# INFO0902 - Data Structures and Algorithms
# Project 1 - Selection Algorithms

DUY VU DINH (S2401627)

This report presents a theoretical and experimental analysis of four selection algorithms: `SelectionSelect`, `HeapSelect`, `QuickSelect`, and `FRSelect`. The objective of each algorithm is to determine the $k$-th smallest element in an array.

## 1 THEORETICAL ANALYSIS

### 1.1 Complexity table and justifications

Table 1 shows the time and space complexities in the worst case and the best case of the implemented algorithms, corresponding to their pseudocodes provided in Appendix A.

Table 1. Complexity of selection algorithms

| Algorithm | Time complexity | | Space complexity | |
|---|---|---|---|---|
| | Worst case | Best case | Worst case | Best case |
| SelectionSelect | $\Theta(Nk)$ | $\Theta(Nk)$ | $\Theta(1)$ | $\Theta(1)$ |
| HeapSelect | $\Theta(N + k\log N)$ | $\Theta(N + k\log N)$ | $\Theta(1)$ | $\Theta(1)$ |
| QuickSelect | $\Theta(N^2)$ | $\Theta(N)$ | $\Theta(N)$ | $\Theta(\log N)$ |
| FRSelect | $\Theta(N)$ | $\Theta(N)$ | $\Theta(\log N)$ | $\Theta(\log N)$ |

#### *1.1.1* `SelectionSelect`.

- **Time complexity**: The algorithm loops and compares every time. For each of the first $k$ positions, the algorithm performs up to $N$ comparisons. Therefore, the time complexity in both cases is $\Theta(Nk)$.
- **Space complexity**: The algorithm works in-place, meaning it runs without needing extra memory. So, $\Theta(1)$ is the complexity for both cases.

#### *1.1.2* `HeapSelect`.

- **Time complexity**: A heap building stage takes $\Theta(N)$. Then, a heapify takes $\Theta(\log N)$. Therefore, the whole algorithm takes $\Theta(N + k\log N)$.
- **Space complexity**: The algorithm works in-place, meaning it runs without needing extra memory.

#### *1.1.3* `QuickSelect`.

- **Time complexity**
  - **Best case**: The pivot divides the array into two balanced parts. Since the algorithm recurses only on one side, a subarray, the expected number of comparisons follows the recurrence $T(N) = T(N/2) + \Theta(N)$, resulting in a time complexity of $\Theta(N)$. The best case usually occurs for a random array.
  - **Worst case**: The pivot is always the smallest or largest element (e.g., for sorted arrays when the last element is used as pivot), the recursion depth becomes $N$, and the recurrence becomes $T(N) = T(N-1) + \Theta(N)$, resulting in $\Theta(N^2)$.
- **Space complexity**. The space complexity depends on the depth of the recursive calls.
  - **Best case**: It is $\Theta(\log N)$ due to the recursive stack.
  - **Worst case**: It becomes $\Theta(N)$ if the recursion goes as deep as the array size.

*1.1.4* `FRSelect` *(Floyd-Rivest select algorithm).*

- **Time complexity**
  - The FRSelect algorithm improves QuickSelect by estimating a good pivot range using sampling and probabilistic analysis. It reduces the likelihood of unbalanced partitions.
  - This approach ensures a worst-case and average-case time complexity of $\Theta(N)$, mentioned in Kiwiel and Krzysztof's work [2].
- **Space complexity**
  - The algorithm involves recursive pivot range refinement, which results in a space complexity of $\Theta(\log N)$ due to the recursion stack.

## 1.2 Complexity analysis of the average case of `Quickselect` in the case where $k = 0$ (the minimum)

*1.2.1 Intuition.* `QuickSelect` uses the same partitioning logic as `QuickSort`. However, unlike `QuickSort` which recursively explores both sides, `QuickSelect` only recurses into one subarray, either the left or the right, depending on where the $k$-th element lies.

In our case: $k = 0$ (i.e., looking for the minimum): If the pivot ends up in position 0 (smallest), we're done. Otherwise, we only work on the left side of the pivot (elements are smaller than it).

On average, about half of the array each time is eliminated, a partition step that costs $\Theta(N)$. And only one recursive path is gone down instead of two (unlike `QuickSort`). Therefore, the average time complexity of `QuickSelect` is $\Theta(N)$.

