Optimally selecting the top k values from X+Y with layer-ordered heaps

Oliver Serang [Corresp. 1]

[1] Department of Computer Science, University of Montana, Missoula, Montana, United States

Corresponding Author: Oliver Serang
Email address: oliver.serang@umt.edu

Selection and sorting the Cartesian sum, $X+Y$, are classic and important problems. Here, a new algorithm is presented, which generates the top k values of the form X_i+Y_j. The algorithm relies on layer-ordered heaps, partial orderings of exponentially sized layers. The algorithm relies only on median-of-medians and is simple to implement. Furthermore, it uses data structures contiguous in memory, and is cache efficient and fast in practice. The presented algorithm is demonstrated to be theoretically optimal.

Optimally selecting the top k values from $X + Y$ with layer-ordered heaps

Oliver Serang[1]

[1]**University of Montana, Dept. of Computer Science, Missoula, MT, USA**

Corresponding author:
Oliver Serang[1]

Email address: oliver.serang@umontana.edu

ABSTRACT

Selection and sorting the Cartesian sum, $X + Y$, are classic and important problems. Here, a new algorithm is presented, which generates the top k values of the form $X_i + Y_j$. The algorithm relies on layer-ordered heaps, partial orderings of exponentially sized layers. The algorithm relies only on median-of-medians and is simple to implement. Furthermore, it uses data structures contiguous in memory, and is cache efficient and fast in practice. The presented algorithm is demonstrated to be theoretically optimal.

1 INTRODUCTION

Given two vectors of length n, X and Y, top-k on $X + Y$ finds the k smallest values of the form $X_i + Y_j$. Note that this problem definition is presented w.l.o.g.; X and Y need not share the same length. Top-k is important to practical applications, such as selecting the most abundant k isotopologue peaks from a compound (Kreitzberg et al., 2020b). Top-k is $\in \Omega(n + k)$, because loading the vectors is $\in \Theta(n)$ and returning the minimal k values is $\in \Theta(k)$.

1.1 Naive approach

Top-k can be solved trivially in $O(n^2 \log(n) + k) = O(n^2 \log(n))$ steps by generating and sorting all n^2 values of the form $X_i + Y_j$. By using median-of-medians (Blum et al., 1973), this can be improved to $O(n^2)$ steps by generating all n^2 values and performing k-selection on them.

1.2 Existing, tree-based methods for top-k

In 1982, Frederickson & Johnson introduced a method reminiscent of median-of-medians (Blum et al., 1973); their method selects only the kth minimum value from $X + Y$ in $O(n + \min(n,k) \log(\frac{k}{\min(n,k)}))$ steps (Frederickson and Johnson, 1982).

Frederickson subsequently published a second algorithm, which finds the k smallest elements from a min-heap in $O(k)$, assuming the heap has already been built (Frederickson, 1993). Combining this method with a combinatoric heap on $X + Y$ (described below for the Kaplan *et al.* method) solves top-k in $O(n + k)$. The tree data structure in Frederickson's method can be combined with a combinatoric heap to compute the kth smallest value from $X + Y$.

Kaplan *et al.* described an alternative method for selecting the kth smallest value (Kaplan et al., 2019); that method explicitly used Chazelle's soft heaps (Chazelle, 2000). By heapifying X and Y in linear time (*i.e.*, guaranteeing w.l.o.g. that $X_i \le X_{2i}, X_{2i+1}$), $\min_{i,j} X_i + Y_j = X_1 + Y_1$. Likewise, $X_i + Y_j \le X_{2i} + Y_j, X_{2i+1} + Y_j, X_i + Y_{2j}, X_i + Y_{2j+1}$. The soft heap is initialized to contain tuple $(X_1 + Y_1, 1, 1)$. Then, as tuple (v, i, j) is popped from soft heap, lower-quality tuples are inserted into the soft heap. These lower-quality tuples of (i, j) are

$$\begin{cases} \{(2i,1),(2i+1,1),(i,2),(i,3)\}, & j = 1 \\ \{(i,2j),(i,2j+1)\}, & j > 1. \end{cases} \quad (1)$$

34 In the matrix $X_i + Y_j$ (which is not realized), this scheme progresses in row-major order, thereby avoiding
35 a tuple being added multiple times.
36   To compute the kth smallest value from $X + Y$, the best k values are popped from the soft heap. Even
37 though only the minimal k values are desired, "corruption" in the soft heap means that the soft heap will
38 not always pop the minimal value; however, as a result, soft heaps can run faster than the $\Omega(n \log(n))$
39 lower bound on comparison sorting. The free parameter $\varepsilon \in (0, \frac{1}{2}]$ bounds the number of corrupt elements
40 in the soft heap (which may be promoted earlier in the queue than they should be) as $\leq t \cdot \varepsilon$, where t
41 is the number of insertions into the soft heap thus far. Thus, instead of popping k items (and inserting
42 their lower-quality dependents as described in equation 1), the total number of pops p can be found: The
43 maximal size of the soft heap after p pops is $\leq 3p$ (because each pop removes one element and inserts
44 ≤ 4 elements according to equation 1); therefore, $p - corruption \geq p - 4p \cdot \varepsilon$, and thus $p - 4p \cdot \varepsilon \geq k$
45 guarantees that $p - corruption \geq k$. This leads to $p = \frac{k}{1-4\varepsilon}$, $\varepsilon \leq \frac{1}{4}$. This guarantees that $\Theta(k)$ values,
46 which must include the minimal k values, are popped. These values are post-processed to retrieve the
47 minimal k values via linear time one-dimensional selection (Blum et al., 1973). For constant ε, both
48 pop and insertion operations to the soft heap are $\in \tilde{O}(1)$, and thus the overall runtime of the algorithm is
49 $\in O(n + k)$.
50   Note that the Kaplan *et al.* method easily solves top-k in $O(n + k)$ steps; this is because computing
51 the kth smallest value from $X + Y$ pops the minimal k values from the soft heap.

