Optimal selection on $X + Y$ simplified with layer-ordered heaps

Oliver Serang*
University of Montana
Department of Computer Science
32 Campus Drive, Missoula, MT
United States of America
oliver.serang@umontana.edu*

February 1, 2020

Abstract
Selection on the Cartesian sum, $A + B$, is a classic and important problem. Frederickson’s 1993 algorithm produced the first algorithm that made possible an optimal runtime. Kaplan et al.’s recent 2018 paper described an alternative optimal algorithm by using Chazelle’s soft heaps. These extant optimal algorithms are very complex; this complexity can lead to difficulty implementing them and to poor performance in practice. Here, a new optimal algorithm is presented, which uses layer-ordered heaps. This new algorithm is both simple to implement and practically efficient.

*Corresponding author.
1 Introduction

Given two vectors of length $n$, $A$ and $B$, $k$-selection on $A + B$ finds the $k$ smallest values of the form $A_i + B_j$. In 1982, Frederickson & Johnson introduced a method reminiscent of median-of-medians\(^1\); their method runs $\in O(n + \min(n, k) \log(\frac{k}{\min(n, k)}))$\(^2\).

1.1 Optimal method of Frederickson

Frederickson subsequently published the first optimal ($i.e., \in O(n + k)$) algorithm\(^3\). This method uses a tree data structure similar to what would in 2000 be formalized into Chazelle’s soft heap\(^2\), and can be combined with a combinatoric heap to compute the $k$ minimal values in $A + B$.

1.2 Optimal method of Kaplan et al.

Kaplan et al. described an alternative optimal method; that method explicitly used Chazelle’s soft heaps\(^2\). By heapifying $A$ and $B$ in linear time ($i.e.,$ guaranteeing w.l.o.g. that $A_i \leq A_{2i}, A_{2i+1}$), $\min_i j A_i + B_j = A_1 + B_1$. Likewise, $A_i + B_j \leq A_{2i} + B_j, A_{2i+1} + B_j, A_i + B_{2j}, A_i + B_{2j+1}$. The soft heap is initialized to contain tuple $(A_1 + B_1, 1, 1)$. 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

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

(1)

In the matrix $A_i + B_j$ (which is not realized), this scheme progresses in row-major order, thereby avoiding a tuple being added multiple times.

Even though only the minimal $k$ values are desired, “corruption” in the soft heap means that the soft heap will not always pop the minimal value; however, as a result, soft heaps can run faster than the $\Omega(n \log(n))$ bound on comparison sorting. A free parameter to the soft heap, $\epsilon \in (0, 1)$, bounds that the number of corrupted elements in the soft heap (which may be promoted earlier in the queue than they should be) is bounded to be $\leq t \cdot \epsilon$, where $t$ is the number of elements in the soft heap. Thus, instead of popping $k$ items (and inserting their lower-quality dependents as described in equation\(^1\)), the total number of pops $p$ can be found: The maximal size of the soft heap after $p$ pops is $\leq 3p$ (because each pop removes one element and inserts $\leq 4$ elements according to equation\(^1\); therefore, $p - \text{corruption} \geq p - 4p \cdot \epsilon$, and thus $p - 4p \cdot \epsilon \geq k$ guarantees that $p - \text{corruption} \geq k$. This leads to $p = \frac{k}{1 - 4\epsilon}$, $\epsilon < \frac{1}{4}$. This guarantees that $\Theta(k)$ values, which must include the minimal $k$ values, are popped. These values are post-processed to retrieve the minimal $k$ values via linear time one-dimensional selection\(^1\). For constant $\epsilon$, both pop and insertion operations to the soft heap are $\in (1)$, and thus the overall runtime of the algorithm is $\in O(n + k)$.

1.3 Layer-ordered heaps and a novel selection algorithm on $A + B$

This paper uses layer-ordered heaps (LOHs)\(^5\) to produce an optimal selection algorithm on $A + B$. LOHs are stricter than heaps but not as strict as sorting: Heaps guarantee only that $A_i \leq A_{\text{child}(i)}$, but do not guarantee any ordering between one child of $A_i$, $x$, and the child of the sibling of $x$. Sorting is stricter still, but sorting $n$ values cannot be done faster than $\log_2(n!) \in \Omega(n \log(n))$.

