# 1. A linear space alignment algorithm

Slides for the Lecture on
Foundations of Sequence Analysis
Winter 2022/2023
Stefan Kurtz

January 17, 2023

# Motivation for reducing the space requirement

- The classic algorithm to solve the edit distance problem for two sequences $u$ and $v$ of length $m$ and $n$ takes $O(mn)$ time and space.
- This is because it needs to store an $(m+1) \times (n+1)$-table $E_\delta$ from which the alignments are computed by a traceback method.
- It is difficult to improve the running time of this algorithm, if the cost function is arbitrary, independent of whether linear or affine gap costs are used.
- In this section we will improve the space requirement by deriving an algorithm that only takes $O(m+n)$ space to compute a single optimal alignment of $u$ and $v$.
- For ease of presentation we will restrict to linear gap costs, although the method can be generalized to affine gap costs.
- In most cases, it suffices to compute only a single alignment, and so the linear space algorithm is very important in practice.

# Recalling the linear space distance only method

Recall that matrix $E_\delta$ satisfies

$$E_\delta(i,j) = edist_\delta(u[1\ldots i], v[1\ldots j])$$

for all $i, j$, $0 \leq i \leq m$ and $0 \leq j \leq n$. It can be computed according to the following recurrence:

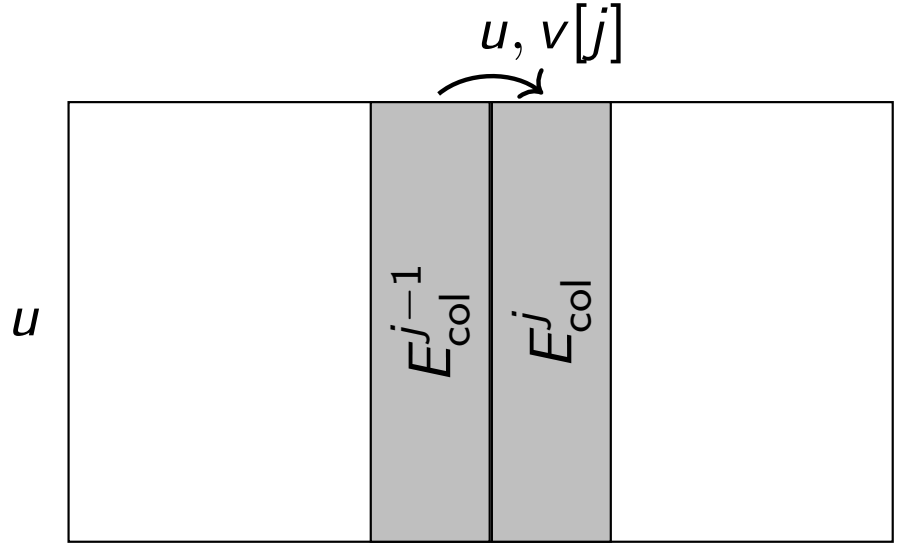
$$E_\delta(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ and } j = 0 \\ E_\delta(i-1,0) + \delta(u[i] \to \varepsilon) & \text{if } i > 0 \text{ and } j = 0 \\ E_\delta(0,j-1) + \delta(\varepsilon \to v[j]) & \text{if } i = 0 \text{ and } j > 0 \\ \min \left\{ \begin{array}{l} E_\delta(i-1,j) + \delta(u[i] \to \varepsilon) \\ E_\delta(i,j-1) + \delta(\varepsilon \to v[j]) \\ E_\delta(i-1,j-1) + \delta(u[i] \to v[j]) \end{array} \right\} & \text{if } i > 0 \text{ and } j > 0 \end{cases}$$

# Recalling the linear space distance only method

For each $j$, $0 \leq j \leq n$, let

$$E^j_{\text{col}} = E_\delta(0,j), \ldots, E_\delta(m,j)$$

denote the $j$th column of table $E_\delta$. $E^j_{\text{col}}$ is a vector of $m+1$ values. We will later refer to these columns as *distance columns*. As already observed earlier, if $j > 0$, then $E^j_{\text{col}}$ can be computed from $E^{j-1}_{\text{col}}$, the sequence $u$, and the current character $v[j]$, as illustrated here:




– This follows from a simple analysis of the data dependencies of the fundamental recurrences for $E_\delta$.

# Recalling the linear space distance only method

– Hence one can compute the sequence of column vectors

$$E^0_{\text{col}}, E^1_{\text{col}}, \ldots, E^n_{\text{col}},$$

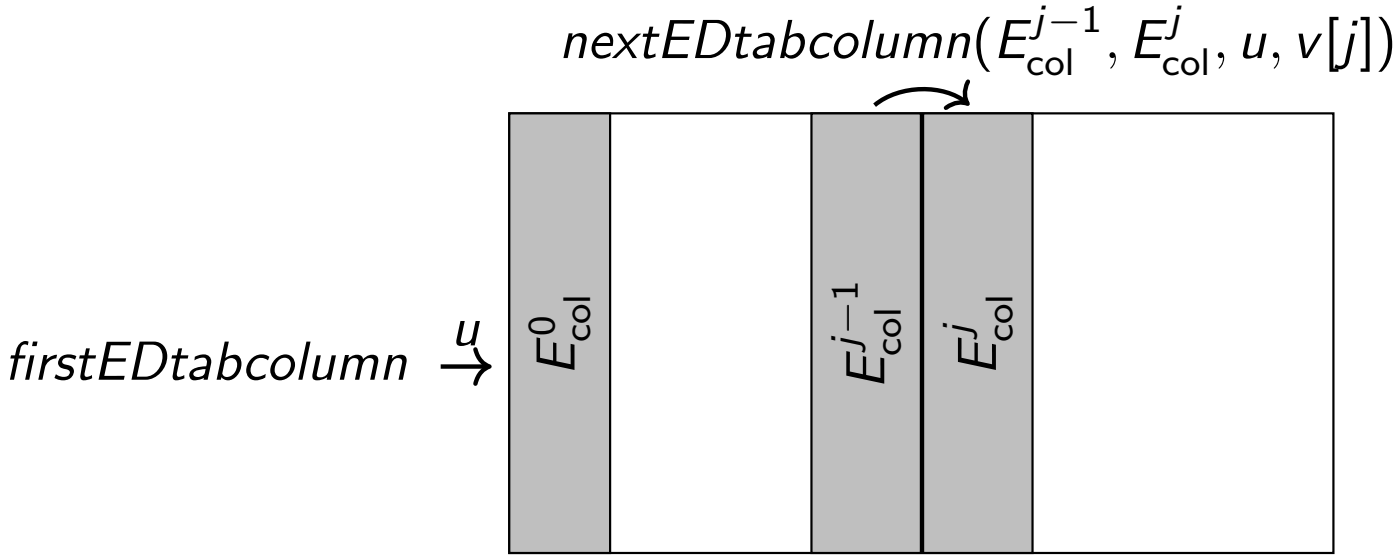
such that only two columns, the previous and the one being computed, need be stored at any time.

– Up until now in this section we have recalled what we already knew about the computation of matrix $E_\delta$.

– We have done this in a more explicit way by introducing a notation for the columns vectors of matrix $E_\delta$.

# Recalling the linear space distance only method

- The computation of the sequence of column vectors can be described by a function *nextEDtabcolumn* which takes four parameters: the vectors storing the current and previous column, the current character $b$, and the sequence $u$.
- This function is complemented by a function *firstEDtabcolumn* which computes $E^0_{\text{col}}$, given $u$.
- Both functions can be used in a function *evaluateallEDtabcolumns* which takes an $(m+1)$-vector $E$ and computes the last column $E^m_{\text{col}}$ of table $E_\delta$, see Algorithm 1 and the following illustration:



## Algorithm 1 (Computation of first and next distance column)

1: **function** *firstEDtabcolumn*$(E, u)$
2: $\quad E(0) \leftarrow 0$
3: $\quad$ **for** $i \leftarrow 1$ **upto** $|u|$ **do** $E(i) \leftarrow E(i-1) + \delta(u[i] \rightarrow \varepsilon)$
4: $\quad$ **end for**
5: **end function**
6: **function** *nextEDtabcolumn*$(E, E', u, b)$
7: $\quad E'(0) \leftarrow E(0) + \delta(\varepsilon \rightarrow b)$
8: $\quad$ **for** $i \leftarrow 1$ **upto** $|u|$ **do** $E'(i) \leftarrow \min \left\{ \begin{array}{l} E'(i-1) + \delta(u[i] \rightarrow \varepsilon) \\ E(i) + \delta(\varepsilon \rightarrow b) \\ E(i-1) + \delta(u[i] \rightarrow b) \end{array} \right\}$
9: $\quad$ **end for**
10: **end function**
11: **function** *evaluateallEDtabcolumns*$(E, u, v)$
12: $\quad$ *firstEDtabcolumn*$(E, u)$;**allocate vector** $E'$ of length $|u| + 1$
13: $\quad$ **for** $j \leftarrow 1$ **upto** $|v|$ **do** *nextEDtabcolumn*$(E, E', u, v[j])$; $E \leftarrow E'$;
14: $\quad$ **end for**
15: **end function**

# Recalling the linear space distance only method

- This distance-only algorithm calls *evaluateallEDtabcolumns*$(E, u, v)$ and delivers $E^n_{\text{col}}$ in vector $E$.
- The cost $E_\delta(m, n) = E^n_{\text{col}}(m)$ of an optimal alignment is thus computed in only $O(m)$ space.
- One may even improve the computation of successive columns, so that only *one* $(m + 1)$-element vector and two additional scalars are required to overwrite the new column values into the vector holding the previous column values.
- The solution to this task, still requiring $O(m)$ space was already developed in an exercise.

# Applying the divide and concquer strategy

- Now consider how to deliver a single optimal alignment using the algorithm of [Powell et al., 1999].
- Some parts of the presentation follow [Myers, 1998].
- First recall that the optimal alignment can be considered as a path in the edit-graph $G_\delta(u, v)$ which can be reconstructed by a traceback from node $(m, n)$ to $(0, 0)$.

# Applying the divide and concquer strategy

- The distance-only algorithm has thrown away all the information needed for the traceback approach, as it only has available the last column of the table upon completion.
- The idea of the linear-space optimal alignment algorithm is to keep some of the history and apply a divide-and-conquer strategy to obtain a representation of an optimal alignment from this history.
- The method works as follows: consider the column $j_0 = \lfloor n/2 \rfloor$ and find a node $(i_0, j_0)$ that is on a minimizing path from node $(0, 0)$ to node $(m, n)$ in the edit-graph $G_\delta(u, v)$.
- Then recursively apply the same method to find a minimizing path from $(0, 0)$ to $(i_0, j_0)$ and a minimizing path from $(i_0, j_0)$ to $(m, n)$.
- Each step of this recursion delivers an additional crosspoint besides the two crosspoints $(0, 0)$ and $(m, n)$.
- In particular, up to recursion level $k$, where $k \leq \log_2 n$, the method has computed $cpc(k) = \sum_{i=1}^{k} 2^{i-1} = 2^k - 1$ crosspoints.
- E.g. $cpc(1) = 1$, $cpc(2) = 3$, $cpc(\log_2 n) = 2^{\log_2 n} - 1 = n - 1$.

Figure 1: The divide and conquer strategy of the linear space alignment algorithm

Figure 1: The divide and conquer strategy of the linear space alignment algorithm

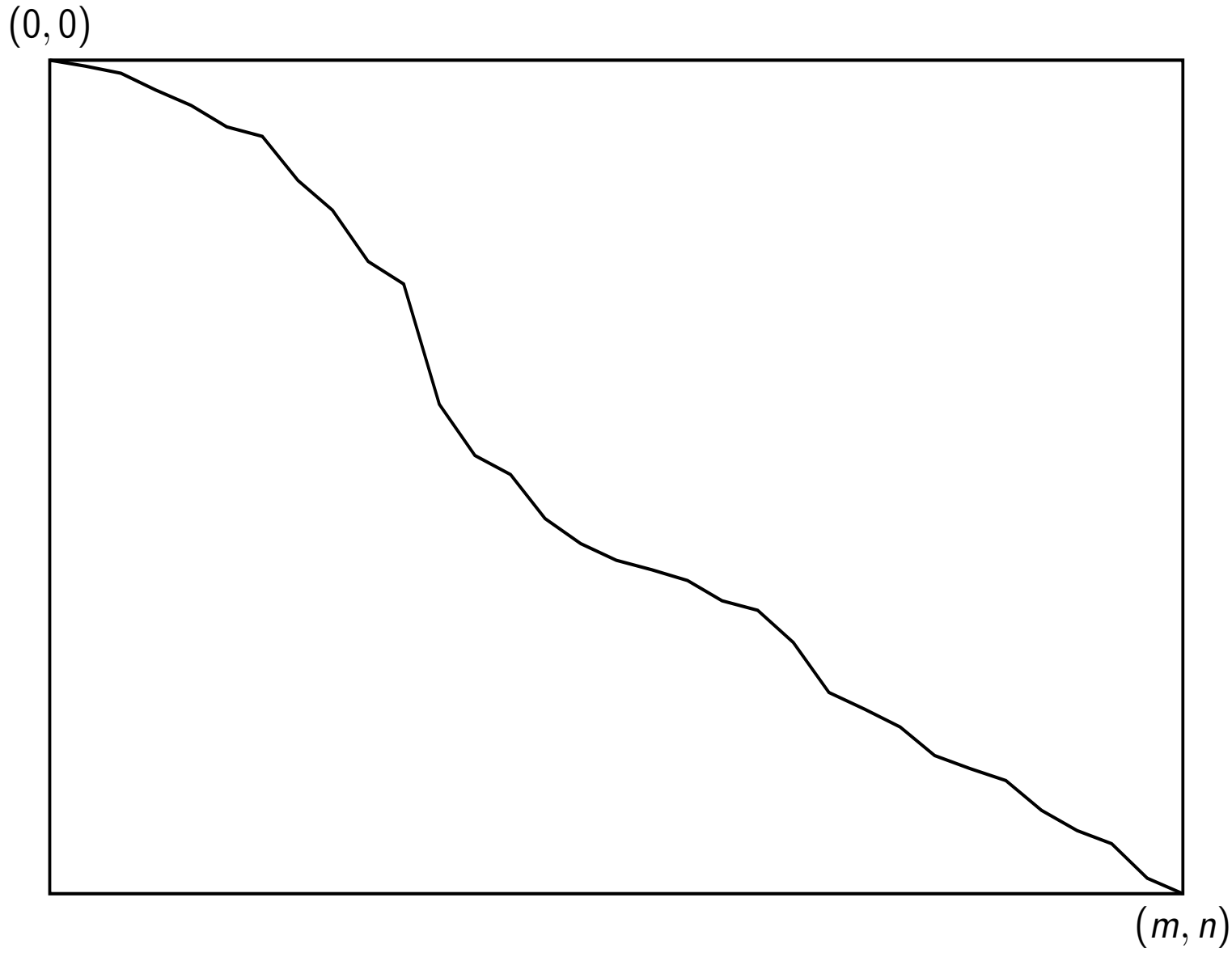


Figure 1: The divide and conquer strategy of the linear space alignment algorithm

