Golsa Emdadi

CMSC 451 6381

Design and Analysis of Computer Algorithms

Project 2

10/15/2017

Selection Sort

The data set size are saved in the array. We divide the given array into two parts: sorted part and unsorted part. In iterative and recursive selection sort, an array is sorted by repeatedly finding the minimum element (considering ascending order) from unsorted part of the array and putting it at sorted part in the array. In fact, we take the current element (first element at the beginning) and exchange it with the smallest element on the right-hand side of the current element until all elements of the array are checked. In iterativeSort() function, loops repeat the code (finding the minimum element with its index in the array, count it and swap it). In recursiveSort() function, the function calls itself again to repeat the code (finding the minimum element with its index in the array, count it and swap it). countIteration in iterativeSort() and countRecursion in recursiveSort() are used to count critical operations. Also, timeIteration and timeRecursion use to find execution time of two versions of the algorithm. Average Critical Operation Count, Standard Deviation of Count, Average Execution Time, and Standard Deviation of Time are used to compare iterative and recursive algorithms in the selection sort.

Critical operation of selection sort = Comparing of current element (first element of the unsorted part) of the array with other elements in unsorted part of the array. These comparisons are main and critical operations because they are related to size of the array. For example, if we increase size of the array(data set size), total numbers of elements, and total numbers of required compressions will increase.

Critical operation in iterative version:

countIteration = 0;

**long** startTime = System.*nanoTime*(); **//start time of iteration**

// One by one move through elements of the array

**for** (**int** i = 0; i < array.length - 1; i++){

//the current element in unsorted part as default is min

**int** min = array[i];

**int** minIndex = i;

// compare elements to find minimum

**for** (**int** j = i + 1; j < array.length; j++){

**if** (min > array[j]){

min = array[j];

minIndex = j;

}

countIteration++; **// counting critical operation**

}

//swap found minimum element with the default element

**if** (minIndex != i){

array[minIndex] = array[i];

array[i] = min;

}

}

**long** endTime = System.*nanoTime*(); **// end time of iteration**

timeIteration = endTime - startTime; **// actual time of iteration**

Critical operation in recursive version:

countRecursion = 0;

// call function recursiveSort

recursiveSort(array, 0, array.length - 1);

-------------------------------------------------------

**public** **void** recursiveSort(**int** [] array, **int** firstIndex, **int** lastIndex)**throws** Exception{

// compare elements, which is related to finding critical operation count

//(comparing loop)

**long** startTime = System.*nanoTime*(); **// start time of recursion**

if (firstIndex < lastIndex) {

//default min is current element in unsorted part of array

**int** min = array[firstIndex];

**int** minIndex = firstIndex;

// compare elements to find minimum

**for** (**int** i = firstIndex + 1; i <= lastIndex; i++){

**if** (array[i] < min) {

min = array[i];

minIndex = i;

}

countRecursion++; **// counting critical operation**

}

//swap found minimum element with the default element

array[minIndex] = array[firstIndex];

array[firstIndex] = min;

// continue sort operation recursively

recursiveSort(array, firstIndex + 1, lastIndex);

}

**long** endTime = System.*nanoTime*(); **// end time of recursion**

timeRecursion = endTime - startTime; // actual time of recursion

Both iterative and recursive versions have the same total numbers of comparisons in each data set (array length), Therefore, we expect both iterative and recursive versions to have the same Average Critical Operation Count (the output result of project 1 proves it). For example, when n=1000, average of Critical Operation Count in all 50 sets is 499500.0 for two versions.

Selection sort is an in-place sort, so it requires no space other than that required by the input data (the array size) and a few variables whose number does not depend on the size of the set being sorted (The Selection Sort, n.d.). Therefore, its asymptotic space complexity is O(n) in two versions (the additional space complexity used by the algorithm = O(1) ).However, there is an inherent overhead to using recursion, since they must store the stack as well, which uses more memory.

In iterative selection sort, selecting the lowest element requires scanning all n elements (this takes n − 1 comparisons) and then swapping it into the first position. Finding the next lowest element requires scanning the remaining n − 1 elements and so on, for (n − 1) + (n − 2) + ... + 2 + 1 = n (n - 1) / 2 ∈ O(n2) comparisons. Each of these scans requires one swap for n − 1 elements (the final element is already in place). We can also analyze the time complexity of selection sort by writing a recurrence system from the recursive form of the algorithm. If T(n) is the number of comparisons required to sort a set of n elements, the following recurrence equations is (Big-Oh, n.d.):