*1.2.2 Mathematical model.*

- Number of elements in the array must be positive: $N > 0$
- Number of comparisons for partitioning: $N - 1$.
- Probability that the pivot is at position $i$: $\frac{1}{N}$.
- Sizes of the sub-array in this case: $i - 1$
  - With the assumption of $k = 0$, it means looking for the smallest.
  - Unlike `QuickSort`, which recursively explores both sides, `QuickSelect` only recurses into one subarray, the left subarray $A[0, i - 1]$ when $k = 0$.

So, the recurrence is:

$$C_0 = 0 \tag{1}$$

$$C_N = (N-1) + \sum_{i=1}^{N-1} \frac{1}{N} C_i \tag{2}$$

Multiply Eq. (2) by $n$:

$$NC_N = N(N-1) + \sum_{i=1}^{N-1} C_i \tag{3}$$

The same formula of Eq. (4) for $(N - 1)$:

$$(N-1)C_{N-1} = (N-1)(N-2) + \sum_{i=1}^{N-2} C_i \tag{4}$$

Eq. (3) − Eq. (4):

$$NC_N - (N-1)C_{N-1} = N(N-1) - (N-1)(N-2) + C_{N-1} \tag{5}$$

$$\Leftrightarrow \quad NC_N = NC_{N-1} + 2(N-1) \tag{6}$$

$$\Leftrightarrow \quad C_N = C_{N-1} + \frac{2(N-1)}{N} \tag{7}$$

Then, Eq. (7) is telescoped:

$$C_N = C_1 + \sum_{a=2}^{N} \frac{2(a-1)}{a} \tag{8}$$

$$\Leftrightarrow \quad C_N = C_1 + 2N - 2\sum_{a=1}^{N} \frac{1}{a} \tag{9}$$

The sum of $\sum_{a=1}^{N} \frac{1}{a}$ simplifies into a form involving harmonic numbers $H_N = \sum_{a=1}^{N} \frac{1}{a}$, then:

$$C_N = 2N - 2H_N + C_1 \tag{10}$$

Due to $H_N \in \Theta(\log N)$, therefore:

$$C_N \in \Theta(N) \tag{11}$$

So, the expected complexity to find the minimum ($k = 0$) using QuickSelect is $\Theta(N)$.

### 1.3 The stable of algorithm

All of the mentioned algorithms are unstable. Justifications:

- `SelectionSelect` swaps non-adjacent elements.
- `HeapSelect` includes Heapify and extract-min disrupt the original positions of duplicates.
- Both of `QuickSelect` and `FRSelect` use partitioning moves to equal elements without considering original order (not order-preserving).

## 2 EXPERIMENTS

### 2.1 Experiment results

The experiment results, including the average computation times and the average number of comparisons over 10 experiments, of each mentioned algorithm for the search of median and 10-th percentile are shown in Table 2 and Table 3, respectively. Unfortunately, due to the limitation of resources, I cannot run the cases for $N = 10^6$ for the `SelectionSelect` and `QuickSelect` algorithms. However, the experiments regarding $N = 10^4$ and $N = 10^5$ are quite enough for them to analyze the complexity. In addition, the computation times and the number of comparisons for each algorithm with $N = 10^6$ are expected to follow the theoretical evolution of complexity in Section 2.2.1.

Table 2. Computation times and the number of computations of the algorithms for the search of the median ($k = N/2$).

| Metric | Type of array | random | | | increasing | | | decreasing | | | constant | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Size | $10^4$ | $10^5$ | $10^6$ | $10^4$ | $10^5$ | $10^6$ | $10^4$ | $10^5$ | $10^6$ | $10^4$ | $10^5$ | $10^6$ |
| q Time [s] | SelectionSelect | 0.05569 | 5.51715 | - | 0.05480 | 5.49899 | - | 0.05839 | 6.95391 | - | 0.05491 | 5.51944 | - |
| | HeapSelect | 0.00078 | 0.00940 | 0.11738 | 0.00059 | 0.00700 | 0.08255 | 0.00067 | 0.00755 | 0.09756 | 0.00005 | 0.00051 | 0.00551 |
| | QuickSelect | 0.00014 | 0.00140 | - | 0.12229 | 12.39652 | - | 0.11824 | 11.03308 | - | 0.12279 | 12.37940 | - |
| | FRSelect | 0.00010 | 0.00083 | 0.00717 | 0.00001 | 0.00015 | 0.00223 | 0.00010 | 0.00081 | 0.00744 | 0.00005 | 0.00045 | 0.00465 |
| #comps | SelectionSelect | 37502499 | 3750024999 | - | 37502499 | 3750024999 | - | 37502499 | 3750024999 | - | 37502499 | 3750024999 | - |
| | HeapSelect | 137082 | 1704351 | 20315900 | 128095 | 1609845 | 19421172 | 142375 | 1756943 | 20865881 | 20001 | 200001 | 2000001 |
| | QuickSelect | 33510 | 337053 | - | 37497500 | 3749975000 | - | 49530691 | 4567259156 | - | 37497500 | 3749975000 | - |
| | FRSelect | 28662 | 228256 | 2053122 | 10237 | 128132 | 3023096 | 60642 | 516838 | 3715446 | 21036 | 205538 | 2027434 |