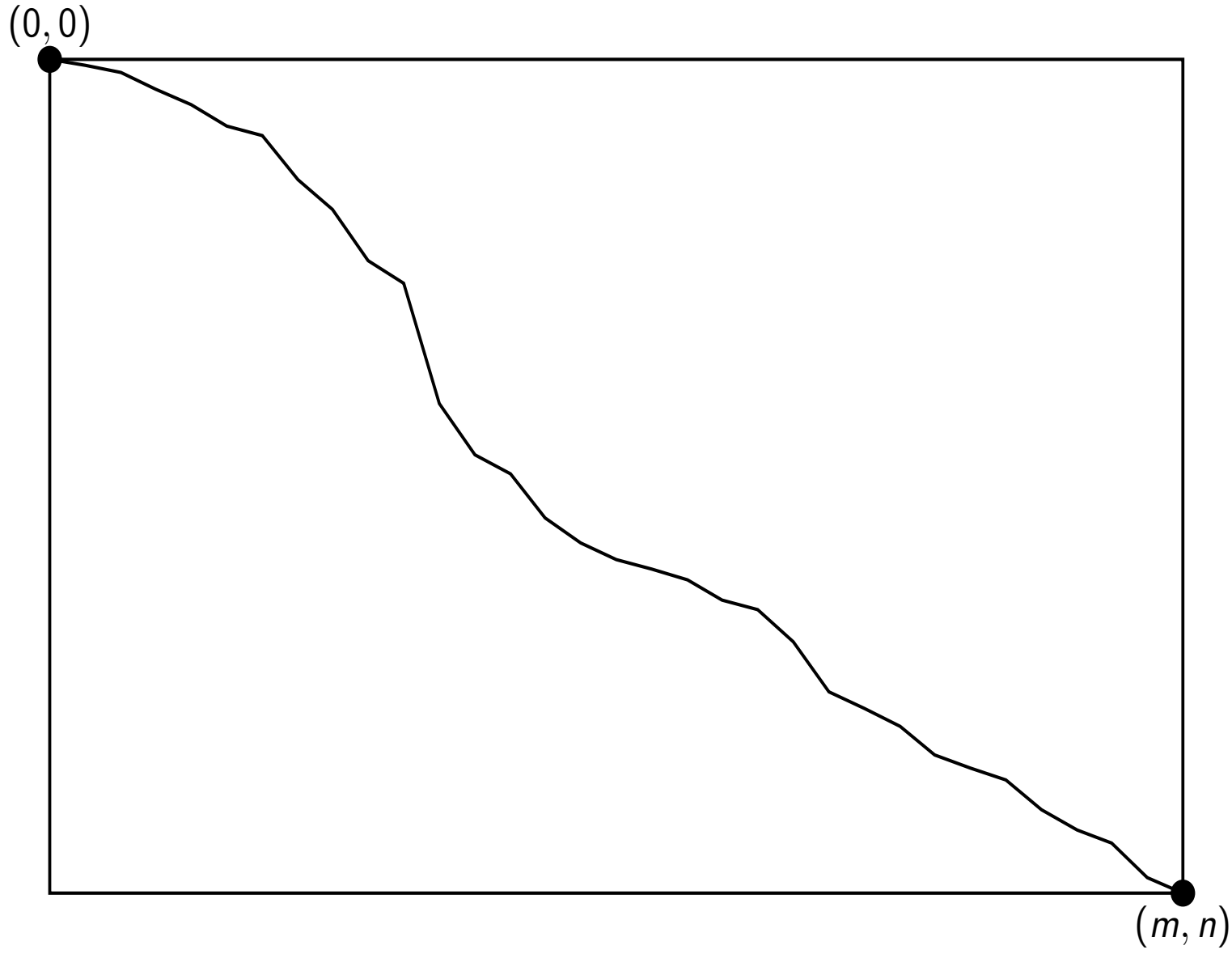


Figure 1: The divide and conquer strategy of the linear space alignment algorithm

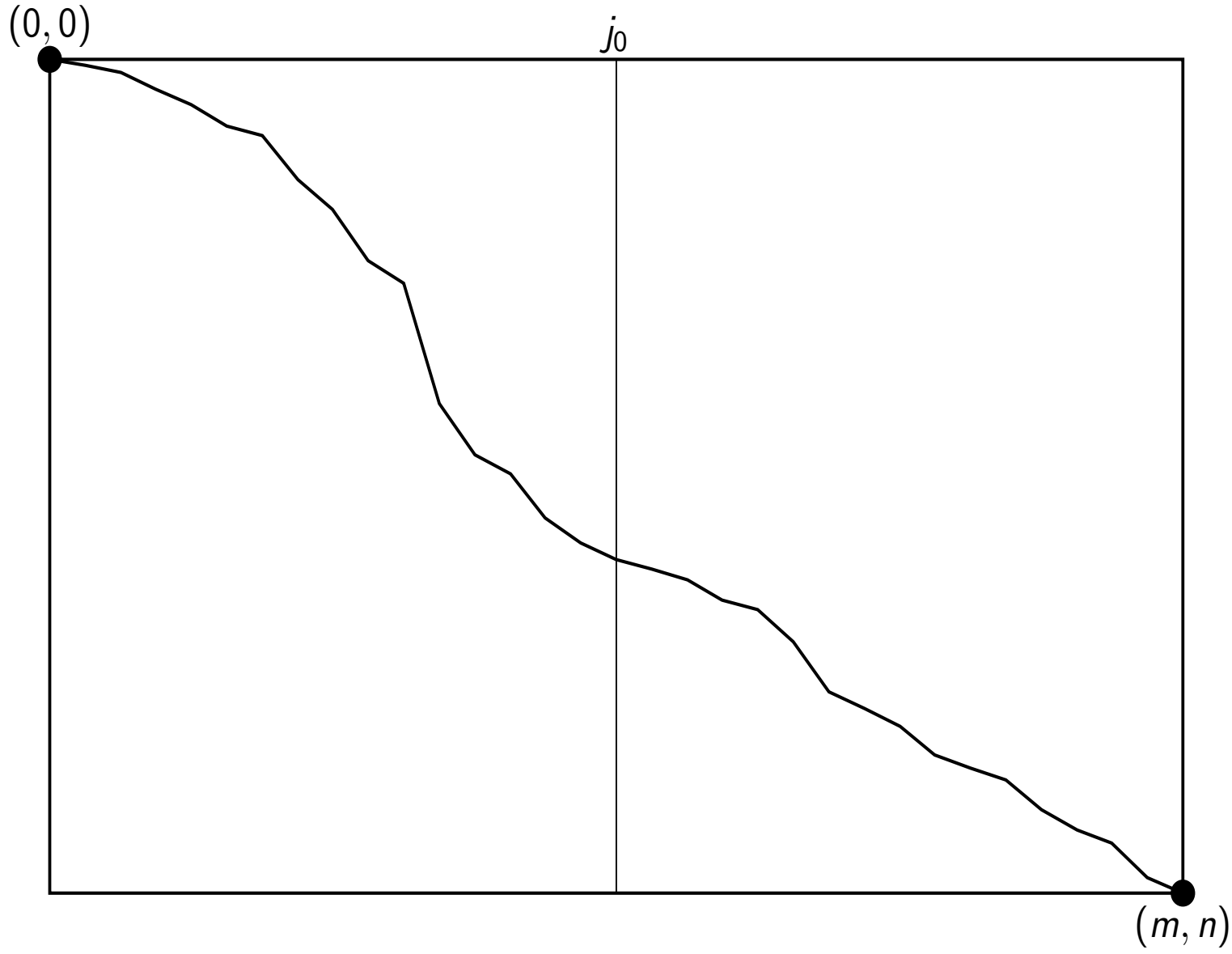


Figure 1: The divide and conquer strategy of the linear space alignment algorithm

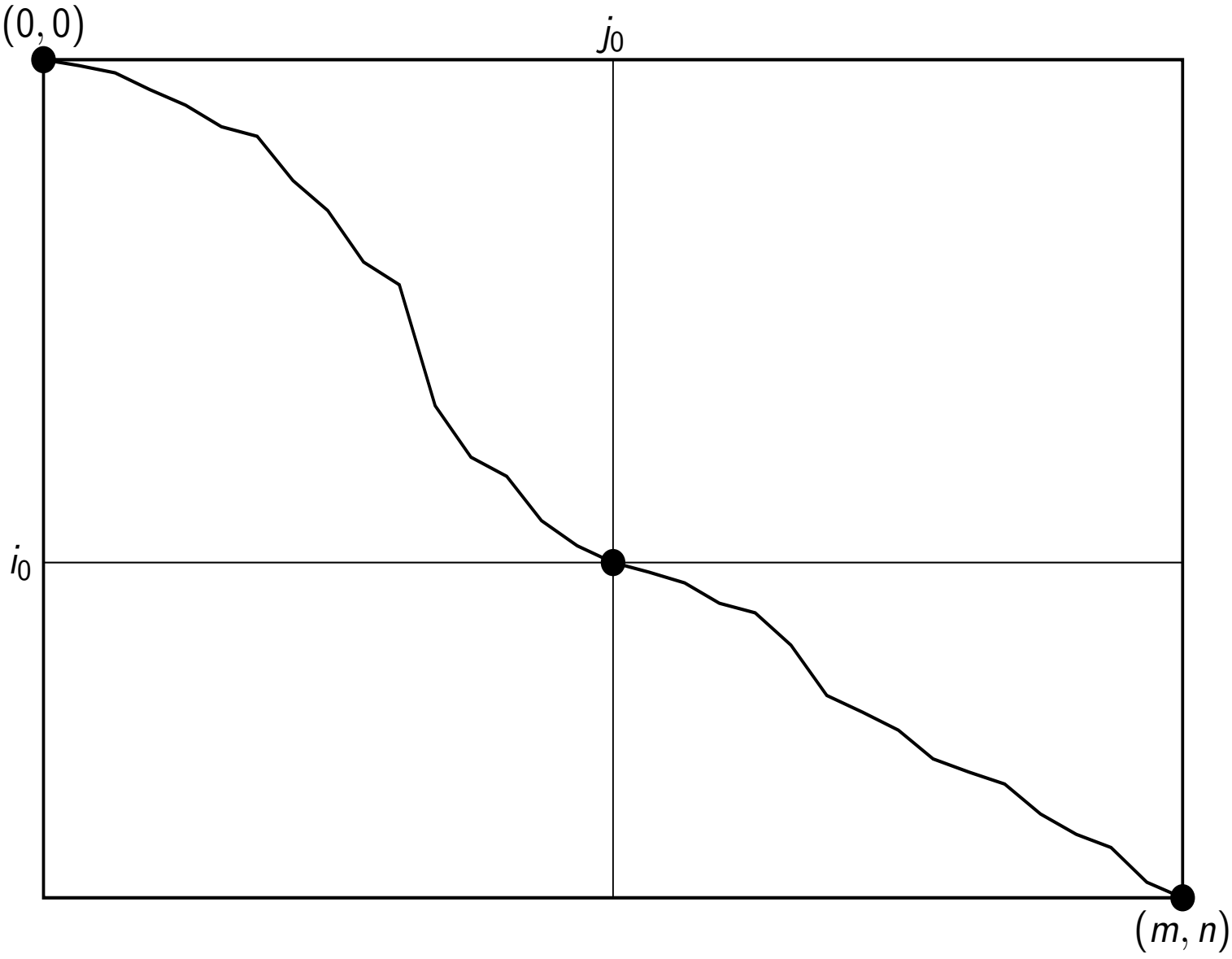


Figure 1: The divide and conquer strategy of the linear space alignment algorithm

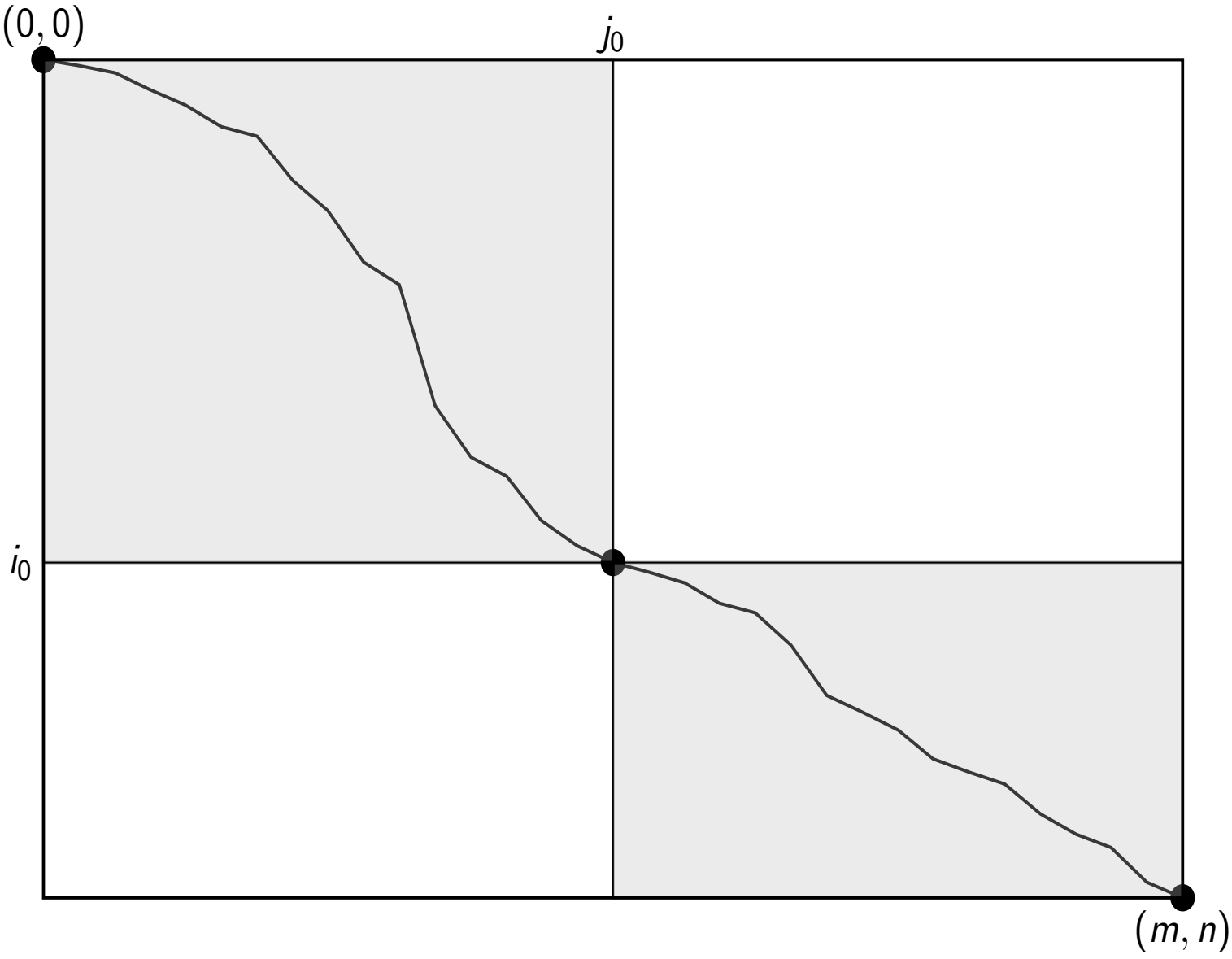


Figure 1: The divide and conquer strategy of the linear space alignment algorithm

