code discussed in this report has been uploaded to a GitHub repository on the following link: https://github.com/CozyKuma/SelectionProblem_ADSSE

Pseudo-code & Explanations

Selection by Sorting - Bubble Sort:

Using sorting, a selection problem can be reduced to that of choosing a sorting algorithm, and then accessing the index of the *i*th element that you are interested in. In the case of arrays, this selection operation can be done in O(1), whereas for linked lists, it would be O(n), due to the lack of direct access.
For my implementation of sorting, I decided on implementing Bubble Sort. This was implemented in C++, and is shown in the pseudo-code below:

```
1  BUBBLESORT(A,n)
2        for i = 0 to n - 1
3              for j = 0 to n - i - 1
4                    if A[j] > A[j + 1]
5                          switch A[j] with A[j+1]
```

In line 1, the BUBBLESORT function is defined together with its input parameters *A* (the array to be sorted) and *n* (the size of the array).
In lines 2-3, two for-loops will walk the array based on the array-size minus 1, to compensate for zero-indexing. For each index in the first for-loop, the second for-loop will go through the whole array minus the index of the first loop and one, to account for the way that Bubble Sort sorts from the last index to the first over iterations. This is also where the sorting algorithm got its name from, as it “bubbles” smaller or larger elements to the end of the list or array.
In lines 4-5, an if-statement checks whether the current index is larger than the next index, and if it evaluates true, it will swap the values and continue the inner-loop.
If you wanted to sort the array or list in descending order, you would swap the ‘Larger Than (>)’ for a ‘Smaller Than (<)’ in the if-statement.
This Bubble Sort algorithm can be improved slightly by making a conditional check if a swap has occurred during the inner-loop before continuing the outer-loop. If no swap happened during the inner-loop, then the rest of the array must already be sorted. This Bubble Sort algorithm, I have named “Optimized Bubble Sort”, and can be found as the BubbleSortOptimized-function in my repository (GeeksForGeeks, 2020, Bubble Sort, https://www.geeksforgeeks.org/).

Randomized Select:

Randomized Select is, as also exemplified in class, a divide-and-conquer algorithm that is modelled after the *Quicksort* algorithm. Randomized Select is separated into three different functions,

RANDOMIZED-SELECT, RANDOMIZED-PARTITION and, PARTITION. The three functions were implemented in C++, and are shown in the pseudo-code below:

```
1  RANDOMIZED-SELECT(A, start, end, target)
2         if start == end
3                return A[start]
4         pivot = RANDOMIZED-PARTITION(A, start, end)
5         k = pivot - start + 1
6         if target == k
7                return A[pivot]
8         elseif target < k
9                return RANDOMIZED-SELECT(A, start, pivot - 1, target)
10        else return RANDOMIZED-SELECT(A, pivot + 1, end, target - k)
11
12 RANDOMIZED-PARTITION(A, start, end)
13        i = RANDOM(start, end)
14        switch A[end] with A[i]
15        return PARTITION(A, start, end)
16
17 PARTITION(A, start, end)
18        x = A[r]
19        i = start - 1
20        for j = start to end
21               if A[j] ≤ x
22                      i = i + 1
23                      switch A[i] with A[j]
24        switch A[i + 1] with A[end]
25        return i + 1
```

In line 1, the RANDOMIZED-SELECT function is defined with its parameters, *A*, the array, *start* and *end*, which controls the range of the subarray that the algorithm goes through, and *target*, which determines the *i*th smallest element for the algorithm to return.
Line 2-3 checks for the case that the array (or subarray) is only 1 element, in which it simply returns that one element.
Line 4 uses the RANDOMIZED-PARTITION function to generate a random pivot, which the subarray *A* will be partitioned around. The RANDOMIZED-PARTITION function has the array *A*, the *start*, and the *end* parameters passed as arguments.
Line 5 computes the number *k*, which is the number of elements on the low side of the partition, subarray *A[start . . pivot]*, plus one to also account for the pivot.

