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

Extendiendo IIQS para soportar secuencias con elementos repetidos

ERIK ANDRÉS REGLA TORRES

Profesor Guia: 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

Extendiendo IIQS para soportar secuencias con 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

Extending IIQS to support sequences with repeated elements

ERIK ANDRÉS REGLA TORRES

Supervisor: RODRIGO ANDRÉS PAREDES MORALEDA

Report to apply for a Civil Computer Engineer title

Curicó – Chile
mes, año

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

Extending IIQS to support sequences with repeated elements

ERIK ANDRÉS REGLA TORRES

Guide: RODRIGO ANDRÉS PAREDES MORALEDA

Advisor: NN 1

Advisor: NN 2

Report to apply for a Civil Computer Engineer title

This document was graded with a score of: _____

Curicó – Chile

mes, año

Dedicated to whoever is reading this document.

ACKNOWLEDGEMENTS

This is probably the hardest section to write. Honestly. I'll begin stating that people here will be named in no particular order, as they are mentioned as soon as they pop into my head. There is a lot of people that I want to thank. To my parents Eraclio, Licean and my little sister Licean —yes, there is another—, for being there even if I have been a terrible son from the beginning.

To all the staff of my university's CS department —both old and new people— with a special mention to Marcela Pacheco and Ruth Garrido, which have been supporting me from the start.

I would also like to acknowledge Felipe Astroza from which I started looking at him as a rival and now we're even working together in the same place. I've always admired how he done things and he did not only taught me probably most of the quirks that I have even today when it comes to programming, but also that knowledge should be for everyone.

A shoutout to Brayan Gonzalez, my best friend which I met during first years of studying CS for all his support during this time and for always being there, even for the silliest of things. In these times of quarantine I must acknowledge that I miss you a lot, specially those drunken programming moments.

Awarded with a unique special mention is Rapa (my advisor). Both of us procrastinate a lot, we leave everything for last moment, we argue and yell each other a lot. Yet here we are. I'm thankful for the transmited experience during all this time and also for being a reliable friend. As I said before, he is the worst advisor that I could choose, yet the only one that it is up to the task.

Also I want to thank all the people from the former #weebs IRC channel, now a rather huge group of people for supporting me when I decided to go to live in Santiago and start as a VJ. A special mention to Jorge Solar and Felipe Rojas for being so encouraging when I was starting.

A special mention to Daniel Hedencrona for all his support during this time and for being a reliable friend for more than 7 years so far. He is not only one of the few guys which can understand my hobbies, but also knowledge on many topics. I mean, years ago we discussed implementations for a duck hunt AI just for fun — and because he was learning to implement bayesian trees.

Finally I would like to thank Benjamin Haller from the Messer Lab not only for being the source of inspiration for this work, but also for all his disposition during the development of the initial prototypes of bIIQS which are now installed onto SLiM.

To my students from the competitive programming workshop, for allowing me being their coach, even if we did not won any award. And also to the local ICPC staff for allowing me being part of such a wonderful event. You are so cool, really.

To the trio that teased me for years until I said that I was going to get my degree: Cristian Ruz, Carla Gallardo and Alejandro Sazo. I really miss going to eat peruvian food with you guys. Also I want to thank Martin Gutierrez and Rodolfo Allendes because if it were not for their influence during my time studying bioinformatics I would have given up on studying engineering.

Talking about bioinformatics, a big thanks and hugs to my friend Samuel Ortega, which went to US to study and still does not return. Hope he does not get any weird virus over there.

I also want to acknowledge to all the people whom I'm no longer in contact anymore. Some of them because of fights, others are no longer with us and many others because I am usually afraid of talking with other people. It's not like I have forgotten about you.

CONTENT INDEX

Page

Dedicatory **i**

Acknowledgements **ii**

Content Index **iv**

Figure Index **vii**

Table Index **ix**

Summary **x**

0.1 Context 11
0.2 Application areas 12
0.3 Problem description 12
0.4 Goals 13
0.4.1 General goals 13
0.4.2 Specific goals 13
0.5 Document Structure 13

1 Background **15**

1.1 Sorting algorithms 15
1.1.1 Types of sorting algorithms 15
1.1.2 Measuring disorder 16
1.2 Incremental Sorting 21
1.2.1 Incremental QuickSort 22
1.2.2 Introspective Incremental QuickSort 23
1.3 Experimental algorithmics 25
1.3.1 Methodology foundations 26
1.3.2 Considerations 28
1.3.3 Methodology overview 28
1.3.4 Result driven development 29
1.4 Related Software and Tools 30

1.4.1 C++ Standard Template Library 30
1.4.2 C++ Standard Library 31
1.4.3 Boost.org 31
1.4.4 SciPy 31
1.4.5 Valgrind 32

2 Methodology **33**
2.1 Rationale 33
2.2 Methodology foundational 34
2.2.1 Algorithm instantiation hierarchy 34
2.2.2 Algorithm design hierarchy 36
2.3 Experimental process breakdown 42
2.3.1 Experimental cycle 42
2.3.2 Metrics and indicators 44

3 Pilot experiments **48**
3.1 Base benchmarking 48
3.1.1 Average and worst case 48
3.1.2 Influence of repeated elements 53
3.2 Partitioning Schemes 57
3.2.1 Rationale 57
3.2.2 Partition schemes 57
3.2.3 Lomuto's partition scheme 57
3.2.4 Hoare's partition scheme 58
3.2.5 Dutch flag problem 58
3.2.6 Problem definition and solution 58
3.2.7 Integration into IQS as base implementation 60
3.3 Influence of pivot selection 64
3.3.1 Understanding pivot bias 64
3.3.2 Effects on the partitioning 65
3.4 IIQS exclusive parameters 67
3.4.1 Understanding median-of-medians usage 67
3.4.2 Relaxation of α and β constraints 70
3.4.3 Tightening α and β constraints for unique elements 70

3.4.4 Effect of repeated elements in the sequence 72
3.5 A note on median selection techniques 75

4 Workhorse experiment **76**
4.1 Repeating elements problem 76
4.1.1 Current paradigm 77
4.1.2 Paradigm shift 78
4.1.3 Counting elements to reduce complexity 78
4.1.4 Mapping 78
4.1.5 Integrating counting strategies as an external process 79
4.1.6 Integrating counting strategies as part of the partition process 80
4.2 Binned IIQS 81
4.3 Experimental evaluation 83
4.3.1 Performance comparison for unique sequences 83
4.3.2 Performance comparison for repeated sequences 83

5 Summary **91**
5.1 Future work 93

Bibliography **95**

FIGURE INDEX

Page

3.1 Base IQS and IIQS benchmark for a random sequence with 1×10^5 elements. 49
3.2 Base IQS and IIQS benchmark for an ascending sequence with 1×10^5 elements. 51
3.3 Base IQS and IIQS benchmark for a descending sequence with 1×10^5 elements. 52
3.4 Base IQS and IIQS benchmark for class impact on extraction time. . 54
3.5 Base IQS and IIQS benchmark for random noise impact on extraction time. 54
3.6 Base IQS and IIQS benchmark for a random sorted sequence with 1×10^5 repeated elements. 56
3.7 Dutch Flag partition influence on IQS and IIQS benchmark for a random sorted sequence with 100×10^3 elements. 61
3.8 Dutch Flag partition influence on IQS and IIQS benchmark for an ascending sorted sequence with 100×10^3 elements. 62
3.9 Dutch Flag partition influence on IQS and IIQS benchmark for a descending sorted sequence with 100×10^3 elements. 63
3.10 Random noise and pivot bias impact on extraction time benchmarks. 65
3.11 Pivot bias and extraction impact on extraction time benchmarks. . . 66
3.12 IIQS α and β benchmark impact on extraction time for a randomly sorted sequence. 68
3.13 IIQS α and β benchmark impact on extraction time for an ascending sorted sequence. 69
3.14 IIQS α and β benchmark impact on extraction time for a descending sorted sequence. 69
3.15 Rough IIQS α and β benchmark for a random sorted sequence with 1×10^6 repeated elements. 71
3.16 Rough IIQS α and β benchmark for an ascending sorted sequence with 1×10^6 repeated elements. 71
3.17 Rough IIQS α and β benchmark for a descending sorted sequence with 1×10^6 repeated elements. 72

3.18 IIQS variable α and β benchmark for 1×10^5 elements. 73
3.19 IIQS α and β benchmark for a sequence with repeating elements of size 1×10^6 with a unique class. 74
3.20 IIQS α and β benchmark for a sequence with repeating elements of size 1×10^6 with a unique class. Cumulative stats. 74

4.1 bIIQS benchmark for a random sequence with 100×10^3 elements. . . 84
4.2 bIIQS benchmark for an ascending sorted sequence with 100×10^3 elements. 85
4.3 bIIQS benchmark for a descending sorted sequence with 100×10^3 elements. 86
4.4 bIIQS benchmark for class impact on extraction time. 87
4.5 bIIQS benchmark for a random sorted sequence with 100×10^3 repeated elements. 88
4.6 Random noise and pivot bias benchmark for bIIQS impact on extraction time. 89
4.7 Pivot bias benchmark for bIIQS impact on extraction time. 90
4.8 Random noise impact on bIIQS extraction time. 90

TABLE INDEX

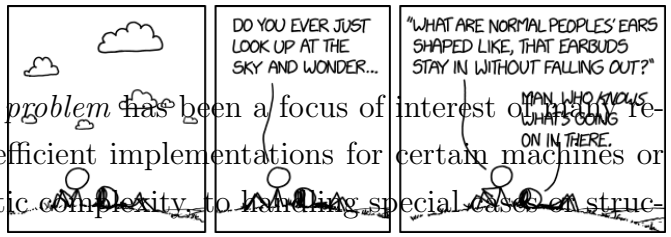
Page

2.1 Arguments for main driver program 37
2.2 Snapshot structure 39
2.3 Test machine specs 42
2.4 Timed sections 47

SUMMARY

We fixed IIQS and finished what the original work started.

0.1 Context



Since early computing, *the sorting problem* has been a focus of interest of many researchers. Challenges range from efficient implementations for certain machines or architectures, minimizing asymptotic complexity, to handling special cases of structures. Not surprisingly, most programs —if not all of them— rely on a sorting process as part of their procedures.

In algorithm design, pure theoretical analyses are the preferred way to prove both correctness and limits of algorithms. Usually these analyses are performed by using a series of mathematical techniques, which aim to provide generality of the results. As such, these analysis are usually developed to establish space and time complexities which are later used to determine best use cases and feasibility of implementation.

But in practice, when it comes to implementing these algorithms, there are many factors that can hinder the development process. The same algorithm can perform better depending on the distribution of the input, on which case this becomes the subject of our analysis. On others, computer architecture has a huge impact on the running time. Even initial values for a variable —regardless if it is being assigned during compile time or run time— can induce different results which usually lead into false and misleading outputs.

In order to close the gap generated between theory and practice, the concept of *Experimental Algoritmics* is born. Extrapolating *Design of Experiments* practices to algorithm design and engineering, this approach targets to ease the development of new programs by taking a "nuts and bolts" approach complementing theoretical analysis with practical results.

There are many algorithms developed using this process, and one of those is an incremental sorting algorithm known as *Introspective Incremental Quick Sort*. This is a extension of an incremental sorting algorithm called *Incremental Quick Sort*, intended to overcome their worst-case scenario by using the concept of *algorithm introspection* used on *IntroSort*[17].

0.2 Application areas

In contrast to the most common sorting algorithms like *QuickSort*, incremental sorting algorithms are designed to retrieve elements from a sequence in an ordered fashion, without sorting the entire array. This kind of behaviour is exploited on structures like the optimal minimum binary heaps used to implement priority queues.

Thus, as a continuous extraction of all elements via incremental sorting ends with a completely sorted dataset, incremental sorting is by extension a *resumeable* sorting.

This is useful in process which rely on heavy sorting processes that demand constant updates to the user. This is the example of *Haplotype Plots* generation in programs like *SLiM* [10], on which incremental sorting provides a way to constantly inform the user in cases on which the sorting process can take hours to complete.

0.3 Problem description

Both Incremental Quick Sort and Introspective Incremental Quick Sort as defined in their original papers [21] do not enforce any particular subroutines to be used to fill in the gaps of the procedures to follow. As such, the behaviour of both algorithms are subjected to change in function on which subroutines are used to implement it as an actual program. This leads to problems like being unable to sort sequences when repeated elements are present, due to the implementation of the partition subroutine.

The original work for Introspective Incremental QuickSort (IIQS) [21] contains a rough theoretical analysis for its worst-case performance, and a rather simple execution analysis used it to compare it with other state-of-the-art algorithms. While most of the time this information is enough to introduce an algorithm and validate their theoretical constraints, developers may struggle as they try to reproduce the reference implementation and test them in order to make use of them into actual applications[1].

[1]For Introspective Incremental Quick Sort, there is a reference implementation available on one of the author's GitHub page, but in order to comply with the conference standards at the moment of publishing, such data were not included in the paper but mentioned its presentation.

0.4 Goals

The objective of this document is to compile the efforts behind the use of an experimental algorithmics approach to develop an improved version of Introspective Incremental Quick Sort. This implementation aims to maintain its properties for the standard case of sorting sequences with no repeated elements, but it is also capable of managing repeated elements without changing its complexity.

0.4.1 General goals

The main objective in this work is to *"use an experimental algorithmics approach to extend Introspective Incremental Quick Sort's implementation to accept repeated sequences as input"*.

0.4.2 Specific goals

In order to reach our specific goal, we need to complete the following tasks:

- Research on the foundations of Incremental Quick Sort and Introspective Incremental Quick Sort.
- Plan pilot experiments to gain insight on potential improvements.
- Apply this insight to design a new version of the target algorithm.
- Empirically prove its correctness by benchmarking it.

0.5 Document Structure

The document is organized in the same way as the design process has been followed, from the initial background research, each experiment performed and final implementation.

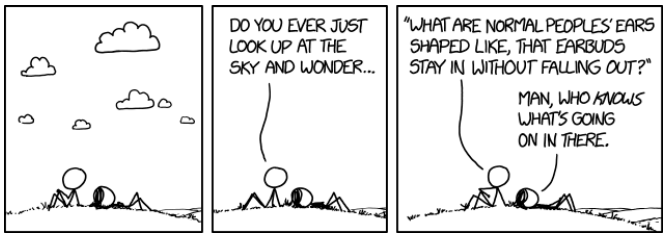
Chapter 1: This chapter is focused on explaining all relevant details in order to understand the problem at hand. It presents an introduction to our base algorithms, the methodology used, tools and techniques needed for the reader to understand the next chapters.

Chapter 2: This chapter explains how the methodology introduced on the previous section is applied to this work. Details as experimental setup and implementation concerns, measuring criteria and a breakdown of the process is presented here.

Chapter 3: This chapter presents a series of pilots experiments used to understand more the problem and to guide our algorithm engineering process. Each experiment covers their own additional background as research is part of this process.

Chapter 4: This chapter presents our algorithm proposal and the experiments used to prove its correctness.

Chapter 5: This chapter is focused on discussing the results of this process and to expose our findings.



1. Background

1.1 Sorting algorithms

One of the fundamental problems on algorithm design is the *sorting problem*, defined as for a given input sequence A of n numbers $\langle a_1, a_2, ..., a_n \rangle$ to find a permutation $A' = \langle a'_1, a'_2, ..., a'_n \rangle$ that yields $\forall\ a'_i \in A', a'_i \leq a'i + 1$, that is, $a'_1 \leq a'_2 \leq ... \leq a'_n$.

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 information known beforehand to accelerate the process, etc.

1.1.1 Types of sorting algorithms

The best reference on how to classify and understand which algorithm is the 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* [16], *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 disorder measure 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.

1.1.2 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 is 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

Now we dive on design strategies for adaptive sorting algorithms. *Expected-case adaptive (internal*[1]*) sorting* and *Worst-case adaptive (internal) sorting*. In the former, it is assumed that worst cases are unlikely to happen in practice, and the former assumes a pessimistic view and ensures a deterministic worst case running time and their asymptotic 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 behaviors are 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.