Table 3. Computation times and the number of computations of the algorithms for the search of the 10-th percentile ($k = N/10$).

| Metric | Type of array | random | | | increasing | | | decreasing | | | constant | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Size | $10^4$ | $10^5$ | $10^6$ | $10^4$ | $10^5$ | $10^6$ | $10^4$ | $10^5$ | $10^6$ | $10^4$ | $10^5$ | $10^6$ |
| Time [s] | SelectionSelect | 0.01400 | 1.40042 | - | 0.01396 | 1.39636 | - | 0.01574 | 1.94957 | - | 0.01389 | 1.39717 | - |
| | HeapSelect | 0.00023 | 0.00282 | 0.03382 | 0.00013 | 0.00156 | 0.01867 | 0.00018 | 0.00215 | 0.02580 | 0.00003 | 0.00028 | 0.00290 |
| | QuickSelect | 0.00011 | 0.00124 | - | 0.16220 | 16.31266 | - | 0.04240 | 3.97832 | - | 0.16169 | 16.34995 | - |
| | FRSelect | 0.00006 | 0.00043 | 0.00359 | 0.00002 | 0.00016 | 0.00208 | 0.00006 | 0.00054 | 0.00538 | 0.00005 | 0.00046 | 0.00467 |
| #comps | SelectionSelect | 9508499 | 950084999 | - | 9508499 | 950084999 | - | 9508499 | 950084999 | - | 9508499 | 950084999 | - |
| | HeapSelect | 43315 | 499237 | 5645068 | 34385 | 411295 | 4778017 | 45756 | 517608 | 5783012 | 12001 | 120001 | 1200001 |
| | QuickSelect | 26776 | 292732 | - | 49495500 | 4949955000 | - | 17828014 | 1642308451 | - | 49495500 | 4949955000 | - |
| | FRSelect | 16991 | 148697 | 1538045 | 10238 | 110162 | 1634026 | 42912 | 380860 | 3564183 | 21144 | 205874 | 2028430 |

### 2.2 Comments

*2.2.1 Compare the evolution of computation times.* Table 4 shows the increasing rates of computation times from $N = 10^4$ to $N = 10^5$ and from $N = 10^5$ to $N = 10^6$ regarding the experiments, while Table 5 provides the expected growth based on the complexity.

According to Table 4 and Table 5 as well as Table 1, the experiments generally match these expectations:

- `SelectionSelect` consistently witnesses a near 100-time increase, regardless of array type or $k$, which aligns perfectly with its $\Theta(Nk)$ time complexity. Since both $N$ and $k$ grow by 10 times, the overall work scales by 100 times.
- `HeapSelect` demonstrates a time increase in the range of 9.33 to 12.26 times, closely matching the theoretical complexity of $\Theta(N + k \log N)$. The growth is slightly sublinear in $k$ as expected (both $N$ and $k$ linearly grow by 10 times, while $\log N$ increases slightly over 1).

Table 4. The observed evolution of computation times.

| Evolution range | Type of array | random | | increasing | | decreasing | | constant | |
|---|---|---|---|---|---|---|---|---|---|
| | $k$ | $N/2$ | $N/10$ | $N/2$ | $N/10$ | $N/2$ | $N/10$ | $N/2$ | $N/10$ |
| $10^4 \longrightarrow 10^5$ | SelectionSelect | 99.07 | 100.03 | 100.35 | 100.03 | 119.09 | 123.86 | 100.52 | 100.59 |
| | HeapSelect | 12.05 | 12.26 | 11.86 | 12.00 | 11.27 | 11.94 | 10.20 | 9.33 |
| | QuickSelect | 10.00 | 11.27 | 101.37 | 100.57 | 93.31 | 93.83 | 100.82 | 101.12 |
| | FRSelect | 8.30 | 7.17 | 15.00 | 8.00 | 8.10 | 9.00 | 9.00 | 9.20 |
| $10^5 \longrightarrow 10^6$ | SelectionSelect | - | - | - | - | - | - | - | - |
| | HeapSelect | 12.49 | 11.99 | 11.79 | 11.97 | 12.92 | 12.00 | 10.80 | 10.36 |
| | QuickSelect | - | - | - | - | - | - | - | - |
| | FRSelect | 8.64 | 8.35 | 14.87 | 13.00 | 9.19 | 9.96 | 10.33 | 10.15 |