LOHs partition the array into several layers such that the values in a layer are $\leq$ to the values in subsequent layers: $A^{(u)} = A^{(u)}_1, A^{(u)}_2, \ldots \leq A^{(u+1)}$. The size of these layers starts with $A^{(1)} = 1$ and grows exponentially such that $\lim_{i \to \infty} \frac{|A^{(u+1)}|}{|A^{(u)}|} = \alpha \geq 1$ (note that $\alpha = 1$ is equivalent to sorting


because all layers have size 1). By assigning values in layer \( u \) children from layer \( u + 1 \), this can be seen as a more constrained form of heap; however, unlike sorting, for any constant \( \alpha > 1 \), LOHs can be constructed \( \in O(n) \) by performing iterative linear time one-dimensional selection, iteratively selecting and removing the largest layer until all layers have been partitioned.

LOHs were first used in conjunction with a soft heap scheme to perform selection on the high-dimensional \( X_1 + X_2 + \cdots + X_m \) [5].

The optimal algorithm for selection on \( A + B \) proposed in this paper is simple to implement, does not rely on anything more complicated than linear time one-dimensional selection, and has fast performance in practice.

2 Methods

2.1 Algorithm

2.1.1 Phase 0

The algorithm first LOHifies \( A \) and \( B \). This is performed by using linear time one-dimensional selection to iteratively remove the largest remaining layer.

2.1.2 Phase 1

Now layer products of the form \( A^{(u)} + B^{(v)} = A_1^{(u)} + B_1^{(v)}, A_1^{(u)} + B_2^{(v)}, \ldots A_2^{(u)} + B_1^{(v)}, \ldots \) are considered, where \( A^{(u)} \) and \( B^{(v)} \) are layers of their respective LOHs.

In phases 1–2, the algorithm initially considers only the minimum and maximum values in each layer product: \([\langle u, v \rangle] = (\min(A^{(u)} + B^{(v)}), (u, v), false), [\langle u, v \rangle] = (\max(A^{(u)} + B^{(v)}), (u, v), true)\). Note that \( false \) is used to indicate that this is the minimal value in the layer product, while \( true \) indicates the maximum value in the layer product. Let \( false = 0, true = 1 \) so that \( [\langle u, v \rangle] < [\langle u, v \rangle] \).

Scalar values can be compared to tuples: \( A_i + B_j \leq [\langle u, v \rangle] = (\max(A^{(u)} + B^{(v)}), (u, v), true) \leftrightarrow A_i + B_j \leq \max(A^{(u)} + B^{(v)})\).

Heap \( H \) is initialized to contain tuple \([\langle 1, 1 \rangle] \). A set of all tuples in \( H \) is maintained to prevent duplicates from being inserted into \( H \). The algorithm proceeds by popping the lexicographically minimum tuple from \( H \). W.l.o.g., there is not guaranteed ordering of the form \( A^{(u)} + B^{(v)} \leq A^{(u+1)} + B^{(v)} \), because it may be that \( \max(A^{(u)} + B^{(v)}) > \min(A^{(u+1)} + B^{(v)}) \); however, lexicographically, \([\langle u, v \rangle] < [\langle u + 1, v \rangle] \), \([\langle u, v + 1 \rangle] \), \([\langle u, v \rangle] \); thus, the latter tuples need be inserted into \( H \) only after \([\langle u, v \rangle] \) has been popped from \( H \). \([\langle u, v \rangle] \) tuples do not insert any new tuples into \( H \) when they’re popped.

Whenever a tuple of the form \([\langle u, v \rangle] \) is popped from \( H \), the index \( (u, v) \) is appended to list \( q \) and the size of the layer product \( |A^{(u)} + B^{(v)}| = |A^{(u)}| \cdot |B^{(v)}| \) is accumulated into integer \( s \). This method proceeds until that accumulated value \( s \geq k \).

2.1.3 Phase 2

Any remaining tuple in \( H \) of the form \([\langle u', v' \rangle] = (u', v'), true) \) 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 \( q \) is generated. A linear time one-dimensional \( k \)-selection is performed on these values and returned.

2.2 Proof of correctness

Lemma 3 proves that at termination all layer products found in \( q \) must contain the minimal \( k \) values in \( A + B \). Thus, by performing one-dimensional \( k \)-selection on those values in phase 3, the minimal \( k \) values in \( A + B \) are found.

Lemma 1. If \([ (u, v) \] is popped from \( H \), then both \([ (u-1, v) \] (if \( u > 1 \)) and \([ (u, v-1) \] (if \( v > 1 \)) must previously have been popped from \( H \).

