

# Extending the Core Problems

13.1. Parametric Sequence Alignment

**13.2. Computing Suboptimal Alignments**

**13.3. Chaining Diverse Local Alignments**

2016.06.07

박종훈

# Where are we?

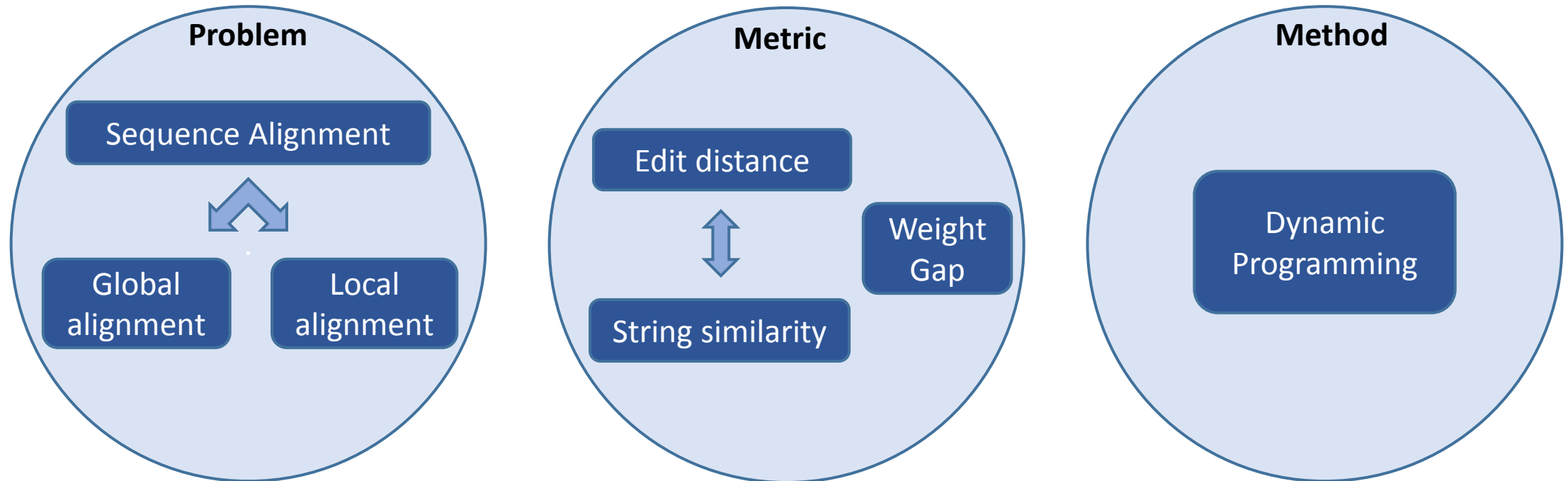
---

- I. Exact String Matching
- II. Suffix Trees and Their Uses
- III. Inexact Matching, Sequence Alignment, Dynamic Programming
  - 11. Core String Edits, Alignments, and Dynamic Programming
  - 12. Refining Core String Edits and Alignments
  - 13. Extending the Core Problems**

# Summary so far



<b>S<sub>1</sub></b>	q	q	c	-	d	b	d
<b>S<sub>2</sub></b>	q	a	w	x	-	b	-



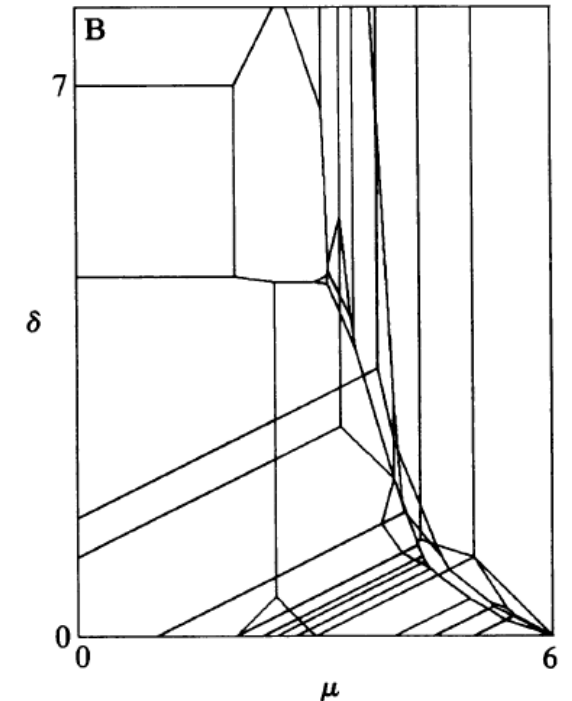
# Summary so far

---

- 12. Refining Core String Edits and Alignments
  - Methods to make the core algorithms better.
    - Time
    - Space
- How to reduce (time or space) complexity of the algorithms.

# 13. Extending the Core Problem

- Looking in detail at alignment problems.
  - Arising in computational molecular biology.
  - Extending the core alignment methods.
- 13.1 Parametric Sequence Alignment
  - Situation
    - Need to decide how to weight matches, mismatches, insertions and deletions, and gaps.
  - Problem
    - Disagreement about how to weight those.
  - Solution
    - Using a polygonal decomposition.



# 13.2 Computing suboptimal alignment

---

- Global or Local **optimal** alignment.
- Optimal alignment does not always identify the biological phenomena.
- Problems
  1. Objective functions might not reflect the full range of biological forces that cause differences between strings
  2. Data might contain errors that confound the algorithms
  3. There may be ties for the optimal alignment
  4. There may be many nearly optimal alignments that are biologically more significant than any optimal one.

# 13.2 Computing suboptimal alignment

---

- Solutions

1. Just accept that optimal alignment is only a crude reflection of biological significance.
2. **Finding suboptimal alignments with manageable size. (better than random alignments)**

Optimal set

Suboptimal set 1

Suboptimal set 2

Suboptimal set 3

Suboptimal set 4

# Near-optimal alignments

---

- Finding near-optimal alignments.
- How?
  - Alignment graph
  - Reweighting



# Alignment Graph

---

	-	A	N	N
-	0	-1	-2	-3
C	-1	-1	-2	-3
A	-2	1	0	-1
N	-3	0	3	2

# Alignment Graph

Graph =  $\langle V, E \rangle$

**Vertex (Node):**

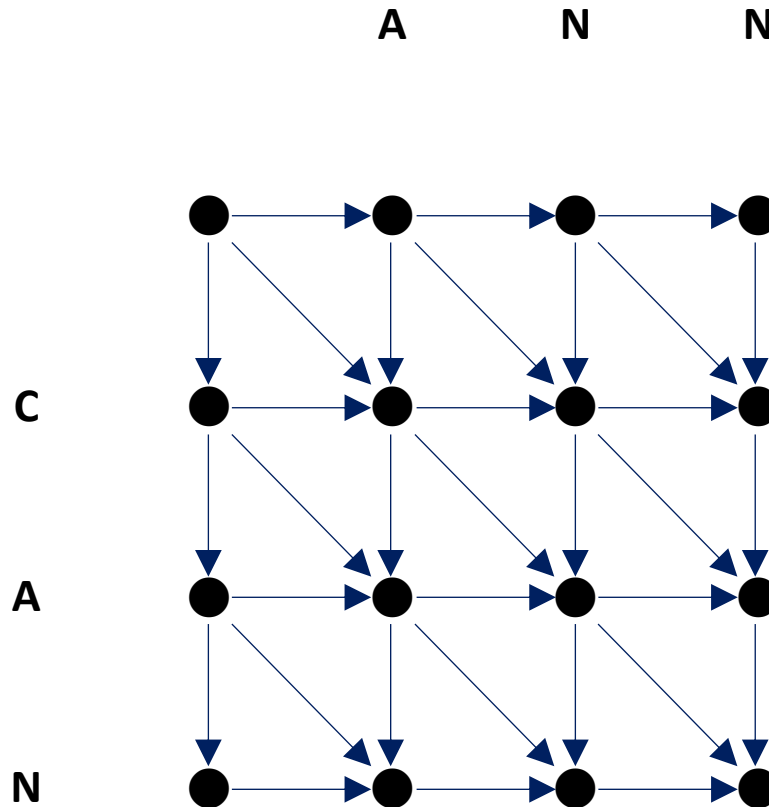
each cell of dynamic programming table

**Directed edge:**

From each node  $(i, j)$  to each of the nodes  $(i, j+1)$ ,  $(i+1, j)$ , and  $(i+1, j+1)$

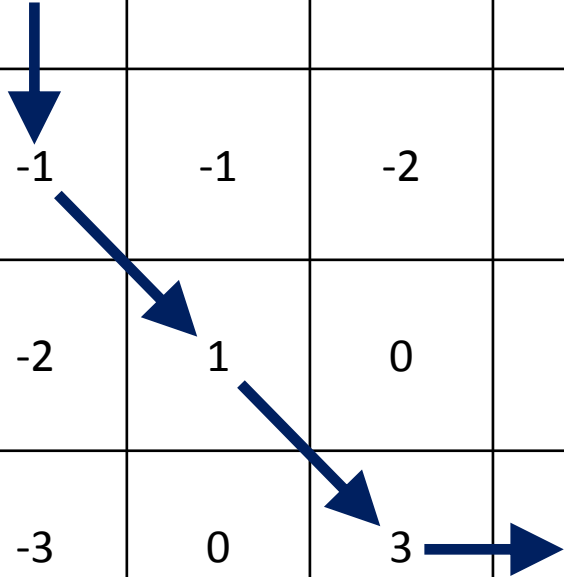
**Weight on the edges:**

the specific values for aligning a specific pair of characters or a character against a space.



