



UNIVERSIDAD DE TALCA
FACULTAD DE INGENIERÍA
ESCUELA DE INGENIERÍA CIVIL EN COMPUTACIÓN

**Análisis experimental de IIQS para extender el
soporte a secuencias de elementos repetidos**

ERIK ANDRÉS REGLA TORRES

Profesor Guía: RODRIGO ANDRÉS PAREDES MORALEDA

Memoria para optar al título de
Ingeniero Civil en Computación

Curicó – Chile
mes, año



UNIVERSIDAD DE TALCA
FACULTAD DE INGENIERÍA
ESCUELA DE INGENIERÍA CIVIL EN COMPUTACIÓN

Análisis experimental de IIQS para extender el soporte a secuencias de elementos repetidos

ERIK ANDRÉS REGLA TORRES

Guide: RODRIGO ANDRÉS PAREDES MORALEDA

Profesor Informante: NN 1

Profesor Informante: NN 2

Memoria para optar al título de
Ingeniero Civil en Computación

Este documento fue evaluado con una nota de: _____

Curicó – Chile

mes, año



UNIVERSITY OF TALCA
ENGINEERING FACULTY
CIVIL COMPUTER ENGINEERING SCHOOL

**Experimental analysis of (I)IQS to fine-tune
support for arrays with repeated elements**

ERIK ANDRÉS REGLA TORRES

Supervisor: RODRIGO ANDRÉS PAREDES MORALEDA

Report to apply for a Civil Com-
puter Engineer degree

Curicó – Chile
mes, año



UNIVERSIDAD DE TALCA
FACULTAD DE INGENIERÍA
ESCUELA DE INGENIERÍA CIVIL EN COMPUTACIÓN

**Experimental analysis of (I)IQS to fine-tune
support for arrays with repeated elements**

ERIK ANDRÉS REGLA TORRES

Guide: RODRIGO ANDRÉS PAREDES MORALEDA

Advisor: NN 1

Advisor: NN 2

Report to apply for a Civil Com-
puter Engineer degree

This document was graded with a score of: _____

Curicó – Chile

mes, año

Dedicated to... someone ?

ACKNOWLEDGEMENTS

Agradecimientos a ... (how the fuck do I choose whom to acknowledge?)

CONTENT INDEX

	Page
Dedicatory	i
Acknowledgements	ii
Content Index	iii
Figure Index	vi
Table Index	viii
Summary	ix
1 Introduction	10
1.1 Context	10
1.2 Application areas	10
1.3 Problem description	10
1.4 Goals	10
1.4.1 General goals	10
1.4.2 Specific goals	10
1.5 Document Structure	11
2 Background	12
2.1 Sorting algorithms	12
2.1.1 Types of sorting algorithms	12
2.1.2 Measuring disorder	12
2.2 Incremental Sorting	17
2.2.1 Incremental QuickSort	18
2.2.2 Introspective Incremental QuickSort	19
2.3 Experimental algorithmics	21
2.3.1 Methodology foundations	21
2.3.2 Considerations	23
2.3.3 Methodology overview	23

2.3.4	Result driven development	24
2.4	Related Software and Tools	25
2.4.1	C++ Standard Template Library	25
2.4.2	C++ Standard Library	25
2.4.3	Boost.org	26
2.4.4	SciPy environment	26
2.4.5	Valgrind	26
3	Methodology	27
3.1	Rationale	27
3.2	Methodology fundamentals	28
3.2.1	Algorithm instantiation hierarchy	28
3.2.2	Algorithm design hierarchy	29
3.3	Experimental process breakdown	36
3.3.1	Experimental cycle	36
3.3.2	Metrics and indicators	37
4	Pilot experiments	42
4.1	Experimental Setup	42
4.2	Base benchmarking	42
4.2.1	Average and worst case	42
4.2.2	Influence of repeated elements	47
4.3	Partitioning Schemes	53
4.3.1	Rationale	53
4.3.2	Partition schemes	53
4.3.3	Lomuto's partition scheme	53
4.3.4	Hoare's partition scheme	54
4.3.5	Dutch flag problem	54
4.3.6	Problem definition and solution	54
4.3.7	Integration into IQS as base implementation	55
4.4	Influence of pivot selection	56
4.4.1	Controlling pivot bias	56
4.4.2	Effects on the partitioning	56
4.4.3	Suggestions	56

4.5	Median selection	56
4.5.1	Current median selection approach	56
4.5.2	Iterative median of medians	56
4.5.3	Effects on future calls	56
4.5.4	Suggestions	56
4.6	Stack structure	56
4.6.1	Stack performance	56
4.6.2	Paired stack performance	56
4.6.3	Performance comparison for repeated sequences and unique sequences	56
4.6.4	Stack management comparison for repeated sequences and unique sequences	56
5	Workhorse experiments	57
5.1	Implementation	57
5.2	Benchmarking	57
6	Summary	58
	Glossary	59
	Bibliography	60
	Annex	
A:	El Primer Anexo	63
A.1	La primera sección del primer anexo	63
A.2	La segunda sección del primer anexo	63
A.2.1	La primera subsección de la segunda sección del primer anexo	63
B:	El segundo Anexo	64
B.1	La primera sección del segundo anexo	64

FIGURE INDEX

	Page
4.1 Benchmark for random case for sequences with 1×10^4 elements. Both IQS and IIQS executions are shown on the same chart. First column represents all extractions using a linear scale. Second column depicts a logarithmic scale and shows the first 1×10^3 extractions.	43
4.2 Benchmark for an ascending sorted sequence with 1×10^4 elements. Both IQS and IIQS executions are shown on the same chart. First column represents all extractions using a linear scale. Second column depicts a logarithmic scale and shows the first 1×10^3 extractions. . .	44
4.3 Benchmark for an descending sorted sequence with 1×10^4 elements. Both IQS and IIQS executions are shown on the same chart. First column represents all extractions using a linear scale. Second column depicts a logarithmic scale and shows the first 1×10^3 extractions. . .	46
4.4 Benchmark for randomly sorted sequence with repeated elements 1×10^4 elements. IQS and IIQS executions are shown on the first and second column respectively. First row represents all extractions using a linear scale. Second row depicts a logarithmic scale and shows the first 3×10^1 extractions. Classes are uniformly distributed among the repeated portion of the array for all experiments.	48
4.5 Benchmark for randomly sorted sequence with repeated elements 1×10^4 elements. IQS and IIQS executions are shown on the first and second column respectively. All extractions are being shown using a linear scale.	49
4.6 Benchmark for randomly sorted sequence with repeated elements 1×10^4 elements. IQS and IIQS executions are shown on the first and second column respectively. All extractions using a symlog scale. . . .	50
4.7 Benchmark for randomly sorted sequence with repeated elements 1×10^4 elements. IQS and IIQS executions are shown on the first and second column respectively. All extractions using a symlog scale. . . .	51

- 4.8 Benchmark for randomly sorted sequence with repeated elements 1×10^4 elements consisting of a unique class with no random noise added. Both IQS and IIQS executions are shown on the same chart. First column represents all extractions using a linear scale. Second column depicts a logarithmic scale and shows the first 1×10^3 extractions. . . 52

TABLE INDEX

Page

SUMMARY

I'm gonna write the summary as the last part.

1. Introduction

Aquí va el texto del capítulo 1...

1.1 Context

Aquí va el texto de la primera sección del capítulo 1...

1.2 Application areas

Aquí va el texto de la primera subsección de la primera sección del capítulo 1...

1.3 Problem description

Aquí va el texto de la segunda subsección de la primera sección del capítulo 1...

1.4 Goals

Aquí va el texto de la segunda subsección de la primera sección del capítulo 1...

1.4.1 General goals

Aquí va el texto de la segunda subsección de la primera sección del capítulo 1...

1.4.2 Specific goals

Aquí va el texto de la segunda subsección de la primera sección del capítulo 1...

1.5 Document Structure

Aquí va el texto de la segunda subsección de la primera sección del capítulo 1...

2. Background

2.1 Sorting algorithms

One of the basic problems on algorithm design is the *sorting problem*, defined as for a given input sequence A of n numbers $\langle a_1, a_2, \dots, a_n \rangle$ to find a permutation $A' = \langle a'_1, a'_2, \dots, a'_n \rangle$ that yields $\forall a'_i \in A', a'_1 \leq a'_2 \leq \dots \leq a'_n$.

Sorting algorithms are commonly used as intermediate steps for other processes, making them one of the most fundamental procedures to execute on computing problems. strategies for solving this problem can vary depending on the input case constraints. For example, the number of repeated elements, their distribution, if there is some known info beforehand to accelerate the process, etc.

2.1.1 Types of sorting algorithms

The best reference on how to classify and understand which algorithm is best suitable for a given case is *A survey of adaptive sorting algorithms* by Vladimir Estivil[9] which gathers all the information at that time regarding *adaptive sorting algorithms* [14], *disorder measures* and *expected-case and worst-case* sorting.

A sorting algorithm is said to be adaptive if the time taken to solve the problem is a smooth growing function of the size and the measure of disorder of a given sequence. Note that the term array is not used on this definition as it extrapolates any generic sequence that is not bound to be contiguous.

2.1.2 Measuring disorder

The concept of disorder measure is highly relative to the problem to be solved and as expected, not all measures work for all cases. One of the most common metrics

used on partition-based algorithms is the *number of inversions* required to sort a given array. While this holds true for algorithms like *insertsort* [7] which have their running time affected by how the elements are arranged in the sequence, it's not the case of *mergesort*, which is not an adaptive algorithm given that has a stable running time regardless on how the elements are distributed. Whilst the running time is a function of the size, it does not in function of the sequence. Estivil [9] on his survey describes ten functions that can be used to measure disorder on an array when used on adaptive sorting algorithms.

Expected case and Worst-case adaptive internal sorting

One of the classification of adaptive sorting algorithms is the *Expected-case adaptive (internal¹) sorting*, on which their design is driven by that worst cases are unlikely to happen in practice so, there is no harm on using it, in contrast of the definition of *Worst-case adaptive (internal) sorting*, which assumes a pessimistic view hence the design is driven to ensure a deterministic worst case running time and asymmetric complexity.

The approach taken by such algorithms can be classified as *distributional*-in which a “natural distribution” of the sequence is expected to be solved — or *randomized* — on which their behaviours is not related on how the sequence is distributed at all. There is a huge problem when dealing with distributional approaches as they tend to be very sensible to changes on the sequence distribution, making them suitable to highly constrained problems on specific-purpose algorithms.

On the other hand, randomized approaches have the benefit of generality and being rather simple to port to other implementations due to their nature.

By example, let us consider QuickSelect algorithm used to find the element whose ranking is the $k - th$ position if the sequence A were ordered beforehand. This searching algorithm can be classified as *partition-based*, given that the process in charge of preserving the invariant is the partition stage.

As it can be seen, the behaviour of *quickselect* depends on how the element is selected in the *select* procedure. Then, we can implement two versions of *select*, namely *select_fixed* and *select_random* which yields different values in order to introduce randomization into *quickselect* (see Algorithms 2 and 3 respectively).

¹We denote all algorithms that are not intended to work on secondary memory as internal.

Algorithm 1 QuickSelect

```

1: procedure quickselect( $A, i, j, k$ )
2:    $pIdx \leftarrow select(i, j)$ 
3:    $pIdx \leftarrow partition(A, pIdx, i, j)$ 
4:   if  $pIdx = k$  then return  $A_k$ 
5:   if  $pIdx < k$  then return quickselect( $A, k, j$ )
6:   if  $pIdx > k$  then return quickselect( $A, i, k$ )

```

Algorithm 2 Fixed Selection

```

1: procedure select_fixed( $i, j$ )
2:   return  $\frac{(i+j)}{2}$ 

```

In such cases, whilst the randomized version of QuickSelect will take average time of $O(n)$ to complete the task, we can see that for the fixed pivot version, it depends on the distribution of data, which can bias the pivot result. Now, we have two versions of QuickSelect algorithm, with both distributional and randomized strategies.

