Convex Hull and Linear Programming in Read-only Setup with Limited Work-space

Minati De$^1$, Subhas C. Nandy$^1$, and Sasanka Roy$^2$

$^1$ Indian Statistical Institute, Kolkata - 700108, India
$^2$ Chennai Mathematical Institute, Chennai - 603103, India

Abstract. Prune-and-search is an important paradigm for solving many important geometric problems. We show that the general prune-and-search technique can be implemented where the objects are given in read-only memory. As examples we consider convex-hull in 2D, and linear programming in 2D and 3D. For the convex-hull problem, designing sub-quadratic algorithm in a read-only setup with sub-linear space is an open problem for a long time. We first propose a simple algorithm for this problem that runs in \( O(n^{3/2+\epsilon}) \) time and \( O(n^{1/2}) \) space. Next, we consider a restricted version of the problem where the points in \( P \) are given in sorted order with respect to their \( x \)-coordinates in a read-only array. For the linear programming problems, the constraints are given in the read-only array. The last three algorithms use prune-and-search, and their time and extra work-space complexities are \( O(n^{1+\epsilon}) \) and \( O(\log n) \) respectively, where \( \epsilon \) is a small constant satisfying \( \sqrt{\frac{\log \log n}{\log n}} < \epsilon < 1 \).

1 Introduction

Designing algorithmic tools with fast and limited size memory (e.g caches) but having capability of very fast processing of a massive high quality data is a challenging field of research [1, 3, 4]. The problem becomes much more difficult if the input data is given in a read-only array, and very small amount of work-space is available in the system. Such a situation arises in the concurrent programming environment where many processes access the same data, and hence modifying the data by a process during the execution is not permissible [2]. In this paper, we show that the general prune-and-search technique can be implemented where the objects are given in read-only array. As examples we consider convex-hull in 2D, and linear programming in 2D and 3D.

Given a set \( P = \{p_1, p_2, \ldots, p_n\} \) of points in 2D, the problem of designing sub-quadratic time algorithm for computing convex hull for \( P \) with sub-linear extra work-space is an important problem and is being studied for a long time. Bronnimann et al. [5] showed that Graham’s scan algorithm for computing convex hull of a planar point set of size \( n \) can be made in-place maintaining \( O(n \log n) \) time complexity. Here, the extra workspace required is \( O(1) \), and the output is
available in the same input array. In the same paper they also showed that (i) the convex hull of a point set in 2D can be computed in an in-place manner in $O(n \log h)$ time and with $O(1)$ extra workspace where $h$ is the number of hull vertices, and (ii) the linear programming in 2D with $n$ constraints can be solved in $O(n)$ time. Very recently, Vahrenhold [11] showed that the prune-and-search algorithm by Kirkpatrick and Seidel [8] for computing the convex hull of a planar point set can also be made in-place maintaining the $O(n \log h)$ time complexity and using only $O(1)$ work-space. All these algorithms permute the input array after the execution. If a planar point set $P$ is given in a read-only array, then the well-known Jarvis March algorithm computes the convex hull in $O(nh)$ time with $O(1)$ extra space. The problem of designing a sub-quadratic algorithm for computing convex hull in a read-only environment with sub-linear work-space is an open problem for a long time. Chan and Chen [6] proposed an algorithm that can compute the convex hull in $O(n(\log n + \frac{n}{s}))$ time using $O(s)$ space where $s \leq n$ is a chosen integer. In the same paper, they proposed an $O(n)$ time randomized algorithm for the linear programming problem in fixed dimension using $O(\log n)$ extra space in a read-only environment. They also considered the problem of computing the convex hull where the points are sorted by their $x$-coordinates. The proposed algorithm is a randomized one and runs in $O(\frac{1}{\delta}n)$ expected time and $O(\frac{1}{\delta}n\delta)$ extra space for any fixed $\delta > 0$. The algorithm can be made deterministic if the running time is increased to $O(2^{O(\frac{1}{\delta})}n)$. The convex hull of a simple polygon with $n$ vertices can be computed in a read-only setup in $O\left(\frac{n \log n}{\log \log \pi}\right)$ time with $O\left(\frac{\log n}{\log \log \pi}\right)$ extra workspace [2].

In this paper, we first address the open problem related to the convex hull problem in 2D. We show that if the points in $P$ are given in a read-only array then the convex hull can be computed in $O(n^{1+\epsilon})$ time and $O(n^{1+\epsilon})$ extra space. Next, we consider a restricted version of the convex hull problem, where the input points are given in sorted order of their $x$-coordinates. Here, we can apply prune-and-search technique to compute the convex hull of $P$ in $O(n^{1+\epsilon})$ time and $O(\log n)$ space, where $\epsilon$ is a constant satisfying $\frac{\log \log n}{\log n} < \epsilon < 1$. We also show that similar technique works for solving the linear programming problem in 2D and 3D in the read-only setup with the same time complexity. In this context, it needs to be mentioned that a similar technique is adopted to solve the minimum enclosing circle problem for a set of points in 2D, where the input points are given in a read-only array [7].

2 Convex hull

2.1 Unrestricted version

Given a set $P$ of $n$ points in 2D in a read-only array, the objective is to report the vertices of the convex hull of $P$. We describe the method of reporting the upper-hull; the lower-hull can be computed in a similar manner. We use three arrays,
Theorem 1. Given a set \( P \) and report only the portion from denoted by \( CH \) first pass. In each stage where \( x \) less than \( x \) lies between properly merged with \( CH \) \( O \) time using \( \text{namely } A, B \) and \( C \), each of size \( O(\sqrt{n}) \) as the work-space. For the notational simplicity, we will use \( P(i) \) and \( P_i \) to denote the set of points whose \( x \)-coordinate lies between \( x((i-1)\sqrt{n}) \) to \( x(i\sqrt{n}) \), and the set of points whose \( x \)-coordinate is less than \( x((i-1)\sqrt{n}) \) respectively, where \( x(k) \) denotes the \( k \)-th smallest element among the \( x \)-coordinates of the points in \( P \). The upper hull of \( P(i) \) and \( P_i \) are denoted by \( CH(i) \) and \( CH_i \) respectively. Our algorithm executes in two passes. Each pass consists of \( \lceil \sqrt{n} \rceil \) stages. In the \( i \)-th stage of the first pass, we pick up the points in \( P(i) \) in the array \( A \). We assume that \( (i-1) \) stages are complete; the vertices of \( CH(i-1) \) in the convex hull \( CH_{i-1} \) are stored in the array \( B \). The \( j \)-th element of the array \( C \) (denoted by \( C[j] \)) contains the first and last hull-vertices \((f_j, \ell_j)\) among the points in \( P(j) \) in the convex hull \( CH_{i-1}, j \leq i - 1 \). If no such hull-vertex exists then \( C[j] \) contains \((-1, -1)\). We execute the following steps in the \( i \)-th stage.