# Optimal Path

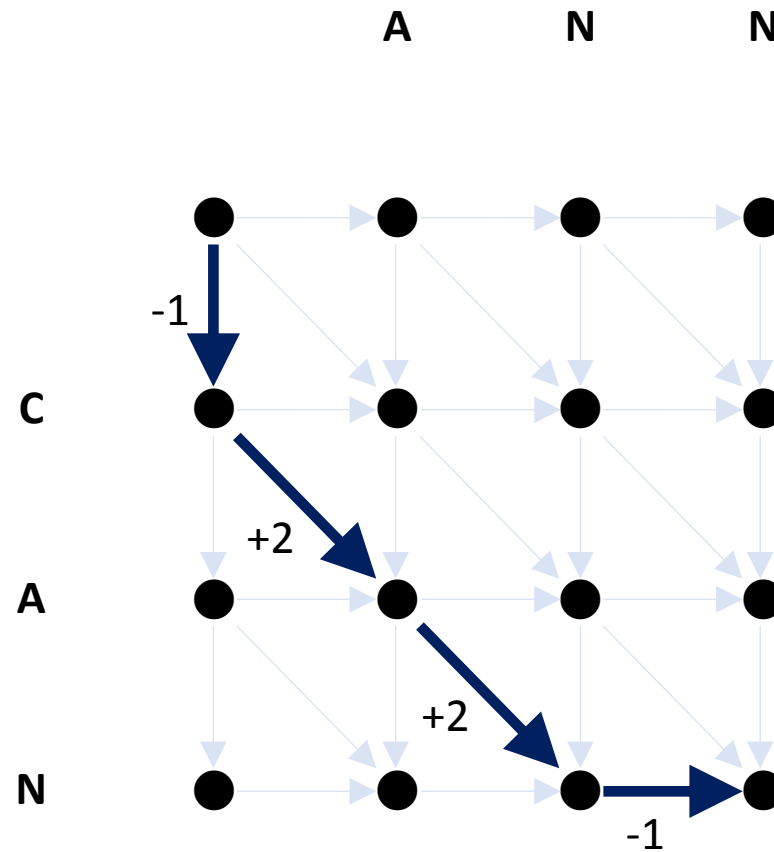
	-	A	N	N
-	0	-1	-2	-3
C	-1	-1	-2	-3
A	-2	1	0	-1
N	-3	0	3	2



# Maximum scoring path

Optimal Path

Maximum scoring path

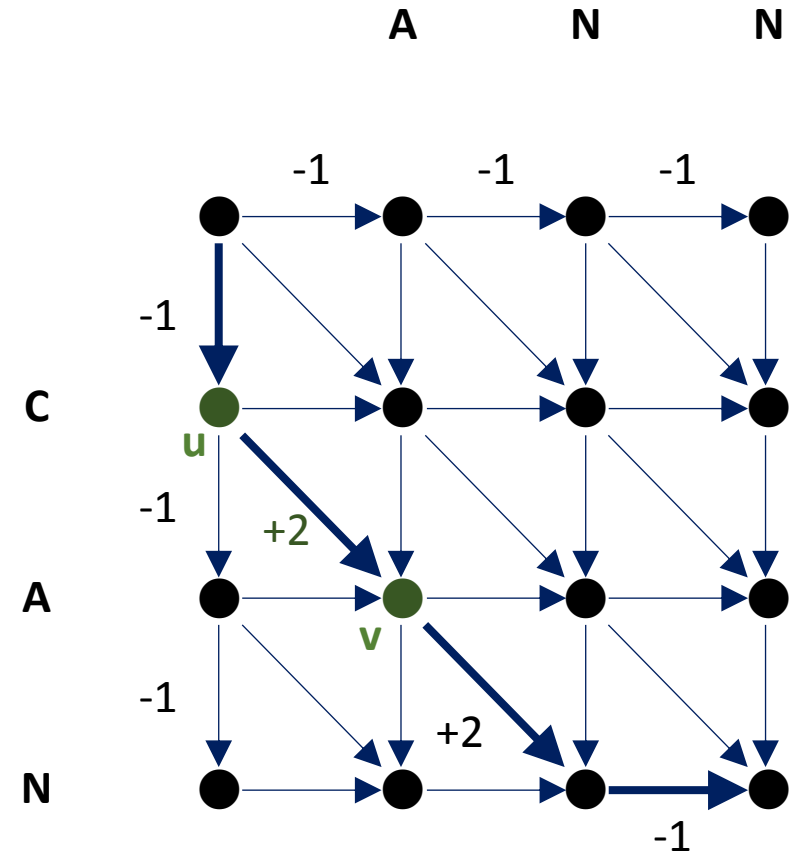


# Alignment Graph



## Definition

- $s(u, v)$ : the value of the edge  $(u, v)$  in the alignment graph.
- $V(S_1, S_2)$ : the value of the optimal alignment of string  $S_1$  and  $S_2$ .  
(maximum scoring path)



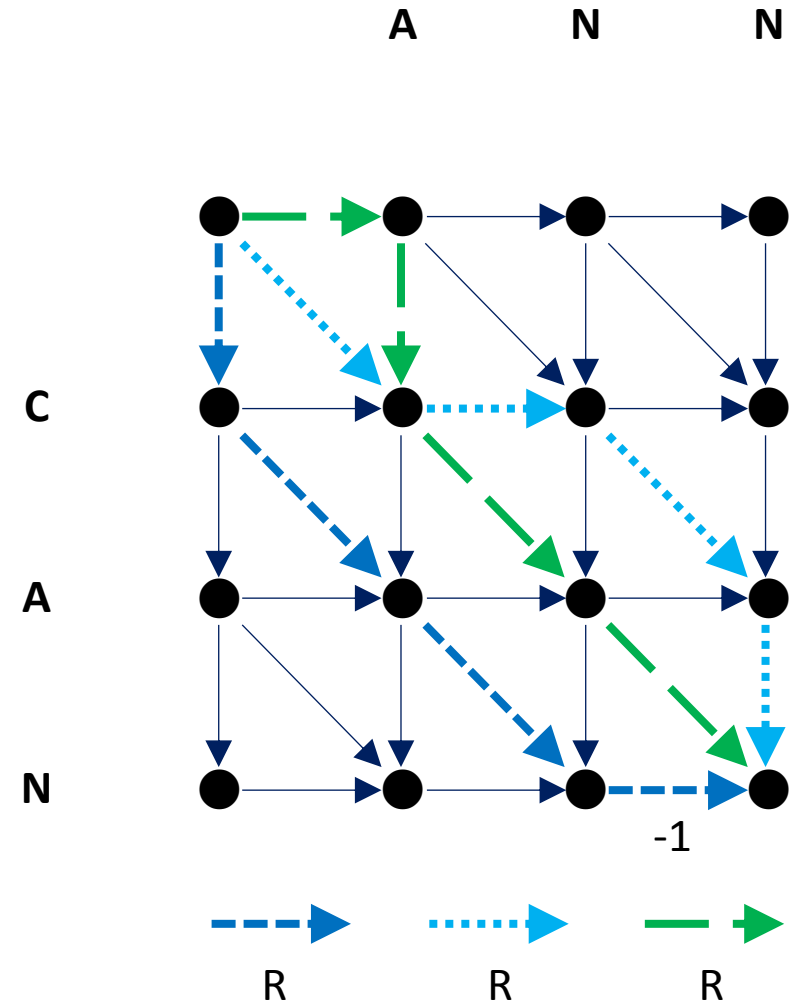
# Alignment Graph



## Definition

- Let  $\mathbf{R}$  be a path in the alignment graph from the start node  $\mathbf{s} \equiv (0, 0)$  to the destination node  $\mathbf{t} \equiv (n, m)$ .

Path  $\mathbf{R}$  corresponds to some global alignment (not necessarily optimal) of strings  $S_1$  and  $S_2$ .

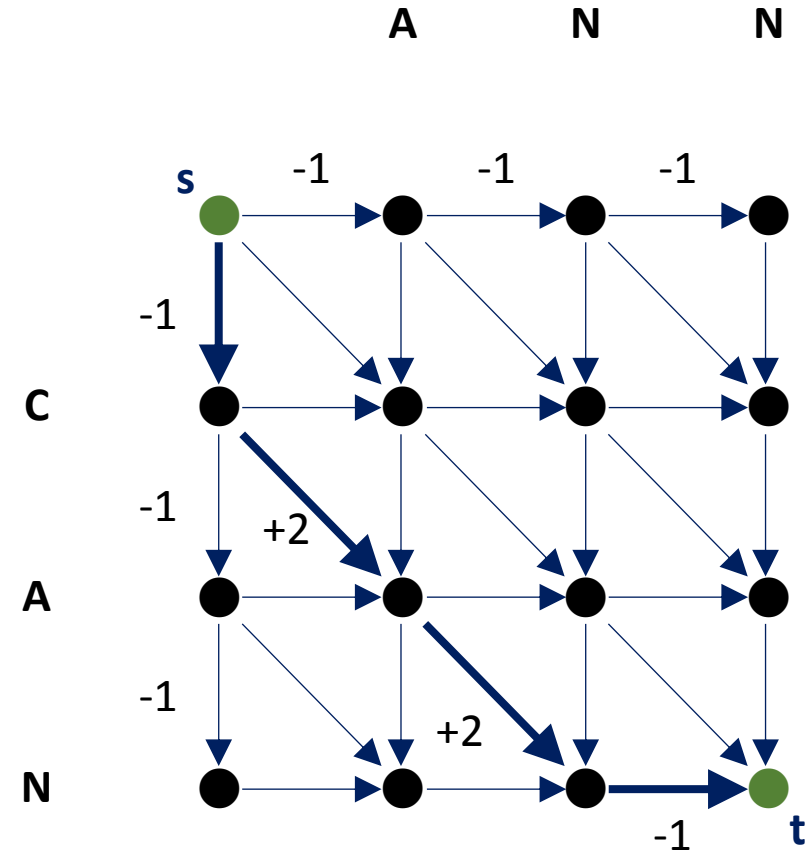


# Alignment Graph



## Definition

- For any pair of nodes  $x, y$  in the alignment graph, let  $l(x, y)$  be the length of the longest path from  $x$  to  $y$ .
- Let  $R^*$  be the path corresponding to an optimal global alignment.
- The length of  $R^*$  is equal to  $l(s, t) = V(S_1, S_2)$