Estivil [9] on his survey lists 10 different *metrics of disorder* to be used when studying adaptive sorting algorithms. For the sake understanding, we will assume that S is any sequence of numbers in any order and W_1, W_2 are instances of S defined for this example as:

$$W_1 = 6, 2, 4, 7, 3, 1, 9, 5, 10, 8$$

$$W_2 = 6, 5, 8, 7, 10, 9, 12, 11, 4, 3, 2$$

The definitions are as follows:

Maximum inversion distance (Dis)

Defined as the largest distance determined by an inversion of a pair of elements in a given sequence [8]. By example, in W_1 , 5 and 6 are the elements which require an inversion in order to be locally sorted whom are the most furthest apart in the sequence, hence $Dis(W_1) = 7$.

Algorithm 3 Random selection

```

1: procedure select_random( $i, j$ )
2:   return random_between( $i, j$ )

```

Maximum sorting distance (Max)

This metric considers that local disorder is not as important as global disorder, under the premise that when indexing objects if they are grouped in some way, then it is easy to find similar elements; on the other hand, if there is an element belonging to a group and it is on another place far away in the sequence, then it is hard to find such element. Then *Max* is defined as the largest distance that an element of the sequence needs to travel in order to be in its sorted position[8]. By example, $Max(W_1) = 5$, given that 1 require to move 5 positions to the left in order to be sorted. Alternatively, 6 requires to move positions to the right, which in terms of *Max* is equivalent.

Minimum number of exchanges (Exc)

Based on the premise that the number of performed operations is important to evaluate a sorting algorithm, a simple operation to measure is the minimum number of swaps between indices involved on a given sorting operation. Then *Exc* is defined as the minimum number of exchanges required to sort the entire sequence[12]. By example, as it is impossible to sort W_1 in fewer than 7 exchange operations, then $Exc(W_1) = 7$.

Minimum elements to be removed (Rem)

Another definition of disorder is as a phenomena produced by the incorrect insertion of elements in a sequence[10]. In this fashion, we can define as the minimum amount of elements to be removed from the sequence in order to get the longest sorted sequence available. By example, by removing 5 elements from W_1 we can obtain a sorted sequence, then $Rem(W_1) = 5$.

Minimum number of ascending portions (Runs)

Driven by the definition of partial sorting[5], any sorted subsequence of S implies that locally has a minimum amount of ascending runs as 1 to be sorted. Then another measure is the minimum number of ascending runs that can be found on any sequence, given that the elements that compose the sequence must be in the same order as found in the original sequence. In this case, W_1 has 5 ascending runs, hence

$Runs(W_1) = 5$. Knuth defined this phenomena as *step-downs* [10.5555/270146]. By definition all sequences are contiguous with each other as the shortest of a ascending portion is composed of a single element for a sequential read, in other words, $\forall a \in [0, \|W\| [: W_a < W_{a+1}$.

Minimum number of shuffled subsequences (SUS)

A generalization of *Runs* but ignoring the fact that the elements can be in the same sequential order as found in the original sequence by removing elements from it. Then *SUS* is defined as the minimum number of ascending sub-sequences in which we can partition a sequence [4]. In this case W_2 has 7 ascending runs, then, $SUS(W_2) = 5$. This metric works the same way as the previous one, but it's not needed for the elements to be contiguous, but they have to be in the same relative order as the original sequence.

Minimum number of shuffled monotone subsequences (SMS.SUS)

A generalization of *SUS* now the elements can be grouped as a subsequence as long as they are sorted in any way generating a monotone subsequence. For W_2 we can get 2 ascending shuffled sequences [4] and 1 descending shuffled sequence, hence $SMS(W_2) = 3$. In this case, instead of skipping elements, descending sequences which are not considered on the *Runs* metric are also taken into account in this one.

Sorted lists constructed by Melsort (Enc)

Skiena's Melsort [18] takes another approach at presortnedness by treating sequences as a set of enroached lists, which is similar to mergesort but chunks are generated not by the recursive call itself but rather by a series of deque operations [2]. Then the number of enroached lists generated by Melsort are a measure of disorder which Skiena denoted as *Enc*. For W_1 the number of enroached lists generated is 2, hence $Enc(W_1) = 2$.

Oscilation of elements in a sequence (Osc)

Defined by Levkopoulos as a metric of presortnedness for heapsort, *Osc* is defined for each element as the number of intersections for a given element over the cartesian

tree² of a sequence[11], motivated by the geometric interpretation of the sequence itself. In the case of W_1 as the cartesian tree manifests 5 crosses between its elements, $Osc(W_1) = 5$.

Regional insertion sort (Reg)

Based on the internal workings of *regional insertion sort* [1] which is a historical sorting algorithm. In contrast to typical sorting algorithms, historical sorting solves the problem by determining in which iteration (time) of a certain process a desired element is inserted on their corresponding index. Then *Reg* is the value of the time dimension required to sort a certain sequence.

2.2 Incremental Sorting

While sorting algorithms can be seen as a straightforward process, the definition of sorting can be extended as *partial sorting* and *incremental sorting*, as in practice, even though is used as an intermediate step of many procedures, it is not mandatory to always sort the entire array, sometimes it is needed to sort only a fragment of interest.

When partitioning a sequence using a pivot, the relationship between the pivot and the other two sub-sequences generated can be seen as a equivalence relationship [5]. Then for any sequence $A' \in A$, we can define a *partial order* if the relationship on the elements of A is reflexive, antisymmetric and transitive and then A' is called a *partially ordered sequence*.

Using this very same definition of partial order, if we retrieve the elements of a sequence and store them as A_s — a partially sorted sequence of A —, if the elements are retrieved in a way that sub-sequential pushes to the A_s is always ordered, then it is said that A is being *incrementally sorted*.

A good example of the uses of this kind of sorting are the results given by a web search engine. When a user inputs a query, regardless of the size of the database, the search engine paginates the results and presents only the first page of results. It is not actually needed to sort all the results, rather to get the most relevant,

²In computer science, a Cartesian tree is a binary tree derived from a sequence of numbers; it can be uniquely defined from the properties that it is heap-ordered and that a symmetric (in-order) traversal of the tree returns the original sequence.

then there is no need to waste time sorting all the elements for a query that can be executed only one time.

2.2.1 Incremental QuickSort

Incremental QuickSort (IQS) [16] is a variant of QuickSelect designed for usage on incremental sorting problems, intended to be a direct replacement of HeapSort on Kruskal's algorithm.

Algorithm overview

Algorithm 4 IncrementalQuickSort

```

1: procedure iqs( $A, S, k$ )
2:   if  $k \leq S.top()$  then
3:      $S.pop()$  return  $A[k]$ 
4:    $pivot \leftarrow select(k, S.top() - 1)$ 
5:    $pivot' \leftarrow partition(A, pivot, k, S.top() - 1)$ 
6:    $S.push(pivot')$ 
7:   return iqs( $A, S, k$ )

```

As it can be seen on Algorithm 4, the execution of IQS is equivalent to sorting the array by executing QuickSelect searching for the element which belongs to the corresponding position in sequential order. The advantage of using IQS for this task is that the stack stores all the previous pivots generated making next minima extraction calls cheaper than executing QuickSearch from scratch, hence the $n * \log_2(n)$ running time.

Worst case

Since this is a partition-based sorting algorithm, the performance of the sorting operation relies on the *quality* of the pivots generated along each iteration. As QuickSort, the *quality* of the a given pivot is the capability to separate the sequence into two sub-sequences of similar length, thus reducing the size of sequences being solved on next calls.

If the pivot is not able to split the sequence into similar sized sub-sequences, one side of the recursion is bound to continue with little to no reduction of the elements.

On IQS case, it executes the partition stage which has $O(n)$ complexity n times, yielding $O(n^2)$ running time.

A way to force a worst case execution is to force the pivot selection to choose each time a pivot that makes a whole partition of the array and leaves it at the end. To force this we use a sequence of elements ordered in a decreasing way and we force the pivot selection to always select the first element of the sequence.

2.2.2 Introspective Incremental QuickSort

A slightly more complex version of Incremental QuickSort were developed to avoid the worst case running time of IQS by changing the pivot selection strategy on function of how many recursive calls has executed so far [17].

The modification consists on extending the capabilities of the partition stage of IQS in order to determine if the pivot has a minimum expected quality. For such effects, the information of the relative position of the extracted pivot towards the partitioned sequence is calculated to determine if the pivot obtained can be refined or not by using another pivot selection technique.

In this case the used algorithm is the *median of medians* [3]. Median of medians algorithm guarantees that the median selected will belong to $P_{70} \cap P_{30}$ of the sequence to be partitioned, which sets the measures of quality for IIQS.

If the median returned by *select* does not belong to a position between $P_{70} \cap P_{30}$, then median of medians is executed in order to guarantee a decrease of the search space for the next call of at least 30%. The details on how this comparison is performed can be seen on Algorithm 5.

Algorithm overview

The reason behind why use median of medians is that it has $O(n)$ complexity, being the same complexity as same as the partition, thus it does not increase the asymptotic complexity of the iteration if it were to be used as a fallback to reduce the problem space.

Algorithm 5 Introspective IncrementalQuickSort

```

1: procedure IIQS( $A, S, k$ )
2:   while  $k < S.top()$  do
3:      $pid x \leftarrow random(k, S.top() - 1)$ 
4:      $pid x \leftarrow partition(A_{k, S.top()-1}, pid x)$ 
5:      $m \leftarrow S.top() - k$ 
6:      $\alpha \leftarrow 0.3$ 
7:      $r \leftarrow -1$ 
8:     if  $pid x < k + \alpha m$  then
9:        $r \leftarrow pid x$ 
10:       $pid x \leftarrow pick(A_{r+1, S.top()-1})$ 
11:       $pid x \leftarrow partition(A_{r+1, S.top()-1}, pid x)$ 
12:     else if  $pid x > S.top() - \alpha m$  then
13:        $r \leftarrow pid x$ 
14:        $pid x \leftarrow pick(A_{k, pid x})$ 
15:        $pid x \leftarrow partition(A_{k, r}, pid x)$ 
16:        $r \leftarrow -1$ 
17:     S.push( $pid x$ )
18:     if  $r > -1$  then
19:       S.push( $r$ )
20:   S.pop()
21:   return  $A_k$ 

```

2.3 Experimental algorithmics

Experimental algorithmics can be seen as a spin-off of *Design of experiments*, which is a statistic technique applied to understand how a process is affected by the relationships between its variables and external factors, by systematically studying and explaining the variation of information provided by altering the environment of the same process [19].

Mostly in algorithm design, pure mathematical and theoretical approaches are taken in order to devise how to create, develop and optimize new algorithms, but the gap between theoretical analysis and the actual implementation is still huge with the rise of new platforms and compute architectures nowadays.

In the case of computer science, computational experiments by any means are not to replace theoretical analysis but rather to complement and speed up the discovery process by guiding their research via empirical results, tuning and cyclical implementation.

The most complete recompilation of techniques and recommendations on how to apply design of experiments to algorithm design is currently made by Catherine McGeoch on her book *A guide to experimental Algorithmics* [13], on which she introduces a compressive guide on how to apply this method by introducing a fair amount of guidelines for computer scientists.

2.3.1 Methodology foundations

Algorithm instantiation hierarchy

On experimental algorithmics there is no such difference between algorithms and programs as programmers usually use, instead the differences on the abstraction level introduced as how specific is their representation on a target environment. Driven by this, we can divide our scale of instantiation as follows:

- *Metaheuristics and algorithm paradigms*, describing algorithmic structures which are not needed to be tied to a specific problem or domain at hand. By example, *MinMax* is a greedy algorithm while *2-opt* is a metaheuristic. As such, we can consider them as blueprint to solve generic problems.
- *Algorithms*, being as the description step-by-step of the process for solving a specific instance of a problem. In our case, the pseudocode used to describe

MinMax belongs to this level. Note that since at this point we have more information on what is happening, we can add more details as needed or desired.