1. Compute \( x((i-1)\sqrt{n}) \) and \( x(i\sqrt{n}) \) among the points in the array \( P \), and identify all the points in \( P(i) \) to store them in the array \( A \).
2. Compute the upper hull \( CH(i) \) of the points in \( A \) using the in-place convex hull algorithm of [5].
3. Merge \( CH(i) \) with \( CH_{i-1} \) as follows: (i) draw the common tangent \( L = [a, b] \) of \( CH(i-1) \) (stored in \( B \)) and \( CH(i) \), where \( a \in CH(i-1) \) and \( b \in CH(i) \). If \( a \) is not the first vertex of \( CH(i-1) \), then update \( C[i] \) by \((f_{i-1}, a)\) and put \([b, \ell_i)\] in \( C[i] \) (\( \ell_i \) is obtained from \( A \)). Otherwise (i.e., if \( a \) is the first vertex of \( CH(i-1) \)) then traverse the array \( C \) to identify a hull-vertex \( u \in CH(i) \) of a preceding block \( j \) \((j < i - 1)\) that is connected with \( a \in CH(i - 1) \).

Note that, if \( j \leq i - 2 \), then all the array elements \( C[k], j + 1 \leq k \leq i - 1 \) will contain \((-1, -1)\). We recompute \( CH(j) \) and draw the common tangent of \( CH(i) \) and \( CH(j) \). The same is followed until we get a tangent of \( CH(i) \) and \( CH(j') \) \((j' < i - 1)\) that does not touch the vertex \( f_j \). We update \( C[j'] \) and set \( C[i] \) with appropriate vertex pair.

In the second pass, we compute \( CH(i) \) for all the blocks whose \( C[i] \neq (-1, -1) \), and report only the portion from \( f_i \) to \( \ell_i \).

**Theorem 1.** Given a set \( P \) of \( n \) points in 2D in a read-only array, the convex hull of \( P \) can be correctly computed in \( O(n^{3+\epsilon}) \) time using \( O(n^{2}) \) extra-space, where \( \sqrt{\frac{\log \log n}{\log n}} < \epsilon < 1 \).

**Proof.** The correctness follows from the fact that in the \( i \)-th stage, \( CH(i) \) is appropriately merged with \( CH_{i-1} \). We now analyze the time complexity of the first pass. In each stage \( i \), \( x((i-1)\sqrt{n}) \) to \( x(i\sqrt{n}) \) can be computed in \( O(n^{1+\epsilon}) \) time using \( O(\sqrt{n}) \) extra space, where \( \sqrt{\frac{\log \log n}{\log n}} < \epsilon < 1 \) (see the algorithm of [10] in Appendix 1). Next, the convex hull \( CH(i) \) in the array \( A \) is computed in \( O(\sqrt{n} \log n) \) time [5]. While merging \( CH(i) \) with \( CH_{i-1} \), we may need to recompute \( CH(j) \) for different \( j \leq i - 1 \). However, the recomputation of the convex hull
of a block implies that there exists a block whose no vertex participate in the convex hull of $P$. Thus the amortized complexity of pass 1 is $O(n^{\frac{3}{2}} + \varepsilon + n \log n)$.

The second pass needs the same amount of time. The space complexity follows from the size of $A$, $B$ and $C$, and the fact that $\frac{\log n}{\log \log n} < n$. 

### 2.2 Restricted version

Given a set $P$ of $n$ points in 2D sorted with respect to their $x$-coordinates in a read-only array, the objective is to report the edges of the convex hull of the points in $P$. We will show how Kirkpatrick and Seidel’s [8] deterministic prune-and-search algorithm for computing convex hull can be implemented in this framework.

The algorithm in [8] computes upper-hull and lower-hull separately and report them. The basic steps of computing upper-hull for a set of points $P$ is given in Algorithm 1. Lower-hull can be computed in a similar way. Algorithm 1 follows divide-and-conquer paradigm. It uses a procedure $\text{Compute-Bridge}$ to compute the bridge between two disjoint subsets of $P$ using the prune-and-search technique. The details of this procedure is described in Algorithm 2.

**Algorithm 1: KS-Upper-Hull($P$)**

**Input:** A set of points $P$ in 2D sorted according to $x$-coordinates  
**Output:** The upper-hull of $P$  
*(Uses divide-and-conquer technique)*

**STEP 1:** Find the point $p_m \in P$ having median $x$-coordinate;  
**STEP 2:** Partition $P$ into two subset $P_l$ and $P_r$ where $P_l$ contains all the points in $P$ whose $x$-coordinate is less than or equal to $x(p_m)$ and $P_r = P \setminus P_l$;  
**STEP 3:** $(a, b) = \text{Compute-Bridge}(P_l, P_r)$; (* This procedure computes the bridge between $P_l$ and $P_r$; $a \in P_l$, $b \in P_r$ *)  
**STEP 4:** Report $(a, b)$;  
**STEP 5:** Compute $P_l = P_l \setminus P'_l$, where $P'_l = \{p \in P_l | x(p) > x(a)\}$, and $P_r = P_r \setminus P'_r$, where $P'_r = \{p \in P_r | x(p) < x(b)\}$;  
**STEP 6:** KS-Upper-Hull($P_l$);  
**STEP 7:** KS-Upper-Hull($P_r$);

