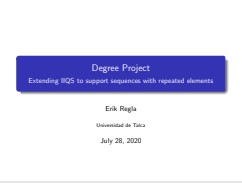
Degree Project

Extending IIQS to support sequences with repeated elements

Erik Regla

Universidad de Talca

July 28, 2020



Overview

Overview
1. Introduction
   - Context
   - Goals
   - Methodology
2. Methodology foundations
   - Algorithm instantiation hierarchy
   - Algorithm design hierarchy
   - Experimental process breakdown
3. Pilot experiments
   - Partitioning algorithms
   - Base benchmarks
4. Workhorse experiment
5. Experimental evaluation
6. Summary
   - Future work

La estructura de esta presentación se divide en los siguientes puntos:

Introduction

La estructura de esta presentación se divide en los siguientes puntos:

Primary Goal

To use an experimental algorithmics approach to design an extension of Introspective Incremental Quick Select which is tolerant to repeated elements in the input sequence.

Primary Goal

To use an experimental algorithmics approach to design an extension of Introspective Incremental Quick Select which is tolerant to repeated elements in the input sequence.

1. La meta principal de este trabajo es utilizar una metodología de algorimos experimentales para diseñar una extensión de Introspective Incremental Quick Select, el cual sea tolerante a entradas con elementos repetidos.
2. Para lograr estos objetivos, necesitamos previamente completar las siguientes tareas [next_slide]

Specific Goals

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

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

1. Investigar respecto a los fundamentos teóricos de Incremental Quick Select y de Introspective Incremental Quick Select, de modo de poder comprender como este funciona y cómo diseñar correctamente una implementación para sus pruebas.
2. Planificar una serie de experimentos piloto los cuales nos permitan obtener información al respecto.
3. Utilizar esta información para diseñar una nueva versión del algoritmo.
4. Finalmente, probar su correctitud de manera empírica por medio de experimentos.

Methodology overview

We use *A guide to experimental algorithmics* as our base resource to craft a custom methodology for our problem.

Methodology overview

We use *A guide to experimental algorithmics* as our base resource to craft a custom methodology for our problem.

1. A modo poder afrontar nuestro desafío de ingeniería de algoritmos, se decidió construir una metodología propia en base a diseño de experimentos.
2. Para estos efectos, se utiliza el libro de Catherine McGeoch *A guide to experimental algorithmics* como guia para el diseño de nuestro proceso.

Experimental algorithmics

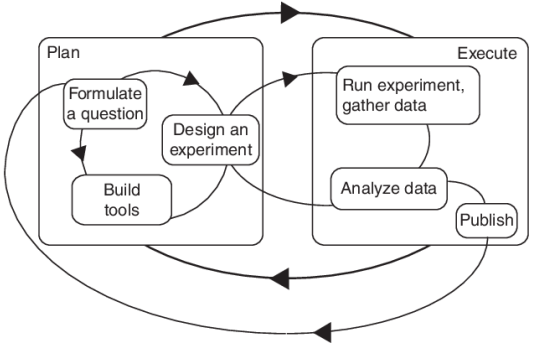


Figure: Experimental algorithmics cycle

Experimental algorithmics

Figure: Experimental algorithmics cycle

1. La extrapolación de la metodología de *Diseño de experimentos* a la ingeniería de algoritmos propuesta por McGeoch presenta el desarrollo de esta como un proceso iterativo compuesto de dos componentes principales.

Experimental algorithmics

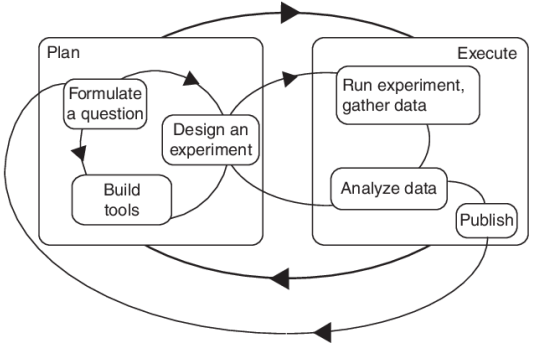


Figure: Experimental algorithmics cycle

Experimental algorithmics

Figure: Experimental algorithmics cycle

1. Por un lado, tenemos la etapa de planificación de nuestros experimentos en las cuales formulamos preguntas y en paralelo diseñamos nuestros experimentos y desarrollamos nuestras herramientas para el estudio.