1.3 Layer-ordered heaps and a novel selection algorithm on $X + Y$

53 This paper uses layer-ordered heaps (LOHs) (Kreitzberg et al., 2020a) to produce an optimal selection
54 algorithm on $X + Y$. LOHs are stricter than heaps but not as strict as sorting: Heaps guarantee only that
55 $X_i \leq X_{children(i)}$, but do not guarantee any ordering between one child of X_i, a, and the child of the sibling
56 of a. Sorting is stricter still, but sorting n values cannot be done faster than $\log_2(n!) \in \Omega(n \log(n))$. LOHs
57 partition the array into several layers such that the values in a layer are $\leq$ to the values in subsequent layers:
58 $X^{(u)} = X_1^{(u)}, X_2^{(u)}, \ldots \leq X^{(u+1)}$. The size of these layers starts with $|X^{(1)}| = 1$ and grows exponentially
59 such that $\lim_{u \to \infty} \frac{|X^{(u+1)}|}{|X^{(u)}|} = \alpha \geq 1$ (note that $\alpha = 1$ is equivalent to sorting because all layers have size 1). By
60 assigning values in layer u children from layer $u + 1$, this can be seen as a more constrained form of the
61 heap; however, unlike sorting, for any constant $\alpha > 1$, LOHs can be constructed $\in O(n)$ by performing
62 iterative linear time one-dimensional selection, iteratively selecting and removing the largest layer until
63 all layers have been partitioned. For example, $8, 1, 6, 4, 5, 3, 2$ can be LOHified with $\alpha = 2$ into an LOH
64 with three layers ($1 \leq 3, 2 \leq 8, 4, 6, 5$) by first selecting the largest 4 values on the entire list ($8, 4, 6, 5$),
65 removing them, and then selecting the largest 2 values from the remaining 3 values ($3, 2$).
66   Although selections reminiscent of LOHs may have been used previously, formalization of rank α
67 LOHs has been necessary to demonstrate that for $1 \ll \alpha \ll 2$, a combination of LOHs and soft heaps allow
68 generating the minimum k values from $X_1 + X_2 + \cdots + X_m$ (where each X_i has length n) in $o(n \cdot m + k \cdot m)$
69 (Kreitzberg et al., 2020a). Furthermore, efficiently constructing an LOH of rank α is not trivial when
70 $\alpha \ll 2$; after all, $\alpha \to 1$ results in layers of size $|X^{(1)}| = |X^{(2)}| = \cdots = 1$, indicating a sorting, which
71 implies a runtime $\in \Omega(n \log(n))$ (Pennington et al., 2020).
72   A `python` implementation of a LOH is shown in listing 1.

1.4 Contribution in this manuscript

74 The new, optimal algorithm for solving top-k presented here makes extensive use of LOHs. It is simple to
75 implement, does not rely on anything more complicated than linear time one-dimensional selection (*i.e.*,
76 it does not use soft heap). Due to its simplicity and contiguous memory access, it has a fast performance
77 in practice.

2 METHODS

2.1 Algorithm

80 The algorithm presented is broken into phases. An illustration of these phases is provided in figure 1.

2.1.1 Phase 0

82 The algorithm first LOHifies (*i.e.*, constructs a layer order heap from) both X and Y. This is performed
83 by using linear time one-dimensional selection to iteratively remove the largest remaining layer (*i.e.*, the
84 simplest LOH construction method, which is optimal when $\alpha \gg 1$).
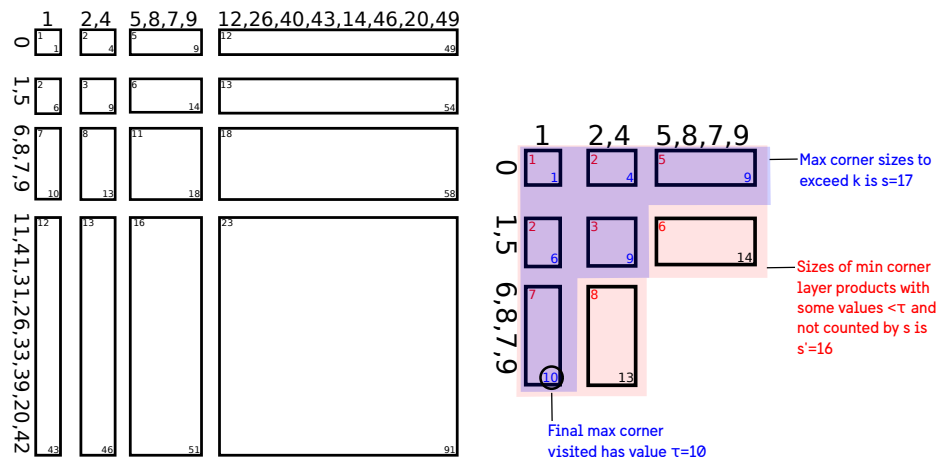
Figure 1. Illustration of method for selecting the $k = 14$ minimal values from $X + Y$: Phase 0: $X = \{31, 5, 11, 7, 33, 6, 39, 42, 20, 0, 9, 1, 41, 26, 8\}$ and $Y = \{12, 26, 40, 9, 14, 49, 8, 2, 20, 1, 46, 43, 4, 5, 7\}$ are both LOHified to axes in $O(n)$ time. Note that the minimum and maximum values in a layer are placed at the first and last positions in the layer, respectively; otherwise values within layers are themselves unordered. Phase 1: The minimum and maximum corners of all layer products (grid, left panel) are visited together in ascending order until the area of the layer products whose max corners are visited exceeds k (inset, right panel), and the largest value visited is labeled as $\tau = 10$. Phase 2: The layer products whose max corners have been visited (blue) has area s that exceeds k but has $s \in O(k)$. Likewise, the layer products whose min corners have been visited but whose max corners have not been visited, and which therefore contain some elements $< \tau$, have area $s' \in O(k)$. Phase 3: Together, these layer products (red and blue) contain all values that may be in minimal $k = 14$. Since there are $\Theta(k)$ such values, they can be selected using median-of-medians in $O(k)$ time.