For example, let us consider QuickSelect — see Algorithm 1 — used to find the

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

element whose ranking is the $k-th$ position if the sequence A were ordered beforehand. This search strategy can be classified as *partition-based*, given that the process in charge of preserving the invariant is the partition stage and partitions the sequence in — at least — two subsequences.

Algorithm 1 QuickSelect

```
1: procedure quickselect(A, i, j, k)
2:     pIdx ← select(i, j)
3:     pIdx ← 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)
```

As it is shown, the behavior of *quickselect* depends on how the element is selected in the *select* procedure. Then, we implement two versions of *select* — our pivot selection algorithm[2] —, namely $select\_fixed$ and $select\_random$ which yields different values in order to introduce randomization into *quickselect* (see Algorithms 2 and 3 respectively).

Algorithm 2 Fixed Selection

```
1: procedure select_fixed(i, j)
2:     return (i+j)/2
```

Algorithm 3 Random selection

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

In such cases, whilst the randomized version of QuickSelect takes an 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.

[2]In literature is also known as "Pick".

Estivil [9] on his survey lists 10 different *metrics of disorder* to be used when studying adaptive sorting algorithms. For the sake understanding, we 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]. For example, in W_1, 5 and 6 are the elements which require an inversion in order to be locally sorted whom are the furthest apart in the sequence, hence $Dis(W_1) = 7$.

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]. For example, $Max(W_1) = 5$, given that 1 requires moving five positions to the left in order to be sorted. Alternatively, 6 requires moving five 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[14]. For 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 phenomenon produced by the incorrect insertion of elements in a sequence [11]. 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. For example, by removing 5 elements from W_1 we can obtain a sorted sequence, then $Rem(W_1) = 5$.[3]

Minimum number of ascending portions (Runs)

Driven by the definition of partial sorting [4], 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 these phenomena as *step-downs* [11]. By definition all sequences are contiguous with each other as the shortest of an 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 [3]. In this case W_2 has 7 ascending runs, then,

[3]While we use a sequence to establish our measure of disorder, not all elements can be actually used. In this case, as we are removing elements, there is a chance that the resulting sequence ends being smaller than the original one — as in the example —.

$SUS(W_2) = 5$.

This metric works the same way as the previous one, but it is 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[3] 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 [22] takes another approach at *presortnedness* by treating sequences as a set of encroached lists, which is similar to mergesort, but chunks are generated not by the recursive call itself but rather by a series of dequeue operations [1]. Then the number of encroached lists generated by Melsort are a measure of disorder which Skiena denoted as *Enc*. For W_1 the number of encroached lists generated is 2, hence $Enc(W1) = 2$.

Oscillation of elements in a sequence (Osc)

Defined by Levcopoulos 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 tree[4] of a sequence [13], 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,

[4]In 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.

$Osc(W_1) = 5$.

Regional insertion sort (Reg)

Based on the internal working of *regional insertion sort* [20], 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.

1.2 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 an equivalence relationship [4]. 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. Then

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

1.2.1 Incremental QuickSort

Incremental QuickSort (IQS) [19] is a variant of QuickSelect designed for using 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 ≤ S.top() then
3:         S.pop() return A[k]
4:     pivot ← select(k, S.top() − 1)
5:     pivot′ ← 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 $O(m + k\ log_2(k))$ running time on which m is the size of the sequence and $k\ log_2(k)$ is the cost of subsequent extraction. When all elements are extracted, then the complexity yields $m = k$, being comparable to the running time of QuickSort.

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 given pivot is the capability to separate the sequence into two subsequences 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 sub-sequences of similar size, 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 the 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.

1.2.2 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 [21].

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 algorithm used for the alternative pivot selection is *median of medians* [2]. Median of medians algorithm guarantees that the median selected belongs to central 40% of elements in the sequence. Let P_{30} and P_{70} the 30th and 70th percentile of an S sequence of m size. For any sorted sequence S on which graphically the leftmost element is the lowest element and the rightmost element is the greater, P_{30} is a subsequence of S which contains the $\lfloor 0.3 * |S| \rfloor$ the lowest elements in S, and $P_{70} \setminus P_{30}$ is a subsequence which contains the $\lceil 0.3 * |S| \rceil$ the greatest elements of S.

If the median returned by *select* does not belong to a position between $P_{70} \setminus P_{30}$, then median of medians is executed in order to guarantee a decrease of the search space for the next call of at least 30%. As the pivot index can be mapped to their relative position on the sequence S, then $\alpha = 0.3$ and $\beta = 0.7$ denotes the values used to compare the pivot position. The details on how this comparison is performed can be seen on Algorithm 5.

Algorithm overview

Algorithm 5 Introspective IncrementalQuickSort

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

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.

1.3 Experimental algorithmics

Experimental algorithmics (EA) can be seen as a spin-off[5] of *Design of Experiments* (DoE). DoE are a collection of statistic techniques 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 [23].

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, tuning and cyclical implementation via empirical results.

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* [15], on which she introduces a compressive guide on how to apply this method by introducing a fair amount of guidelines for computer scientists.

[5]We use the term "spin-off" as we are emphasizing the fact that it is a DoE process directly applied to a field which relies heavily on exact mathematical analysis to give results.

1.3.1 Methodology foundations

Algorithm instantiation hierarchy

On experimental algorithmics there is no such difference between algorithms and programs as programmers usually use[6]. 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. For example, *Dijkstra's algorithm* [4] is a *dynamic programming* algorithm while *2-opt* is a *metaheuristic* [5]. As such, we can consider them as blueprints 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 the 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.

[6]For developers, algorithms are usually referred as an abstract blueprint for describing a process while the program is the executable instance usually without any middle ground in it. Concepts like processes or the intermediate products of their implementation tend to be ignored as they are part of the previous aforementioned definitions.

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 and environment support.
- *Algorithm and data structure design* specify the exact problem that is being solved on each module and decomposes its implementation (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, transformations to the base code 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.

1.3.2 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 ones, which could lead 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, it's also important to remark that this process is not linear. It is 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.

1.3.3 Methodology overview

On experimental algorithmics the first step is to plan the involved experiments according to 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 analyze 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, a *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 configurations 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.

1.3.4 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 there is no guarantee of returning optimal solutions or even worse, converging into one. Experiments are used to fill the gap by simplifying assumptions necessary to theory and the realistic use cases in a *nuts-and-bolts* fashion[7].

Characterizing and improve profiling, gain insight on average and worst cases, suggest new theorems and proof strategies and to extend theoretical analyses to realistic inputs becomes part of this cycle driven by a necessity of replicate the effects at a later time.

An example of the use of this methodology is the building of *LZ-index* [18], 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.

1.4 Related Software and Tools

1.4.1 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 frequent use classes, 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 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.

[7]By randomly turning dials and pressing switches in a machine until something happens

1.4.2 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 Standardization* effort.

1.4.3 Boost.org

Boost Library is a free, open source and platform-wide available set of general purpose operations implemented on C++ for many application fields. It has 167 libraries up to the date (year 2020) which range from smart pointer structures, image processing operations, advanced threading management, and so on.

From those libraries, ten of them are already included on the Library Technical Report (TR1) for continuous standardization. It is developed by Boost.org, which is a community-led organization supports research and education into the best possible uses of C++ and libraries.

1.4.4 SciPy

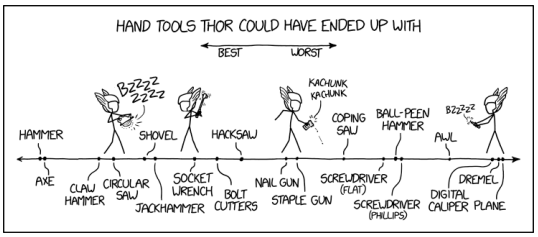
SciPy is a Python-based ecosystem of open-source software for mathematics, science, and engineering. With scientific computing as their primary field of application, it includes libraries to support the use of data frames via *pandas*, statistics operation and formulas via *SciPy*, interactive notebook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

We use *Jupyter* as out primary tool for organizing our experiments, which allows usage of IPython commands via a browser or other IDEs using instances called *ker-*

nels. 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.

1.4.5 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.



2. Methodology

In this chapter we revisit the methodology explained in Section 1.3 in order to understand better how it is applied to our problem at hand.

2.1 Rationale

IncrementalQuickSort and its introspective version, IntrospectiveQuickSort, 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, it is the equivalent to run continuously a process which reduces the overall disorder of the sequence[1].

This effect displaces the elements on the sequence towards their expected position on it due to the adaptive-sorting nature of both IQS and IIQS makes the overall

[1]We do not talk explicitly of any disorder metric seen in Section 1.1.2 as the effect depends on the process behind each execution. In this sense, we want our definition to be abstract and not to be tied with any algorithm implementation.

running time dependent on the element distribution. This makes really hard the use of standard techniques like amortized analysis to study the behavior 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 analyze this new instance and use the results to guide the development of an extension of this algorithm.

2.2 Methodology foundational

2.2.1 Algorithm instantiation hierarchy

The main goal of this work is to devise if we can design version of both IQS and IIQS which can avoid the worst case when dealing with sequences that hold repeated elements. Once this goal is accomplished, then we need to study its behavior in order to check ways to deliver the same performance for both repeating and non repeating sequences. Thus, our instantiation hierarchy is as follows:

Metaheuristics and algorithm paradigms

IncrementalQuickSort and IntrospectiveIncrementalQuickSort are algorithms used for partial sorting, belonging to the *partition-based adaptive sorting* family which are our optimization target using repeated elements on a sequence.

Algorithms

As both IQS and IIQS are partition-based algorithms, they both share common elements and routines, namely:

- **Next**:This is the main process on both IQS and IIQS. It performs a minima extraction. It 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 $\forall\ p_i \in p_3,\ \forall\ p_j \in p_2 :\ p_i < p_j$. It expected average running time is $O(m)$ on which m is the size of the sequence being passed as input following $m \leq n$ on which n is the total sequence length.
- **Swap**: Swaps two elements in the sequence in-place. It expected average running time is $O(1)$.
- **PushStack**: Pushes an element into the stack. It expected average running time is $O(1)$.
- **PullStack**: Pulls an element from the stack. It expected average running time is $O(1)$.

As for IIQS exclusive use routines we can mention:

- **BFPRT:** Implementation of median of medians algorithm [2]. This algorithm is used as a fallback option for the random selection pivot selection performed during each iteration of IQS. It 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[2] and then retrieves the element in the middle position. It expected average running time is $O(1)$, despite the complexity of the sorting mechanism used as the time used is constant and is not in function of the sequence length.

[2]Whilst on literature a median of medians of five elements is suggested, we do not want to tie the implementation of the algorithm to a fixed value, but rather become this size a parameter of our algorithm.

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 are executed on userspace without any extra execution privileges.

2.2.2 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 having 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 entry point 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 shown in the table shown in Table 2.1:

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

Table 2.1: Arguments for main driver program

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 precompiled 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. This snapshot is only saved on the record array log on RAM at certain key points. This record array log is initialized 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 table on Table 2.2 shows the logged metrics described on the `snapshot_t` structure:

Algorithm and data structure design

Test bench implementation is divided into four 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 `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.

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

Table 2.2: Snapshot structure

Implementation and algorithm tuning

On the original IIQS analysis, randomized sequences and sorted sequences were used as tests. The original problem constrained all worst case input instances to be ordered sequences in order to ease understanding. On ordered sequences is easier to see when a pivot selection fails by misusing the stack, thus 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 sequences**: This is our classical test case, on which all the elements are shuffled without any special criteria.
- **Ascending sequences**: Used to generate a synthetic worst-case instance for IQS, this sequence is ordered in ascending order.
- **Descending sequences**: 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 [4] 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 and 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 instance**: In order to minimize memory consumption and allocation operations only one snapshot instance is initialized for a whole experiment, and it is 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 are disabled in order to track time more accurately.

Platform and hardware

The specs of the machine used are shown in the table on Table 2.3:

Item	Product ID
Processor	Ryzen 5 Series 3600, 6C/12T 3.6 GHz base processor clock 4.2 GHz max boost, 35 MB cache, unlocked clock settings
Memory	Team T-FORCE DARK Za 32 GB (2 x 16 GB) DDR4 3600 MHz (PC4 28800) TDZAD432G3600HC18JDC01, dual-channel enabled, XMP profile 1 enabled
Storage	WD M.2 SSD 480 GB WDS480G2G0B
Motherboard	MSI B450M PRO-VDH MAX, AM4
Power Suppy 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 #33~1589834512~20.04~ff6e79e-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 (buildd@lcy01-amd64-013) (gcc version 9.3.0 (Ubuntu 9.3.0-10ubuntu2)) #33~1589834512~20.04~ff6e79e-Ubuntu SMP Mon May 18 23:29:32 UTC

Table 2.3: Test machine specs

2.3 Experimental process breakdown

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

2.3.1 Experimental cycle

As mentioned before in Section 1.3.2, this process is being handled in a cyclic manner, and each experiment is controlled by their own Jupyter Notebook. Now we 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 2.2 how the inputs are being generated, as for each experiment inputs are to 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 set up the experiment environment. From now on, we refer to the cross product of the combination of inputs and program parameters as the *search space*.

Execution

At this point we execute a GridSearch over out *search space* in sequential order. This way we reduce the amount of disturbances that the experiment can suffer. 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.

2.3.2 Metrics and indicators

In order to evaluate the experiments, we need to define beforehand some metrics to determine which aspects of the execution are 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 1.1.2, we can establish metric for complete sorting algorithms. Both IQS and IIQS falls into the definition of incremental sorting as shown on Section 1.2, making its use natural to us. But as our study consists on analyzing the behavior of extractions and not of a complete sorting execution, such metrics are unrealistic to such purposes and unnatural.

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 the partial sorting the same way as *QuickSort*.
- QuickSort, being an 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 behavior 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 does not 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 in function of the partitions that are being generated during the extraction of a minimum.

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 minimum. The values for **N_SWAPS** are in the range $[0, n^2]$ for a given iteration.
- **MAX_SWAP**: As the longest swap performed during an 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. This makes the first extraction the most time-consuming operation as it partitions over n elements and pushes $log_2(n)$ pivots into the stack in average.

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 behavior 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 also considers other extra metrics such as number of pulled elements and number of pushed elements per iteration and per minima extraction, as their behavior 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, the 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, 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.

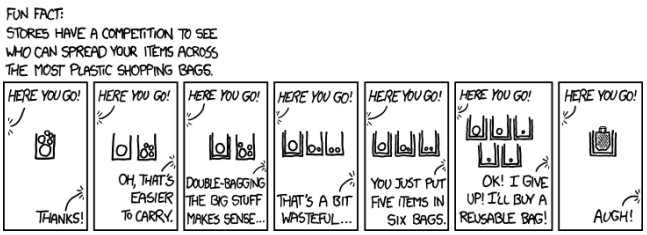
Clocked routines

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 shown in the table on Table 2.4:

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

Table 2.4: Timed sections



3. Pilot experiments

3.1 Base benchmarking

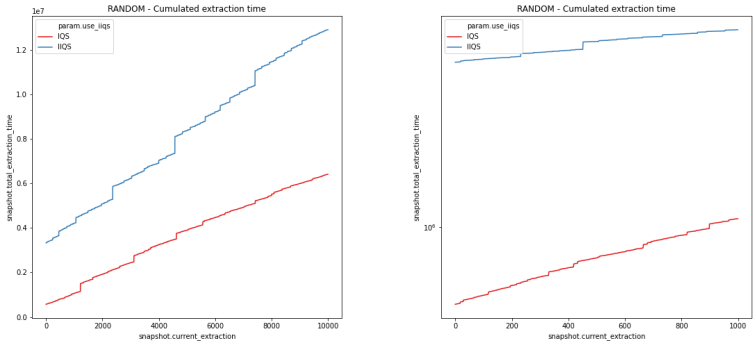
In experimental algorithms, the first step before engaging into an optimization job is to benchmark our current solutions in order to formulate our hypothesis and expectations. In this case we check the behavior of base implementations IQS and IIQS to understand better what IQS and IIQS is, what is happening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

3.1.1 Average and worst case

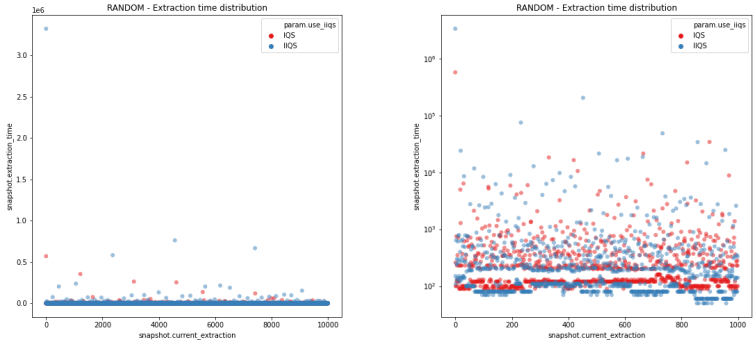
We start by revisiting the results shown on [21], in order to understand when the worst case appears and what it does imply for the execution of the algorithm. In order to not dive too deep into other aspects of the experimental design, we stick with the same metrics used in the original article.

