Optimal Incremental Sorting *

Rodrigo Paredes † Gonzalo Navarro †

Abstract

Let A be a set of size m. Obtaining the first $k \leq m$ elements of A in ascending order can be done in optimal $O(m+k \log k)$ time. We present an algorithm (online on k) which incrementally gives the next smallest element of the set, so that the first k elements are obtained in optimal time for any k. We also give a practical algorithm with the same complexity on average, which improves in practice the existing online algorithm. As a direct application, we use our technique to implement Kruskal's Minimum Spanning Tree algorithm, where our solution is competitive with the best current implementations. We finally show that our technique can be applied to several other problems, such as obtaining an interval of the sorted sequence and implementing heaps.

1 Introduction

There are cases where we need to obtain the smallest elements from a fixed set without knowing how many elements we will end up needing. Prominent examples are Kruskal's Minimum Spanning Tree (MST) algorithm [14] or ranking by Web search engines [1]. Given a graph, Kruskal's MST algorithm processes the edges one by one, from smaller to larger, until forming the MST. At this point, remaining edges are not considered. On the other hand, Web search engines display a very small sorted subset of the most relevant documents among all those satisfying the query. Later, if the user wants more results, the search engine displays the next group of most relevant documents, and so on. In both cases, we could first sort the whole set and later return the desired objects, but obviously this is more work than necessary.

This problem can be called *Incremental Sorting*. It can be stated as follows: Given set A of m numbers, output the elements of A from smallest to largest, so that the process can be stopped after k elements have been output, for any k that is unknown to the algorithm. Therefore, *Incremental Sorting* is the online version of the *Partial Sorting* problem: Given set A of m numbers and integer $k \leq m$, output the smallest k elements of A in ascending order.

Partial Sorting can be easily solved by first finding p, the k-th smallest element of A, using $O(m)$ time Select algorithm [2], and then collecting and sorting the elements smaller than p. We call SelectSort this algorithm. Its complexity, $O(m + k \log k)$, is optimal under the comparison model, as there are $m^{\underline{k}} = m!/(m-k)!$ possible answers and $\log(m^{\underline{k}}) = \Omega(m + k \log k)$.

A practical version of the above, QuickSelectSort (**QSS**), uses QuickSelect [9] and QuickSort [10] as the selection and sorting algorithms, obtaining $O(m+k \log k)$ average complexity. Recently, it has been shown that selection and sorting can be interleaved. The result, PartialQuickSort (**PQS**), has the same average complexity but smaller constant terms [15].

To solve the online problem, we must select the smallest element, then the second one, and so on until the process finishes at some unknown value $k \in [0, m-1]$. One can do this by using Select to find each of the first k elements, which costs $O(km)$ overall. We can improve this by transforming A into a min-heap in time $O(m)$ [5], and then performing k extractions. This costs $O(m + k \log m)$ worst-case complexity. Note that $m + k \log m = O(m + k \log k)$, as they can differ only if $k = o(m^\alpha)$ for any $\alpha > 0$, and then m dominates $k \log m$. However, this scheme is much slower than the offline practical algorithm **PQS**. Then the quest for a practical online algorithm for partial sorting is raised.

In this paper we present the IncrementalSelect (**IS**) algorithm, which solves the online problem in optimal $O(m + k \log k)$ time. We also present IncrementalQuickSelect (**IQS**), a practical variant of **IS**, which is $O(m + k \log k)$ time on average. Our experimental results show that **IQS** is almost as efficient as its offline version **PQS**, and is faster in practice than alternative solutions.

As an application, we show how to use our algorithm to boost the performance of Kruskal's MST algorithm [14]. Given a graph $G(V, E)$, we compute its MST in $O(|E| + |V| \log^2 |V|)$ average time, which is optimal in medium or high density graphs. In practice, by using **IQS** we obtain an efficient MST implementation, which

*Supported in part by the Millennium Nucleus Center for Web Research, Grant P04-067-F, Mideplan, Chile.

†Center for Web Research, Dept. of Computer Science, University of Chile. Blanco Encalada 2120, Santiago, Chile. {raparede,gnavarro}@dcc.uchile.cl

is much faster than any other Kruskal's implementation we could program or find for any graph density. As a matter of fact, our Kruskal's version is faster than the Prim's algorithm [18], even as optimized by Moret and Shapiro [16], and also competitive against the best alternative implementations we could find [12, 13].

We finally show that our algorithm can be used to solve other basic problems, such as obtaining an incremental segment of the sorted sequence A, and implementing a heap data structure. The algorithm can obviously be used to find the largest elements instead of smallest.

2 Incremental sorting

In this section we describe **IQS** algorithm. At the end we show how it can be converted into its worst-case version **IS**. Essentially, to output the k smallest elements, **IQS** calls QUICKSELECT to find the smallest element on arrays $A[0, m-1]$, $A[1, m-1]$, ..., $A[k-1, m-1]$. This naturally leaves the k smallest elements sorted in $A[0, k-1]$. **IQS** avoids the $O(km)$ complexity by reusing the work among calls to QUICKSELECT.

Let us recall how QUICKSELECT works. Given an integer k, QUICKSELECT aims to find the k-th smallest element from a set A of m numbers. For this sake it chooses an object p, the pivot, and partitions A so that the elements lower than p are allocated to the left-side partition, and the others to the right-side one. After the partitioning, p is placed in its correct position, let idx be such place. So, if $idx = k$, QUICKSELECT returns p and finishes. Otherwise, if $k < idx$ it recursively processes the left partition, else the right partition (with a new $k \leftarrow k - idx - 1$).

Note that it is possible to reuse the work made by previous calls to QUICKSELECT. When we call QUICKSELECT on $A[1, m-1]$, a decreasing sequence of pivots has already been used to partially sort A since the previous invocation on $A[0, m-1]$. **IQS** manages this sequence of pivots to reuse previous work. Specifically, it uses a stack S of decreasing pivots that are relevant for the next calls to QUICKSELECT.