2.1.2 Phase 1

Now layer products of the form $X^{(u)} + Y^{(v)} = X_1^{(u)} + Y_1^{(v)}, X_1^{(u)} + Y_2^{(v)}, \ldots, X_2^{(u)} + Y_1^{(v)}, \ldots$ are considered, where $X^{(u)}$ and $Y^{(v)}$ are layers of their respective LOHs.

In phases 1–2, the algorithm initially considers only the minimum and maximum values in each layer product: $\lfloor (u, v) \rfloor = (\min(X^{(u)} + Y^{(v)}), (u, v), 0)$, $\lceil (u, v) \rceil = (\max(X^{(u)} + Y^{(v)}), (u, v), 1)$. It is unnecessary to compute the Cartesian product of values to build a layer product; instead, only the minimum or maximum values in $X^{(u)}$ and $Y^{(v)}$ are needed. Note that the final value in the tuple uses 0 to indicate that this is the minimum value in the layer product or 1 to indicate the maximum value in the layer product; this ensures that even layer products with homogeneous values satisfy $\lfloor (u, v) \rfloor < \lceil (u, v) \rceil$. Scalar values can be compared to tuples: $X_i + Y_j \le \lceil (u, v) \rceil = (\max(X^{(u)} + Y^{(v)}), (u, v), 1) \leftrightarrow X_i + Y_j \le \max(X^{(u)} + Y^{(v)})$.

Binary heap H is initialized to contain tuple $\lfloor (1, 1) \rfloor$. A set of all tuples in H is maintained to prevent duplicates from being inserted into H (this set could be excluded by using the Kaplan *et al.* proposal scheme). The algorithm proceeds by popping the lexicographically minimum tuple from H. W.l.o.g., there is no guaranteed ordering of the form $X^{(u)} + Y^{(v)} \le X^{(u+1)} + Y^{(v)}$, because it may be that $\max(X^{(u)} + Y^{(v)}) > \min(X^{(u+1)} + Y^{(v)})$; however, lexicographically, $\lfloor (u, v) \rfloor < \lfloor (u + 1, v) \rfloor, \lfloor (u, v + 1) \rfloor, \lceil (u, v) \rceil$; thus, the latter tuples need be inserted into H only after $\lfloor (u, v) \rfloor$ has been popped from H. Note that for this reason and to break ties where layer products contain identical values, (u, v) are included in the tuple. $\lceil (u, v) \rceil$ tuples do not insert any new tuples into H when they're popped.

Whenever a tuple of the form $\lceil (u, v) \rceil$ is popped from H, the index (u, v) is appended to list q and the size of the layer product $|X^{(u)} + Y^{(v)}| = |X^{(u)}| \cdot |Y^{(v)}|$ is accumulated into integer s. This method proceeds until $s \ge k$.

2.1.3 Phase 2

Any remaining tuple in H of the form $(\max(X^{(u')} + Y^{(v')}), (u', v'), 1)$ has its index (u', v') appended to list q. s' is the total number of elements in each of these (u', v') layer products appended to q during phase 2.

2.1.4 Phase 3

The values from every element in each layer product in q are generated. A linear time one-dimensional k-selection is performed on these values and returned.

2.2 Proof of correctness

Lemma 2.4 proves that at termination all layer products found in q must contain the minimal k values in $X + Y$. Thus, by performing one-dimensional k-selection on those values in phase 3, the minimal k values in $X + Y$ are found.

Lemma 2.1. *If $\lfloor (u,v) \rfloor$ is popped from H, then both $\lfloor (u-1,v) \rfloor$ (if $u > 1$) and $\lfloor (u,v-1) \rfloor$ (if $v > 1$) must previously have been popped from H.*

Proof. There is a chain of pops and insertions backwards from $\lfloor (u,v) \rfloor$ to $\lfloor (1,1) \rfloor$.

When both $u, v = 1$, the lemma is true.

W.l.o.g. if $u = 1$ this chain is of the form $\lfloor (1,1) \rfloor, \ldots, \lfloor (1,2) \rfloor, \ldots \lfloor (1,3) \rfloor, \ldots \lfloor (u,v) \rfloor$, proving the lemma for that case.

Otherwise, both $u, v > 1$. Because insertions into H increment either row or column, something of the form $\lfloor (a,v-1) \rfloor$ with $a \leq u$ must be inserted into H before inserting $\lfloor (u,v) \rfloor$. $\lfloor (a,v-1) \rfloor < \lfloor (u,v) \rfloor$, so $\lfloor (a,v-1) \rfloor$ must precede $\lfloor (u,v) \rfloor$ in the chain of pops. If $a = u$, then $\lfloor (u,v-1) \rfloor$ is popped before $\lfloor (u,v) \rfloor$. If $a < u$, then from the insertion of $\lfloor (a,v-1) \rfloor$ into H, until $\lfloor (u,v-1) \rfloor$ is popped, H must contain something of the form $\lfloor (a',v-1) \rfloor : a' \leq u$, because popping $\lfloor (a',v-1) \rfloor$ inserts $\lfloor (a'+1,v-1) \rfloor$. $\lfloor (a',v-1) \rfloor < \lfloor (u,v) \rfloor$ when $a' \leq u$; therefore, $\lfloor (u,v) \rfloor$ cannot be popped before any $\lfloor (a',v-1) \rfloor$ currently in H. Because there are a finite number of these a' and they are not revisited, before $\lfloor (u,v) \rfloor$ is popped, $\lfloor (u,v-1) \rfloor$ must be popped. This same process can be repeated with $\lfloor (u-1,b) \rfloor : b \leq v$ to show that $\lfloor (u-1,v) \rfloor$ must be popped before $\lfloor (u,v) \rfloor$, proving the lemma for the final case. $\square$ $\square$