As depicted in Figure 3.1, 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.

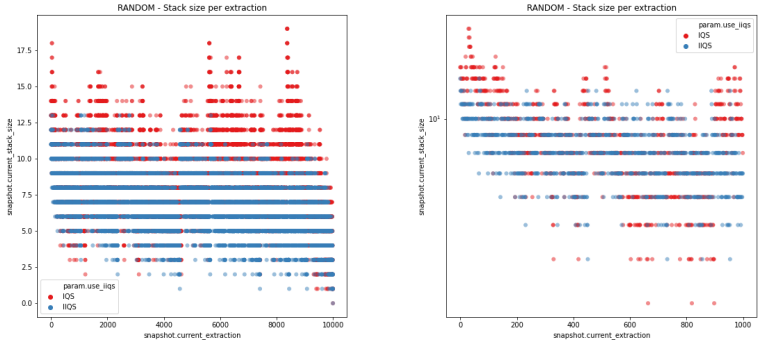
Another interesting aspect of both algorithms is that the stack size remains stable among all performed extractions. This is proof that in average, the stack does not



(a) Cumulated extraction time



(b) Extraction time



(c) Stack size

Figure 3.1: Base IQS and IIQS benchmark for a random sequence with 1×10^5 elements.

grow asymptotically than $log_2(n)$.

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 to run a partition which yields the element on the same position as it is being found. It is shown on Figure 3.2.

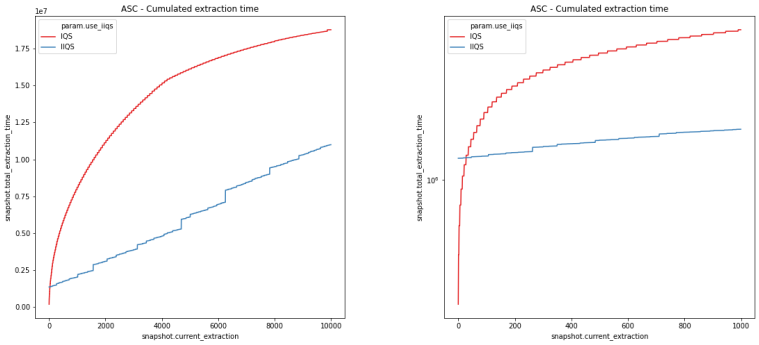
There is one noticeable difference between this experiment and the original study and is that the stack in our case grows, but barely reaching the minimum expected on the average case. This is caused by the implementation of the partition algorithm. Usually a partition algorithm uses two pivots which 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 behavior 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 3.2.5.

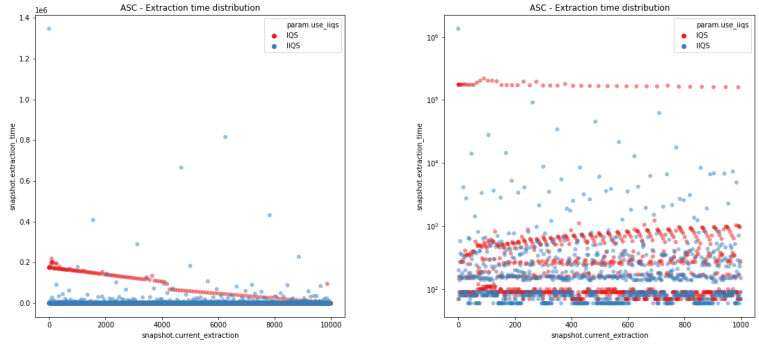
On this implementation, since the partitioning swaps an element to the back of the to-be-partitioned segment, on the last step the last element alternates positions with the first on the partitioned sequence, as such, when the next element in the left-most position is selected, it ends being an element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

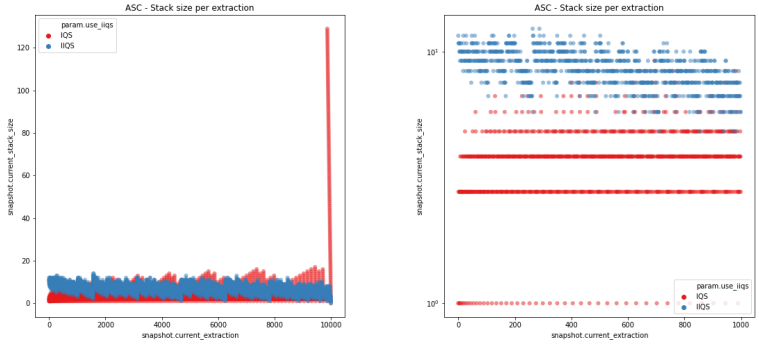
The most extreme case for IQS is when the pivot selection always yields an element which does not reduce the search space –not for a useful value at least– and the



(a) Cumulated extraction time

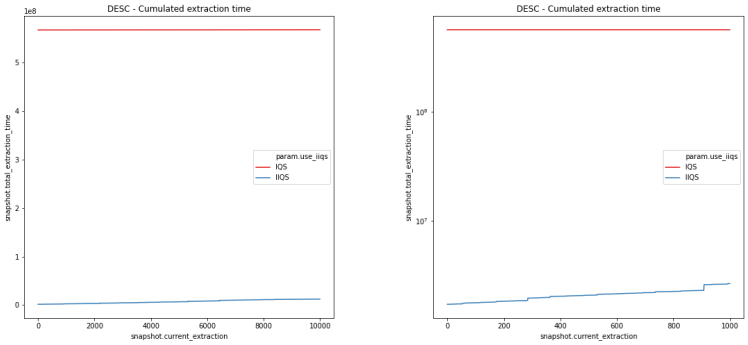


(b) Extraction time

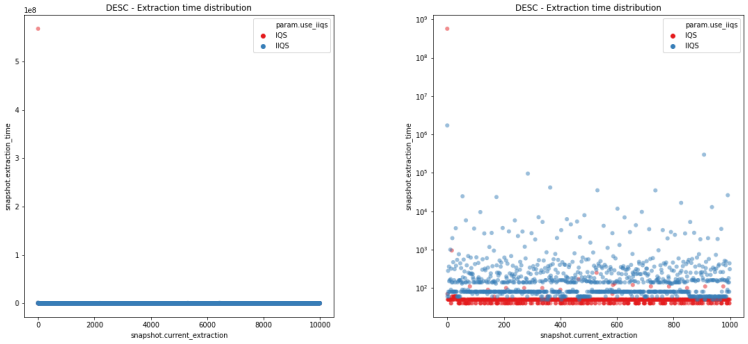


(c) Stack size

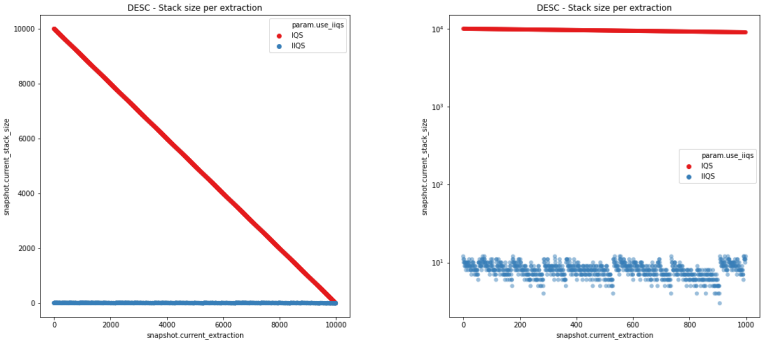
Figure 3.2: Base IQS and IIQS benchmark for an ascending sequence with 1×10^5 elements.



(a) Cumulated extraction time



(b) Extraction time



(c) Stack size

Figure 3.3: Base IQS and IIQS benchmark for a descending sequence with 1×10^5 elements.

stack gets populated with all the elements at the first call as it can be seen on Figure 3.3. 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.

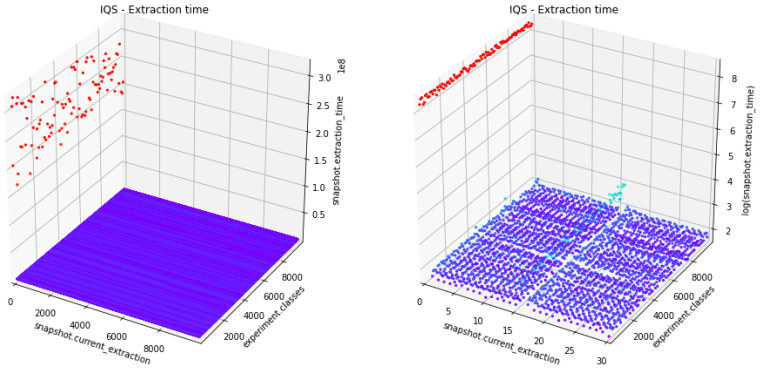
As it can be seen, for the both three cases that can be found –synthetically– on IQS, its introspective version performs with the same behavior on all of them, but it does cost a fixed constant time more than the original version. In this context it is useful to remember that the purpose of incremental sorting algorithms is to speed up the first calls under the premise that the number of minima extracted is unknown while preserving the optimal asymptotic complexity in case of sorting the entire sequence. In this regard, IIQS manages to perform just as expected.

3.1.2 Influence of repeated elements

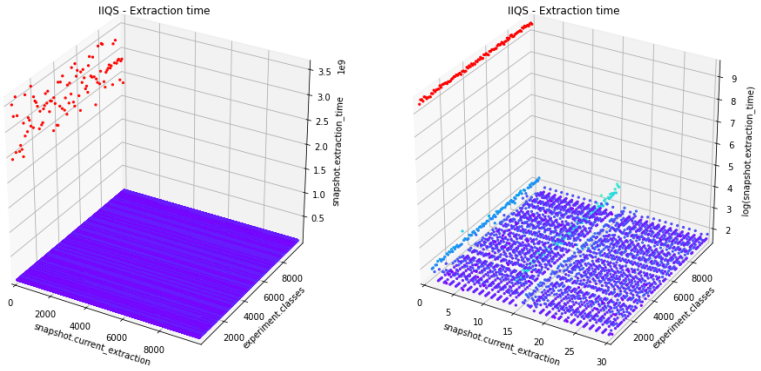
The question at hand now is what happens when repeated elements are found on the sequence. As mentioned before, this is the reason behind the change of the partition algorithm. In contrast to the original study, now we need to introduce three new factors to considerate:

- **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 from now on consider any distribution of the repeated element classes an 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.

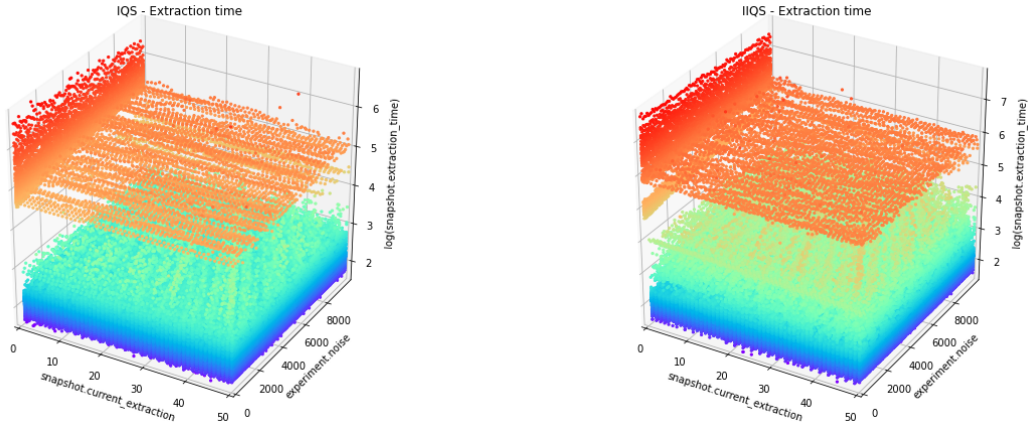


(a) IQS



(b) IIQS

Figure 3.4: Base IQS and IIQS benchmark for class impact on extraction time.



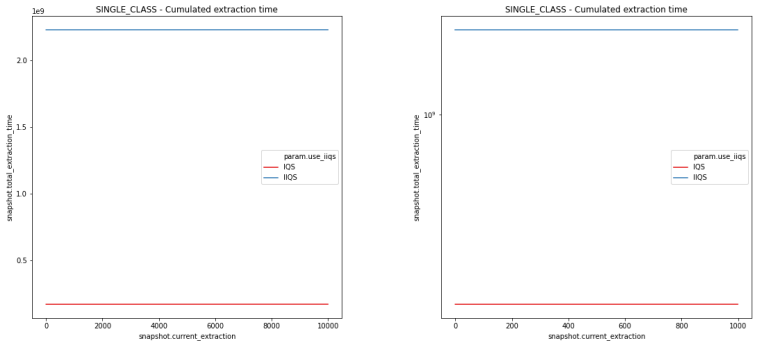
(a) IQS (b) IIQS

Figure 3.5: Base IQS and IIQS benchmark for random noise impact on extraction time.

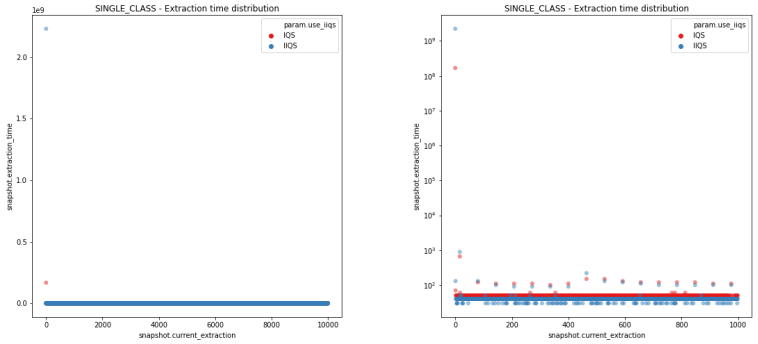
As for the random noise in the sequence[1], as by itself as it does not make any visible changes on the behavior of the algorithm in relation to its iterations. This is important, because the fact that a single class with low noise represents a case with the same behavior as found on the worst case of IQS for both algorithms, as it can be checked on Figure 3.6.

At first glance there is no sign of the number of classes affecting the overall running time of the algorithm, as the Figure 3.4 exhibits a similar behavior to the one found on both IQS and IIQS, except for a small decrease of the extraction time when fewer classes are found. Thus, we can conclude that the mere fact of having at least a single repeated element can make our algorithm to fall into a worst-case scenario, comparable to the same intended to avoid on the original IIQS implementation.

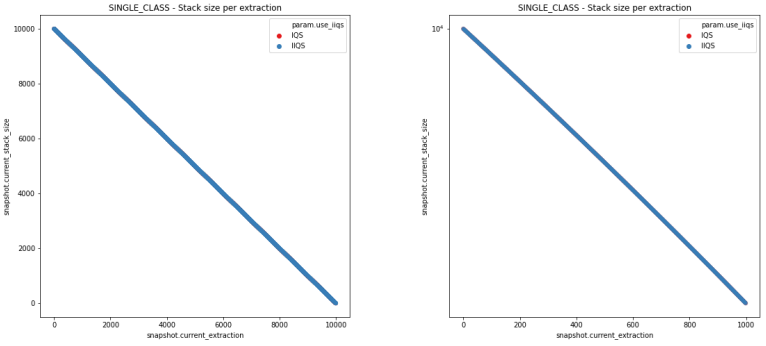
[1]We define as the random noise as the number of elements which does not belong to any controlled classes for the experiment. In this case, how many random elements are introduced as fillers on the array.