The straight-forward implementation of the algorithm KS-Upper-Hull in a read-only memory requires $O(n)$ space for the procedure $\text{Compute-Bridge}$ as it needs to remember which points were pruned in the previous iterations. In addition, the algorithm KS-Upper-Hull($P$) reports the hull-edges in an arbitrary fashion (not in order along the boundary of the convex hull) and takes $O(\log n)$ space for the recursions. So, the main hurdle in read-only model is to (i) report the hull edges in order, and (ii) implement the procedure $\text{Compute-Bridge}$ using only $O(\log n)$ extra-space. In the next subsections, we describe how to resolve these issues. With this we have following main result:
Algorithm 2: Compute-Bridge($P_\ell, P_r$)

Input: Two sets of points in 2D, $P_\ell$ and $P_r$ sorted according to $x$-coordinates
Output: The bridge between $P_\ell$ and $P_r$

* (Uses prune-and-search technique)*

**STEP 1:**

while $|P_\ell| > 1$ and $|P_r| > 1$ do

**STEP 1.1:** Arbitrarily pair-up points in $P_\ell \cup P_r$; Let $L$ be the set of these pairs. Each such pair of points $(p, q) \in L$ will signify a line $pq$ which will pass through $p, q$. We denote the slope of the line $pq$ as $\alpha(pq)$.

**STEP 1.2:** Consider the slopes of these $|P_\ell \cup P_r|$ lines and compute their median. Let $\alpha_m$ be the median slope.

**STEP 1.3:** Compute the supporting line of $P_\ell$ and $P_r$ with slope $\alpha_m$; Suppose these are at points $a(\in P_\ell)$ and $b(\in P_r)$ respectively.

**STEP 1.4:** Now compare $\alpha(ab)$ with $\alpha_m$. Here one of the three cases may arise: (i) $\alpha(ab) = \alpha_m$, (ii) $\alpha(ab) < \alpha_m$ or (iii) $\alpha(ab) > \alpha_m$.

if $\alpha(ab) = \alpha_m$ then

$ab$ is the required bridge between the points in $P_\ell$ and $P_r$. So, the procedure returns $(a, b)$. Otherwise

Decide whether $\alpha(ab) < \alpha_m$ or $\alpha(ab) > \alpha_m$. Without loss of generality, assume that $\alpha(ab) < \alpha_m$, then we will consider all the pairs $(p, q) \in L$ whose $\alpha(pq) \geq \alpha_m$. We can ignore the one point among $(p, q)$ which is to the left of the other one. So, at least $\frac{|P_\ell \cup P_r|}{4}$ points are ignored for further consideration.

**STEP 2:**

Find the bridge in brute-force manner and return the bridge.

Theorem 2. Given a set of $n$ sorted points $P$ of 2D in a read-only array, the convex-hull of $P$ can be computed in $O(n^{1+\epsilon})$ time using $O(\log n)$ extra-space, where $\sqrt{\frac{\log \log n}{\log n}} < \epsilon < 1$.

**Reporting the hull-edges in-order** Now, we will show how to report the hull vertices in clock-wise order using no more than $O(\log n)$ extra-space. Consider the recursion tree $T$ of the algorithm KS-UPPER-HULL. Its each node represents the reporting of a hull-edge. In the algorithm KS-UPPER-HULL, as the reporting is done according to pre-order traversal of the tree $T$, the hull edges are not reported in clock-wise order. In order to report them in clock-wise order, we need to traverse the recursion tree in in-order manner, i.e. STEP 3 and STEP 4 of the Algorithm 1 should be in between STEP 5 and STEP 6. But, if we do this, then we can not evoke KS-UPPER-HULL($P_\ell$) on the updated set $P_r$. To resolve this problem, we will compute the bridge in STEP 3 itself but will not report it then. We push it in the stack in STEP 3 and pop it from stack in between STEP 5 and STEP 6. The size of this stack depends on the depth of the recursion tree which is $O(\log h)$, where $h$ is the number of hull-edges. The details of this change
Algorithm 3: \textsc{Read-Only-Upper-Hull}(\texttt{start}, \texttt{end})

\begin{description}
\item[Input:] A portion of read-only array $P[\texttt{start},\ldots,\texttt{end}]$ containing points in 2D sorted according to the $x$-coordinates.
\item[Output:] The upper-hull for the points in $P[\texttt{start},\ldots,\texttt{end}]$.
\end{description}

\begin{enumerate}
\item[STEP 1:] Let $m = \lceil \frac{\texttt{end}-\texttt{start}}{2} \rceil$; The point $P[m] \in P[\texttt{start},\ldots,\texttt{end}]$ have the median $x$-coordinate;
\item[STEP 2:] (* Now $P_L = P[\texttt{start},\ldots,m]$ and $P_R = P[m+1,\ldots,\texttt{end}]$ *)
\item[STEP 3:] $(i, j) = \textsc{Compute-Bridge}(\texttt{start}, m, \texttt{end})$; (* This procedure returns a pair of indices of points in array $P$ that defines the bridge between $P_L$ & $P_R$ *), Push the edge $(P[i], P[j])$ in STACK;
\item[STEP 4:] (* Modified $P_L = P[\texttt{start},\ldots,i]$ and $P_R = P[j,\ldots,\texttt{end}]$ *)
\item[STEP 5:] \textsc{Read-Only-Upper-Hull}(\texttt{start}, i);
\item[STEP 6:] Report the edge $(P[i], P[j])$ popping the top element from STACK;
\item[STEP 7:] \textsc{Read-Only-Upper-Hull}(j, \texttt{end});
\end{enumerate}

is given as \textsc{Read-Only-Upper-Hull}(\texttt{start}, \texttt{end}) in Algorithm 3. Thus we have the following result:

**Lemma 1.** Given a set of $n$ sorted points $P$ in a read-only array, the reporting of the hull edges can be done in clock-wise order using only $O(\log n)$ extra-space (assuming that the procedure \textsc{Compute-Bridge} takes no more than $O(\log n)$ space).

\textsc{Compute-Bridge} with $O(\log n)$ extra-space Here the input set of points $P$ of this procedure is first partitioned into two parts $P_L$ and $P_R$ by choosing the point $P[m]$ having median $x$-coordinate. Since the array is sorted with respect to the $x$-coordinates, this needs $O(1)$ time. Now an iterative procedure (while-loop) is executed to compute the bridge of $P_L$ and $P_R$. In each iteration of the while-loop of the procedure \textsc{Compute-Bridge}, $\frac{1}{4}$th of the points from the set $P_L \cup P_R$ are pruned. The points which are pruned in $i$th iteration are not considered in any $j$th iteration, where $j > i$. After an iteration of the while-loop, either the bridge is returned or the iteration continues until $|P_L| = 1$ or $|P_R| = 1$. So, the number of iterations of the while loop is $O(\log n)$, where $n$ is the total number of points in $P$.