- *Source program* as the implementation of the algorithm in a particular high-level language intended to be deployed on a given environment. At this level, the specification is given by the language standards and conventions but at source code level there is no machine-specific code, so target platform is not relevant at this point.
- *Object code* as the result of the compilation (or execution) process of the source code, being influenced by the architecture, environment and machine specs.
- *Process* is the highest level of representation as it is an active program running on a particular machine at a particular moment. At this level, any metric can be affected by both internal or external conditions such as currently running processes, shape of memory, electronics, etc.

Algorithm design hierarchy

One of the main goals of algorithm engineering is to combine results from different instantiation levels and strategies in order to overcome the roadblocks generated by the gap present between theory and experience. In order to understand this, we need to divide the algorithm design process as a linear hierarchy composed by six elements, namely:

- *System structure* is the decomposition of the software into modules that present an efficient interaction layer between themselves. At this point the developer needs to check the target runtime, sequential or parallel processing, environment support.
- *Algorithm and data structure design* specify the exact problem that is being solved on each module and decompose implementation taking it into consideration (useful for class-oriented languages). At this point the developer should take care on selecting algorithms and data structures that are asymptotically efficient and that offer an accurate problem representation.
- *Implementation and algorithm tuning* or maybe building and tuning a family of them depending on the goals of the study. Timing can be performed at

high level structures in relation to its paradigm but can also be tuned from the input or cost computation model, depending on which problem is being solved.

- *Code tuning* being performed at low-level code-specific properties, ranging from procedural calls, loops, memory management, etc. When performing code tuning, transform must be made systematically in order to get an equivalent program with better performance.
- *System software* can also affect the performance. To help alleviate this, tweaks to the runtime environment related elements like compilers can help to improve performance on certain cases.
- *Platform and hardware* being the last instance of tuning, by moving the implementation to another architecture, another CPU or adding coprocessors by example.

2.3.2 Considerations

It is important to always take into account all levels of algorithm design and instantiation hierarchies. Usually in academia, algorithm designers tend to focus only on the first two levels, completely ignoring the remaining three levels, which results on algorithms with attractive asymptotic bounds but with worse than desirable performance in practice.

But as important of taking into account of all level of design, its also important to remark that this process is not linear. It's bound that some changes can present new opportunities, roadblocks, paradigm changes and so on so forth, and when it happens it may be needed to start from scratch or to skip some steps depending on the situation.

2.3.3 Methodology overview

On experimental algorithmics the first step is to plan the involved experiments by following the following steps in a cyclic way:

Planning

- Formulate a question.

- Assemble or build the test environment.
- Experiment design to address the question.

At this stage we do not analyse any data yet as we only design the process to study at a later stage. Given that the experimental setup can alter the question at hand, this process tends to repeat until the experiment is fully assembled.

In order to determine if a given experiment is viable -namely the *workhorse experiment*, the practitioner must perform a series of *pilot experiments* beforehand, in order to understand the environment, implementation challenges and the problem itself. Pilot experiments are expected to be small experiments which answer a highly constrained and specific question that drives towards the construction of the workhorse experiment, as workhorse experiments are expected to be complex in both setup, execution and analysis.

At this step, it is expected that the practitioner develops metrics, indicators and setups which leads to a deeper understanding of the original question proposed.

Execution

Whilst this step is taken locally as a sequential process, there is a mutual dependency between the execution step and the planning stages. The execution of both pilot and workhorse experiments generates data which is used to gain information and insight on the context. But at the same time, in order to set the context, a set of previous results are needed. Because of this mutual dependency between those two processes, it is agreed that if the question at hand is answered then the process is finished. Otherwise the process starts again from the planning stage taking the results from this stage as input.

2.3.4 Result driven development

As results from experiment execution and analysis yield data from all the levels of algorithm design, those results are used to tune up existing algorithms in order to optimize their execution for certain cases, specific architectures or given constraints.

One of the key benefits of this strategy is that whilst a pure theoretical approach can be difficult or even being infeasible in some cases, a systematic experimental

approach can help to both guide theoretical analysis by giving insight and validating theoretical results.

This approach is widely used on metaheuristics tuning, as experiments are used to fill the gap by simplifying assumptions necessary to theory and the realistic use cases, characterising and improve profiling, gain insight on average and worst cases, suggest new theorems and proof strategies and to extend theoretical analyses to realistic inputs.

A example of the use of this methodology is the building of *LZ-index* [15], a compressed data structure designed to support indexing and fast lookup. Navarro used experiments to guide the choices of during the implementation process and compare the finished product against current competing strategies.

2.4 Related Software and Tools

2.4.1 C++ Standard Template Library

The C++ Standard Template Library (C++ STL from now on) is a library for the C++ language which provides the developer with a set of common use classes C++ divided into four categories: *algorithms*, *containers*, *functions* and *iterators*. These classes are made available through the use of C++ templates, which allow compile-time polymorphism which is by definition is more efficient and lightweight than run-time polymorphism. This library has the benefit that most compilers have already optimized routines to accelerate C++ STL based code during the compilation stage.

2.4.2 C++ Standard Library

The C++ Standard Library is a collection of classes, structures and functions which specifies the semantics of generic algorithms using C++ ISO standard heavily influenced by C++ STL. This library is made available as a set of headers which the developer can include into their code to make use of already optimized routines available as binaries by the underlying operating system. It also provides an abstraction layer to invoke C libraries, and it is part of the *C++ ISO Standardisation* effort.

2.4.3 Boost.org

Boost.org is an community-led organization supports research and education into the best possible uses of C++ and libraries developed for it, being Boost Library as part of the efforts to achieve a library which provides a standardized set of general purpose operations on many application fields while maintaining it free, open source and available for many platforms. Boost has 167 libraries up to the date which range from smart pointer structures, image processing operations, advanced threading management, and so on. From those libraries, 10 of them are already included on the Library Technical Report (TR1) for continuous standardisation.

2.4.4 SciPy environment

SciPy is a Python-based ecosystem of open-source software for mathematics, science, and engineering. With scientific computing as their primary field of application, it includes libraries to support the use of data frames via *pandas*, statistics operation and formulas via *scipy* , interactive notebook environments using *IPython* as its core, accelerated math operations with *numpy*, among many other tasks.

As our primary tool for organizing our experiments we will be using *Jupyter* as a IPython frontend, which allows usage of IPython commands via a browser or other IDEs using instances called *kernels*. Each kernel gets allocated their resources dynamically and it is not restricted to execute python code, as it can execute external commands, shell instructions among other dialects.

2.4.5 Valgrind

Valgrind is an instrumentation framework for building dynamic analysis tools, popular with C developers due to its direct integration with LLVM pipeline. Valgrind is typically used as a debugging tool to diagnose and detect memory problems such as memory leaks, stray associations, threading bugs and program profiling.

3. Methodology

In this chapter we will revisit the methodology explained in Section 3.2 in order to understand better how it is applied to our problem at hand.

3.1 Rationale

IncrementalQuickSelect and its introspective version, IntrospectiveQuickSelect, already have their theoretical analyses for the worst case instances. But such theoretical analysis is not always feasible, sometimes not easy and most of the time it is not realistic. As a practical example of it, when testing IQS against HeapSort for a full array sorting under architectures with small cache memory, IQS outperforms vastly HeapSort as it trashes the cache on each iteration, but when cache units are large enough to support the entire array, there is no point on using IQS, as most operations are actually solved on cache directly. There is a huge gap when it comes to practice on algorithm design, and IIQS is also not free of such problems.

The main issue arose when it comes to the analysis of the median-of-medians effects on the partition, as the execution of this algorithm offsets itself on each partition, so, it is the equivalent to run a partial sorting on fixed size contiguous segments of the array continuously. This effect displaces the elements but as it is in function of the positioning of the elements itself rather than their distribution, which makes really hard the use of standard techniques like amortized analysis to study the behaviour of IIQS. Even worse, due to the increased complexity of the algorithm, its theoretical analysis is likely to differ from the practical results.

The problem at hand is to modify the current implementation of IQS and IIQS to support an extra case, which is when the sequence of elements can have repeated

elements on it. This is now a problem for many reasons as it messes up the pivot selection heuristics and partition stages of the original algorithm.

Due to the aforementioned reasons, we want to take an experimental approach to analyse this new instance and use the results to guide the development of an extension of this algorithm.

3.2 Methodology fundamentals

3.2.1 Algorithm instantiation hierarchy

Metaheuristics and algorithm paradigms

For this experiment we will be testing IncrementalQuickSelect and IntrospectiveIncrementalQuickSelect, which are algorithms used for partial sorting, belonging to the *partition-based adaptive sorting* family. For this series of experiments, the goal is to devise if we can design and tune a version of both algorithms that it can deliver the same performance for both repeating and non repeating sequences, which is a pitfall of the original version.

Algorithms

As both IQS and IIQS are partition-based algorithms, there are many common elements and routines to it, namely:

- **Next:** The main process on both IQS and IIQS, in charge of performing a minima extraction. Its expected average running time is $O(n + \log_2 n)$ on which n is the size of the sequence being passed as input.
- **Partition:** Partitions a given sequence into three subsequences p_1 , p_2 and p_3 which follows that $\forall p_i \in p_1, \forall p_j \in p_2 : p_i < p_j$ and $\forall p_i \in p_3, \forall p_j \in p_2 : p_i < p_j$. Its expected average running time is $O(m)$ on which $m \leq n$ is the size of the sequence being passed as input.
- **Swap:** Swaps two elements in the sequence in-place. Its expected average running time is $O(1)$.
- **PushStack:** Pushes an element into the stack. Its expected average running time is $O(1)$.

- **PullStack**: Pulls an element from the stack. Its expected average running time is $O(1)$.

As for IIQS exclusive use routines we can mention:

- **BFPRT** Implementation of median of medians algorithm [3]. This algorithm is used as a fallback option for the random selection pivot selection performed during each iteration of IQS. Its expected average running time is $O(m)$ on which $m \leq n$ is the size of the sequence being passed as input.
- **Median** Sorts in-place an array of fixed size and then retrieves the element in the middle position. Its expected average running time is $O(1)$, despite the complexity of the sorting mechanism used.

Source program

As for the implementation, our language of choice was C++ in conjunction with Boost libraries for argument parsing.

Object code

Object code is obtained via direct compilation of the main source file. Due to portability concerns, this process is triggered via Makefiles but no special or separate treatment is done for the compilation artifacts.

Process

All experiments were run on userspace without any extra execution privileges.

3.2.2 Algorithm design hierarchy

System structure

Experiments are structured in a way that prioritizes execution flexibility over convention. Initial versions of the implemented algorithm were developed using C, but this language as is has some problems when it comes to understanding from the developer's standpoint. In this aspect, C++ which can be seen as a class-oriented version of C helps to perform an easier modularization of the code, while maintaining

the macro flexibility of C. Apart from the main algorithm implementations, we can identify the following structures:

Main driver

The entrypoint of our compilation unit is our daily driver, which takes charge of argument parsing, main execution and high-level operations as snapshot dumping. Runtime options are parsed with the help of Boost's `boost::program_options` library, which takes charge of parsing and converting arguments to their corresponding types.