(a) Cumulated extraction time



(b) Extraction time



(c) Stack size

Figure 3.6: Base IQS and IIQS benchmark for a random sorted sequence with 1×10^5 repeated elements.

3.2 Partitioning Schemes

3.2.1 Rationale

Before diving into the results for this experiment, we 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.

3.2.2 Partition schemes

As explained before in 1.2, 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 optimize.

3.2.3 Lomuto's partition scheme

The most noticeable 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 1.1.2 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.

Algorithm 6 Lomuto Partition

```
1: procedure lomuto(A, p, r)
2:     x ← A_r
3:     i ← p − 1
4:     for j ∈ [p, r − 1] do
5:         if A_j ≤ x then
6:             i ← i + i
7:             swap(A_i, A_j)
8:     swap(A_{i+1}, A_r)
9:     return i + 1
```

3.2.4 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 ← A_p
3:      i ← p − 1
4:      j ← r + 1
5:      while true do
6:          do
7:              j ← j − 1
8:          while A_j ≤ x
9:          do
10:             i ← i + 1
11:         while A_j ≥ x
12:         if i < j then
13:             swap(A_i, A_j)
14:         else
15:             return j
```

3.2.5 Dutch flag problem

Both of the aforementioned algorithms were designed to operate on sets, but when it comes to sequences, to use such partition methods fails dramatically. As it treats repeated elements as unique elements, in the 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.

3.2.6 Problem definition and solution

Let us 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 is clear that if we chose p equal to 5, the element in the fifth position of S_1 is the pivot on its correct place. But it is 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 as they are following the definition of partitioning. There is no safety guarantee that the resulting pivot partitions the array in half in order to ensure an log_2 decay on the problem space. Situation worsens if all the elements are repeated, as it defeats the purpose of partitioning the sequence [21].

This problem is also known as the Dutch Flag problem [6], which for given a sequence it partitions 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 returns the indices of the beginning and the end of the middle portion.

Algorithm 8 Three-way Partition