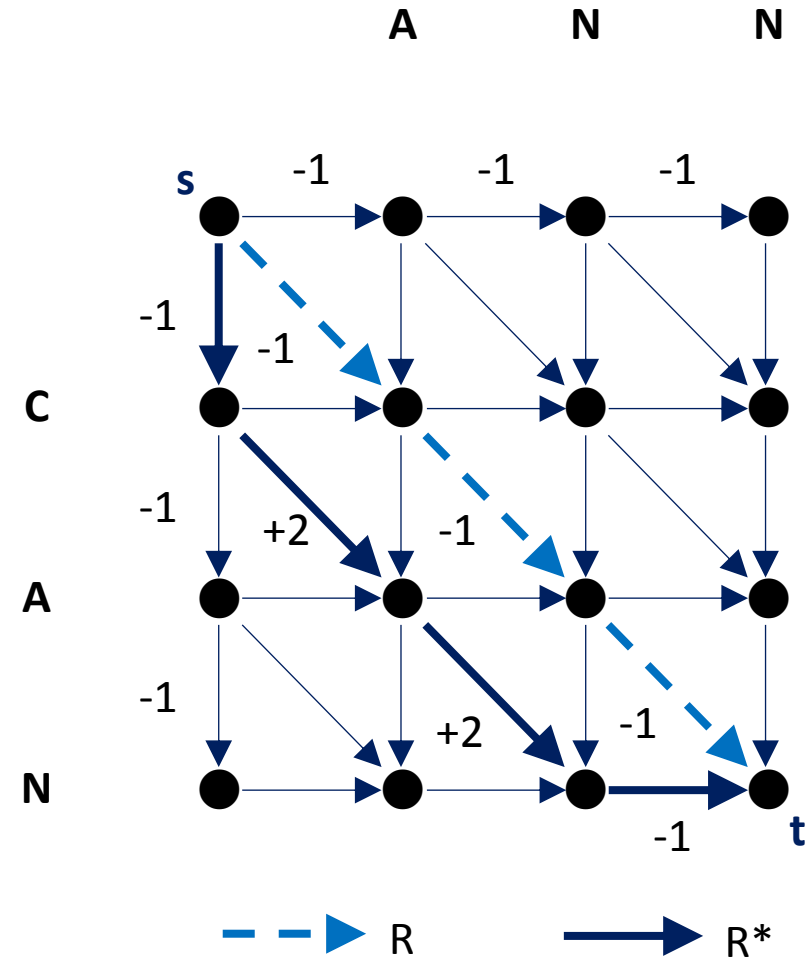


# Alignment Graph



## Definition

- For path  $R$ , let  $\delta(R)$  be the length of  $R^*$  minus the length of  $R$ .
- $\delta(R)$  is called the “deviation” of  $R$  (from the optimal), and  $R$  is called a  ***$\delta(R)$ -near-optimal-path***.



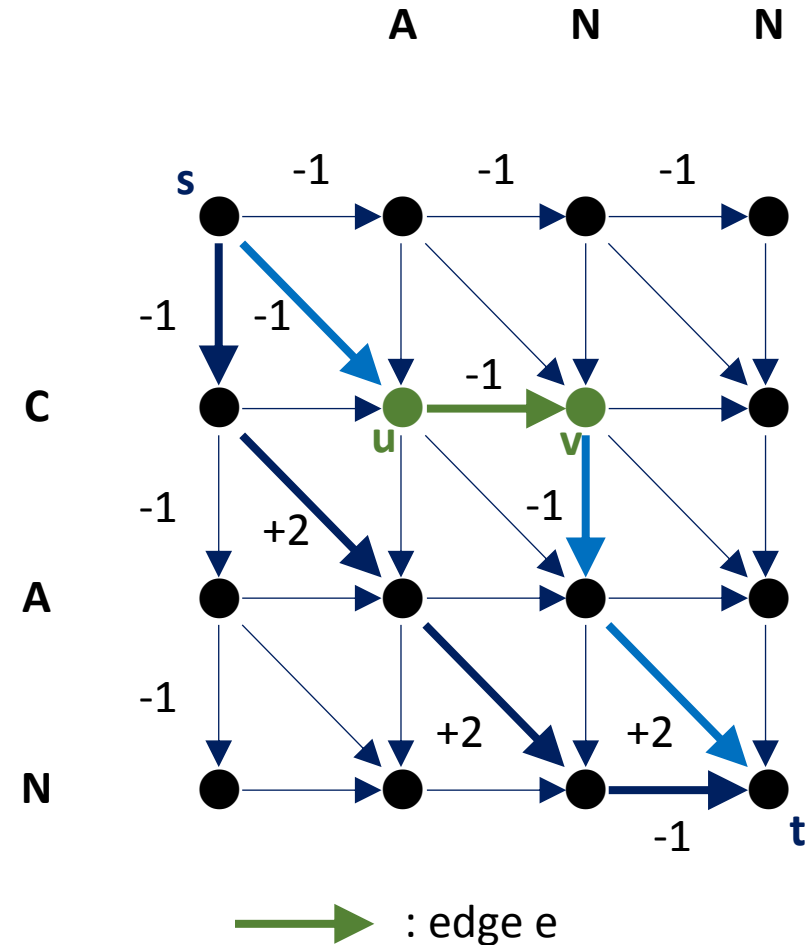


# Alignment Graph



## Definition

- For an **edge**  $e = (u, v)$ ,  
$$\delta(e) = l(s, t) - [l(s, u) + s(u, v) + l(v, t)].$$
- That is,  $\delta(e)$  is the difference between the length of  $R^*$  and the length of the longest  $s$ -to- $t$  path that is forced to go through edge  $e$ .



# Summary of Definitions



$s(u, v)$	The value of the edge $(u, v)$ in the alignment graph.
$V(S_1, S_2)$	The value of the optimal alignment of string $S_1$ and $S_2$ .
$R$	a path in the alignment graph from the start node $s \equiv (0, 0)$ to the destination node $t \equiv (n, m)$ .
$R^*$	the path corresponding to an optimal global alignment.
$l(x, y)$	the length of the longest path from $x$ to $y$ .
$\delta(R)$	the length of $R^*$ minus the length of $R$ ( $ R^*  -  R $ ).
$\delta(e)$	the difference between the length of $R^*$ and the length of the longest $s$ -to- $t$ path that is forced to go through edge $e$ .

# Lemma 13.2.1

---

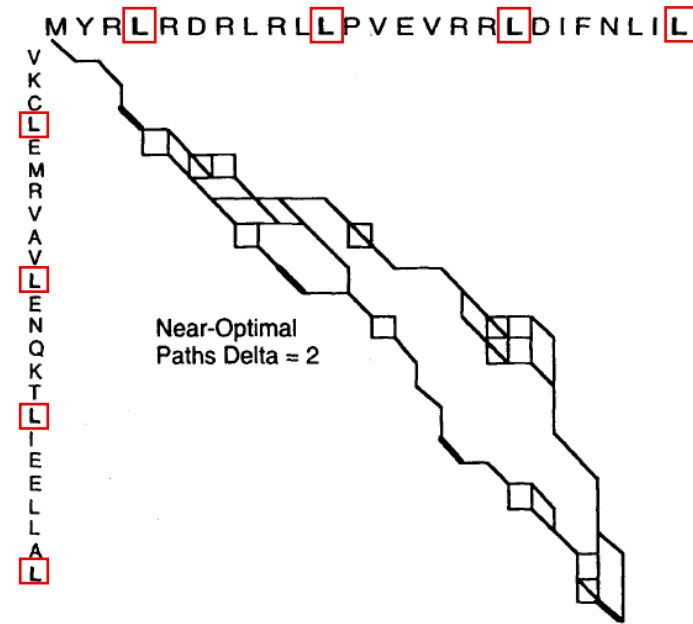
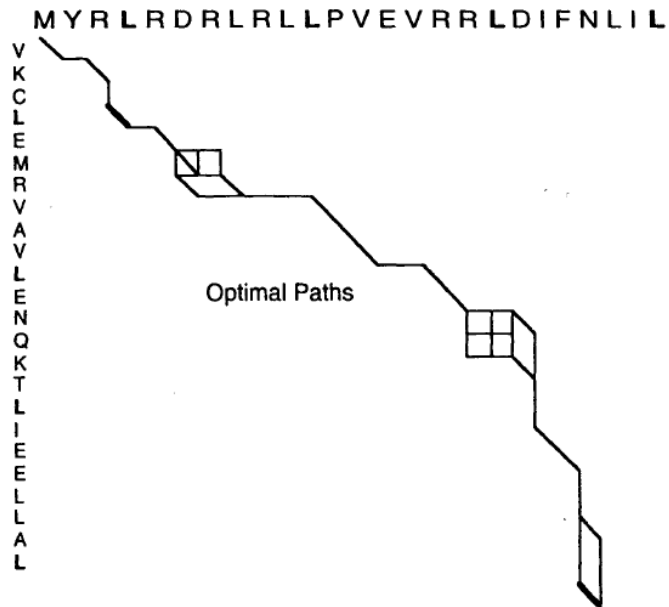
- Lemma 13.2.1
  - $\delta(e)$  can be computed for all edges in the time used to compute two optimal alignments plus time proportional to the number of edges.

- For an edge  $e = (u, v)$ ,  
$$\delta(e) = l(s, t) - [l(s, u) + s(u, v) + l(v, t)].$$



# $\Delta$ near-optimal alignments

- $\delta$ -near-optimal-path.