Experimental algorithmics

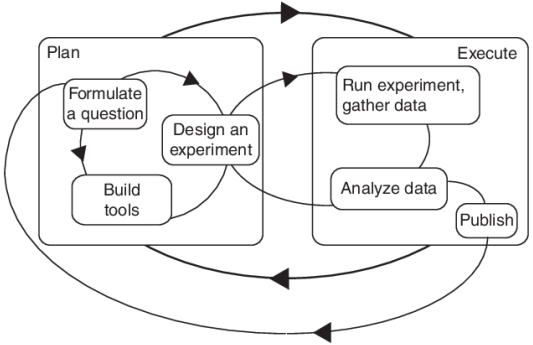


Figure: Experimental algorithmics cycle

Experimental algorithmics

Figure: Experimental algorithmics cycle

1. Por el otro tenemos la ejecución propiamente tal de nuestros experimentos, junto con el análisis e interpretación de nuestros resultados.
2. Hay que destacar que en esta extrapolación tanto las etapas de planificación como ejecución se ejecutan de manera cíclica sin tener un límite definido en las iteraciones ni orden.
3. Esto es asi ya que la etapa de ejecución puede llevarnos a nuevas preguntas y durante la planificación pueden surgir nuevas propuestas. Es tarea de quien ejecuta esta metodología el controlar el proceso.

Methodology foundations

TODODO.

Methodology foundations

TODODO.

1. El primer paso en el diseño de nuestra metodología de trabajo es separar las jerarquías de instanciación de algoritmos contra las jerarquías de diseño de algoritmos.
2. Seguido de esto, necesitamos formalizar las etapas de nuestros ciclos. Esto considera el cómo vamos a ejecutar nuestros experimentos, la información que recolectaremos y las consideraciones de iteración.
3. Todo esto está articulado gracias a la investigación previa realizada, sin embargo, cada iteración del proceso afecta el alcance de manera incremental.

Algorithm instantiation hierarchy

1. Para las jerarquías de instanciación de algoritmos (son los distintos niveles de abstracción de nuestro algoritmo las cuales son específicas de su representación final)...
2. Sabemos de antemano que el paradigma sobre el cual opera IQS e IIQS pertenecen a la familia de algoritmos de ordenamiento interno-adaptivo, basado en particiones. El cual será nuestro objetivo a optimizar.

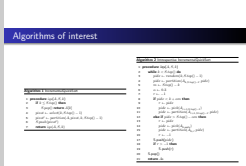
Algorithms of interest

Algorithm 1 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)
```

Algorithm 2 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
```

1. Por la naturaleza de ambos algoritmos, estos comparten una serie de rutinas de uso común. Entre las cuales se cuentan
2. Next, encargado de extraer el siguiente elemento (mínimo) en la secuencia -este es el algoritmo en si.
3. Partition, encargado de dividir el arreglo.
4. Swap, encargado de intercambiar elementos.
5. Push y pull, los cuales son operaciones propias de la pila subbyacente.
6. Adicionalmente, tenemos BFPRT y Median que son rutinas de uso exclusivo de IIQS, las cuales son utilizadas de acuerdo al resultado de la evaluación de salud de la ejecución.

Implementation and execution concerns

1. Nuestro programa fuente para la ejecución de los experimentos será C++ en conjunto de Boost para apoyar el paso de argumentos al binario resultante.
2. Respecto al código objeto, este es obtenido por compilación directa de nuestro código fuente, automatizado por medio de Makefiles sin tratamiento especial de los artefactos resultantes.
3. A nivel de proceso, todos nuestros experimentos son ejecutados sobre espacio de usuario sin privilegios adicionales de ejecución.