Lemma 2.2. *If $\lceil (u,v) \rceil$ is popped from H, then both $\lceil (u-1,v) \rceil$ (if $u > 1$) and $\lceil (u,v-1) \rceil$ (if $v > 1$) must previously have been popped from H.*

Proof. Inserting $\lceil (u,v) \rceil$ requires previously popping $\lfloor (u,v) \rfloor$. By lemma 2.1, this requires previously popping $\lfloor (u-1,v) \rfloor$ (if $u > 1$) and $\lfloor (u,v-1) \rfloor$ (if $v > 1$). These pops will insert $\lceil (u-1,v) \rceil$ and $\lceil (u,v-1) \rceil$ respectively. Thus, $\lceil (u-1,v) \rceil$ and $\lceil (u,v-1) \rceil$, which are both $< \lceil (u,v) \rceil$, are inserted before $\lceil (u,v) \rceil$, and will therefore be popped before $\lceil (u,v) \rceil$. $\square$ $\square$

Lemma 2.3. *All tuples will be visited in ascending order as they are popped from H.*

Proof. Let $\lfloor (u,v) \rfloor$ be popped from H and let $\lfloor (a,b) \rfloor < \lfloor (u,v) \rfloor$. Either w.l.o.g. $a < u, b \leq v$, or w.l.o.g. $a < u, b > v$. In the former case, $\lfloor (a,b) \rfloor$ will be popped before $\lfloor (u,v) \rfloor$ by applying induction to lemma 2.1.

In the latter case, lemma 2.1 says that $\lfloor (a,v) \rfloor$ is popped before $\lfloor (u,v) \rfloor$. $\lfloor (a,v) \rfloor < \lfloor (a,v+1) \rfloor < \lfloor (a,v+1) \rfloor < \cdots < \lfloor (a,b) \rfloor < \lfloor (u,v) \rfloor$, meaning that $\forall r \in [v,b]$, $\lfloor (a,r) \rfloor < \lfloor (u,v) \rfloor$. After $\lfloor (a,v) \rfloor$ is inserted (necessarily before it is popped), at least one such $\lfloor (a,r) \rfloor$ must be in H until $\lfloor (a,b) \rfloor$ is popped. Thus, all such $\lfloor (a,r) \rfloor$ will be popped before $\lfloor (u,v) \rfloor$.

Ordering on popping with $\lceil (a,b) \rceil < \lceil (u,v) \rceil$ is shown in the same manner: For $\lceil (u,v) \rceil$ to be in H, $\lfloor (u,v) \rfloor$ must have previously been popped. As above, whenever $\lceil (u,v) \rceil$ is in H, then $\lfloor (a,v) \rfloor$ must have been popped, inserting $\lfloor (a,v+1) \rfloor$ into H. Each $\lfloor (a,r) \rfloor$ popped inserts $\lfloor (a,r+1) \rfloor$, so at least one $\lfloor (a,r) \rfloor, r \in [v,b]$ must also be in H until $\lfloor (a,b) \rfloor$ is popped. These $\lfloor (a,r) \rfloor \leq \lfloor (a,b) \rfloor < \lceil (a,b) \rceil < \lceil (u,v) \rceil$, and so $\lceil (a,b) \rceil$ will be popped before $\lceil (u,v) \rceil$.

Identical reasoning also shows that $\lfloor (a,b) \rfloor$ will pop before $\lceil (u,v) \rceil$ if $\lfloor (a,b) \rfloor < \lceil (u,v) \rceil$ or if $\lceil (a,b) \rceil < \lfloor (u,v) \rfloor$.

Thus, all tuples are popped in ascending order. $\square$ $\square$

Lemma 2.4. *At the end of phase 2, the layer products whose indices are found in q contain the minimal k values.*

Proof. Let (u,v) be the layer product that first makes $s \geq k$. There are at least k values of $X + Y$ that are $\leq \max(X^{(u)} + Y^{(v)})$; this means that $\tau = \max(select(X + Y, k)) \leq \max(X^{(u)} + Y^{(v)})$. The quality of the elements in layer products in q at the end of phase 1 can only be improved by trading some value for a smaller value, and thus require a new value $< \max(X^{(u)} + Y^{(v)})$.

159   By lemma 2.3, tuples will be popped from H in ascending order; therefore, any layer product (u', v')
160   containing values $< \max(X^{(u)} + Y^{(v)})$ must have had $\lfloor (u', v') \rfloor$ popped before $\lceil (u, v) \rceil$. If $\lceil (u', v') \rceil$ was
161   also popped, then this layer product is already included in q and cannot improve it. Thus the only layers
162   that need be considered further have had $\lfloor (u', v') \rfloor$ popped but not $\lceil (u', v') \rceil$ popped; these can be found
163   by looking for all $\lceil (u', v') \rceil$ that have been inserted into H but not yet popped.
164   Phase 2 appends to q all such remaining layer products of interest. Thus, at the end of phase 2, q
165   contains all layer products that will be represented in the k-selection of $X + Y$. $\square$ $\square$

166   A `python` implementation of this method is shown in listing 2.

2.3 Runtime

168 Theorem 2.8 proves that the total runtime is $\in O(n + k)$.

Lemma 2.5. *Let (u', v') be a layer product appended to q during phase 2. Either $u' = 1$, $v' = 1$, or*
170 *$(u' - 1, v' - 1)$ was already appended to q in phase 1.*

Proof. Let $u' > 1$ and $v' > 1$. By lemma 2.3, minimum and maximum layer products are popped in as-
172 cending order. By the layer ordering property of X and Y, $\max(X^{(u'-1)}) \leq \min(X^{(u')})$ and $\max(Y^{(v'-1)}) \leq$
173 $\min(Y^{(v')})$. Thus, $\lceil (u' - 1, v' - 1) \rceil < \lfloor (u', v') \rfloor$ and so $\lceil (u' - 1, v' - 1) \rceil$ must be popped before $\lfloor (u', v') \rfloor$.
174 $\square$ $\square$