Proof. There is a chain of pops and insertions backwards from \([ (u, v) \] to \([ (1, 1) \). This chain must include structures of pops of the form \([ (a-1, b-1) \], \([ (a, b-1) \], \([ (a, b) \] or \([ (a-1, b-1) \], \([ (a-1, b) \], \([ (a, b) \]. W.l.o.g., pops of \([ (a-1, b-1) \], \([ (a, b-1) \], \([ (a, b) \] mean that \([ (a, b) \] would be inserted into \( H \) before \([ (a, b) \], and since \([ (a, b-1) \] < \([ (a, b) \], it must be popped before \([ (a, b) \]. By that reasoning, \([ (u-1, v) \] and \([ (u, v-1) \] must be popped before \([ (u, v) \].

□

Lemma 2. If \([ (u, v) \] is popped from \( H \), then both \([ (u-1, v) \] (if \( u > 1 \)) and \([ (u, v-1) \] (if \( v > 1 \)) must previously have been popped from \( H \).

Proof. Inserting \([ (u, v) \] requires previously popping \([ (u, v) \]. By lemma 1 this requires previously popping \([ (u-1, v) \] (if \( u > 1 \)) and \([ (u, v-1) \] (if \( v > 1 \). These pops will insert \([ (u-1, v) \] and \([ (u, v-1) \] respectively. Thus, \([ (u-1, v) \] and \([ (u, v-1) \], which are both < \([ (u, v) \], are inserted before \([ (u, v) \], and will therefore be popped before \([ (u, v) \].

□

Lemma 3. Minimum and maximum tuples from all layer products will be popped from \( H \) in ascending order.

Proof. Let \([ (u, v) \] be popped from \( H \) and let \([ (a, b) \] < \([ (u, v) \]. Either w.l.o.g. \( a < u, b \leq v \), or w.l.o.g. \( a < u, b > v \). In the former case, \([ (a, b) \] will be popped before \([ (u, v) \] by applying induction to lemma 1.

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

Ordering on popping with \([ (a, b) \] < \([ (u, v) \] is shown in the same manner: For \([ (u, v) \] to be in \( H \), \([ (u, v) \] must have previously been popped. As above, whenever \([ (u, v) \] is in \( H \) at least one \([ (a, r) \], v \geq r \leq b \) must also be in \( H \) until \([ (a, b) \] is popped. These \([ (a, r) \] \leq \([ (a, b) \] \leq \([ (u, v) \), and so \([ (a, b) \] will be popped before \([ (u, v) \].

Identical reasoning also shows that \([ (a, b) \] will pop before \([ (u, v) \] if \([ (a, b) \] < \([ (u, v) \] or if \([ (a, b) \] < \([ (u, v) \].

Thus, all tuples are popped in ascending order.

□
Lemma 7. 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 \(A + B\) that are \(\leq \max(A^{(u)} + B^{(v)})\); this means that \(\tau = \max(\text{select}(A + B, k)) \leq \max(A^{(u)} + B^{(v)})\). 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 \(< \max(A^{(u)} + B^{(v)})\).

By lemma 3, tuples will be popped from \(H\) in ascending order; therefore, any layer product \((u', v')\) containing values \(< \max(A^{(u)} + B^{(v)})\) must have had \([u', v']\) popped before \([u, v]\). If \([u', v']\) was also popped, then this layer product is already included in \(q\) and cannot improve it. Thus the only layers that need be considered further have had \([u', v']\) popped but not \([u', v']\) popped; these can be found by looking for all \([u', v']\) that have been inserted into \(H\) but not yet popped.

Phase 2 appends to \(q\) all such remaining layer products of interest. Thus, at the end of phase 2, \(q\) contains all layer products that will be represented in the \(k\)-selection of \(A + B\).

\(\square\)

2.3 Runtime

Theorem 1 proves that the total runtime is \(\in O(n + k)\).

Lemma 5. Let \((u', v')\) be a layer product appended to \(q\) during phase 2. Either \(u' = 1, v' = 1\), or \((u' - 1, v' - 1)\) was already appended to \(q\) in phase 1.

Proof. Let \(u' > 1\) and \(v' > 1\). By lemma 3, minimum and maximum layer products are popped in ascending order. By the layer ordering property of \(A\) and \(B\), \(\max(A^{(u' - 1)}) \leq \min(A^{(u')})\) and \(\max(B^{(v' - 1)}) \leq \min(B^{(v')})\). Thus, \([u' - 1, v' - 1]\) is the layer product whose inclusion during phase 1 in \(q\) achieves \(s \geq k\); therefore, \(s - |A^{(u')} + B^{(v')}| < k\). This happens when \([u, v]\) is popped from \(H\).