Algorithm design hierarchy

1. Ahora, las jerarquías de diseño de algoritmos (que son la separación lineal de su implementación desde su representación abstracta hasta su implementación)...
2. Tenemos que nuestros sujetos a experimentación estan diseñados de manera de prioritizar la flexibilidad de ejecucucion por sobre la convención.

Design overview

1. Para esto, los binarios a utilizar durante el proceso de experimentación fueron diseñados en base a tres unidades principales.
2. El ejecutor principal, quien es el encargado de recibir los argumentos de nuestro programa, configurar y ejecutar los experimentos sobre nuestros algoritmos.
3. La especificación de snapshots, al cual permite la recolección y almacenaje de datos de manera estructurada.
4. Los cronómetros que ejecutan las detenciones del código para perfilar su ejecución, los cuales están implementados utilizando el sistema de macros provisto por C para ejecutar los cambios en la etapa de precompilacion.

Implementation overview

1. A modo de mejorar la replicabilidad de los experimentos se optò por utilizar un sistema de aleatorización sistemática combinado con un inicialización fija de parámetros iniciales. De esta manera nuestra ejecucucion cuenta con una única fuente de verdad.
2. Por otro lado para poder mejorar la calidad de los resultados, se optó por declarar globalmente una instancia única de snapshots junto con pre-alocación de memoria de manera adelantada para evitar interferencias de llamadas a sistema para tales efectos.

Metrics

1. Dentro de las métricas definidas para nuestro proceso de experimentación contamos:
2. Número de intercambios, el cual tiene propósitos de verificación.
3. Operaciones de la pila, la cual ya en los trabajos previos se ha visto que estas se manifiestan de forma directa en el tamaño que esta tiene a lo largo de cada extracción e impacta directamente en el tiempo de ejecución.
4. Y el tiempo de ejecución.

Partitioning repeated elements

1. Antes de comenzar con los experimentos necesitamos asegurarnos que nuestra implementación de IQS no caiga en casos donde este no pueda responder.
2. Ya que la implementación no especifica el mecanismo de particionado, cambiamos nuestra versión de el algoritmo de particionado a una versión compatible con secuencias repetidas.

Base benchmark

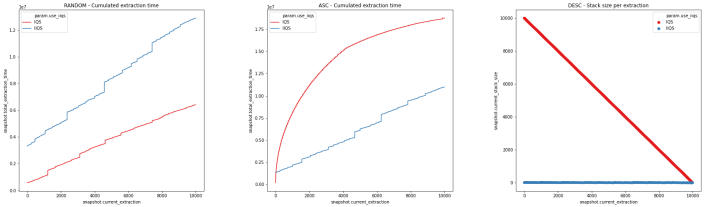


Figure: Base IQS and IIQS benchmark.

Base benchmark

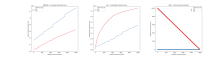


Figure: Base IQS and IIQS benchmark.

1. Podemos observar que ambos algoritmos se comportan de manera similar cuando la entrada es una secuencia ordenada de manera aleatoria.
2. No a su vez cuando los distintos tipos de entrada cambian a secuencias ordenadas. El preordenamiento de estas secuencias afecta de manera negativa la ejecución de IQS.
3. Si bien el fundamento de ambos algoritmos es exactamente el mismo, la implementación de una evaluacion para el cambio de estrategia de selección de pivotes permite que IIQS mantenga su complejidad temporal.
4. Antes de comenzar con los experimentos necesitamos asegurarnos que nuestra implementación de IQS no caiga en casos donde este no pueda responder.
5. Ya que la implementación no especifica el mecanismo de particionado, cambiamos nuestra versión de el algoritmo de particionado a una versión compatible con secuencias repetidas.

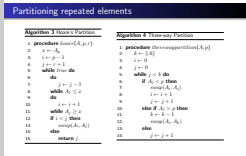
Partitioning repeated elements

Algorithm 3 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
```

Algorithm 4 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
```

1. Para poder soportar el ordenamiento de secuencias repetidas, necesitamos cambiar la implementación utilizada durante el trabajo original de IIQS (la implementación de partición de Hoare para secuencias) por la implementación del problema de la bandera danesa (para repetidos).
2. Habiendo hecho eso, la siguiente pregunta es si este cambio afecta el comportamiento de IQS de alguna manera. Ya que teóricamente, para efectos de diseño del algoritmo, si bien su complejidad asintótica permanece de igual manera, la forma en la cual los elementos se mueven dentro del arreglo es diferente.

