ADSSE Report:

The selection problem solved with Randomized-Select and Gnome-Sort

Mathias N. G. Faldt

Algorithms, Data Structures and Software Engineering – 2021

GitHub link: https://github.com/HamPaa35/ADSSE_MINI_PROJECT

Table of contents

Abstract

This report aims to solve the selection problem using Randomized-Select and Gnome Sort. Pseudocode and a description of each algorithm is presented, along with a theoretical evaluation of their running time. The average theoretical running time of Randomized-Select is $\Theta(n)$ and for Gnome-Sort it is $\Theta(n^2)$. A practical evaluation of the running time was performed on data sets of different sizes. It was concluded that the theoretical and practical running time roughly matched. Furthermore it was concluded that Randomized-Select was the optimal solution to the selection problem, as long as memory usage was not of concern.

Introduction

This mini project report will aim to solve the selection problem using Randomized-Select and Gnome-Sort. The selection problem entails that given a set, A of size n, containing n numbers and an integer, i, with 1 <= i <= n, compute the element, x, in A that is larger than exactly i – 1 other elements of A.
Furthermore the report will evaluate the theoretical and practical running time of the algorithms, and conclude which of the algorithms is optimal under certain circumstances.

Pseudocode

This will show pseudocode for the two algorithms, along with text to explain it.

Random Select

Randomized-Select (Cormen et al., 2009, 215) is a version of the Quick Sort algorithm that can be used to recursively solve the selection problem.
Randomized-Select differs from Quick Sort since it does not recursively sort at both sides of its pivot point, and only partially sorts the low side of the pivot.

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            swap Array[i] with Array[j]
8  swap Array[i + 1] with Array[End]
9  return i + 1
```

As part of Randomized-Select we need to generate a pivot point in the array where we know that all values in the indices above are larger, and all values in the indices below are smaller. To do this Partition is used.
In *line 2* we retrieve the value at the **End** index of the array, this value will be used as the pivot point.
Line 3 sets the **i** value outside the range of the array since this should return the first value of the array in case no values smaller than the pivot point are found.
Line 4 starts a for loop that loops from the start of the array until the index before the pivot point.
Line 5 checks if the value at index **j** of the array is smaller than the pivot point value. If this is the case **i** is increased by one and the values at **i** and **j** in the array are swapped, in *line 6 and 7*. This ensures that all values below the pivot point index are smaller.
Line 8 swaps the values at index **i+1** with the value at the **End** of the array, this ensures that the pivot point value is placed between the smaller and larger values.
Line 9 then returns the index of the pivot point.

Randomized-Partition

```
1  Randomized-Partition(Array[], Start, End)
2  i = Random(Start, End)
3  swap Array[End] with Array[i]
4  return Partition(Array, Start, End)
```

A part of the Randomized-Select algorithm is the Randomized-Partition function. This function generates a random pivot point for the partition function above. In *line 2* a pseudo random value between the Start and End index is created. In *line 3* this random number is then used as an index to swap the values of the array at the **i** and **End** index. This swap ensures that the value at the randomly generated index will be used as the comparison value (x) and pivot point in the Partition function that is called in *line 4*.

Randomized-Select

```
1   Randomized-Select(Array[], Start, End, ith-element)
2   if(Start == End)
3         return Array[Start]
4   pivot = Randomized-Partition(Array, Start, End)
5   k = pivot - start + 1
6   if (ith-element == k)
7         return Array[pivot]
8   else if (ith-element <  k)
9         return Randomized-Select(Array, Start, Pivot - 1, ith-element)
10  else return Randomized-Select(Array, Pivot + 1, End, ith-element-k)
```

The above pseudocode is the actual Randomized-Select function.
In *Line 2* we start by checking if the **Start** and **End** is the same, if this is the case we have reached the base of the recursion, and the **Array** consists of just 1 element. This must mean we have found the ith smallest element, and we return this in *line 3*.
If this is not the case we generate a random **pivot** point using the Randomized-Partition function described above, this happens in *line 4*.
Line 5 computes **k** which is the number of elements on the low side of the pivot point, including the pivot point itself.
Line 6 checks if the **k** value is the same as the **ith-element** we are looking for, if this is the case we return the value of the **Array** at the pivot, *see line 7*, since this is the element we are looking for.
If line 6 is not true *line 8* checks if the **ith-element** is on the low side of the pivot point, if this is the case *Line 9* recursively checks the low side of the pivot in the array.
Line 10 runs if the **ith-element** is on the high side of the pivot, and recursively checks the high side of this array.

Gnome-Sort

Gnome-Sort for selection problem

