Randomized Algorithms: Assignment 2

Adrià Lisa <adria.lisa@estudiantat.upc.edu> <Repository>

For this assignment, we are asked to implement an adaptative variant of Quickselect called Sesquickselect and test its performance in terms of *number of scanned elements* (SE), which is said to be an indicator of cache misses [4]

Implementation in Julia

My first step was to take inspiration from basic sorting algorithms in Julia, which can be found in the Julia program directory at "/share/julia/base/sort.jl".

Julia's multiple dispatch allows different methods to be called using the same function name but with different arguments. All the sorting algorithms in the `Base` package are wrapped under the name `sort!(v,...)`, where the default is Quicksort.

I wrote my implementation in the same philosophy under the name `sesquickselect!(v, m, ...)`, where the "!" character denotes that the sorting is done in place. Moreover, all the comparisons between elements of the target array are performed with the method "`lt(o::Ordering, v[i], v[i])`" instead of "v[i] $<$ v[j]". The default ordering is `Base.Order.Forward` which corresponds to "$<$", but one could switch to "`Base.Order.Backwards`" and the algorithm would output the m-th highest element, or even use custom ordering created via the method `Base.Order.ord(...)`.

Here is an overview of the project files and their contents:

- `main.jl` Defines the module "SelectionAlgorithms" (aliased as "sa") and exports the names of all the methods of the project so they can be used in the REPL. There are also some testing scripts commented out between block comment clauses `#= =#`
- `sesquickselect.jl` Contains the implementations of the algorithm and the auxiliary functions for partitioning the array and selecting the pivots. It also has a method called `get_scanned_elements(...)` that runs several experiments on the algorithm as demanded in the assignment.
- `sesquickselect_equal_elements` Contains a preliminary implementation of the algorithm dealing with the case of arrays of equal elements.
- `sort_impl.jl` Has a copy of the default sorting methods implemented in Julia, and `quickselect!`. It is interesting to note that for less than 20 elements all sort/select methods just perform an Insertion Sort.
- `theoretical.jl` Computes the theoretical expected proportion of scanned elements (SE/n) following the math in paper [1]. Note that the code uses the same notation as the paper, like $\Delta(\nu)$, which is very convenient but the reader may not be able to visualize those characters on any code editor.
- `plots.jl` Just shows the code used to produce the plots for the assignment.

1 Experimental results

If a final Integer argument "S" is passed, `sesquickselect!(v, m, nu, S)` will return the count of SE. At each recursive call of a subsection of v with size n, S will increment

by $n-1$ if single pivot partition was performed, or by $n-2+\#\{\text{leftmost part}\}$ if dual pivot YBB partition [3] was performed.

A sequence of experiments was run exactly as suggested in the assignment, obtaining the average of SE over T different random permutations of the array $[1,\ldots,n]$ with $n = 30000$ applied at ranks $i \in I = [1, 100, \ldots, 29900, 30000]$. We plot the proportion of SE over array size n as in paper [2].

In Figures 1, 3, and 5 for different values of ν we can see the theoretical SE/n (black) vs the experimental (colour) averaged across different values for T. Note that the x-axis of the plots represents the ratio between the tested rank and the array size $\alpha = i/n$. Clearly, the experimental mean SE/n matches the theoretical functions, ever more closely as T increases. Thus, we can infer that the expected number of SE of `sesquickselect!` matches the algorithmic theoretical.

I roughly inspected average values of SE/n for several values of ν which indeed showed that the optimal value ν^* is indeed around ≈ 2.65.

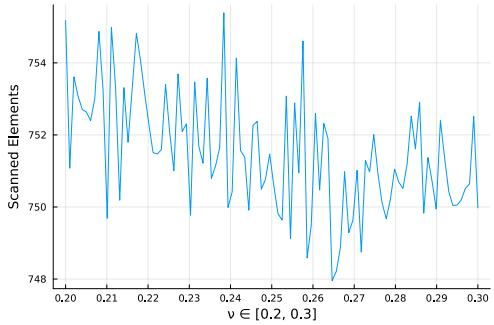

Figure 0: rough estimation of average SE across several ν values.