Partitioning repeated elements

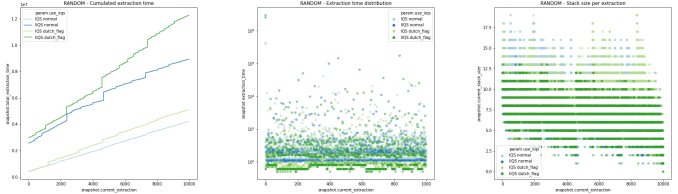


Figure: Base IQS and IIQS benchmark for a random sequence with 1×10^5 elements.

Partitioning repeated elements

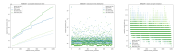


Figure: Base IQS and IIQS benchmark for a random sequence with 1×10^5 elements.

1. Podemos observar que la implementación de una estrategia de particionado de tres vías no altera la complejidad para el caso aleatorio, mostrando el mismo comportamiento en ambos algoritmos.

Ascending sequences

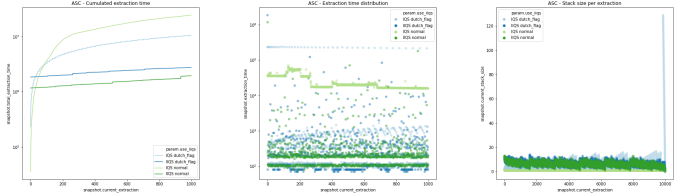


Figure: Base IQS and IIQS benchmark for a ascending sequence with 1×10^5 elements.

Ascending sequences
Figure: Base IQS and IIQS benchmark for a ascending sequence with 1×10^5 elements.

1. Podemos apreciar también que para el caso de secuencias ascendentes existe un degrade progresivo de la ejecución dado al pobre uso que se le da a la pila.
2. Esto repercute especialmente al final de la ejecución, debido al ordenamiento hecho por la versión ajustada de nuestro particionador.

Descending sequences

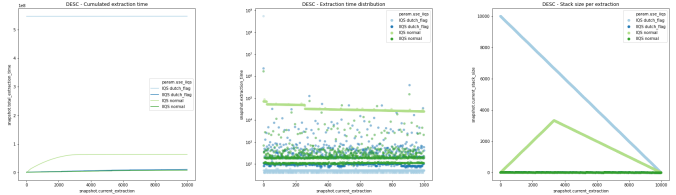


Figure: Base IQS and IIQS benchmark for a descending sequence with 1×10^5 elements.

1. Finalmente este fenomeno repercute de manera negativa cuando la secuencia es decreciente, ya que todos los pivotes son almacenados en la pila.
2. A este punto, podemos decir que la implementación del algoritmo de particionado de tres vias, si bien realiza un cambio en el comportamiento de IQS, este sigue manifestando su misma complejidad y falencias.

Repeating elements

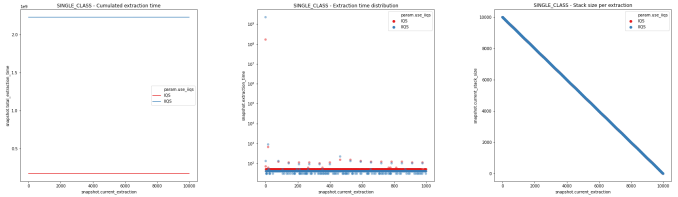


Figure: Base IQS and IIQS benchmark for a sequence with 1×10^5 repeated elements.

Repeating elements

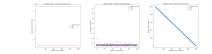


Figure: Base IQS and IIQS benchmark for a sequence with 1×10^5 repeated elements.