The list of available arguments to pass to the main driver are as follows:

STD type	CLI option	Description
bool	<code>--log_pivot_time</code>	Enables clock for pivot selection runtime
bool	<code>--log_iteration_time</code>	Enables clock for iteration runtime
bool	<code>--log_extraction_time</code>	Enables clock for element extraction runtime
bool	<code>--log_swaps</code>	Enables logging of swap information
bool	<code>--use_bfprrt</code>	Enables median of medians instead of random selection (only for IIQS)
bool	<code>--use_iiqs</code>	Enables use of IIQS instead of IQS
bool	<code>--use_random_pivot</code>	Enables random pivot selection
bool	<code>--enable_reuse</code>	Enables pivot reuse
double	<code>--alpha_value</code>	Sets α value
double	<code>--beta_value</code>	Sets β value
double	<code>--pivot_bias</code>	Sets pivot selection bias
int	<code>--random_seed_value</code>	Sets random seed value
<code>std::size_t</code>	<code>--input_size</code>	Experiment input size
<code>std::size_t</code>	<code>--extractions</code>	Experiment extractions to perform
<code>std::string</code>	<code>--input_file_value</code>	Input file path
<code>std::string</code>	<code>--output_file_value</code>	Output file path

Snapshot spec

The information about the program execution is stored on *snapshots*, which are inspired on Valgrind profiler stats. Snapshot contains a rather large amount of information, from program configurations, options, and current state. In order to capture snapshots, a set of precompiler macros has been defined to enable timing of whole routines, partial sections and conditional values.

In order to minimize the performance hit, there is only one instance of `snapshot_t` being kept on memory at all times which is passed as reference to the instances of IQS and IIQS, and this snapshot is only saved on the record array log on RAM at certain keypoints. This record array log is preinitialized before the execution in order to avoid allocation calls during the execution phase. Currently the time unit used for benchmarking is `std::chrono::nanoseconds`, defined on the `TIME_UNIT` macro.

The logged metrics described on the `snapshot_t` structure are as follows:

STD type	logged variable
TIME_UNIT	iteration_time
TIME_UNIT	total_iteration_time
TIME_UNIT	partition_time
TIME_UNIT	total_partition_time
TIME_UNIT	bfprt_partition_time
TIME_UNIT	total_bfprt_partition_time
TIME_UNIT	extraction_time
TIME_UNIT	total_extraction_time
size_t	current_extraction_executed_partitions
size_t	total_executed_partitions
size_t	current_iteration_partition_swaps
size_t	total_executed_partition_swaps
double	current_iteration_longest_partition_swap
double	total_executed_longest_partition_swap
size_t	current_iteration_executed_bfprt_partitions
size_t	total_executed_bfprt_partitions
size_t	current_iteration_bfprt_partition_swaps
size_t	total_executed_bfprt_partition_swaps
double	current_iteration_longest_bfprt_partition_swap
double	total_executed_longest_bfprt_partition_swap
double	current_extracted_pivot
size_t	current_stack_size
size_t	total_pushed_pivots
size_t	total_pulled_pivots
size_t	current_iteration_pushed_pivots
size_t	current_iteration_pulled_pivots
size_t	current_extraction
size_t	input_size
char	snapshot_point

Algorithm and data structure design

Testbench implementation is divided into 4 major components:

- **IQS C++ Implementation:** Base C++ implementation of IQS with support for C++ STD container classes. One of the differences with the standard implementation of IQS is the presence of `IQS::random_between` and `IQS::biased_between` methods, which allows control over the pivot selection methods.
- **IIQS C++ Implementation:** Base C++ implementation of IIQS with support for C++ STD container classes. Inherits all components for IQS so this implementation only overloads `IQS::next` method and adds new routines such as `IIQS::bfprt` and `IIQS::bfprt` intended to support the extra operations needed by IIQS.
- **IQS low-level C++ Implementation:** C++ implementation of IQS without support for C++ STD container classes, relying only on direct memory allocation. This implementation was not benchmarked as it is only intended to be used as reference.
- **IIQS low-level C++ Implementation:** C++ implementation of IIQS without support for C++ STD container classes, relying only on direct memory allocation. This implementation was not benchmarked as it is only intended to be used as reference. All methods from low-level IQS are inherited here.

Implementation and algorithm tuning

On the original IIQS analysis, randomized sequences and sorted sequences were tested as the original problem constraints restricted the worst case to being ordered sequences on which the pivot selection fails continuously to effectively use the provided stack, hence, not reducing the problem size. But since we now are dealing with repeated elements, new sequences for input are needed to test such cases.

- **Randomized sequence:** This is our classical test case, on which all the elements are shuffled without any special criteria.
- **Ascending sequence:** Used to generate a synthetic worst-case instance for IQS, this sequence is ordered in ascending order.
- **Descending sequence:** Used to generate a synthetic worst-case instance for IQS, this sequence is ordered in descending order.

- **Constrained classes:** Given $m < n$ the number of classes on the sequence, we want to test the effect of the ratio $\frac{m}{n}$ for a fixed number classes to devise if there is a relationship between the number of classes and the running time of the algorithm. This input is shuffled after its generation.
- **Constrained classes with random noise:** In addition to the previous instance, we also add a random number of elements which do not belong to any instances of m to induce random noise on the sample. This input is shuffled after its generation.
- **Shuffled sequences with sorted segments:** Based on a mix of *Runs*, *SUS* and *SMS.SUS* metrics for adaptive sorting, this input attempts to test the effect of presortedness on the execution of IQS and IIQS. To generate this input we first generate a shuffled input and then for each subsegment of the shuffled sequence we execute a partial sorting.
- **Randomized sequence with ignored noise:** This input is intended to test if discontinuities on the sorting process can affect the performance by ignoring certain swaps. To accomplish this, we take a randomized sequence and then for a given amount of elements on the sequence we put the value that belong to their position.

Additionally, due to the nature of the problem, we have decided to constrain the following two aspects of the implementation in order to ensure performance and replication of results so they can be peer-validated at a later stage. Replication is achieved by taking a monte-carlo simulation approach using the following means:

- **Fixed seeding** For all experiments, all inputs are generated beforehand on the same instance of the machine in sequential order y providing the same seed for all random number generators.
- **Systematic randomization** For all processes that require randomization, the random values provided are extracted from a separate file beforehand, this ensures that all extractions of random numbers for the use of the algorithm are delivered in the same order across executions.

Code tuning

- **Unique snapshot intance:** In order to minimize memory consumption and allocation operations only one snapshot instance is initialized for a whole experiment, and it's passed as reference along the whole program.
- **Memory pre-allocation:** All test cases, files, snapshot space and random generated number are computed by an external process and they are fed into the program via file inputs which are read using STL `std::ifstream` and initialized before all tests start, so memory is allocated already at this point in order to prevent reallocation operations.
- **Unique source of truth:** All random numbers used along implementations are extracted from a unique source, from the same allocated space during runtime. All allocations are performed before the main execution and clocks start running in order to ensure that initialization process does not affect runtimes due to memory allocation overhead.

System software

Our compiler is GNU GCC 9.3.0 configured for a x86_64-linux-gnu target with posix thread modeling. All compiler optimizations were disabled in order to track time more accurately.

Platform and hardware

The specs of the machine used are as follows:

Item	Product ID
Processor	Ryzen 5 Series 3600, 6C/12T 3.6 GHz base processor clock 4.2 GHz max boost, 35MB cache, unlocked clock settings
Memory	2 x HyperX Fury HX426C16FB3A/16 16GB 2667MHz, dual-channel enabled
Storage	WD M.2 SSD 480GB WDS480G2G0B
Motherboard	MSI B450M PRO-VDH MAX, AM4
Power Supply Units	EVGA 600W W1, 80+ Certified
Video Adapter	Galax Video NVIDIA GeForce GTX1650 1-Click OC
Operating System	Pop!_OS 20.04 LTS
Kernel	Linux rspk-shoukugun 5.4.0-7629-generic #33158983451220.04ff6e79e-Ubuntu SMP Mon May 18 23:29:32 UTC x86_64 x86_64 x86_64 GNU/Linux
Linux Version	Linux version 5.4.0-7629-generic (buldd@lcy01-amd64-013) (gcc version 9.3.0 (Ubuntu 9.3.0-10ubuntu2)) #33158983451220.04ff6e79e-Ubuntu SMP Mon May 18 23:29:32 UTC

3.3 Experimental process breakdown

Now we will present a breakdown of the experimental process definitions needed in order to begin executing our experiments.

3.3.1 Experimental cycle

As mentioned before in 2.3.2, this process is being handled in a cyclic manner, and each experiment is controlled by their own Jupyter Notebook. Now we will describe the steps followed through the realization of the experiments of this report.

Hypothesis

We begin by raising a question and a doable answer to it which we want to prove if it holds for the current experimentation cycle. In the context of this report, hypothesis are code tuning improvements that we want to check before building a solution for our problem.

Input and execution setup

After defining our topic, we start generating inputs and planning the experiment execution. As for the inputs, we already defined in Section 3.2 how the inputs will be generated, as for each experiment inputs will be selected accordingly on what we want to test or explore.

On the other hand, as the main driver already accepts a defined set of parameters to control the execution, those are used to setup the experiment environment. From now on, we will refer to cross product of the combination of inputs and program parameters as the *search space*.

Execution

At this point we execute a GridSearch over our *search space* in sequential order - this way we reduce the amount of disturbances that the experiment can suffer along the way. After the execution of the experiments, snapshots are stored on a single ASCII file using a comma separated schema which is later used for the analysis.

Analysis

The results are gathered on Jupyter and examined in order to get insight on the phenomena and to check if the hypothesis is valid or not. After we gather enough information, a discussion on the results is held, which is written together with the results on this report.

3.3.2 Metrics and indicators

In order to evaluate the experiments we need to define beforehand some metrics to determine which aspects of the execution will become evaluated during the experiments. The difference between comparing raw data and use metrics is the pre-processing being made in order to gain useful insight about what is going on under the hood before executing our first benchmark.

Swaps

Using the definitions for measuring disorder given in Section 2.1.2, we can establish metric for complete sorting algorithms, but in our case, both IQS and IIQS falls

into the definition of incremental sorting as shown on Section 2.2 which renders such metrics unrealistic given that we want to study the behaviour of extractions and not of a complete sorting execution.

Still, such definitions can help us to establish a base to define our own disorder metric given the following known premises:

- Both IQS and IIQS rely on *partition* in order to perform partial sorting the same way as *QuickSort*.
- QuickSort being a adaptive sorting algorithm is influenced by presortedness.
- *Dis*, *Max*, *Exc*, *Rem*, *Runs*, *SUS*, and *SMS.SUS* are applicable metrics for *QuickSort*.
- Both IQS and IIQS perform their heavy lifting at the partition stage.

Then, the minimum common denominator of the aforementioned premises is the behaviour of swap operations. Given the nature of *partition* operation, it is expected to not exchange any elements (*Exc*) on a sorted sequence, and transitively, each segment of every iteration of IQS must follow the same property given that it is being performed in-place.

On the other hand, as it can be seen on the executions for worst case of IQS, as there are long ascending sub-sequences on the input (*Runs*, *SUS*, *SMS.SUS*), the fastest the partition stage ends, as it is not performing any swaps and in some cases not storing any pivots at all.

Now when a swap operation occurs at the partition stage, the elements being swapped and the pivot share a partial sorting relationship between them. Given that this relationship can occur at any point of the sequence we can state the following aspects of the swap operation:

- The longest the distance of the swap being performed from the pivot, the elements are more far away from their actual position (*Dis*, *Max*).
- A sorted sequence doesn't perform any swaps on their partition stage.
- Given that all elements are being sorted in-place, and partitions are executed in a recursive way respect the pivot position, the previous two properties are

transitive respect the partitions that are being generated during the extraction of a minima.

Then, we establish the following two disorder metrics for IQS and IIQS:

- **N_SWAPS**: As the number of swaps performed for a given iteration of IQS. This metric can be extended as cumulative regarding the extraction of a minima. The values for **N_SWAPS** are in the range $[0, n^2]$ for a given iteration.
- **MAX_SWAP**: As the longest swap performed during a iteration of IQS. This metric only applies to each individual execution of partition, as the maximum distance is bound to decrease on each iteration. The values for **MAX_SWAP** are in the range $[0, m]$, on which m is the size of the current partition which follows $m \leq n$.