I also computed the empirical variance $\hat{V}_n^{(i)}$ of our experimental results and plotted a histogram of the frequency of the values $\sqrt{\hat{V}_n^{(i)}}/n$ $\forall i \in I$ in a similar fashion.

We can see in Figures 2, 4, and 6 that the relative peaks in variance are situated at the points of discontinuity of the theoretical functions, that is, at $\alpha = \nu$ and $\alpha = 1-\nu$, which is very intuitive. We also see a decreasing trend at $\alpha \in (\nu, 1-\nu)$, which can be attributed to the fact that $\sqrt{\hat{V}}$ is an unbiased estimator.

2 Future work

Beyond the scope of the assignment lie the following tasks:

- Accommodate the case where elements can be equal whilst minimizing the performance overhead.
- Generalize the algorithm so it can select a vector of several ranks, as the default Julia Quickselect method does under `sort!(..., a=PartialQuickSort, ...)`
- Benchmark the performance in execution time, number of comparisons and selected elements of Sesquickselect against Julia's PartialQuickSort.

If my implementation of Sesquickselect indeed proves to be superior to standard Quickselect, I could contribute it to the Julia language thus making the algorithm open source and available to the whole world.

References

[1] C. Martinez, M. Nebel, and S. Wild. Sesquickselect: One and a half pivots for cache-efficient selection. In *Workshop on Analytic Algorithmics and Combinatorics*, page 54. Curran, 2019.

[2] Conrado Martínez, Daniel Panario, and Alfredo Viola. Adaptive sampling strategies for quickselect. *ACM Transactions on Algorithms*, 6, 06 2010.

[3] Sebastian Wild. Dual-pivot and beyond: The potential of multiway partitioning in quicksort. *it - Information Technology*, 60(3):173–177, 2018.

[4] Sebastian Wild, Markus E. Nebel, and Conrado Martínez. Analysis of pivot sampling in dual-pivot quicksort. *CoRR*, abs/1412.0193, 2014.