```
1 Gnome-Sort (Array[], Data-Size)
2 Current-Position = 0
3 While (Current-Position < Data-Size)
4       if (Current-Position == 0 or Array[Current-Position]
  >= Array[Current-Position - 1])
5             Current-Position = Current-Position + 1
6       else
7             swap Array[Current-Position] with Array[Current-Position - 1]
8             Current-Position = Current-Position - 1
```

The Above pseudocode is the algorithm known as “Gnome-Sort” (Grune, 2000), the algorithm is basically a more inefficient version of “insertion sort”. The main difference between them is that insertion sort remembers the index it has gotten to, when it swaps values that are out of order, and does not check the sorted elements again. Gnome-Sort does not keep track of this, and checks all the sorted elements again, until it reaches an element that is out of order.
Line 1 is the declaration of the function, and *line 2* starts the function at index 0 in the Array.
Line 3 starts a while loop that loops while the **Current-Position** is less then the **Data-Size**. This means the algorithm stops when the end of the array has been reached.

Line 4 checks if the current position is index 0 or if the value at the current array position is larger than the value at the previous array index. If this is the case, these values are sorted and the function will move onto the next array position, in *line 5*. If line 4 is not true the code jumps to *line 6*, here the code swaps the two values which are out of order, this happens in *line 7*. Then the code jumps to the previous array index in *line 8*, to check if the array is in order after the swap in line 7.

Analysis of the algorithms

Asymptotic running time and memory usage

Randomized-Select

This section will describe the asymptotic running time and memory usage of the Randomized-Select algorithm. The algorithm is based on the quicksort algorithm, and the main difference between them is that the Randomized-Select only recursively checks one side of the array, the side where you expect to find the desired ith-element. The average running time for the Randomized-Select algorithm is $\mathbf{\Theta}(n)$, in cases where the elements are distinct. The best case running time is $\mathbf{\Omega}(n)$, since all elements are checked in the algorithm. The worst case running time is $\mathbf{O}(n^2)$, since you can have a case where the algorithm always checks the highest ith-element, when selecting the lowest ith-element, and vice versa. This means that the algorithm could end up checking all elements twice. However the randomness of the algorithm makes this unlikely, and therefore the average running time is expected to be $\mathbf{\Theta}(n)$.
In terms of memory usage the Randomized-Select algorithm works in place meaning the original array is used for sorting and selecting the ith element, this gives the space complexity of $\mathbf{O}(n)$, in addition to the original array, since the algorithm needs n amount of variables to keep track of progress.
There are optimizations that can be made to the Randomized-Select algorithm, however the average case of running time is optimal. The worst case memory usage is however not optimal, and some optimisations to this have also been proposed, such as the Hoare optimisation which will bring the memory usage down to $\mathbf{O}(\log n)$.

Gnome-Sort

This section will analyze the asymptotic running time of the Gnome-Sort algorithm. The average running time of the algorithm is $\mathbf{\Theta}(n^2)$ since the algorithm will have to check all the elements multiple times, first to check if the element is out of order, and then to find the correct place in the sorted array. The best case running time is $\mathbf{\Omega}(n)$ since the algorithm will only check each element once if the array is already sorted. The worst case running time is $\mathbf{O}(n^2)$ since all elements will be checked twice if the array is in reverse order.
In terms of memory usage, the worst case complexity is $\mathbf{O}(1)$, plus the original array of size n, since only one variable is needed to keep track of the current position, and the array is sorted in place.

This algorithm is not optimal in terms of time complexity, the Randomized-Select algorithm above is more optimal in terms of solving the selection problem. Even a regular insertion sort will be more optimal in terms of constant factors. Insertion sort will not check elements that are already sorted, and will keep track of how far it was before putting the element in order.
In terms of memory usage it is optimal, the only additional memory needed is to keep track of the current position. Even insertion sort is worse in terms of constant memory factors, since it will need an additional variable to keep track of where it came from, however small this overhead is.

What to do in case of non-distinct elements

Both the algorithms would have trouble with the cases where the array contained non distinct elements. In these cases you would not be able to guarantee that the ith element would be larger than exactly i-1 other elements. You would only be able to guarantee that it is larger or equal to i-1 other elements.
A solution could be to preprocess the data by removing duplicate values in the array before solving the selection problem. This would however add more time and memory complexity, depending on the implementation. A method could be to go through the original array and store the values that are distinct in a new array.

Experimental analysis

Running time

Experimental setup

The code[1] was run in Visual Studio Code, using the "Code Runner" extension to compile and run. The practical running time of each algorithm was acquired using the code below. Each algorithm was run 3 times and the lowest result was picked, this is the same procedure as used by the python 3.5 timeit() function[2].