Stack operations

A key aspect of IQS and IIQS is the amount of extra memory needed to perform the sorting. Under normal conditions, IQS requires $\log_2(n)$ extra space in order to maintain all pivots for the first extraction (which is the most heavy from all of them as it partition and push pivots by $\log_2(n)$ times the sequence length).

One of the problems of IQS was the fact that in certain cases, n pivots get stored on the stack, forcing for the non-introspective version to fix the stack size at n , whilst the introspective version due to its most stable behaviour can be safely set at $\log_{1.7}(n)$ thanks to the median-of-median algorithm effect.

In light of the aforementioned facts, it makes sense that any modification being made to IQS or IIQS must also compare the performance of the stack growth due to caching and memory consumption concerns. In contrast to the previous work on IIQS, this version of the analysis will also consider other extra metrics such as number of pulled elements and number of pushed elements per iteration and per minima extraction, as their behaviour is directly connected to the time used by the partitioning stage.

Number of executed subroutines

In line with the stack problem, it is sane to establish if the number of elements in the stack is in direct relation to the executions of the partition routine, and in the

same way, the executions of the partition routine has direct relationship with the number of elements extracted. In this regard, we also want to log the time that both routines get called during minima extraction to study if they had any effect on the total running time of IQS and IIQS.

Pivot bias

The preferred way of testing worst-case executions on partition-based sorting algorithms is to fix the pivot selection to a position which ensures the worst outcome each time. This is the main reason on why introduction of randomization for selection for pivots is so effective on maintaining expected average case on such algorithms. For IQS, worst case comes from choosing the lowest or the highest element on the sequence which can be accomplished by fixing the pivot position on the edges of the sequence to be partitioned, given that the entire sequence is already sorted.

But when testing synthetic test cases it is not proven if there is a position which performs better for certain cases than selecting the middle element as pivot on partition based algorithms (given the same constraints as the previous paragraph), nor if this bias for pivot selection can be used to tune the algorithm beforehand for certain cases.

Clocked routines

Not all parts of the program are subjected to monitoring via snapshots, as this approach is both nonsensical and non practical. We just change the execution of the program on certain points of the execution in order to gather metrics or to push elements to the log array.

Currently defined sections to be used as snapshot points are the following:

Program flag	code	Description
EXTRACTION_STAGE_BEGIN	10	Start of the minima extraction
EXTRACTION_STAGE_END	20	End of the minima extraction
ITERATION_STAGE_BEGIN	30	Begin of a IQS or IIQS iteration
ITERATION_STAGE_LOOP	40	Middle point of a IQS or IIQS iteration
ITERATION_STAGE_INTROSPECT	41	Introspect stage of IIQS
ITERATION_STAGE_END	50	End of IQS or IIQS iteration
PARTITION_STAGE_BEGIN	60	Start of partitioning stage
PARTITION_STAGE_END	70	End of partitioning stage

4. Pilot experiments

4.1 Experimental Setup

4.2 Base benchmarking

In experimental algorithms, the first step before engaging into a optimization job is to benchmark our current solutions in order to formulate our hypothesis and expectations. In this case we will check the behaviour of base implementations IQS and IIQS to understand better what IQS and IIQS is, what is happening when a worst case arises and to devise the next steps of this experimental development.

4.2.1 Average and worst case

We start by revisiting the results shown on [17], in order to understand when a worst case appears and what it does imply for the execution of the algorithm. In order to not dive too deep into other aspects of the experimental design, we will stick with the same metrics used in the original article.

As depicted in Figure 4.1, IIQS only shows a constant time added to IQS execution when solving a randomly ordered sequence. Despite forcing the pivot selection to the first element in the sequence, given that the sequence is randomly sorted, the first element on the sequence is randomly selected at all times. Random sequences transitively imply a random pivot selection case.

Another interesting aspect of both algorithms is that the stack size remains stable among all performed extractions. This is proof that in average, the stack does not grow asymptotically than $\log_2(n)$.

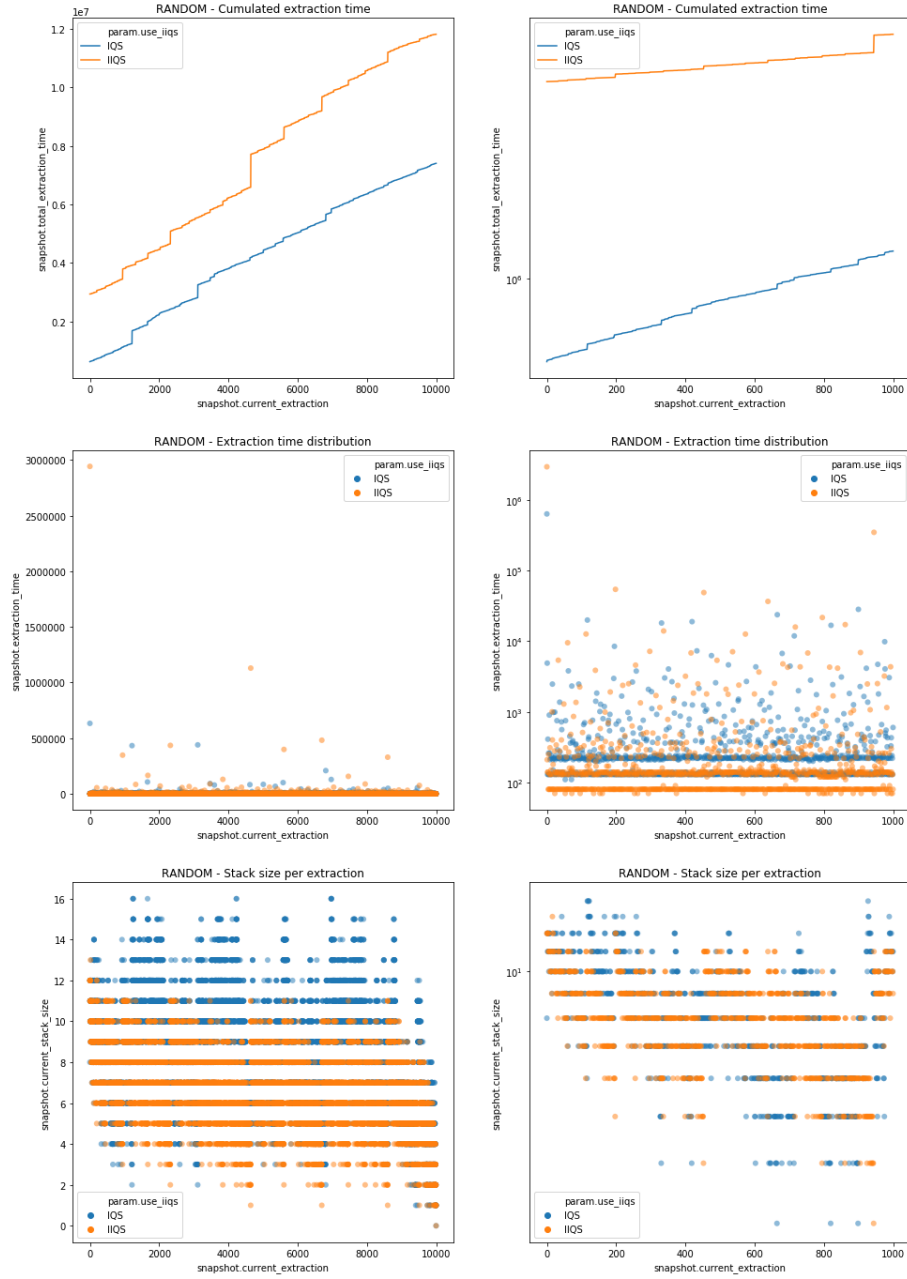


Figure 4.1: Benchmark for random case for sequences with 1×10^4 elements. Both IQS and IIQS executions are shown on the same chart. First column represents all extractions using a linear scale. Second column depicts a logarithmic scale and shows the first 1×10^3 extractions.

When fixing the pivot selection to the first element in the sequence, we are forcing a synthetic best case for the first extraction performed on IQS, as it is only required

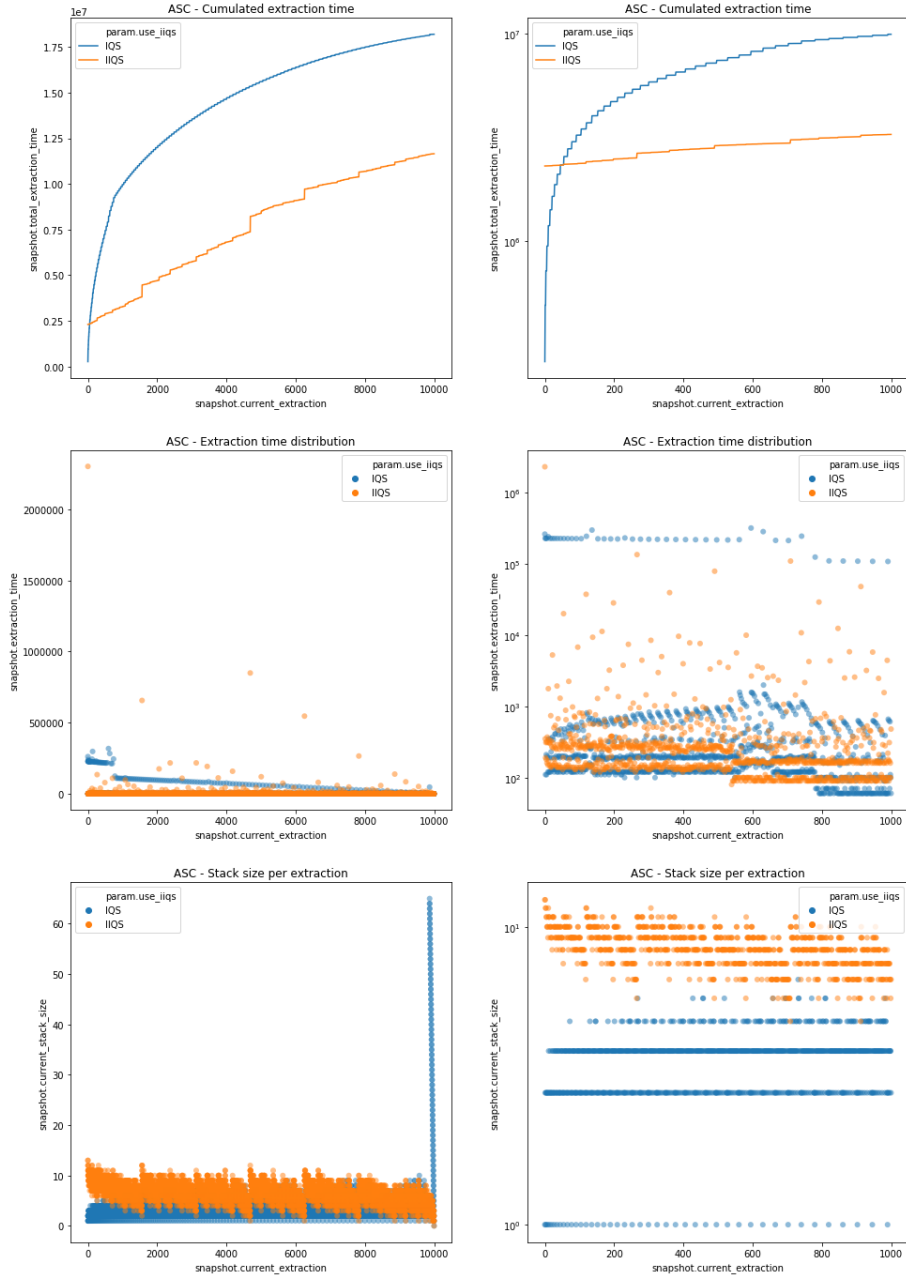


Figure 4.2: Benchmark for an ascending sorted sequence with 1×10^4 elements. Both IQS and IIQS executions are shown on the same chart. First column represents all extractions using a linear scale. Second column depicts a logarithmic scale and shows the first 1×10^3 extractions.

to run a partition which yields the element on the same position as it is being found. it is shown on Figure 4.2.