While executing the $i$th iteration, we want to remember the points which were pruned in the previous $i - 1$ iterations, $i \in \{1, \ldots \log n\}$. If we use mark-bits to remember which points are valid/invalid, then we need $O(n)$ bits. But, we have only $O(\log n)$ work-space to be used. So, we take an array $M$ of size $O(\log n)$ and another bit-array $B$ of size $O(\log n)$. At each $i$th iteration, ignoring all the pruned points, we pair the valid points and consider the slopes of all the lines defined by the paired points. We compute the median slope value $\mu_i$ of these lines and store it at $M[i]$. If the supporting lines are at points $a$ and $b$, we set
\(B[i]\) as 1 or 0 depending on whether the slope \(\alpha(ab)\) is greater than or less than \(\mu_i\) (since \(\alpha(ab) = \mu_i\) implies that we already get the bridge). Thus, \(B[i]\) signifies whether \(M[i]\) is greater than or less than the slope \(\alpha^*\) of the desired bridge.

Now, we will describe a pairing scheme which will satisfy the following invariants:

**Invariant 1**

(i) If a point \(p\) is pruned at some iteration \(i\), then it will not participate to form a pair for any \(j\)-th iteration, where \(j > i\).

(ii) If \((p, q)\) are paired at the \(i\)-th iteration of the while-loop, and none of the points \(p, q\) is pruned at the end of this iteration and we need to go for \(i + 1\)-th iteration, then \((p, q)\) will again form a valid pair at \(i + 1\)-th iteration.

(iii) If \((p, q)\) is a valid pair at \(i\)-th iteration and \((p, s)(s \neq q)\) are paired at \(i + 1\)-th iteration of the while-loop, then there exist some \(r\) such that \((r, s)\) were paired at \(i\)-th iteration, and \(q\) and \(r\) were pruned at the end of \(i\)-th iteration.

The iteration starts with the points \(\{P[start], P[start + 1], \ldots, P[end]\}\). In the first iteration of the while-loop, we consider the consecutive points, i.e, \((P[start], P[start + 1]), (P[start + 2], P[start + 3]), \ldots\) as valid pairs.

Assume that first \(i - 1\) iterations of the while-loop are over, and we are at the beginning of the \(i\)-th iteration; \(M[t]\) contains median slope of \(t\)-th iteration and \(B[t]\) contains 0 or 1 depending on \(M[t] > \alpha^*\) or \(M[t] < \alpha^*\) for all \(1 \leq t \leq i - 1\). Now, we want to detect all the valid points and pair them up maintaining the Invariant 1. We use another array \(IndexP\) of size \(O(\log n)\) whose all elements are set to -1 at the beginning of this iteration.

We consider the point-pairs \((P[start+2\nu], P[start+2\nu+1]), \nu = 0, 1, \ldots, \lfloor (end-start+1)/2 \rfloor\) in order. For each pair, we compute the slope \(\gamma = \alpha(P[start+2\nu], P[start+2\nu+1])\) of the corresponding line, and perform the level 1 test using \(M[1]\) and \(B[1]\) to see whether both of them remain valid at iteration 1. If the test succeeds, we perform level 2 test for \(\gamma\) by using \(M[2]\) and \(B[2]\). We proceed similarly until (i) we reach up to \(i - 1\)-th level and both the points remain valid at all the levels, or (ii) one of these points becomes invalid at some level, say \(j\) (\(< i - 1\)). In Case (i), the point pair \((P[start+2\nu], P[start+2\nu+1])\) will form a valid pair and participates in computing the median value \(m_i\). In case (ii), suppose \(P[start+2\nu]\) remains valid and \(P[start+2\nu+1]\) becomes invalid. Here two situations need to be considered depending on the value of \(IndexP[j]\). If \(IndexP[j] = -1\) (no point is stored in \(IndexP[j]\)), we store \(start + 2\nu\) or \(start + 2\nu + 1\) in \(IndexP[j]\) depending on whether \(P[start+2\nu]\) or \(P[start+2\nu+1]\) remains valid at level \(j\). If \(IndexP[j] = \beta(\neq -1)\) (index of a valid point), we form a pair \((P[start+2\nu], P[\beta])\) and proceed to check starting from \(j + 1\)-th level (i.e., using \(M[j + 1]\) and \(B[j + 1]\)) onwards until it reaches the \(i\)-th level or one of them is marked invalid in some level between \(j\) and \(i\). Both the situations are handled in a manner similar to Cases (i) and (ii) as stated above. Thus, each valid point in the \(i\)-th iteration has to qualify as a valid point in the tests of all the \(i - 1\)
levels. For any other point the number of tests is at most \(i - 2\). This leads to the following result:

**Lemma 2.** In the \(i\)-th iteration, the amortized time complexity of finding all valid pairs is \(O(ni)\).

The main task in the \(i\)-th iteration is to find the median of the slope of lines corresponding to valid pair of points. We essentially use the median finding algorithm of Munro and Raman [10] for this purpose (see Appendix 1). Notice that, in order to get each slope, we need to get a valid pair of points, which takes \(O(i)\) time (see Lemma 2). The time required for finding the lowest slope is \(O(ni)\). Similarly, computing the second lowest needs another \(O(ni)\) time. Proceeding similarly, the time complexity of the procedure \(A_0\) of [10] is \(O(n^2i^2)\) (see the Appendix). Similarly, \(A_1\) takes \(O(i^2n^{1+\sqrt{i}}\log n)\) time, and so on. Finally, \(A_k\) takes \(O(i^2n^{(1+\sqrt{i})} \log^k n)\) time. Choosing \(k = \sqrt{\log n \log \log n} < \log n\), we need \(O(\log n)\) space in total. Thus, we have the following result:

**Lemma 3.** The time complexity of the \(i\)-th iteration of the while-loop of the procedure **Compute-Bridge** is \(O(i^2n^{(1+\sqrt{i})} \log^k n)\), where \(1 \leq k \leq \sqrt{\log n \log \log n}\). The extra space required is \(O(\log n)\).