Fig. 1 shows how **IQS** searches for the smallest element of an array by using a stack initialized with a single value $m = 16$. To find the next minimum, we first check whether p, the top value in S, is the index of the element sought, in which case we pop and return it. Otherwise, because of previous partitionings, it holds that elements in $A[1, p-1]$ are smaller than all the rest, so we run QUICKSELECT on that portion of the array, pushing new pivots into S.

The algorithm is given in Fig. 2. Stack S is initialized as $S = \{|A|\}$. **IQS** receives the set A, the index idx ($= k - 1$) of the element sought (that is, we seek the smallest element in $A[idx, m-1]$), and the current stack S (with former pivot positions). First it checks whether the top element of S is the desired index idx, in which case it pops idx and returns $A[idx]$. Otherwise it chooses a random pivot index $pidx$ from $[idx, S.\text{top}()-1]$. Pivot $A[pidx]$ is used to partition $A[idx, S.\text{top}()-1]$. After the partitioning, the pivot has reached its final position $pidx'$, which is pushed in S. Finally, a recursive invocation continues the work on the left hand of the partition.

We remind that **partition**(A, $A[pidx]$, i, j) rearranges $A[i, j]$ and returns the new position $pidx'$ of the original element in $A[pidx]$, so that, in the rearranged array, all the elements smaller/larger than $A[pidx']$ appear before/after $pidx'$. Thus, pivot $A[pidx']$ is left at

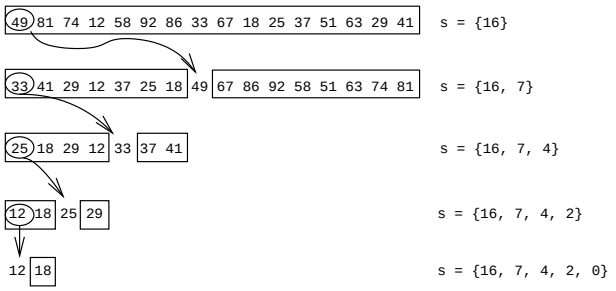


Figure 1: Example of how **IQS** finds the first element of an array. Each line corresponds to a new partition of a sub-array. Note that all the pivot positions are stored in stack S. In the example we use the first element as the pivot but it could be any other element.

IQS (Set A, Int idx, Stack S)

1. **If** $idx = S.\text{top}()$ **Then** $S.\text{pop}()$, **Return** $A[idx]$
2. $pidx \leftarrow$ **random**$[idx, S.\text{top}()-1]$
3. $pidx' \leftarrow$ **partition**(A, $A[pidx]$, idx, $S.\text{top}()-1$)
   // $A[0] \leq \ldots \leq A[idx-1] \leq A[idx, pidx'-1]$
   // $\leq A[pidx'] \leq A[pidx'+1, S.\text{top}()-1]$
   // $\leq A[S.\text{top}(), m-1]$
4. $S.\text{push}(pidx')$
5. **Return IQS**(A, idx, S)

Figure 2: INCREMENTALQUICKSELECT (**IQS**) algorithm. Stack S is initialized as $S \leftarrow \{|A|\}$. Both S and A are modified and rearranged during the algorithm. Note that the search range is limited to the array segment $A[idx, S.\text{top}()-1]$. Procedure **partition** returns the position of pivot $A[pidx]$ after the partition completes. Note that the tail recursion can be easily removed.

the correct position it would have in the sorted array $A[i, j]$. The next lemma shows that it is correct to search for the minimum just within $A[i, S.\text{top}() - 1]$, from which the correctness of **IQS** immediately follows.

LEMMA 2.1. *After i minima have been obtained in $A[0, i-1]$, (1) the pivot indices in S are decreasing bottom to top, (2) for each pivot position $p \neq m$ in S, $A[p]$ is not smaller than any element in $A[i, p-1]$ and not larger than any element in $A[p+1, m-1]$.*

Proof. Initially this holds since $i = 0$ and $S = \{m\}$. Assume this is valid before pushing p, when p' was the top of the stack. Since the pivot was chosen from $A[i, p'-1]$ and left at some position $i \leq p \leq p'-1$ after partitioning, property (1) is guaranteed. As for property (2), after the partitioning it still holds for any pivot other than p, as the partitioning rearranged elements at the left of it. With respect to p, the partitioning ensures that elements smaller than p are left at $A[i, p-1]$, while larger elements are left at $A[p+1, p'-1]$. Since $A[p]$ was already not larger than elements in $A[p', m-1]$, the lemma holds. It obviously remains true after removing elements from S. ∎

The worst-case complexity of **IQS** is $O(m^2)$, but it is easy to derive worst-case optimal **IS** from it. The only change is in line 2 of Fig. 2, where the random selection of the next pivot position must be changed to choosing the median of $A[idx, S.\text{top}() - 1]$, using the linear-time selection algorithm [2]. Section 3 analyzes the worst-case of **IS** and Section 4 considers the average-case of **IQS**, both of which are $O(m + k \log k)$.

3 IS worst-case complexity

In this section we analyze **IS**, which is not as efficient in practice as **IQS**, but has good worst-case performance. In particular, the analysis serves as a basis for the average-case analysis of **IQS** in Section 4. In **IS**, the partition is perfectly balanced since each pivot position is chosen as the median of its array segment.

In this analysis we assume that m is of the form $2^j - 1$. We recall that array indices are in the range $[0, m-1]$. Fig. 3 illustrates the incremental sorting process when $k = 5$ in a perfect balanced tree of $m = 31$ elements, $j = 5$. Black nodes are the elements already reported, grey nodes the pivots that remain in stack S, and white nodes and trees are the other elements of A.

