Worst-case Optimal Incremental Sorting ⋆

Erik Regla, Rodrigo Paredes
Departamento de Ciencias de la Computación
Universidad de Talca
Curicó, Chile
Email: eregla09@alumnos.utalca.cl, raparede@utalca.cl

Abstract

We present an online algorithm to, given a fixed array A, retrieve its k smallest elements in optimum time for the worst case, that presents fast response in practice. For this sake, we devise an introspective version of the Incremental Quicksort (IQS) *algorithm, which controls the size of the auxiliary stack of the IQS algorithm via an introspective criteria.*

1. Introduction

"Ideally, quicksort partitions sequences of size N into sequences of size approximately $\frac{N}{2}$, those sequences in sequences of size $\frac{N}{4}$, and so on, implicitly producing a tree of subproblems whose depth is approximately $\log_2 N$*. The cuadratic-time problem arises on sequences that cause many unequal partitions, resulting the problem tree depth growing linearly rather than logarithmically"* [1].

This problem is known to appear in many partition based[1] algorithms and *incremental quicksort* (IQS) [2] is no exception to the rule when the data renders the random pivot selection useless.

Blum et. al [3] *pick* algorithm works as a median selection method that performs on lineal time in worst case but its stability comes at the cost of an slower performance for the average case compared to *quickselect*. This served as a stepping stone for *introsort* [1], which guarantees the worst-case $O(n \log_2(n))$ running time without a loss of performance on the average case by making *quicksort* "aware" of when begins to hit worst case by counting the recursion depth and limiting its own execution.

IQS is currently one of the answers to the online incremental sorting problem in optimal time, having applications as a subroutine in algorithms like *Kruskal*'s minimum spanning tree algorithm and even used to implement a *heap* [2]. But, depending on the data distribution, it can hit the worst-case when the pivot selection and the partition stage does not contribute to divide the problem. So IQS as is, it is not suitable for critical applications because there is no guarantee that the data allows optimal time execution.

Hence, this raises another research opportunity: devise an online algorithm to, given a fixed array A, retrieve its k smallest elements in optimum time for the worst case, that presents fast response in practice.

The rest of this paper is organised as follows: Section 2 shows the basic IQS algorithm. Section 3 briefly explains our proposal to solve the worst-case IQS issue. Section 4 introduces known solutions for the median selection problem as well the *stack problem* on IQS, an elemental problem derived from partition-based algorithms. Section 5 explains how an introspective step is implemented on IQS in order to control the recursion depth. Section 6 shows the experimental results made to test our algorithm. Section 7 presents our conclusions and a summary of this paper.

2. Incremental Quicksort

In [2], the IQS algorithm was introduced. Given an unsorted set A of size n, IQS retrieves its k smallest elements in increasing order for any k, that is unknown to the algorithm, in optimal expected time $O(n + k \log_2 k)$. The IQS algorithm can be seen as a nonrecursive variant of *Quicksort* [4].

The work performed the first time we process an array A using IQS is the same as that performed by

⋆ *Partially funded by FONDECYT grant 1131044, Chile.*

[1] As well as *Divide and Conquer* algorithms

	0	1	2	3	4	5	6	7	8	9	10	11
A:	54	12	34	89	67	5	2	3	7	68	23	∞
S:	11											

	0	1	2	3	4	5	6	7	8	9	10	11
A:	12	5	2	3	7	**23**	34	54	89	67	68	∞
S:	11	5										

	0	1	2	3	4	5	6	7	8	9	10	11
A:	5	2	3	**7**	12	**23**	34	54	89	67	68	∞
S:	11	5	3									

	0	1	2	3	4	5	6	7	8	9	10	11
A:	2	**3**	5	**7**	12	**23**	34	54	89	67	68	∞
S:	11	5	3	1								

	0	1	2	3	4	5	6	7	8	9	10	11
A:	**2**	**3**	5	**7**	12	**23**	34	54	89	67	68	∞
S:	11	5	3	1	0							

Figure 1. Example of a normal IQS execution retrieving the first element. Under normal conditions, the pivot selection produces a division of the problem in half, then the stack grows at a logarithmic rate.