There is one noticeable difference between this experiment and the original study and is that the stack in our case grows, but barely reaching the minimum expected on the average case. This is caused by the implementation of the partition algorithm. Usually a partition algorithm uses two pivots which converge to the index of the expected pivot position, but for a repeated case scenario such implementation is not feasible, as when repeated elements are found such indices need to move segments of the sequence when displaced, and there is no guarantee on the repeated property the next element in the iteration.

The purpose of this work is to study the behaviour of our algorithms in scenarios which has a closer match to reality and under the same conditions. For such effects, a Dutch-Flag problem implementation is used as our partition algorithm. The implementation details of this algorithm can be found on Section 4.3.5.

On this implementation, since the partitioning swaps an element to the back of the to-be-partitioned segment, on the last step the last element alternates positions with the first on the partitioned sequence, as such, when the next element in the left-most position is selected, it ends being an element which belongs to the end of the array, instead of being the first on all cases.

Even on this scenario, there the stack size does not grow larger than the minimum expected size for a random case, which is an inefficient usage of the stack, causing the degradation of the algorithm. Even if some pivots are stored on them, as such pivots do not contribute to reduce the search space, there is no reduction on the execution time for subsequent calls.

The most extreme case for IQS is when the pivot selection always yields an element which does not reduce the search space –not for a useful value at least– and the stack gets populated with all the elements at the first call as it can be seen on Figure 4.3. We can achieve this case by forcing the pivot selection to the first element on a decreasing sorted sequence. Then all the extractions yield the last element in the sequence, causing it to be stored on the stack and reducing the next iteration length by just one element without swapping any element on it.

As it can be seen, for the both three cases that can be found –synthetically– on IQS, its introspective version performs with the same behaviour on all of them, but it does cost a fixed constant time more than the original version. In this context it is useful to remember that the purpose of incremental sorting algorithms is to speed up the first calls under the premise that the number of minima extracted is

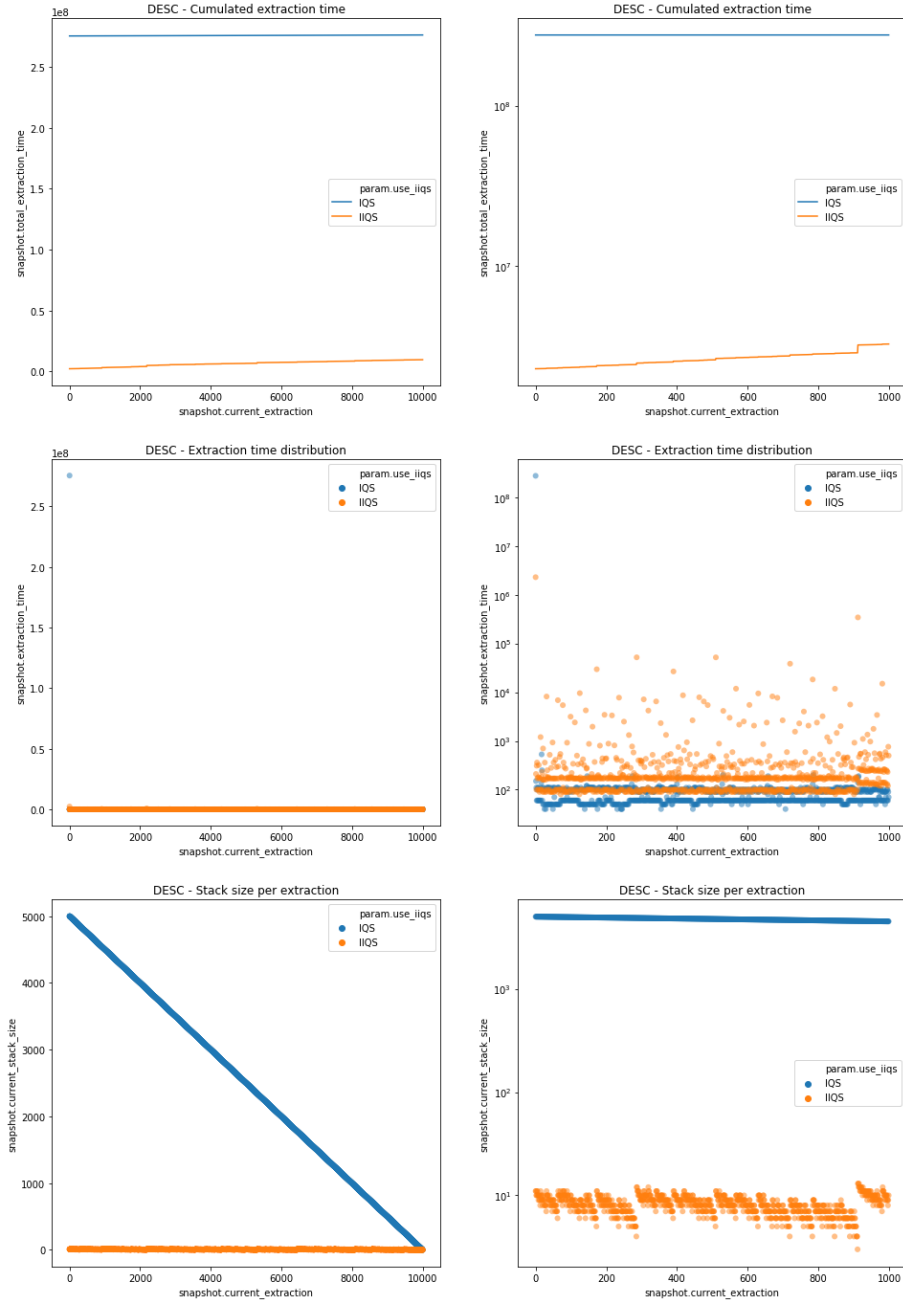


Figure 4.3: Benchmark for an descending sorted sequence with 1×10^4 elements. Both IQS and IIQS executions are shown on the same chart. First column represents all extractions using a linear scale. Second column depicts a logarithmic scale and shows the first 1×10^3 extractions.

unknown while preserving the optimal asymptotic complexity in case of sorting the

entire sequence. In this regard, IIQS manages to perform just as expected.

4.2.2 Influence of repeated elements

The question at hand now is what happens when repeated elements are found on the sequence. As mentioned before, this is the reason behind the change of the partition algorithm. In contrast to the original study, now we need to introduce three new factors to considerate:

- **Number of classes:** How many classes have the repeated portion of the sequence.
- **Amount of random noise:** How many unique elements are present on the sequence.
- **Redundant bias:** Bias of the partition algorithm to deliver the pivot in regard of the repeated sequence.

Whilst not notorious, we will from now on consider any distribution of the repeated element classes a instance of a uniformly distributed sequence. We can make this assumption based on how IQS works, as it reduces in average case by half on each iteration. Thus, for any distribution, the complexity toll is paid by the largest class found on the sequence.

At first glance there is no sign of the number of classes affecting the overall running time of the algorithm, as the Figure 4.4 exhibits a similar behaviour to the one found on both IQS and IIQS, except for a small decrease of the extraction time when less classes are found.

But it is not the case when a single repeated class is found on the sequence and examining both the noise amount and the bias over the partition algorithm result. As it can be seen on Figure 4.5, there is a strong relationship between a low random noise amount and when the partition algorithm shows a strong bias towards the right-most part of the sequence.

By furthe examining the effect of the partition bias on the Figure 4.6 we can observe that biasing the partition algorithm to return elements on the leftmost part of the repeated sequence yields higher extraction times in average than other values, and on the opposite side, biasing the algorithm to return elements towards the rightmost

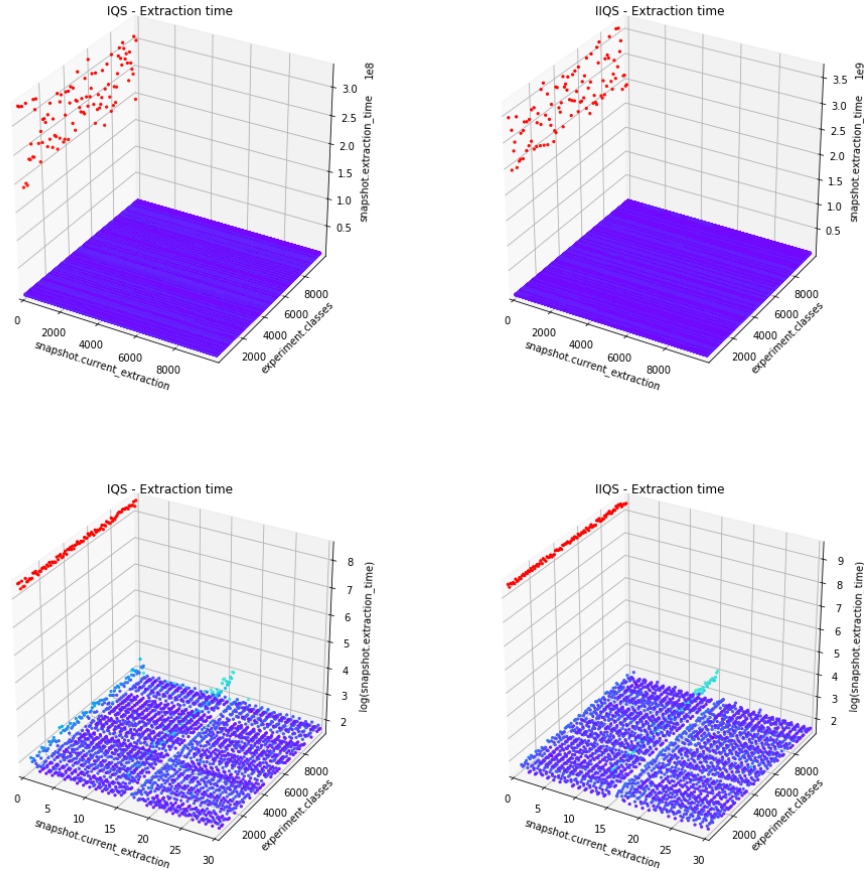


Figure 4.4: Benchmark for randomly sorted sequence with repeated elements 1×10^4 elements. IQS and IIQS executions are shown on the first and second column respectively. First row represents all extractions using a linear scale. Second row depicts a logarithmic scale and shows the first 3×10^1 extractions. Classes are uniformly distributed among the repeated portion of the array for all experiments.

part of the sequence increments the time spent on the first extraction by a many orders of magnitude, similar to the former case. On the other hand, we can see a rather stable behaviour when fixing the bias to a value in the middle.

This is not the case of the random noise in the sequence, as by itself as it does not make any visible changes on the behaviour of the algorithm in relation to the iterations on the algorithm. This is important, because the fact that a single class with low noise represents a case with the same behaviour as found on the worst case of IQS for both algorithms, as it can be checked on Figure 4.8.