Line 6-7 checks whether the random pivot is the *i*th smallest element by random and returns it if true. Lastly, lines 8-10 determines whether the target *i*th smallest element is in the 'low' or 'high' side of the partition, and the RANDOMIZED-SELECT function is then called recursively on the determined subarray, given the known sizes of the two sides of the partition.

In line 12, the RANDOMIZED-PARTITION function is defined, which was the function called earlier in line 4. This function takes the parameters: *A,* the array, *start* and *end*, which determines the range of the subarray as before.
Line 13 uses a random function to pick a random number between the *start* and *end* ranges. We assume true randomness in this context, however it must be mentioned that for more important algorithms, a better random number generator than the ***rand()***-function should be considered (Stephan T. Lavavej, 2013, https://channel9.msdn.com/).
Line 14 then switches the last element of the subarray with the random pivot index, which is then subsequently used as an argument in line 15 for the PARTITION function.

In line 17, the PARTITION function is defined, which takes the parameters *A*, the array, *start* and *end*, which again are the ranges of the subarray. However, the *end* element has been swapped with the random index in the RANDOMIZED-PARTITION function (line 13), thus distinguishing itself from the standard QuickSort approach, which always uses the last element as the pivot value.
Lines 18-19 sets *x* as the value that the array is partitioned around, and *i* which keeps track of how many elements there are in the partition of smaller values.
Lines 20-23 loops the value *j* from *start* to *end* and checks if *A[j]* is less than or equal to *x*. If this evaluates to true, *i* is incremented and *A[i]* is switched with *A[j]*. In the first iteration of the loop, if the first value is smaller than the pivot, it is swapped with itself, and the loop continues.
In lines 24-25, as the loop finishes, the pivot value is switched from the *end* index to the *i + 1* index, thus ending in the middle between the two partitions, with lower values on the left side, and higher values on the right side. Lastly, the index of the pivot is returned, which in this case first returns to the RANDOMIZED-PARTITION function, which subsequently returns to the RANDOMIZED-SELECT function in line 4 (Cormen et al., 2009, pp. 170-179, 213-217).

Partial Selection Sort:

My last example is Partial Selection Sort, which can be useful when the *i*th smallest elements are all of practical use. One example being that you want the 10 smallest elements, but the rest of the array is unimportant to you.
The Partial Selection Sort function was implemented in C++, and isshown in the pseudo-code below:

```
1  PARTIAL-SELECTION-SORT(A, n, target)
2         for i = 0 to target
```

```
3               minIndex = i
4               minValue = A[i]
5               for j = i + 1 to n
6                     if A[j] < minValue
7                           minIndex = j
8                           minValue = A[j]
9               switch A[i] with A[minIndex]
10        return A[target-1]
```

In line 1, the PARTIAL-SELECTION-SORT is defined with the parameters *A*, the array, *n*, the size of the array, and *target*, the *i*th element that we wish to find, and sort the array in ascending order up to.
Lines 2-4 creates an outer-loop that will increment from 0 to the *target*, thus for each iteration find the *i*th smallest element. At the start of each iteration, the *minIndex* is set to the current loop counter variable *i*, and the *minValue* is set to that index' value.
Line 5 then implements the inner-loop, using the loop counter *j*, that for each iteration of the outer-loop, will walk the whole length of the array *A*, given the size of the array *n*.
Lines 6-8 then checks each value encountered in the inner-loop and evaluates it against the currently saved *minValue* for this iteration of the outer-loop. Subsequently, if the if-statement evaluates to true, a new *minIndex* and *minValue* has been found, and will be saved as such.
Line 9 will switch the current loop counter index *i* from the outer-loop with the found minimum value, using the saved index in *minIndex*.
Lastly, the return-statement returns the *target* value minus one, to account for zero-indexing, thus returning the *i*th smallest value, given the value of *target* (Wikipedia, https://en.wikipedia.org/wiki/Selection_algorithm#Partial_selection_sort).