Quickselect [5] to find the smallest element of $A_{0,n-1}$ (while this cost is quadratic in the worst case, IQS was focused on expected costs, in which case the operation takes linear time). The first row of Figure 1 depicts the original array A and the second and following rows, how A is modified during the execution of *Quickselect* using the last element of the current partition as the pivot, but it could be any other element. The key point is that *Quickselect* generates a sequence of pivots in decreasing order and we can use this pivot sequence in the following operations. In the figure, we have written in **bold** the elements used as pivots and stored their positions in an auxiliary stack S. Note that S is initialized with the size of the array. Once the execution ends, we report the first element of the array.

If we need to find the second smallest element, we could call *Quickselect* on $A[1, n-1]$. Instead, IQS checks if we have the position 1 on top of S, that is, if $S.top() = 1$, in which case we are done as $A[1]$ *is* the second smallest element. Else, it calls *Quickselect* on

```
procedure IQS($A, S, k$)
    while $k < S.top()$ do
        $pidx \leftarrow random(k, S.top() - 1)$
        $pidx \leftarrow partition(A_{k,S.top()-1}, pidx)$
        $S.push(pidx)$
    end while
    $S.pop()$
    return $A_k$
end procedure
```

Figure 2. Incremental Quicksort. Stack S is initialized as $S \leftarrow |A|$.

$A[1, S.top() - 1]$ and pushes the new resulting pivots onto S. We can use IQS to retrieve the next elements, if we need to. In order to preserve the IQS invariant (either the minimum is the element on top of S or it is within the leftmost chunk) from the first time, we initialize S containing the position of the fictitious pivot ∞ after the end of A. The experimental results [2] showed that IQS is up to 4 times faster than the classical alternative that uses binary heaps. Figure 2 depicts the algorithm IQS.

3. Sketch of our proposal

In order to ensure worst case $O(n + k\log_2(k))$ running time we need skip or fix recursive steps that do not contribute solving the problem or to reduce it before the algorithm deteriorates itself to worst case due to the nature of the *partition* algorithm and the distribution of the input data.

Those steps occure when the pivot used to partition the array is near the beginning or the end of the segment. But, we do not know whose position the pivot falls until it is placed on A in linear time. Therefore, storing these pivots positions (close to segment start or end) on the stack S does not help to reduce the problem. In fact, if after the partitioning the pivot is placed near the end of the segment, the recursion must follow in the left partition which is almost as large as was before the partitioning, hitting the worst case. On the other hand, if the pivot reach the beginning of the segment, even though this could mean than future IQS callings have extra work (when the algorithm processes the segment at the right of the pivot), we know that the left partition is small and than after few iterations we will find out the desired element.

Since the stack S content controls how recursion behaves on each query for the current minimum, we can ensure that the problem is divided on each iteration by improving the quality of the pivots stored on S. Therefore $O(n + k \log_2(k))$ worst-case running time regardless of data distribution in A.

4. Known solutions and issues

4.1.Approximated median selection

The *pick* algorithm by Blum et al [3] allows us to select an approximated median in between the 30-th and 70-th percentile of A in linear time [6], by computing iteratively the medians of subsets of A.

When using this element as a new pivot on the partition process, we know that in the worst case, the recursive call will operate over a $\frac{7}{10}$ of the current segment at most. Then, it is safe to assume that we eliminate the worst case iteration step by replacing the random pivot selection with an approximated median selection algorithm (the pick algorithm). As we need only $O(\log n)$ recursive calls to find an element, we switch to the pick algorithm when more than $O(\log n)$ calls are invoked using the random pivot selection.

Musser noticed that if we replace *quicksort* random pivot selection with the *pick* algorithm, the worst-case complexity is avoided [1], but in practice its running time is not feasible for all cases. If we were to implement IQS that way, we have to afford not only the n time due to the partition stage, now also an $O(n)$ for the median selection computation (which is about $10n$ in practice), instead of the random selection that is done in constant time. Therefore the running time is not feasible in practice due to its high cost ($11n$) for the first elements in average case.