If \(k = 1\), popping \([1, 1]\) ends phase 1 with \(s = 1 \in O(k)\).

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 1, popping \([u, v]\) from \(H\) requires previously popping \([u - 1, v]\). \(\max(|A^{(u)} + B^{(v)}|) = |A^{(u)}| \cdot |B^{(v)}| = \alpha \cdot |A^{(u - 1)}| \cdot |B^{(v)}| = \alpha \cdot |A^{(u - 1)} + B^{(v)}|\); therefore, \(|A^{(u)} + B^{(v)}| \in O(|A^{(u - 1)} + B^{(v)}|)\). \(A^{(u - 1)} + B^{(v)}\) is already counted in \(s - |A^{(u')} + B^{(v')}| < k\), and so \(A^{(u - 1)} + B^{(v')} < k\) and \(|A^{(u')} + B^{(v')}| \in O(k)\). \(s < k + |A^{(u')} + B^{(v')}| \in O(k)\) and hence \(s \in O(k)\).

\(\square\)

Lemma 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, \(s - |A^{(u)} + B^{(v)}| < k\). This happens when \([u, v]\) is popped from \(H\).

If \(k = 1\), popping \([1, 1]\) ends phase 1 with \(s = 1 \in O(k)\).

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 1, popping \([u, v]\) from \(H\) requires previously popping \([u - 1, v]\). \(\max(|A^{(u)} + B^{(v)}|) = |A^{(u)}| \cdot |B^{(v)}| = \alpha \cdot |A^{(u - 1)}| \cdot |B^{(v)}| = \alpha \cdot |A^{(u - 1)} + B^{(v)}|\); therefore, \(|A^{(u)} + B^{(v)}| \in O(|A^{(u - 1)} + B^{(v)}|)\). \(A^{(u - 1)} + B^{(v)}\) is already counted in \(s - |A^{(u')} + B^{(v')}| < k\), and so \(|A^{(u - 1)} + B^{(v')}| < k\) and \(|A^{(u')} + B^{(v')}| \in O(k)\). \(s < k + |A^{(u')} + B^{(v')}| \in O(k)\) and hence \(s \in O(k)\).

\(\square\)

Lemma 7. \(s'\), the total number of elements in all layer products appended to \(q\) in phase 2, is \(\in O(n + k)\).

Proof. Each layer product appended to \(q\) in phase 2 has had \([u', v']\) popped in phase 1. By lemma 5, either \(u' = 1\) or \(v' = 1\) or \([u' - 1, v' - 1]\) must have been popped before \([u', v']\).

First consider when \(u' > 1\) and \(v' > 1\). Each \((u', v')\) matches to exactly one layer product \((u' - 1, v' - 1)\). Because \([u' - 1, v' - 1]\) must have been popped before \([u', v']\), then \([u' - 1, v' - 1]\) was also popped during phase 1. \(s\), the count of all elements whose layer products were inserted into
Table 1: Average runtimes on random uniform integer $A$ and $B$ with $n = k = 4096$. The layer-ordered heap implementation used $\alpha = 2$ and resulted in $\frac{4s + 2}{k} = 3.438$ on average.

$|A(u') + B(v')|$ but does not include $A(u') + B(v')$ (the latter is appended to $q$ during phase 2). By exponential growth of layers in $A$ and $B$, $|A(u') + B(v')| \approx 2 \cdot |A(u'' - 1) + B(v'' - 1)|$. These $|A(u'' - 1) + B(v'' - 1)|$ values were included in $s$ during phase 1, and thus the total number of elements in all such $(u' - 1, v' - 1)$ layer products is $\leq s$. Thus the sum of sizes of all layer products $(u', v')$ with $u' > 1$ and $v' > 1$ that are appended to $q$ during phase 2 is $\approx \leq 2 \cdot s$. By lemma 6, $s \in O(k)$, and so the contribution of all such $u' > 1, v' > 1$ layers added in phase 2 is $\in O(k)$.

The maximum possible contributions from any $u' = 1$ or $v' = 1$ are found by $\sum\{A(u') + B(v')\} + \sum\{A(u') + B(v')\} = 2n \in O(n)$.

Therefore, $s'$, the total number of elements found in layer products appended to $q$ during phase 2, is $\in O(n + k)$.