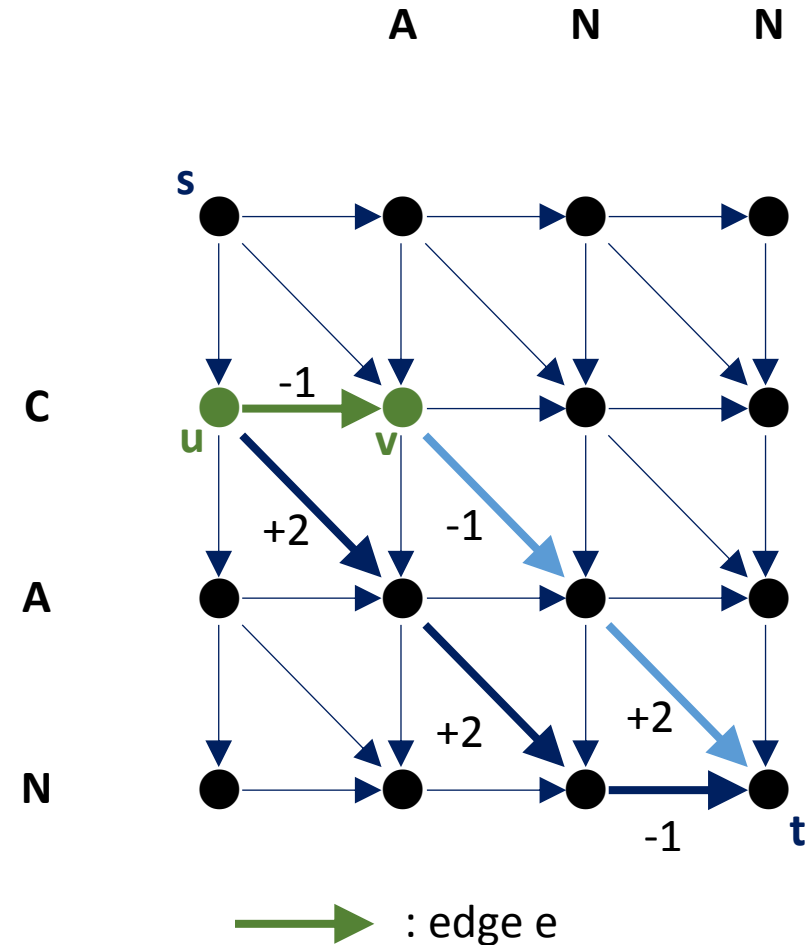
- How to efficiently **count** and **enumerate** this set of paths?
  - Reweighting!

# A useful reweighting



## Definition

- For an edge  $e = (u, v)$ ,  
let  $\epsilon(e) = l(u, t) - [s(u, v) + l(v, t)]$
- $\epsilon(e)$  is the **additional penalty for using  $e$  on the path from  $u$  to  $t$** , rather than following the optimal (longest) path from  $u$  to  $t$  directly.



# Theorem 13.2.1

---

- Theorem 13.2.1

- For any s-to-t path  $R$ ,  $\delta(R) = \sum_{e \in R} \epsilon(e)$

- For an edge  $e = (u, v)$ ,

**let  $\epsilon(e) = l(u, t) - [s(u, v) + l(v, t)]$**

additional penalty for using  $e$  on the path from  $u$  to  $t$



# Theorem 13.2.1

- Proof

Let  $R = v_0, v_1, \dots, v_{k-1}, v_k$ , where  $v_0 = s$  and  $v_k = t$ . Also, let  $|R|$  denote the length of  $R$ . Then

$$\begin{aligned}\sum_{e \in R} \epsilon(e) &= \sum_{i=0}^{k-1} \epsilon(v_i, v_{i+1}) \\ &= \sum_{i=0}^{k-1} [l(v_i, t) - s(v_i, v_{i+1}) - l(v_{i+1}, t)] \\ &= \sum_{i=0}^{k-1} [l(v_i, t) - l(v_{i+1}, t)] - \sum_{i=0}^{k-1} s(v_i, v_{i+1}) \\ &= \left( \sum_{i=1}^{k-1} [l(v_i, t) - l(v_i, t)] + l(v_0, t) - l(v_k, t) \right) - |R| \\ &= l(s, t) - |R| = |R^*| - |R| = \delta(R).\end{aligned}$$

# Corollary 13.2.1

---

- Corollary 13.2.1

- Consider a path  $R'$  from  $s$  to  $u$  and let  $\delta$  denote  $\sum_{e \in R'} \epsilon(e)$ .

Then the  $s$ -to- $t$  path  $R$  consisting of path  $R'$  followed by the longest  $u$ -to- $t$  path is a  $\delta$ -near-optimal path.

- Proof

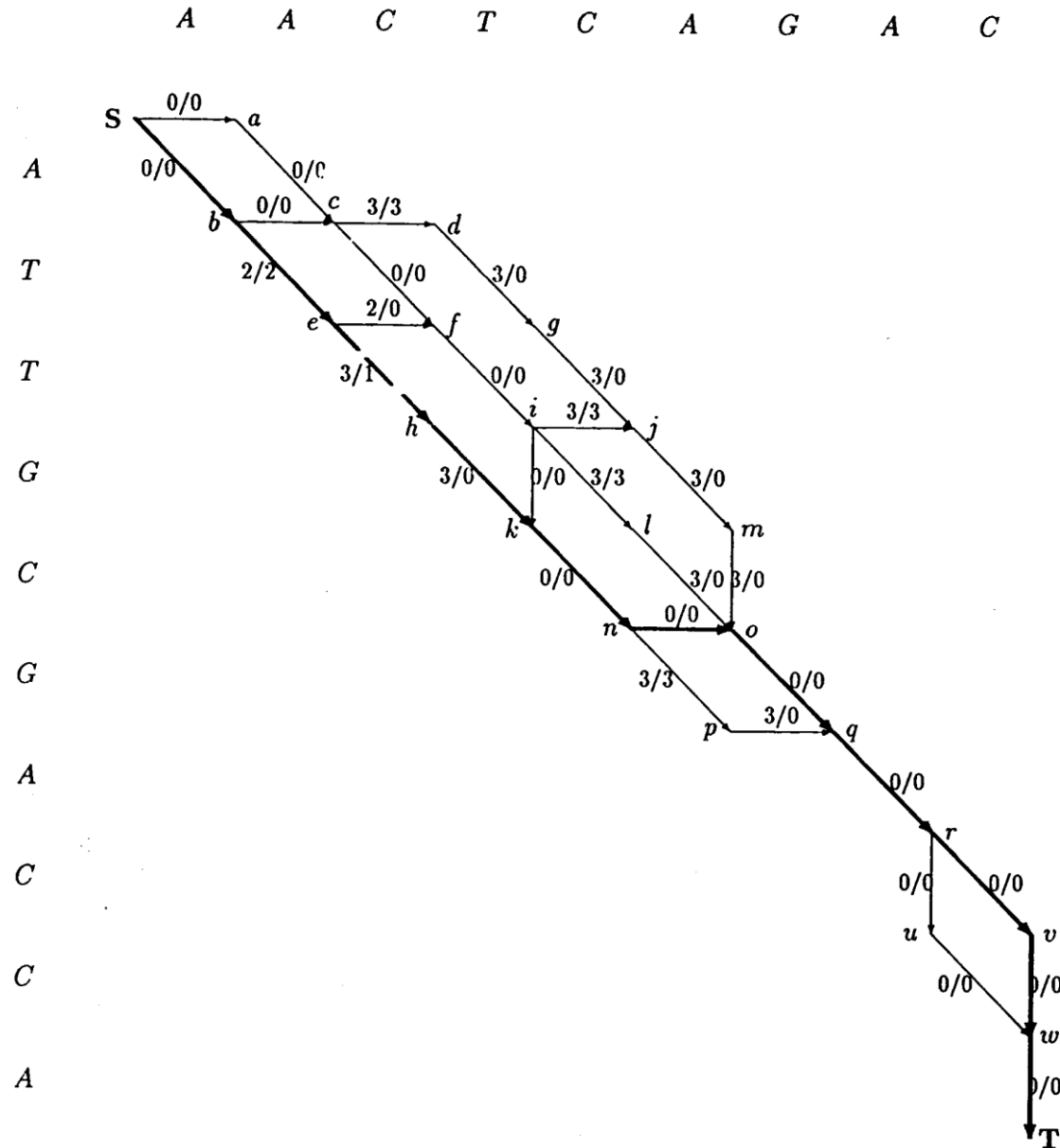
- By definition of  $\epsilon(e)$ ,  $\epsilon(e) = 0$  for any edge  $e$  on the longest  $u$ -to- $t$  path. Hence  $\delta(R)$  is  $\delta$  by Theorem 13.2.1

- If we use reweighting ( $\epsilon(e)$ ), pruning is possible without looking at the full path.



# Corollary 13.2.1

- E.g.)



Every edge  $\delta$  and  $\epsilon$  values  
(in the form  $\delta/\epsilon$ )

3-near optimal paths

# Counting near-optimal paths

---

- Why count?
  - The **number of near-optimal alignment** is used as a rule-of-thumb.
  - From empirical studies of specific molecular sequences,  
They report that alignments with **a large number of suboptimals** close to the optimal **tend not to correctly highlight important** biological information.

# How to count



## Definition

- Let  $N(v, \delta)$  be the number of  $\delta$ -near-optimal  $s$ -to- $t$  paths that go through node  $v$ .
- For a given value  $\Delta$ , the number of  $s$ -to- $t$  paths whose deviation from  $R^*$  is at most  $\Delta$  is

$$\sum_{\delta \leq \Delta} N(t, \delta)$$

- We compute the sum by evaluating the following recurrence for each node  $v$  and for each “needed” value of  $\delta$ :

$$N(v, \delta) = \sum_{(u,v) \in E} [N(u, \delta + \epsilon(u, v))]$$

# Enumerating near-optimal paths

---

- A search tree.
  - Each internal node  $x$  in the tree corresponds to a path  $R'$  from  $s$  to a node  $u$  in the alignment graph. The value  $d(x) = \sum_{e \in R'} \epsilon(e)$  is stored at node  $x$ , and by Corollary 13.2.1 there is a path from  $s$  to  $t$  of exactly that deviation from  $R^*$ .
  - The tree is expanded in a “best first search” manner. (select a internal node with the minimum  $d(x)$ ).