4.2.The stack size problem

One of the benefits of using IQS for the online partial sorting problem is that its space complexity is at most $O(\log_2(n))$. In fact, IQS uses that space to store the pivot positions inside the stack S, which ideally are distributed at an exponential rate (see Figure 1).

Then another issue besides time consumption arises: the stack growth. In most cases, due to the pivot distribution the size of S is logarithmic on n, assuming that the pivots stored belong to a position near the median of current segment. When this does not occur, the stack overflows (that is, grows superlogarithmically) because starts storing all pivots obtained. This is illustrated in Figure 3, when each time the algorithm choses the last element of the segment and it turns out to be the largest in the segment, and later is experimentally evaluated in Figure 5c. When the pivot is always at the left size, the stack does not storage any information at all, but in the next call has to process the partition at its right, which has $n - 1$ elements. On the other hand, if the pivot is at the right extreme, then the next call is again executed for almost the whole array length adding one extra element on S. In worst case, IQS will iterate n times using extra n space to answer a query overflowing S as shown in Figure 3.

	0	1	2	3	4	5	6	7	8	9	10	11
A:	2	3	5	7	12	23	34	54	67	68	89	∞
S:	11											

	0	1	2	3	4	5	6	7	8	9	10	11
A:	2	3	5	7	12	23	34	54	67	68	**89**	∞
S:	11	10										

	0	1	2	3	4	5	6	7	8	9	10	11
A:	2	3	5	7	12	23	34	54	67	**68**	**89**	∞
S:	11	10	9									

	0	1	2	3	4	5	6	7	8	9	10	11
A:	2	3	5	7	12	23	34	54	**67**	**68**	**89**	∞
S:	11	10	9	8								

	0	1	2	3	4	5	6	7	8	9	10	11
A:	2	3	5	7	12	23	43	**54**	**67**	**68**	**89**	∞
S:	11	10	9	8	7							

⋮

	0	1	2	3	4	5	6	7	8	9	10	11
A:	**2**	**3**	**5**	**7**	**12**	**23**	**43**	**54**	**67**	**68**	**89**	∞
S:	11	10	9	8	7	6	5	4	3	2	1	0

Figure 3. Example of a normal IQS execution when retrieving the first element on a sorted array. Due to the failed pivot selection every pivot is stored on S overflowing the stack ($log(k)$), because of that every scan is made in lineal time.

5. Introspection on IQS

As mentioned in the previous section, we need to store pivots in S at a exponential rate, but also we need to answer each single query as fast as we can (which is not possible by just replacing random pivot selection with median of medians as explained in Section 4.1). In average case, $\log_2(n)$ recursive steps are expected to find the k-th smallest element on A. One could argue that will suffice to compare the number of iterations needed to find the current minimum with it expected value to prevent the worst case, but on IQS, the recursion depth is not affected by the query itself, but by the distribution of the pivots stored on S.

Let m be the size of the current segment, that is, $m = S.top() - k$. In order to control the size of S, we accept a pivot if its resulting position p belongs to the range $[k + \alpha m, S.top() - \alpha m]$, where $\alpha = 0.3$ (according to Blum et al algorithm guarantee). We can identify two cases of pivot rejection:

1) $p < k + \alpha m$: The pivot does not contribute to amortise next calls but helps to answer the query. Then this pivot is stored after storing an extra pivot obtained via Blum's algorithm.

2) $p > S.top() - \alpha m$: The pivot does not contribute to amortise next calls neither to answer the query, and only it is stored the pivot obtained via Blum's algorithm.

Note that in both cases, the position of the rejected pivot is used to reduce the range for Blum's algorithm.