The pivot at the tree root is the first to be obtained (the median of A), at cost linear in m (both to obtain the median and to partition the array). The two root children are the medians of $A[0, \frac{m-3}{2}]$ and $A[\frac{m+1}{2}, m-1]$. Obtaining those pivots and partitioning with them will cost time linear in $m/2$. The left child of the root

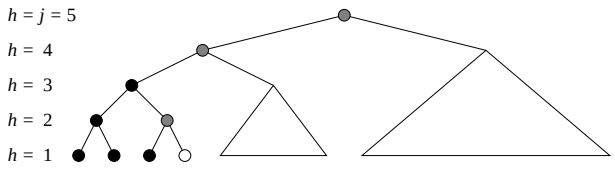


Figure 3: **IS** partition tree for incremental sorting when $k = 5$, $m = 31$, $j = 5$.

will actually be the second pivot to be processed. The right child, on the other hand, will be processed only if $k > \frac{m-1}{2}$, that is, at the moment we ask **IS** to output the $\frac{m+1}{2}$-th minimum. In general, processing the pivots at level h will cost $O(2^h)$, but only some of these will be required for a given k. The maximum level is $j = \log_2(m+1)$.

It is not hard to see that, in order to obtain the k smallest elements of A, we will require $\lceil \frac{k}{2^h} \rceil$ pivots of level h. Adding up their processing cost we get Eq. (3.1), where we split the sum into the cases $\lceil \frac{k}{2^h} \rceil > 1$ and $\lceil \frac{k}{2^h} \rceil = 1$. Only then, in Eq. (3.3), we use $k + 2^h$ to bound the terms of the first sum, and redistribute terms to obtain that **IS** is $O(m + k \log k)$ worst-case time. The extra space used by **IS** is $O(\log m)$ pivot positions.

$$T(m,k) = \sum_{h=1}^{\log_2(m+1)} \left\lceil \frac{k}{2^h} \right\rceil 2^h \tag{3.1}$$

$$= \sum_{h=1}^{\lfloor \log_2 k \rfloor} \left\lceil \frac{k}{2^h} \right\rceil 2^h + \sum_{h=\lfloor \log_2 k \rfloor + 1}^{\log_2(m+1)} 2^h \tag{3.2}$$

$$\leq \sum_{h=1}^{\lfloor \log_2 k \rfloor} k + \sum_{h=1}^{\log_2(m+1)} 2^h \tag{3.3}$$

$$T(m,k) = k\lfloor \log_2 k \rfloor + 2m + 1 \tag{3.4}$$

4 IQS average-case complexity

In this case the final pivot position p after the partitioning of $A[0, m-1]$ distributes uniformly in $[0, m-1]$. Consider again Fig. 3, where the root is not anymore the middle of A but a random position. We call $T(m, k)$ the average number of key comparisons needed to obtain the k smallest elements of $A[0, m-1]$. After the $m-1$ comparisons used in the partitioning, there are three cases depending on p: (1) $k \leq p$, in which case the right subtree will never be expanded, and the total extra cost will be $T(p, k)$ to solve $A[0, p-1]$; (2) $k = p+1$, in which case the left subtree will be fully expanded to obtain its p elements at cost $T(p, p)$; and (3) $k > p+1$, in which case we pay $T(p, p)$ on the left subtree, whereas the right subtree, of size $m-1-p$, will be expanded so as to obtain the remaining $k-p-1$ elements.

Thus **IQS** average cost follows Eq. (4.5), which is rearranged as Eq. (4.6). It is easy to check that this is exactly the same as Eq. (3.1) in [15], which shows that **IQS** makes exactly the same number of comparisons than its offline version, **PQS**. This is $2m + 2(m+1)H_m - 2(m+3-k)H_{m+1-k} - 6k + 6$. That analysis [15] is rather sophisticated, resorting to bivariate generating functions. In which follows we give a simple development arriving at a solution of the form $O(m + k \log k)$.

(4.5)
$$T(m,k) = m - 1 + \frac{1}{m}\left(\sum_{p=k}^{m-1} T(p,k) \;+\; T(k-1,k-1) + \sum_{p=0}^{k-2}\Big(T(p,p) + T(m-1-p,k-p-1)\Big)\right)$$

(4.6)
$$= m - 1 + \frac{1}{m}\left(\sum_{p=0}^{k-1} T(p,p) + \sum_{p=0}^{k-2} T(m-1-p,k-p-1) + \sum_{p=k}^{m-1} T(p,k)\right)$$

Eq. (4.6) simplifies to Eq. (4.7) by noticing that $T(p,p)$ behaves exactly like QuickSort, $2(p+1)H_p - 4p$ [8] (this can also be seen by writing down $T(p) = T(p,p)$ and noting that the very same QuickSort recurrence is obtained), so that $\sum_{p=0}^{k-1} T(p,p) = k(k+1)H_k - \frac{k}{2}(5k-1)$. We also write p' for $k-p-1$ and rewrite the second sum as $\sum_{p'=1}^{k-1} T(m-k+p',p')$.

(4.7)
$$T(m,k) = m - 1 + \frac{1}{m}\left(k(k+1)H_k - \frac{k}{2}(5k-1) + \sum_{p=1}^{k-1} T(m-k+p,p) \;+\; \sum_{p=k}^{m-1} T(p,k)\right)$$

We make some pessimistic simplifications now. The first sum governs the dependence on k of the recurrence. To avoid such dependence, we bound the second argument to k and the first to m, as $T(m,k)$ is monotonic on both its arguments. The new recurrence, Eq. (4.8), depends only on parameter m and treats k as a constant.

(4.8)
$$T(m) = m - 1 + \frac{1}{m}\left(k(k+1)H_k - \frac{k}{2}(5k-1) + (k-1)T(m) + \sum_{p=k}^{m-1} T(p)\right)$$

We subtract $m\,T(m) - (m-1)T(m-1)$ using Eq. (4.8), to obtain Eq. (4.9) and Eq. (4.10). Since $T(k)$ is actually $T(k,k)$, we use again QuickSort formula in Eq. (4.11). We bound the first part by $2m + 2kH_{m-k}$ and the second part by $2kH_k$ to obtain Eq. (4.12).