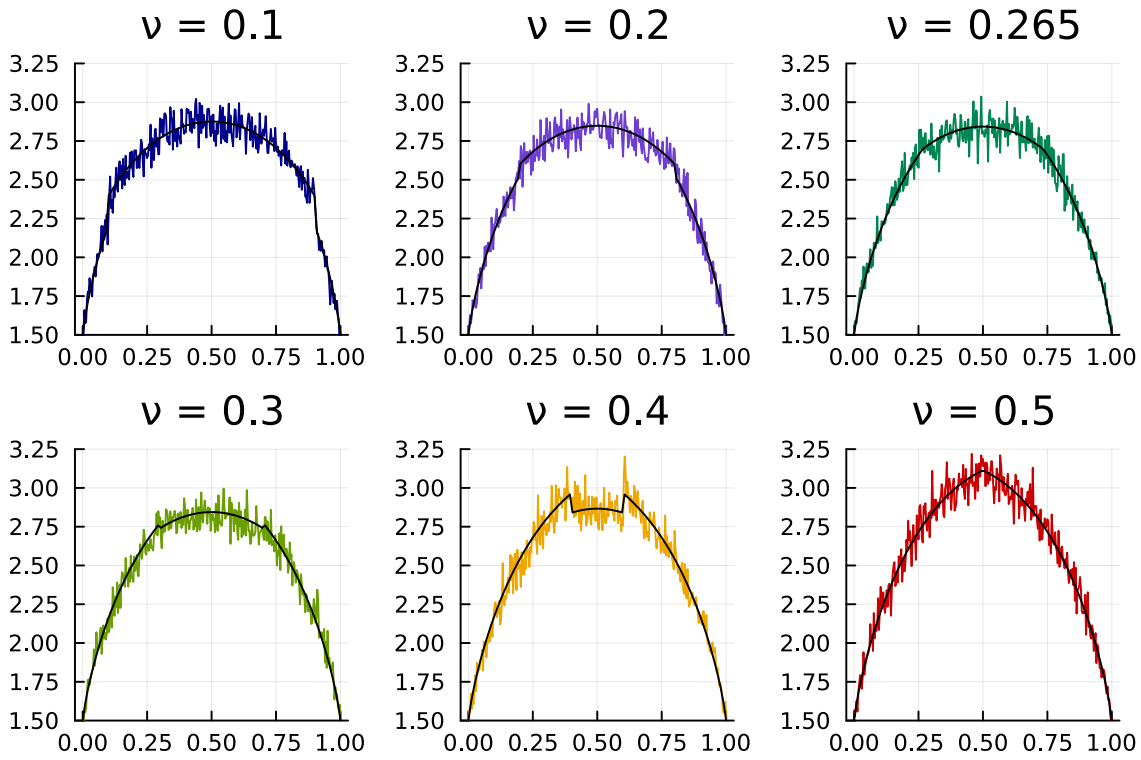

Figure 1: Results of SE/n for T=100

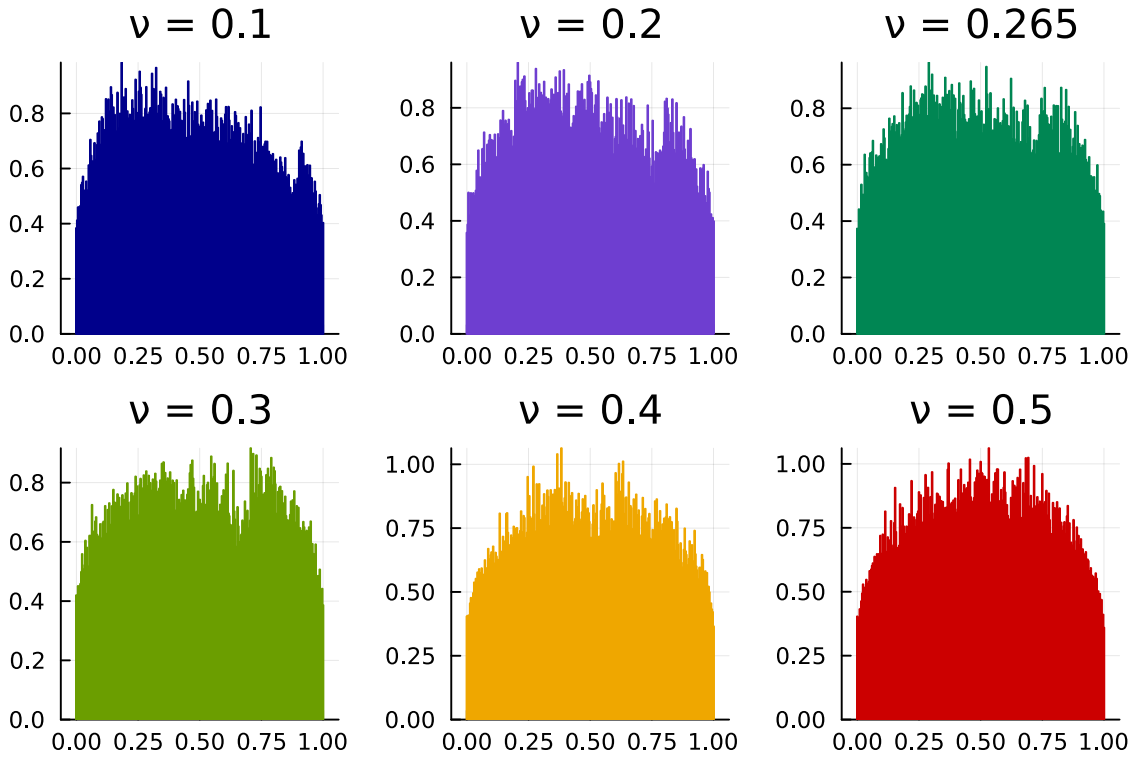