```
#include<chrono>
#include<ctime>

steady_clock::time_point start = steady_clock::now();
//ALGORITHM TO TIME
steady_clock::time_point stop = steady_clock::now();
auto duration = stop - start;
cout << "Duration of algorithm " << duration <double, milli>
(gnome_duration).count() << endl;
```

[1] https://github.com/HamPaa35/ADSSE_MINI_PROJECT
[2] https://docs.python.org/3/library/timeit.html

Three random sets of integers, of sizes listed below were generated using random.org[3]. The sets were in the range of 10-20.000, and then the value 1 was added to all sets manually, for debugging puposes. The ith-element was set as 2.

Set size	101	1001	10.001

Results

Below is the running times obtained by the experimental setup outlined above.

ith-element = 2	Time at 101	Time at 1001	Time at 10001
Randomized-Select	0.0065 ms	0.0175 ms	0.1408 ms
Gnome-Sort	0.0269 ms	1.8957 ms	194.506 ms

Randomized-Select

The results for Randomized-Select above are in line with the calculated expected average running time of $\Theta(n)$. When the data size increases by a factor of 10, the practical running time also increases by a factor of roughly 10.

Gnome-Sort

The practical results for the Gnome algorithm above is in line with the expected worst case and average running time of $\mathbf{O}(n^2)$ and $\Theta(n^2)$. When the data size increases the running time increases exponentially.

Memory usage

Setup and results

To measure the practical memory usage of the algorithm Visual Studio 2019 diagnostic tools were used. The algorithms were run on the largest data set of 10001 values. The results were the following:

ID	Time	Allocations (Diff)	Heap Size (Diff)
1	0.01s	203 (n/a)	68.28 KB (n/a)
2	0.11s	216 (+13 ↑)	72.88 KB (+4.60 KB ↑)
3	0.11s	216 (+0)	72.88 KB (+0.00 KB)
4	0.11s	217 (+1 ↑)	76.92 KB (+4.04 KB ↑)
5	0.11s	217 (+0)	76.92 KB (+0.00 KB)
6	1.50s	217 (+0)	76.92 KB (+0.00 KB)

The snapshot above was acquired using breakpoints in the Visual Studio Debugger. ID 1 is at the beginning of main(), ID 2 is just after the setup is run, and the array values are loaded, ID 3 is the Randomized-Select algorithm. ID 4, is just before the

[3] https://www.random.org/integer-sets/

Gnome-Sort algorithm, and after the timing code for Gnome-Sort has been initialized, ID 5 is the Gnome-Sort algorithm, and ID 6 is the end of the code.
As can be seen both algorithms have no effect on the overall allocations and heap sizes. The changes in allocations and heap size occur on the setup of the program. This means that the algorithms have no measurable impact on practical memory usage.
This could be due to the KB unit being too large to show the differences, it could be due to the way the .exe is compiled, or it could be due to how Windows 10 garbage collects. In short the memory impact of the algorithms is negligible with a data size of 10001 on a modern PC.

Conclusion

In conclusion this mini project has solved the selection problem for a set of distinct integers. The algorithms presented are the Randomized-Select algorithm and the Gnome-Sort algorithm. The algorithms represent two different methods for solving the selection problem. The average asymptotic running time for Randomized-Select and Gnome-Sort, is $\boldsymbol{\Theta}(n)$ and $\boldsymbol{\Theta}(n^2)$ respectively, the worst case memory usage is $\mathbf{O}(n)$ and $\mathbf{O}(1)$ respectively.
A practical evaluation of the algorithms were performed on data sets of different sizes. The practical running time was found to confirm the theoretical asymptotic running time. Randomised-Select had a linear increase in practical running time and Gnome-Sort has an exponential increase in practical running time. The practical evaluation of memory usage was not able to obtain results of a quality that could confirm or disconfirm the theoretical memory usage.
The theoretical and practical results from this mini project show that the Randomized-Select algorithm is optimal in terms of computational time needed to solve the selection problem. However, the theoretical memory usage shows that Gnome-Sort is the optimal choice in cases where memory usage is of concern. The Randomized-Select algorithm would however be the all round best choice for most modern computers.

References

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (Third Edition ed.). Massachusetts Institute of Technology. 9780262033848

Grune, D. (2000). *Gnome Sort - The Simplest Sort Algorithm*. Dick Grune. https://www.dickgrune.com/Programs/gnomesort.html