Table 5. The theoretical evolution in terms of complexity.

| Type of complexity | $\Theta(Nk)$ | | $\Theta(N + k \log N)$ | | $\Theta(N)$ | | $\Theta(N^2)$ | |
|---|---|---|---|---|---|---|---|---|
| $k$ | $N/2$ | $N/10$ | $N/2$ | $N/10$ | $N/2$ | $N/10$ | $N/2$ | $N/10$ |
| $10^4 \longrightarrow 10^5$ | 100.00 | 100.00 | 12.17 | 11.43 | 10.00 | 10.00 | 100.00 | 100.00 |
| $10^5 \longrightarrow 10^6$ | 100.00 | 100.00 | 11.79 | 11.25 | 10.00 | 10.00 | 100.00 | 100.00 |

- `QuickSelect` aligns well with its average-case complexity $\Theta(N)$ only for random arrays, where pivot positions are likely to be balanced. However, for sorted and constant arrays, it exhibits $\Theta(N^2)$ behavior, due to always choosing the last element as pivot. This leads to highly unbalanced partitions, which aligns to the worst case of complexity provided in Section 1.1, and thus a recursive depth approaching $N$, as confirmed by over 100-time growth in time.
- `FRSelect` shows the most consistent and optimal growth, close to 8-10 times, which aligns with its theoretical $\Theta(N)$ performance across all array types. It performance is stable even on sorted, constant arrays, unlike `QuickSelect`.

The results of `SelectionSelect` and `QuickSelect` in terms of $N = 10^6$ are expected to correspond to the increase of complexity from $N = 10^5$ to $N = 10^6$.

*2.2.2 Comment on the relative order of the different algorithms.* Based on the experiment results, the relative order depends on the type of array:

- For random arrays: `FRSelect` < `QuickSelect` < `HeapSelect` < `SelectionSelect`
- For increase, decrease or constant arrays: `FRSelect` < `HeapSelect` < `SelectionSelect` < `QuickSelect`

In general, `FRSelect` dominates when input order is random or when values are uniform, while `HeapSelect` is the best fallback in worst-aligned cases. `QuickSelect` is highly efficient only under favorable pivot conditions.

- `FRSelect` consistently outperforms the other algorithms across all array types. Its refined pivot selection, ensures balanced partitioning and avoids worst-case behavior. It achieves linear performance in all tested cases, closely matching its theoretical complexity of $\Theta(N)$.
- `QuickSelect` performs very well on random arrays due to its average-case complexity of $\Theta(N)$. However, it performs poorly on sorted or constant arrays because the current implementation always chooses the last element as pivot, resulting in highly unbalanced partitions and $\Theta(N^2)$ behavior in the worst case. This caused significant slowdowns and made it infeasible to run for large inputs like $N = 10^6$.

- `HeapSelect` exhibits stable and predictable performance, regardless of the input type. Its complexity of $\Theta(N + k \log N)$ reflects the cost of heap construction and repeated extract-min operations. While not the fastest, it avoids the extreme behavior seen in QuickSelect and consistently ranks in the middle of the group.
- `SelectionSelect` is the slowest algorithm in all caeses. Its simply design resembles selection sort and has a time complexity of $\Theta(Nk)$, making it scale poorly with both $n$ and $k$. As a result, it becomes impractical for large datasets and was unable to complete execution for $N = 10^6$ due to resource limitations.

*2.2.3 Discuss the impact of the parameter $k$.* Table 6 presents the ratio of computation times between selecting the median ($k = N/2$) and the 10th percentile ($k = N/10$) for each algorithm.