Figure 2: Results of $\sqrt{\hat{V}}/n$ for T=100

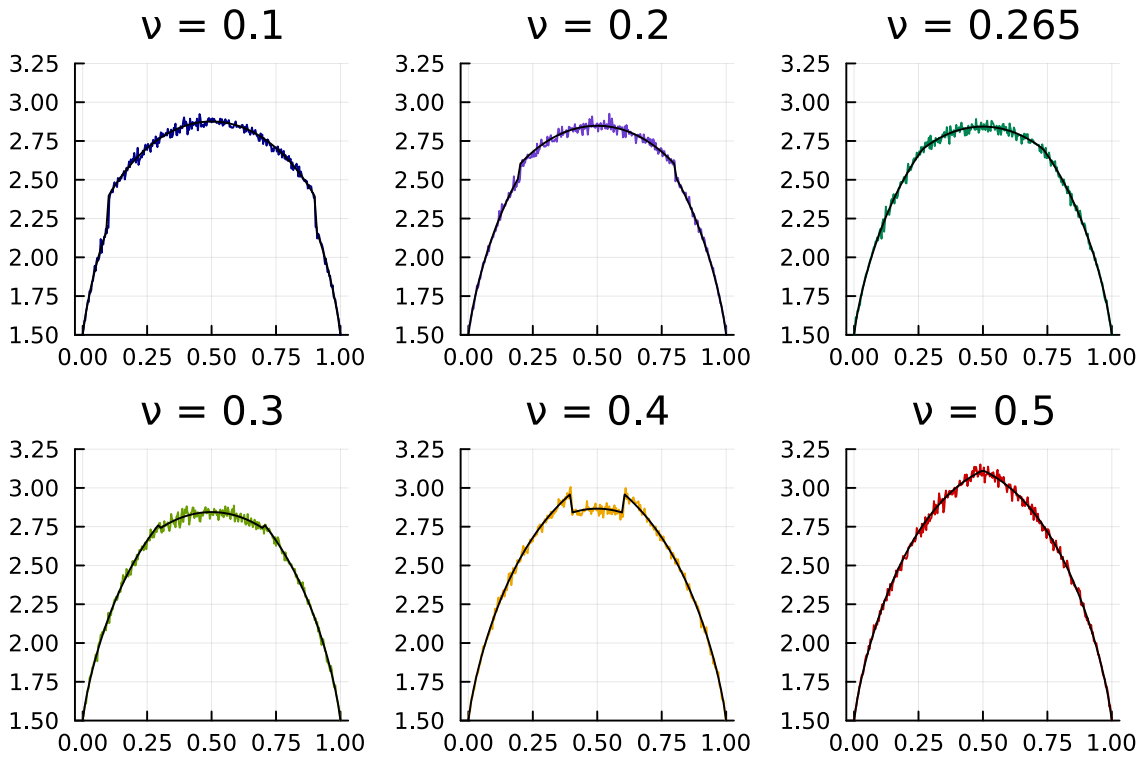

Figure 3: Results of SE/n for T=1000