```
1:  procedure threewaypartition(A, p)
2:      k ← ||A||
3:      i ← 0
4:      j ← 0
5:      while j < k do
6:          if A_j < p then
7:              swap(A_i, A_j)
8:              i ← i + 1
9:              j ← j + 1
10:         else if A_j > p then
11:             k ← k − 1
12:             swap(A_i, A_k)
13:         else
14:             j ← j + 1
```

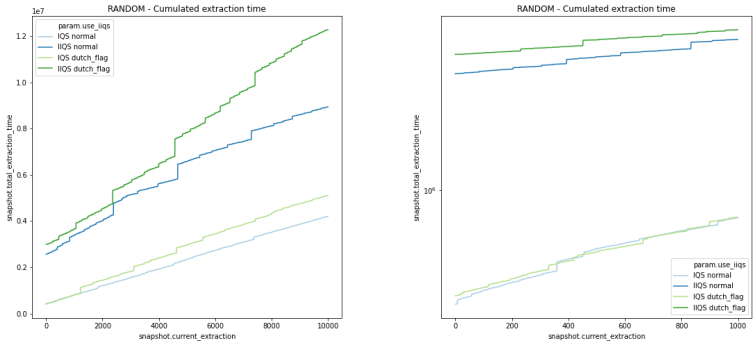
3.2.7 Integration into IQS as base implementation

It makes no sense to check our base partition implementation against repeated elements, as its behavior is undefined. But we can do the opposite, to test our dutch-flag implementation in order to check if the time bounds are equivalent to each other. In this regard, we are not to make assumptions that the following test apply for all implementations available of partition-based sorting algorithms.

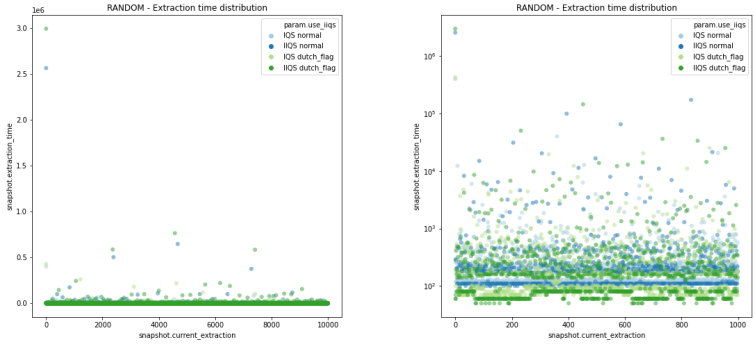
As depicted in Figure 3.7, there is no noticeable change in terms of complexity for both implementations of the partitioning algorithm when dealing with shuffled sequences. As such, it does result interesting how for IIQS both implementations of partition break up the array at the same index, which confirms that both implementations perform the same operations in the same order. On the other hand, it does look that for IQS there is a scalar overhead which seems to be reduced over time.

Figure 3.8 and Figure 3.9 confirms our hypothesis in Section 3.1.1 about the different behavior of IQS and IIQS when dealing with repeating sequences and how the stack is consumed, validating our base benchmark results.

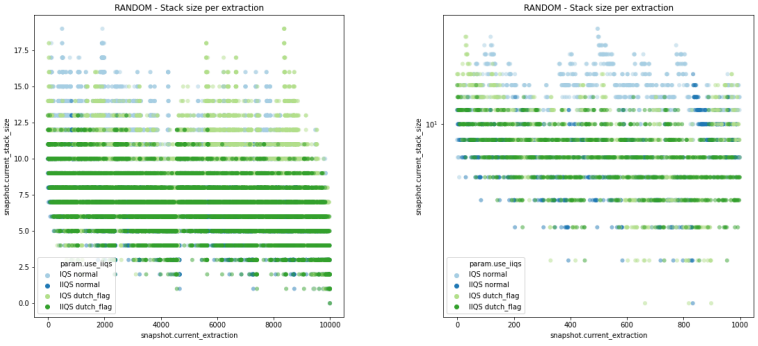
Apart from those behavior deviations, there is no noticeable difference between running time of our implementations for three-way partitioning and standard partitioning algorithms in terms of complexity. As such, from now on we use *three-way-partition* as our default implementation for the partitioning stage. This allows us to contrast results between IQS and IIQS with such modifications using both repeated and non-repeated elements as dataset inputs under the same partitioning conditions.



(a) Cumulated extraction time

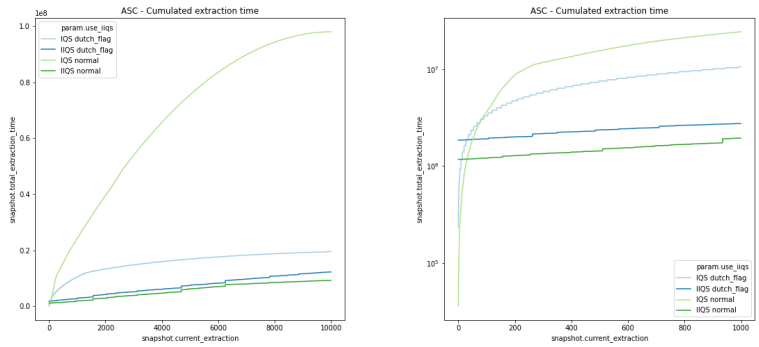


(b) Extraction time

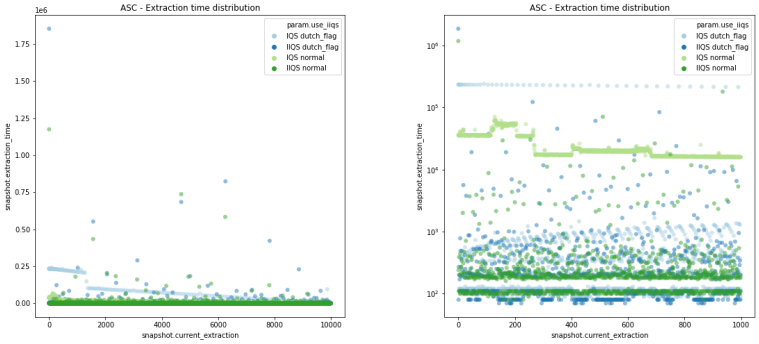


(c) Stack size

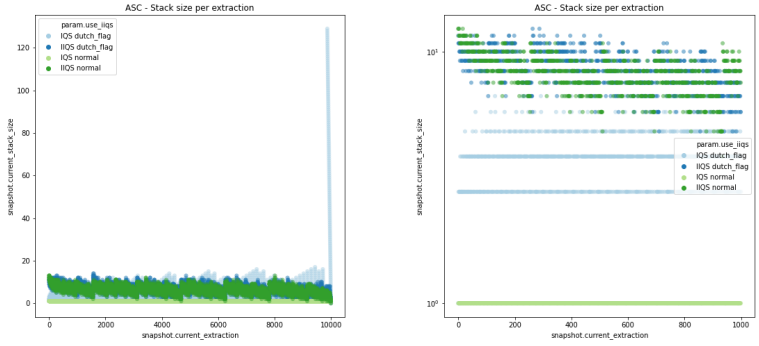
Figure 3.7: Dutch Flag partition influence on IQS and IIQS benchmark for a random sorted sequence with 100×10^3 elements.



(a) Cumulated extraction time

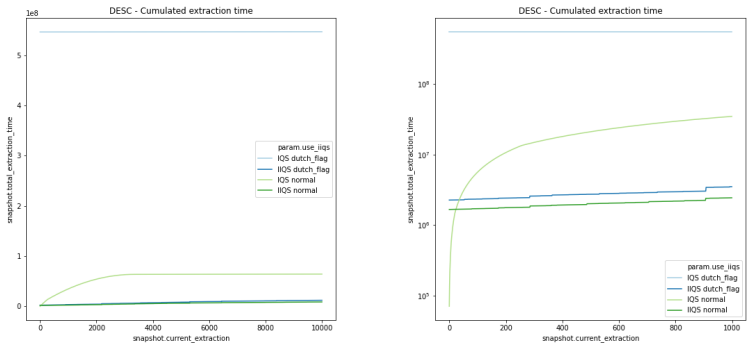


(b) Extraction time

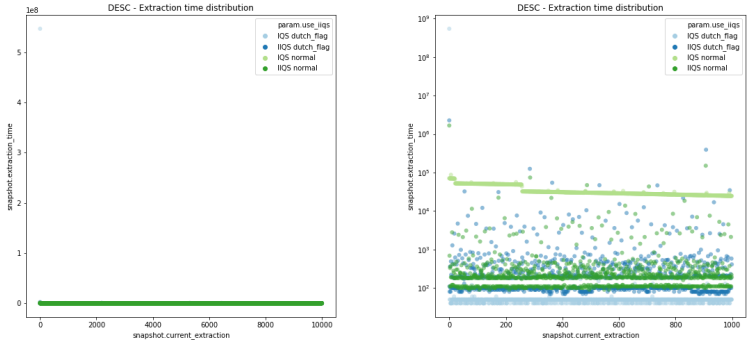


(c) Stack size

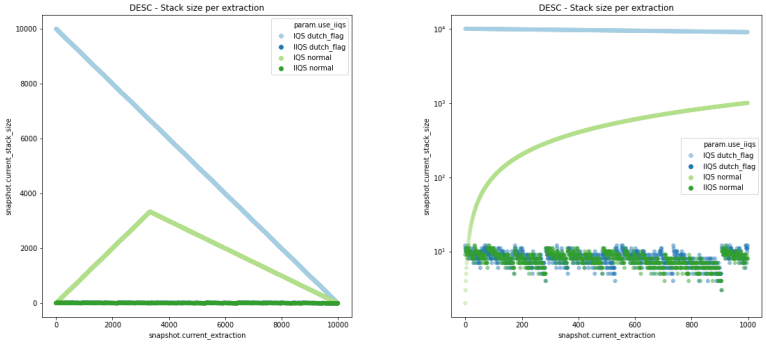
Figure 3.8: Dutch Flag partition influence on IQS and IIQS benchmark for an ascending sorted sequence with 100×10^3 elements.



(a) Cumulated extraction time



(b) Extraction time



(c) Stack size

Figure 3.9: Dutch Flag partition influence on IQS and IIQS benchmark for a descending sorted sequence with 100×10^3 elements.

3.3 Influence of pivot selection

In this experiment we establish a baseline for selecting pivots on partition-based sorting algorithms. While it is commonly known that randomized approaches are the most effective way of dealing with unknown distributions [9], in this approach we want to confirm if by biasing the selection on the partitioning stages of the algorithms have any direct influence over its running time.

3.3.1 Understanding pivot bias

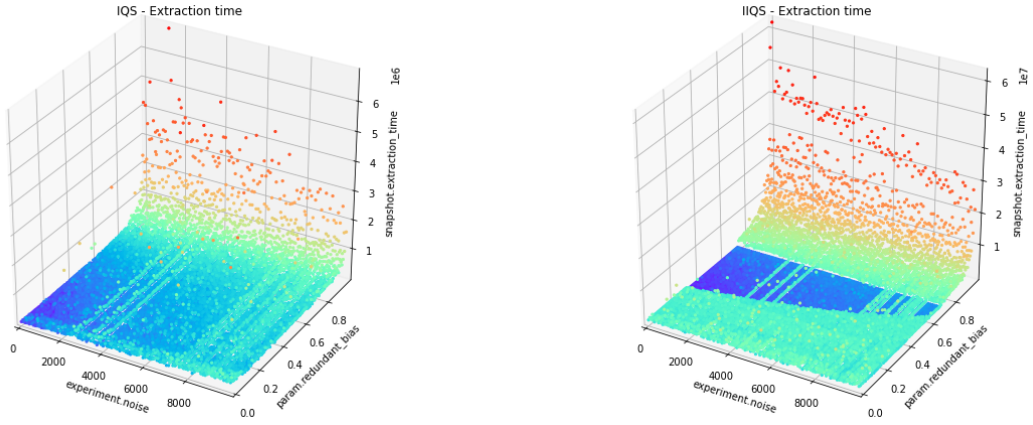
As explained before in Section 3.2, all partition based algorithms are based in the idea of generating two equal sized partitions, which contributes to a continuous decrease by a factor of 2 on each iteration. Under normal circumstances, it is expected that $log_2(n)$ partition operations are executed for a given sequence of n size.

But this assumption only yields when there is only one pivot to select. Meaning that there is a single element which divides both partitions generated by the algorithm, so for the case of a *three-way-partition* it does not apply. Let us take an example of a sequence S of size n which contains n unique integer elements. So, by induction on the integer number definition, each element if used as pivot for a partition can only yield one and only one pair of partition sets on S.

When repeated elements are found on S we face a problem with the definition itself of partition presented in Section 3.2.6, as there is no guarantee that the partitioned element is present or not in the resulting array. Using a three-way partitioning element does not solve the problem of reducing the search space and in this regard we have the following two options:

- Group the repeated elements and threat that set of repeated element as they were a unique element on S — we discuss this approach on Section 4.1.
- Preserve the elements into the array, and we apply a set of rules to choose which element deliver as our pivot.

This makes a lot of sense when revisiting the plot at Figure 3.4, as the more predominant is a certain class in a sequence, the less important becomes the pivot



(a) IQS (b) IIQS

Figure 3.10: Random noise and pivot bias impact on extraction time benchmarks.

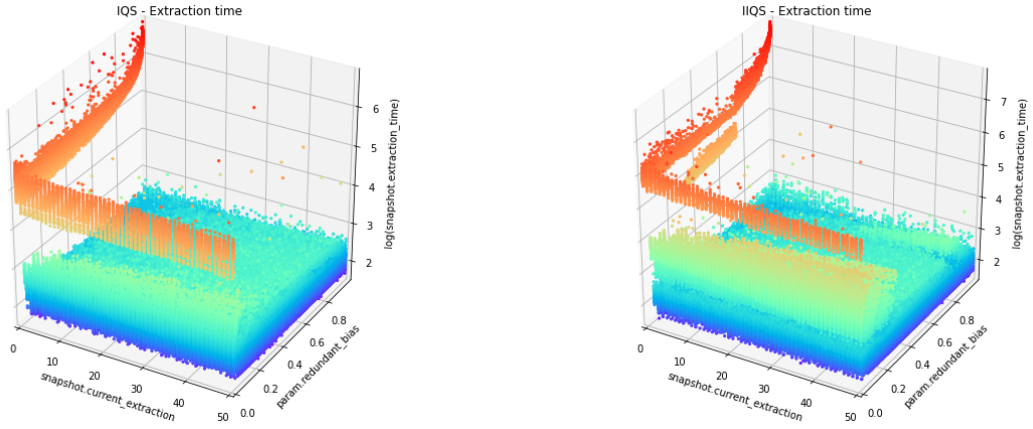
value selected but their final position.

3.3.2 Effects on the partitioning

Let us consider the pivot bias as a rational number $p_b \in [0, 1]$. This bias allows us to map to indexes in S to a floating point representation by assuming $S_i = \lfloor n \cdot p_b \rfloor, \forall\ i \in [0, \|S\|($. In colloquial terms, this means that an index of 0.0 means that the returned pivot belongs to the leftmost index of the middle segment, a value of 0.5 a pivot in the middle of the segment and 1.0 the rightmost element, assuming that the elements are sorted in an ascending fashion.

For our first observation we limit the result to only examine the first extraction performed by IQS, as it is already known that it is the most compute intensive operation in this algorithm. Then, we can observe in Figure 3.10 that both the noise amount and the bias over the partition have a huge impact when extracting the first element in the sequence which is also different for both implementations of the algorithm.

There are some interesting effects of the sequence noise for the repeating case instance. IQS shows the best performance as the noise decreases but also as the



(a) IQS (b) IIQS

Figure 3.11: Pivot bias and extraction impact on extraction time benchmarks.

pivot bias is more inclined towards selecting the smaller elements. But this effect it is only an illusion generated by the same case depicted on Figure 3.2, as less noise is found in the sequence, the greater the chance to select as pivot an element belonging to the repeating portion of the sequence, hence the bias effect studied on IQS for the partition comes into full effect. Both changing the bias towards greater values and to increase the noise contributes to increase the extraction time, it is clear that the bias has a greater effect than noise.

Then something interesting happens on IIQS, as the behavior is not as smooth as on IQS case. There is a valley that it appears to be in function of the bias, which for certain amounts of noise in the sequence it favors their execution. This is because of a forced execution of the introspective steps. Both noise and pivot bias have a huge impact on the times that this step is executed, hence, additional $O(n)$ time is consumed on the iteration when the noise exceeds IIQS fixed α and β constraints [21]. In the same fashion as our previous observations, the fact that this effect decreases when inducing more noise is related to the probability of randomly selecting a pivot which belong to a repeated portion of the array. When this happens we notice a great reduction of the running time, which suggest that we should adjust our pivot bias to match our α and β values in order to get the best performance.

Let us assume now that there is no noise in our sequence, in order to match the

best case execution for IQS. Then by looking the effects of the pivot bias against the subsequent extractions in Figure 3.11 we notice setting our pivot bias towards the leftmost elements in the pivot segment is not a good idea. When there is only one class and there is no noise in the sequence, the extractions become more costly as the pivot bias deviates from the ideal 0.5 value. This holds for both the first extraction and the subsequent ones. Also, it does clearly state that fixing the pivot bias to one extreme or another is not a good idea.

For IIQS the same rules apply, with the difference that there is a boost of performance when the bias belongs to the $[\alpha, \beta]$ range, as the introspective step rearranges the element and this ends being cheaper in some cases than to continue the partition stage blindfolded.

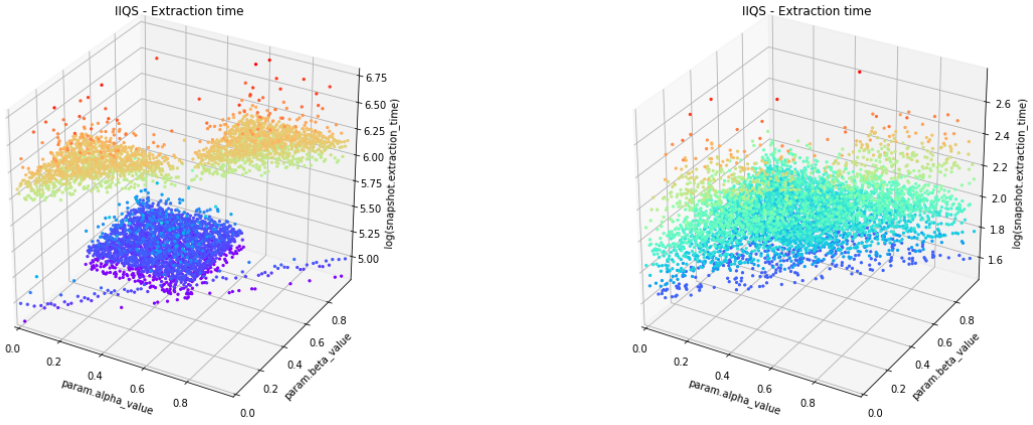
3.4 IIQS exclusive parameters

In this section we revisit a key parameter of IIQS, the values for α and β parameters. As explained previously on Section 1.2.2 and in the original work of IIQS [21], these two values are used during the evaluation of IIQS in order to trigger the execution of an extra partition stage. This extra stage uses a median-of-medians algorithm instead of just relying on the default random pivot selection. The benefit of changing the execution scheme is that it guarantees that the returned median belongs to a position in the central 40 percent of the array. This extra partition stage has $O(n)$ complexity, hence preserves the original asymptotic bound of the partition algorithm.

3.4.1 Understanding median-of-medians usage

While stated that it does not affect the asymptotic complexity of IQS, it increases its running time by a huge constant factor, as operations performed on BFPRT are not cache friendly for large arrays, specially when implemented in an in-place fashion[2].

[2]This is because when implemented in-place the idea is not to return the value of the median but rather the index of it. This forces us to move the partial medians generated to a place on which can be cached for next executions and easy controlled. As it is of common knowledge for computer scientists, this position is the beginning of the array.



(a) IIQS first extraction

(b) IIQS second extraction

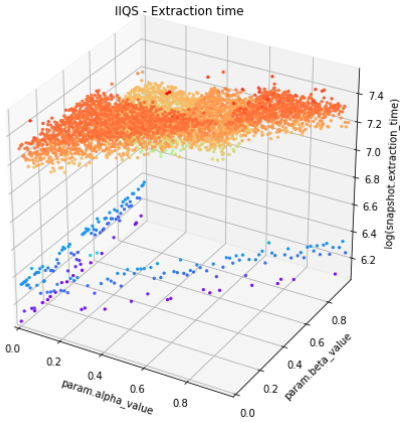
Figure 3.12: IIQS α and β benchmark impact on extraction time for a randomly sorted sequence.

There is no point on shrinking the *valid area*[3] for triggering BFPRT after its evaluation, as there is no guarantee on the resulting distribution of the returned index. Moreover, as BFPRT does not return a median but rather an area on which the median can be found and a partial partitioning of elements [2], we cannot use this technique as more than an approximate median selection algorithm which shuffles elements along the array.

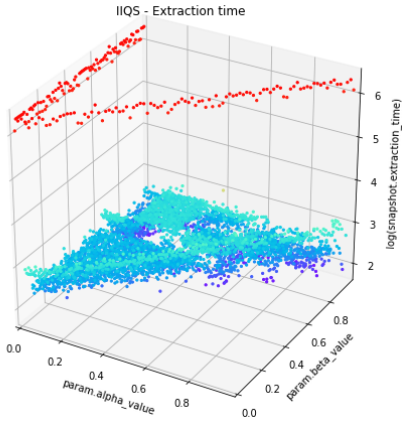
There is an interesting effect resulting of the selection of α and β parameters. By just examining the first and second extractions we can appreciate that if we tight the bounds to a point that median-of-medians is executed each time, for all instances we get that the first extraction is accelerated by a huge factor, but that comes at a cost of the second extraction being three orders of magnitude larger than the average execution. There is a ton of room for different behaviors on this implementation as we can check when comparing the execution for each dataset, the regions that seems to be best performing for one are the worst for another case.

But something that it is common is that closing the bound too tight affects negatively the running time as well as to completely ignore the value of α parameter,

[3]Our central 40%.

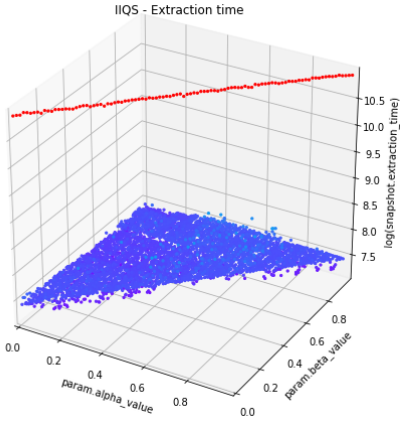


(a) IIQS first extraction

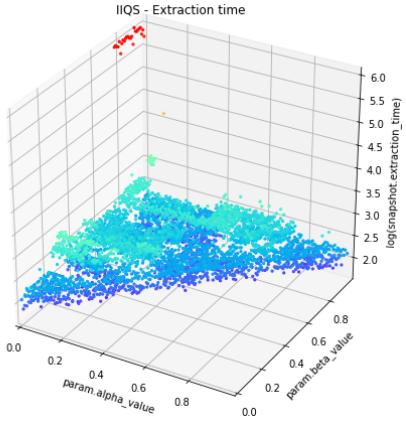


(b) IIQS second extraction

Figure 3.13: IIQS α and β benchmark impact on extraction time for an ascending sorted sequence.



(a) IIQS first extraction



(b) IIQS second extraction

Figure 3.14: IIQS α and β benchmark impact on extraction time for a descending sorted sequence.

at least when fixing all constraints to force a worst-case running time. That is, by fixing the pivot-bias to select the leftmost element. By extension, this applies to the other possible values for the pivot bias.

3.4.2 Relaxation of α and β constraints

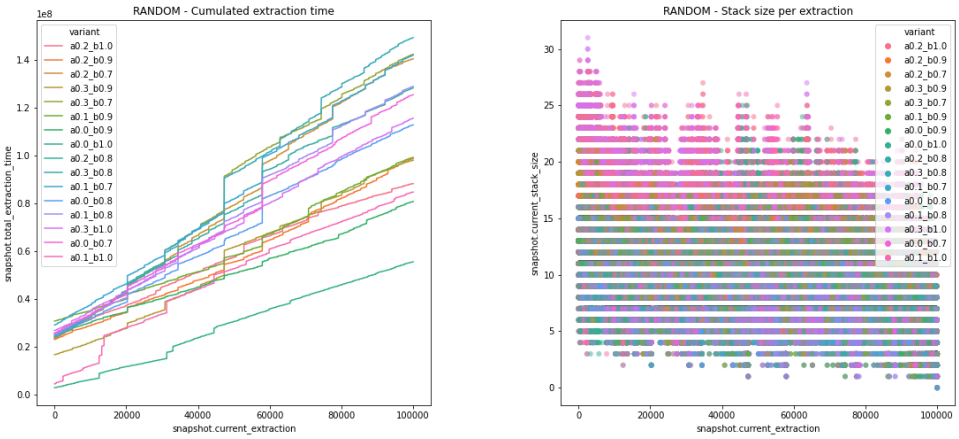
As stated by BFPRT definition, as the pivot returned belongs to the middle 40 percent of the array, there is no need to run IIQS if our pivot already belongs to this segment. But yet there is no insight on if relaxing the constraints for IIQS α and β parameters does have any impact on its running time.

As seen on figures there are some configurations which favor certain scenarios. On this first analysis presented on Figures 3.15, 3.16 and 3.17 we can appreciate that further relaxation of α and β parameters contribute to noticeable improvements over running time without affecting in most cases the quality of the stack in certain cases. Furthermore, as there is no combination which gets closer to an all-cases resistant implementation, we conclude that relaxation of such parameters outside their guaranteed bounds does not contribute to the development of a general usage version of IIQS.

3.4.3 Tightening α and β constraints for unique elements

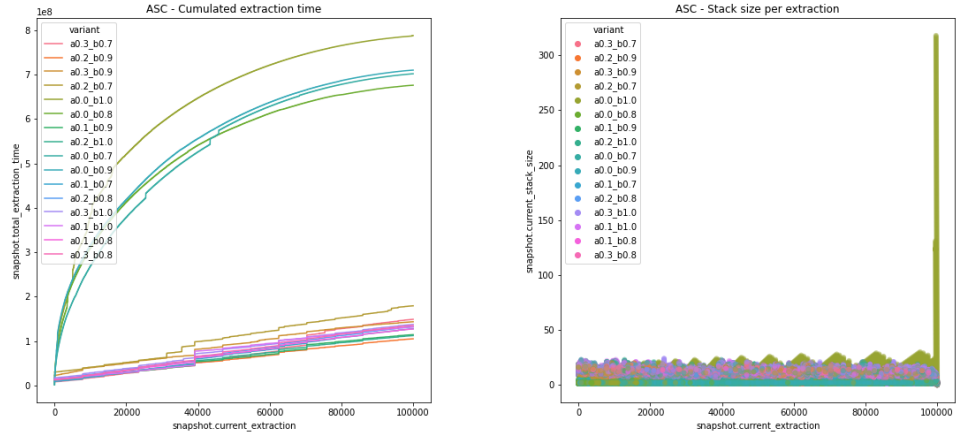
As for independent tuning of α beta shown in Figures 3.18a, 3.18d, and 3.18g and β parameter shown in Figures 3.18b, 3.18e, and 3.18h, there is no relaxation available outside the expected bounds for BFPRT unless we are expecting to optimize the execution for certain distributions.