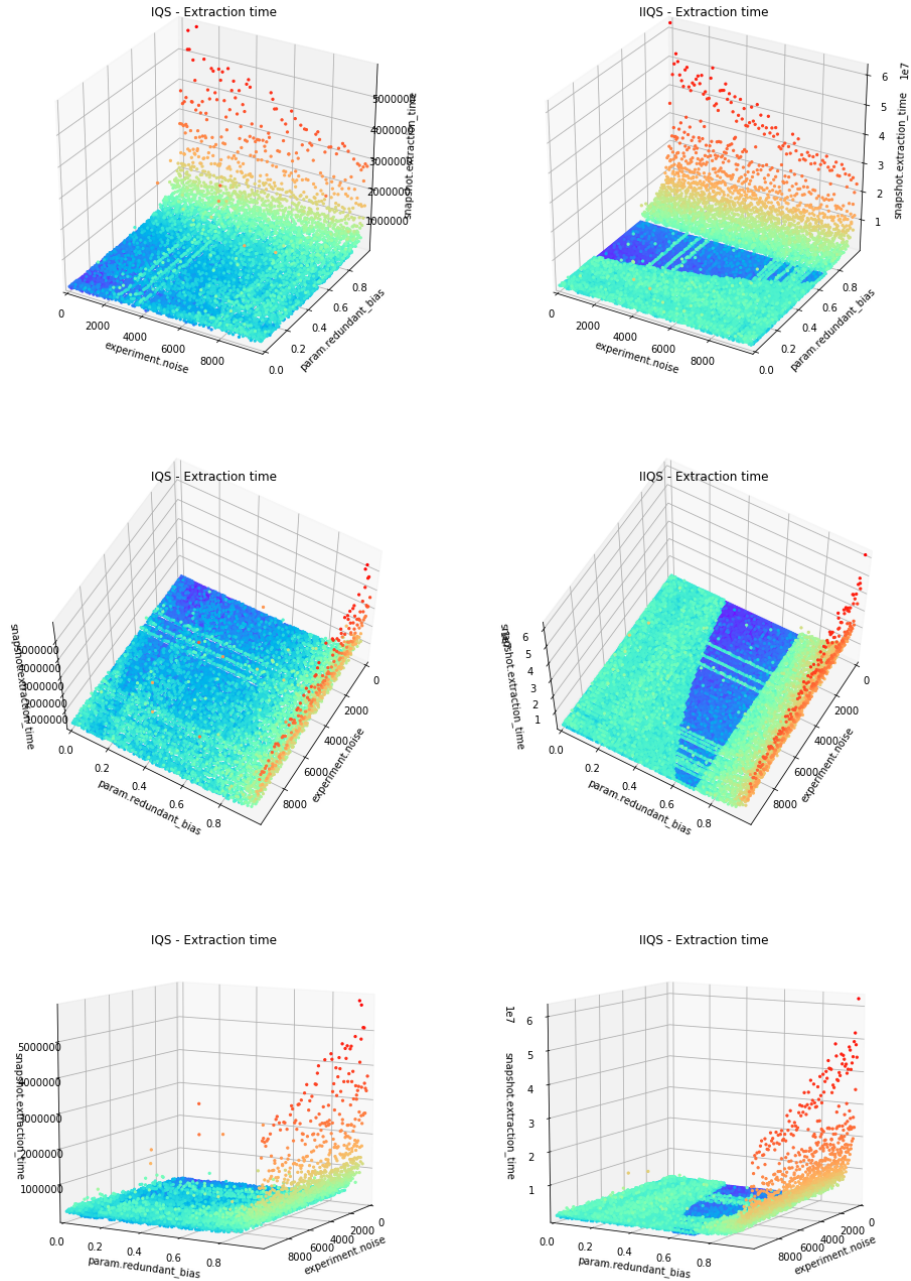


Figure 4.5: Benchmark for randomly sorted sequence with repeated elements 1×10^4 elements. IQS and IIQS executions are shown on the first and second column respectively. All extractions are being shown using a linear scale.

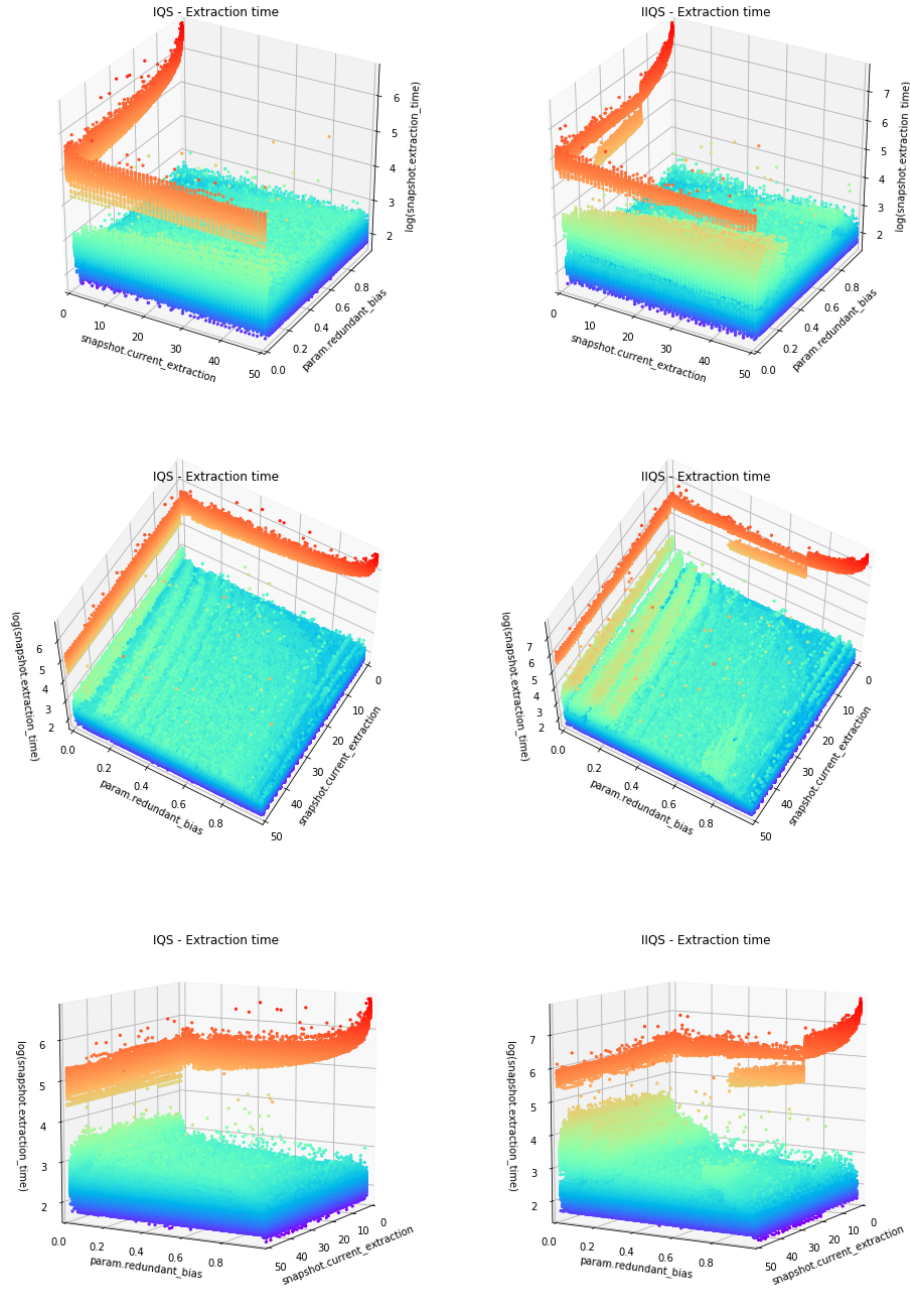


Figure 4.6: Benchmark for randomly sorted sequence with repeated elements 1×10^4 elements. IQS and IIQS executions are shown on the first and second column respectively. All extractions using a symlog scale.

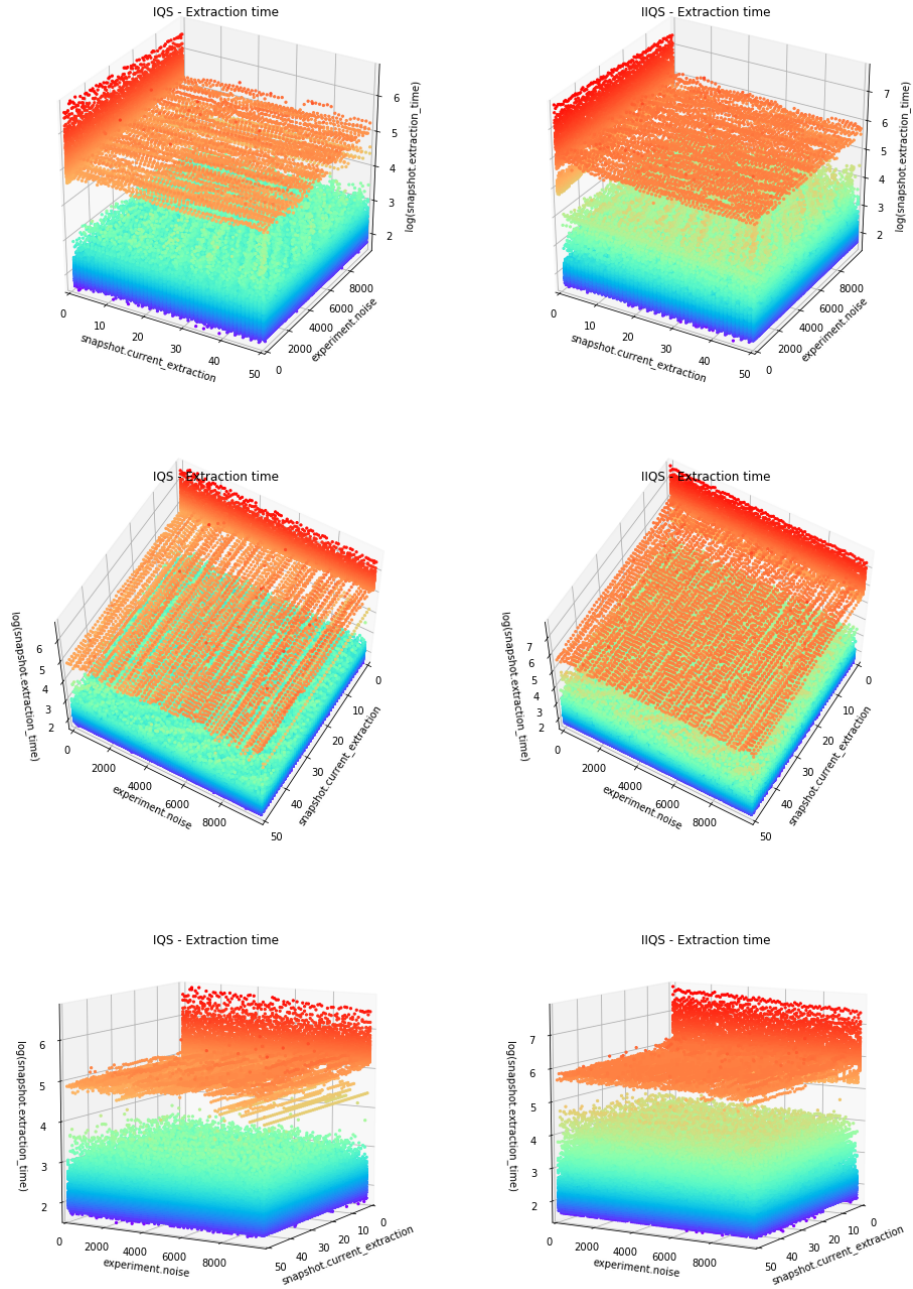


Figure 4.7: Benchmark for randomly sorted sequence with repeated elements 1×10^4 elements. IQS and IIQS executions are shown on the first and second column respectively. All extractions using a symlog scale.

