# Randomized select and shellsort select


## Abstract

This report presents two algorithms as solutions to the selection problem. The algorithms are randomized select and shellsort select. Each of these algorithms are implemented and have their complexity analysed theoretically and experimentally. The findings from this analysis are that randomized select is generally the better choice for the selection problem.


## The Selection Problem

A selection algorithm is used to find the *i*th smallest element in an array. In other words it identifies the value which would be in the *i*th index if the array was sorted. This can be done by simply sorting the array and then extracting said element though more efficient methods exist. The implementations in this report assume that the input contains only distinct elements.

## Randomized select

### Pseudo-code

```
1   Partition(PArray[], Pstart, Pend)
2       x = PArray[Pend]
3       i = Pstart - 1
4       for j = Pstart to Pend - 1
5          if PArray[j] <= x
6                i = i + 1
7                swap PArray[j] and PArray[i]
8       swap PArray[i+1] and PArray[Pend]
9       return i + 1
10
11  RandomizedPartition(RPArray[], RPstart, RPend)
12      i = random index between RPstart and RPend
13      swap RPArray[RPend] and RPArray[i]
14      return Partition(RPArray, RPstart, RPend)
15
16  RandomizedSelect(Array[], start, end, target) {
17      if start == end
18         return Array[start]
19      pivot = RandomizedPartition(Array, start, end)
20      k = pivot - start + 1
21      if target == k
```

```
22          return Array[pivot]
23       else if target < k
24          return RandomizedSelect(Array, start, pivot - 1, target)
25       else
26          return RandomizedSelect(Array, pivot + 1, end, target - k)
```

## Explanation

The randomized select algorithm is a variation on the quicksort to optimize for the selection problem. It does this by only sorting the side of each partition which contains the target index [Cormen, 2009].
The first step of the algorithm is to check for the trivial case of an array of 1 length. This is done in line 17-18. The algorithm then identifies the pivot through the RandomizedPartition() function.
This partition is a randomized quicksort. The random element is introduced in lines 12-13 where the end element of the subarray is swapped with a random other element in the array.
The Partition() function then compares the new end element with the rest of the subarray beginning from the start of the subarray. The subarray is sorted so that the elements smaller than the end elements are placed before it and the elements larger than it are placed after it. This is done in lines 2-8 and the index is returned.
This index is then the pivot used to evaluate if the correct index has been found or if the algorithm should continue searching. This is done in lines 21-22. If the algorithm should continue it then chooses the input arguments of the next recursion by checking if the found index is larger or smaller than the target index.

In the case of repeated elements, randomized select will handle it without errors but will be slower the more non-distinct elements there are. This is because the worst case partition becomes more likely as the number of repeated elements increases as selecting the same element twice in a row will only remove one element from the subarray.

# Shellsort selection

## Pseudo-code

```
1     Shellsort(Array[], SubStart, SubEnd)
2         n = (SubEnd - SubStart) + 1
3
4         for gap = n/2 to 1 halving each iteration
5           for i = gap + SubStart to SubEnd
6                 temp = Array[i]
7                 j = i
8                 while j >= gap and Array[j - gap] > temp
9                       j -= gap
10                      Array[j] = Array[j - gap]
11                Array[j] = temp;
12        return
13
14    ShellsortSelect(Array[], SubArrayStart, SubArrayEnd, Target)
15        Shellsort(Array, SubArrayStart, SubArrayEnd)
16        return Array[Target - 1 + SubArrayStart]
```

## Explanation

Shellsort is a sorting algorithm based on the insertion sort. One could also consider it a generalization of the insertion sort. When used for selection the algorithm is simply run normally and the *i*th index is then returned from the sorted array.
The algorithm seeks to optimize the insertion sort by first sorting subarrays containing elements which are far from each other in the full array. This is done using gaps which become smaller each iteration until they become one and the algorithm operates the same as a normal insertion sort [Shell, 1959].
In this implementation the gap starts as half of the length of the array and halves each iteration. This is done in lines 2-4.
The algorithm then iterates from the first gap up to the end of the array. Each iteration it checks if an element should be moved back in its subarray. It does this decrementing with the gap each iteration and comparing to the element which is one gap before. Within these subarrays the algorithm otherwise functions as an insertion sort and checks each element to all previous elements to find its correct position. This process can be seen in lines 5-11. This is repeated until a normal insertion sort with a gap size of one has been completed and the algorithm is then done and can return the value at the *i*th index as seen on line 16.

In the case of repeated elements shellsort will become faster as repeated elements increase the chances of an already sorted array which reduces the number of comparisons shellsort needs to make.

# Analysis

This section will analyse the complexity of the two algorithms. It will be done for both the theoretical complexity as well as the measured running times and memory usages.