(4.9)
$$T(m) = 2\frac{m-1}{m-k+1} + T(m-1)$$

(4.10)
$$= 2\sum_{i=k+1}^{m}\left(1 + \frac{k-2}{i-k+1}\right) \;+\; T(k)$$

(4.11)
$$= 2(m-k) + 2(k-2)(H_{m-k+1} - 1) + (2(k+1)H_k - 4k)$$

(4.12)
$$< 2(m + kH_{m-k} + kH_k)$$

This result does not yet look good enough, but we plug it again into Eq. (4.7). In this case, we bound the sum $\sum_{p=1}^{k-1} T(m-k+p,p)$ with $\sum_{p=1}^{k-1} 2(m-k+p+pH_{m-k}+pH_p) = (k-1)\left(2m + k\left(H_{m-k} + H_k - \frac{3}{2}\right)\right)$. Upper bounding again and multiplying by m we get a new recurrence in Eq. (4.13). Note that this recurrence only depends on m.

(4.13)
$$m\,T(m) = m(m-1) + k(k+1)H_k - \frac{k}{2}(5k-1) + (k-1)\left(2m + k\left(H_{m-k} + H_k - \frac{3}{2}\right)\right) + \sum_{p=k}^{m-1} T(p)$$

Subtracting again $m\,T(m) - (m-1)T(m-1)$ we get Eq. (4.14). Noting that $\frac{(k-1)k}{(m-k)m} = (k-1)\left(\frac{1}{m-k} - \frac{1}{m}\right)$, we get Eq. (4.15), which is solved in Eq. (4.16).

(4.14)
$$T(m) = 2\frac{m+k-2}{m} + \frac{(k-1)k}{(m-k)m} + T(m-1)$$

(4.15)
$$< \sum_{i=k+1}^{m}\left(2 + 2\frac{k-2}{i} + (k-1)\left(\frac{1}{i-k} - \frac{1}{i}\right)\right) + T(k)$$

(4.16)
$$= 2(m-k) + 2(k-2)(H_m - H_k) + (k-1)(H_{m-k} - H_m + H_k) + (2(k+1)H_k - 4k)$$

Note that $H_m - H_k < \frac{m-k}{k+1}$ and thus $(k-2)(H_m - H_k) < m-k$. Also, $H_{m-k} \le H_m$, so collecting terms we obtain Eq. (4.17). Therefore, **IQS** is also $O(m + k \log k)$ in the average-case when we choose pivots at random.

(4.17)
$$T(m,k) \;<\; 4m - 8k + (3k+1)H_k \;<\; 4m + 3kH_k$$

As a final remark, note that when we use **QSS** a portion of the QuickSort partitioning work repeats the work made in the previous QuickSelect calling.

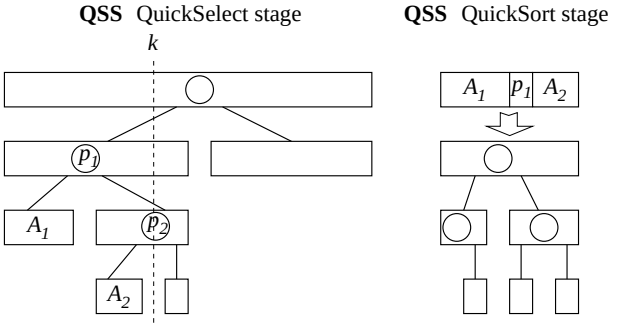


Figure 4: Partition work performed by **QSS**. First, **QSS** uses QUICKSELECT for finding the k-th element (left). Later, it uses QUICKSORT on the left array segment as a whole ($[A_1\ p_1\ A_2]$) neglecting the previous partitioning work (right).

Fig. 4 illustrates this, showing that upon finding the k-th element, the QUICKSELECT stage has produced partitions A_1 and A_2, however the QUICKSORT that follows processes the left partition as a whole ($[A_1 p_1 A_2]$), thus ignoring the previous partitioning work done over it. On the other hand, **IQS** sorts the left segment by processing each partition independently, because it knows their limits (as they are stored in the stack S). This also applies to **PQS** and it explains the finding of Conrado Martínez that **PQS**, and thus **IQS**, makes $2k - 4H_k + 2$ less comparisons than **QSS** [15].

5 IQS and the minimum spanning tree

In this section we explore a practical application of **IQS**: improving the performance of Kruskal's Minimum Spanning Tree (MST) algorithm.

Let us recall the MST problem. Let $G(V, E)$ be a connected graph with a nonnegative cost function $d(e)$ assigned to its edges $e \in E$. A minimum spanning tree mst of the graph $G(V, E)$ is a tree composed by edges of E that connect all the vertices of V at the lowest total cost $\sum_{e \in mst} d(e)$. Note that, given a graph, its MST is not necessarily unique.

Let $n = |V|$, $m = |E|$. The most popular algorithms to solve the MST problem are Kruskal's [14] and Prim's [18], whose basic versions have complexity $O(m \log m)$ and $O(n^2)$, respectively. We call **Kruskal1** and **Prim1** these basic versions. In sparse graphs, with $|E| = O(n)$, it is recommended to use **Kruskal1**, whereas in dense graphs, with $|E| = O(n^2)$, **Prim1** is recommended [4, 21]. There are other MST algorithms compiled by Tarjan [20].

Recently, Chazelle [3] gave an $O(m\alpha(m, n))$ algorithm, where $\alpha \in \omega(1)$ is the very slowly-growing inverse Ackermann's function. Later, Pettie and Ramachandran [17] proposed an algorithm that runs in optimal time $O(\mathcal{T}^*(m, n))$, where $\mathcal{T}^*(m, n)$ is the minimum number of edge-weight comparisons needed to determine the MST of any graph $G(V, E)$ with m edges and n vertices. The best known upper bound of this algorithm is also $O(m\alpha(m, n))$. These algorithms almost reach the lower bound $\Omega(m)$, yet they are so complicated that their interest is mainly theoretical.