Lemma 2.6. *s, the number of elements in all layer products appended to q in phase 1, is $\in O(k)$.*

Proof. (u, v) is the layer product whose inclusion during phase 1 in q achieves $s \geq k$; therefore, $s - |X^{(u)} +$
177 $Y^{(v)}| < k$. This happens when $\lceil (u, v) \rceil$ is popped from H.
178   If $k = 1$, popping $\lceil (1, 1) \rceil$ ends phase 1 with $s = 1 \in O(k)$.
179   If $k > 1$, then at least one layer index is > 1: $u > 1$ or $v > 1$. W.l.o.g., let $u > 1$. By lemma 2.1,
180 popping $\lceil (u, v) \rceil$ from H requires previously popping $\lceil (u - 1, v) \rceil$. $|X^{(u)} + Y^{(v)}| = |X^{(u)}| \cdot |Y^{(v)}| \approx \alpha \cdot$
181 $|X^{(u-1)}| \cdot |Y^{(v)}| = \alpha \cdot |X^{(u-1)} + Y^{(v)}|$ (where $\approx$ indicates asymptotic behavior); therefore, $|X^{(u)} + Y^{(v)}| \in$
182 $O(|X^{(u-1)} + Y^{(v)}|)$. $|X^{(u-1)} + Y^{(v)}|$ is already counted in $s - |X^{(u)} + Y^{(v)}| < k$, and so $|X^{(u-1)} + Y^{(v)}| < k$
183 and $|X^{(u)} + Y^{(v)}| \in O(k)$. $s < k + |X^{(u)} + Y^{(v)}| \in O(k)$ and hence $s \in O(k)$. $\square$ $\square$

Lemma 2.7. *s', the total number of elements in all layer products appended to q in phase 2, $\in O(k)$.*

Proof. Each layer product appended to q in phase 2 has had $\lfloor (u', v') \rfloor$ popped in phase 1. By lemma 2.5,
186 either $u' = 1$ or $v' = 1$ or $\lceil (u' - 1, v' - 1) \rceil$ must have been popped before $\lfloor (u', v') \rfloor$.
187   First consider when $u' > 1$ and $v' > 1$. Each (u', v') matches exactly one layer product $(u' - 1, v' - 1)$.
188 Because $\lceil (u' - 1, v' - 1) \rceil$ must have been popped before $\lfloor (u', v') \rfloor$, then $\lceil (u' - 1, v' - 1) \rceil$ was also popped
189 during phase 1. s, the count of all elements whose layer products were inserted into q in phase 1, includes
190 $|X^{(u'-1)} + Y^{(v'-1)}|$ but does not include $X^{(u')} + Y^{(v')}$ (the latter is appended to q during phase 2). By
191 exponential growth of layers in X and Y, $|X^{(u')} + Y^{(v')}| \approx \alpha^2 \cdot |X^{(u'-1)} + Y^{(v'-1)}|$. These $|X^{(u'-1)} + Y^{(v'-1)}|$
192 values were included in s during phase 1, and thus the total number of elements in all such $(u' - 1, v' - 1)$
193 layer products is $\leq s$. Thus the sum of sizes of all layer products (u', v') with $u' > 1$ and $v' > 1$ that are
194 appended to q during phase 2 is asymptotically $\leq \alpha^2 \cdot s$.
195   When either $u' = 1$ or $v' = 1$, the number of elements in all layer products must be $\in O(n)$: $\sum_{u'} |X^{(u')} +$
196 $Y^{(1)}| + \sum_{v'} |X^{(u')} + Y^{(1)}| < 2n$; however, it is possible to show that contributions where $u' = 1$ or $v' = 1$
197 are $\in O(k)$:
198   W.l.o.g. for $u' > 1$, $\lfloor (u', 1) \rfloor$ is inserted into H only when $\lfloor (u' - 1, 1) \rfloor$ is popped. Thus at most
199 one $\lfloor (u', 1) \rfloor$ can exist in H at any time. Furthermore, popping $\lfloor (u', 1) \rfloor$ from H requires previously
200 popping $\lceil (u' - 1, 1) \rceil$ from H: layer ordering on X implies $\max(X^{(u'-1)}) \leq \min(X^{(u')})$ and $|Y^{(1)}| = 1|$
201 implies $\min(Y^{(1)}) = \max(Y^{(1)})$, and so $\lceil (u' - 1, 1) \rceil = (\max(X^{(u'-1)} + Y^{(1)}), (u' - 1, 1), 1) < \lfloor (u', 1) \rfloor =$
202 $(\min(X^{(u')} + Y^{(1)}), (u', 1), 0)$. Thus $\lceil (u' - 1, 1) \rceil$ has been popped from H and counted in s. By the
203 exponential growth of layers, the contribution of all such $u' > 1, v' = 1$ will be $\approx \leq \alpha \cdot s$, and so the
204 contributions of $u' > 1, v' = 1$ or $u' = 1, v' > 1$ will be $\approx \leq 2\alpha \cdot s$.
205   When $u' = v' = 1$, the layer product contains 1 element.
206   Therefore, s', the total number of elements found in layer products appended to q during phase 2, has
207 $s' \leq (\alpha^2 + 2\alpha) \cdot s + 1$. By lemma 2.6, $s \in O(k)$, and thus $s' \in O(k)$. $\square$ $\square$