With this criteria we can filter what items are stored on the stack and simultaneously make sure that even in the worst case the stack grows slowly and steady. This way we can control the execution steps taken by IQS by controlling at which rate that pivots are stored in S instead of controlling the number of iterations required to find a single minimum (as Musser proposes for QuickSort). At the same time, all those pivots that can answer the query faster are used and loaded onto the stack, leading to a faster execution on the first queries in the average case. Figure 4 shows the algorithm IIQS.

```
procedure IIQS($A, S, k$)
    while $k < S.top()$ do
        $pidx \leftarrow random(k, S.top() - 1)$
        $pidx \leftarrow partition(A_{k,S.top()-1}, pidx)$
        $m \leftarrow S.top() - k$
        $\alpha \leftarrow 0.3$
        $r \leftarrow -1$
        if $pidx < k + \alpha m$ then
            $r \leftarrow pidx$
            $pidx \leftarrow pick(A_{r+1,S.top()-1})$
            $pidx \leftarrow partition(A_{r+1,S.top()-1},$
                                  $pidx)$
        else if $pidx > S.top() - \alpha m$ then
            $r \leftarrow pidx$
            $pidx \leftarrow pick(A_{k,pidx})$
            $pidx \leftarrow partition(A_{k,r}, pidx)$
            $r \leftarrow -1$
        end if
        S.push($pidx$)
        if $r > -1$ then
            S.push($r$)
        end if
    end while
    S.pop()
    return $A_k$
end procedure
```

Figure 4. Introspective Incremental Quicksort.

5.1.Worst case complexity

Assuming that we select the pivots using the pick algorithm and the recursion ends when the size of the segment is one, the recurrence for the worst case of the first call of Introspective IQS (IIQS) is:

$$T(n) = \overbrace{T\left(\frac{7}{10}n\right)}^{\text{next iteration}} + \overbrace{n-1}^{\text{partition cost}} + \overbrace{11n}^{\substack{\text{introspective}\\\text{step cost}}} \tag{1}$$
$$T(1) = 0$$

The first term in (1) corresponds to the next recursive call (remember that the pick algorithm ensures that the larger partition has at most 70% of the current segment) [6][3]. The term $n - 1$ corresponds to the cost of the execution of *partition* and the checking if the next introspective step is to be done. The final term corresponds to the cost of executing Blum's algorithm and a *partition* step. Collecting terms of (1),

$$T(n) = T\left(\frac{7}{10}n\right) + 12n - 1$$

Using $n = \gamma^k$, where $\gamma = \frac{10}{7}$

$$T(\gamma^k) = T\left(\gamma^{k-1}\right) + 12\gamma^k - 1$$

Using telescopic series we obtain

$$T(\gamma^k) = 12\sum_{i=1}^{k}\gamma^i - k = 12\frac{\gamma(\gamma^k - 1)}{\gamma - 1} - \log_\gamma n$$

