The Selection Problem

Abstract

This miniproject will look into “The Selection Problem”, and what it consists of, and try to solve it using the “Cocktail Sort” algorithm and the “Randomized Select” algorithm. There will be pseudocode of both the algorithms and text explaining the pseudocode. This mini project will also analyse on the running time and the memory usage of the two algorithms both theoretically and practically.

Introduction

The Selection problem consists of “Given a set, A, of n(distinct) numbers and an integer, i, with 1 <=i<=n, compute the element, x, in A that is larger than exactly i-1 other elements in A.”, and this can be solved by sorting A and then it should always be true, but the Random Selection Algorithm can often do this faster, and often has a smaller complexity than other sorting algorithms, but in worst case, this algorithm might have to go through every single element in the given set, which ends up having a complexity, that is worse than some sorting algorithms.

Pseudocode - and explanation

This section is going to show the pseudocode of “Randomized Select“ and “Cocktail Sort”, and then explain these algorithms.

Randomized Select

The Randomized Select algorithm[1] is described as being modelled after the quicksort algorithm, where the array is partitioned recursively, and processes both sides of the partition, but the Randomized Select will only process recursively on one side of the partition. This means that the quicksort algorithm will have an expected running time of $\Omega(n \lg n)$, and the Randomized Select will have an expected running time of $\Omega(n)$, as long as the elements in the given dataset are distinct. The Randomized Select consists of three parts: A Partition which can generate a pivot in the array, which makes sure that we know that all values after this pivot is larger, and all the

values before are lower than the pivot point. The second part is the Randomized-Partition, which generates a random pivot point based on the Partition function, described before. And then the final part is the Randomized Selection itself, which is based on the Randomized Partition and should return the i^{th} smallest element of the array A, that was given in the beginning.

Partition

```
1   PARTITION(Array, Start, End)
2   x = Array[end]
3   i = Start - 1
4   for (j = Start to End - 1) {
5       if (Array[j]<=x){
6           i = i + 1
7           exchange Array[i] with Array[j]
8       }
9   exchange A[i+1] with A[End]
10  return i + 1
11  }
```

Line 1 shows the name of the function, and its inputs.

In line 2, x is assigned to be the end-value of the array, and is selected as a ***pivot*** element.

In Line 3, ***i*** is assigned to be the Start-value minus 1 of the array, which means it is out of the array.

Line 4 is a for-loop that loops through the array from the beginning until the End-value of the array is reached.

Line 5 checks if the current value is less than or equal to the ***pivot*** element. And if that is the case, line 6 will add 1 to ***i*** and

line 7 will exchange Array[i] with Array[j]. Line 8 will exchange A[i+1] with A[end], and finally line 9 will return ***i*** + 1.

Randomized Partition

```
1   RANDOMIZED-PARTITION(Array, Start, End)
2   i = RANDOM(Start, End)
3   exchange Array[End] with A[i]
4   return PARTITION(Array, Start, End)
```

Line 1 does once again show the name of the function and its inputs.

Line 2 assigns ***i*** to be a random value between the Start number and the End number.

Line 3 then exchanges the End-value of the Array with the random chosen number from line 2 in the Array.

Line 4 then calls the Partition function from before, which will now use the new randomly selected value as its ***pivot*** point.

Randomized-Select

```
1  RANDOMIZED-SELECT(Array, Start, End, i-th element)
2  if Start == End
3      return Array[Start]
4  q = RANDOMIZED-PARTITION(Array, Start, End)
5  k = q - End + 1
6  if i == k //The pivot value is the answer
7      return Array[q]
8  elseif i < k
9      return RANDOMIZED-SELECT(Array, Start, q-1, i-th element)
10 else return RANDOMIZED-SELECT(Array, q + 1, End, i-th element - 1)
```

Line 1 is the name of the function and its inputs.

Line 2 checks if the Start value is the same as the End value, and if that is the case, then the array must consist of a single element, which must be the i-th smallest element, and

Line 3 will then return that number.

