Empirical Comparison of QuickSelect and LazySelect Algorithms

Student Charels Hugo
Course INFO-F413
Instructor Jean Cardinal

Abstract

This report presents the implementation and empirical comparison of two randomized selection algorithms: QuickSelect and LazySelect. We validate the theoretical number of comparisons through extensive testing on input sizes ranging from 10^4 to 10^7. The report highlights the implementation details, experimental setup, and conclusions on the efficiency and practical use of each algorithm. Based on our results, LazySelect shows a clear advantage for input sizes larger than two million elements, while QuickSelect performs better on smaller inputs.

1 Introduction

The selection problem involves finding the k-th smallest (or largest) element in an unsorted array. QuickSelect is a widely used randomized algorithm for this task, while LazySelect is an optimized variant aimed at reducing the number of comparisons and improving overall efficiency for large datasets.

This report implements, analyzes, and empirically compares these two algorithms, focusing on the number of comparisons, the actual running time, and the number of comparisons depending of k. The latter metric is particularly interesting as it highlights how the efficiency of each algorithm changes with the position of the target element. Our goal is to determine which algorithm is more suitable for different input sizes, with practical recommendations based on the results.

2 Algorithm Descriptions

2.1 QuickSelect

QuickSelect is a divide-and-conquer algorithm that selects a random pivot, partitions the array, and recursively selects from one partition. The expected number of comparisons is given by:

$$\mathbb{E}[\#\text{comparisons}] = 2n + 2k \ln\left(\frac{n-k}{k}\right) + 2n \ln\left(\frac{n}{n-k}\right),$$

which can be approximated asymptotically as $2n(1+h(\alpha))$, where $\alpha = \frac{k}{n}$ and $h(\alpha)$ is the entropy function describing uncertainty in partitioning. For simplicity, the expected number of comparisons is at most $2n(1+\ln 2) \simeq 3.386n$, providing a straightforward estimation.

2.2 LazySelect

LazySelect improves upon QuickSelect by using a sampling-based strategy. It selects a subset of elements and employs two pivots to partition the array, reducing the search space. This method ensures that, with high probability, the algorithm performs only $2n + o(n)$ comparisons. As a result, LazySelect achieves performance comparable to the best deterministic selection algorithms, which require at least $3n$ comparisons in the worst case.

3 Implementation Details

Both algorithms were implemented in Rust. The QuickSelect implementation follows the pseudocode from [2], while LazySelect was based on [1]. Some adjustments were made to correct minor issues and enhance performance. The full source code is included in Appendix, where detailed comments and explanations are provided for each part of the implementation.

4 Experimental Plan

The experiment is designed to evaluate and compare the performance of QuickSelect and LazySelect by analyzing both the number of comparisons and the execution time. In addition, we will examine how the number of comparisons varies for different k values within the same list size. All executions are performed on lists of random numbers between 0 and the list size, with k also chosen randomly within the same range. The plan consists of the following steps:

- Implement QuickSelect and LazySelect in Rust.
- Track the number of comparisons using a custom 'Integer' class that increments a counter for each comparison.
- Run both algorithms on input arrays of sizes ranging from 10^4 to 10^7, with a specific increment to obtain 20 points between the range each time.
- Measure the runtime and number of comparisons for each input size.
- Perform 100 trials for each input size, averaging the results to obtain reliable data.
- Additionally, compare the number of comparisons for value of k between 0 and the list size -1 within the same input size to observe how the algorithms behave depending on the position of the selected element.
- Generate graphs to visualize the relationship between input size, k-values, number of comparisons, and runtime execution.
- Analyze the results and discuss any patterns or deviations from theoretical expectations, particularly focusing on large input sizes and variations in k-values.

5 Experimental Results

The following section presents and analyzes the experimental results obtained from running the QuickSelect and LazySelect algorithms on varying list sizes. The experiments focus on two key metrics: the number of comparisons and the runtime (in milliseconds). Additionally, we analyze the number of comparisons based on different k-values within a fixed list size.

5.1 Number of Comparisons and Runtime per List Size

We first compare the number of comparisons and the runtime for varying list sizes, starting from $n = 5000$ to $n = 10^7$, with intervals chosen to show the algorithm's performance over both small and large data sets.