Experimental studies on MST are given in [16, 12, 13]. In [16], they compare several Kruskal's, Prim's and Tarjan's versions, concluding that the best MST version is Prim's using paring heaps [6], we call **Prim2** this algorithm. Their experiments show that neither Cheriton and Tarjan's [20] nor Fredman and Tarjan's algorithm [7] ever approach the speed of **Prim2**. On the other hand, they show that **Kruskal1** can run very quickly when it uses an array of edges that can be overwritten during sorting, instead of an adjacency list. Moreover, they show that it is possible to use heaps to improve Kruskal's algorithm, calling *Kruskal's with demand sorting* this variant (we call it **Kruskal2**). The result is a rather efficient MST version.

In [12, 13], they give an algorithm that works as follows. It generates a subgraph G' by selecting $\sqrt{mn}$ edges from G at random. Later, it builds the minimum spanning forest T' of G'. Then, it filters each edge $e \in E$ using the cycle property: discard e if it is the heaviest edge on a cycle in $T' \cup \{e\}$. Finally, it builds the MST of the subgraph that contains the edges of T' and the edges that were not filtered out. We call **iMax** this algorithm. We obtain the **iMax** and also the optimized **Prim2** implementations from `www.mpi-sb.mpg.de/~sanders/dfg/iMax.tgz`.

5.1 Kruskal's MST algorithm.

The Kruskal's algorithm starts with n single-node components, and it merges them until it produces a sole connected component. To do this, **Kruskal1** begins by setting the mst to $(V, \emptyset)$, that is, n single-node trees. Later, in each iteration, it adds to the mst the cheapest edge of E that does not produce a cycle on the mst, that is, it only adds edges whose vertices belong to different connected components. Once the edge is added, both components are merged. The process ends when the mst becomes a single connected component. At this point the mst is a minimum spanning tree of $G(V, E)$.

To manage the component operations, we use the *Union-Find* data structure C with path compression, see [4, 21] for a comprehensive explanation. Given two vertices u and v, we use the **find**(u) operation to compute which component u belongs to, and use **union**(u, v) to merge the components of u and v. The amortized cost of **find**(u) is $O(\log^* n)$ and the cost of

Kruskal1 (Graph $G(V, E)$)

1. UnionFind $C \leftarrow \{v \in V, \{v\}\}$
   // the set of all connected components
2. $mst \leftarrow \emptyset$ // the growing minimum spanning tree
3. **ascendingSort**(E), $k \leftarrow 0$
4. **While** $|C| > 1$ **Do**
   // select an edge in ascending order
5. $(e = \{u, v\}) \leftarrow E[k]$, $k \leftarrow k + 1$
6. **If** C.**find**$(u) \neq C$.**find**(v) **Then**
7. $mst \leftarrow mst \cup \{e\}$, C.**union**(u, v)
8. **Return** mst

Figure 5: The basic version of Kruskal's MST algorithm (**Kruskal1**). To carry out the heap-based optimization (**Kruskal2**), we change line 3 to **heapify**(E) and line 5 to $(e = \{u, v\}) \leftarrow E$.**getMin**(), E.**extractMin**().

union(u, v) is constant.

Fig. 5 depicts the basic Kruskal's MST algorithm. We need $O(n)$ time to initialize both C and mst, and $O(m \log m)$ time to sort the edge set E. Then we make at most m $O(\log^* n)$-time iterations of the **While** cycle. Therefore, **Kruskal1** complexity is $O(m \log m)$.

Assuming we are using graphs whose cost edges are assigned at random independently of the rest (using any continuous distribution), the subgraph composed by V with the edges reviewed by the algorithm is a random graph [11]. Therefore, based on [11], we expect to finish the MST construction upon reviewing $\frac{1}{2}n \ln n$ edges, which can be much lower than m. So, it is not necessary to sort the whole set E, but it is enough with selecting and extracting one by one the minimum-cost edges until we complete the MST. The common solution of this type consists in min-heapifying the set E, and later performing as many min-extraction of the lowest cost edge as needed (in [16], they do this in their Kruskal's demand sorting version). This is an implementation of *Incremental Sort*. For this sake we modify lines 3 and 5 of Fig. 5: line 3 changes to **heapify**(E) and line 5 to $(e = \{u, v\}) \leftarrow E$.**getMin**(), E.**extractMin**().

Kruskal2 needs $O(n)$ time to initialize both C and mst, and $O(m)$ time to heapify E. We expect to review $\frac{1}{2}n \ln n$ edges in the **While** cycle. For each of these edges, we use $O(\log m)$ time to select and extract the minimum element of the heap, and $O(\log^* n)$ time to perform the **union** and **find** operations. Therefore, **Kruskal2** average complexity is $O(m + n \log n \log m)$. As $n - 1 \leq m \leq n^2$, **Kruskal2** average complexity can also be written as $O(m + n \log^2 n)$.

Kruskal3 (Graph $G(V, E)$)

1. UnionFind $C \leftarrow \{v \in V, \{v\}\}$
   // the set of all connected components
2. $mst \leftarrow \emptyset$ // the growing minimum spanning tree
3. Stack S, S.**push**(m), $k \leftarrow 0$ // $m = |E|$
4. **While** $|C| > 1$ **Do**
   // select the lowest edge incrementally
5. $(e = \{u, v\}) \leftarrow$ **IQS**(E, k, S), $k \leftarrow k + 1$
6. **If** C.**find**$(u) \neq C$.**find**(v) **Then**
7. $mst \leftarrow mst \cup \{e\}$, C.**union**(u, v)
8. **Return** mst

Figure 6: Our Kruskal's MST variant (**Kruskal3**). Note the changes in lines 3 and 5 with respect to **Kruskal1**.