Analysis:

Selection by Sorting - Bubble Sort:

Running time:
Bubble Sort is, compared to many other sorting algorithms, not a very practical sorting algorithm in respect to time complexity. It has an average running time of Θ(n^2), compared to Quicksort or Mergesort that have average running times of Θ(n log(n)). Due to how Bubble Sorts operates, it does poorly in very large datasets, as it will have to move larger objects much further through comparisons and swaps, before getting to the end. The worst-case scenario for Bubble Sort, which is O(n^2), is a list that is already sorted, but in reverse order. Best case would be a list that is already sorted, which for the aforementioned Optimized Bubble Sort would be Ω(n), as it would run through once without

any swaps and determine that it is already sorted. Lastly, to access the *i*th element in the resulting sorted list, only a constant time operation is necessary.

Space:

Since Bubble Sort sorts in place, without creating any auxiliary arrays, it stays very efficient in respect to its space complexity. Thus the auxiliary space complexity ends at constant time O(1). In total, we do have to also consider the array being sorted, which will be of *n* size, meaning that in total, we have a total space complexity of O(n).

Non-Distinct Elements:

In respect to non-distinct elements, Bubble Sort is not affected by this, and easily continues on past identical values through its comparisons. However, if you wish to find the *i*th smallest element, not counting identical values, you would have to walk the sorted array, incrementing the counter only when the next value is **larger and not equal to the current index**, until the counter is equal to the target *i*th element.

Randomized Select:

Running time:

Modelled after the Quicksort algorithm, Randomized Select is a divide-and-conquer algorithm that only works on one side of a partition, by determining whether the *i*th element-index is in the larger or smaller than the pivot, thus having an expected running time of Θ(n). The Partition-part in itself takes Θ(n) time, as it has to sort values to be either smaller or larger than the randomly selected pivot value. The worst-case scenario is in Randomized Select not a due to a specific input, but due to the randomness utilized, meaning that if it randomly chooses the smallest or largest element as a pivot at each iteration, it would have a running time of Θ(n^2), as the partition size would only be reduced by one at each iteration, while searching for a value in the other extreme (Cormen et al., 2009, pp. 216-217).

Space:

Randomized Select, like Bubble Sort, sorts in place and does not require more arrays, as it uses indexes instead of multiple initialized subarrays for its partitioning. This means that it only requires a number of constant time integer initialization to control pivots and the starts and ends for the subarrays. Due to its random nature, it is expected to initialize some values more than once. Thus, like the quicksort algorithm, the space complexity is on average O(log n), meaning that in total, counting the input array, has a total space complexity of O(n log n).

Non-Distinct Elements:

In a set with distinct values, each value has a 1/n probability of being the pivot (assuming true randomness), however, if values are not distinct, this probability will be skewed towards choosing one value as a pivot, more than others due to frequency, thus having a chance of repeating pivots with the same value. Furthermore, in the case that you want the *i*th smallest element of distinct values, a modification would have to be considered. One modification that can be considered, is a conversion

of the original array to a hash table or some other type of mapping structure, that would move duplicates to a chained/connected layer, such that the Randomized Select can be run on the generated array, without considering the chained duplicates. This however increases the required space, as auxiliary structures would be necessary to chain.

Partial Selection Sort:

Running time:

Partial Selection Sort has a rather simple set of operations, in which it loops the whole array up to *k* times, where *k* is the target *i*th element. Thus the expected average running time of Partial Selection Sort is Θ(kn).

Space:

Like Bubble Sort and Randomized Selection Sort, it has been implemented as an in-place sorting algorithm. With only little need for auxiliary integers to keep track of minimum indices and values, it has a space complexity of O(1). This, including the size of the input array, means that it has a total space complexity of O(n).