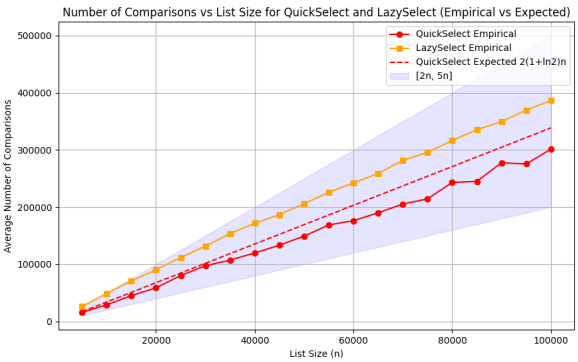

Figure 1: Number of comparisons per list size, $n \in [5000, 100000]$

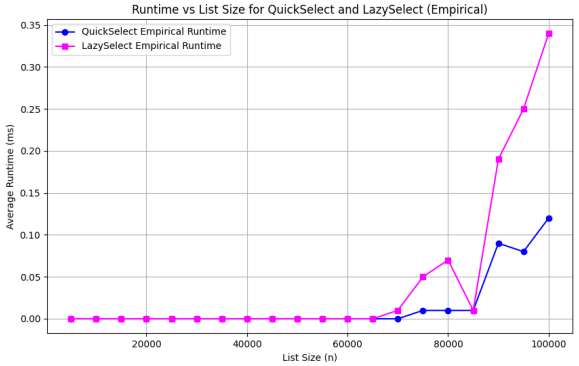

Figure 2: Runtime (ms) per list size, $n \in [5000, 100000]$

In the first pair of graphs, for smaller input sizes ($5000 \leq n \leq 100000$):

- The **number of comparisons** increases linearly as the list size grows. This aligns with the expected behavior for both QuickSelect and LazySelect, where comparisons scale with $O(n)$. In this range, we observe that the growth in comparisons is more than $2n$ but less than $5n$, confirming the linear complexity.

- The **runtime** for these sizes remains nearly constant up to a list size of $n = 70000$, due to Rust's efficient compilation and optimizations at the O3 level. Beyond this size, the runtime begins to increase.

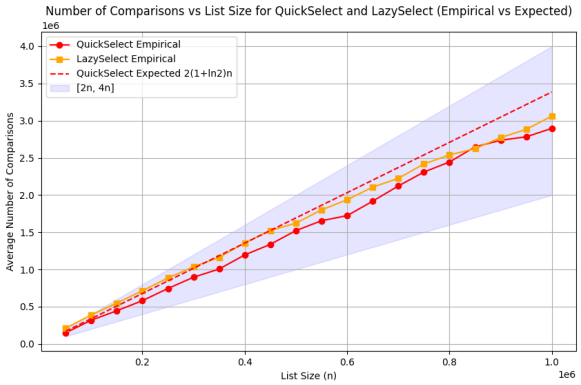

Figure 3: Number of comparisons per list size, $n \in [50000, 1000000]$

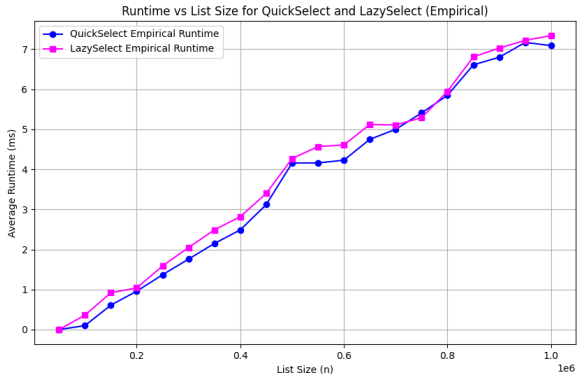

Figure 4: Runtime (ms) per list size, $n \in [50000, 1000000]$

For **medium-sized lists** ($50000 \leq n \leq 10^6$):

- The **number of comparisons** continues to show linear scaling, but the range decreases compared to smaller sizes, now falling between $2n$ and $4n$.
- The **runtime** curve mirrors the comparison results, increasing consistently as the list size grows larger.

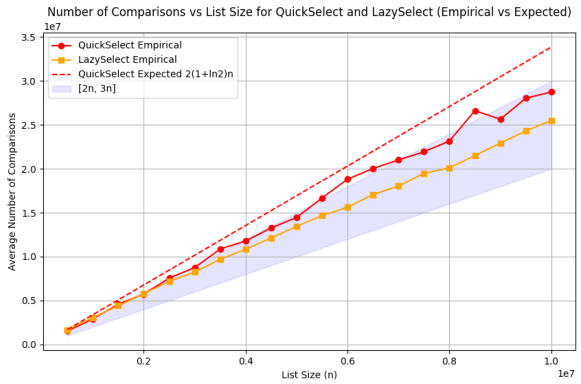

Figure 5: Number of comparisons per list size, $n \in [500000, 10000000]$

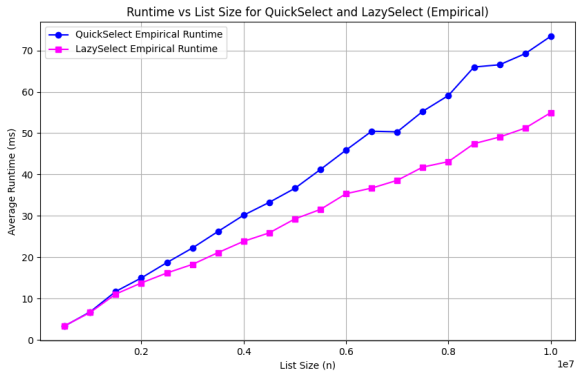

Figure 6: Runtime (ms) per list size, $n \in [500000, 10000000]$

For **large lists** ($n = 5 \times 10^5$ to $n = 10^7$):

- The **number of comparisons** shows that LazySelect outperforms QuickSelect, with the number of comparisons now ranging between $2n$ and $3n$, significantly better than QuickSelect's higher comparison counts.
- The **runtime** follows the same trend, with LazySelect running faster than QuickSelect for these larger list sizes.