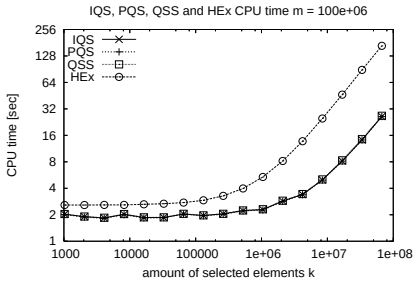
5.2 IQS-based implementation of the Kruskal's MST algorithm. We can use **IQS** in order to incrementally sort E. After initializing C and mst, we create the stack S, and push m into S. Later, inside the **While** cycle, we call **IQS** in order to obtain the k-th edge of E incrementally. Fig. 6 shows our Kruskal's MST variant, that we call **Kruskal3**. Note that the expected number of pivoting edges that we store in S is $O(\log m)$.

We need $O(n)$ time to initialize both C and mst, and constant time for S. We expect to review $\frac{1}{2}n \ln n$ edges within the **While** cycle, thus we need $O(m + n \log^2 n)$ overall expected time for IQS and $O(n \log n \log^* n)$ time for all the **union** and **find** operations. Therefore, **Kruskal3** average complexity is $O(m + n \log^2 n)$, just as **Kruskal2**.

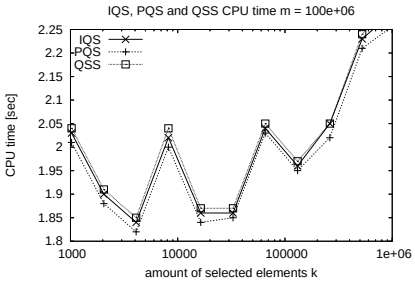
6 Experimental results

We ran two experimental series with **IQS**. In the first series we compare **IQS** against other alternatives. In the second we evaluate our **Kruskal3** algorithm. The experiments were run on an Intel Xeon of 3.06 GHz, 2 GB of RAM and local disk. The weighted least square fittings were performed with R [19]. In order to illustrate the precision of our fittings, we also show the average percent error of residuals with respect to real values ($|\frac{y-\hat{y}}{y}|100\%$) for fittings belonging to the 44% of the largest values.

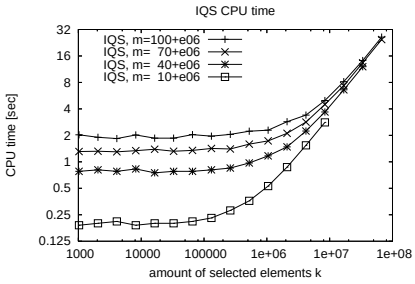
6.1 Evaluating IQS. We compared **IQS** against **PQS**, **QSS**, and the heap-based approach (called **HEx**). The idea is to verify that **IQS** is in practice a competitive algorithm for the *Partial Sorting* problem. For this sake, we use random permutations in $[0, m-1]$, for $m \in [10^5, 10^8]$, and we select the k first elements with



(a) CPU time for the four algorithms.



(b) Detail of CPU time for **IQS**, **PQS** and **QSS**.



(c) **IQS** CPU time as a function of k and m.

Figure 7: Performance comparison between **IQS**, **PQS**, **QSS** and **HEx** as a function of the amount of searched elements k for different values of set size m. Note the logscales in the plots.

	Fitting	Error
$\mathbf{PQS}_{cpu}$	$19.70m + 14.21k \log_2 k$	3.90%
$\mathbf{PQS}_{cmp}$	$2.047m + 1.301k \log_2 k$	3.55%
$\mathbf{IQS}_{cpu}$	$19.88m + 14.21k \log_2 k$	3.89%
$\mathbf{IQS}_{cmp}$	$2.047m + 1.301k \log_2 k$	3.55%
$\mathbf{QSS}_{cpu}$	$20.00m + 14.52k \log_2 k$	3.89%
$\mathbf{QSS}_{cmp}$	$2.050m + 1.362k \log_2 k$	3.61%
$\mathbf{HEx}_{cpu}$	$25.96m + 85.88k \log_2 m$	5.05%
$\mathbf{HEx}_{cmp}$	$1.892m + 1.875k \log_2 m$	0.65%

Table 1: **IQS**, **PQS**, **QSS** and **HEx** weighted least square fittings. CPU time is measured in nanoseconds.

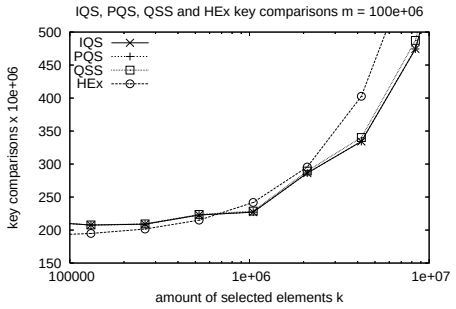


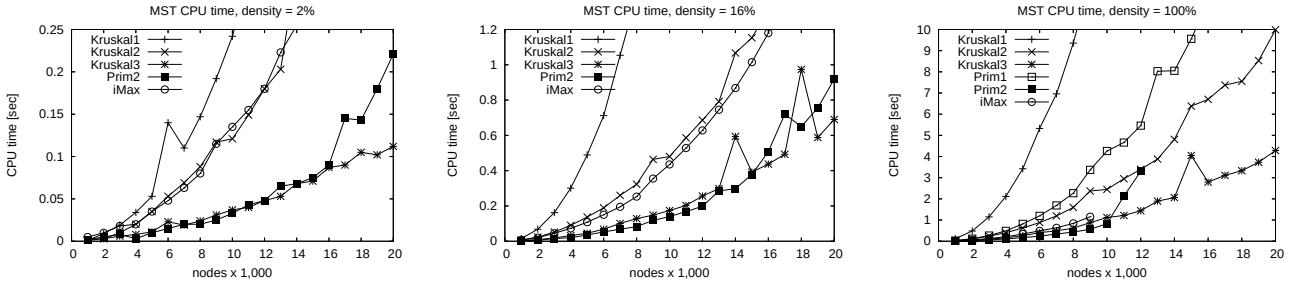
Figure 8: Detail of key comparisons for **IQS**, **PQS**, **QSS** and **HEx** for $m = 10^8$ varying k. Note the logscale in the plot.