1. Cuando todos los elementos son iguales, IIQS se comporta igual que IQS en su peor caso.

Class impact

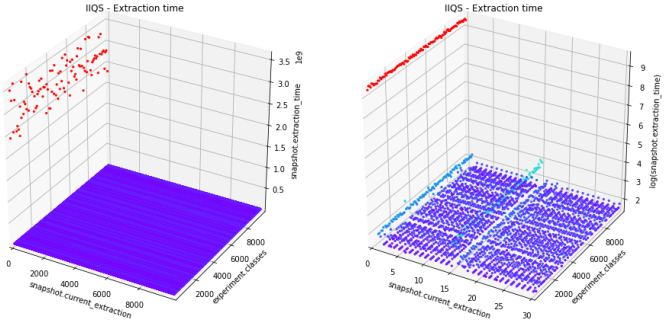


Figure: Impact of classes on IIQS execution.

Class impact

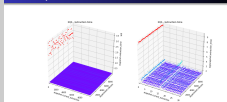


Figure: Impact of classes on IIQS execution.

1. Al variar el numero de clases dentro de nuestra secuencia de entrada, nos damos cuenta que existe una diferencia de sobre 8 órdenes de magnitud.
2. Sin embargo, esta solo ocurre cuando hay una clase presente. Esto es porque en el momento que hay una segunda clase, si la clase mayoritaria no cuenta con el 50 porciento de los datos, entonces ese segmento, causante del fenómeno solo representa la mitad de los elementos. En cuyo caso, de pasar por un proceso de particionamiento, ya reduce la complejidad del problema a la mitad.
3. Esta es la primer indicio que si podemos eliminar esa clase mayoritaria o bien controlar la distribución de los elementos, podemos evitar este nuevo peor caso.

Pivot bias impact

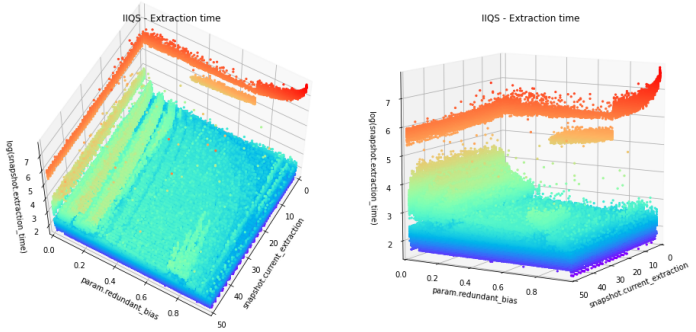


Figure: Impact of classes on IIQS execution.

Pivot bias impact

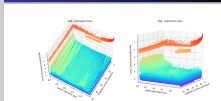


Figure: Impact of classes on IIQS execution.

1. El sesgo de la elección del pivote a devolver durante la etapa de partición de tres vias adolece del mismo problema del peor caso de IQS. Este es, un impacto directo sobre como los elementos son rescatados y de como la pila es usada.

Random noise impact

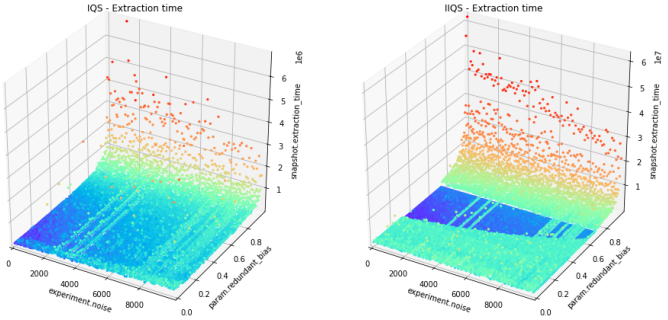


Figure: Impact of classes on IQS and IIQS execution.

Random noise impact

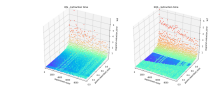


Figure: Impact of classes on IQS and IIQS execution.

1. Esta es la primer indicio que si podemos eliminar esa clase mayoritaria o bien controlar la distribución de los elementos, podemos evitar este nuevo peor caso.