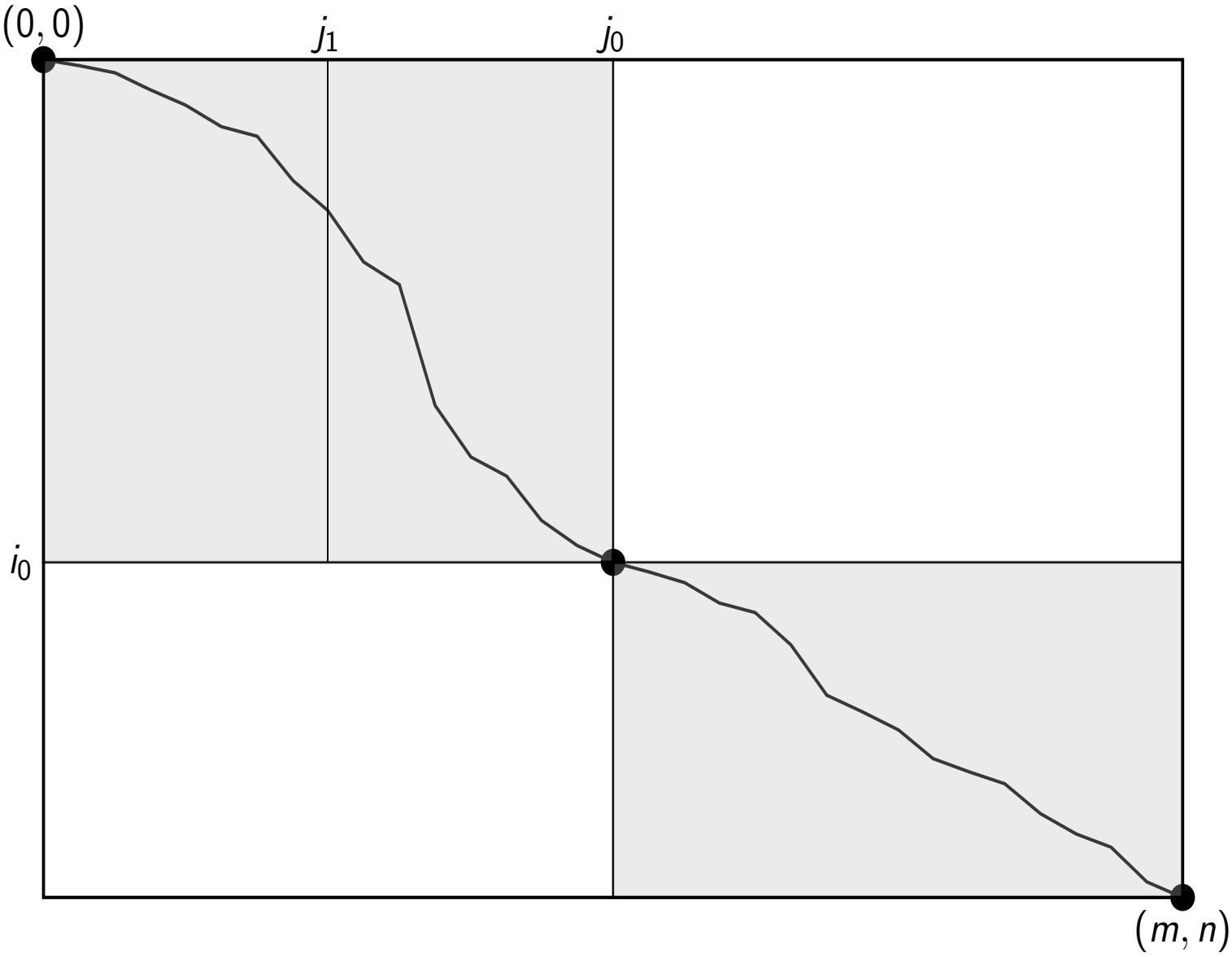


Figure 1: The divide and conquer strategy of the linear space alignment algorithm

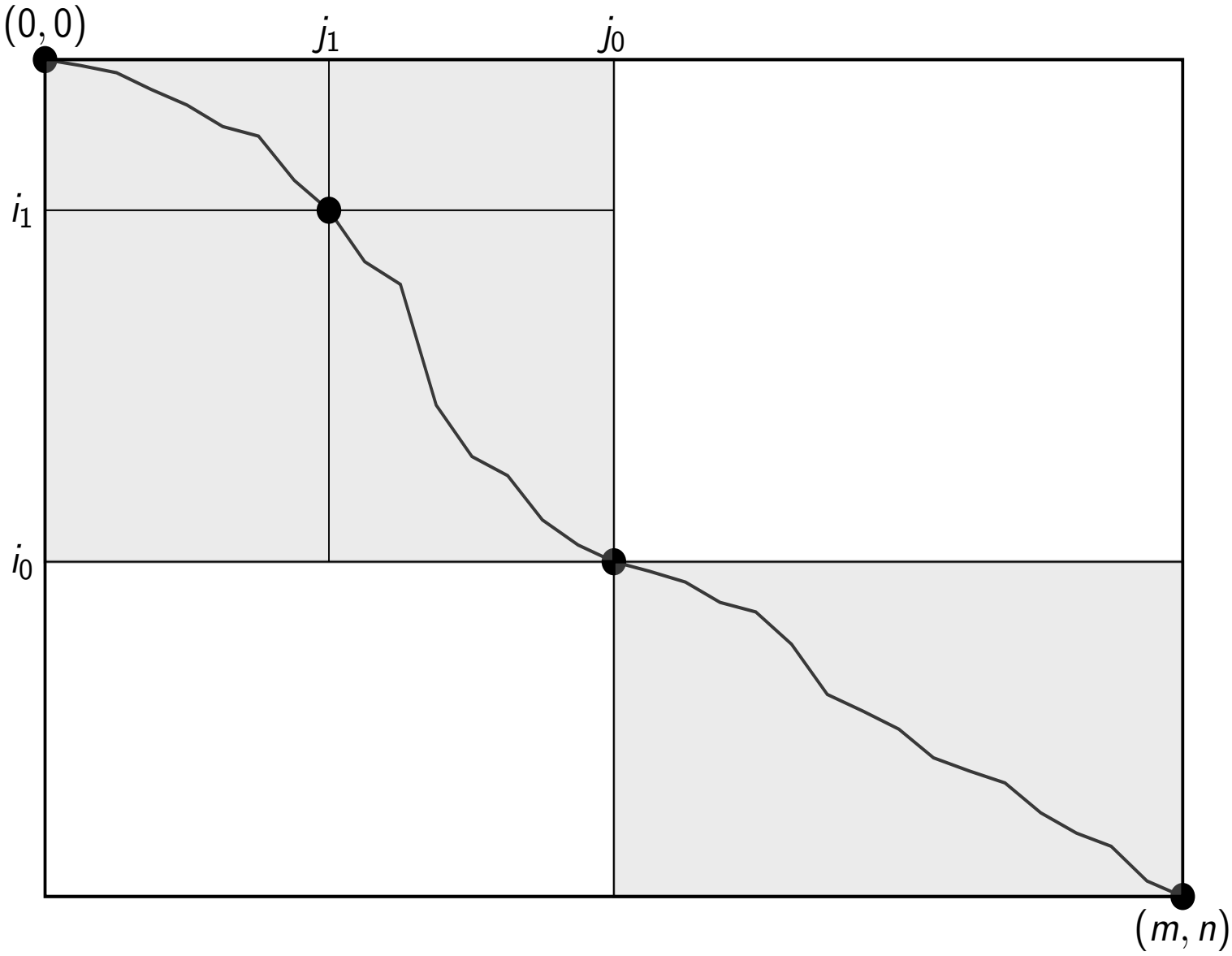


Figure 1: The divide and conquer strategy of the linear space alignment algorithm

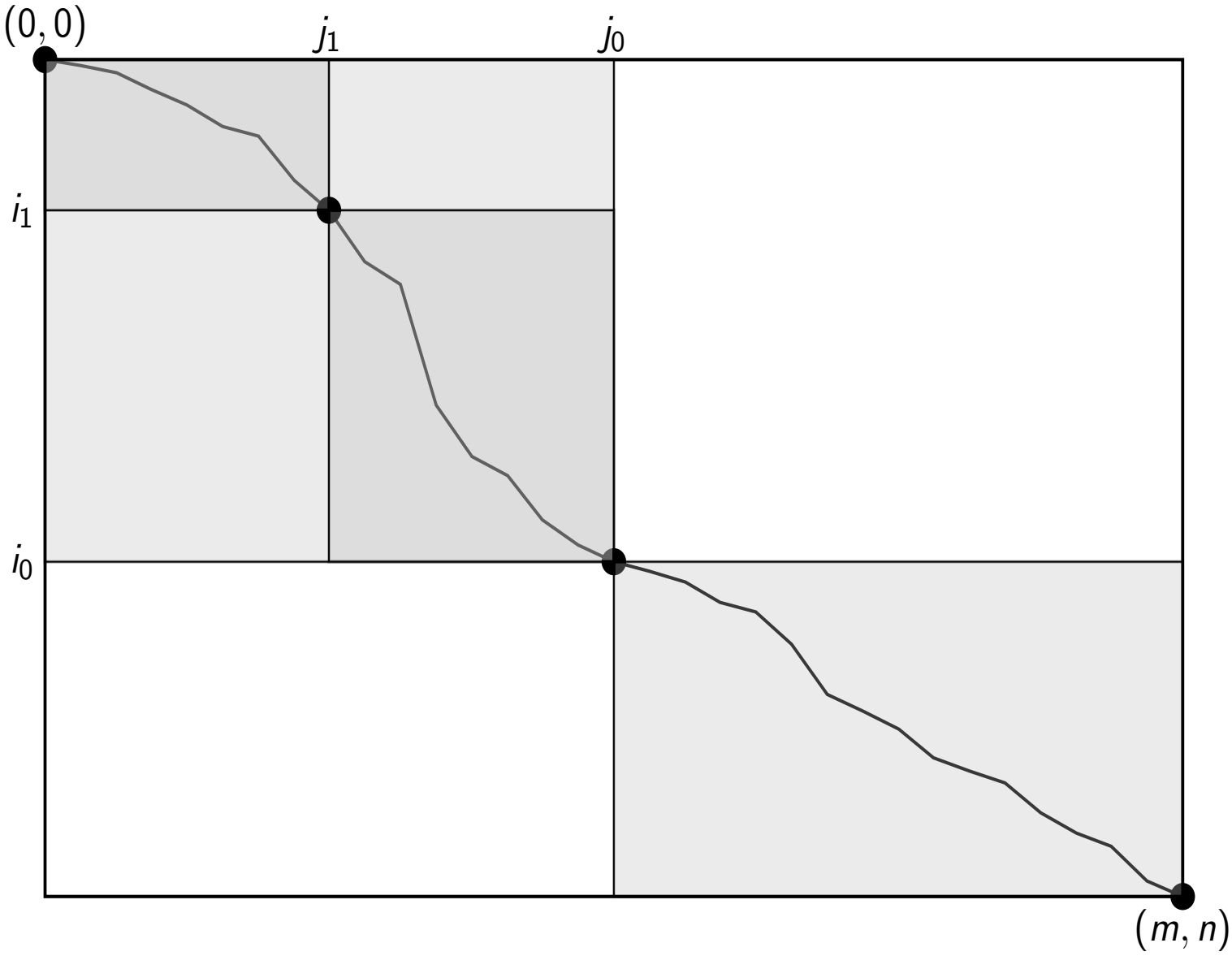


Figure 1: The divide and conquer strategy of the linear space alignment algorithm

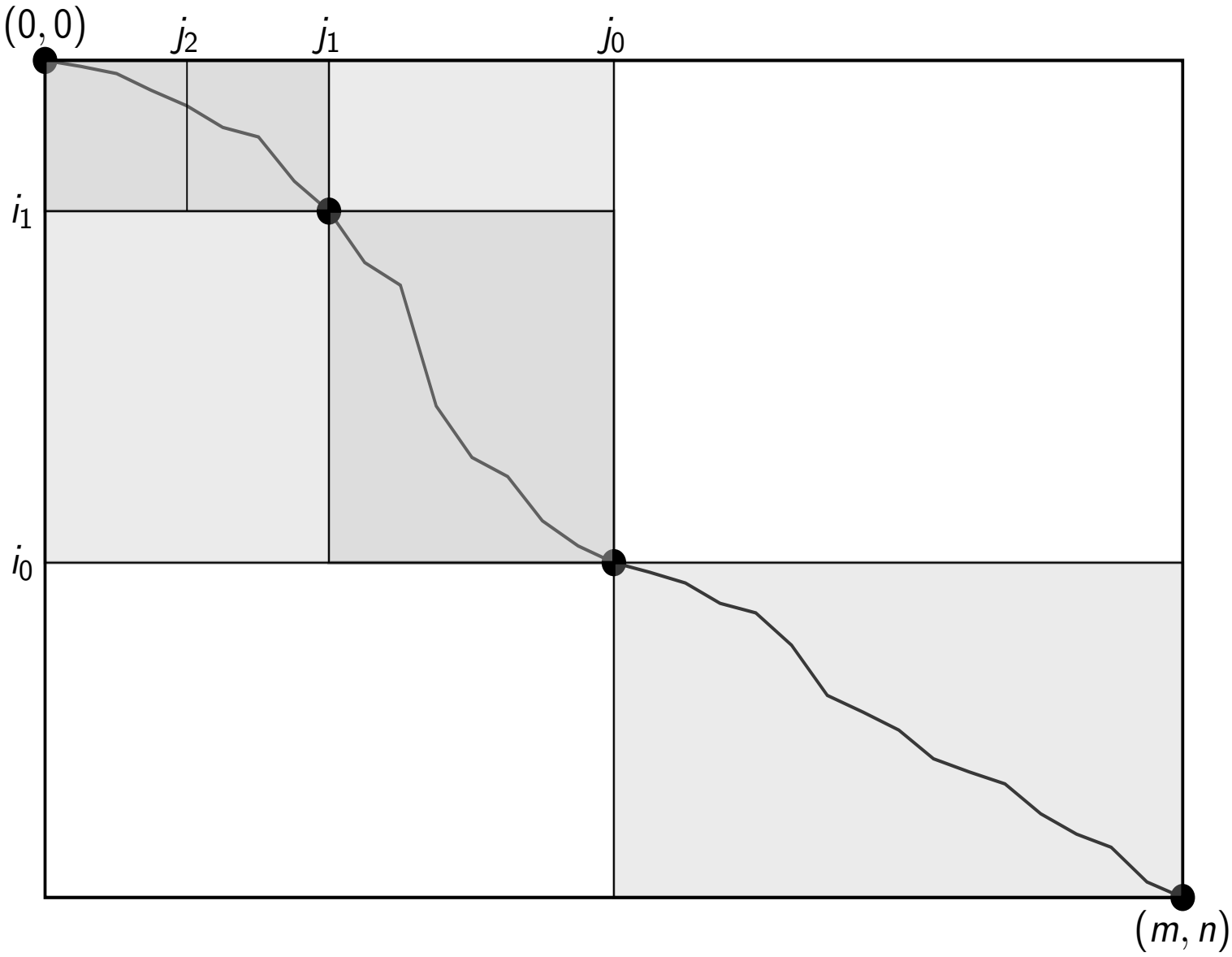


Figure 1: The divide and conquer strategy of the linear space alignment algorithm

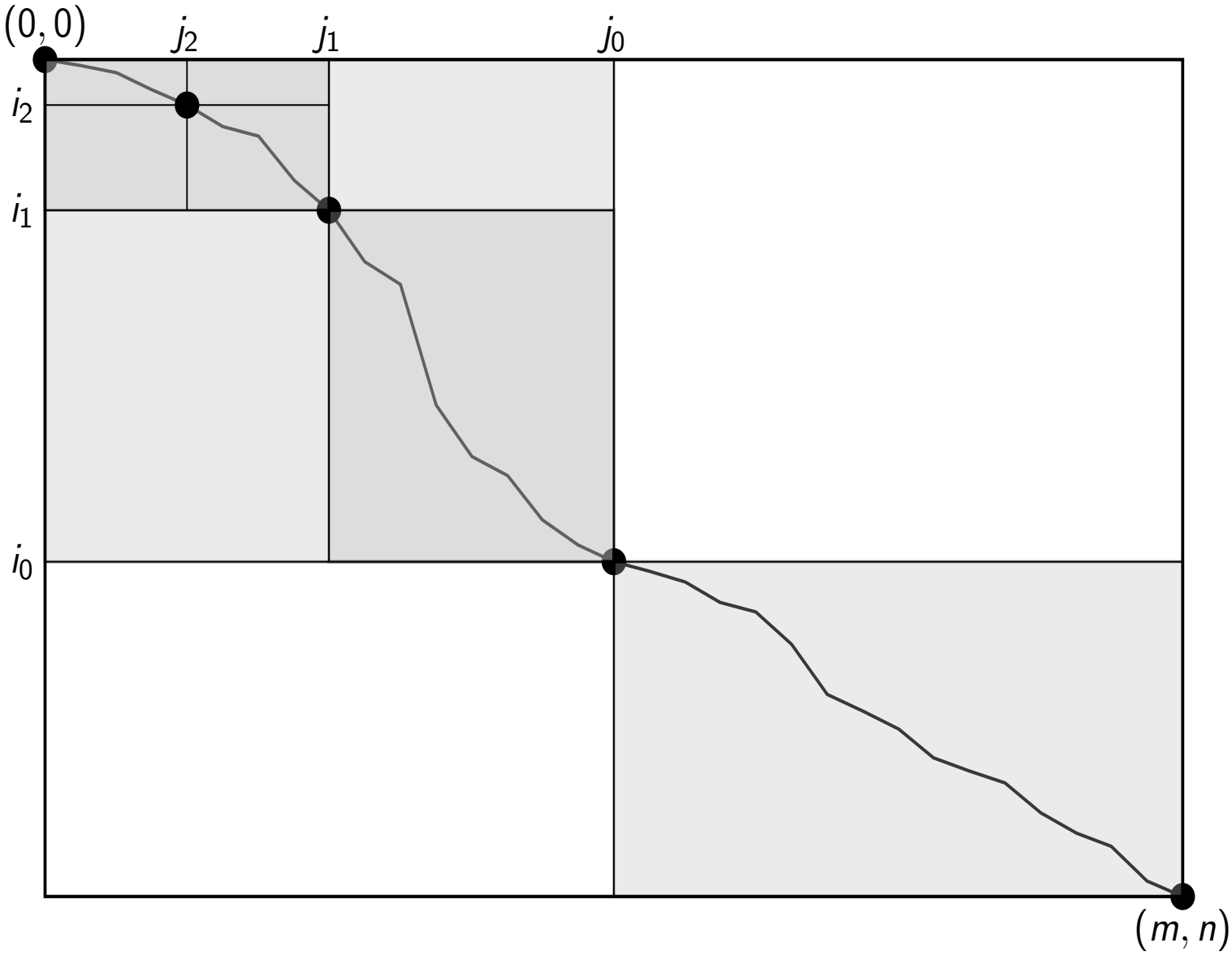


Figure 1: The divide and conquer strategy of the linear space alignment algorithm