$k = 2^j < m$. The selection is incremental for **IQS** and **HEx**, and in one shot for **PQS** and **QSS**. We measure CPU time and the number of key comparisons.

We summarize the experimental results in Figs. 7 and 8, and Table 1. As can be seen from the least squares fittings of Table 1, **IQS** CPU time performance is only 0.18% slower than that of its offline version **PQS**. The number of key comparisons is exactly the same, as we expected from Section 4. This is an extremely small price for permitting incremental sorting without knowing in advance how many elements we wish to retrieve, and shows that **IQS** is practical. Moreover, as the pivots in the stack help us reuse the partitioning work, our online **IQS** is 1.99% faster in CPU time and uses 4.00% less key comparisons than the offline **QSS**. Finally, the online **HEx** is slowest by far, as it takes 6.02 times more CPU time and 44% more key comparisons than **IQS**.

Fig. 7(a) compares the four algorithms. As can be seen, **HEx** has by far the worst CPU performance for all k, despite that it uses less key comparisons than others when extracting few objects, see Fig. 8 (for high values of k, **HEx** also uses more key comparisons than others). This is because the heap-based approach has more overhead. Fig. 7(b) shows that **PQS** is the fastest algorithm, but **IQS** and **QSS** have rather similar behavior, confirming the results of our fittings of Table 1. Finally, Fig. 7(c) shows that, as k grows, **IQS** behavior changes as follows. When $k \leq 0.01m$, there is no difference in the first k element incremental sorting, namely, the term m dominates the cost. When $0.01m < k \leq 0.04m$, there is a slight increase of both CPU time and key comparisons, that is, both terms m and $k \log k$ take part in the cost. Finally, when $0.04m < k \leq m$, term $k \log k$ leads the cost.

6.2 Evaluating Kruskal3. We now evaluate how **IQS** improves Kruskal's MST algorithm, so we compare its three versions. To do this, we use synthetic graphs with edges chosen at random, and with edge costs uniformly distributed in $[0, 1]$. We consider graphs



(a) MST CPU time, dependency on ρ. (b) MST CPU time, dependency on n. (c) MST CPU time, dependency on n.

Figure 9: Evaluating **Kruskal3**. MST CPU time, dependence on $n = |V|$ in (a), (b) and (c) for $\rho = 2\%$, 16% and 100%, respectively. For $n = 20,000$, in (a) **Kruskal1**, **Kruskal2** and **iMax** reaches 1.01, 0.43 and 0.57 seconds; in (b) **Kruskal1**, **Kruskal2** and **iMax** reaches 9.22, 1.82 and 2.36 seconds; in (c) **Kruskal1** and **Prim1** reaches 68.40 and 18.02 seconds, respectively.

	Fitting	Error
Sorted edges	$0.532n \ln n$	1.47%
Kruskal1$_{cpu}$	$12.23m \log_2 m$	6.74%
Kruskal2$_{cpu}$	$51.62m + 34.84n \log_2 n \log_2 m$	9.62%
Kruskal3$_{cpu}$	$21.67m + 10.01n \log_2^2 n$	9.75%

Table 2: Weighted least square fittings for Kruskal's MST versions ($n = |V|$, $m = |E|$). CPU time is measured in nanoseconds.

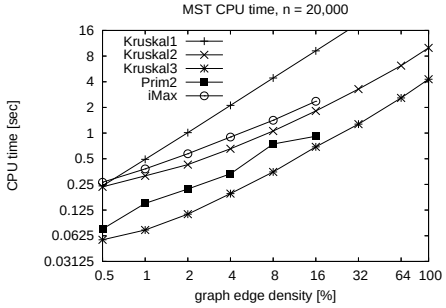


Figure 10: Evaluating **Kruskal3**. MST CPU time, dependency on ρ. For $\rho = 100\%$ **Kruskal1** reaches 68.4 seconds. Note the logscale.

with $|V| \in$ [1,000, 20,000], and graph edge densities $\rho \in [0.5\%, 100\%]$, where $\rho = \frac{2m}{n(n-1)}100\%$. We also show results for **Prim1** in complete graphs. Additionally, we compare **Kruskal3** against the **iMax** and **Prim2** implementations from [12].

For Kruskal's versions we measure the CPU time and the size of the edge subset reviewed during the MST construction. Note that those edges are the ones we incrementally sort. As the three versions run over the same graphs, they review the same subset of edges. For **Prim1**, **Prim2** and **iMax** we only measure CPU time.

We summarize the experimental results in Figs. 9 and 10, and Table 2. Fig. 10 compares the Kruskal's versions, **Prim2** and **iMax** for $n = 20,000$ and graph edge density $\rho \in [0.5\%, 100\%]$. As can be seen, **Kruskal1** is, by far, the costlier version, and, **Kruskal3** is systematically the best for all ρ. We also notice that, as ρ increases, the advantage of our MST variant is more remarkable against basic Kruskal's MST algorithm. We could not complete the series for **Prim2** and **iMax**, as their structures require too much space. For 20,000 vertices and $\rho \geq 32\%$ these algorithms reach the 2 GB out-of-memory threshold of our machine.

Figs. 9(a), 9(b) and 9(c) show the comparison for three edge densities $\rho = 2\%$, 16% and 100%, respectively. In the three plots **Kruskal3** is always the best Kruskal's version for all sizes of set V and all edge densities ρ. Moreover, Fig. 9(c) shows that **Kruskal3** is also better than **Prim1**, even in complete graphs. On the other hand, **Kruskal3** is better than **iMax** in the three plots, and very competitive against **Prim2**, beating **Prim2** in graphs with many nodes (for $|V| \geq$ 17,000, 16,000 and 11,000 vertices in $\rho = 2\%$, 16% and 100%, respectively). We suspect that this is due the high memory usage of **Prim2**, which affects cache efficiency. Note that with $\rho = 100\%$ we could not finish the series with **Prim2** and **iMax** because of their memory requirements.