At the end of \(O(\log n)\) iterations, we could discard all the points except at most \(|\text{IndexP}| + 3\) points, where \(|\text{IndexP}|\) is the number of cells in the array \(\text{IndexP}\) that contain valid indices of \(P (\neq -1)\). This can be at most \(O(\log n)\) in number. We can further prune the points in the \(\text{IndexP}\) array using the in-place algorithm for **Compute-Bridge** described in [11]. Thus, we have the following result:

**Lemma 4.** The read-only version of the procedure **Compute-Bridge** is correct and the time complexity is \(O(n^{(1+\sqrt{i})} \log^{k+3} n)\), where \(1 \leq k \leq \sqrt{\log n \log \log n}\). Apart from the input array, it requires \(O(\log n)\) extra space.

**Proof.** The correctness of this read-only version of the procedure **Compute-Bridge** follows from the fact that the Invariant 1 is correctly maintained.

By Lemma 3, the time complexity of the \(i\)-th iteration is \(O(i^2n^{(1+\sqrt{i})} \log^k n)\), where \(i = 1, 2, \ldots, \log n\). Thus, the total time complexity of all the \(O(\log n)\) iterations is \(O(n^{(1+\sqrt{i})} \log^{k+3} n)\). The time required by the in-place algorithm for considering all the entries in the array \(\text{IndexP}\) is \(O(\log n)\).

The space complexity obviously follows since the same set of arrays \(M, B, \text{IndexP}\) and the stack for finding the median can be used for all the iterations, and each one is of size at most \(O(\log n)\). \(\square\)
Correctness and Complexity - Proof of Theorem 2  

The correctness of the algorithm Read-Only-Upper-Hull follows from the correctness of Kirkpatrick and Siedel’s algorithm [8], as we are following the basic structure of this. The procedure Compute-Bridge is evoked \(h\) times, where \(h\) is the number of hull-edges. Consider the recursion tree of the algorithm Read-Only-Upper-Hull. Note that the depth of this tree is \(O(\log n)\) (more specifically, \(\log h\)), and total time complexity of a single level is \(O(n^{1+\frac{1}{k+1}} \log k^3 n)\), where \(1 \leq k \leq \sqrt{\frac{\log n}{\log \log n}}\) (see Lemma 4). As there are at most \(\log n\) levels, so the total time complexity of the algorithm Read-Only-Upper-Hull is \(O(n^{1+\epsilon})\), where \(\epsilon\) satisfies \(\frac{\log \log n}{\log n} < \epsilon < 1\).

For the recursion of Read-Only-Upper-Hull we need \(\log n\) space and for each node of the recursion tree we need another \(O(\log n)\) space for the procedure Compute-Bridge. However, we can re-use the same space for each of the nodes for computing the bridge. Hence the total space complexity of the algorithm Read-Only-Upper-Hull is \(O(\log n)\).

3 2D Linear Programming

In this section, we consider the problem of solving 2D linear programming in a read-only setup, i.e., the constraints are given in a memory where swapping of elements or modifying any entry is not permissible. Megiddo proposed a linear time prune-and-search algorithm for this problem which takes \(O(n)\) space [9]. We will show that Megiddo’s algorithm for 2D linear programming can be implemented when the constraints are stored in a read-only memory using \(O(\log n)\) extra-space and the running time would be \(O(n^{1+\epsilon})\), where \(\epsilon\) satisfies \(\sqrt{\frac{\log n}{\log \log n}} < \epsilon < 1\).

3.1 Overview of Megiddo’s 2D Linear Programming

The 2D linear programming problem is as follows:

\[
\min_{x_1, x_2} c_1 x_1 + c_2 x_2 \\
\text{subject to:} \\
a_i' x_1 + b_i' x_2 \geq \beta_i, \ i \in I = \{1, 2, \ldots n\}.
\]

For ease of designing a linear time algorithm, Megiddo transformed it to an equivalent form, stated below:

\[
\min_{x, y} y \\
\text{subject to:} \\
y \geq a_i x + b_i, \ i \in I_1, \\
y \leq a_i x + b_i, \ i \in I_2, \\
|I_1| + |I_2| \leq n.
\]

Megiddo’s 2D linear programming algorithm uses prune-and-search technique. It maintains an interval \([a, b]\) of feasible values of \(x\) (i.e., \(a \leq x \leq b\)). At the
Algorithm 4: MEGIDDO’S-2D-LP($I, c_1, c_2$)

**Input:** A set of $n$ constraints $a_i'x_1 + b_i'x_2 \geq \beta_i$, for $i \in I = \{1, 2, \ldots, n\}$. 
**Output:** The value of $x_1, x_2$ which minimizes $c_1x_1 + c_2x_2$
*(Uses prune-and-search technique)*

**STEP 1:** Convert the form into the following: $\min_{x,y} \text{subject to}\ (i) \ y \geq a_i x + b_i, i \in I_1, (ii) \ y \leq a_i x + b_i, i \in I_2$, where $|I_1| + |I_2| \leq n$.