## Theoretic

### Randomized select

Randomized select has a worst case time complexity of $\Theta(n^2)$, a best case time complexity of $\Theta(n)$, and an average time complexity of $\Theta(n)$ [Cormen, 2009]. This is not an optimal solution.

The worst case happens when each partition selects the largest element in the subarray which only results in a reduction in one element. Partitioning would then be done n times and each partition takes $\Theta(n)$ time resulting in $\Theta(n^2)$ time. The best case happens when the first pivot is the target index and takes the time of one partition which is $\Theta(n)$ time.

The worst case space complexity of randomized select is $\Theta(n)$ which happens in the same case as the algorithms worst case time complexity. In this case the recursive algorithm reaches the base case.

### Shellsort selection

The implemented version of shellsort's worst case time complexity is $O(n^2)$, the best case is $\Theta(n \log n)$, while a lower bound for the average case has been found to be $\Omega(pn^{1+1/p})$ where $p$ is the number of passes the sort executes [Jiang, 2000]. This is not an optimal solution.

The best case happens when the array is already sorted. In this case the final insertion sort takes $\Theta(n)$ time while each gap sort also takes $\Theta(n)$ time. The number of gap sorts is log(n) or the number of times you can halve n. This results in a Shellsort has a worst case space complexity of $\Theta(n \log n)$.

An upper bound for the time complexity of shellsort can be calculated if we assume each gap sort takes $\Theta(n^2)$ time which would result in $\Theta(n^2 \log n)$ time. This can never happen in practice though and as such it is not a tight bound and is therefore $O(n^2 \log n)$. This can be further bounded by looking closer at each individual gap sort. Insertion sort still has a worst case of $\Theta(n^2)$ but the second to last gap can at worst do $2 * (n/2)^2$ comparisons (two subarrays of length n/2) and the third to last $4 * (n/4)^2$ comparisons. This results in a series of $(n^2 + n^2/2 + n^2/4 + n^2/8...)$.

$(n^2 + n^2/2 + n^2/4 + n^2/8...) = n^2 * (1 + 1/2 + 1/4 + 1/8...) = n^2 * 2$

With this we can set an upper bound of $O(n^2)$ [Kline].

The space complexity of shellsort is $\Theta(1)$ as the algorithm sorts in place.

## Practical

The running time and memory usage measured on the two algorithms. All running times are in milliseconds. Each algorithm was run three times to give a better understanding of the algorithms.

**Randomized select:**

| **Datasize:** | **10** | **10000** | **100000** |
|---|---|---|---|
| Time 1: | 0.0015 | 0.0534 | 1.6575 |
| Time 2: | 0.0016 | 0.152 | 2.4971 |
| Time 3: | 0.0018 | 0.2107 | 0.7961 |
| **Average:** | **0.0016** | **0.1387** | **1.6502** |
| **Smallest:** | **0.0015** | **0.0534** | **0.7961** |
| **Memory Usage:** | **0.00 KB** | **0.00 KB** | **0.00KB** |

**Shellsort select:**

| **Datasize:** | **10** | **10000** | **100000** |
|---|---|---|---|
| Time 1: | 0.0007 | 1.8929 | 40.5209 |
| Time 2: | 0.0006 | 1.9429 | 38.409 |
| Time 3: | 0.0013 | 2.8828 | 31.4091 |
| **Average:** | **0.0009** | **2.2395** | **36.7797** |
| **Smallest:** | **0.0006** | **1.8929** | **31.4091** |
| **Memory Usage:** | **0.00 KB** | **0.00 KB** | **0.00KB** |

# Conclusion

It is apparent from the results that shellsort is better on very small datasets while randomized select is better on larger datasets. Meanwhile it can be seen that no discernible difference could be measured in memory usage. This means that in most cases the randomized selection is the better choice as the running time is almost always better and the difference in memory usage is small. This is even more so when it is considered that the theoretical memory usage of each algorithm is calculated without the base usage caused by the array which is sorted. If this memory usage is considered the worst case memory complexity of both algorithms are $\Theta(n)$. One case in which one might consider to use shellsort select over randomized select is if also sorting the array is useful.

# Bibliography

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, Third Edition* (3rd. ed.). The MIT Press.

Tao Jiang, Ming Li, and Paul Vitányi. 2000. A lower bound on the average-case complexity of shellsort. *J. ACM* 47, 5 (Sept. 2000), 905–911. DOI:https://doi.org/10.1145/355483.355488

Robert M. Kline. Shellsort & Algorithmic Comparisons. Retrieved April 29, 2021 from https://www.cs.wcupa.edu/rkline/ds/shell-comparison.html

D. L. Shell. 1959. A high-speed sorting procedure. Commun. ACM 2, 7 (July 1959), 30–32. DOI:https://doi.org/10.1145/368370.368387