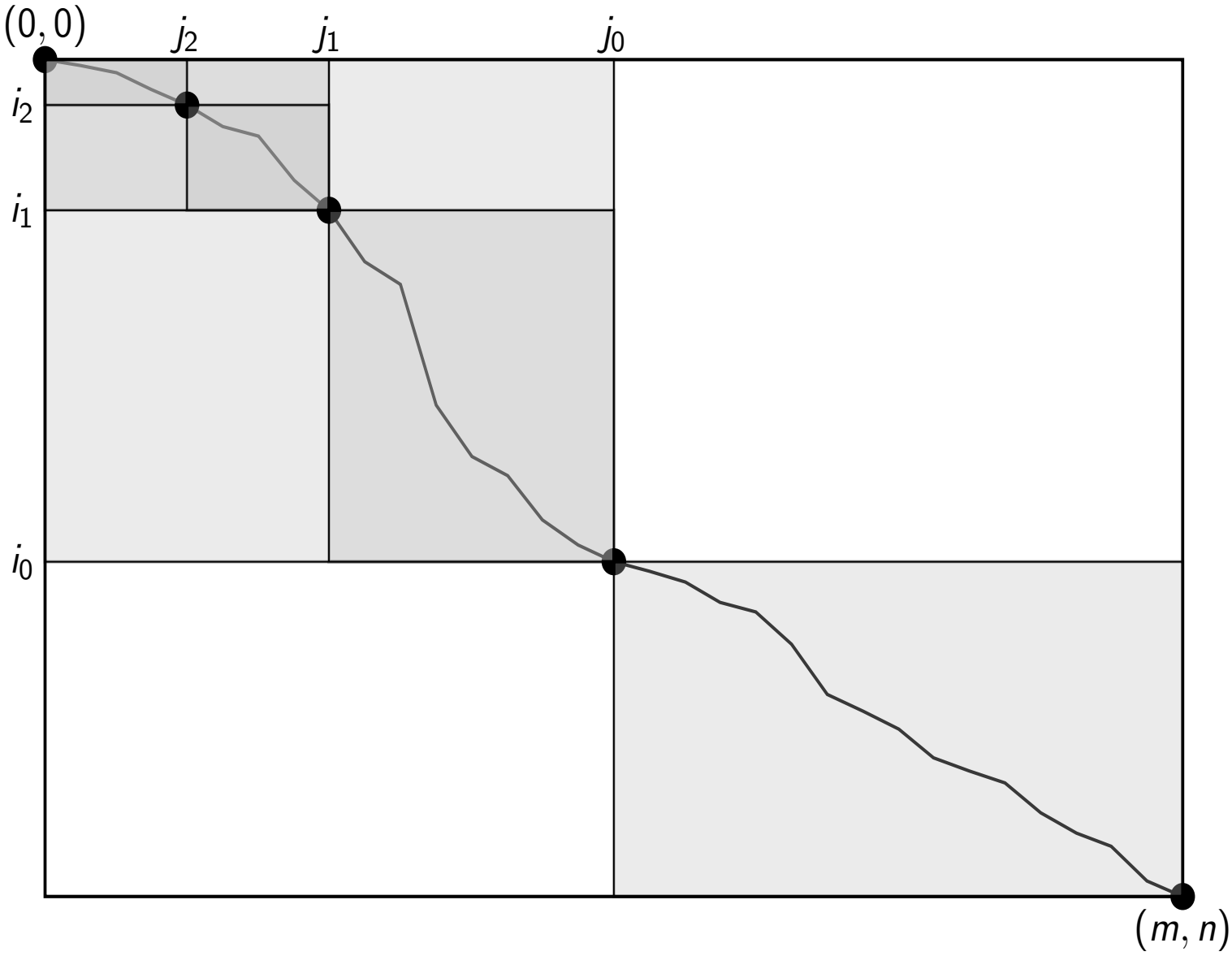


Figure 1: The divide and conquer strategy of the linear space alignment algorithm

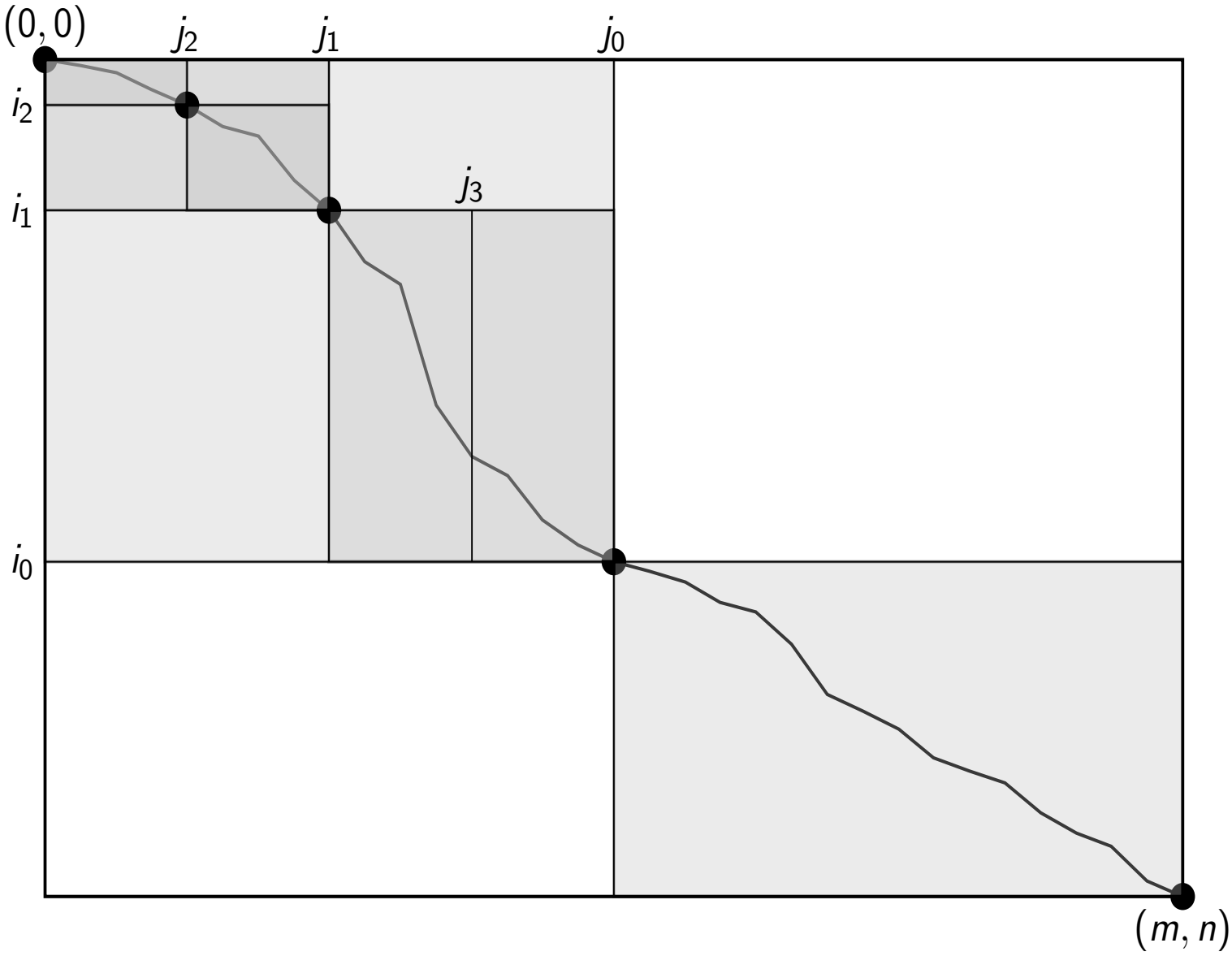


Figure 1: The divide and conquer strategy of the linear space alignment algorithm

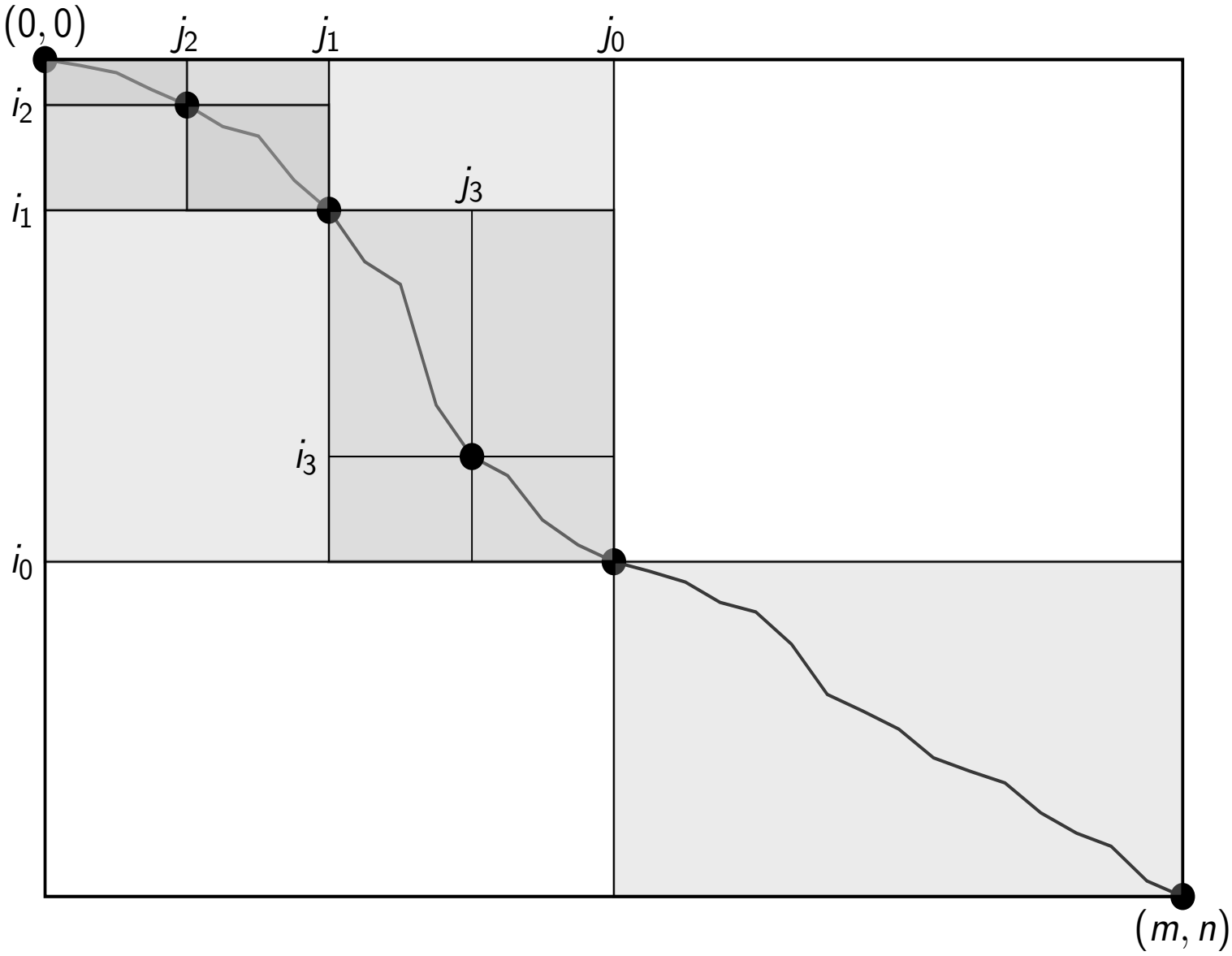


Figure 1: The divide and conquer strategy of the linear space alignment algorithm

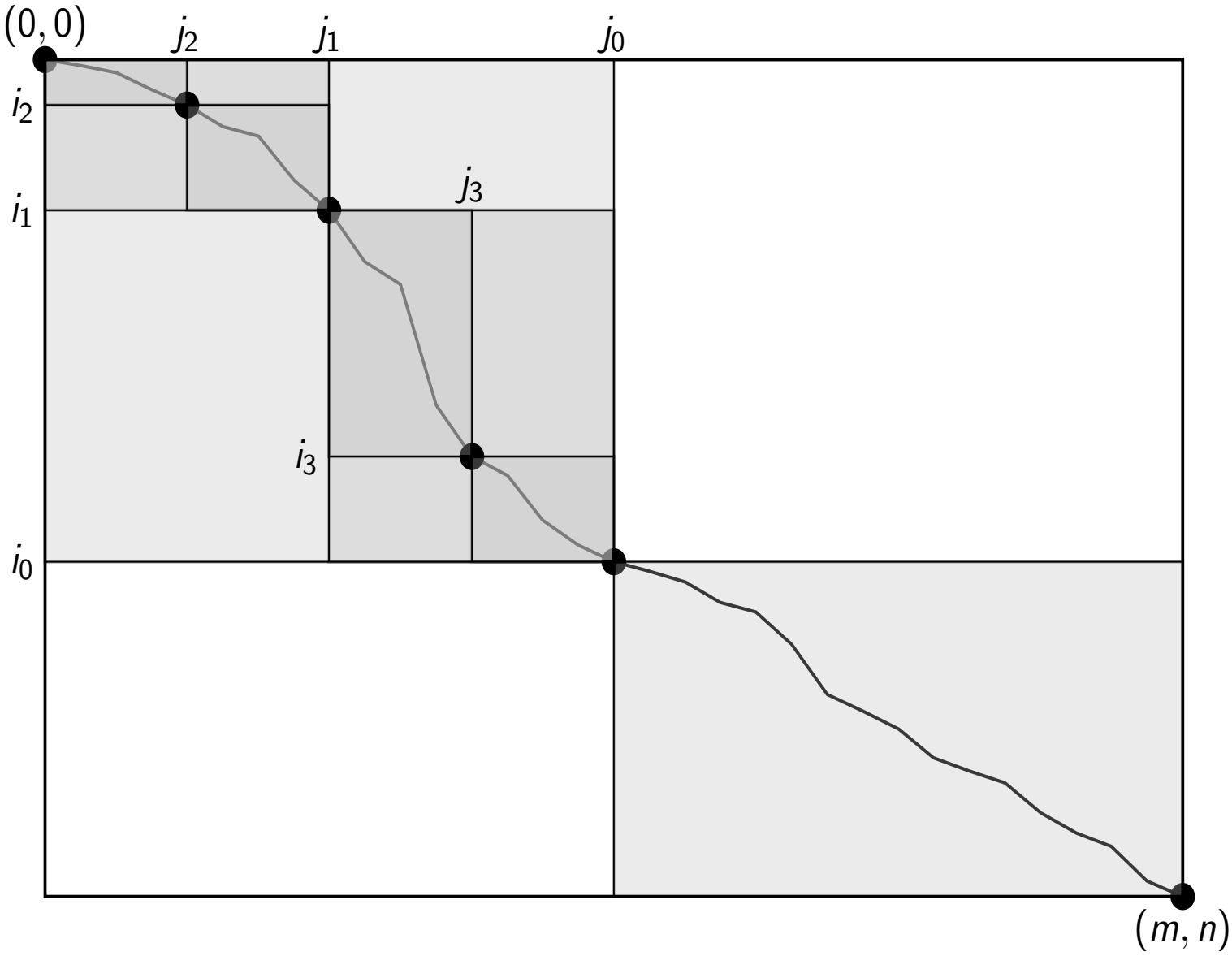


Figure 1: The divide and conquer strategy of the linear space alignment algorithm