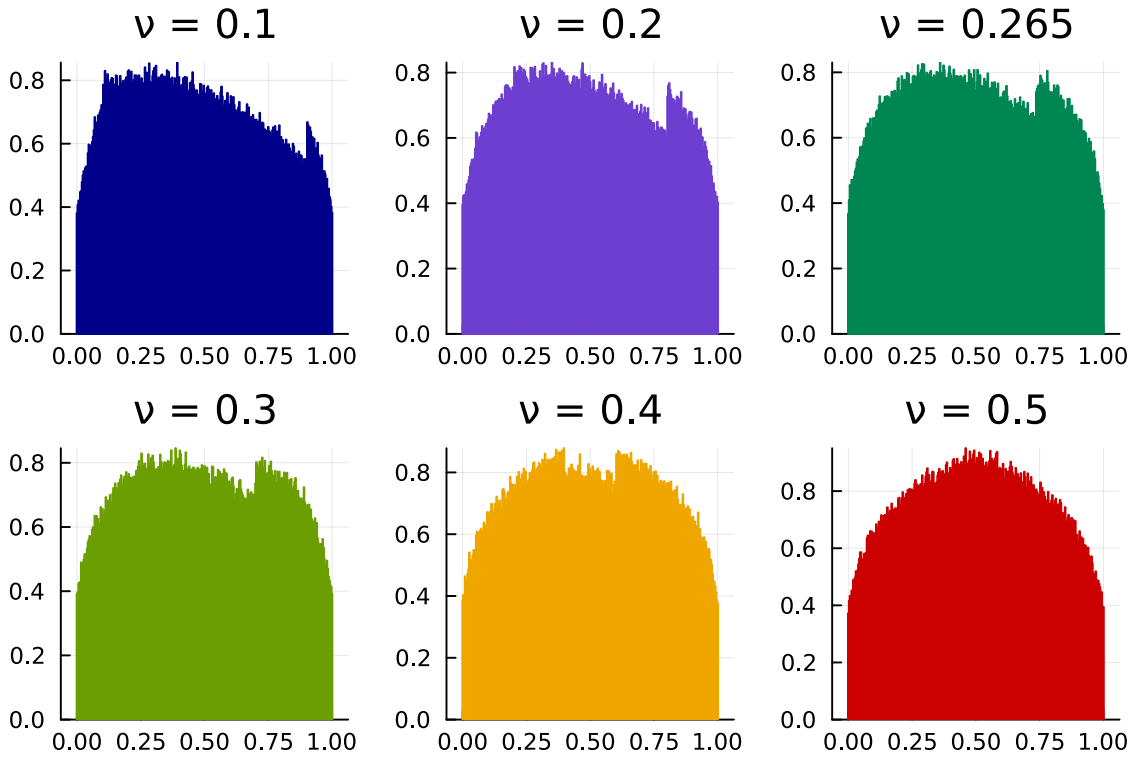

Figure 4: Results of $\sqrt{\hat{V}_n}/n$ for T=1000

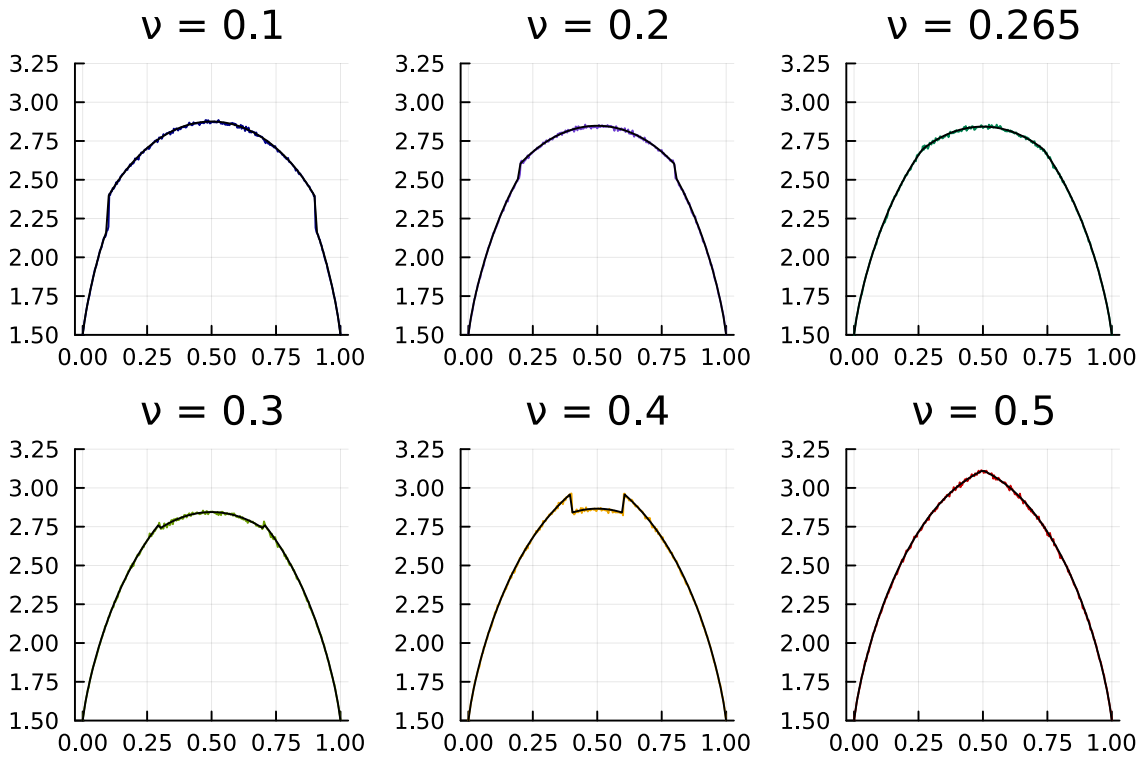