IIQS parameters impact

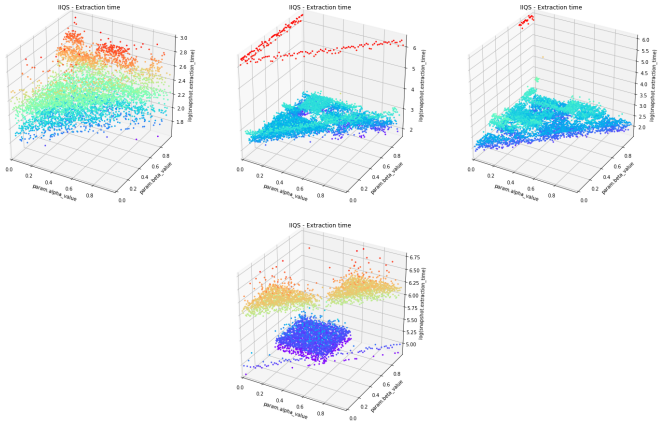
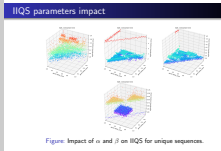


Figure: Impact of α and β on IIQS for unique sequences.



1. Por otro lado, no vemos relación alguna entre los comportamientos esperados para el caso de secuencias con elementos únicos a con una única secuencia. Es mas, incluso el mejor tiempo de ejecucucion es comparable a los peores casos vistos de elección de los parámetros alpha y beta.
2. Esto nos advierte inmediatamente que la posibilidad de poder ejecutar mejoras a nivel de los parámetros de IIQS no existe ya que no tiene mayor papel.
3. Entonces, sabemos que el problema de este caso es el número de clases.
4. Sabemos también que si logramos hacer que la clase mayoritaria ocupe menos del 50 porciento, volvemos a un caso esperado de ejecución.
5. Sabemos también que si logramos hacer que la clase mayoritaria ocupe menos del 50 porciento, volvemos a un caso esperado de ejecución.

Idea pt. 1

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

Idea pt. 1
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 the mapping induced by counting sort, C_{in}, as a intermediate representation of our original sequence S_{in}.

1. Sabemos que la familia de algoritmos derivados directamente de iqs son algoritmos de ordenamiento adaptivo. Esto quiere decir que su tiempo de ejecución va en función de la entrada.
2. Entonces, la idea principal es convertir directamente la entrada por medio de un proceso auxiliar, de modo que sea compatible con IQS/IIQS.

Idea pt. 2

Let us define $S_{repeated(\gamma)}$ as a sequence $(s_0, s_1, ..., s_{n-1}, s_n)$ 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:

$$roll \colon S_{repeated(\gamma)} \to C_\gamma$$
$$(s_0, s_1, ..., s_{n-1}, s_n) \mapsto (n).$$

And analogously:

$$unroll \colon C_\gamma \to S_{repeated(\gamma)}$$
$$(n) \mapsto (s_0, s_1, ..., s_{n-1}, s_n).$$

Idea pt. 2
Let us define $S_{repeated(\gamma)}$ as a sequence $(s_0, s_1, ..., s_{n-1}, s_n)$ 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:
$roll \colon S_{repeated(\gamma)} \to C_\gamma$
$(s_0, s_1, ..., s_{n-1}, s_n) \mapsto (n).$
And analogously:
$unroll \colon C_\gamma \to S_{repeated(\gamma)}$
$(n) \mapsto (s_0, s_1, ..., s_{n-1}, s_n).$

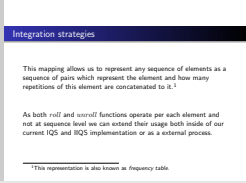
1. Estas dos funciones roll y unroll que modelan funcionan a nivel de elemento.
2. Por tanto podemos extender su definición a secuencias ordenadas concatenando estos elementos S sub gamma.

Integration strategies

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.[1]

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 a external process.

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



1. Lo que estamos haciendo, es definir de manera modular la implementación de un ordenamiento por conteo o una tabla de frecuencias.
2. Esto nos permite agregar esta reducción como un proceso externo o interno a nuestro algoritmo objetivo

High-level external implementation

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

Algorithm 5 External reduction

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

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 a external process.