# 13.3 Chaining diverse local alignments

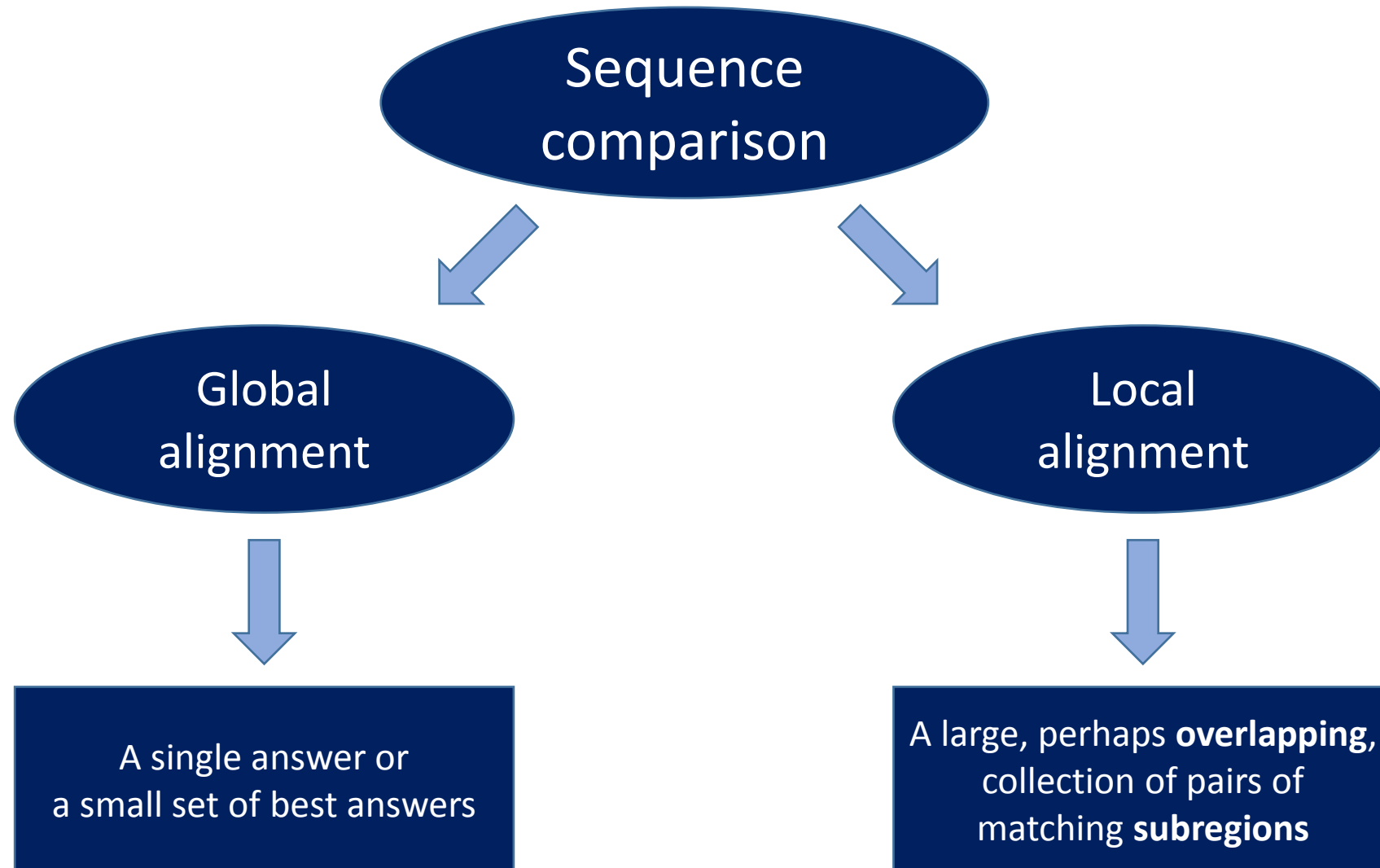
---

- Motivation
  - Common problem in genetic sequence comparison.
  - The Problem is to determine the functionality of a DNA or amino acid sequence by comparing the sequence to a database of sequences whose functionality is known.



## 13.3 Chaining diverse local alignments

---



# 13.3 Chaining diverse local alignments

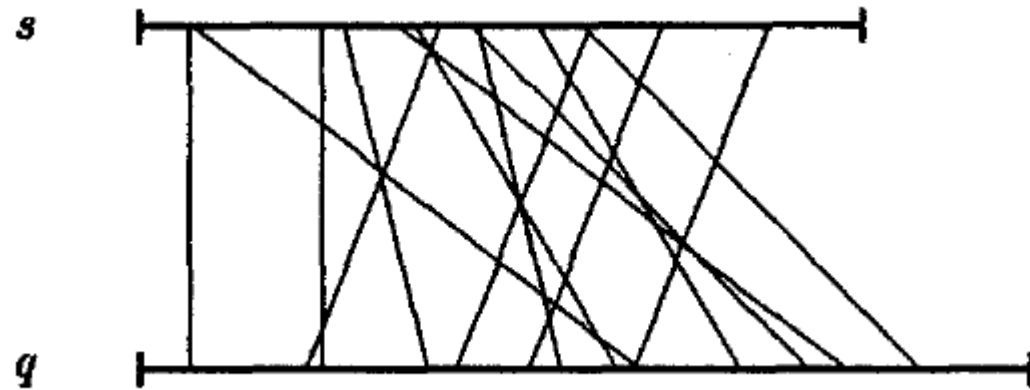
---

- Output of local alignment
  - Likely to have large number of overlapping matched regions.  
→ very hard to further analyze manually.
- Why local comparison is needed?
  - Local comparison methods are appropriate when the goal is to predict functionality.  
e.g.) gene assembly

# Selecting a best subset of pairs

---

- How should a “good” subset of optimal or suboptimal pairs be selected to display a relationship between strings  $S$  and  $S'$  over their entire length?  
→ **Chaining** together regions of high similarity given a set of **diverse local alignments**.
- Selecting a best subset of pairs!



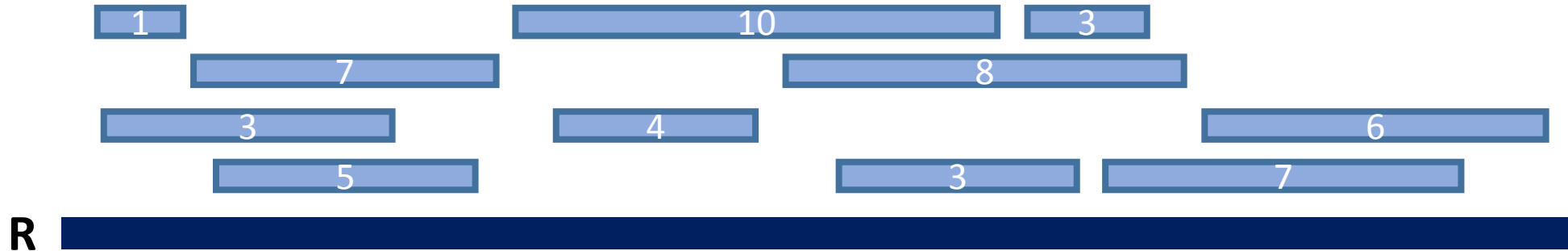


# A One-dimensional chaining problem

---

- Problem

- Line  $R$  and the  $r$  smaller intervals in  $R$ . Each interval has an associated value, and the problem is to pick a subset of non-overlapping intervals with largest total value.

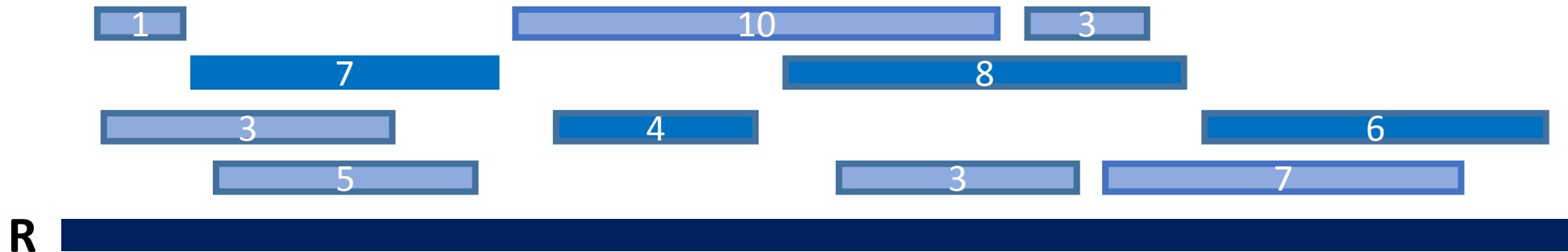


# A One-dimensional chaining problem

---

- Problem

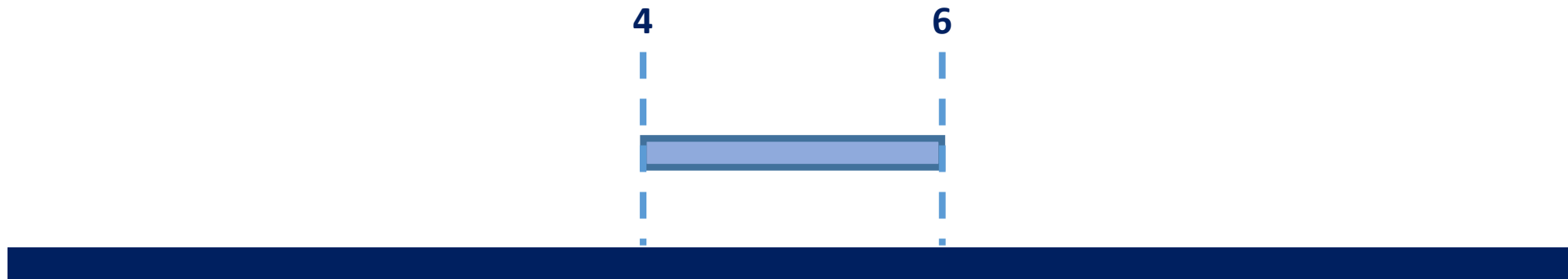
- Line  $R$  and the  $r$  smaller intervals in  $R$ . Each interval has an associated value, and the problem is to pick a subset of non-overlapping intervals with largest total value.