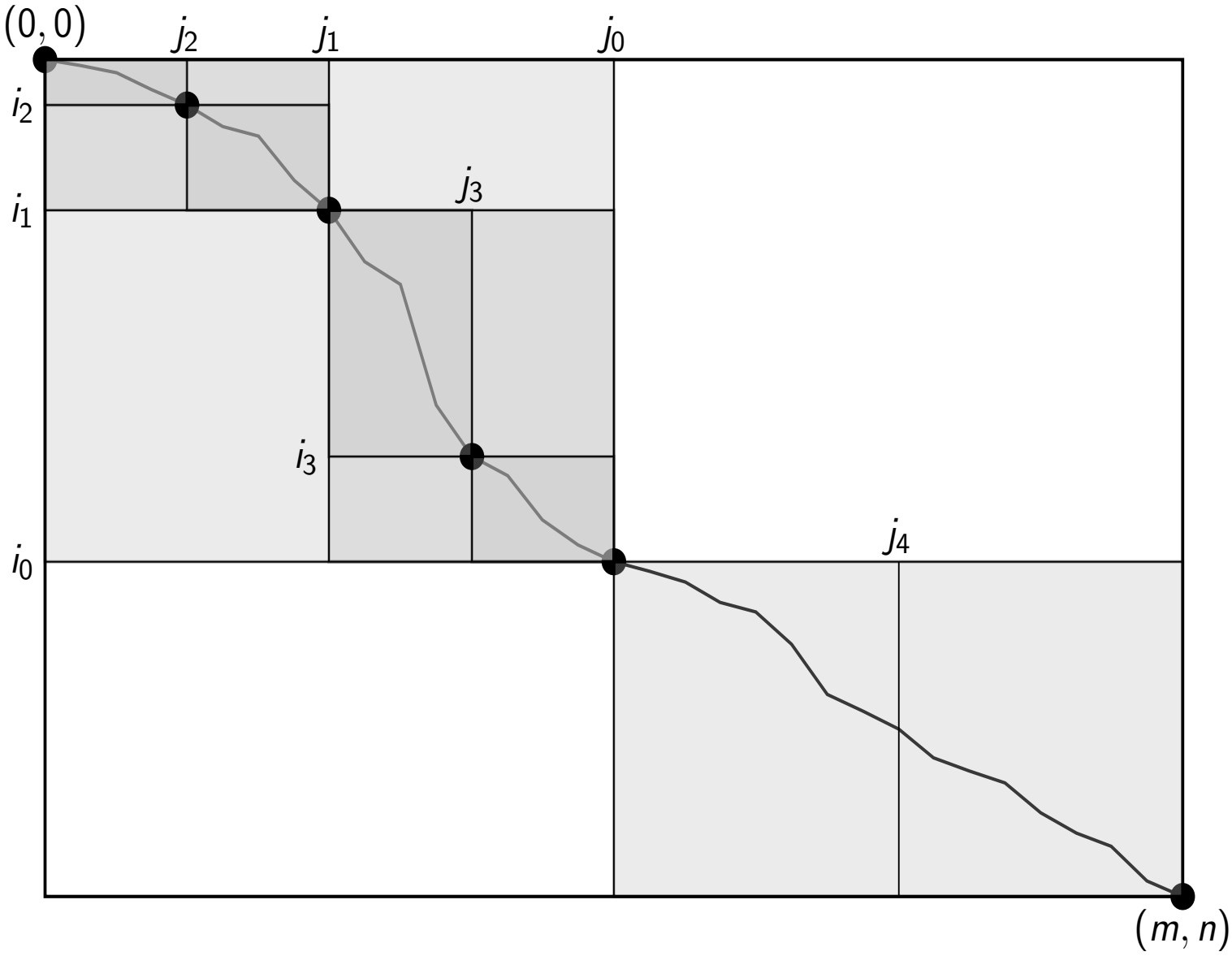


Figure 1: The divide and conquer strategy of the linear space alignment algorithm

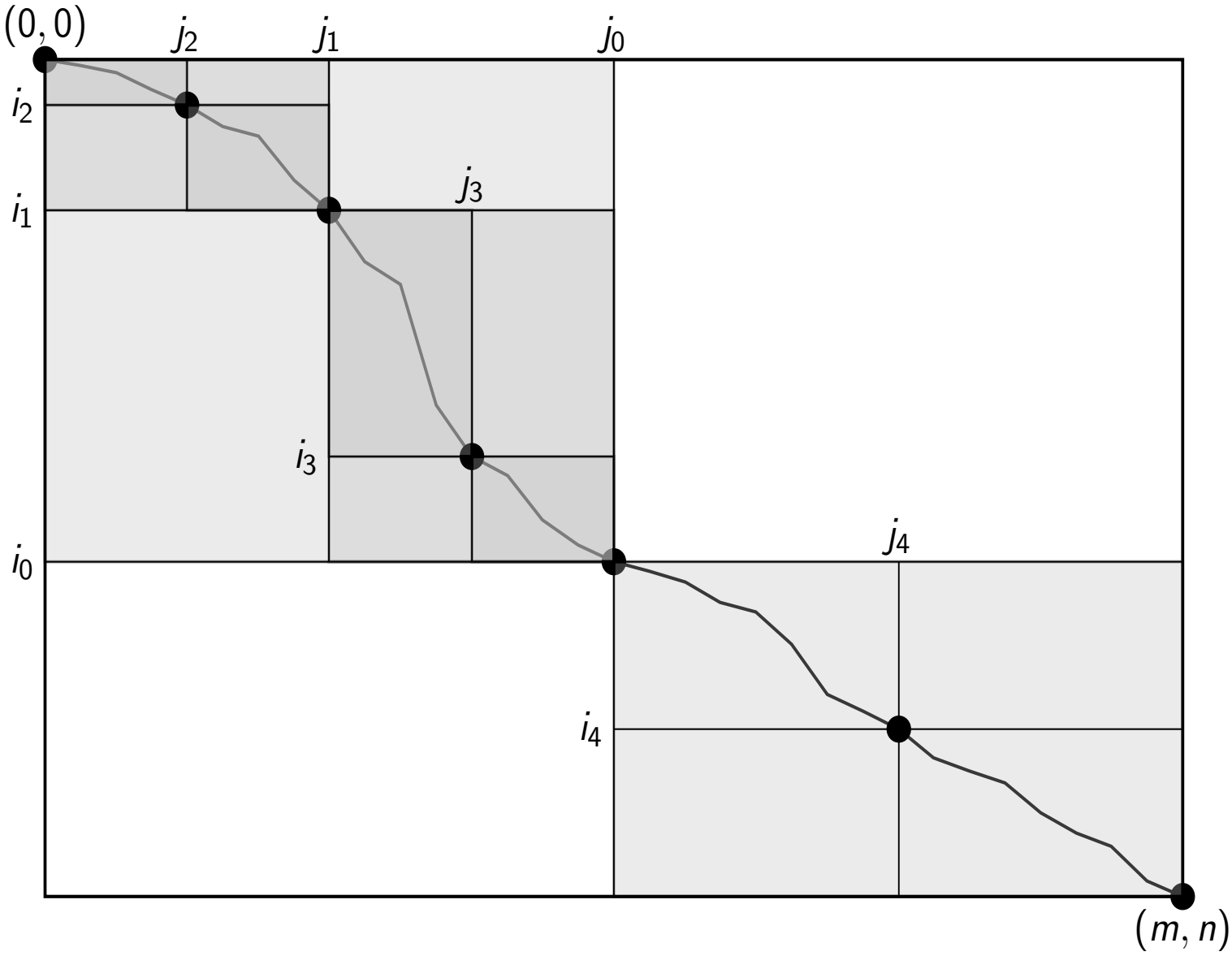


Figure 1: The divide and conquer strategy of the linear space alignment algorithm

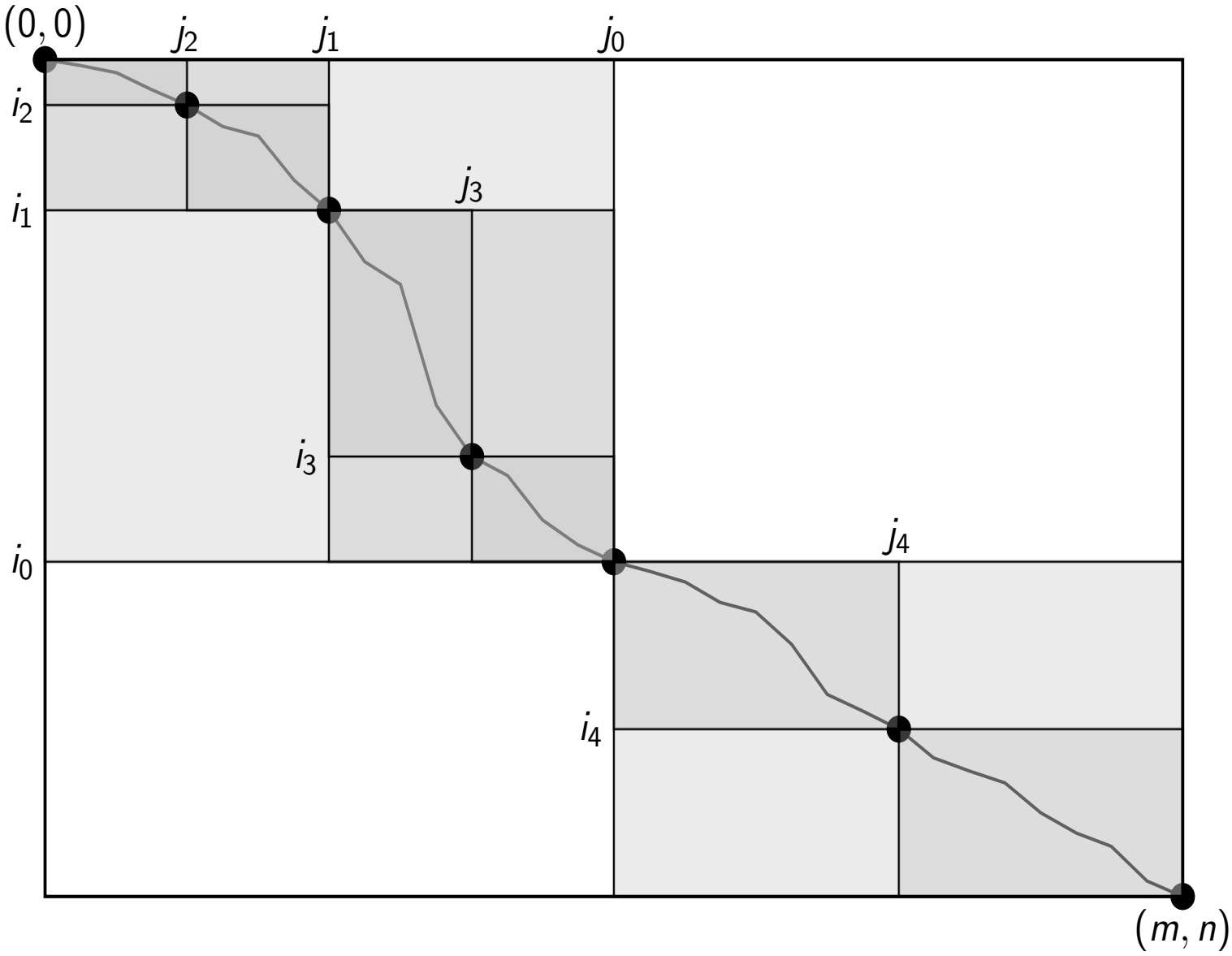


Figure 1: The divide and conquer strategy of the linear space alignment algorithm

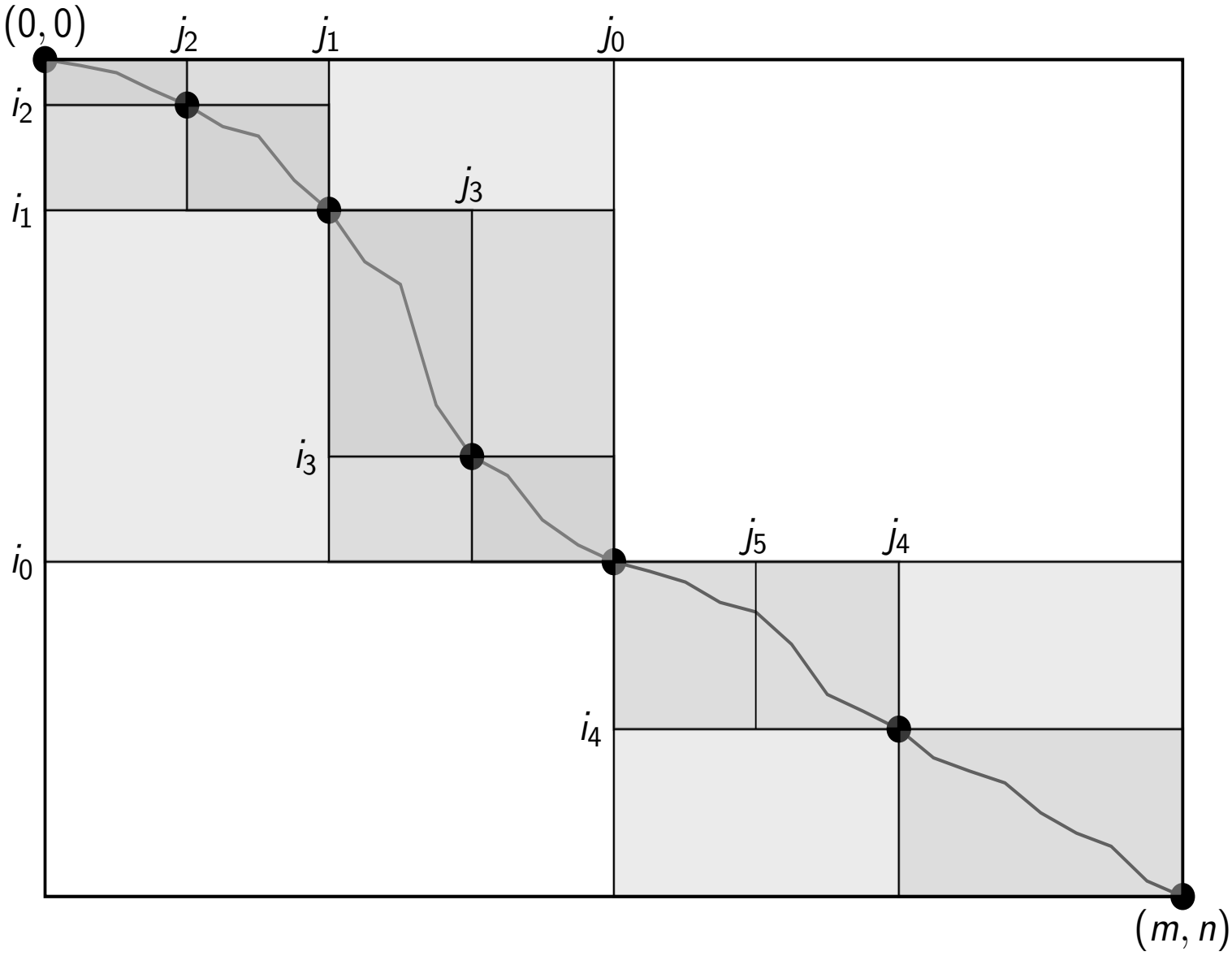


Figure 1: The divide and conquer strategy of the linear space alignment algorithm