Figure 5: Results of SE/n for T=10000

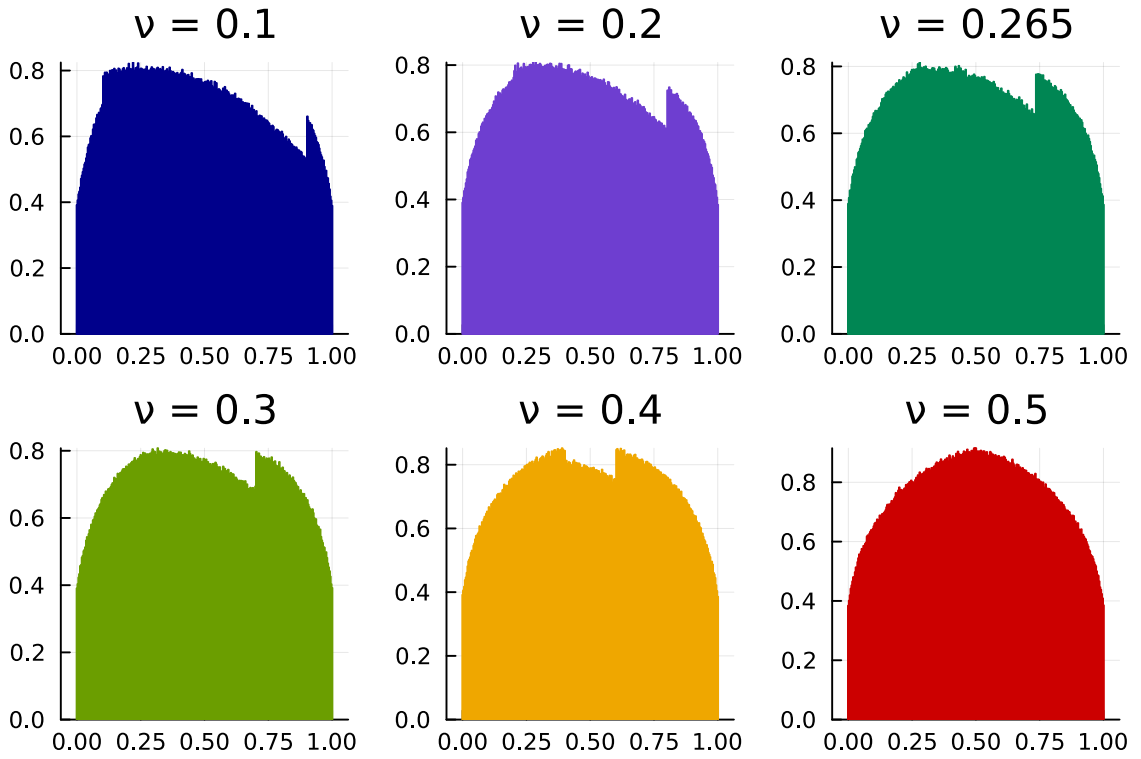

Figure 6: Results of $\sqrt{\hat{V}_n}/n$ for T=10000