# One-dimensional Algorithm

---

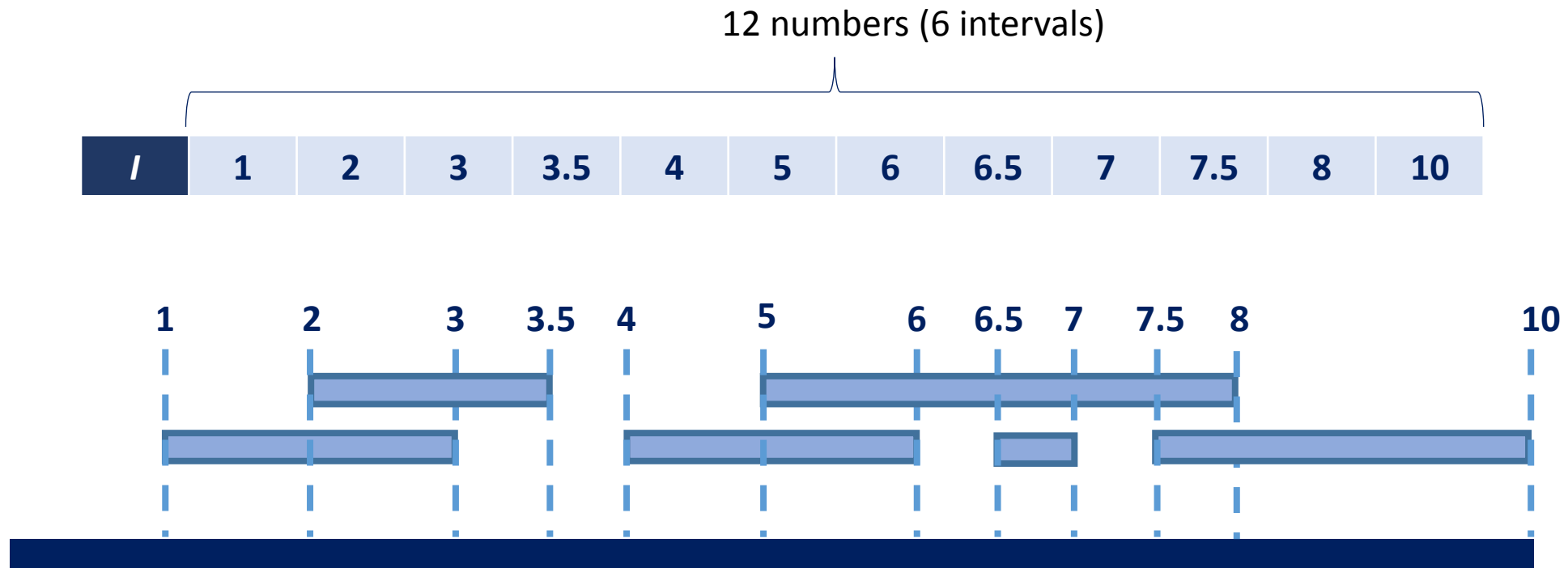
- Let  $I$  be a list of all the  **$2r$  numbers** representing the **locations of the endpoints** of the intervals.



- The number of intervals:  $r$

# One-dimensional Algorithm

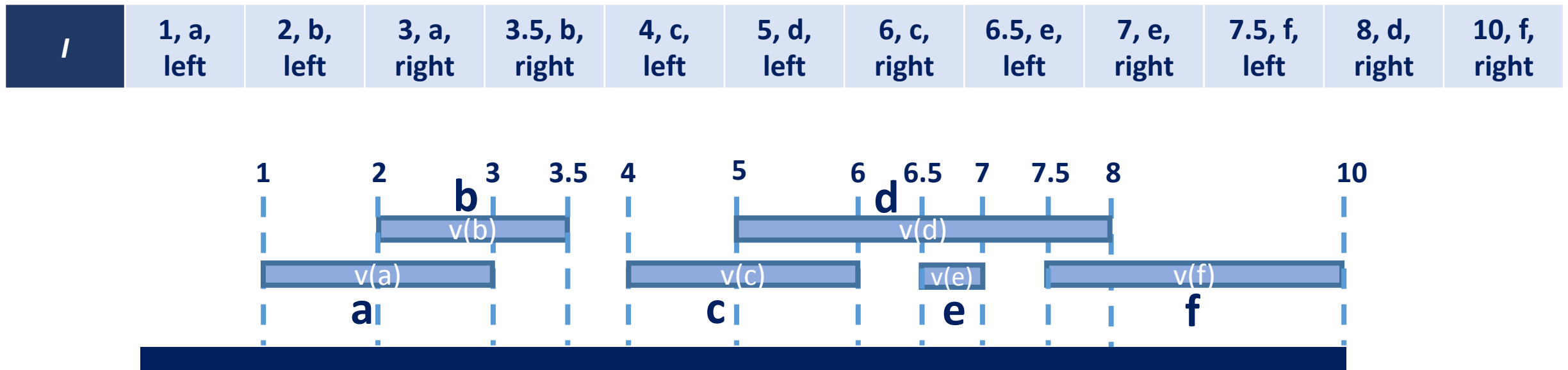
- Let  $I$  be a list of all the  **$2r$  numbers** representing the **locations of the endpoints** of the intervals.



- The number of intervals:  $r$

# One-dimensional Algorithm

- Sort the **numbers** in  $I$ , and annotate each entry in  $I$  with the **name of the interval** it is part of and whether it is a **left or a right** endpoint.  
For convenience, let  $I$  be a one-dimensional array.

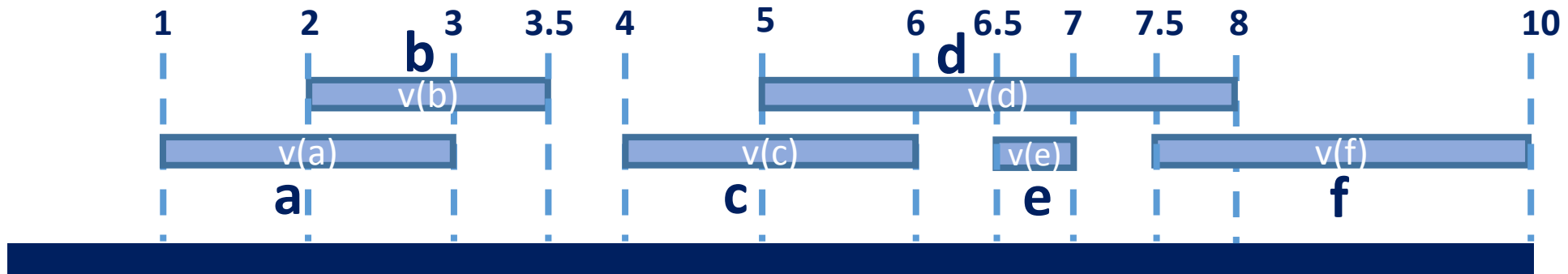


- The number of intervals:  $r$

# One-dimensional Algorithm

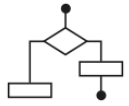
- Let  $V$  is a list of  $r$  values that indicate the best chain value containing the rectangular  $r$ .

$I$	1, a, left	2, b, left	3, a, right	3.5, b, right	4, c, left	5, d, left	6, c, right	6.5, e, left	7, e, right	7.5, f, left	8, d, right	10, f, right
$V$	$V(a)$	$V(b)$	$V(c)$	$V(d)$	$V(e)$	$V(f)$						



- The number of intervals:  $r$

# One-dimensional Algorithm



Set **max** to zero.

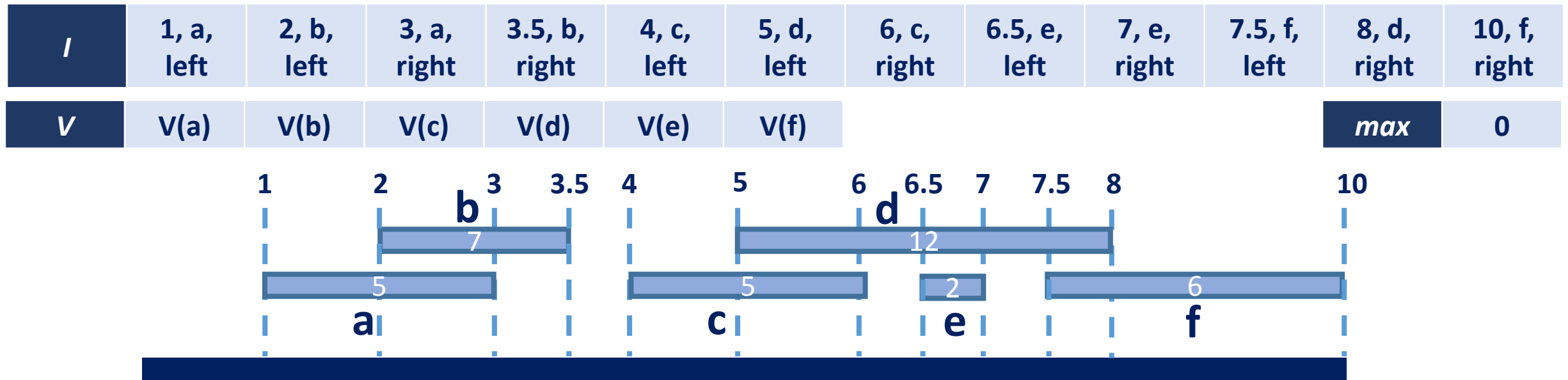
For  $i$  from 1 to  $2r$  do

Begin

If  $I[i]$  represents the **left** end of an interval, say interval  $j$ , then set  $V[j]$  to  $v(j) + \text{max}$ .