Table 6. Ratio of computation times between selecting the median and the 10-th percentile.

| Type of array | random | | | increasing | | | decreasing | | | constant | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Size | $10^4$ | $10^5$ | $10^6$ | $10^4$ | $10^5$ | $10^6$ | $10^4$ | $10^5$ | $10^6$ | $10^4$ | $10^5$ | $10^6$ |
| SelectionSelect | 3.98 | 3.94 | - | 3.93 | 3.94 | - | 3.71 | 3.57 | - | 3.95 | 3.95 | - |
| HeapSelect | 3.39 | 3.33 | 3.47 | 4.54 | 4.49 | 4.42 | 3.72 | 3.51 | 3.78 | 1.67 | 1.82 | 1.90 |
| QuickSelect | 1.27 | 1.13 | - | 0.75 | 0.76 | - | 2.79 | 2.77 | - | 0.76 | 0.76 | - |
| FRSelect | 1.67 | 1.93 | 2.00 | 0.50 | 0.94 | 1.07 | 1.67 | 1.50 | 1.38 | 1.00 | 0.98 | 1.00 |

Overall, only `SelectionSelect` and `HeapSelect` exhibit strong dependences on $k$, while `QuickSelect` and `FRSelect` tend to behave uniformly regardless of $k$, assuming favorable pivot distributions:

- `SelectionSelect` is directly and significantly influenced by $k$, as it performs one full scan per increment of $i$ up to $k$. As a result, time grows linearly with $k$, as confirmed by the 4-time difference between times for $k = N/10$ and $k = N/2$.
  - The 4-time difference is reasonable. Although $k$ increases by a factor of 5 between the median and 10-th percentile due to the time complexity of $\Theta(Nk)$, and the inner loop becomes slightly shorter for higher values of $j$ in the outer loop. As a result, the growth in runtime is sublinear in practice despite the theoretical $\Theta(Nk)$ bound, and an around 4-time increase instead of 5-time aligns well with this behavior.
- `HeapSelect` is also impacted by $k$, but less dramatically. Since each *MinHeapify* operation costs logarithmic time, performance scales in the order of $k \log N$, which was visible in the time increases observed.
- `QuickSelect` and `FRSelect` were generally insensitive to the value of $k$. In these algorithms, the number of recursive steps depends on the pivot position rather than $k$ itself. However, input structure still plays a larger role than $k$ in determining actual performance.

## REFERENCES

[1] Robert W Floyd and Ronald L Rivest. Algorithm 489: The algorithm select—for finding the i th smallest of n elements [m1]. *Communications of the ACM*, 18(3):173, 1975.

[2] Krzysztof C Kiwiel. On floyd and rivest's select algorithm. *Theoretical Computer Science*, 347(1-2):214–238, 2005.

## A APPENDIX: PSEUDOCODE

### A.1 SelectionSort

```
SelectionSelect(A, k)
1  for i = 1 to k
2      smallest = i
3      for j = i + 1 to A.length
4          if A[j] < A[smallest]
5              smallest = j
6      swap(A[i], A[smallest])
7  return k
```

### A.2 HeapSelect

```
HeapSelect(A, k)
1  BuildMinHeap(A)
2  A.heap_size = A.length
3  for i = 1 to k
4      swap(A[1], A[A.heapSize − 1])
5      heap_size = heap_size − 1
6      MinHeapify(A, 1)
7  return A.length − k // or heap_size
```

```
BuildMinHeap(A)
1  for i = ⌊length/2⌋ downto 1
2      MinHeapify(A, i)
```

```
minHeapify(A, i)
1   smallest = i
2   l = LEFT(i) // 2i
3   r = RIGHT(i) // 2i + 1
4   if l ≤ heap_size and A[l] < A[smallest]
5       smallest = l
6   if r ≤ heap_size and A[r] < A[smallest]
7       smallest = r
8   if smallest ≠ i
9       swap(A[i], A[smallest])
10      minHeapify(A, smallest)
```

### A.3 QuickSelect

```
QuickSelect(A, p, r, k)
1  if p = r
2      return p
3  q = Partition(A, p, r)
4  if k = q
5      return q
6  elseif k < q
7      return QuickSelect(A, p, q − 1, k)
8  else return QuickSelect(A, q + 1, r, k)
```

```
Partition(A, p, r)
1  i = p − 1
2  for j = p to r − 1
3      if A[j] ≤ A[r]
4          i = i + 1
5          swap(A[i], A[j])
6  swap(A[i + 1], A[r])
7  return i + 1
```

### A.4 FRSelect

Floyd et al. introduced the Floyd-Rivest Algorithm with the pseudocode [1].