1. Nuestra estrategia se reduce entonces a ejecutar una reducción de la entrada mapeandola a un conjunto de pares (llave, valor), ordenar las llaves y posteriormente al momento de retornar los elementos, consumir estos por medio de nuestra función de mapeo.

High-level external implementation

We asume that roll operation as it is an instance of counting sort it takes $O(n)$ time and 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}.

In worst case, $l = \|S_{in}\|$, which is suboptimal in relation to IQS.

High-level external implementation
We asume that roll operation as it is an instance of counting sort it takes $O(n)$ time and 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}.
In worst case, $l = \|S_{in}\|$, which is suboptimal in relation to IQS.

1. Este overhead introducido por el proceso externo, si bien mantiene la complejidad temporal, no conserva la complejidad espacial, ya que en peor caso, necesitamos n extra espacio para poder ejecutar el ordenamiento debido a una tabla de frecuencias con puros unos.
2. Resulta obvio que no es posible aplicar esta estrategia a IQS ya que no tenemos manera de controlar la distribución de las llaves de los retornados por la tabla de frecuencia. De modo que esta implementación no es factible.

High-level internal implementation

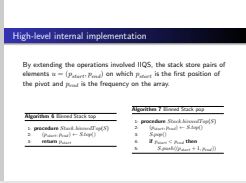
By extending the operations involved IIQS, the stack store pairs of elements $u = (p_{start}, p_{end})$ on which p_{start} is the first position of the pivot and p_{end} is the frequency on the array.

Algorithm 6 Binned Stack top

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

Algorithm 7 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))
```

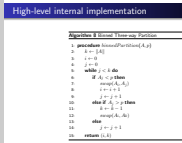



1. Lo primero que hay que hacer para implementar esta extensión de manera interna a IIQS, es darle la capacidad a la pila de almacenar rangos en vez de un solo elemento.

High-level internal implementation

Algorithm 8 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_j, A_k)
13:         else
14:             j ← j + 1
15:     return (i, k)
```

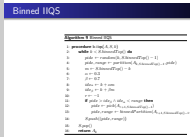



1. Posteriormente, nuestro particionador debe soportar el la devolución de rangos de elementos en vez de solamente un elemento. Esto elimina el sesgo de particionado visto anteriormente.