If  $I[j]$  represents the **right** end of interval  $j$ , then set **max** to the maximum of **max** and  $V[j]$ .

End.



For  $i$  from 1 to  $2r$  do

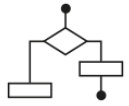
If  $I[i]$  represents the **left** end of an interval, say interval  $j$ , then set  $V[j]$  to  $v(j) + \max$ .

End.

40



# One-dimensional Algorithm



Set **max** to zero.

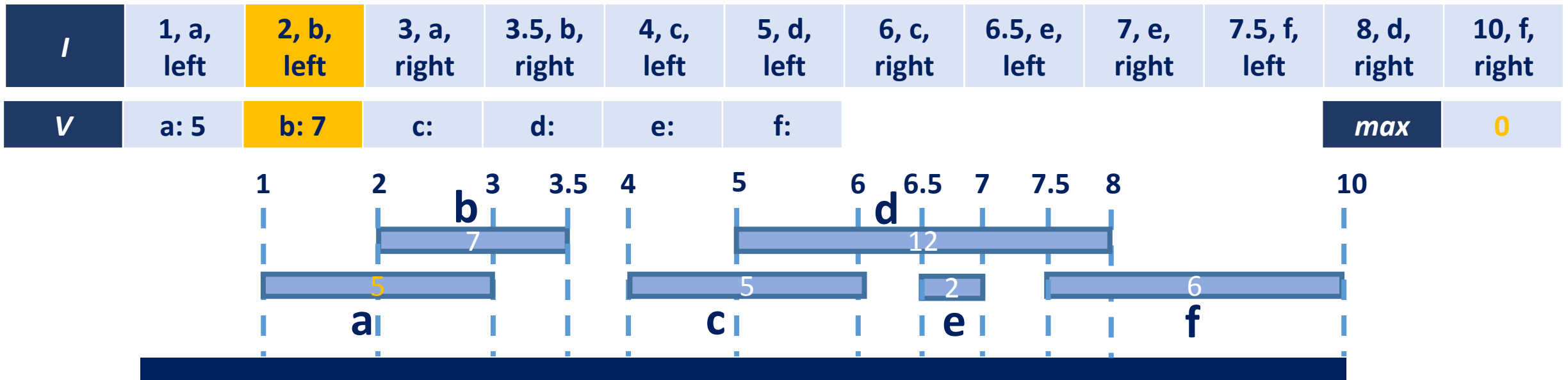
For  $i$  from 1 to  $2r$  do

Begin

If  $I[i]$  represents the **left** end of an interval, say interval  $j$ , then set  $V[j]$  to  $v(j) + \text{max}$ .

If  $I[j]$  represents the right end of interval  $j$ , then set **max** to the maximum of **max** and  $V[j]$ .

End.



For  $i$  from 1 to  $2r$  do

If  $l[j]$  represents the left end of an interval, say interval  $j$ , then set  $V[j]$  to  $v(j) + \max$ .

End.

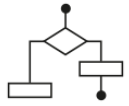
42

For  $i$  from 1 to  $2r$  do

If  $l[j]$  represents the left end of an interval, say interval  $j$ , then set  $V[j]$  to  $v(j) + \max$ .

End.

# One-dimensional Algorithm



Set **max** to zero.

For  $i$  from 1 to  $2r$  do

Begin

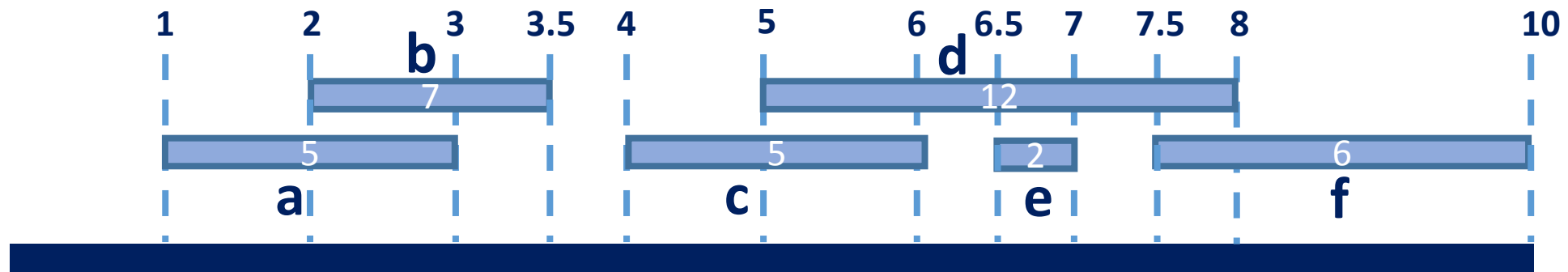
If  $I[i]$  represents the left end of an interval, say interval  $j$ , then set  $V[j]$  to  $v(j) + \text{max}$ .

If  $I[j]$  represents the **right** end of interval  $j$ , then set **max** to the maximum of **max** and  $V[j]$ .

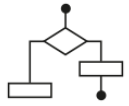
End.

$I$	1, a, left	2, b, left	3, a, right	3.5, b, right	4, c, left	5, d, left	6, c, right	6.5, e, left	7, e, right	7.5, f, left	8, d, right	10, f, right
-----	------------	------------	-------------	---------------	------------	------------	-------------	--------------	-------------	--------------	-------------	--------------

$V$	a: 5	b: 7	c:	d:	e:	f:							<b>max</b>	5
-----	------	------	----	----	----	----	--	--	--	--	--	--	------------	---



# One-dimensional Algorithm



Set **max** to zero.

For  $i$  from 1 to  $2r$  do

Begin

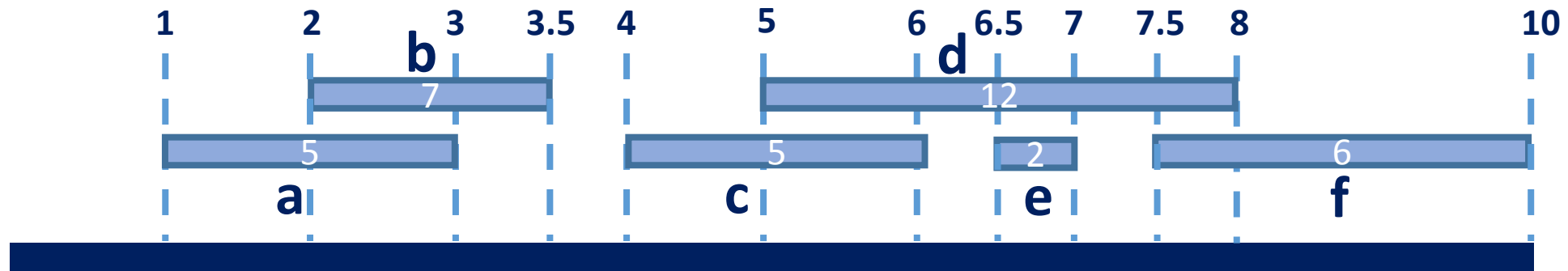
If  $I[i]$  represents the left end of an interval, say interval  $j$ , then set  $V[j]$  to  $v(j) + \text{max}$ .

If  $I[j]$  represents the **right** end of interval  $j$ , then set **max** to the maximum of **max** and  $V[j]$ .

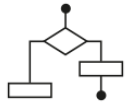
End.

$I$	1, a, left	2, b, left	3, a, right	3.5, b, right	4, c, left	5, d, left	6, c, right	6.5, e, left	7, e, right	7.5, f, left	8, d, right	10, f, right
-----	------------	------------	-------------	---------------	------------	------------	-------------	--------------	-------------	--------------	-------------	--------------

$V$	a: 5	b: 7	c:	d:	e:	f:							<b>max</b>	<b>7</b>
-----	------	------	----	----	----	----	--	--	--	--	--	--	------------	----------



# One-dimensional Algorithm



Set **max** to zero.

For  $i$  from 1 to  $2r$  do

Begin

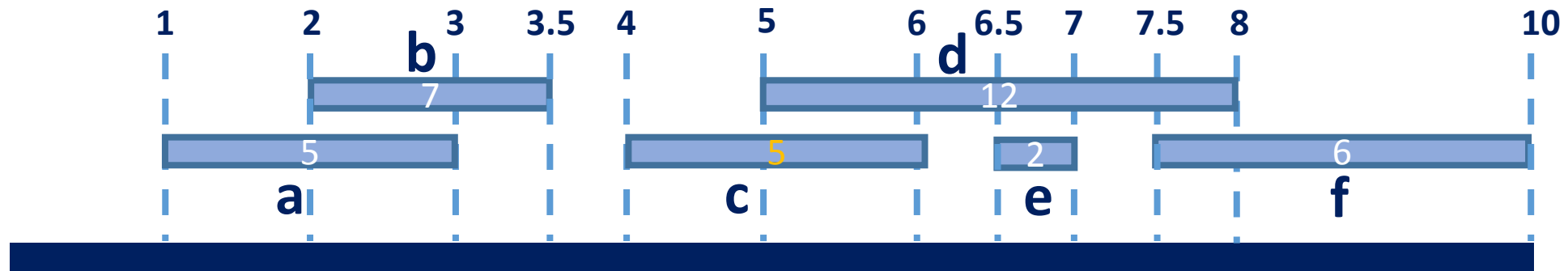
If  $I[i]$  represents the **left** end of an interval, say interval  $j$ , then set  $V[j]$  to  $v(j) + \text{max}$ .

If  $I[j]$  represents the right end of interval  $j$ , then set **max** to the maximum of **max** and  $V[j]$ .

End.