Non-Distinct Elements:

In its current implementation, Partial Selection Sort handles duplicates fine, however, like Bubble Sort and Randomized Selection Sort, if you want to discard duplicates from the elements that are considered to be smaller than the target *i*th element, modifications are necessary. One example of this could be to convert the outer-loop to a while loop, which will only break once a counter reaches the target value. This counter would then be incremented only when the last index is smaller than the current. An example of this implementation can be seen in the function PartialSelectionSortDupAware in the included repository.

Performance Metrics:

Running time:

The following section will apply data of different sizes to the algorithms to evaluate their performance against their expected performance from the analysis.

Array Size:	**100**	**1.000**	**10.000**
	0.4 kB	4 kB	40 kB

The following running times are based on an average calculated over 5 executions of the code with no changes in between. The target *i*th smallest value has been set to 50.

Target: 50	**Time at 100**	**Time at 1.000**	**Time at 10.000**
Bubble Sort Θ(n^2)	0.0213 ms	1.7857 ms	206.5960 ms
Randomized Selection Sort Θ(n)	0.0054 ms	0.0165 ms	0.1142 ms
Partial Selection Sort Θ(kn)	0.0130 ms	0.1043 ms	0.9748 ms

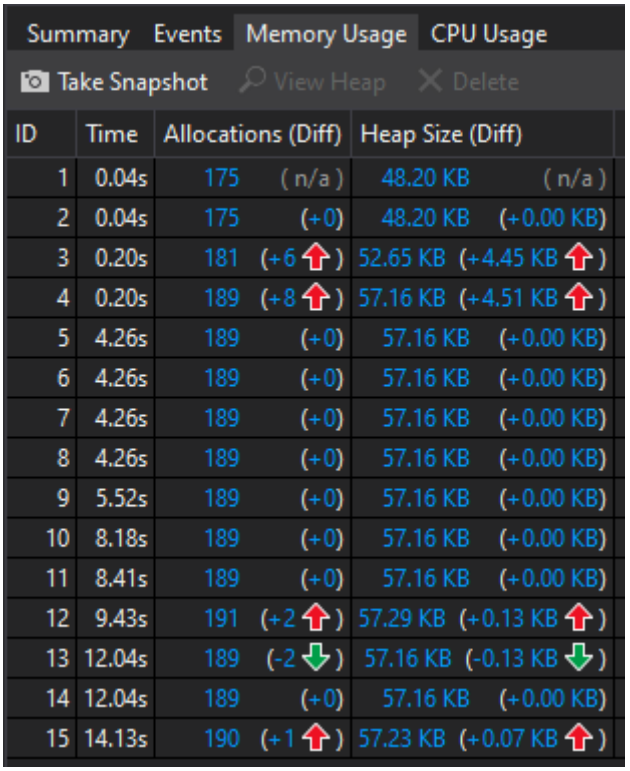
To account for the change in *k* in Partial Selection Sort, the following changes in *i*th smallest element (target) have also been tested with the Partial Selection Sort algorithm with an average over 5 executions. For this, the array size *n* was set to a static 1000.

Array Size / Target:	**1.000 / 100**	**1.000 / 250**	**1.000 / 500**
Partial Selection Sort	0.1908 ms	0.4435 ms	0.7866 ms

From the measured running times, compared with the expected running times in the analysis, we can see that they perform as expected. Given an increase by x 10 at each step, Bubble Sort increases by an approximate x 10^2, fitting the expected Θ(n^2). Randomized Selection Sort grows with an approximate x 10, fitting its expected running time of Θ(n). And Partial Selection Sort grows with a x 10, with a static *k*, meaning that this also fits the expected running time of Θ(kn).
Furthermore, we can see from the extra analysis of Partial Selection Sort, that the increase in the target *k* also has the expected effect on the data. An approximate doubling of the target value has a likewise approximate doubling of the running time.