Line 4 creates a new ***pivot*** point using the RANDIMIZED-PARTITION function, and this happens if Start is not the same as End in line 2

Line 5 assigns ***k*** to be the new pivot point created in line 4 minus the End value plus 1, which means that it calculates the number of elements that are below the pivot point, and then plus 1 to include the pivot itself.

Line 6 checks if *i* == *k* and if that is the case, then we have found i^{th} smallest element.

Line 7 will return Array[q], which happens if line 6 is true.

Line 8 will look into the case where ***i*** does not equal ***k*** but is smaller than ***k***, which means that it checks if the i^{th} element is on the smaller side of the pivot point.

If Line 8 is true, then line 9 will check the values that are lower than the pivot point recursively. But if Line 8 is not true, then

Line 10 will check the values that are greater than the pivot point recursively.

Cocktail Sort

The Cocktail Sort is a sorting algorithm[2] that is described as being a variation of the Bubble Sort algorithm. The Bubble sort algorithm goes through the array of data from left to right, and when it makes it to the end, it will start over again and move from left to right until it moves through the entire array without making a change. The Cocktail sort also moves from left to right, but instead of starting over, it will then move from right to left and swap elements if they are

```
1   CocktailSort(int[] Data,int ArraySize)
2   bool Swapped = true
3   int CocktailSortStart = 0
4   int CocktailSortEnd = ArraySize - 1
5
6   while(swapped == true){
7       swapped = false
8       for(int i = csStart; i < csEnd; i++){
9           if (a[i] > a[i+1]) {
10              swap(a[i], a[i+1]);
11              swapped = true;
12          }
13      }
14      if(!swapped){
15          break;
16      }
17
18      swapped = false;
19
20      --csEnd;
21
22      for(int i = csEnd - 1; i >= csStart; --i){
23          if (a[i] > a[i + 1]) {
24              swap(a[i], a[i+1]);
25              swapped = true;
26          }
27      }
28      ++csStart;
29  }
30  }
31  }
```

Line 1 is the name of the function and its inputs

Line 2 assigns a Boolean named “Swapped” and makes it “True”

Line 3 assigns CocktailSortStart to be 0, which is also the number of the first element in the Array

Line 4 assigns CocktailSortEnd to be ArraySize minus 1, which is the number of the last element in the Array.

Line 6 initiates a While-loop that means that everything from line 6 to line 31 will called until the codes goes back to the Boolean “Swapped” and it is set to “False”.

Line 7 will set the Boolean to “Swapped” to “False”, since this will be the state unless something gets swapped.

Line 8 is a for-loop that lopps from the beginning of the Array throughout the Array until it reaches the end. This loop goes from Line 8 until Line 13.

Line 9 checks if the i^{th} element in the Array is larger than the next element in the Array, and if that is the case, Line 10 will swap these two in the Array. And then line 11 will set the Boolean "Swapped" to be "True", because something has been swapped.

Line 14 then checks if the Boolean "Swapped" is not true, and in that case, line 15 would break out of the while loop that began in Line 6.

However, if "Swapped" is true, and the break function is not called, Line 18 set "Swapped" back to "False", and then Line 20 will set the last considered value of the Array one back, since the last element in the array is in its right spot.

Line 22 initiates another for-loop, where the i^{th} element is set to be the second to last element in the array, and then it loops through the array towards the beginning of it.

Line 23 checks if the current number is bigger than the latter element in the array, line 24 will then make these two elements swap, and Line 25 will set the Boolean "Swapped" to "True".

Line 28 ends with moving the considered start point of the array one element higher, since the one before this must be in its correct spot.