	Naive $n^2 \log(n) + k$	Kaplan *et al.* soft heap	Layer-ordered heap (total=phase 0+phases 1–3)
$n = 1000, k = 250$	0.939	0.0511	0.00892=0.00693+0.002
$n = 1000, k = 500$	0.952	0.099	0.0102=0.00648+0.00374
$n = 1000, k = 1000$	0.973	0.201	0.014=0.00764+0.00639
$n = 1000, k = 2000$	0.953	0.426	0.0212=0.00652+0.0146
$n = 1000, k = 4000$	0.950	0.922	0.0278=0.00713+0.0206
$n = 2000, k = 500$	4.31	0.104	0.0194=0.0160+0.00342
$n = 2000, k = 1000$	4.11	0.203	0.0211=0.0139+0.00728
$n = 2000, k = 2000$	4.17	0.432	0.0254=0.0140+0.0114
$n = 2000, k = 4000$	4.16	0.916	0.0427=0.0147+0.0280
$n = 2000, k = 8000$	4.13	2.03	0.0761=0.0143+0.0617
$n = 4000, k = 1000$	17.2	0.207	0.0507=0.0459+0.00488
$n = 4000, k = 2000$	17.2	0.422	0.409=0.0268+0.0141
$n = 4000, k = 4000$	17.1	0.907	0.0481=0.0277+0.0205
$n = 4000, k = 8000$	17.3	1.98	0.0907=0.0278+0.0629
$n = 4000, k = 16000$	17.3	4.16	0.133=0.0305+0.103

Table 1. Average runtimes (in seconds) on random uniform integer X and Y with $|X| = |Y| = n$. The layer-ordered heap implementation used $\alpha = 2$ and resulted in $\frac{s+s'}{k} = 3.637$ on average. Individual and total runtimes are rounded to three significant figures.

Theorem 2.8. *The total runtime of the algorithm is* $\in O(n+k)$.

Proof. For any constant $\alpha > 1$, LOHification of X and Y runs in linear time, and so phase 0 runs $\in O(n)$. The total number of layers in each LOH is $\approx \log_\alpha(n)$; therefore, the total number of layer products is $\approx \log_\alpha^2(n)$. In the worst-case scenario, the heap insertions and pops (and corresponding set insertions and removals) will sort $\approx 2\log_\alpha^2(n)$ elements, because each layer product may be inserted as both $\lfloor \cdot \rfloor or \lceil \cdot \rceil$; the worst-case runtime via comparison sort will be $\in O(\log_\alpha^2(n) \log(\log_\alpha^2(n))) \subset o(n)$. The operations to maintain a set of indices in the heap have the same runtime per operation as those inserting/removing to a binary heap, and so can be amortized out. Thus, the runtimes of phases 1–2 are amortized out by the $O(n)$ runtime of phase 0.

Lemma 2.6 shows that $s \in O(k)$. Likewise, lemma 2.7 shows that $s' \in O(k)$. The number of elements in all layer products in q during phase 3 is $s + s' \in O(k)$. Thus, the number of elements on which the one-dimensional selection is performed will be $\in O(k)$. Using a linear time one-dimensional selection algorithm, the runtime of the k-selection in phase 3 is $\in O(k)$.

The total runtime of all phases $\in O(n + k + k + k) = O(n+k)$. $\square$

$\square$

3 RESULTS

Runtimes of the naive $O(n^2 \log(n))$ method (chosen for reference because it is the easiest method to implement and because of the fast runtime constant on `python`'s built-in sorting routine), the soft heap-based method from Kaplan *et al.*, and the LOH-based method in this paper are shown in table 1. The proposed approach achieves a $> 295\times$ speedup over the naive approach and $> 18\times$ speedup over the soft heap approach. LOHs are more lightweight than soft heaps, including contiguous memory access patterns and far fewer pointer dereferences than soft heaps.

4 DISCUSSION

The algorithm can be thought of as "zooming out" as it pans through the layer products, thereby passing the value threshold at which the kth best value $X_i + Y_j$ occurs. It is somewhat reminiscent of skip lists (Pugh, 1990); however, where a skip list begins coarse and progressively refines the search, this approach begins finely and becomes progressively coarser. The notion of retrieving the best k values while "overshooting" the target by as little as possible results in some values that may be considered but which will not survive the final one-dimensional selection in phase 3. This is reminiscent of "corruption" in Chazelle's soft heaps. Like soft heaps, this method eschews sorting in order to prevent a runtime $\in \Omega(n\log(n))$ or $\in \Omega(k\log(k))$. But unlike soft heaps, LOHs can be constructed easily using only an implementation of median-of-medians (or any other linear time one-dimensional selection algorithm).

240 Phase 3 is the only part of the algorithm in which k appears in the runtime formula. This is significant
241 because the layer products in q at the end of phase 2 could be returned in their compressed form (*i.e.*,
242 as the two layers to be combined). The total runtime of phases 0–2 is $\in O(n)$. It may be possible to
243 recursively perform $X + Y$ selection on layer products $X^{(u)} + Y^{(v)}$ to compute layer products constituting
244 exactly the k values in the solution, still in factored Cartesian layer product form. Similarly, it may be
245 possible to perform the one-dimensional selection without fully inflating every layer product into its
246 constituent elements. For some applications, a compressed form may be acceptable, thereby making it
247 plausible to remove the requirement that the runtime be $\in \Omega(k)$.
248 As noted in theorem 2.8, even fully sorting all of the minimum and maximum layer products would
249 be $\in o(n)$; sorting in this manner may be preferred in practice, because it simplifies the implementation
250 (Listing 3) at the cost of incurring greater runtime in practice when $k \ll n^2$. Furthermore, listing 3 is
251 unsuitable for online processing (*i.e.*, where X and Y are extended on the fly or where several subsequent
252 selections are performed), whereas listing 2 could be adapted to those uses.
253 Phase 0 (which performs LOHification) is the slowest part of the presented python implementation; it
254 would benefit from having a practically faster implementation to perform LOHify.
255 The fast practical performance is partially due to the algorithm's simplicity and partially due to the
256 contiguous nature of LOHs. Online data structures like soft heap are less easily suited to contiguous
257 access, because they support efficient removal and therefore move pointers to memory rather than moving
258 the contents of the memory.
259 The choice of α affects performance through the cost of LOHifying and the amount by which the
260 number of generated values overshoots the k minimum values wanted: when $\alpha \approx 1$, LOHify effectively
261 sorts X and Y, but generates few extra values; $\alpha \gg 1$, LOHify has a linear runtime, but generates more
262 extra values, which need to be removed by the final k-selection.