A total opposite of the previous case, as it does seem that tightening BFPRT parameters does not contribute at lowering our current upper bound nor to changing it at all. But different to our initial expectations, it does seem that for certain combinations we obtain in average a better performance on all cases for non-repeated elements in the sequence. This is the case of $\alpha = 0.5$ and $\beta = 0.65$, which appears as the best performant combination of parameters on Figures 3.18c, 3.18f, and 3.18i.



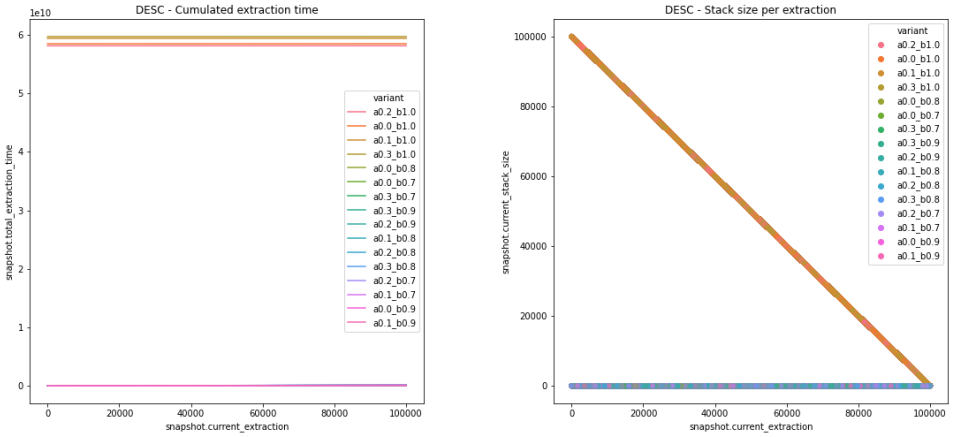
(a) Cumulated extraction time (b) Stack size

Figure 3.15: Rough IIQS α and β benchmark for a random sorted sequence with 1×10^6 repeated elements.



(a) Cumulated extraction time (b) Stack size

Figure 3.16: Rough IIQS α and β benchmark for an ascending sorted sequence with 1×10^6 repeated elements.



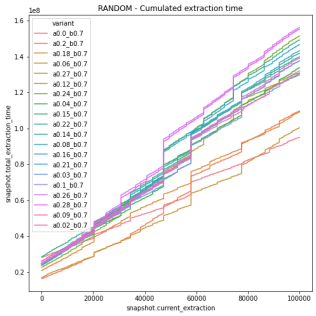
(a) Cumulated extraction time (b) Stack size

Figure 3.17: Rough IIQS α and β benchmark for a descending sorted sequence with 1×10^6 repeated elements.

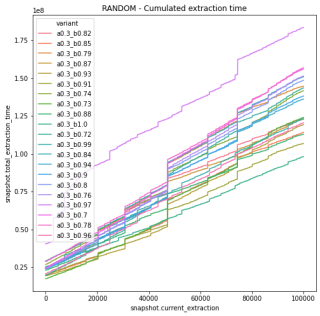
3.4.4 Effect of repeated elements in the sequence

Further, testing against the more aggressive situation for IIQS reveals that there are two cases when dealing with repeated elements on IIQS. When there is only one class across the entire sequence a performance boost for all extractions is only achieved by continuously executing IIQS on all iterations. In other cases, the performance is degraded by at least two orders of magnitude. More interesting, for all cases on which BFPRT were executed, the stack remained solid, showing the highest possible size along all executions.

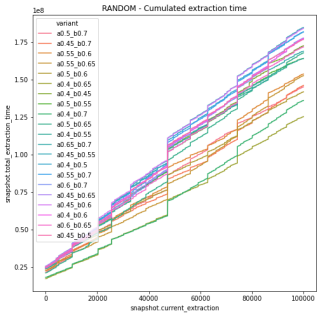
As by examining the results depicted in Figures and 3.20, we can conclude that changing IIQS parameters does not affect the performance when dealing with repeated elements, as the bias for returning a position on the BFPRT result as pivot is the one directly impacting the performance and not the bounds for BFPRT execution itself.



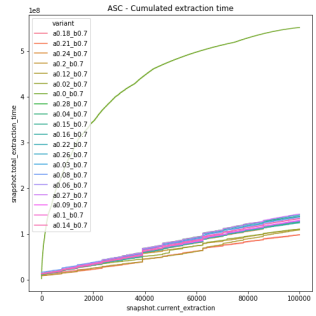
(a) Variable α, random sequence



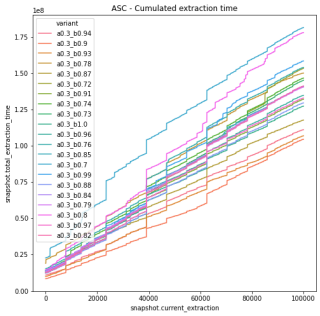
(b) Variable β, random sequence



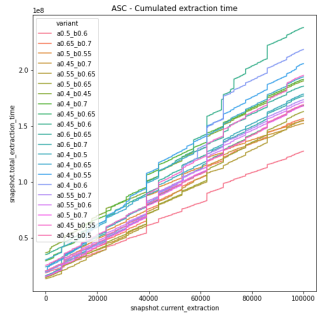
(c) Variable α and β, random sequence



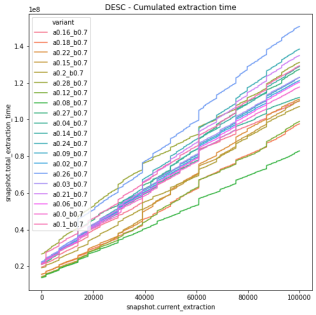
(d) Variable α, ascending sequence



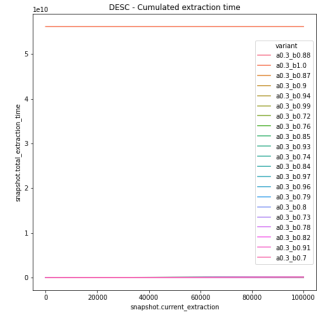
(e) Variable β, ascending sequence



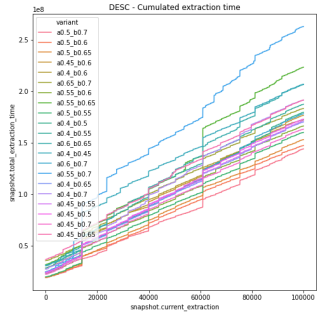
(f) Variable α and β, ascending sequence



(g) Variable α, descending sequence

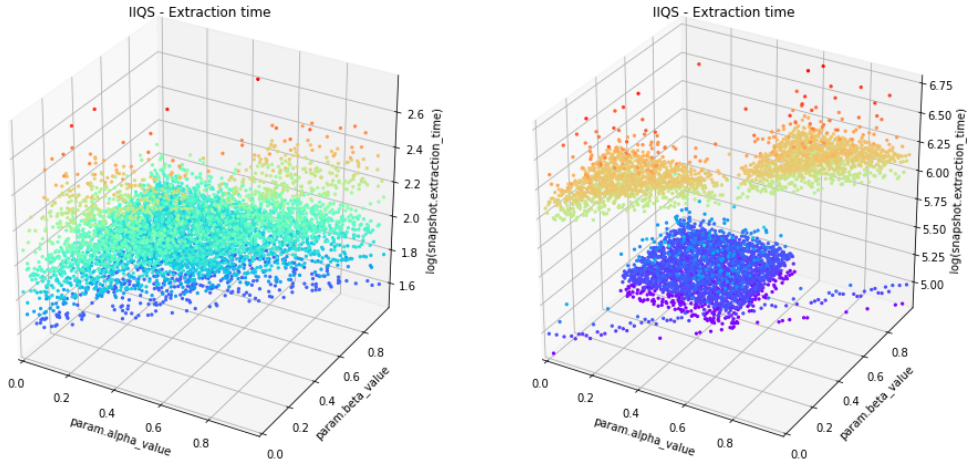


(h) Variable β, descending sequence



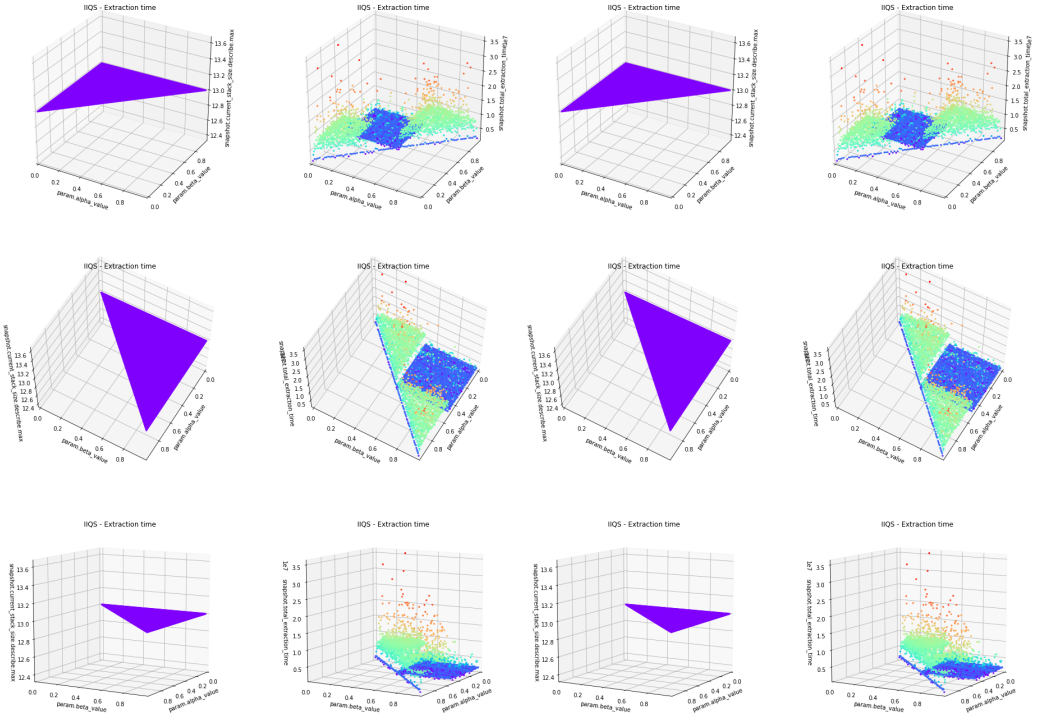
(i) Variable α and β, descending sequence

Figure 3.18: IIQS variable α and β benchmark for 1×10^5 elements.



(a) First extraction (b) Second extraction

Figure 3.19: IIQS α and β benchmark for a sequence with repeating elements of size 1×10^6 with a unique class.



(a) First extraction (b) Second extraction

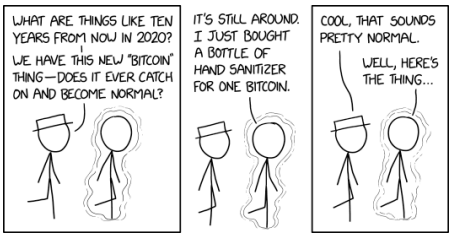
Figure 3.20: IIQS α and β benchmark for a sequence with repeating elements of size 1×10^6 with a unique class. Cumulative stats.

3.5 A note on median selection techniques

As our attention is now focused on BFPRT, it is worth questioning if the side effects of sorting the small segments used on BFPRT does affect IIQS behavior. While this does hold for elements with repeated elements, as the scope of this work is to support the worst possible case for repeating elements in a sequence, experiments related to this parameter are not covered in this document.

In order to understand why BFPRT modifications are our of our interest, let us consider the size of BFPRT segments as a parameter for IIQS execution. This is important as for each execution of BFPRT the elements on each index moves to their corresponding position in their local segment. Let us denote as k the size of the median used on BFPRT, for now we fix this value for all our examples as $k = 5$ unless stated otherwise.

Then, when a sequence S is composed of $\|S\|$ elements which follows $\forall\, S_i, S_j \in S,\, \forall i \neq j : S_i = S_j$, as $S_i = S_j$ holds for all cases, any permutation σ and τ of size k is bound to $\sigma = \tau$. Hence, sorting any portion of any size in the sequence does not change the running time as the elements value remains the same along the sequence. The only way to make a change is by controlling how stack is allocated on the execution, which is shown at the beginning of this section.



4. Workhorse experiment

In this section, we present the consolidation of our previous experiments in the form of a bigger experiment with a more in-depth analysis.

4.1 Repeating elements problem

Contrary to our initial expectations, BFPRT parameters did not influence the result when dealing with a sequence with repeated elements. So, for this experiment we change tune a different level, as we now see the structures involved on its resolution.

Reminiscing the main concept of incremental sorting, this approach was designed to tackle the problem of extracting elements from a sequence in a sorted fashion in optimal time when there is no information beforehand on how many elements are needed to be extracted from its source.

In all our previous experiments we always took as granted the definition of our input sequence, which from now we denote as S_{in}. For all our implementation and structural concerns, S_{in} have a defined size $\|S_{in}\|$ over a $\mathcal{Z}^{\geq 0}$ domain.[1]

Depending on the problem to be solved using IQS/IIQS, we may need different representations for the output. In this regard, we can see IQS and IIQS as a function which maps a sequence S_{in} into another sequence S_{out} which contains the same elements but binned via a certain criterion. We do not specify the criteria at this

[1]This may seem obvious, but as this structured can be implemented in an arbitrating way, we need to constrain our restrictions for the program as much as possible in order to clearly define our algorithm bounds.

point, as it is not the point of this work to compare different comparison mechanisms. For such effects is safe to assume that our comparison criteria is the one in Section 1.2.

4.1.1 Current paradigm

In rough terms, both IQS and IIQS algorithms belong to the divide-and-conquer paradigm as they recursively break our problem into a smaller one which are solved later. Contrary to a complete sorting algorithm, on partial sorting we do not expect to solve the problem for our entire input, in other words, we do not need to solve all the recursion steps in the execution tree in order to get an accepted output.

As an example of this let us consider a sequence $T = 4, 1, 2, 3$. Now let us say that we need a partial sorting over the first two elements of T. Then our solutions are $T_1 = 1, 2, 3, 4$ and $T_2 = 1, 2, 4, 3$. Both T_1 and T_2 are accepted inputs as the first two elements are sorted. This aforementioned definition does not bound our final implementation to solve all the recursion tree induced by our paradigm and considers this solution as a state of our recursion tree.

Another —less graphical— way of depicting this situation is by modeling our execution tree as a non-deterministic stacked state machine, on which our root node the final node, the leaves of the recursion tree are the initial states (the elements that can be placed in their correct position) with strategically placed elements to be consumed in the stack. There is no need to go through all the states as long as we reach an accepted final state.

In practice[2], few sorting algorithms run pre-checks in order to avoid certain cases. As these *sanity checks* are bound to cost at least $O(n)$ time, they are not feasible in most cases. That is why introspective algorithms like introsort [17] and IIQS [21] integrate such process to be part of already existing processes, in order to not affect the overall complexity.

[2]As another way to say, in common implementations.

4.1.2 Paradigm shift

Sequences with repeating elements are a different story. Let us take as example a sequence of elements $T' = 1, 1, 1, 1$, which shares the same size as our previously introduced T sequence, but this one has only one class of elements. Extrapolating the definition of a sequence with only one class among all sequence elements stated in 3.5, this sequence is already sorted as it is. As all the possible permutations on T' share the same values, and the comparison are being made on the values itself, then there is no problem to solve. As our minimal problem has the same size of our original one, our paradigm is no longer valid anymore.[3]

If we want to continue using IQS or IIQS to solve our incremental sorting problem we first need to validate our paradigm and then solve the problem at hand. Paraphrasing this statement, we need to integrate a process which allows us to map our input to an accepted one without affecting our current complexity.

4.1.3 Counting elements to reduce complexity

As shown in Section 3.1, our ideal input for both IQS and IIQS is a sequence with unique elements. Counting sort already solves this problem by mapping our elements present in S_{in} as a set C of ordered pairs (v, f) on which v is the value in S_in, and v is the frequency of the value in S_in. [12] Sorting is achieved by the container of the set or by extending the set to a sequence of elements and storing them accordingly — this being the preferred way of implementation.

Unfortunately, while counting sort performs in lineal time, this algorithm does not exhibit a reasonable space-complexity performance when compared to IIQS, as in the worst case it requires extra $O(n)$ space whereas IIQS requires $O(log_2(n))$.

4.1.4 Mapping

As shown in Section 3.5, our sorting problem does not take into account the position of the elements and only relies on the value of such elements. This allows us to use

[3] In fact, the invalidation of the problem paradigm is the real reason behind IIQS failure.

the mapping induced by counting sort, C_{in}, as an intermediate representation of our original sequence S_{in}.

Let us define $S_{repeated(\gamma)}$ as a sequence of elements $(s_0, s_1, ..., s_{n-1}, s_n)$ on which $\forall\ s_i \in [0, n] :\ s_i = \gamma$. Then every sequence $S_i n$ can be seen as a concatenation of m sequences of size k on which $k \leq m$. Then we can establish two functions *roll* and *unroll* defined as follows:

$$\begin{aligned} roll\colon S_{repeated(\gamma)} &\rightarrow C_\gamma \\ (s_0, s_1, ..., s_{n-1}, s_n) &\mapsto (n). \end{aligned}$$

And analogously:

$$\begin{aligned} unroll\colon C_\gamma &\rightarrow S_{repeated(\gamma)} \\ (n) &\mapsto (s_0, s_1, ..., s_{n-1}, s_n). \end{aligned}$$

This mapping allows us to represent any sequence of elements as a sequence of pairs which represent the element and how many repetitions of this element are concatenated to it.[4]

4.1.5 Integrating counting strategies as an external process

As both *roll* and *unroll* functions operate per each element and not at sequence level we can extend their usage both inside of our current IQS and IIQS implementation or as an external process.

We denote our candidate sorting algorithm as an anonymous function λ_{sort}. Then our external reduction strategy is as follows:

Algorithm 9 shows an implementation of an external reduction strategy. We assume that roll operation as it is an instance of counting sort it takes $O(n)$ time and

[4]This representation is also known as *frequency table*.

Algorithm 9 External reduction

1: **procedure** $external\_strategy(S_{in}, k, \lambda_{sort})$
2: $C_{in} \leftarrow roll(S_{in})$
3: $S'_{sorted} \leftarrow \lambda_{sort}(C_{in}, S, k)$
4: $S'_{out} \leftarrow [\,]$
5: **for** $s \in S'_{sorted}$ **do**
6: $S'_{out} \leftarrow S'_{out} \frown unroll(s, C_{in}(s))$
7: **return** S_{out}

line 4 onwards also take $O(n)$.

As the average running time for IQS and IIQS is $O(n + k\ log_2(k))$, then the addition of our external strategy to deal with repeating elements introduces a fixed $n + l$ overhead on which l denotes the number of classes present on S_{in}. But this overhead does not change the complexity of both algorithms. But this does not prevent IQS from hitting its worst case scenario unless we control how the keys retrieved from C_{in} are retrieved.

Another drawback of this implementation is that since we are using an external strategy to reduce the repeating elements problem, we need l extra space. In the worst case, $l = \|S_{in}\|$, which is suboptimal in relation to IQS.

4.1.6 Integrating counting strategies as part of the partition process

When integrating this problem reduction stage directly into the algorithm, then we need to materialize it at implementation level. For such effects the ideal place to install it in on the stack which tracks the information for the next calls.

In order to accomplish this, we change the structure of the stack used by extending the operations involved on it, allowing to group a range of elements instead of just a single element. For such effects, the stack store pairs of elements $u = (p_{start}, p_{end})$ on which p_{start} is the first position of the pivot and p_{end} is the frequency on the array.

By extension of the concepts shown in Section 4.1.4 and 3.5, we extend the

operations *partition*, *top* and *pop* to support both *roll* and *unroll* definitions as follows:

Algorithm 10 Binned Stack top