**STEP 2:** Set $a = -\infty$ and $b = \infty$;
while $|I_1 \cup I_2| > 4$ do
  **STEP 2.1:** Arbitrarily pair-up the constraints of $I_1$ (resp. $I_2$). Let $M_1$ (resp. $M_2$) be the set of aforesaid pairs, where $|M_1| = \lfloor \frac{|I_1|}{2} \rfloor$ and $|M_2| = \lfloor \frac{|I_2|}{2} \rfloor$.
  Each pair of constraints in $M_1 \cup M_2$ are denoted by $(i,j)$ where $i$ and $j$ indicate the $i$-th and $j$-th constraints.
  **STEP 2.2:** for each pair $(i,j) \in M_1 \cup M_2$
do if $a_i \neq a_j$ then Compute $x_{ij} = \frac{b_i - b_j}{a_i - a_j}$; Find the median $x_m$ among all $x_{ij}$’s which are in the interval $[a,b]$ ;
  **STEP 2.3:** Test whether optimum $x^*$ satisfies $x^* = x_m$ or $x^* > x_m$ or $x^* < x_m$ as follows:
  **STEP 2.3.1:** Compute $g = \max_{i \in I_1} a_i x_m + b_i; \ h = \min_{i \in I_2} a_i x_m + b_i$;
  **STEP 2.3.2:** Compute $s_g = \min_{i \in I_1} a_i x_m + b_i = g; \ S_g = \max_{i \in I_1} a_i x_m + b_i = g$; $s_h = \min_{i \in I_2} a_i x_m + b_i = h; \ S_h = \max_{i \in I_2} a_i x_m + b_i = h$;
  **STEP 2.3.2:** (*$g \leq h \Rightarrow x_m$ is feasible ; $g > h \Rightarrow x_m$ in infeasible region *)
  if $g > h$ then
    if $s_g > S_h$ then (* $x_m < x^*$ *) $b = x_m$
    if $S_g < s_h$ then (* $x_m > x^*$ *) $a = x_m$
    else Report there is no feasible solution of the LP problem; Exit
  else
    if $s_g > 0$ & $s_g \geq S_h$ then (* $x_m < x^*$ *) $b = x_m$
    if $S_g < 0$ & $S_g \leq s_h$ then (* $x_m > x^*$ *) $a = x_m$
    else Report optimum solution $x_1 = x_m \ & \ x_2 = \frac{a - b}{c_2}$; Exit
  **STEP 2.4:** *(Pruning step - The case where iteration continues.*
  Without loss of generality Assume that $x^* > x_m$; *)
  for each pair $(i,j) \in M_1 \cup M_2$
do if $a_i = a_j$ then Ignore one of the two constraints;
  if $a_i \neq a_j$ and $x_{ij} < x_m$ then Ignore one of the two constraints;
  **STEP 3:** *(The case when $|I_1 \cup I_2| \leq 4$)*

The problem can be solved directly.
beginning of the algorithm, \( a = -\infty \) and \( b = \infty \). After each iteration of the algorithm, either it finds out that at some \( x = x_m \) \((a \leq x_m \leq b)\) the optimal solution exist (so the algorithm stops) or the interval \([a, b]\) is redefined (the new interval is either \([a, x_m]\) or \([x_m, b]\)) and at least \(\frac{n}{4}\) constraints are pruned for the next iteration. The detail steps of the algorithm is given in the algorithm MEGIDDO’S-2D-LP(\(I, c_1, c_2\)).

Megiddo’s 2D linear programming algorithm needs \(O(n)\) time and \(O(n)\) space. In the next subsection we will show how to tailor this algorithm to work in the read-only setup such that it does not take more than \(O(\log n)\) space and running time is \(O(n^{1+\epsilon})\), where \(\sqrt{\log \log n / \log n} < \epsilon < 1\).

### 3.2 2D-Linear Programming in Read-only setup

We will give step by step description of implementing MEGIDDO’S-2D-LP in a read-only setup. The straight-forward conversion of one form into another mentioned in STEP 1 would take \(O(n)\) extra-space. Note that remembering only the objective function \(y = c_1x_1 + c_2x_2\), will enable one to reformulate the newer version of the constraints on-the-fly substituting \(x_2\) in terms of \(x_1\) and \(y\) (replacing \(x_1\) by \(x\)) in each constraint. So, we need not to worry about storing this new form. It is also to be noted that 2D linear programming can be implemented in an in-place model in \(O(n)\) time using \(O(1)\) extra-space \([5]\). So, we can implement the pruning activities in Step 2 in read-only environment in a manner similar to COMPUTE-BRIDGE as described in section 2.2 using \(O(\log n)\) space and \(O(n^{1+\epsilon})\) time, where \(\epsilon\) satisfies \(\sqrt{\log \log n / \log n} < \epsilon < 1\). Step 3 can obviously be implemented when the constraints are given in a read-only memory. Hence, we have the following result:

**Theorem 3.** 2D linear programming can be implemented in a read-only model in \(O(n^{1+\epsilon})\) time using \(O(\log n)\) extra-space, where \(\sqrt{\log \log n / \log n} < \epsilon < 1\).

### 4 3D Linear Programming

In the same paper \([9]\) Megiddo proposed a linear time algorithm for 3D linear programming. The problem is stated as follows:

\[
\begin{align*}
\min_{x_1, x_2, x_3} & \quad d_1x_1 + d_2x_2 + d_3x_3 \\
\text{subject to:} & \quad a'_i x_1 + b'_i x_2 + c'_i x_3 \geq \beta_i, \quad i \in I = \{1, 2, \ldots, n\}.
\end{align*}
\]

As earlier, Megiddo transformed the problem into the following equivalent form:

\[
\begin{align*}
\min_{x, y, z} & \quad z \\
\text{subject to:} & \quad z \geq a_i x + b_i y + c_i, \quad i \in I_1, \\
& \quad z \leq a_i x + b_i y + c_i, \quad i \in I_2, \\
& \quad 0 \geq a_i x + b_i y + c_i, \quad i \in I_3, \\
& |I_1| + |I_2| + |I_3| \leq n.
\end{align*}
\]
The Megiddo’s-3D-LP algorithm also follows prune-and-search paradigm. It pairs-up constraints \((C^i_k, C^j_k)\) where \(C^i_k, C^j_k\) are from same set \(I_k, k \in \{1, 2, 3\}\); So, there are at most \(n^2\) pairs. Let \(C^i_k\) (resp. \(C^j_k\)) corresponds to \(a_i x + b_i y + c_i\) (resp. \(a_j x + b_j y + c_j\)). If \((a_i, b_i) = (a_j, b_j)\), then we can easily ignore one of the constraints. Otherwise (i.e., if \((a_i, b_i) \neq (a_j, b_j)\)), then each pair signifies a line \(L_{ij}: a_i x + b_i y + c_i = a_j x + b_j y + c_j\) which divides the plane into two halves.