T(0) = T(1) = 0

T(n) =T(n -1) + n-1 for n>1

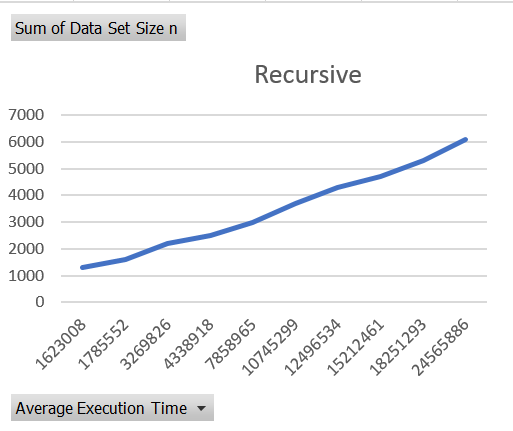
T(n)= (n-1)+(n-2)+…. +2+1= n(n-2)/2

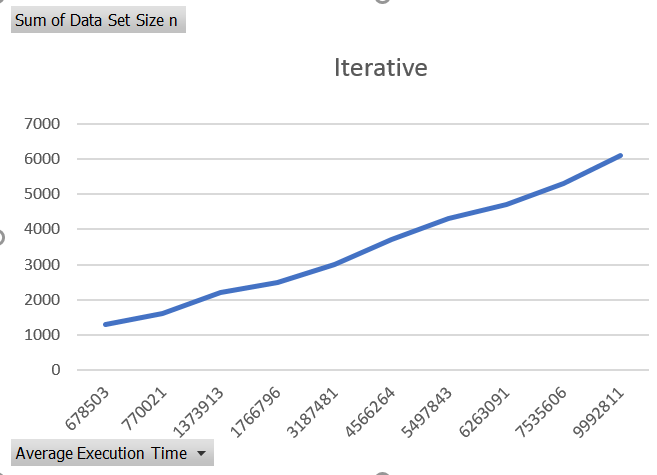
Therefore, Big-O for execution time= O(n2) in both iterative and recursive versions. Measured by comparisons and swap both together, the asymptotic time complexity of selection sort is O(n2) in two versions. O(n2) shows a quadratic function as the graph, and we expect Average Execution Time to increase when data set size of n increase in two versions. In each iteration, selection sort has to search through the entire unsorted part of the array looking for the minimum element, so algorithm’s best, worst and average running times are exactly equal O(n2).

Table 1 of project 1 result after modifying data set size of n

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Data Set Size n**   |  | | --- | |  | | **Iterative** | | | | **Recursive** | | | |
| **Average Critical Operation Count** | **Standard Deviation of Count** | **Average Execution Time** | **Standard Deviation of Time** | **Average Critical Operation Count** | **Standard Deviation of Count** | **Average Execution Time** | **Standard Deviation of Time** |
| 1300 | 844350 | 649.5 | 678503 | 2582.558691615556 | 844350 | 126652.5 | 1623008 | 282456.92047493864 |
| 1600 | 1279200 | 799.5 | 770021 | 686.1257680959902 | 1279200 | 155902.5 | 1785552 | 218370.01328399067 |
| 2200 | 2418900 | 1099.5 | 1373913 | 696.8970859433127 | 2418900 | 214402. 49 | 3269826 | 291460.1479178956 |
| 2500 | 3123750 | 1249.5 | 1766796 | 795.8331437477079 | 3123750 | 243652.5 | 4338918 | 329663.09825672384 |
| 3000 | 4498500 | 1499.5 | 3187481 | 1700.4212079712945 | 4498500 | 292402.49 | 7858965 | 529844.2054954683 |
| 3700 | 6843150 | 1849.5 | 4566264 | 4741.149724124376 | 6843150 | 360652.5 | 1.0745299E7 | 601880.49486978 |
| 4300 | 9242850 | 2149.5 | 5497843 | 3943.900682438962 | 9242850 | 419152.5 | 1.2496534E7 | 563398.6219274448 |
| 4700 | 1.104265E7 | 2349.5 | 6263091 | 1915.8224216656442 | 1.104265E7 | 458152.5 | 1.5212461E7 | 615134.6109284593 |
| 5300 | 1.404235E7 | 2649.5 | 7535606 | 1450.6733336335776 | 1.404235E7 | 516652.5 | 1.8251293E7 | 672307.7388388361 |
| 6100 | 1.860195E7 | 3049.5 | 9992811 | 1664.9048617560964 | 1.860195E7 | 594652.5 | 2.4565886E7 | 785668.6166704883 |

Table (the code recult)

Graphs of the results:



A comparison of Big-O (time complexity), actual time, and performance of iterative and recursive versions of the selection sort (Figure1).