These results suggest that LazySelect becomes the more efficient algorithm for large input sizes, both in terms of the number of comparisons and runtime, while QuickSelect is better suited for smaller lists due to its lower overhead.

5.2 Number of Comparisons for Each Value of k

In this section, we investigate how the number of comparisons varies with different values of k, which represents the rank of the element to be selected in the list.

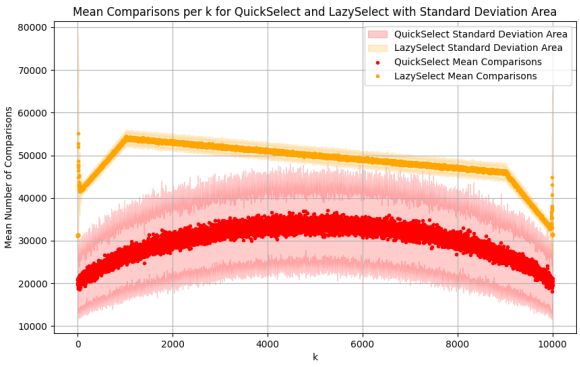

Figure 7: Number of comparisons per k, $n = 10000$ and $k \in [0, 10000]$

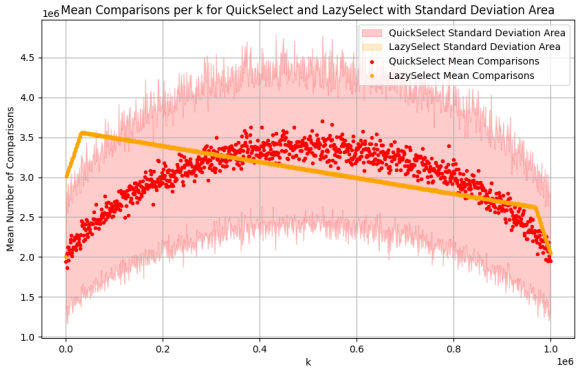

Figure 8: Number of comparisons per k, $n = 1000000$ and $k \in [0, 1000000]$

The results show that for LazySelect, the number of comparisons is highest when k is near the median, and decreases as k approaches the extremum values, i.e., near $k = 0$ or $k = n$. LazySelect displays a completely different behavior from QuickSelect, one that would be interesting to explore further and analyze in depth. However, this is beyond the scope of this report. Additionally, QuickSelect has a larger standard deviation in the number of comparisons compared to LazySelect, which maintains almost no standard deviation across different values of k.

6 Discussion

The experimental results demonstrate a clear relationship between the input size and the efficiency of QuickSelect and LazySelect. For smaller list sizes, QuickSelect performs better due to its lower overhead and fewer constant factors, making it faster in practice. However, as the list size increases, LazySelect surpasses QuickSelect, thanks to its improved sampling strategy and fewer overall comparisons.

For each value of k, we observe that QuickSelect's performance is particularly interesting. It requires the highest number of comparisons when selecting elements near the median of the list, and this number decreases as k approaches the extremum values. QuickSelect also shows a greater variance in its performance compared to LazySelect, which maintains a much more stable number of comparisons regardless of k.

Despite the additional overhead in sampling, the runtime results indicate that LazySelect's overall performance is not negatively impacted. The runtime curves align closely with the comparison curves, suggesting that the sampling process does not introduce a significant penalty, further emphasizing LazySelect's efficiency in larger datasets.

These results suggest that LazySelect has distinct advantages, especially for large datasets, as it minimizes comparisons more effectively than QuickSelect. Its unique behavior when k is near the extremum values presents a potential for further study, but that is not the focus of this report.

7 Conclusion

In this report, we have implemented and empirically compared two randomized selection algorithms: QuickSelect and LazySelect. Our experiments validate the theoretical performance claims for both algorithms. QuickSelect performs efficiently for smaller datasets, while LazySelect outperforms QuickSelect for larger lists, with the tipping point occurring around two million elements.

These findings offer practical guidance on selecting the appropriate algorithm depending on the dataset size and the specific requirements of the task. LazySelect is particularly advantageous when large datasets or a significant reduction in comparisons is needed. Future work could extend this analysis to other selection algorithms and explore further optimizations within LazySelect.