Let \(L\) be the set of lines obtained in this way. We compute the median \(\mu\) of the gradients of the members in \(L\). Next, we pair-up the members in \(L\) such that one of them have gradient less than \(\mu\) and the other one have gradient greater than \(\mu\). Let \(\Pi\) be the set of these paired lines. Each of these pairs will intersect. We compute the intersection point \(a\) having median \(y_m\) among the \(y\)-coordinates of these intersection points. Next, we execute the procedure Testing-Line as stated below with respect to the line \(L_H: y = \mu x + y_m \sqrt{\mu^2 + 1}\) (having gradient \(\mu\) and passing through \(a\)). This determines in which side of \(L_H\) the optimum solution lies. Next, we identify the pairs in \(\Pi\) which intersect on the other side of the optimum solution. Among these pairs, we compute the intersection point \(b\) having median \(x\)-coordinates of their intersections, and execute Testing-Line with the line \(L_V\) having gradient \(\frac{1}{\mu}\) and passing through \(b\). \(L_H\) and \(L_V\) determines a quadrant \(Q\) containing the optimum solution. Now consider the paired lines in \(\Pi\) that intersect in the quadrant \(Q'\), diagonally opposite to \(Q\). Let \((L_{ij}, L_{k\ell})\) be a paired line of \(\Pi\) that intersect in \(Q'\). For at least one of the lines \(L_{ij}\) and \(L_{k\ell}\), it is possible to correctly identify the side containing the optimum solution without executing Testing-Line (see [9]). Thus, for each of such lines we can prune one constraint. As a result, after each iteration it can prune at-least \(\frac{n}{16}\) constraints for next iteration or report optimum.

The procedure Testing-Line takes a straight line \(L\) in the \(x-y\) plane and tests whether the solution on the line \(L\) (i) does not exist, or (ii) unbounded, or (iii) unique optimum solution exists using a 2D linear programming. In Case (ii), the solution of the given 3D linear programming problem is unbounded. In Case (i) and (iii), we need to decide in which side of \(L\) the optimum solution of the given linear programming problem lies by executing two other 2D linear programming. Both of them can be executed if the constraints are given in read-only memory.

The detail description of the algorithm Megiddo’s-3D-LP is given in Appendix 2. As Megiddo’s-3D-LP is a prune-and-search algorithm, one can easily show that this can be implemented in a read-only setup. Thus, we have the following result:

**Theorem 4.** 3D linear programming can be implemented in a read-only model in \(O(n^{1+\epsilon})\) time using \(O(\log n)\) extra-space, where \(\sqrt{\frac{\log \log n}{\log n}} < \epsilon < 1\).
References

1. T. Asano, W. Mulzer, G. Rote, and Y. Wang. Constant-work-space algorithms for geometric problems. JoCG, 2(1):46–68, 2011.
2. L. Barba, M. Korman, S. Langerman, K. Sadakane, and R. I. Silveira. Space-time trade-offs for stack-based algorithms. CoRR, abs/1208.3663, 2012.
3. H. Blunck and J. Vahrenhold. In-place algorithms for computing (layers of) maxima. Algorithmica, 57(1):1–21, 2010.
4. P. Bose, A. Maheshwari, P. Morin, J. Morrison, M. H. M. Smid, and J. Vahrenhold. Space-efficient geometric divide-and-conquer algorithms. Comput. Geom., 37(3):209–227, 2007.
5. H. Brönnimann, J. Iacono, J. Katajainen, P. Morin, J. Morrison, and G. T. Toussaint. In-place planar convex hull algorithms. In LATIN, pages 494–507, 2002.
6. T. M. Chan and E. Y. Chen. Multi-pass geometric algorithms. Discrete & Computational Geometry, 37(1):79–102, 2007.
7. M. De, S. C. Nandy, and S. Roy. Minimum enclosing circle with few extra variables. In FSTTCS, 2012.
8. D. G. Kirkpatrick and R. Seidel. The ultimate planar convex hull algorithm? SIAM J. Comput., 15(1):287–299, 1986.
9. N. Megiddo. Linear-time algorithms for linear programming in $\mathbb{R}^3$ and related problems. SIAM J. Comput., 12(4):759–776, 1983.
10. J. I. Munro and V. Raman. Selection from read-only memory and sorting with minimum data movement. Theor. Comput. Sci., 165(2):311–323, 1996.
11. J. Vahrenhold. On the space efficiency of the “ultimate planar convex hull algorithm”. In CCCG, pages 131–136, 2012.
Appendix 1

Munro and Raman’s median finding algorithm: Given a set of \( n \) real numbers in a read-only array \( P \), the algorithm in [10] for finding their median is designed by using a set of procedures \( A_0, A_1, A_2, \ldots, A_k \), where procedure \( A_i \) finds the median by evoking the procedure \( A_{i-1} \) for \( i \in \{1, 2, \ldots, k\} \). The procedures \( A_0, A_1, A_2, \ldots, A_k \) are stated below.

Procedure \( A_0 \): In the first iteration, after checking all the elements in \( P \), it finds the largest element \( p_{(1)} \) in linear time. In the second iteration it finds the second largest \( p_{(2)} \) by checking only the elements which are less than \( p_{(1)} \). Proceeding in this way, in the \( j \)-th iteration it finds the \( j \)-th largest element \( p_{(j)} \) considering all the elements in \( P \) that are less than \( p_{(j-1)} \). In order to get the median we need to proceed up to \( j = \lceil \frac{n}{2} \rceil \). Thus, this simple median finding algorithm takes \( O(n^2) \) time and \( O(1) \) extra-space.

Procedure \( A_1 \): It divides the array \( P \) into blocks of size \( \sqrt{n} \) and in each block it finds the median using Procedure \( A_0 \). After computing the median \( m \) of a block, it counts the number of elements in \( P \) that are smaller than \( m \), denoted by \( \rho(m) \), by checking all the elements in the array \( P \). It maintains two best block medians \( m_1 \) and \( m_2 \), where \( \rho(m_1) = \max\{\rho(m) | \rho(m) \leq \frac{n}{2}\} \), and \( \rho(m_2) = \min\{\rho(m) | \rho(m) \geq \frac{n}{2}\} \). Thus, this iteration needs \( O(n\sqrt{n}) \) time.

After this iteration, all the elements \( P[i] \) satisfying \( P[i] < m_1 \) or \( P[i] > m_2 \) are marked as invalid. This does not need any mark bit; only one needs to remember \( m_1 \) and \( m_2 \). In the next iteration we again consider same set of blocks, and compute the median ignoring the invalid elements.