Space:

To analyze the memory usage of my implementation, I have used Visual Studio 2019's Diagnostic Tool to profile the memory usage (Microsoft, 2018, https://docs.microsoft.com/). The following screenshot is taken from Visual Studio 2019 directly, with an array size of 10000:



ID	Time	Allocations (Diff)	Heap Size (Diff)
1	0.04s	175 (n/a)	48.20 KB (n/a)
2	0.04s	175 (+0)	48.20 KB (+0.00 KB)
3	0.20s	181 (+6 ↑)	52.65 KB (+4.45 KB ↑)
4	0.20s	189 (+8 ↑)	57.16 KB (+4.51 KB ↑)
5	4.26s	189 (+0)	57.16 KB (+0.00 KB)
6	4.26s	189 (+0)	57.16 KB (+0.00 KB)
7	4.26s	189 (+0)	57.16 KB (+0.00 KB)
8	4.26s	189 (+0)	57.16 KB (+0.00 KB)
9	5.52s	189 (+0)	57.16 KB (+0.00 KB)
10	8.18s	189 (+0)	57.16 KB (+0.00 KB)
11	8.41s	189 (+0)	57.16 KB (+0.00 KB)
12	9.43s	191 (+2 ↑)	57.29 KB (+0.13 KB ↑)
13	12.04s	189 (-2 ↓)	57.16 KB (-0.13 KB ↓)
14	12.04s	189 (+0)	57.16 KB (+0.00 KB)
15	14.13s	190 (+1 ↑)	57.23 KB (+0.07 KB ↑)

The snapshots have been taken using breakpoints at multiple points in the code. The breakpoints are placed both before and after the function calls (IDs 6 - 14) , during points in the setup (IDs 1 - 5), and at the end of the main-function (ID 15). As can be seen in the screenshot above, only minor changes occur in the Heap Size, and mostly during setup of the data arrays. This means that the implemented algorithms do not have a measurable impact on the memory usage, excluding the size of the array, even when compared to the other array sizes tested (100 and 1000). Smaller memory usage changes could be hidden as smaller byte differences might be hidden due to the Diagnostic Tool showing the difference in kilobytes, but such small changes can be considered negligible on modern hardware.

Conclusion:

From my analysis and measured performance metrics I can conclude that the expectations hold true in my implementations of the same algorithms. In the context of the original problem of selecting the *i*th smallest element in a set of distinct values, I can conclude that the Randomized Selection Sort is the best performing algorithm, as it in all measured averages performed better than the other two implemented algorithms. Furthermore, Bubble Sort, which is a generally inefficient sorting algorithm, performs the worst of the three as expected from the analysis. Partial Selection Sort performs reasonably well, and has the added benefit of sorting all the elements smaller than the target, which in

some cases might be appropriate, as the resulting sorted subarray can be directly accessed through the indices less than *k*, i.e. the target *i*th smallest element.
All in all, this analysis shows how a computational problem can be approached in multiple different ways, with each their own advantages and disadvantages, and that the size of the input data has a significant effect on the running times of the individual algorithms. Thus a computational problem should always be approached in the context of how the resulting data is to be used.

Bibliography:

1. Cormen, T.H., Leiserson, C., Rivest, R., & Stein, C. (2009). Introduction to Algorithms, 3rd Edition.
2. https://www.geeksforgeeks.org/bubble-sort/ (2020), “Bubble Sort” by GeeksForGeeks.
3. https://channel9.msdn.com/Events/GoingNative/2013/rand-Considered-Harmful (2013), “rand() Considered Harmful” by Stephan T. Lavavej.
4. https://en.wikipedia.org/wiki/Selection_algorithm#Partial_selection_sort, “Selection Algorithm - Partial Selection Sort” retrieved from Wikipedia.
5. https://docs.microsoft.com/en-us/visualstudio/profiling/memory-usage?view=vs-2019 (2018), “Measure memory usage in Visual Studio” by Microsoft.