Analysis of asymptotic running time and memory usage

Randomized Selection

Asymptotic Running Time

The Randomized Select is based on a quicksort algorithm, and they both partition the input array recursively, but they differ from the fact that the quicksort algorithm processes both sides of the partition, whereby the Randomized Select processes one side of the partition. This also means that the quick-sort algorithm will have an expected running time of $\Theta(n \lg n)$, but since the Randomized Selection only processes one side of the partition, it ends up having an expected running time of $\Theta(n)$, assuming that the elements are distinct. Despite that it has an expected running time to be $\Theta(n \lg n)$, it is still a random algorithm, which means that it can be unlucky and check the highest i^{th} element when it is selecting the lowest, or if it checks the lowest i^{th}element, when selecting the highest, which means that it will check every single element twice. This situation leaves the Randomized Selection algorithm with a worst-case running time of $O(n^2)$.

Memory Usage

The Randomized Selection algorithm does only need to consider the array in addition to the selection it is performing on its pivot point. In the best case, where the array is already sorted, the algorithm only needs to consider one element at the time, meaning it has a best-case space complexity of $\theta(1)$. That is a rather unlikely, so the expected situation is that the algorithm has a relatively large number of elements it need to move, but as it gets sorted, the complexity should be lowered as the algorithm progresses through the array, meaning it will have an expected space complexity of $\Theta(\log n)$. In the opposite of the best case, where we had the already sorted algorithm, do we have the worst case, where the algorithm keeps choosing the worst possible element as pivot for its partitioning, which will end up having the algorithm consider all the elements n times, which gives the algorithm a worst-case space complexity of $O(n^2)$.

Cocktail Sort

Asymptotic Running Time

The Cocktail Sort algorithm is based on the bubble sort algorithm, and they are both expected to go over each element multiple times in average, which gives the cocktail sort an expected running time of $\mathrm{O}(n^2)$. The worst case happens, when array is in sorted in reverse, such the algorithm has to traverse through n elements n times, which leaves it with a worst-case running time of $O(n^2)$. If the array is already sorted, the algorithm will only traverse through the array one time and consider n elements, which will give the Cocktail Sort a best-case running time of $\mathrm{O}(n)$.

Memory Usage

Unlike the Randomized Selection which is considered a “Divide and Conquer” algorithm, the Cocktail Sort is considered a recursive sorting algorithm, which means it will always only consider one element at the time. This means that it might take a long time, but it will never take up more than one element of space at the time. This leaves the Cocktail Sort with a best-case space complexity of $\mathrm{O}(1)$, a worst-case space complexity of $O(1)$, and an average-case space complexity of $\mathrm{O}(1)$.

Experimental running time and memory usage

For both algorithms, I used Visual Studio to calculate the running time and memory usage. Both the Memory usage and the running time were measured on different data sizes being 100

elements, 1000 elements, and 10000 elements, created by Random.org[3]. Because of inconsistency in the running time, there was made three separate measurements, which was used to find an average. The following tables show the results:

Randomized Select:

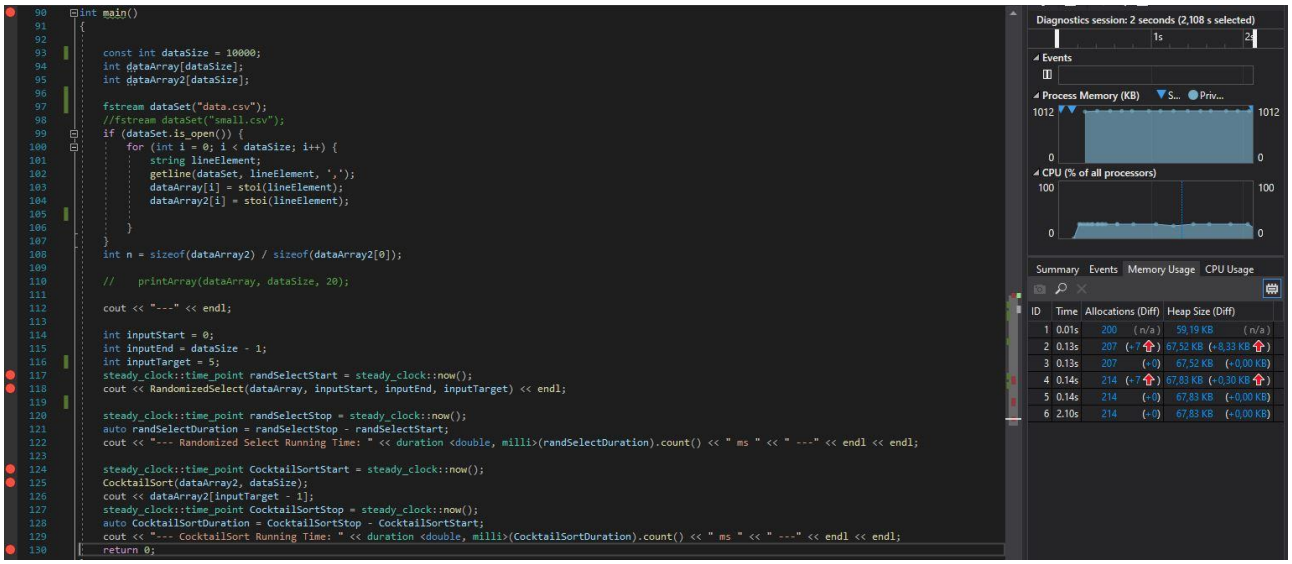
Datasize	100	1000	10000
Time 1	1.6948 ms	0.3432 ms	0.3243 ms
Time 2	0.6254 ms	0.3014 ms	0.3412 ms
Time 3	0.3061 ms	0.2599 ms	1.0912 ms
Average Time	**0.8754 ms**	**0.3015 ms**	**0.5856 ms**
Memory Usage	**0.00 KB**	**0.00 KB**	**0.00 KB**

Cocktail Sort

Datasize	100	1000	10000
Time 1	0.3230 ms	19.1945 ms	2450.78 ms
Time 2	0.4006 ms	19.8966 ms	2687.20 ms
Time 3	0.3255 ms	18.9648 ms	2274.39 ms
Average Time	**0.3497 ms**	**19.3520 ms**	**2470.79 ms**
Memory Usage	**0.00 KB**	**0.00 KB**	**0.00 KB**

This show that Cocktail Sort can compete well with Randomized Select on small data sets, but when the data sets get bigger, Randomized Select is very much more efficient than the Cocktail Sort. It is obvious from these averages, that the Cocktail Sort is relatively fast with small data sets and relatively slow with larger data sets. This is caused from the fact that is a recursive sorting algorithm with a low amount of constants. The Randomized Select however uses this “Divide and Conquer” method, which consists of some more constants, and divides the array into subarrays. This method however means that when the array of data becomes bigger, the algorithm does not take that much more time, and works more effective on larger data sets compared to recursive sorting algorithms.

The method for measuring the memory usage[4] can be seen on the picture below, where the break points are shown and the results from the measurements as well. This picture is taken for the code, when the datasize is 10.000 elements.



Conclusion

This mini-project found that the Selection Sort problem can be solved faster with the Cocktail Sort than the Randomized Select as long as the data size is very limited. For general use, the Randomized Select Algorithm would still be the better algorithm. These results also make sense when looking at the asymptotic running time, which would expect that the Randomized Select would perform better than Cocktail Sort, as the data set becomes larger. When looking at the memory usage, there was not much to analyse since it was always measured to be using 0.00 KB despite the data set containing 10.000 distinct elements.

Bibliography

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

[2]: https://www.geeksforgeeks.org/cocktail-sort/ - Cocktail Sort, by GeeksForGeeks

[3]: https://www.random.org/integer-sets/ - Create data sets, by Random.org

[4]: https://docs.microsoft.com/en-us/visualstudio/profiling/memory-usage?view=vs-2019 - Memory Usage Measuring in Visual Studio, by Microsoft