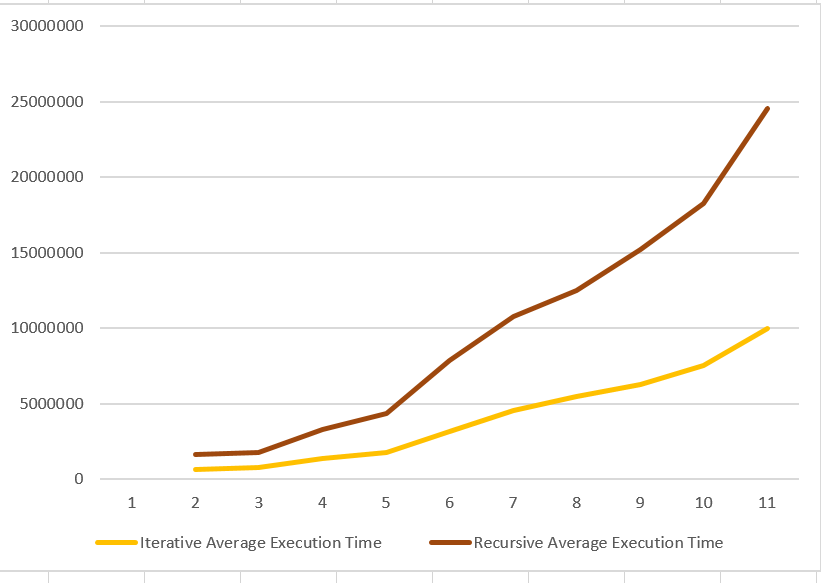


Figure 1:Time Complexity

Figure 1 shows that both iterative and recursive versions has the same time complexity of O(n2) because both versions have graphs with quadratic function behavior. Also, with increasing data set size, time complexity increases in two versions. However, slope of the recursive graph is more that iterative graph in all data set sizes which shows that elapsed time of recursive executions is more than iterative version. Also, Figure 1 shows when the data set size is small the iterative and recursive versions of selection sort behave nearly the same way in term of execution time. For large data set sizes, the recursive version consumes more time than the iterative version. System.nanoTime() is used to calculate start Time and end Time executions of two versions of iteration and recursion executions. The nanoTime() are placed at beginning and end of both recursiveSort and iterativeSort functions.

Therefore, actual execution time = endTime - startTime; in each version, which is used to find Average Execution Time(The actual time is sum of critical operation, swap operation and execution of whole part of each sort function). Recursion can be slower than iteration because in addition to processing the loop content, it must deal with the recursive call stack frame (overhead), which takes more time. Therefore, recursion performance can be less than iteration.

A comparison of the critical operation results of iterative and recursive results:

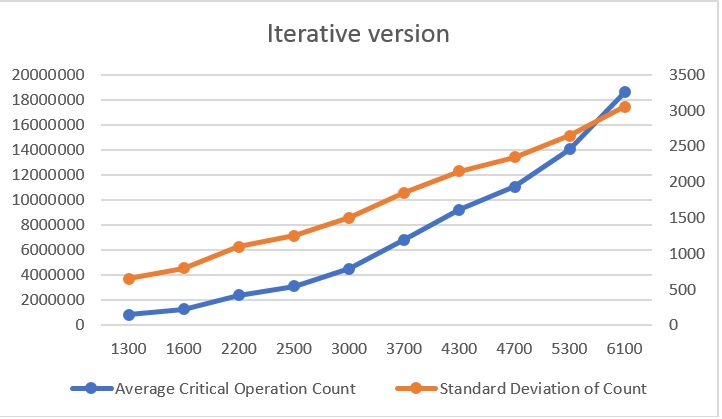
Critical operation of selection sort = Comparing of current element of the array with other elements in unsorted part of the array and finding minimum element. In both versions, countIteration++ and countRecursion++ are used to count critical operations. No matter how random numbers will change, the critical operation is dependent on the array length (data set size) in each set. Therefore, we expect average critical operations count of two versions to be the same in each data set. Also, we expect the average critical operation increase when data set size increase in both versions (see the Table 1). The critical operation impacts the entire operations, so impact time complexity.

Both average critical operation count and average execution time (actual time) increase by increasing data set size n in two versions.

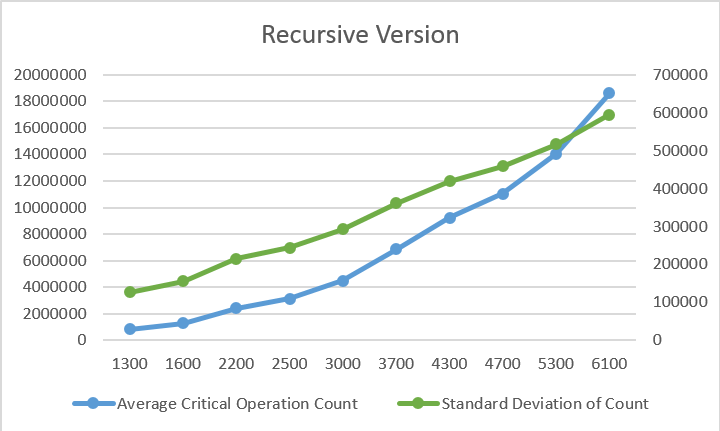
Average (or mean) and Standard Deviation In two versions:

Standard deviation is used to tell how measurements for each group of data set are spread out from the average (mean), or expected value. A low standard deviation means that most of the numbers are very close to the average. A high standard deviation means that the numbers are spread out (Standard Deviation, n.d.). Therefore, a normal distribution of data means that most of the examples in a set of data are close to the average, while relatively few examples tend to one extreme or the other.

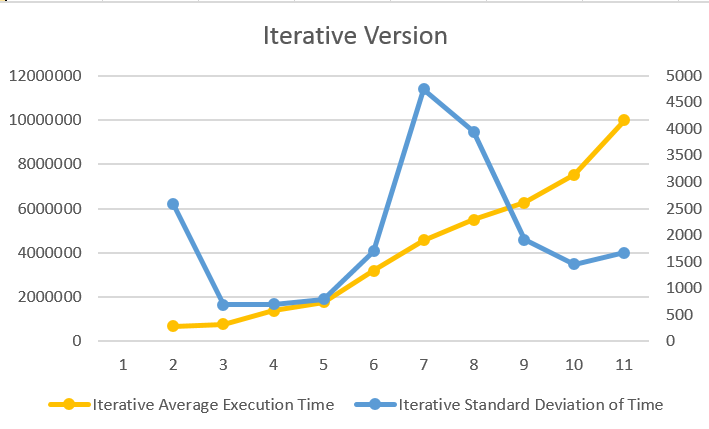
The standard deviation of count for the iterative version shows that with increasing data set size, the critical operation counts are closed to the average critical operation counts (decrease of deviation with increase data set size).



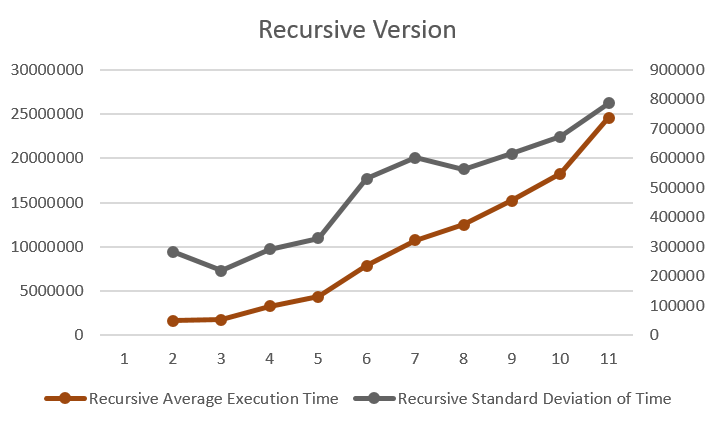
The standard deviation of count for the recursive version shows that with increasing data set size, the critical operation counts are closed to the average critical operation counts (decrease of deviation with increase data set size).



The standard deviation of time for the iterative version shows that during the 50 runs and increasing data set size, the execution times were very close to the average execution times in some data sets, and not close in some other data sets (various dispersion pattern).



The standard deviation of time for the recursive version shows that during the 50 runs and increasing data set size, the execution times were not close to the average execution times in some data sets.



Conclusion:

Selecting the lowest element requires scanning all n elements (this takes n − 1 comparisons) and then swapping it into the first position. Finding the next lowest element requires scanning the remaining n − 1 elements and so on, for (n − 1) + (n − 2) + ... + 2 + 1 = n(n − 1) / 2 ∈ Θ(n2) comparisons. Each of these scans requires one swap for n − 1 elements. Both iterative and recursive versions show the same graph pattern in execution time and critical operation count (increasing executing time and critical operation by increasing data set size). However, Average Execution Time and Average Critical Operation Count of recursive version is more than iterative version in all data set sizes. Therefore, recursive version has less performance that iterative version because for each step we make a recursive call to the function which occupies significant amount of stack memory with each step.

# References

*Big-Oh for Recursive Functions: Recurrence Relations*. (n.d.). Retrieved from users.cs.duke.edu: https://users.cs.duke.edu/~ola/ap/recurrence.html

*Standard Deviation and Variance*. (n.d.). Retrieved from Mathsisfun: http://www.mathsisfun.com/data/standard-deviation.html

*The Selection Sort*. (n.d.). Retrieved from interactivepython.org: http://interactivepython.org/runestone/static/pythonds/SortSearch/TheSelectionSort.html