References

[1] Rajeev Motwani. *Randomized Algorithms*, pages 47–51. Cambridge University Press, 1995.

[2] Wikipédia. Quickselect — wikipédia, l'encyclopédie libre, 2024. [En ligne; Page disponible le 1-mai-2024].

A Source Code

A.1 QuickSelect Implementation

```rust
pub fn quick_select<T: Ord + Copy>(s: &[T], k: usize) -> T {
    // create a list copy so that the original list is not modified
    let mut new_s = s.to_vec();
    _quick_select(&mut new_s, 0, s.len() - 1, k)
}

fn _quick_select<T: Ord + Copy>(s: &mut [T], low: usize, high: usize, k: usize) -> T {
    if low == high {
        return s[low];
    }
    let pivot_index = _partition(s, low, high);
    if k == pivot_index {
        s[k]
    } else if k < pivot_index {
        _quick_select(s, low, pivot_index - 1, k)
    } else {
        _quick_select(s, pivot_index + 1, high, k)
    }
}

fn _partition<T: Ord + Copy>(s: &mut [T], low: usize, high: usize) -> usize {
    let pivot = s[high];
    let mut i = low;
    for j in low..high {
        if s[j] <= pivot {
            s.swap(i, j);
            i += 1;
        }
    }
    s.swap(i, high);
    i
}
```

Listing 1: QuickSelect

A.2 LazySelect Implementation

```rust
pub fn lazy_select<T: Ord + Copy>(s: &[T], k: usize) -> T {
    let n = s.len();
    let n_3_4 = (n as f64).powf(0.75) as usize;
    let n_1_4 = (n as f64).powf(0.25) as usize;
    loop {
        // Step 1: Pick n^(3/4) elements randomly with replacement
        let mut rng = rand::thread_rng();
        let mut r: Vec<T> = (0..n_3_4).map(|_| *s.choose(&mut rng).unwrap()).collect
            ();

        // Step 2: Sort R
        r.sort();

        // Step 3: Select a and b
        let x = k as f64 * (n as f64).powf(-0.25);
        let l = (x - (n as f64).sqrt().floor()).max(1.0) as usize;
        let h = (x + (n as f64).sqrt().ceil()).min(n_3_4 as f64) as usize;
        let a = r[l-1];
        let b = r[h-1];

        // Step 4: Partition S based on a and b and find the rank of a
```

```
        let mut p: Vec<T> = Vec::new();
        let index;

        match k {
            _ if k < n_1_4 => {
                p = s.iter().filter(|&&y| y <= b).cloned().collect();
                index = k;
            },
            _ if k > n - n_1_4 => {
                p = s.iter().filter(|&&y| y >= a).cloned().collect();
                index = k - (n - p.len()); // rank of a is equal to n - p.len()
            },
            _ => {
                let mut rank_a = 0;
                for &y in s.iter() {
                    if y < a {
                        rank_a += 1;
                    } else if y <= b {
                        p.push(y);
                    }
                }
                if rank_a > k { continue; }
                index = k - rank_a;
            }
        }

        if p.len() <= 4 * n_3_4 + 2 && index < p.len() {
            // Step 5: Sort P and find the k-th smallest element
            p.sort();
            return p[index];
        }
    }
}
```

Listing 2: LazySelect

A.3 Integer Class Implemenation

```
use std::cmp::Ordering;

#[derive(Debug, Copy, Clone)]
pub(crate) struct Integer {
    value: i32,
}

// Static variable to store the number of comparisons
static mut COMPARISONS: usize = 0;

impl Integer {
    // Constructor
    pub(crate) fn new(value: i32) -> Self {
        Integer { value }
    }

    // Method to clear the comparison count
    pub(crate) fn clear_comparisons() {
        unsafe {
            COMPARISONS = 0;
        }
    }

    // Helper method to update the comparison count
    fn update_comparisons() {
        unsafe {
```

```
            COMPARISONS += 1;
        }
    }

    // Function to get the current comparison count
    pub(crate) fn get_comparisons() -> usize {
        unsafe { COMPARISONS }
    }
}

// Implement the Eq, PartialEq, PartialOrd, and Ord traits for Integer
impl Eq for Integer {}

impl PartialEq<Self> for Integer {
    fn eq(&self, other: &Self) -> bool {
        Integer::update_comparisons();
        self.value == other.value
    }
}

impl Ord for Integer {
    fn cmp(&self, other: &Self) -> Ordering {
        Integer::update_comparisons();
        self.value.cmp(&other.value)
    }
}

impl PartialOrd<Self> for Integer {
    fn partial_cmp(&self, other: &Self) -> Option<Ordering> {
        Integer::update_comparisons();
        Some(self.value.cmp(&other.value))
    }
}
```

Listing 3: LazySelect