Binned IIQS

Algorithm 9 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 < range 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
```

1. De esta manera sin mayores modificaciones al algoritmo original, obtenemos una versión de IIQS que integra la generación de una tabla de frequencias para poder reducir los casos de entrada al caso esperado de la implementación original.

BIIQS overview

As both Algorithms 6 and 7 have $O(1)$ worst-case complexity and Algorithm 8 maintains $O(n)$ complexity on which n dennotes 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.

BIIQS overview
As both Algorithms 6 and 7 have $O(1)$ worst-case complexity and Algorithm 8 maintains $O(n)$ complexity on which n dennotes 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.

1. Esta versión de IIQS preserva el misma orden de complejidad asintótica tanto temporal como espacial que su versión original.

Base benchmark

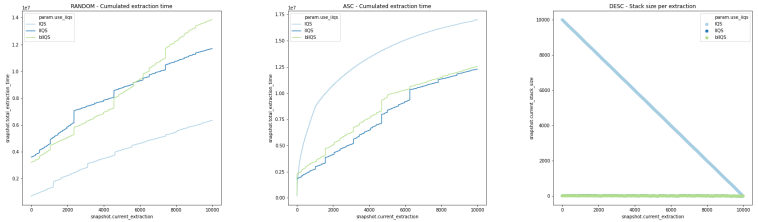


Figure: BIIQS benchmark.

Base benchmark

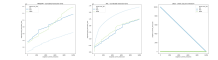


Figure: BIIQS benchmark.

1. Podemos observar que en todas las instancias originales del problema, biiqs entrega los mismos resultados que su versión original.

Repeating elements

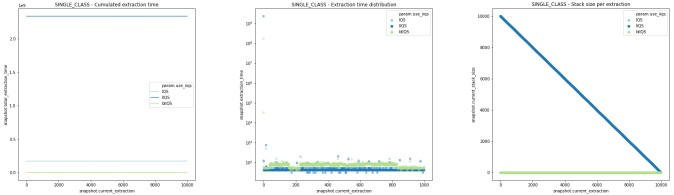


Figure: BIIQS benchmark for a sequence with 1×10^5 repeated elements.

Repeating elements

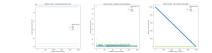


Figure: BIIQS benchmark for a sequence with 1×10^5 repeated elements.

1. Sin embargo, muestra el comportamiento esperado cuando recibe como entrada elementos repetidos, incluso en el peor caso con solo una clase a lo largo de este.

Class impact

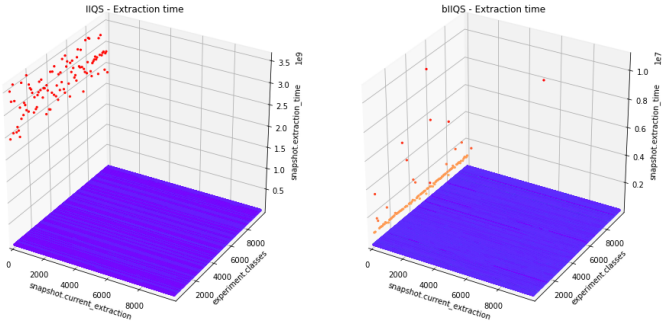


Figure: Impact of classes on BIIQS execution.

Class impact

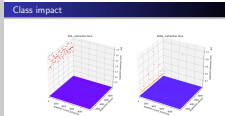


Figure: Impact of classes on BIIQS execution.

1. adicionalmente, podemos ver una baja importante en el orden de magnitud de la primera extracción del mismo sobre la cual la cantidad de clases no afecta su tiempo de ejecucuón de ninguna forma.

Bias impact

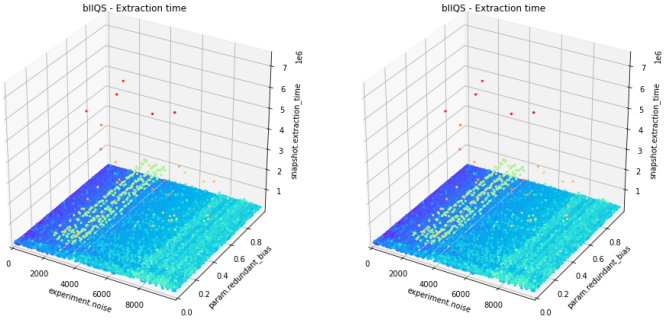


Figure: Impact of random noise and partition bias on IIQS execution.

Bias impact

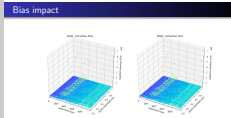


Figure: Impact of random noise and partition bias on IIQS execution.

1. Una gran diferencia por sobre la implementación original, es que ahora ni el sesgo de partición ni el ruido aleatorio cambian el tiempo de ejecución.
2. Incluso para las combinaciones donde el fallo de cache es constante, se mantiene dentro de los márgenes aceptables de ejecución.
3. De hecho, ni siquiera fue necesario utilizar una escala logaritmica.

summary

- We have presented bIIQS, an alternative to IIQS which provides support for repeated elements on its input sequence, in optimal time and space.
- We have provided with a starter framework to replicate the process by using an experimental algorithmics approach.
- We delivered an experimental analysis of those three algorithms along with a portable implementation.

summary
- We have presented bIIQS, an alternative to IIQS which provides support for repeated elements on its input sequence, in optimal time and space.
- We have provided with a starter framework to replicate the process by using an experimental algorithmics approach.
- We delivered an experimental analysis of those three algorithms along with a portable implementation.

summary

- Constrain timing test by using specialized hardware implementations.
- Develop a full theoretical analysis of both IIQS and bIIQS.

summary
- Constrain timing test by using specialized hardware implementations.
- Develop a full theoretical analysis of both IIQS and bIIQS.

Degree Project

Extending IIQS to support sequences with repeated elements

Erik Regla

Universidad de Talca

July 28, 2020

Degree Project
Extending IIQS to support sequences with repeated elements
Erik Regla
Universidad de Talca
July 28, 2020