5 CONCLUSION

264 LOHs can be constructed in linear time and used to produce a theoretically optimal algorithm for selecting
265 the minimal k values from $X + Y$. The new optimal algorithm presented here is faster in practice than the
266 existing soft heap-based optimal algorithm.

6 ACKNOWLEDGEMENTS

268 Thanks to Patrick Kreitzberg, Kyle Lucke, and Jake Pennington for fruitful discussions and kindness.

7 DECLARATIONS

7.1 Funding

271 This work was supported by grant number 1845465 from the National Science Foundation.

7.2 Code availability

273 Python source code and LaTeX for this paper are available at `https://doi.org/10.6084/m9.`
274 `figshare.13708564` (MIT license, free for both academic and commercial use). C++ code (MIT
275 license, free for both academic and commercial use) for the same algorithm can be found in `https:`
276 `//bitbucket.org/orserang/neutronstar/`, a fast isotope calculator built with this selection
277 method.

7.3 Conflicts of interest

279 The authors declare that they have no conflicts of interest.

8 SUPPLEMENTAL INFORMATION

8.1 Python code

Listing 1. `LayerOrderedHeap.py`: A class for LOHifying, retrieving layers, and the minimum and
maximum value in a layer.

```
282  # https://stackoverflow.com/questions/10806303/python-implementation-of-median-of-
283      medians-algorithm
```

```
284  def median_of_medians_select(L, j): # returns j-th smallest value:
285    if len(L) < 10:
286      L.sort()
287      return L[j]
288    S = []
289    lIndex = 0
290    while lIndex+5 < len(L)-1:
291      S.append(L[lIndex:lIndex+5])
292      lIndex += 5
293    S.append(L[lIndex:])
294    Meds = []
295    for subList in S:
296      Meds.append(median_of_medians_select(subList, int((len(subList)-1)/2)))
297    med = median_of_medians_select(Meds, int((len(Meds)-1)/2))
298    L1 = []
299    L2 = []
300    L3 = []
301    for i in L:
302      if i < med:
303        L1.append(i)
304      elif i > med:
305        L3.append(i)
306      else:
307        L2.append(i)
308    if j < len(L1):
309      return median_of_medians_select(L1, j)
310    elif j < len(L2) + len(L1):
311      return L2[0]
312    else:
313      return median_of_medians_select(L3, j-len(L1)-len(L2))
314
315  def partition(array, left_n):
316    n = len(array)
317    right_n = n - left_n
318
319    # median_of_medians_select argument is index, not size:
320    max_value_in_left = median_of_medians_select(array, left_n-1)
321
322    left = []
323    right = []
324    for i in range(n):
325      if array[i] < max_value_in_left:
326        left.append(array[i])
327      elif array[i] > max_value_in_left:
328        right.append(array[i])
329    num_at_threshold_in_left = left_n - len(left)
330    left.extend([max_value_in_left]*num_at_threshold_in_left)
331    num_at_threshold_in_right = right_n - len(right)
332    right.extend([max_value_in_left]*num_at_threshold_in_right)
333    return left, right
334
335  def layer_order_heapify_alpha_eq_2(array):
336    n = len(array)
337    if n == 0:
338      return []
339    if n == 1:
340      return array
341    new_layer_size = 1
342    layer_sizes = []
343    remaining_n = n
344    while remaining_n > 0:
345      if remaining_n >= new_layer_size:
346        layer_sizes.append(new_layer_size)
347      else:
348        layer_sizes.append(remaining_n)
349      remaining_n -= new_layer_size
350      new_layer_size *= 2
351    result = []
352    for i,ls in enumerate(layer_sizes[::-1]):
353      small_vals,large_vals = partition(array, len(array) - ls)
```

```
354        array = small_vals
355        result.append(large_vals)
356      return result[::-1]
357
358  class LayerOrderedHeap:
359    def __init__(self, array):
360      self._layers = layer_order_heapify_alpha_eq_2(array)
361      self._min_in_layers = [ min(layer) for layer in self._layers ]
362      self._max_in_layers = [ max(layer) for layer in self._layers ]
363      #self._verify()
364
365    def __len__(self):
366      return len(self._layers)
367
368    def _verify(self):
369      for i in range(len(self)-1):
370        assert(self.max(i) <= self.min(i+1))
371
372    def __getitem__(self, layer_num):
373      return self._layers[layer_num]
374
375    def min(self, layer_num):
376      assert( layer_num < len(self) )
377      return self._min_in_layers[layer_num]
378
379    def max(self, layer_num):
380      assert( layer_num < len(self) )
381      return self._max_in_layers[layer_num]
382
383    def __str__(self):
384      return str(self._layers)
```