Since, in each iteration \( \frac{1}{3} \) fraction of the existing valid elements are marked invalid, we need at most \( O(\log n) \) iterations to find the median \( \mu \). Thus the time complexity of this procedure is \( O(n\sqrt{n}\log n) \).

Procedure \( A_2 \): It divides the whole array into \( n^{1/3} \) blocks each of size \( n^{2/3} \), and computes the block median using the procedure \( A_1 \). Thus, the overall time complexity of this procedure for computing the median is \( n^{1+\frac{1}{3}} \log^{2} n \).

Proceeding in this way, the time complexity of the procedure \( A_k \) will be \( O(n^{(1+\epsilon)k}\log^{k} n) \). As it needs a stack of depth \( k \) for the recursive evoking of \( A_{k-1}, A_{k-2}, \ldots, A_0 \), the space complexity of this algorithm is \( O(k) \).

Setting \( \epsilon = \frac{1}{k+1} \), gives the running time as \( O(n^{(1+\frac{k+1}{k+1})}\log^{k} n) \). If we choose \( n' = \log^{\frac{1}{k+1}} n \), then \( \epsilon \) will be \( \sqrt{\frac{\log \log n}{\log n}} \), and this will give the running time \( O(n^{1+2\epsilon}) \), which is of \( O(n^{1+2\epsilon}) \). So, the general result is as follows:

Result 1 For a set of \( n \) points in \( \mathbb{R} \) given in a read-only memory, the median can be found in \( O(n^{1+\epsilon}) \) time with \( O(\frac{1}{\epsilon}) \) extra-space, where \( 2\sqrt{\frac{\log \log n}{\log n}} < \epsilon < 1 \).
Algorithm 5: Appendix 2 - MEGIDDÔ’S-3D-LP($I, c_1, c_2$)

Input: A set of $n$ constraints $a_i x + b_i y + c_i \geq \beta_i$, for $i \in I = \{1, 2, \ldots, n\}$.
Output: The value of $x_1, x_2$ which minimizes $c_1 x_1 + c_2 x_2$

(* Uses prune-and-search technique *)

**STEP 1:** Convert the form into following: $\min_{x,y,z} x, z$, subject to (i)

$y \geq a_i x + b_i y + c_i$, $i \in I_1$; (ii) $y \leq a_i x + b_i y + c_i$, $i \in I_2$; (iii) $0 \geq a_i x + b_i y + c_i$, $i \in I_3$, where $|I_1| + |I_2| + |I_3| \leq n$.

**STEP 2:**

while $|I_1 \cup I_2 \cup I_3| \geq 16$ do

**STEP 2.1:** Arbitrarily pair-up the constraints $a_i x + b_i y + c_i$, $a_j x + b_j y + c_j$ where $i, j$ are from same set $I_k, k \in \{1, 2, 3\}$. Let $L_1$, $L_2$ and $L_3$ be the set of aforesaid pairs and $L = L_1 \cup L_2 \cup L_3$.

**STEP 2.2:** Let $L_C = \{(i,j) \in L | (a_i, b_i) \neq (a_j, b_j)\}$ and $L_P = \{(i,j) \in L | (a_i, b_i) = (a_j, b_j)\}$

Compute the median $\mu$ of the slopes $\alpha(L_{ij})$ of all the straight lines $L_{ij}$: $a_i x + b_i y + c_i = a_j x + b_j y + c_j$, $(i,j) \in L_C$.

**STEP 2.3:**

Arbitrarily pair up $(L_{ij}, L'_{i'j'})$ where $\alpha(L_{ij}) \leq \mu \leq \alpha(L'_{i'j'})$ and $(i,j), (i',j') \in L_C$.

Let $M_P = \{(L_i, L_j) \in M | \alpha(L_i) = \alpha(L_j) = \mu\}$ (* parallel line-pairs *) and $M_I = \{(L_i, L_j) \in M | \alpha(L_i) \neq \alpha(L_j)\}$ (* intersecting line-pairs *).

for each pair $(L_i, L_j) \in M_P$ do

compute $y_j = \frac{d_i + d_j}{2}$, where $d_i$ = distance of $L_i$ from the line $y = \mu x$ for each pair $(L_i, L_j) \in M_I$ do

Let $a_{ij}$ = point of intersection of $L_i \& L_j$, and $b_{ij}$ = projection of $a_{ij}$ on $y = \mu x$. Compute $y_{ij}$ = signed distance of the pair of points $(a_{ij}, b_{ij})$, and $x_{ij}$ = signed distance of $b_{ij}$ from the origin.

Next, compute the median $y_m$ of the $y_{ij}$ values corresponding to all the pairs in $M$;

**STEP 2.4:** Consider the line $L_H : y = \mu x + y_m \sqrt{\mu^2} + 1$, which is parallel to $y = \mu x$ and at a distance $y_m$ from $y = \mu x$.

Test on which half-plane defined by the line $L_H$ contains the optimum by evoking Testing-Line($L_H$)

**STEP 2.5:** Let $M'_I = \{(L_i, L_j) \in M_I | a_{ij} \& \pi \text{ lie in different sides of } L_H\}$

Compute the median $x_m$ of $x_{ij}$ values for the line-pairs in $M'_I$. Define a line $L_V$ perpendicular to $y = \mu x$ and passing through a point on $y = \mu x$ at a distance $x_m$ from the origin;

Execute the procedure Testing-Line($L_V$) and decide in which side of $L_V$ the optimum lies;

We consider $L_H$ and $L_V$ as horizontal and vertical lines respectively;

W.L.O.G., assume that optimum lies in the top-left quadrant;

**STEP 2.6:** (* Pruning step *)

for all the members $(L_i, L_j) \in M_I$ whose points of intersection $(a_{ij})$ lie in the bottom-right quadrant do

Discard one of the four constraints defined by the pair of lines $L_i, L_j$

for all the members $(L_i, L_j) \in M_P$ whose $y_{ij} \leq y_m$ do

Discard one of the four constraints defined by the pair of lines $L_i, L_j$

**STEP 3:** (*The case when $|I_1 \cup I_2 \cup I_3| \leq 16$*)

The problem can be solved directly by brute-force manner.