Reverting changes to $T(n)$ and evaluating γ

$$T(n) = 40(n-1) - log_{\frac{10}{7}} n \tag{2}$$

$$T(n) = O(n)$$

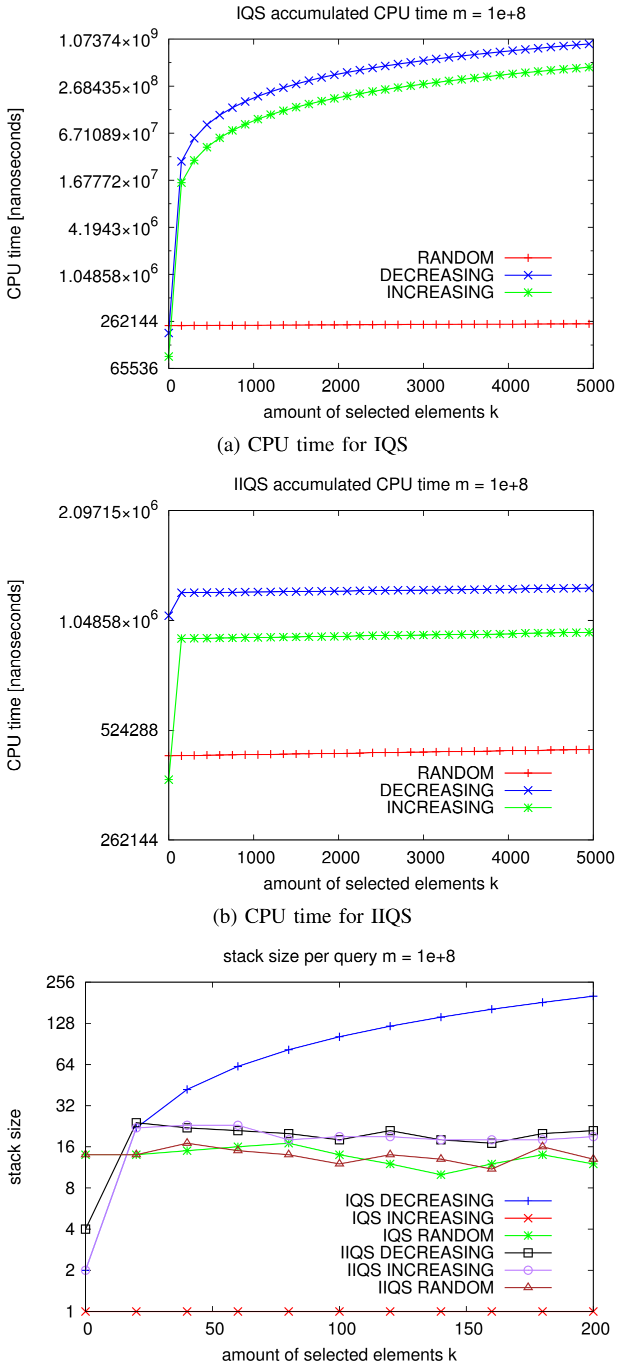
Hence, the time complexity for the worst case of the first call of IIQS (2) shows a lineal behaviour in comparison to IQS worst case which is $O(n^2)$. Also the pivots are exponentially stored on S, leading to an amortised cost $k\log_2(k)$ on all its future calls.

6. Experimental results

The experiments performed for both IQS and IIQS show how the stack size affects the overall performance of the algorithm and the dependency of next calls on the contents of S. All tests were performed forcing the worst pivot selection for the random version fixing it to the first element retrieved. There was no significant difference on performance when using the first or the last element unless the partition algorithm changes. Also, repeating the random selection process two or more times before changing to median of medians did not improve its overall performance and only contributes to increase stack load.

As can be seen from Figures 5a and 5b, in the case of random input, IQS is almost a 40% faster than IIQS, but in the worst case input, the IQS performance falls down dramatically, as expected. In the range tested, the increasing of time of IQS reaches almost 10,000 times in the worst case. On the other hand, IIQS also degrades its performance with worst case input, but only three times.

This is in agreement with the Figure 5c that shows the size of the stack S. For IIQS, both random input and worst case input has similar behaviour with respect to the size of S due to the introspective selection of pivots, thus preserving execution time. Once again, random input offer the best behaviour, but in the worst case, the stack is just two times bigger than with random input. The max size observed on the stack on introspective versions is about 24 pivot positions and its behaviour is constant among executions regardless of number of minima queried or its data distribution, similar to the average case in IQS. The amortised cost



(a) CPU time for IQS

(b) CPU time for IIQS

(c) Stack size of IQS and IIQS

Figure 5. Performance comparison between IQS and IIQS as a function of the amount of searched elements k. Note the logscales in the plots.

for each call plus the cost of the first one is lower than IQS after the 5th iteration. On the other hand, in the worst case we verify two extremely anomalous behaviour for IQS. If the input is increasing, the stack only has the fictitious pivot, as each call to partition delivers the minimum element. Whilst, if the input is decreasing, the stack stores a linear amount of pivots.