Listing 2. `CartesianSumSelection.py`: A class for efficiently performing selection on $X + Y$ in $\Theta(n + k)$ steps.

```
385  from LayerOrderedHeap import *
386  import heapq
387
388  class CartesianSumSelection:
389    def _min_tuple(self,i,j):
390      # True for min corner, False for max corner
391      return (self._loh_a.min(i) + self._loh_b.min(j), (i,j), False)
392
393    def _max_tuple(self,i,j):
394      # True for min corner, False for max corner
395      return (self._loh_a.max(i) + self._loh_b.max(j), (i,j), True)
396
397    def _in_bounds(self,i,j):
398      return i < len(self._loh_a) and j < len(self._loh_b)
399
400    def _insert_min_if_in_bounds(self,i,j):
401      if not self._in_bounds(i,j):
402        return
403
404      if (i,j,False) not in self._hull_set:
405        heapq.heappush(self._hull_heap, self._min_tuple(i,j))
406        self._hull_set.add( (i,j,False) )
407
408    def _insert_max_if_in_bounds(self,i,j):
409      if not self._in_bounds(i,j):
410        return
411
412      if (i,j,True) not in self._hull_set:
413        heapq.heappush(self._hull_heap, self._max_tuple(i,j))
414        self._hull_set.add( (i,j,True) )
415
416    def __init__(self, array_a, array_b):
417      self._loh_a = LayerOrderedHeap(array_a)
418      self._loh_b = LayerOrderedHeap(array_b)
419
```

```
420        self._hull_heap = [ self._min_tuple(0,0) ]
421        # False for min:
422        self._hull_set = { (0,0,False) }
423
424        self._num_elements_popped = 0
425        self._layer_products_considered = []
426
427        self._full_cartesian_product_size = len(array_a) * len(array_b)
428
429     def _pop_next_layer_product(self):
430        result = heapq.heappop(self._hull_heap)
431        val, (i,j), is_max = result
432        self._hull_set.remove( (i,j,is_max) )
433
434        if not is_max:
435          # when min corner is popped, push their own max and neighboring mins
436          self._insert_min_if_in_bounds(i+1,j)
437          self._insert_min_if_in_bounds(i,j+1)
438          self._insert_max_if_in_bounds(i,j)
439        else:
440          # when max corner is popped, do not push
441          self._num_elements_popped += len(self._loh_a[i]) * len(self._loh_b[j])
442          self._layer_products_considered.append( (i,j) )
443
444        return result
445
446     def select(self, k):
447        assert( k <= self._full_cartesian_product_size )
448
449        while self._num_elements_popped < k:
450          self._pop_next_layer_product()
451
452        # also consider all layer products still in hull
453        for val, (i,j), is_max in self._hull_heap:
454          if is_max:
455            self._num_elements_popped += len(self._loh_a[i]) * len(self._loh_b[j])
456            self._layer_products_considered.append( (i,j) )
457
458        # generate: values in layer products
459
460        # Note: this is not always necessary, and could lead to a potentially large
461            speedup.
462        candidates = [ val_a+val_b for i,j in self._layer_products_considered for val_a in
463            self._loh_a[i] for val_b in self._loh_b[j] ]
464        print( 'Ratio_of_total_popped_candidates_to_k:_{}'.format(len(candidates) / k) )
465        k_small_vals, large_vals = partition(candidates, k)
466        return k_small_vals
```

Listing 3. `SimplifiedCartesianSumSelection.py`: A simplified implementation of Listing 2. This implementation is slower when $k \ll n^2$; however, it has the same asymptotic runtime for any n and k: $\Theta(n+k)$.

```
467   from LayerOrderedHeap import *
468
469   class SimplifiedCartesianSumSelection:
470     def _min_tuple(self,i,j):
471        # True for min corner, False for max corner
472        return (self._loh_a.min(i) + self._loh_b.min(j), (i,j), False)
473
474     def _max_tuple(self,i,j):
475        # True for min corner, False for max corner
476        return (self._loh_a.max(i) + self._loh_b.max(j), (i,j), True)
477
478     def __init__(self, array_a, array_b):
479        self._loh_a = LayerOrderedHeap(array_a)
480        self._loh_b = LayerOrderedHeap(array_b)
481
482        self._full_cartesian_product_size = len(array_a) * len(array_b)
483
```

```
484    self._sorted_corners = sorted([self._min_tuple(i,j) for i in range(len(self._loh_a
485        )) for j in range(len(self._loh_b))] + [self._max_tuple(i,j) for i in range(
486        len(self._loh_a)) for j in range(len(self._loh_b))])
487
488  def select(self, k):
489    assert( k <= self._full_cartesian_product_size )
490
491    candidates = []
492    index_in_sorted = 0
493    num_elements_with_max_corner_popped = 0
494    while num_elements_with_max_corner_popped < k:
495      val, (i,j), is_max = self._sorted_corners[index_in_sorted]
496      new_candidates = [ v_a+v_b for v_a in self._loh_a[i] for v_b in self._loh_b[j] ]
497      if is_max:
498        num_elements_with_max_corner_popped += len(new_candidates)
499      else:
500        # Min corners will be popped before corresponding max corner;
501        # this gets a superset of what is needed (just as in phase 2)
502        candidates.extend(new_candidates)
503      index_in_sorted += 1
504
505    print( 'Ratio of total popped candidates to k: {}'.format(len(candidates) / k) )
506    k_small_vals, large_vals = partition(candidates, k)
507    return k_small_vals
```

REFERENCES

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

Chazelle, B. (2000). The soft heap: an approximate priority queue with optimal error rate. *Journal of the ACM (JACM)*, 47(6):1012–1027.

Frederickson, G. N. (1993). An optimal algorithm for selection in a min-heap. *Information and Computation*, 104(2):197–214.

Frederickson, G. N. and Johnson, D. B. (1982). The complexity of selection and ranking in $X+Y$ and matrices with sorted columns. *Journal of Computer and System Sciences*, 24(2):197–208.

Kaplan, H., Kozma, L., Zamir, O., and Zwick, U. (2019). Selection from heaps, row-sorted matrices and $X+Y$ using soft heaps. *Symposium on Simplicity in Algorithms*, pages 5:1–5:21.

Kreitzberg, P., Lucke, K., and Serang, O. (2020a). Selection on $X_1+X_2+\cdots+X_m$ with layer-ordered heaps. *arXiv preprint arXiv:1910.11993*. In review.

Kreitzberg, P., Pennington, J., Lucke, K., and Serang, O. (2020b). Fast exact computation of the k most abundant isotope peaks with layer-ordered heaps. *Analytical Chemistry*, 92(15):10613–10619.

Pennington, J., Kreitzberg, P., Lucke, K., and Serang, O. (2020). Optimal construction of a layer-ordered heap.

Pugh, W. (1990). Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676.