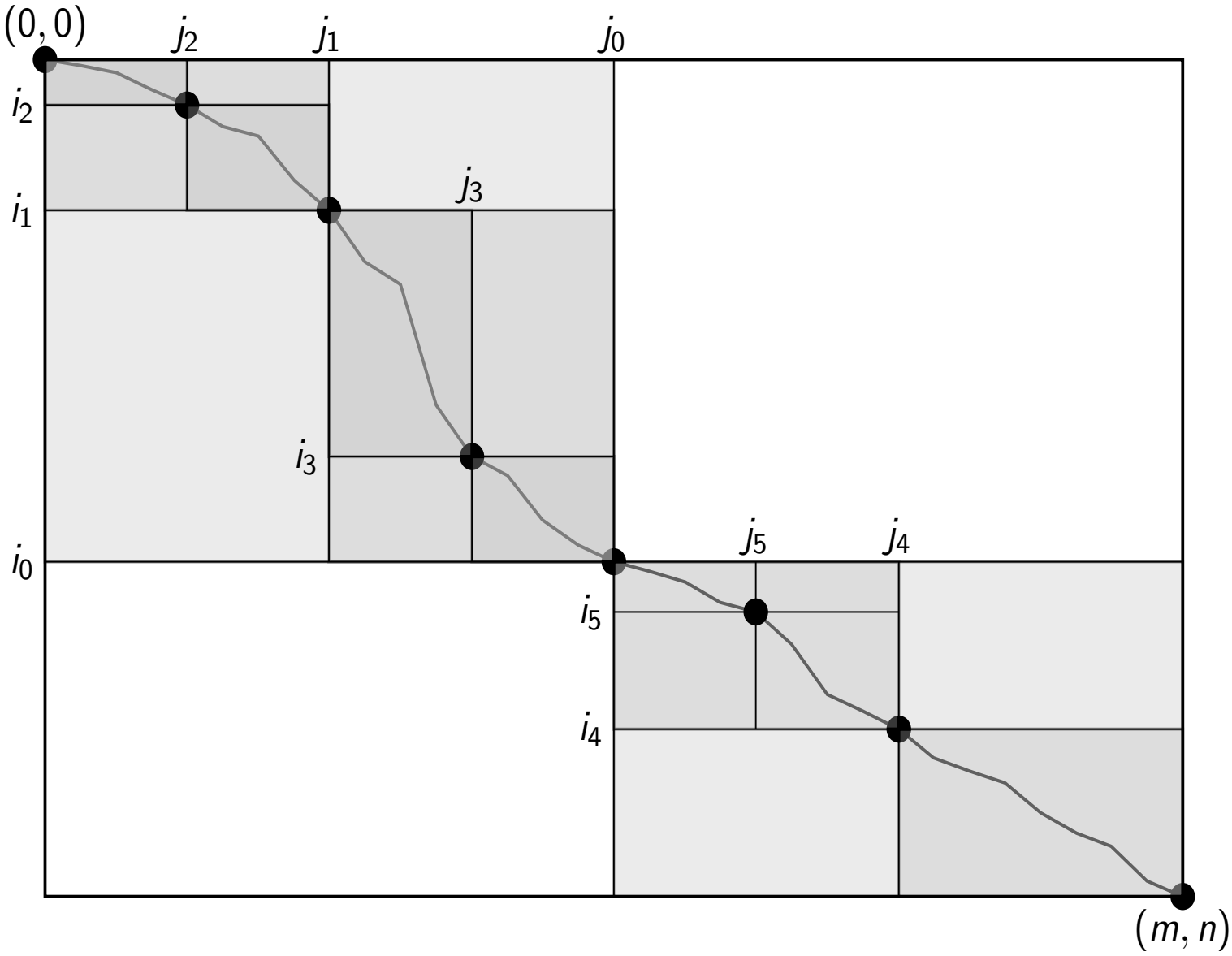


Figure 1: The divide and conquer strategy of the linear space alignment algorithm

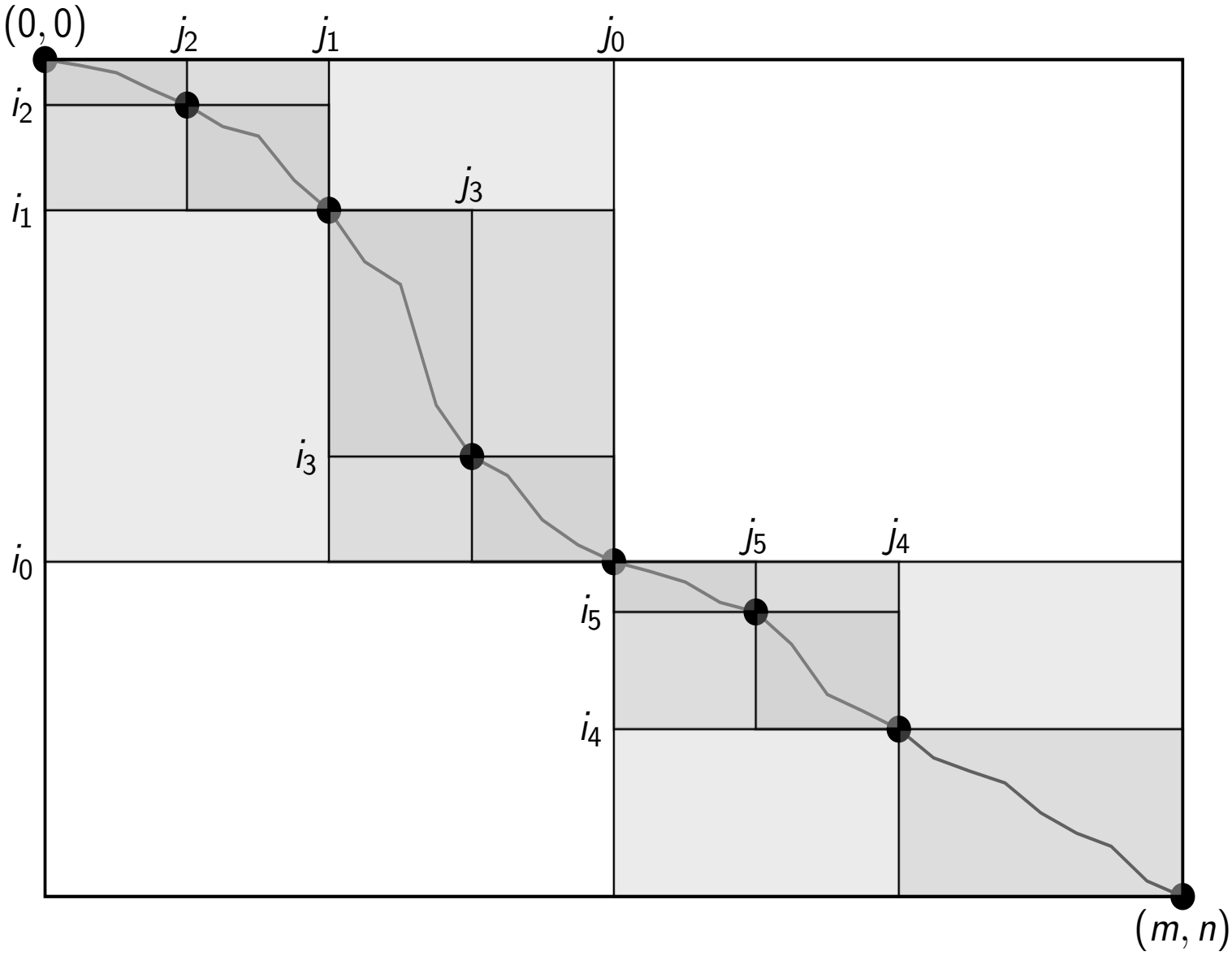


Figure 1: The divide and conquer strategy of the linear space alignment algorithm

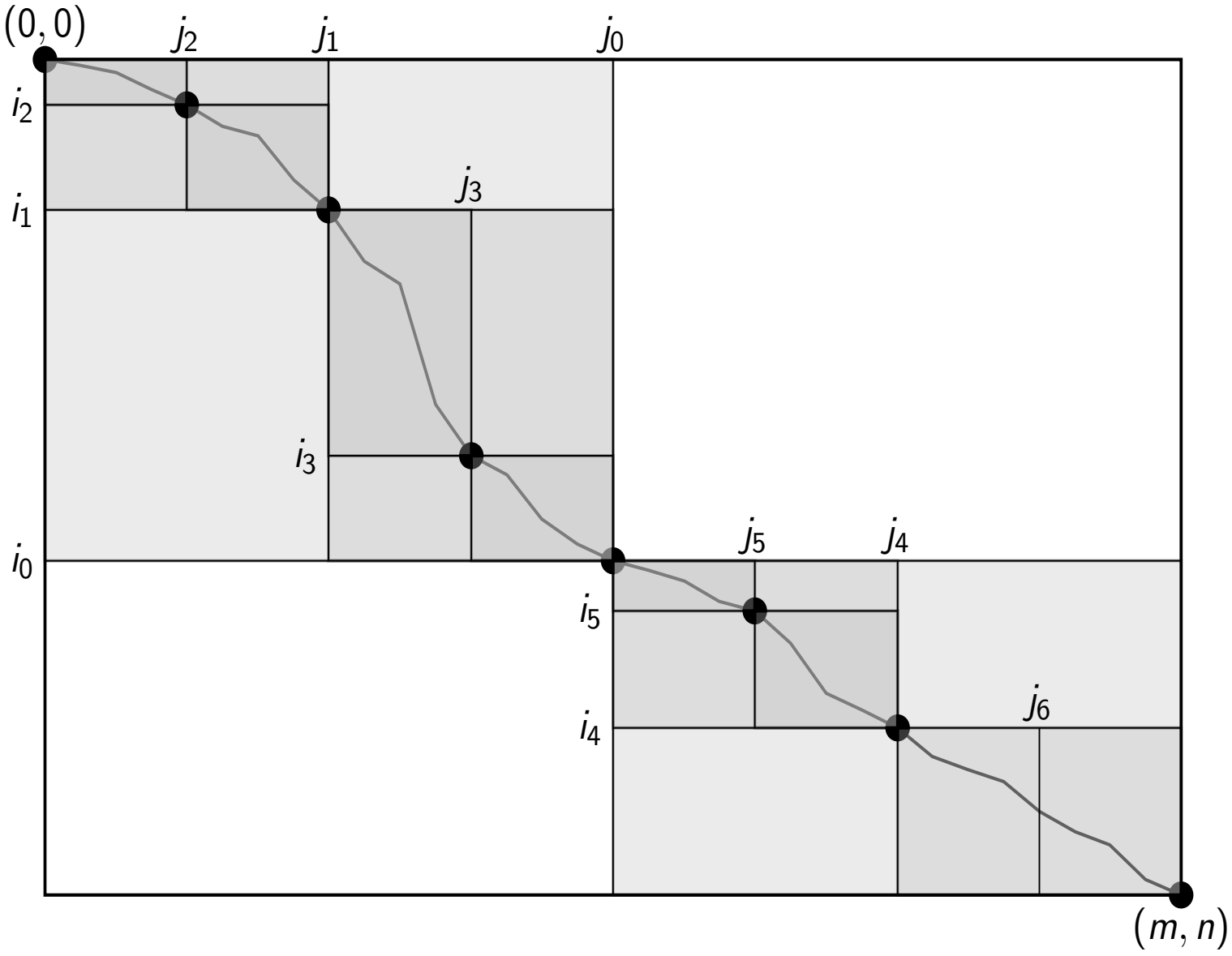


Figure 1: The divide and conquer strategy of the linear space alignment algorithm

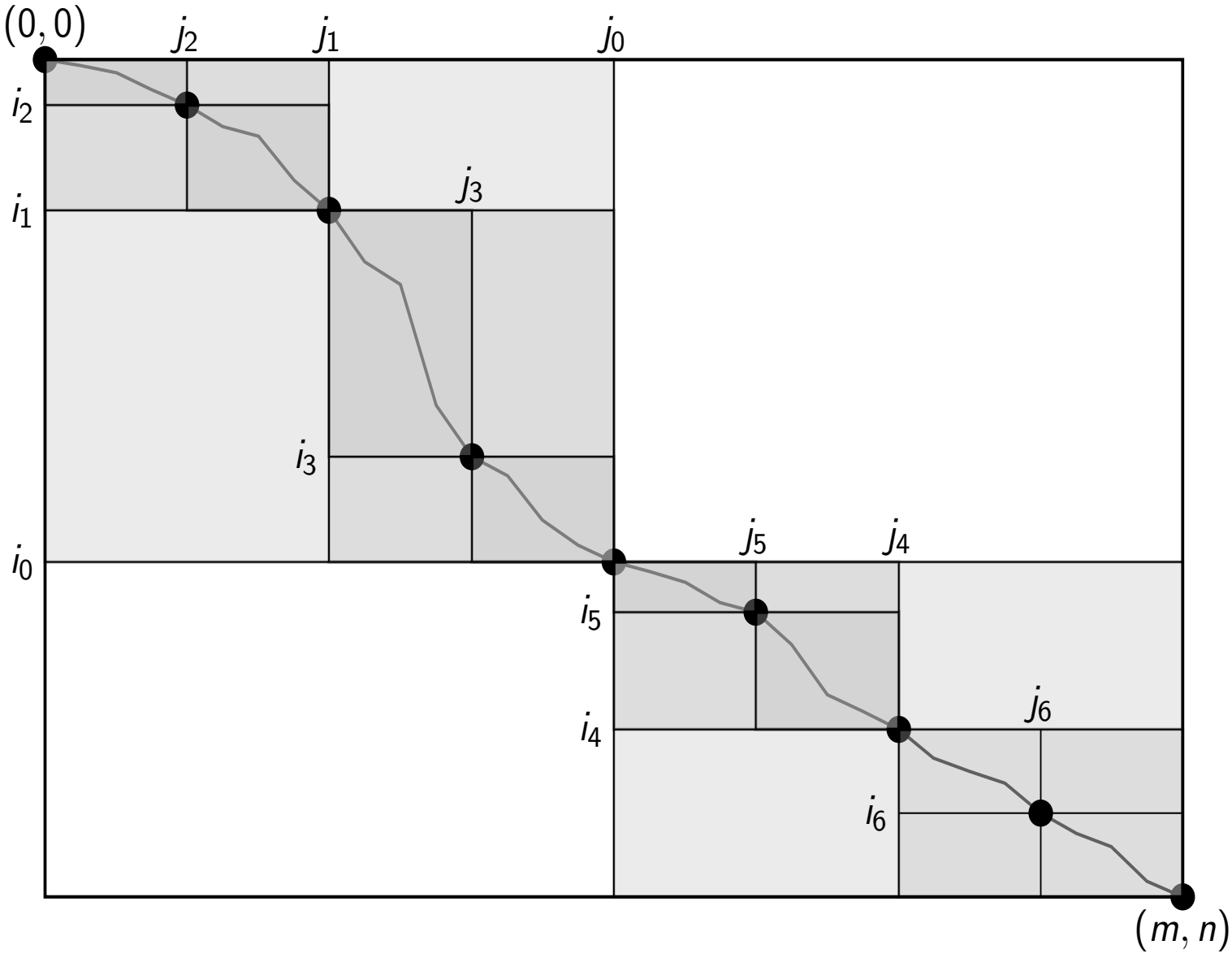


Figure 1: The divide and conquer strategy of the linear space alignment algorithm