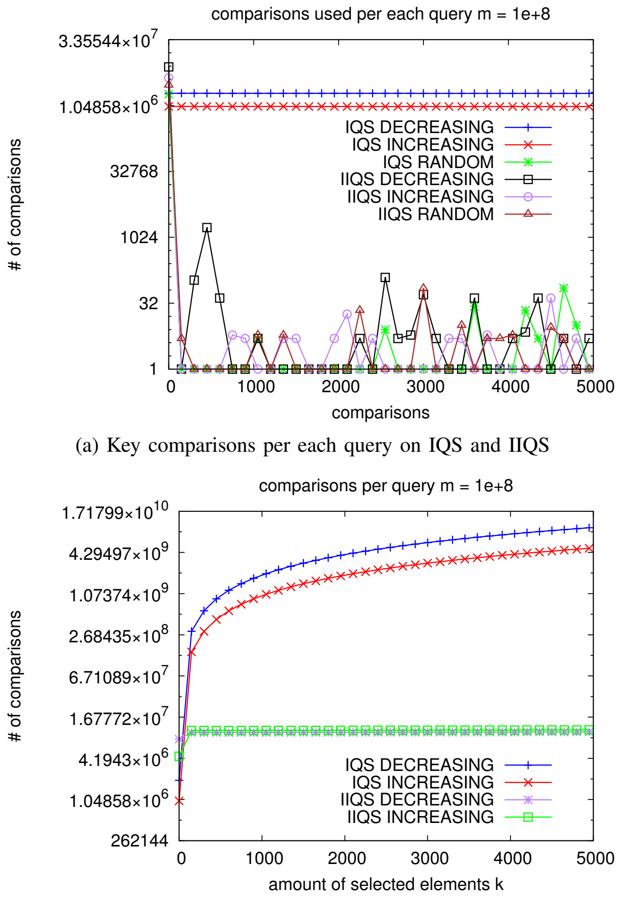
Figure 6a shows the number of key comparisons for IQS and IIQS. As expected, for $k = 1$, all the variants needs a similar amount of key comparison, which is linear in the size of the array. But, in the worst case, IQS continuously needs linear timer per minimum retrieved and IIQS needs marginal work to retrieve the following minima. Finally, Figure 6b shows the accumulated number of key comparisons. As expected, in the worst case IIQS has a very mild increasing in the accumulated work, but IQS explodes.

Experiments for worst case using $m > 10^8$ and for $k > 5000$ were not performed because of the time bounds for worst case.

7. Conclusion

We have presented IIQS, an introspective version of IQS [2] to, given a fixed set, retrieve its k smallest elements in optimal time even for the worst case when k is unknown before hand. IIQS in practice has the same complexity than IQS plus a little more time (almost 40% for random inputs and only three times in the worst case) due to the introspective step. To do that, IIQS carefully chose the elements that are pushed onto the auxiliary stack S in order to balance future executions and to ensure a logarithmic growth of S. We have proven IIQS competitiveness against the current version of IQS.

Future work considers to analyse the worst case of IIQS for several minima extraction, not only for the minimum. Also, we plan to compare our IIQS with other worst case minima selection algorithm, such as using a binary heap to retrieve the minimum of a set.



(a) Key comparisons per each query on IQS and IIQS

(b) Cumulated key comparisons for IQS and IIQS

Figure 6. Key comparisons between IQS and IIQS as a function of the amount of searched elements and per query k. Note the logscales in the plots.

References

[1] D. Musser, “Introspective sorting and selection algorithms,” *Software Practice and Experience*, vol. 27, pp. 983–993, 1997.

[2] G. Navarro and R. Paredes, “On sorting, heaps, and minimum spanning trees,” *Algorithmica*, vol. 57, no. 4, pp. 585–620, 2010, doi:10.1007/s00453-010-9400-6.

[3] M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, Robert, and E. Tarjan, “Time bounds for selection,” *JCSS*, pp. 448–461, 1973.

[4] C. A. R. Hoare, “Quicksort,” *Computer Journal*, vol. 5, no. 1, pp. 10–15, 1962.

[5] ——, “Algorithm 65 (FIND),” *Comm. of the ACM*, vol. 4, no. 7, pp. 321–322, 1961.

[6] D. E. Knuth, *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1998.