```
1: procedure Stack.binnedTop(S)
2:     (p_start, p_end) ← S.top()
3:     return p_start
```

Algorithm 11 Binned Stack pop

```
1: procedure Stack.binnedTop(S)
2:     (p_start, p_end) ← S.top()
3:     S.pop()
4:     if p_start < p_end then
5:         S.push((p_start + 1, p_end))
```

Algorithm 12 Binned Three-way Partition

```
1: procedure binnedPartition(A, p)
2:     k ← ||A||
3:     i ← 0
4:     j ← 0
5:     while j < k do
6:         if A_j < p then
7:             swap(A_i, A_j)
8:             i ← i + 1
9:             j ← j + 1
10:        else if A_j > p then
11:            k ← k − 1
12:            swap(A_i, A_k)
13:        else
14:            j ← j + 1
15:    return (i, k)
```

4.2 Binned IIQS

The extensions shown in Algorithms 10, 11 fixes the position of the index returned from the middle section as pivot to the left-most element. As proven in Section 3.4

this bias generates a synthetic best-case execution, which is used directly after the partition stage. Those two algorithms integrate both *roll* and *unroll* definitions by storing the range of pivots in a map notation.

On the other hand the extensions for Algorithm 12 are integrated into IIQS almost directly with no major modifications, allowing a transparent use without any extra overhead, as shown in Algorithm 13.

Algorithm 13 Binned IIQS

