# Artemis

**SOLUTION**

Observation: Let f(x, y) be the number of trees below and to the left of (x, y). Then the number of the trees in the rectangle bounded by $t_1$ and $t_2$ is

$$f(t_1.x, t_1.y) + f(t_2.x, t_2.y) - f(t_1.x, t_2.y) - f(t_1.y, t_2.x) + 1$$

if $t_1$ lies below and to the left of $t_2$ (or vice versa), and a similar formula if not.

1. Trivial algorithm. Loop over all rectangles, and loop over all trees to count those inside the rectangle.

   $O(n^3)$

2. Use dynamic programming to compute $f(t_1.x, t_2.y)$ for every $t_1$, $t_2$. Then evaluate all rectangles using the formulae.

   $O(n^2)$, but also $O(n^2)$ memory

3. Place an outer loop t over the trees, representing one corner of a potential rectangle. To evaluate rectangles with corners at t, one only needs f(t.x, *) and f(*, t.y). These can be computed with DP as in algorithm (2), and requires only linear memory.

   $O(n^2)$

4. Sort the trees from left to right, and then process them in that order. As each new tree (say $t_n$) is added, it is inserted into a list of current trees that is sorted vertically. From this information one can calculate $f(t.x, t_n.y)$ and $f(t_n.x, t.y)$ for every t to the left of $t_n$, in linear time. Then one can evaluate all rectangles with one corner at $t_n$. This ends up being very similar to algorithm (3).

   $O(n^2)$

5. Algorithm (1), but with optimised counting. As a pre-process, associate a bitfield with each tree representing which trees lie below and to the right, and a similar bitfield for trees below and to the left. The trees inside a given rectangle may be found as the binary AND of two bitfields. A fast counting mechanism (such as a 16-bit lookup table) will accelerate counting.

   $O(n^3)$ and $O(n^2)$ memory, but with low constant factors

Hermis

An $O(n^2)$ algorithm: Let $(x_0,y_0),\ldots,(x_n,y_n)$ be the points where $(x_0,y_0)=(0,0)$ is the starting point. We will compute A[i,j] and B[i,j] where A[i,j] is the cost to align with the i first points and end up at $(x_i,y_j)$ and B[i,j] is the cost to align with the i first points and end up at $(x_j,y_i)$. We have

$A[i+1,j]=\min \{ A[i,j]+d[x_i,x_{\{i+1\}}], B[i,i+1]+d[y_i,y_j] \}$
$B[i+1,j]=\min \{ B[i,j]+d[y_i,y_{\{i+1\}}], A[i,i+1]+d[x_i,x_j] \}$

The final answer is $\min_j \{A[n,j],B[j,n]\}$

Time $O(n^2)$

# IOI'04: Solution of *Polygon*

Ioannis Emiris and Elias Tsigaridas

September 9, 2004

## 1 Minkowski decomposition

### 1.1 The setting

Although computing the Minkowski sum is straightforward ([2]), deciding whether a polygon is the Minkowski sum of 2 polygons is NP-complete.

There is a pseudo-polynomial time algorithm for the latter ([1]), which we now sketch. If the input is the sequence of $N$ edges, then the input size is $O\left(N(\log m + \log E)\right)$, where $m$ is the maximum number of integer points on any edge, and $E$ is the maximum absolute value of any coordinate defining a primitive edge vector, as defined below. The algorithm in [1] runs in $O(tNm)$ time, where $t = |P \cap \mathbb{Z}^2|$.

### 1.2 Solution of the IOI-04 problem *Polygon*

We call the input polygon $P$. Suppose that it consists of $N$ vertices with non-negative coordinates of the form $v_i = (x_i, y_i), 0 \leq i \leq N-1$, given in counter-clockwise order.

**Definition 1.1** *We call a vector $v = (a, b)$* primitive, *if $\gcd(a, b) = 1$, where $a, b \in \mathbb{Z}_{\geq 0}$. Equivalently, $v$ is primitive iff there are no integer points in its interior.*

The edges of $P$ are represented by the vectors $E_i = v_i - v_{i-1} = (a_i, b_i), 1 \leq i \leq N$, where $a_i, b_i \in \mathbb{Z}$ and the indices are taken modulo $N$. Given an edge $(a_i, b_i)$, if $n_i = \gcd(a_i, b_i)$, then we consider the corresponding primitive edge $e_i = (\frac{a_i}{n_i}, \frac{b_i}{n_i})$. We call the sequence of vectors $\{E_i\}_{1 \leq i \leq m} = \{n_i e_i\}_{1 \leq i \leq m}$ edge sequence of the polygon, where $e_i$ is a primitive vector.

There is a crucial observations in computing the summands $A, B$ such that $P = A + B$.

**Lemma 1.2** *Every primitive edge of $P$ must appear as an edge either of $A$ or of $B$. If the edge is not primitive, there is a third possibility: that it is the Minkowski sum of an edge of $A$ and an edge of $B$, where these edges are parallel.*

Now we can deduce the following fact: A polygon is a summand of $P$ iff its edge sequence is of the form $\{k_j e_j\}_{j \in J}$, where $J \subseteq \{1, \ldots N\}$, $0 \leq k_j \leq n_j,\ k_j \in \mathbb{Z}$ and $\sum_{j \in J} k_j e_j = (0, 0)$, i.e. the sum of the vectors that represent its edges is zero.

Below we present algorithms to find summands as required by the problem *Polygon*. In computing summand polygons, the chosen algorithm may produce a summand with one or more negative coordinates. In this case, we must shift both summands to non-negative coordinates. This is a valid operation, because shifting does not change any polygon.