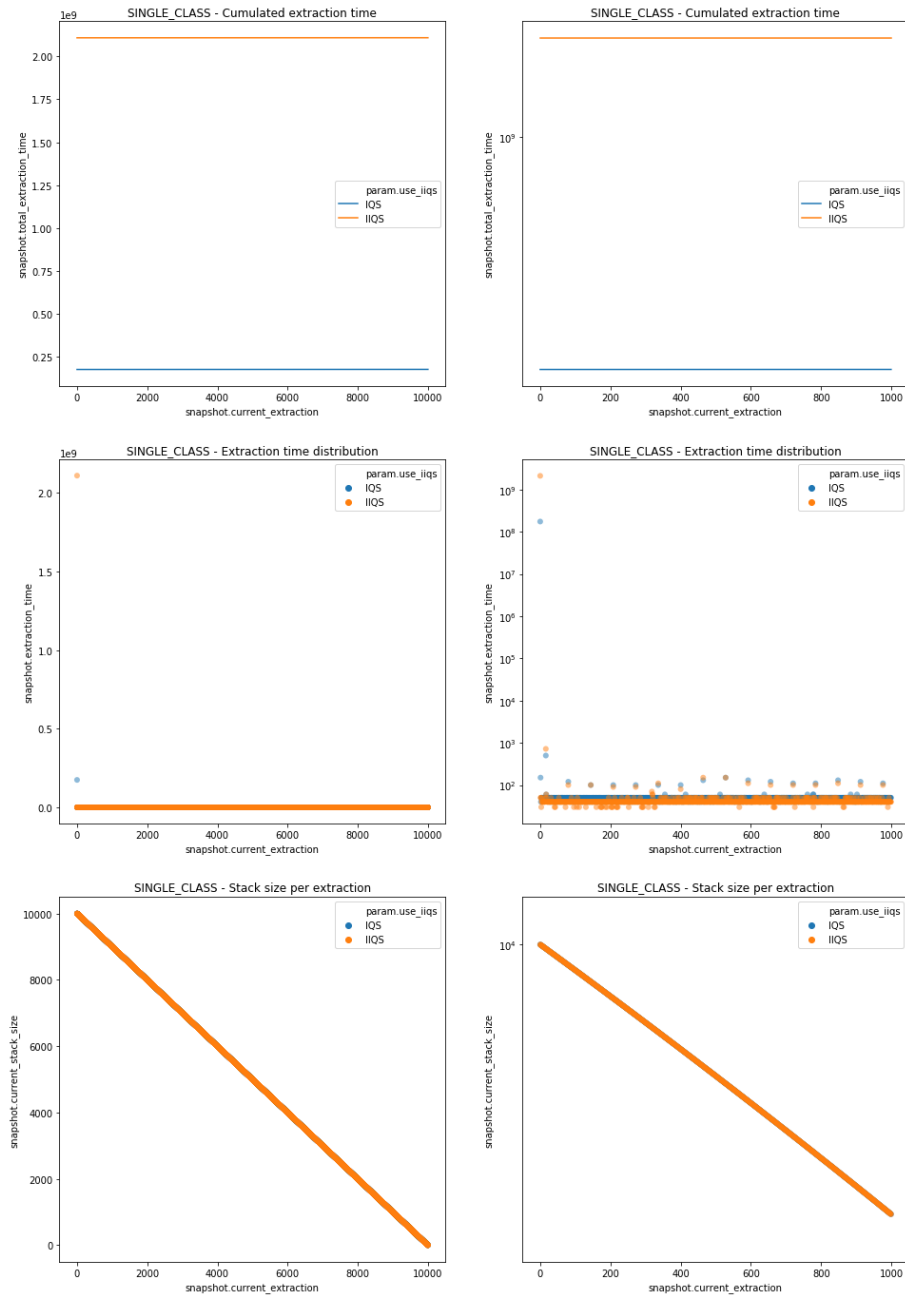


Figure 4.8: Benchmark for randomly sorted sequence with repeated elements 1×10^4 elements consisting of a unique class with no random noise added. Both IQS and IIQS executions are shown on the same chart. First column represents all extractions using a linear scale. Second column depicts a logarithmic scale and shows the first 1×10^3 extractions.

4.3 Partitioning Schemes

4.3.1 Rationale

Before diving into the results for this experiment, we will now explain two partition strategies to take into consideration before designing our first experiment and our second algorithm modification in order to understand the modifications made to IQS.

4.3.2 Partition schemes

As explained before in 2.2, partition algorithms play a fundamental role on sorting algorithms like QuickSort. But partition algorithms can use different schemes in order to partition the array into two sections, depending on which properties of the process we want to optimise.

4.3.3 Lomuto's partition scheme

The most identifiable feature of this algorithm is that it uses the last element as the pivot for partitioning the array, which makes suitable for shuffled sequences but when the sequence follows some of the first disorder metrics seen in 2.1.2 it tends to bias the performance of this partition scheme.

This algorithm is commonly referenced as the easiest way to partition an array, given it is low complexity. o

Algorithm 6 Lomuto Partition

```

1: procedure lomuto( $A, p, r$ )
2:    $x \leftarrow A_r$ 
3:    $i \leftarrow p - 1$ 
4:   for  $j \in [p, r - 1]$  do
5:     if  $A_j \leq x$  then
6:        $i \leftarrow i + 1$ 
7:       swap( $A_i, A_j$ )
8:   swap( $A_{i+1}, A_r$ )
9:   return  $i + 1$ 

```

4.3.4 Hoare's partition scheme

Hoare's partition scheme takes another approach at partitioning elements by using two indices which converge into the position of the pivot chosen at the beginning. When it comes to sort a set of elements it works faster than Lomuto's implementation and it's more stable. and given that the pivot can be chosen randomly, the introduction of randomness helps to ease biased pivot selections.

Algorithm 7 Hoare's Partition

```

1: procedure hoare( $A, p, r$ )
2:    $x \leftarrow A_p$ 
3:    $i \leftarrow p - 1$ 
4:    $j \leftarrow r + 1$ 
5:   while true do
6:     do
7:        $j \leftarrow j - 1$ 
8:     while  $A_j \leq x$ 
9:     do
10:       $i \leftarrow i + 1$ 
11:    while  $A_i \geq x$ 
12:    if  $i < j$  then
13:      swap( $A_i, A_j$ )
14:    else
15:      return  $j$ 

```

4.3.5 Dutch flag problem

Both of the aforementioned algorithms performs over sets, but when it comes to sequences, to use such partition methods fails dramatically as it treats repeated elements as unique elements. In worst case, the pivot is positioned into its corresponding place but it does not guarantee that there are no repetitions of the same element on any portion of the original sequence.

4.3.6 Problem definition and solution

Let's take as example the partitioning problem of the following two sequences:

$$S_1 = 1, 2, 3, 4, 5, 6, 7, 8, 9$$

and

$$S_2 = 1, 2, 5, 5, 5, 5, 5, 8, 9$$

It's clear that if we chose p equal to 5, the element in the fifth position of S_1 will be the pivot on its correct place. But it's not the case for S_2 as we can get a pivot from the third up to the seventh position on the sequence. In this case, as all the positions are valid pivots, there is no safety guarantee that the resulting pivot will partition the array in half in order to ensure a \log_2 decay on the problem space. Situation worsens if all the elements are repeated, as it defeats the purpose of partitioning the sequence [17].

This problem is also known as the Dutch Flag problem [6], which for given a sequence it partition in place the elements lower than the pivot value, the elements equal to the pivot value and the elements greater than the pivot value and it will return the indices of the beginning and the end of the middle portion.

Algorithm 8 Hoare's Partition

```

1: procedure hoare( $A, p, r$ )
2:    $x \leftarrow A_p$ 
3:    $i \leftarrow p - 1$ 
4:    $j \leftarrow r + 1$ 
5:   while true do
6:     do
7:        $j \leftarrow j - 1$ 
8:     while  $A_j \leq x$ 
9:     do
10:       $i \leftarrow i + 1$ 
11:    while  $A_i \geq x$ 
12:    if  $i < j$  then
13:      swap( $A_i, A_j$ )
14:    else
15:      return  $j$ 
```

4.3.7 Integration into IQS as base implementation

SECTION WITH RESULTS HERE!!!

As it can be seen, there no noticeable difference between experimental complexities of implementations for three-way partitioning and standard partitioning algorithms. As such, from now on we will use *three-way-partition* as our default

implementation for the partitioning stage. This will allow us to contrast results between IQS and IIQS with such modifications using both repeated and non repeated elements as dataset inputs.

4.4 Influence of pivot selection

4.4.1 Controlling pivot bias

4.4.2 Effects on the partitioning

4.4.3 Suggestions

4.5 Median selection

4.5.1 Current median selection approach

4.5.2 Iterative median of medians

4.5.3 Effects on future calls

4.5.4 Suggestions

4.6 Stack structure

4.6.1 Stack performance

4.6.2 Paired stack performance

4.6.3 Performance comparison for repeated sequences and unique sequences

4.6.4 Stack management comparison for repeated sequences and unique sequences

5. Workhorse experiments

5.1 Implementation

5.2 Benchmarking

6. Summary

Glossary

El primer término: Este es el significado del primer término, realmente no se bien lo que significa pero podría haberlo averiguado si hubiese tenido un poco mas de tiempo.

El segundo término: Este si se lo que significa pero me da lata escribirlo...

Bibliography

- [1] *Recent Issues in Pattern Analysis and Recognition*. Springer Berlin Heidelberg, 1989.
- [2] *Computer Science*. Springer US, 1992.
- [3] Manuel Blum, Robert W. Floyd, Vaughan Pratt, Ronald L. Rivest, and Robert E. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7(4):448–461, Aug 1973.
- [4] Svante Carlsson, Christos Levcopoulos, and Ola Petersson. Sublinear merging and natural mergesort. *Algorithmica*, 9(6):629–648, Jun 1993.
- [5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.
- [6] Edsger Wybe Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, USA, 1st edition, 1997.
- [7] Vladimir Estivill-Castro. *Sorting and Measures of Disorder*. PhD thesis, CAN, 1992.
- [8] Vladimir Estivill-Castro and Derick Wood. A new measure of presortedness. *Information and Computation*, 83(1):111–119, Oct 1989.
- [9] Vladimir Estivill-Castro and Derick Wood. A survey of adaptive sorting algorithms. *ACM Comput. Surv.*, 24(4):441–476, December 1992.
- [10] Donald E. Knuth. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., USA, 1997.

- [11] C. Levkopoulos and O. Petersson. Adaptive heapsort. *Journal of Algorithms*, 14(3):395–413, May 1993.
- [12] Heikki Mannila. Measures of presortedness and optimal sorting algorithms. *IEEE Transactions on Computers*, C-34(4):318–325, Apr 1985.
- [13] Catherine C. McGeoch. *A Guide to Experimental Algorithmics*. Cambridge University Press, USA, 1st edition, 2012.
- [14] Kurt Mehlhorn. *Data Structures and Algorithms 1*. Springer Berlin Heidelberg, 1984.
- [15] Gonzalo Navarro. Implementing the lz-index. *Journal of Experimental Algorithmics*, 13:1.2, Feb 2009.
- [16] Gonzalo Navarro and Rodrigo Paredes. On sorting, heaps, and minimum spanning trees. *Algorithmica*, 57(4):585–620, Mar 2010.
- [17] E. Regla and R. Paredes. Worst-case optimal incremental sorting. In *2015 34th International Conference of the Chilean Computer Science Society (SCCC)*, pages 1–6, 2015.
- [18] Steven S. Skiena. Encroaching lists as a measure of presortedness. *BIT*, 28(4):775–784, Dec 1988.
- [19] Jr. Wagner, John R., III Mount, Eldridge M., and Jr. Giles, Harold F. *Design of Experiments*, page 291–308. Elsevier, 2014.

ANNEX

A. El Primer Anexo

Aquí va el texto del primer anexo...

A.1 La primera sección del primer anexo

Aquí va el texto de la primera sección del primer anexo...

A.2 La segunda sección del primer anexo

Aquí va el texto de la segunda sección del primer anexo...

A.2.1 La primera subsección de la segunda sección del primer anexo

B. El segundo Anexo

Aquí va el texto del segundo anexo...

B.1 La primera sección del segundo anexo

Aquí va el texto de la primera sección del segundo anexo...