**Theorem 1.** The total runtime of the algorithm is $\in O(n + k)$.

**Proof.** For any constant $\alpha > 1$, LOHification of $A$ and $B$ 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 $\approx 2 \cdot \log_\alpha^2(n)$ elements, because each layer product may be inserted as both $\cdot |\cdot|$ or $\cdot [\cdot]$; the worst-case runtime via comparison sort will be $\in O(\log_\alpha^2(n) \log(\log_\alpha^2(n))) \subset o(n)$. Thus, the runtimes of phases 1–2 are amortized out by the $O(n)$ runtime of phase 0.

Lemma 6 shows that $s \in O(k)$. Likewise, lemma 7 shows that $s' \in O(n + k)$. The number of elements in all layer products in $q$ during phase 3 is $s + s' \in O(n + k)$. Thus, the number of elements on which the one-dimensional selection is performed will be $\in O(n + k)$. Using a linear time one-dimensional selection algorithm, the runtime of the $k$-selection in phase 3 is $\in O(n + k)$.

The total runtime of all phases is dominated by phase 3, and is thus $\in O(n + k)$.

### 2.4 Space

Space $\leq$ time, because each unit of work can only allocate constant space. Thus the space usage is $\in O(n + k)$.

### 3 Results

Runtimes of the naive $O(n^2 \log(n) + k)$ method, 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.
4 Discussion

The algorithm can be thought of as “zooming out” as it pans through the layer products, thereby passing the unknown goal threshold $\tau$ by very little. It is somewhat reminiscent of skip lists\cite{6}; 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).

Phase 3 is the only part of the algorithm in which $k$ appears in the runtime formula. This is significant because the layer products in $q$ at the end of phase 2 could be returned in their compressed form (i.e., as the two layers to be combined). The total runtime of phases 0–2 is $\in O(n)$. It may be possible to recursively perform $A + B$ selection on layer products $A(u) + B(v)$ to compute layer products constituting exactly the $k$ values in the solution, still in factored Cartesian layer product form. Similarly, it may be possible to perform the one-dimensional selection without fully inflating every layer product into its constituent elements. For some applications, a compressed form may be acceptable, thereby removing $k$ from the runtime.

As noted in theorem\cite{1} even fully sorting all of the minimal and maximum layer products would be $\in o(n)$; thus, this may be preferred in practice, because it could further simplify implementation and lead to a better in-practice runtime (compared to using a heap). Similarly, phase 0 (which performs LOHification) is the slowest part of the current implementation; it would benefit from having a practically faster implementation to perform LOHify.

5 Availability

Python source code and LATEX for this paper are available at https://bitbucket.org/orserang/selection-on-cartesian-product/ (MIT license, free for both academic and commercial use).

6 Acknowledgment

This work was supported by grant number 1845465 from the National Science Foundation. Thanks to Patrick Kreitzberg, Kyle Lucke, and Jake Pennington for fruitful discussions and kindness.

7 Supplemental information

7.1 Python code

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

```python
def median_of_medians_select(L, j): # returns j-th smallest value:
    if len(L) < 10:
```
L.sort()
    return L[j]
S = []
lIndex = 0
while lIndex+5 < len(L)-1:
    S.append(L[lIndex:lIndex+5])
    lIndex += 5
S.append(L[lIndex:])
Meds = []
for subList in S:
    Meds.append(median_of_medians_select(subList, int((len(subList)-1)/2)))
med = median_of_medians_select(Meds, int((len(Meds)-1)/2))
L1 = []
L2 = []
L3 = []
for i in L:
    if i < med:
        L1.append(i)
    elif i > med:
        L3.append(i)
    else:
        L2.append(i)
if j < len(L1):
    return median_of_medians_select(L1, j)
elif j < len(L2) + len(L1):
    return L2[0]
else:
    return median_of_medians_select(L3, j-len(L1)-len(L2))

def partition(array, left_n):
    n = len(array)
    right_n = n - left_n
    # median_of_medians_select argument is index, not size:
    max_value_in_left = median_of_medians_select(array, left_n-1)
    left = []
    right = []
    for i in range(n):
        if array[i] < max_value_in_left:
            left.append(array[i])
        elif array[i] > max_value_in_left:
            right.append(array[i])
    num_at_threshold_in_left = left_n - len(left)
    left.extend([max_value_in_left]*num_at_threshold_in_left)
    num_at_threshold_in_right = right_n - len(right)
    right.extend([max_value_in_left]*num_at_threshold_in_right)
    return left, right