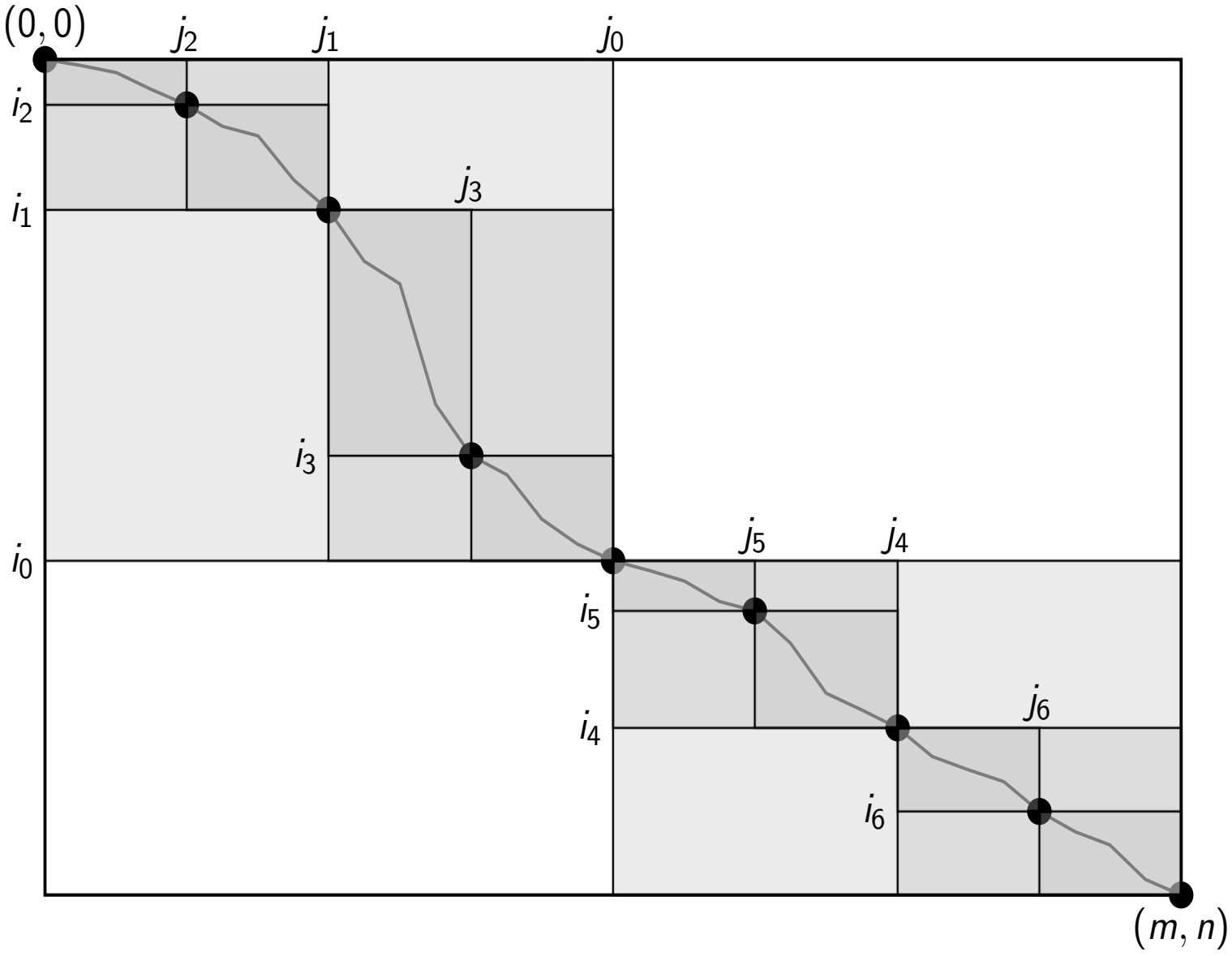
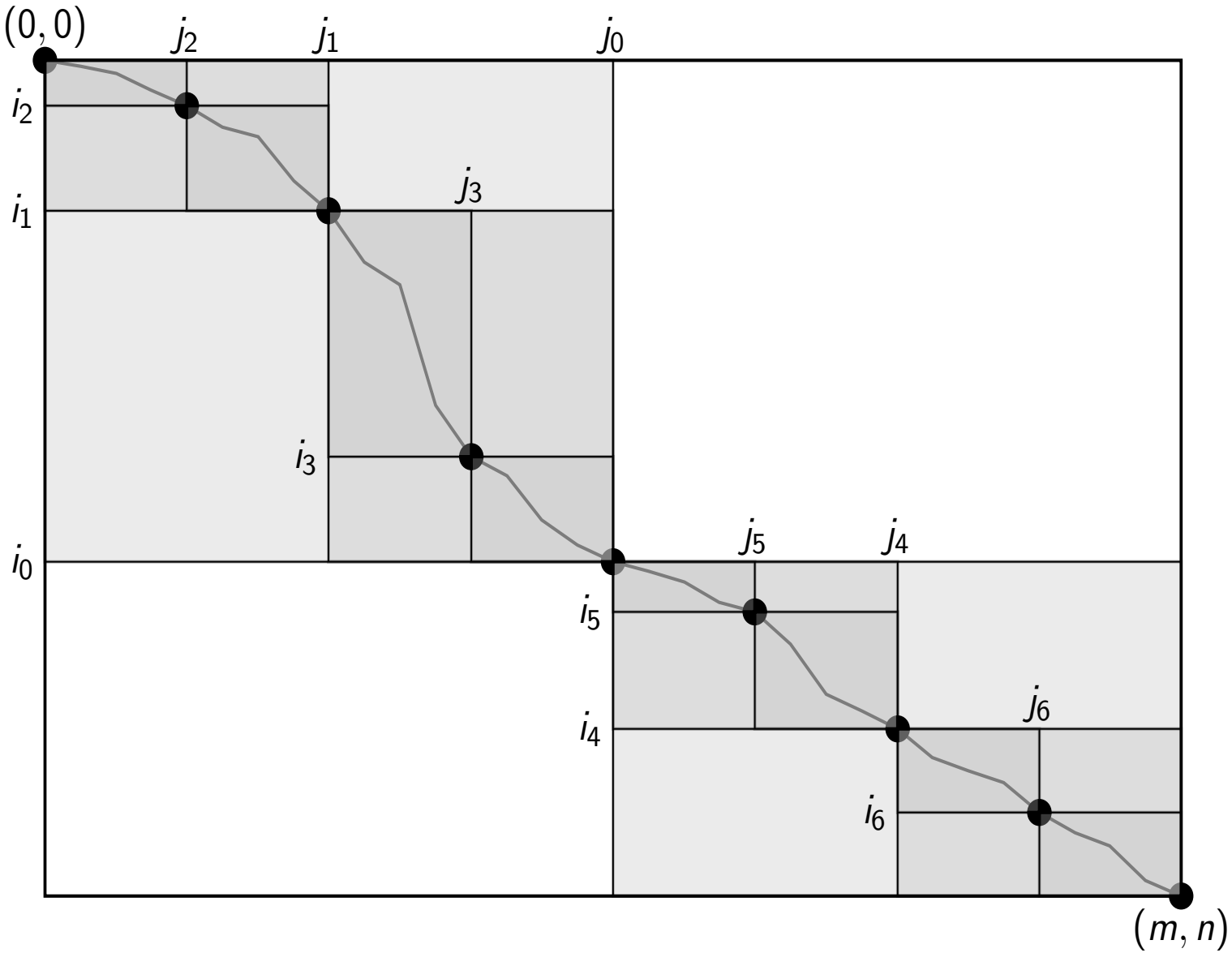


Figure 1: The divide and conquer strategy of the linear space alignment algorithm



# Applying the divide and concquer strategy

- We say that a node $(\_, j_0)$ in the edit-graph $G_\delta(u, v)$ is on the middle of a path from $(0, 0)$ to $(m, n)$ if and only if $j_0 = \lfloor n/2 \rfloor$.
- The divide-and-conquer strategy then needs to find an index $i_0$, $0 \leq i_0 \leq m$ such that $(i_0, j_0)$ is on a minimizing path from $(0, 0)$ to $(m, n)$.
- We obtain $i_0$ by computing an additional $(m + 1) \times (n + 1)$-table $R_{j_0}$ storing row indices.
- The columns of this table are referred to as *row index columns*.
- For each $i, j$, $0 \leq i \leq m$, $0 \leq j \leq j_0 - 1$, $R_{j_0}(i, j)$ is undefined.
- For each $j$, $j_0 \leq j \leq n$ and each $i$, $0 \leq i \leq m$, $R_{j_0}(i, j)$ is defined and the following property holds: if $R_{j_0}(i, j) = i_0$, then there is a minimizing path from node $(i_0, j_0)$ to node $(i, j)$.
- Note that $R_{j_0}(i, j)$ is not uniquely determined, as there can be more than one node in column $j_0$ from which the minimizing path ending in $(i, j)$ starts.

Figure 2: Tables $E_\delta$ and $R_{j_0}$ for sequences $u =$ bcacd, $v =$ dbadad, unit cost function, and $j_0 = \left\lfloor \frac{|v|}{2} \right\rfloor = \lfloor \frac{6}{2} \rfloor = 3$. The two matrices are superimposed into one matrix of pairs $(e, \mathbf{r})$, where $e$ is the distance value and $\mathbf{r}$ is the row index value. Undefined row index values are shown as $\perp$. The first row and column shows the sequence $v$ and $u$, respectively. The second row and second column shows the column indices and row indices, respectively. As $i_0 = R_{j_0}(m, n) = 3$, $(i_0, j_0) = (3, 3)$ (shown in blue) is a crosspoint.

| | | | | | $j_0$ | | | |
|---|---|---|---|---|---|---|---|---|
| | | | d | b | a | d | a | d |
| | $i/j$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| | 0 | (0,$\perp$) | (1,$\perp$) | (2,$\perp$) | (3,**0**) | (4,**0**) | (5,**0**) | (6,**0**) |
| b | 1 | (1,$\perp$) | (1,$\perp$) | (1,$\perp$) | (2,**1**) | (3,**1**) | (4,**1**) | (5,**1**) |
| c | 2 | (2,$\perp$) | (2,$\perp$) | (2,$\perp$) | (2,**2**) | (3,**2**) | (4,**2**) | (5,**2**) |
| a | 3 | (3,$\perp$) | (3,$\perp$) | (3,$\perp$) | (2,**3**) | (3,**3**) | (3,**2**) | (4,**2**) |
| c | 4 | (4,$\perp$) | (4,$\perp$) | (4,$\perp$) | (3,**4**) | (3,**3**) | (4,**3**) | (4,**2**) |
| d | 5 | (5,$\perp$) | (4,$\perp$) | (5,$\perp$) | (4,**5**) | (3,**4**) | (4,**4**) | (4,**3**) |

$R_{j_0}$ can be computed using the recurrence of Figure 3.

Figure 3: The recurrence for table $R_{j_0}$. The values depend on column $E_{\text{col}}^{j-1}$ and $E_{\text{col}}^{j}$. Line 3 handles the case where the insertion edge is minimizing. Line 4 handles the case where the replacement edge is minimizing. Line 5 handles the case where the deletion edge is minimizing. If $j > j_0$, then at least one of the three cases applies, so the last *else if* could be replaced by *otherwise*. As usual, with a distance value $E_{\text{col}}^{j}(i)$ we can store which of the incoming edges is minimizing, as to remove the dependency on other entries in $E_{\text{col}}^{j-1}$ and $E_{\text{col}}^{j}$.

$$R_{j_0}(i,j) = \begin{cases} \textit{undefined} & \text{if } 0 \le j \le j_0 - 1 \\ i & \text{else if } j = j_0 \\ R_{j_0}(i, j-1) & \text{else if } E_{\text{col}}^{j}(i) = E_{\text{col}}^{j-1}(i) + \delta(\varepsilon \to v[j]) \\ R_{j_0}(i-1, j-1) & \text{else if } E_{\text{col}}^{j}(i) = E_{\text{col}}^{j-1}(i-1) + \delta(u[i] \to v[j]) \\ R_{j_0}(i-1, j) & \text{else if } E_{\text{col}}^{j}(i) = E_{\text{col}}^{j}(i-1) + \delta(u[i] \to \varepsilon) \end{cases}$$

# Applying the divide and concquer strategy

- The computation of $R_{j_0}(i,j)$ for all $j$, $j_0 \leq j \leq n$ follows the same scheme as the computation of table $E_\delta$.
- In particular, one computes the $j$th column

$$R^j_{j_0} = R_{j_0}(0,j), \ldots, R_{j_0}(m,j)$$

of $R_{j_0}$ along with $E^j_{\text{col}}$.

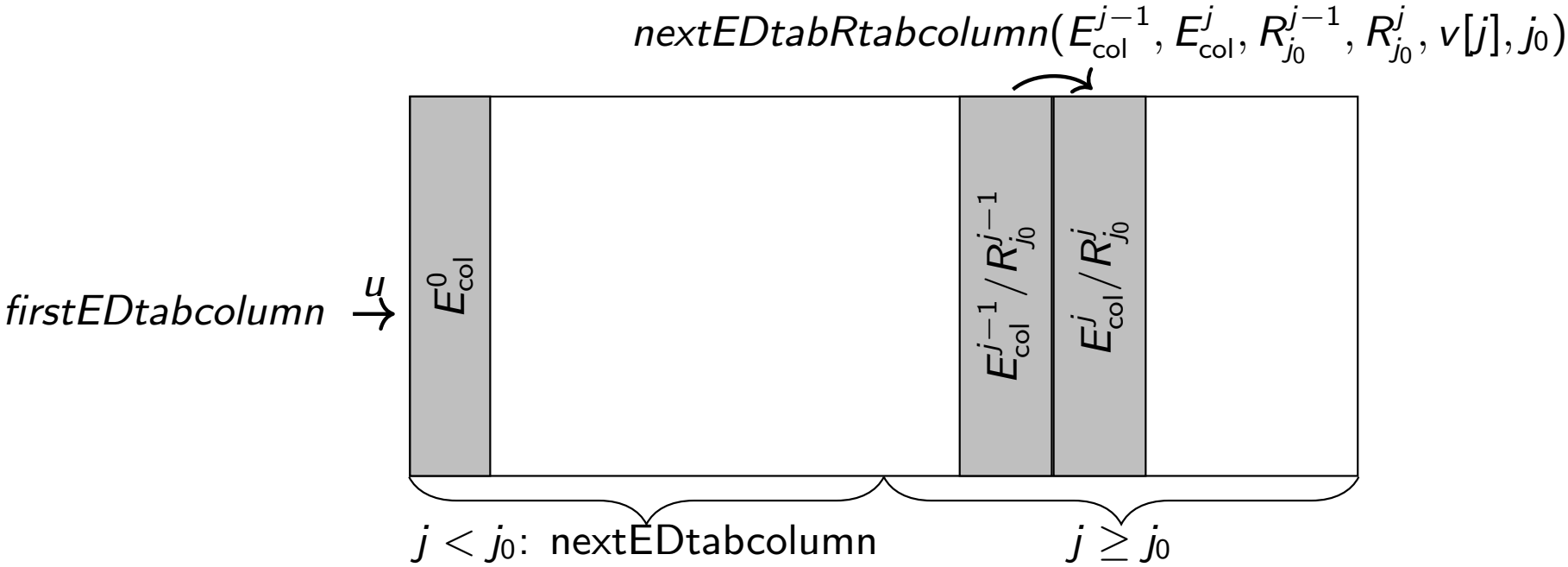
So reformulating the recurrence of Figure 3 in terms of columns we obtain:

$$R^j_{j_0}(i) = \begin{cases} \textit{undefined} & \text{if } 0 \leq j \leq j_0 - 1 \\ i & \text{else if } j = j_0 \\ R^{j-1}_{j_0}(i) & \text{else if } E^j_{\text{col}}(i) = E^{j-1}_{\text{col}}(i) + \delta(\varepsilon \to v[j]) \\ R^{j-1}_{j_0}(i-1) & \text{else if } E^j_{\text{col}}(i) = E^{j-1}_{\text{col}}(i-1) + \delta(u[i] \to v[j]) \\ R^j_{j_0}(i-1) & \text{else if } E^j_{\text{col}}(i) = E^j_{\text{col}}(i-1) + \delta(u[i] \to \varepsilon) \end{cases}$$

# Applying the divide and concquer strategy

- With the same argumentation as above, one shows that the computation of the $R_{j_0}$-columns only requires one $(m+1)$-element vector and two additional scalar values.
- It is not difficult to derive a function *nextEDtabRtabcolumn* (in analogy to *nextEDtabcolumn*) performing this computation in the right halve of the matrix.
- This function computes $E_{\mathrm{col}}^{j}$ and $R_{j_0}^{j}$ from $E_{\mathrm{col}}^{j-1}$ and $R_{j_0}^{j-1}$ given $u$, $v$, $j$, and $j_0$.
- As $R_{j_0}^{0}$ is undefined in the left halve of the matrix, for $j < j_0$ one only needs to compute $E_{\mathrm{col}}^{j}$ using the previously explained functions *firstEDtabcolumn* and *nextEDtabcolumn*, see the following illustration.