```
1:  procedure B-IIQS(A, S, k)
2:      while k < S.binnedTop() do
3:          pidx ← random(k, S.binnedTop() − 1)
4:          pidx, range ← partition(A_{k,S.binnedTop()−1}, pidx)
5:          m ← S.binnedTop() − k
6:          α ← 0.3
7:          β ← 0.7
8:          idx_α ← k + αm
9:          idx_β ← k + βm
10:         r ← −1
11:         if pidx > idx_β ∧ idx_α < pidx then
12:             pidx ← pick(A_{r+1,S.binnedTop()−1})
13:             pidx, range ← binnedPartition(A_{r+1,S.binnedTop()−1}, pidx)
14:         S.push((pidx, range))
15:     S.pop()
16:     return A_k
```

As both Algorithms 10, 11 have $O(1)$ worst-case complexity and Algorithm 12 maintains $O(n)$ complexity on which n denotes the number of elements in the sequence to partition. Both expected and worst-case time complexity for this *Binned IntrospectiveIncrementalQuickSort* (*bIIQS* from now on) remains $O(n + k\ log_2(k))$ and our original $O(log_2(k))$ space complexity is maintained as the stack only doubles its size at most.

As elements in the stack are stored as a pair, this enables the use or custom structures to store such pairs, which enables a better use of DRAM bursts on the target machine. This guarantees a nearly zero overhead on read and write operations

for the stack.

The experiments developed on Sections 3.4 also helps us to fix our parameters for the execution of the introspective steps to $\alpha = 0.5$ and $\beta = 0.65$ as experimentally has been proven to offer the best performance from all tested combinations.

This makes *bIIQS* a direct replacement of *IIQS* for all cases. Additionally, if worst-case execution is not relevant, the same modifications are available for its use on *IQS* implementations in order to support repeated elements on its input.

4.3 Experimental evaluation

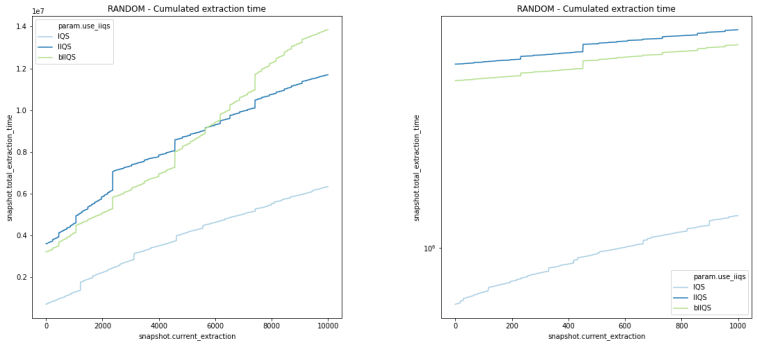
Now we present the experimental results for our proposed implementation of bIIQS against our existing implementations of IQS and IIQS to validate our aforementioned proposal.

4.3.1 Performance comparison for unique sequences

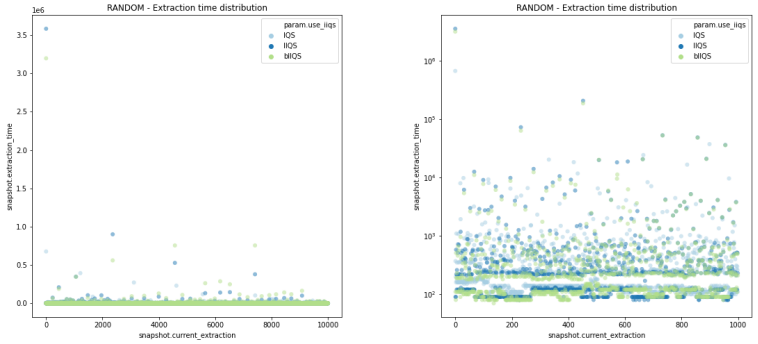
Experiments have shown that bIIQS does not induce an increase of the algorithm complexity for our standard test cases such as random sequences, ascending sorted sequences and descending sorted sequences, as shown in detail in Figures 4.1, 4.2 and 4.3. For then, our hypothesis on the equivalence of IIQS and bIIQS for our base IQS test cases is confirmed.

4.3.2 Performance comparison for repeated sequences

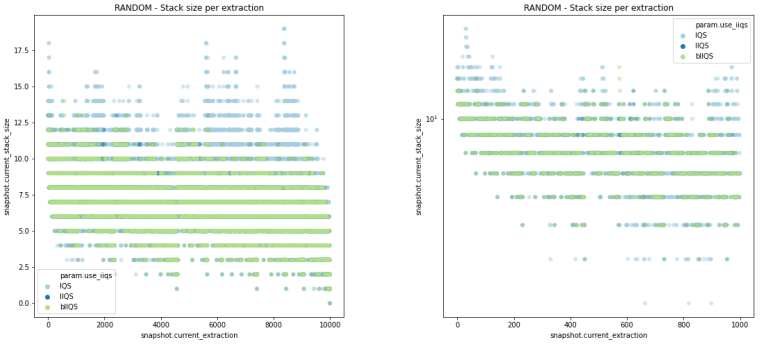
For single class instances which represent the worst case for IIQS when dealing with repeated sequences, bIIQS shows more than three orders of magnitude boost in performance in comparison to IIQS. As shown in Figure 4.6, while our single class input continues to represent our worst case, it does not pose a problem as before.



(a) Cumulated extraction time

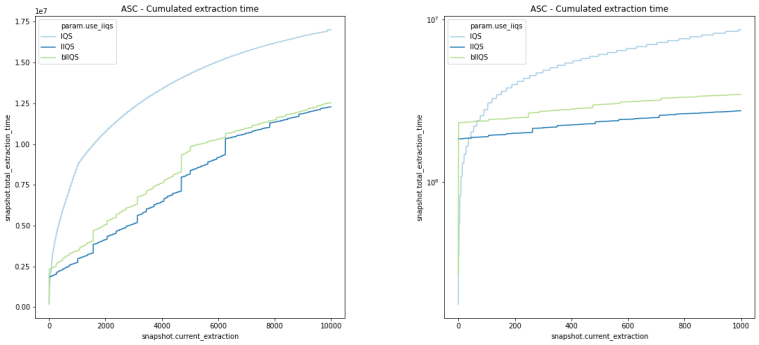


(b) Extraction time

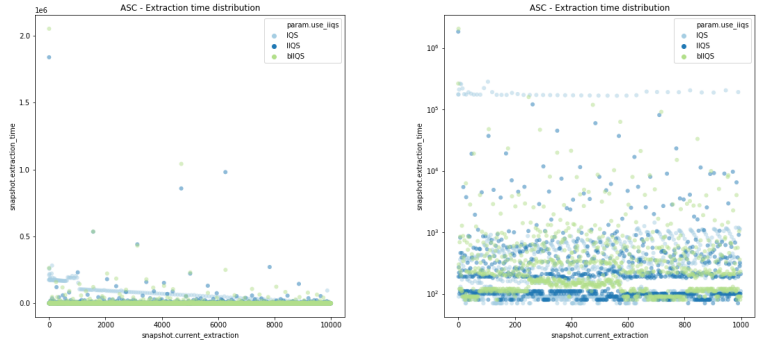


(c) Stack size

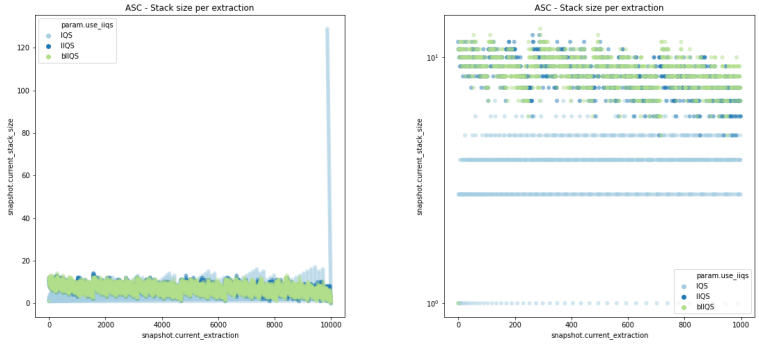
Figure 4.1: bIIQS benchmark for a random sequence with 100×10^3 elements.



(a) Cumulated extraction time

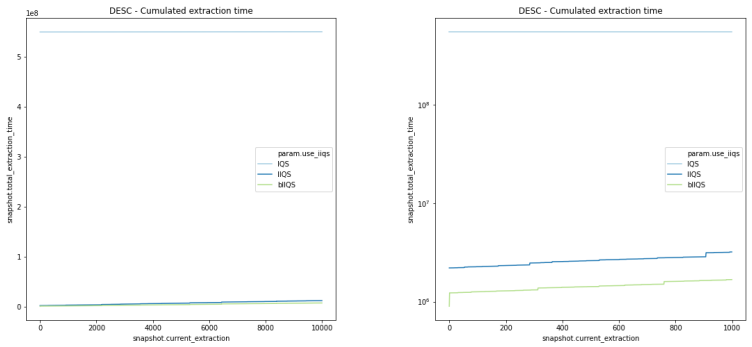


(b) Extraction time

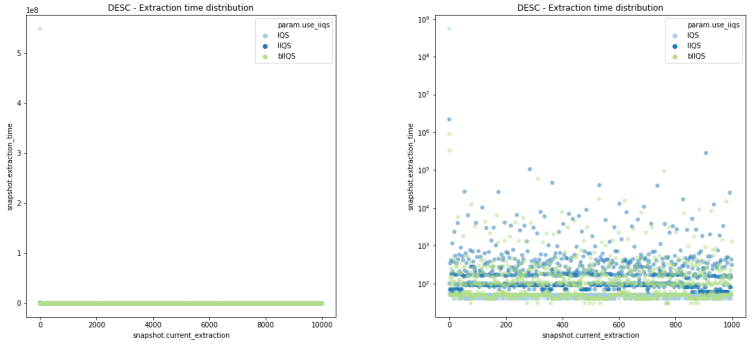


(c) Stack size

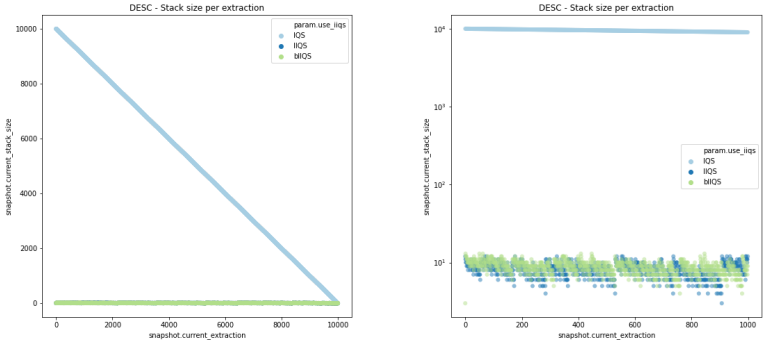
Figure 4.2: bIIQS benchmark for an ascending sorted sequence with 100×10^3 elements.



(a) Cumulated extraction time

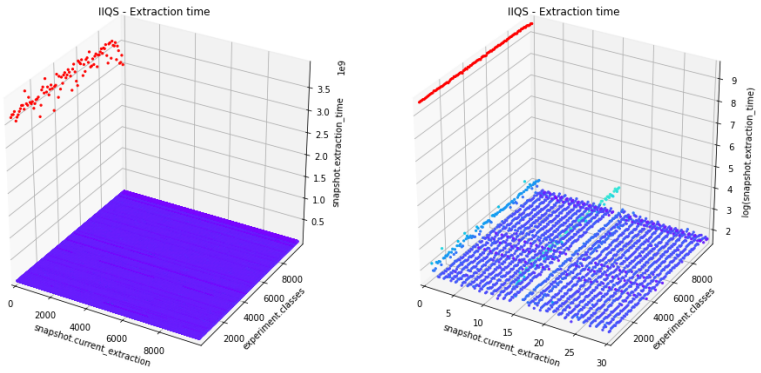


(b) Extraction time

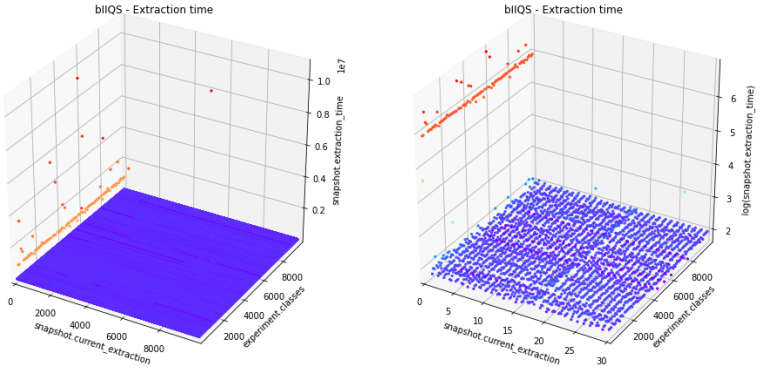


(c) Stack size

Figure 4.3: bIIQS benchmark for a descending sorted sequence with 100×10^3 elements.



(a) IQS

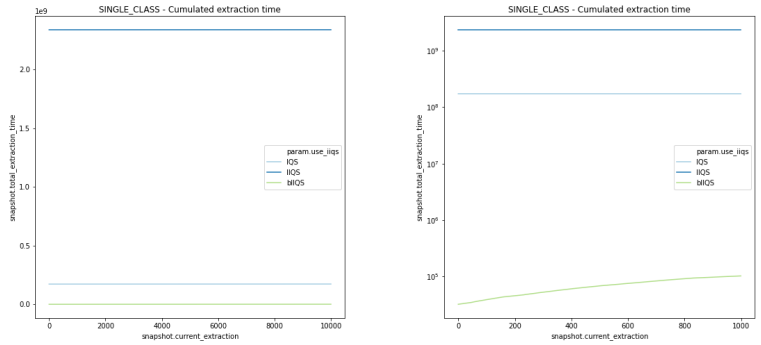


(b) IIQS

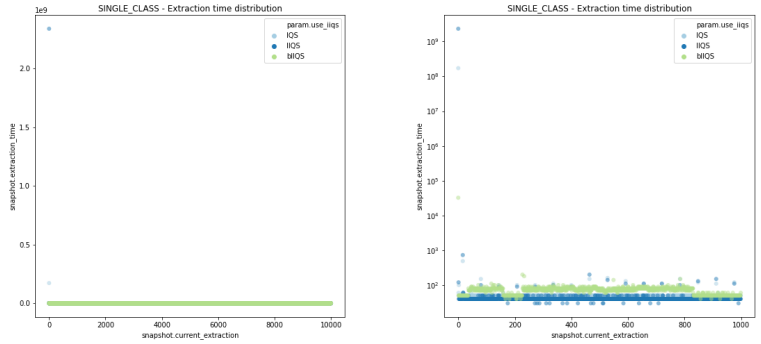
Figure 4.4: bIIQS benchmark for class impact on extraction time.

Experimental results shown in Figure 4.5 reveal that bIIQS does not only perform the fastest first element extraction, but also that formerly stated properties of IIQS as optimal space usage and fastest extractions are also preserved. As the stack now stores a range of elements, each extraction performed on it does not require any extra partition stage as it is being immediately delivered. This makes the —optional— pop-push operation the only overhead in this implementation.

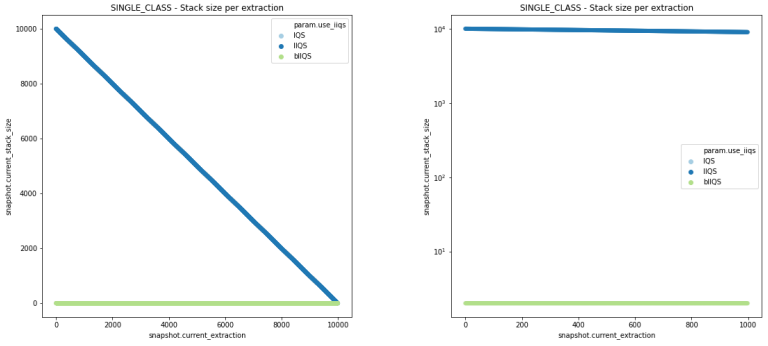
As for the noise impact we can clearly identify that now the bias on the resulting index provided by the three-way partition does not impact on the performance of bIIQS except for very small values (see Figure 4.6). In contrast, the noise now has a slight negative impact on bIIQS which can be attributed to the extra step performed to stack a pair of elements instead of a single one. Even with this degradation, the performance on all cases is stable in comparison to IIQS and delivers results in the



(a) Cumulated extraction time

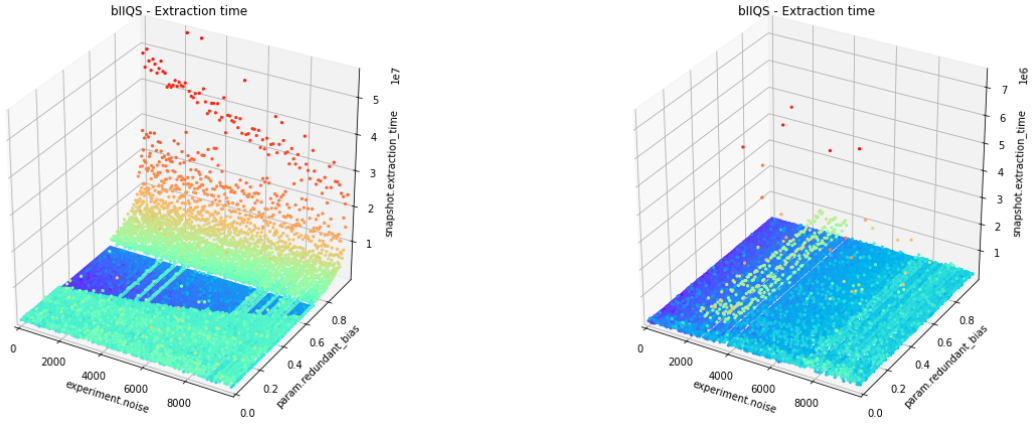


(b) Extraction time



(c) Stack size

Figure 4.5: bIIQS benchmark for a random sorted sequence with 100×10^3 repeated elements.

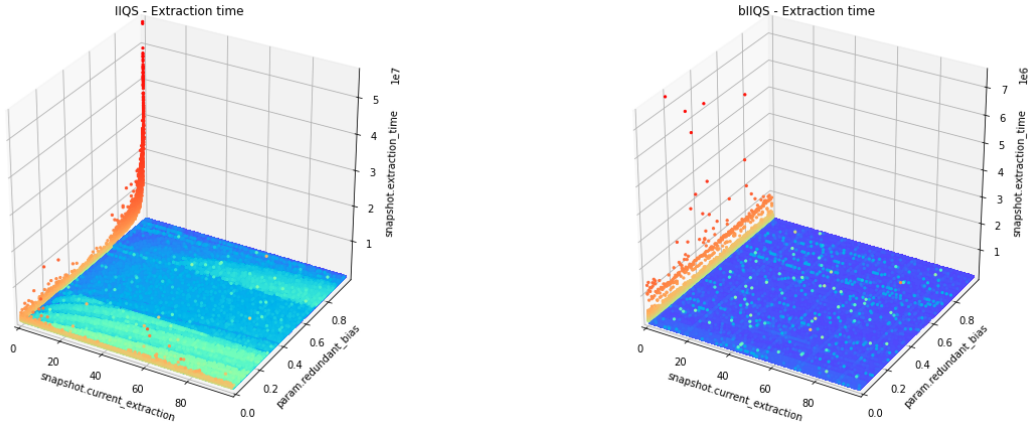


(a) IIQS (b) bIIQS

Figure 4.6: Random noise and pivot bias benchmark for bIIQS impact on extraction time.

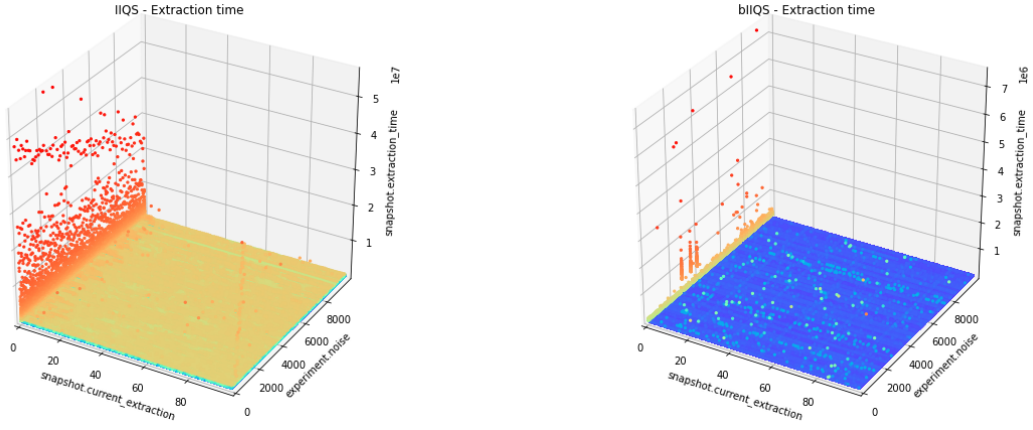
expected running time, on par with random sequences of unique elements.

We confirm our hypothesis by examining both noise impact and three-way-partition bias as separate elements depicted in Figures 4.7 and Figures 4.8 respectively.



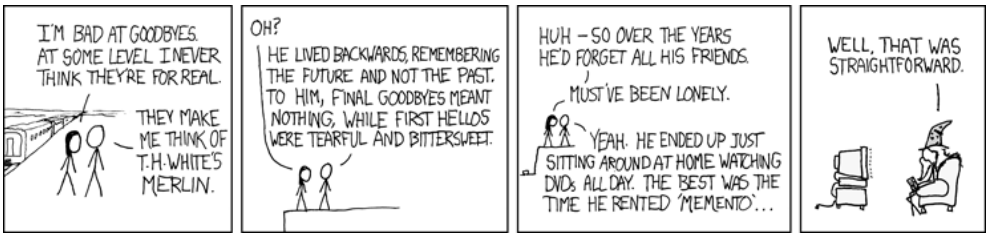
(a) IIQS (b) bIIQS

Figure 4.7: Pivot bias benchmark for bIIQS impact on extraction time.



(a) IIQS (b) bIIQS

Figure 4.8: Random noise impact on bIIQS extraction time.



5. Summary

In this work we have tackled the problem of extending IIQS, an algorithm designed to operate over sets of elements to support sequences of elements which can contain repeated elements. At first instance we present the execution of an experimental algorithmics methodology along with an in-depth overview of each foundation required to analyze and understand out problem. Then we propose a new variant of IIQS which is baptized as bIIQS which is an algorithm developed through the insight gained of those previous experiments.

As we already know, algorithm research is usually driven by formulating the problem and then analyzing its performance theoretically. However, most of those algorithms designed based on purely abstract foundations, most of specific-purpose algorithms lack on the implementation details which can cause confusion and misleading results.

In this work we have applied an experimental approach for our algorithm design process. Using the book *A Guide To Experimental Algorithmics*[15] as our base reference material, we have broken our lineal algorithm design process into a modular cycle. This cycle cam be summarized as understanding the problem at hand, designing experiments, analyze and interpret the results and formulate new questions.

As this process needs to be controlled, our goals are had been set to support a new input case for IIQS.

In order to gain useful insight from experiments, both careful planning and rigorous implementation is due. We had chosen C++ as our language of choice, as it enables us to fine tune both structural and execution aspects directly on the code. As for our tool for examining our data and plan our experimental process we have used Jupyter due to the simplicity of its pipelining. A direct consequence of this is the ease replication of results and validations by third parties.

During our experimental process we have discovered some implementation caveats which were not described on the original paper. We can summarize them as follows:

- IIQS belongs to the *adaptive sorting* algorithms family. This means that the algorithm performance is in function of its input. This allowed us to focus our efforts on later stages.
- Different partitioning schemes do not affect our algorithm complexity nor its running time as long as they divide the elements and provide a way to reduce our search space. But they do alter how the pivots are stored or retrieved.
- The implementation of IIQS stated on the original paper [21] does not state clearly which inputs it does accept. This work corrects this lack of information.
- IIQS auxiliary process —median-of-medians— can be tuned to speed up certain portions of the index if desired and is not needed to be a fixed value.
- The elements stored on the stack follow a certain chainsaw pattern when both IQS and IIQS are executing under normal conditions. As the stack size over time is directly related to the overall performance of IIQS, we can use it as a performance metric.
- Experimentally, IIQS has on average a 3-fold overhead in comparison to IQS.

It is important to state that while bIIQS does represent a huge improvement, given the nature of the extensions made through its development, there is more room for improvement. These improvements come directly related to hardware aspects and input nature.

The main results and findings of this work can be summarized as follows:

- We presented bIIQS, which is a direct replacement of IIQS aimed to support a new input case not considered on the original work.
- We have corrected misleading information from the original paper, by gaining insight using an *Experimental Algorithmics* methodology.
- We have made available a portable and extensible implementations of IQS, IIQS, and bIIQS aimed for further experimentation using C++.

5.1 Future work

As the scope of the work presented here has been bounded to only study how to implement an improved version of IIQS for a new input case and to bound its worst-case execution, we have not covered the study of the effects of different data distributions on IIQS. This can result useful on bioinformatics applications, on which visualization tools like *Haplotype plots* use incremental sorting strategies to give the user feedback on the program state.

A direct extension of this work is a study on the implications of using different computer architectures to run this algorithm. As IQS family obtains their speed thanks to a cache-friendly implementation, different architectures with different memory hierarchy models can yield interesting results for the use of this algorithm on constrained implementations.

A different spin-off of this study is directly related on execution timing. During our experiments we noticed a common problem when gathering execution data. As each snapshot takes a constant time to be stored on memory, as this information is stored it creates a little overhead which can alter results. This occurs because snapshots are stored on the same memory boundary as the program execution. Also, timing operations interrupt the original execution flow of the program.

As our timed sections nest into each other, this snapshot overhead accumulates deviating results. This problem is related to the *observer theory* described in quantum physics, which states that the mere observation of a phenomenon alters it state.

A promising alternative is to separate the implementation of the algorithm and their snapshot mechanism at physical level. That is, implementing them on different physical instances and communicate state changes as analogue signals. This could be accomplished by using bare-metal implementation of this algorithm using an FPGA software-based CPU wired to a separate high-resolution clocking system on the same chip.

Bibliography

[1] Ricardo Baeza-Yates and Udi Manber, editors. *Computer Science: Research and Applications*. Springer Book Archive. Springer New York, NY, 1 edition, 1992. eBook ISBN: 978-1-4615-3422-8. Softcover ISBN: 978-1-4613-6513-6. Published: 30 October 2012. eBook published: 06 December 2012.

[2] 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.

[3] Svante Carlsson, Christos Levcopoulos, and Ola Petersson. Sublinear merging and natural mergesort. *Algorithmica*, 9(6):629–648, Jun 1993.

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[5] G. A. Croes. A method for solving traveling-salesman problems. *Operations Research*, 6(6):791–812, 1958.

[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, University of Waterloo, CAN, 1992.

[8] Vladimir Estivill-Castro and Derick Wood. A new measure of presortedness. *Information and Computation*, 83(1):111–119, Oct 1989.

[9] Vladmir Estivill-Castro and Derick Wood. A survey of adaptive sorting algorithms. *ACM Comput. Surv.*, 24(4):441–476, December 1992.

[10] Benjamin C. Haller and Philipp W. Messer. SLiM 2: Flexible, Interactive Forward Genetic Simulations. *Molecular Biology and Evolution*, 34(1):230–240, 10 2016.

[11] Donald E. Knuth. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., USA, 1997.

[12] Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., USA, 1998.

[13] C. Levcopoulos and O. Petersson. Adaptive heapsort. *Journal of Algorithms*, 14(3):395–413, May 1993.

[14] Heikki Mannila. Measures of presortedness and optimal sorting algorithms. *IEEE Transactions on Computers*, C-34(4):318–325, Apr 1985.

[15] Catherine C. McGeoch. *A Guide to Experimental Algorithmics*. Cambridge University Press, USA, 1st edition, 2012.

[16] Kurt Mehlhorn. *Data Structures and Algorithms 1*. Springer Berlin Heidelberg, 1984.

[17] David R. Musser. Introspective sorting and selection algorithms. *Softw. Pract. Exper.*, 27(8):983–993, August 1997.

[18] Gonzalo Navarro. Implementing the lz-index. *Journal of Experimental Algorithmics*, 13:1.2, Feb 2009.

[19] Gonzalo Navarro and Rodrigo Paredes. On sorting, heaps, and minimum spanning trees. *Algorithmica*, 57(4):585–620, Mar 2010.

[20] Ola Petersson and Alistair Moffat. A framework for adaptive sorting. *Discrete Applied Mathematics*, 59(2):153–179, 1995.

[21] 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.

[22] Steven S. Skiena. Encroaching lists as a measure of presortedness. *BIT*, 28(4):775–784, Dec 1988.

[23] Jr. Wagner, John R., III Mount, Eldridge M., and Jr. Giles, Harold F. *Design of Experiments*, page 291–308. Elsevier, 2014.