Table 2 shows our least squares fittings for the MST experiments. First of all, we compute the fitting for the number of lowest cost edges Kruskal's MST algorithm reviews to build the tree. We obtain $0.532\ |V| \ln |V|$,

which is very close to the theoretically expected value $\frac{1}{2}|V| \ln |V|$. Later, we compute fittings using the theoretical models for the three versions. Note that, in terms of CPU time, **Kruskal1** is 15.6 times and **Kruskal2** is 2.35 times slower than **Kruskal3**.

7 Conclusions

We have presented IncrementalQuickSelect (**IQS**), an algorithm to incrementally retrieve the next smallest element from a set. **IQS** has the same complexity than existing solutions, but it is considerably faster in practice, as fast as the best algorithm that knows beforehand the number of elements to retrieve. We have applied **IQS** to solve the graph MST problem, showing that the **IQS**-based Kruskal's version is competitive against the best state-of-the-art alternatives.

We finish with two remarks. The first is that we can use the **IQS** stack-of-pivots underlying idea to partially sort in increasing/decreasing order starting from any place of the array. For instance, if we want to perform an incremental sorting in increasing order with a stack initialized as the set size, we first use QuickSelect to find the first element we want, storing in the stack all the pivots larger than the first element, and later we use **IQS** with the stack to search for the next elements (the other pivots would be useful for decreasing order, initializing the stack with -1). Moreover, with two stacks we can make centered searching, namely, finding the k-th element, the $(k+1)$-th and $(k-1)$-th, the $(k+2)$-th and $(k-2)$-th, and so on.

The second remark is we can use **IQS** as an underlying implementation of the Heap data structure [4, 21]. (Naturally, this allow us to speed up HeapSort [22].) In this application, we heapify the set A by using **IQS** to search for the first element, paying on average $O(m)$ CPU time, and then we extract elements by using **IQS** incrementally, paying average amortized time $O(\log k)$ for the k-th extraction. To insert a new element x, we need to know which is the array segment that corresponds to x (see Fig. 1). To do this it is enough with reviewing the pivot stack S. Assume $S = \{|A|, p_1, p_2, ..., p_j\}$. From Lemma 2.1, we know that $A[p_1] > A[p_2] > \ldots > A[p_j]$. So, to insert x we need to find the first pivot p_i such that $A[p_i] < x$, so as to place x at $A[p_{i-1}]$. Then, we put $A[p_{i-1}]$ at position $p_{i-1}+1$ (and increment p_{i+1} is S). Then, we move the old $A[p_{i-1}+1]$ value to $A[p_{i-2}]$, and so on. Note that as pivot closer to the bottom cover exponentially large areas, the insertion takes $O(1)$ time on average.

References

[1] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley, 1999.

[2] M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7(4):448–461, 1973.

[3] B. Chazelle. A minimum spanning tree algorithm with inverse-ackermann type complexity. *Journal of the ACM (JACM)*, 47(6):1028–1047, 2000.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.

[5] R. W. Floyd. Algorithm 245 (treesort). *Communications of the ACM*, 7:701, 1964.

[6] M. Fredman, R. Sedgewick, D. Sleator, and R. Tarjan. The pairing heap: a new form of self-adjusting heap. *Algorithmica*, 1(1):111–129, 1986.

[7] M. Fredman and R. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)*, 34(3):596–615, 1987.

[8] G. Gonnet and R. Baeza-Yates. *Handbook of Algorithms and Data Structures*. Addison-Wesley, 2nd edition, 1991.

[9] C. A. R. Hoare. Algorithm 65 (find). *Communications of the ACM*, 4(7):321–322, 1961.

[10] C. A. R. Hoare. Quicksort. *Computer Journal*, 5(1):10–15, 1962.

[11] S. Janson, D. Knuth, T. Łuczak, and B. Pittel. The birth of the giant component. *Random Structures & Algorithms*, 4(3):233–358, 1993.

[12] I. Katriel, P. Sanders, and J. Träff. A practical minimum spanning tree algorithm using the cycle property. Research Report MPI-I-2002-1-003, Max-Planck-Institut für Informatik, October 2002.

[13] I. Katriel, P. Sanders, and J. Träff. A practical minimum spanning tree algorithm using the cycle property. In *11th European Symposium on Algorithms (ESA'03)*, LNCS 2832, pages 679–690, 2003.

[14] J. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7:48–50, 1956.

[15] C. Martínez. Partial quicksort. In *Proc. 6th ACM-SIAM Workshop on Algorithm Engineering and Experiments and 1st ACM-SIAM Workshop on Analytic Algorithmics and Combinatorics*, pages 224–228, 2004.

[16] B. Moret and H. Shapiro. An empirical analysis of algorithms for constructing a minimum spanning tree. In *Proc. 2nd Workshop Algorithms and Data Structures (WADS'91)*, LNCS 519, pages 400–411, 1991.

[17] S. Pettie and V. Ramachandran. An optimal minimum spanning tree algorithm. *Journal of the ACM (JACM)*, 49(1):16–34, 2002.

[18] R. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36:1389–1401, 1957.

[19] R Development Core Team. *R: A language and environment for statistical computing*. R Foundation for Statistical Computing, Vienna, Austria, 2004.

[20] R. Tarjan. *Data structures and network algorithms*.

Society for Industrial and Applied Mathematics, 1983.

[21] M. Weiss. *Data structures & algorithm analysis in Java*™. Addison-Wesley, 1999.

[22] J. Williams. Algorithm 232 (HEAPSORT). *Communications of the ACM*, 7(6):347–348, 1964.