def layer_order_heapify_alpha_eq_2(array):
    n = len(array)
    if n == 0:
        return []
    if n == 1:
        return array
    new_layer_size = 1
    layer_sizes = []
    remaining_n = n
    while remaining_n > 0:
if remaining_n >= new_layer_size:
    layer_sizes.append(new_layer_size)
else:
    layer_sizes.append(remaining_n)
remaining_n -= new_layer_size
new_layer_size *= 2
result = []
for i, ls in enumerate(layer_sizes[::-1]):
    small_vals, large_vals = partition(array, len(array) - ls)
    array = small_vals
    result.append(large_vals)
return result[::-1]

class LayerOrderedHeap:
    def __init__(self, array):
        self._layers = layer_order_heapify_alpha_eq_2(array)
        self._min_in_layers = [min(layer) for layer in self._layers]
        self._max_in_layers = [max(layer) for layer in self._layers]
        #self._verify()
    def __len__(self):
        return len(self._layers)
    def _verify(self):
        for i in range(len(self)-1):
            assert(self.max(i) <= self.min(i+1))
    def __getitem__(self, layer_num):
        return self._layers[layer_num]
    def min(self, layer_num):
        assert(layer_num < len(self))
        return self._min_in_layers[layer_num]
    def max(self, layer_num):
        assert(layer_num < len(self))
        return self._max_in_layers[layer_num]
    def __str__(self):
        return str(self._layers)

Listing 2: CartesianSumSelection.py: A class for efficiently performing selection on $A + B$.

class CartesianSumSelection:
    def _min_tuple(self, i, j):
        # True for min corner, False for max corner
        return (self._loh_a.min(i) + self._loh_b.min(j), (i, j), False)
    def _max_tuple(self, i, j):
        # True for min corner, False for max corner
        return (self._loh_a.max(i) + self._loh_b.max(j), (i, j), True)
    def _in_bounds(self, i, j):
        return i < len(self._loh_a) and j < len(self._loh_b)
    def _insert_min_if_in_bounds(self, i, j):
if not self._in_bounds(i,j):
    return

if (i,j,False) not in self._hull_set:
    heapq.heappush(self._hull_heap, self._min_tuple(i,j))
    self._hull_set.add( (i,j,False) )

def _insert_max_if_in_bounds(self,i,j):
    if not self._in_bounds(i,j):
        return

    if (i,j,True) not in self._hull_set:
        heapq.heappush(self._hull_heap, self._max_tuple(i,j))
        self._hull_set.add( (i,j,True) )

def __init__(self, array_a, array_b):
    self._loh_a = LayerOrderedHeap(array_a)
    self._loh_b = LayerOrderedHeap(array_b)

    self._hull_heap = [ self._min_tuple(0,0) ]
    # False for min:
    self._hull_set = { (0,0,False) }

    self._num_elements_popped = 0
    self._layer_products_considered = []

    self._full_cartesian_product_size = len(array_a) * len(array_b)

def _pop_next_layer_product(self):
    result = heapq.heappop(self._hull_heap)
    val, (i,j), is_max = result
    self._hull_set.remove( (i,j,is_max) )

    if not is_max:
        # when min corner is popped, push their own max and neighboring mins
        self._insert_min_if_in_bounds(i+1,j)
        self._insert_min_if_in_bounds(i,j+1)
        self._insert_max_if_in_bounds(i,j)
    else:
        # when max corner is popped, do not push
        self._num_elements_popped += len(self._loh_a[i]) * len(self._loh_b[j])
        self._layer_products_considered.append( (i,j) )

    return result

def select(self, k):
    assert( k <= self._full_cartesian_product_size )

    while self._num_elements_popped < k:
        self._pop_next_layer_product()

    # also consider all layer products still in hull
    for val, (i,j), is_max in self._hull_heap:
        if is_max:
            self._num_elements_popped += len(self._loh_a[i]) * len(self._loh_b[j])
            self._layer_products_considered.append( (i,j) )

    # generate: values in layer products
candidates = [val_a+val_b for i,j in self._layer_products_considered for val_a in self._loh_a[i] for val_b in self._loh_b[j]]
print('Ratio\, of\, total\, popped\, candidates\, to\, k: {}'.format(len(candidates) / k))
k_small_vals, large_vals = partition(candidates, k)
return k_small_vals

References