## 1.3 Looking for a segment summand

The polygon has a segment summand iff at least one pair of the (sub)vectors of its edge sequence has a zero vector sum. Based on this fact, we present 3 algorithms of increasing speed.

Our algorithms start with finding the primitive vectors for every edge, by performing a GCD computation (at section 1.6 there are two algorithms for the GCD computation). For polygons that contain only primitive edges, this step is unnecessary.

**Naive** First we compute the edges of the polygon. Every edge is of the form

$$E_i = (x_{i+1} - x_i, y_{i+1} - y_i) = (a_i, b_i)$$

where the indices are taken modulo $N$. Next we compute all the gcd's of form $\gcd(a_i, b_i)$ and we form the edge sequence using the primitive vectors as in the preamble. This computation can be done in $O(mN)$ time, where $m = \max n_i = \max \gcd(a_i, b_i)$.

For every vector in the sequence we compute its sum with every other vector in the sequence. This can be done in $O(m^2N^2)$ total time.

**Clever** We compute the edge sequence as before in $O(mN)$ time. We sort the sequence with respect to their $x-$coordinate, in $O(mN \lg(mN))$ time.

For every vector in the sequence we test if another vector in the sequence with the opposite $x-$coordinate add to zero. We perform the searching by binary search, since the sequence is sorted, in $O(\lg(mN))$ time. The total time is $O(mN(\lg(mN)))$.

**Advanced** We compute the edge sequence as before in $O(mN)$ time. We insert the edges in a hash table, using as key value their $x-$coordinate. The insertion is performed in $O(1)$.

For every edge in the sequence we search the hash table for another edge with opposite $x-$coordinate that add to zero. The search is performed in $O(1)$. The total time is $O(mN)$.

## 1.4 Looking for a triangle summand

**Naive** We compute the edge sequence as before in $O(mN)$ time. We can form all the possible triplets and test if they add to zero. The total time is $O(m^3N^3)$.

**Clever** We compute the edge sequence as before in $O(mN)$ time and we sort it with respect to their $x-$coordinate in $O(mN \lg(mN))$ time.

We form all the possible sums of two vectors in $O(m^2N^2)$ time. For every such sum we search, using binary search in $O(\lg(mN))$, for another vector such that their sum is zero. The total time is $O(m^2N^2 \lg(mN))$.

**Advanced** We compute the edge sequence as before in $O(mN)$ time and we sort them in increasing order with respect to their $x-$coordinate in $O(mN\lg(mN))$ time.

For every edge in the sequence, say $k$, scan the sequence from the left to find $i$ and from the right to find $j$, such that $E_k + E_i + E_j = 0$. Because of the sorted order we can advance either $i$ or $j$ according to whether the sum $E_k + E_i + E_j$ is positive or negative. The total time is $O(m^2N^2)$.

Alternatively we can insert the vectors in a hash table, using as a key their $x-$coordinate, and for every sum of 2 vectors we search for a vector in the hash table, in $O(1)$, such that the sum of the three is zero. Again the total time is $O(m^2N^2)$.

## 1.5 Looking for a quad summand

**Naive** We compute the edge sequence as before in $O(mN)$ time. We can form all the possible 4-tuples and test if they add to zero. The total time of the algorithm is $O(m^4N^4)$.

**Clever** We compute the edge sequence as before in $O(mN)$ time. We can form all the possible triplets in $O(m^3N^3)$. For every triplet we search for a vector such that the sum of the four is zero.

If we have sorted the edges with respect to their $x - coordinate$ and we do binary searching then the total time of the algorithm is $O(m^3N^3\lg(mN))$.

If we insert the edges in a hash table, using their $x-$coordinate as key, we perform the search in $O(1)$ time and so the total time of the algorithm is $O(m^3N^3)$.

**Advanced** We compute the edge sequence as before in $O(mN)$ time. We form all the possible sums of 2 edges, in $O(m^2N^2)$ time and we insert them in a hash table, using as key their $x-$coordinate. For every such sum we search the hash table for a vector such that the total sum is zero. The total time is $O(m^2N^2)$.

## 1.6 Computation of the gcd

We present two algorithms that compute the gcd of two integers $a, b$. The first algorithm is the traditional Euclid algorithm, while the second one is the Binary gcd algorithm (see [3] at chapter 4).

---

**Algorithm 1** Euclid_gcd$(a, b)$

---

**Require:** $a, b \in \mathbb{Z}$
**Ensure:** $y = \gcd(a, b)$
  **if** $b = 0$ **then**
    RETURN $a$
  **else**
    RETURN Euclid_gcd$(b, a \mod b)$
  **end if**

---

**Algorithm 2** Binary_gcd$(a, b)$

**Require:** $a, b \in \mathbb{Z}$
**Ensure:** $y = \gcd(a, b)$

```
g = 1
while a is even AND b is even do
   a = a/2 (right shift)
   b = b/2
   g = 2 * g (left shift)
end while
while u > 0 do
   if a is even then
      a = a/2
   else if b is even then
      b = b/2
   else
      t = |a − b|/2
   end if
   if a < b then
      b = t
   else
      a = t
   end if
end while
RETURN g * b
```

## References


[1] S. Gao and A. Lauder, *D*ecomposition of polytopes and Polynomials, Discrete and Computational Geometry 26 (2001), 501-520

[2] L.J. Guibas and R. Seidel, *C*omputing Convolutions by Reciprocal Search, Symposium of Computational Geometry, 2001

[3] D. Knuth, *T*he Art of Computer Programming, vol 2, Addison-Wesley, 3rd edition, 1998