# Combining the different steps




- Combining these three functions one obtains a function *evaluateallcolumns*.
- This computes the final pair of columns of tables $E_\delta$ and $R_{j_0}$, given $u$, $v$, and a middle column index $j_0$.

# Combining the different steps

- Suppose that we have determined $E^n_{\mathrm{col}}$ and $R^n_{j_0}$ as described above, where $j_0 = \lfloor n/2 \rfloor$.
- Then we know that $edist_\delta(u, v) = E^n_{\mathrm{col}}(m)$ and $(i_0, j_0)$, where $i_0 = R^n_{j_0}(m)$, is on a minimizing path from $(0, 0)$ to $(m, n)$.
- We store $i_0$ at index $j_0$ of a table $C$ which is indexed from 0 to $n$.
- $C$ is the table of *crosspoints*.
- Each step of the linear space algorithm computes some value in table $C$ by evaluating a distance column and a row index column from the given substrings of $u$ and $v$ to align.
- Table $C$ stores the crosspoints encoding an optimal alignment.
- In particular, if $C(j) = i$, then $(i, j)$ is on a minimizing path from $(0, 0)$ to $(m, n)$.
- Note that $C(0) = 0$ and $C(n) = m$ as $(0, 0)$ and $(m, n)$ are the initial and final crosspoints.

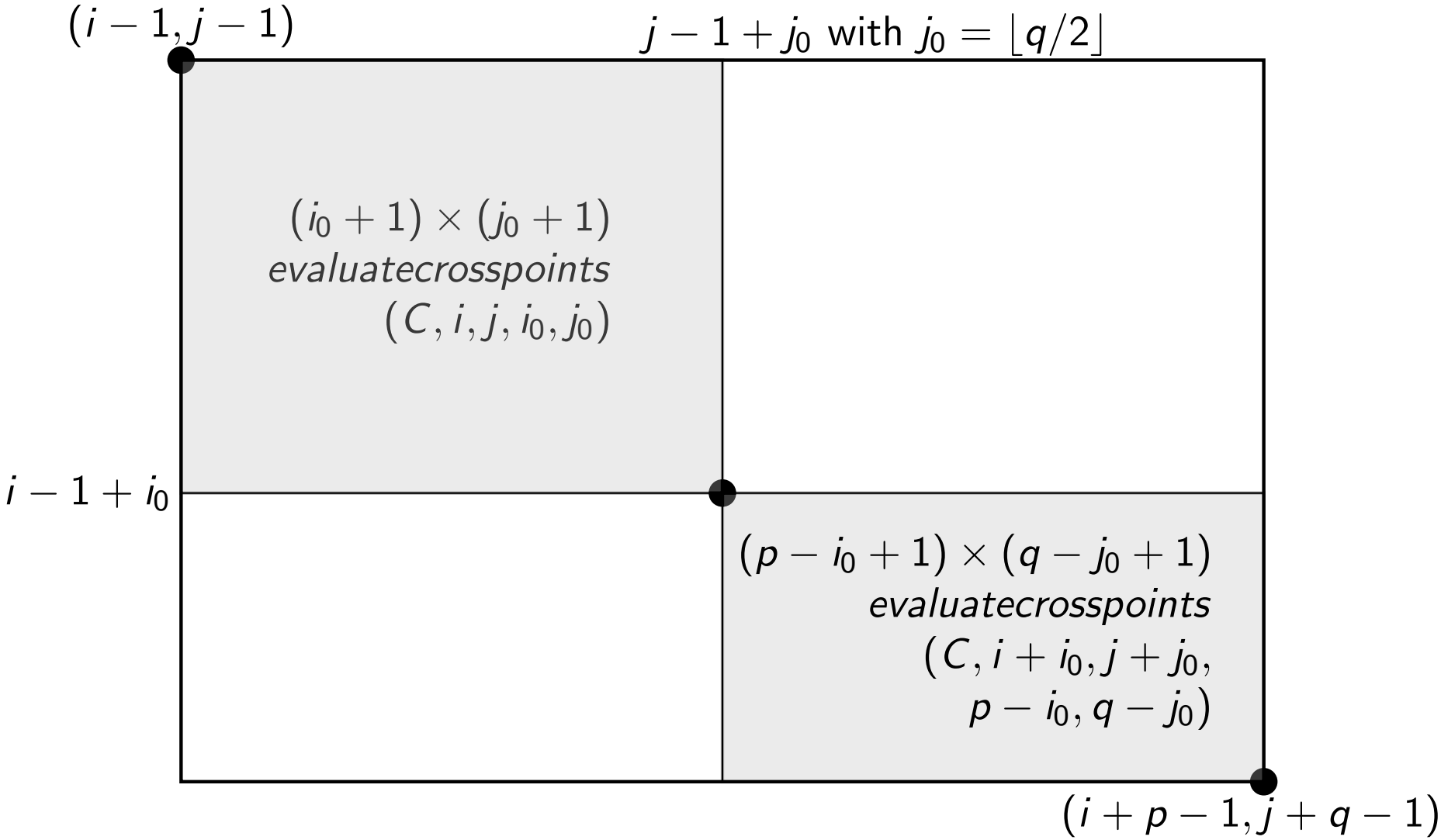
# Combining the different steps

- We now have to consider how to combine the computation of a single recursion step (as described above) into a function implementing the divide-step.
- For this apply the following generalization: suppose, that the two substrings of $u$ and $v$ to align, start at positions $i, j$, $1 \leq i \leq m$ and $1 \leq j \leq n$ and they are of length $p$ and $q$, respectively.
- That is, the algorithm must compute a $(p+1) \times (q+1)$-table.
- Algorithm 2 gives a recursive function *evaluatecrosspoints*, which employs the divide-and-conquer strategy to compute the crosspoints.
- The initial call is *evaluatecrosspoints*$(C, 1, 1, m, n)$ as we want to compute the crosspoints for an alignment of $u = u[1 \ldots m]$ and $v = v[1 \ldots n]$.

# Combining the different steps

- In the algorithm, *evaluateallcolumns* computes the distance and row index columns for the substrings $u[i \ldots i+p-1]$ and $v[j \ldots j+q-1]$ given $j_0$.
- The latter value, i.e. $j_0$, is relative to the start index $j-1$, i.e. the implicit DP-table is divided into two halves at column $j-1+j_0$, see Figure 4.
- To store the columns, *evaluateallcolumns* uses the same two vectors $E$ and $R$, both of length $m+1$.

Figure 4: To compute *evaluatecrosspoints*$(C, i, j, p, q)$, the implicit DP-table is divided into two halves at column $j-1+j_0$ and row index $i-1+i_0$ is determined.



## Algorithm 2 (Recursive function to compute crosspoints)

**Input**: $i$, $j$, $p$, $q$ with $1 \leq i \leq m$, $1 \leq j \leq n$, $0 \leq p \leq m - i + 1$, $0 \leq q \leq n - j + 1$, and uninitialized $(m+1)$-element vectors $E$ and $R$ to store distance and row index columns

**Output**: filled table $C$ of $n+1$ crosspoints

1: **function** *evaluatecrosspoints*$(C, i, j, p, q)$
2: **if** $q \geq 2$ **then**
3: $j_0 \leftarrow \lfloor q/2 \rfloor$
4: *evaluateallcolumns*$(E, R, j_0, u[i \ldots i+p-1], v[j \ldots j+q-1])$
5: $i_0 \leftarrow R[p]$ ▷ last value of last $R_{j_0}$-column
6: $C(j-1+j_0) \leftarrow i-1+i_0$ ▷ $i/j$ refer to strings $\Rightarrow$ subtract 1
7: *evaluatecrosspoints*$(C, i, j, i_0, j_0)$ ▷ eval. upper/left part
8: *evaluatecrosspoints*$(C, i+i_0, j+j_0, p-i_0, q-j_0)$ ▷ low./right
9: **end if**
10: **end function**
11: create table $C$ of $n+1$ entries and set $C(0) \leftarrow 0$ and $C(n) \leftarrow m$
12: *evaluatecrosspoints*$(C, 1, 1, m, n)$

# Combining the different steps

- Note that a crosspoint is not computed if $n < 2$.
- In this case, $v$ is of length 0 or 1 and it is easy to compute an optimal alignment of $u$ and $v$ in linear space.
- Note that all recursive calls of *evaluatecrosspoints* use the same globally-declared vectors $E$ and $R$ to compute the distance and row index columns, and in turn the value $i_0$.
- This is correct, since once $i_0$ is determined, the values stored in vectors $E$ and $R$ are no longer needed.
- So they can be used in the two recursive calls of line 7 and 8 of Algorithm 2.
- This is essential to guarantee that the function only consumes $O(m)$ space.

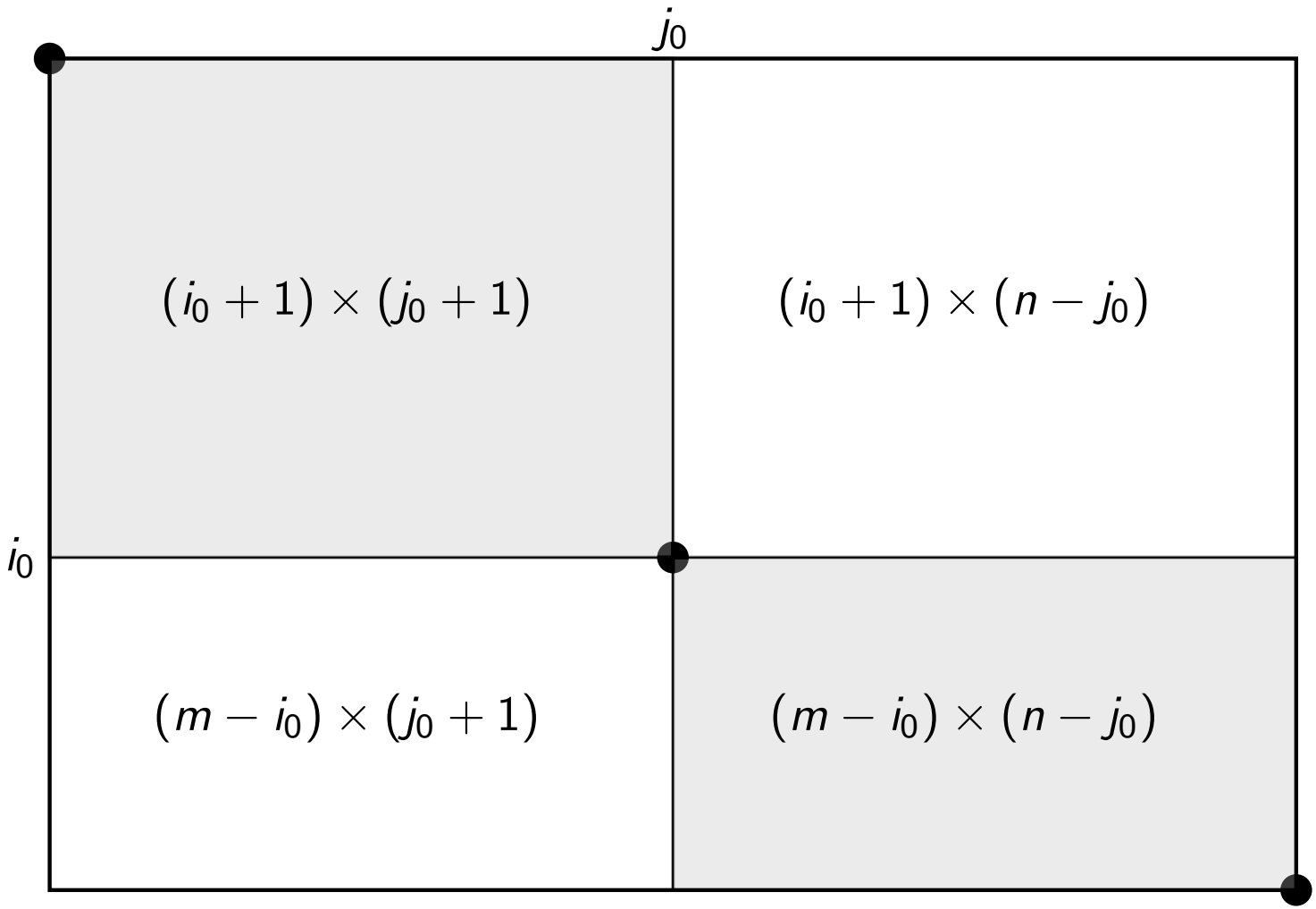
# Analysis of the space requirement

- We now analyze the space requirement of the presented algorithm.
- We previously had pointed out that all calls to the function *evaluatecrosspoints* use the same two $(m+1)$-element vectors.
- Additionally, $O(n)$ space for table $C$ is needed.
- Apart from that, only a constant number of local variables in each call is required.
- Since the maximal depth of the recursion is $\log_2 n$, the space requirement for the variable of the call stack is $O(\log_2 n$.
- We conclude that the total space requirement is $O(m + n + \log n) = O(m + n)$.

# Analysis of the space requirement

- For sequences of length $m$ and $n$, respectively, the function *evaluateallcolumns* evaluates $(m+1)(n+1)$ values in matrix $E_\delta$ and $\frac{1}{2}(m+1)(n+1)$ values in $R_{j_0}$, summing up to $\frac{3}{2}(m+1)(n+1)$ values.
- After determining a crosspoint $(i_0, j_0)$, two further matrices are evaluated: a $(i_0+1) \times (j_0+1)$-matrix (upper left) and a $(m-i_0) \times (n-j_0)$-matrix (lower right).
- As can be seen in the following illustration (consider the size of the areas reflecting the number of values), the sum of the number of values in these two matrices is about halve of the original size.

# Analysis of the space requirement



# Analysis of the space requirement

So the number of matrix values to compute is $\frac{1}{2}\frac{3}{2}(m+1)(n+1)$.
Let $p = \frac{3}{2}(m+1)$ and $q = n+1$. The algorithm computes the following number of values over the whole computation:

$$pq + \frac{1}{2}pq + \frac{1}{4}pq + \ldots\ldots = \sum_{i=0}^{\log_2 n} \frac{1}{2^i} pq$$

$$= \underbrace{\left( \sum_{i=0}^{\log_2 n} \frac{1}{2^i} \right)}_{\substack{\text{geometric} \\ \text{series} \leq 2}} pq \leq 2pq = 3(m+1)(n+1)$$

# Analysis of the space requirement

- Thus the linear space algorithm does not compute more than three times as much values as the quadratic space algorithm.
- Since each value is computed in constant time, the running time remains in $O(mn)$.
- Thus the improvement to linear space does not lead to an increase of the (asymptotic) running time.
- Empirical experience shows that an implementation of the linear space algorithm takes about twice as much time as the basic algorithm, thus even less than determined from the calculation above.


Myers, G. (1998).
The Algorithmic Foundations of Molecular Computational Biology.
Unpublished Manuscript.


Powell, D., Allison, L., and Dix, T. (1999).
A Versatile Divide and Conquer Technique for Optimal String Alignment.
*IPL*, 70:127–139.