$I$	1, a, left	2, b, left	3, a, right	3.5, b, right	4, c, left	5, d, left	6, c, right	6.5, e, left	7, e, right	7.5, f, left	8, d, right	10, f, right
-----	------------	------------	-------------	---------------	------------	------------	-------------	--------------	-------------	--------------	-------------	--------------

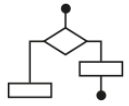
$V$	a: 5	b: 7	c: 12	d:	e:	f:							<b>max</b>	7
-----	------	------	-------	----	----	----	--	--	--	--	--	--	------------	---



End.

The Gantt chart illustrates the execution of six tasks (a, b, c, d, e, f) on a single processor. The timeline is marked from 1 to 10. Task 'a' runs from 1 to 3 with a duration of 5. Task 'b' runs from 2 to 3.5 with a duration of 7. Task 'c' runs from 4 to 5 with a duration of 5. Task 'd' runs from 5 to 8 with a duration of 12. Task 'e' runs from 6.5 to 7 with a duration of 2. Task 'f' runs from 7.5 to 10 with a duration of 6.

# One-dimensional Algorithm



Set **max** to zero.

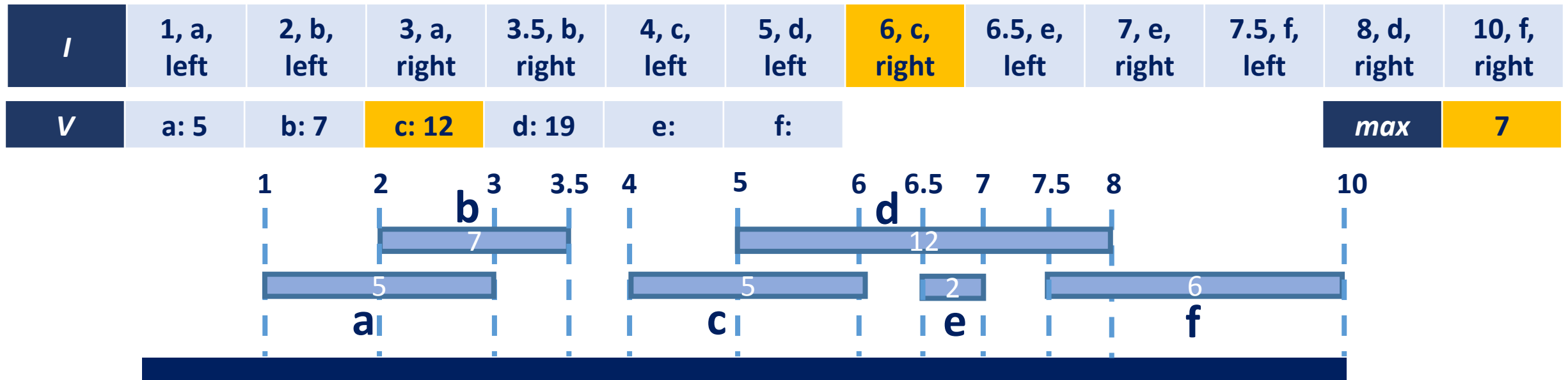
For  $i$  from 1 to  $2r$  do

Begin

If  $I[i]$  represents the left end of an interval, say interval  $j$ , then set  $V[j]$  to  $v(j) + \text{max}$ .

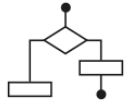
If  $I[j]$  represents the **right** end of interval  $j$ , then set **max** to the maximum of **max** and  $V[j]$ .

End.





# One-dimensional Algorithm



Set **max** to zero.

For  $i$  from 1 to  $2r$  do

Begin

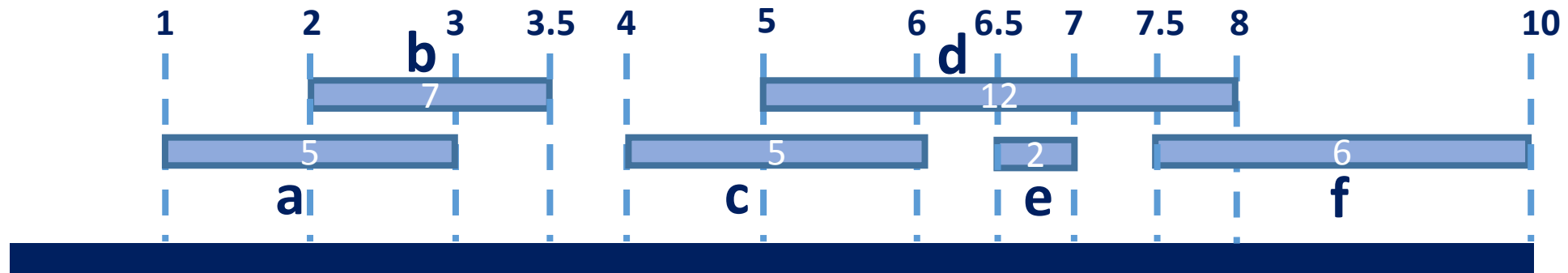
If  $I[i]$  represents the left end of an interval, say interval  $j$ , then set  $V[j]$  to  $v(j) + \text{max}$ .

If  $I[j]$  represents the **right** end of interval  $j$ , then set **max** to the maximum of **max** and  $V[j]$ .

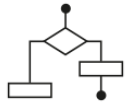
End.

$I$	1, a, left	2, b, left	3, a, right	3.5, b, right	4, c, left	5, d, left	6, c, right	6.5, e, left	7, e, right	7.5, f, left	8, d, right	10, f, right
-----	------------	------------	-------------	---------------	------------	------------	-------------	--------------	-------------	--------------	-------------	--------------

$V$	a: 5	b: 7	c: 12	d: 19	e:	f:							<b>max</b>	<b>12</b>
-----	------	------	-------	-------	----	----	--	--	--	--	--	--	------------	-----------



# One-dimensional Algorithm



Set **max** to zero.

For  $i$  from 1 to  $2r$  do

Begin

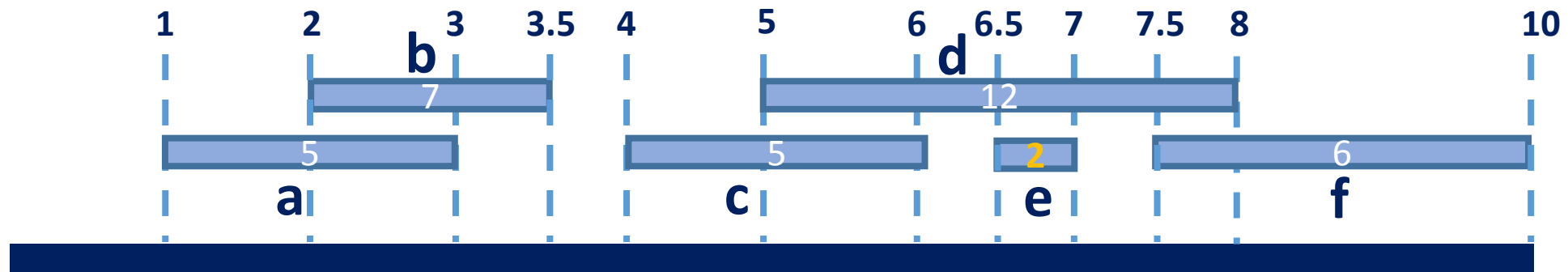
If  $I[i]$  represents the **left** end of an interval, say interval  $j$ , then set  $V[j]$  to  $v(j) + \text{max}$ .

If  $I[j]$  represents the right end of interval  $j$ , then set **max** to the maximum of **max** and  $V[j]$ .

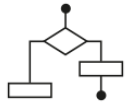
End.

$I$	1, a, left	2, b, left	3, a, right	3.5, b, right	4, c, left	5, d, left	6, c, right	6.5, e, left	7, e, right	7.5, f, left	8, d, right	10, f, right
-----	------------	------------	-------------	---------------	------------	------------	-------------	--------------	-------------	--------------	-------------	--------------

$V$	a: 5	b: 7	c: 12	d: 19	e: 14	f:						
											<b>max</b>	12



# One-dimensional Algorithm



Set **max** to zero.

For  $i$  from 1 to  $2r$  do

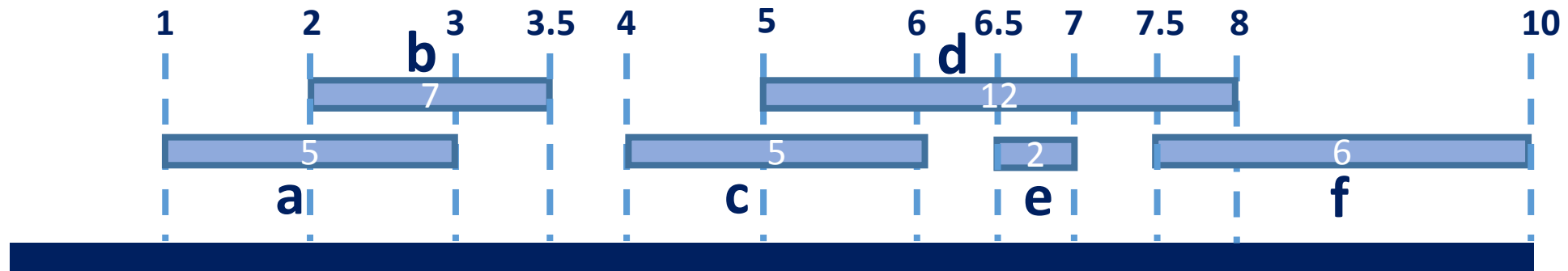
Begin

If  $I[i]$  represents the left end of an interval, say interval  $j$ , then set  $V[j]$  to  $v(j) + \text{max}$ .

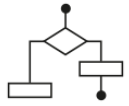
If  $I[j]$  represents the **right** end of interval  $j$ , then set **max** to the maximum of **max** and  $V[j]$ .

End.

<i>I</i>	1, a, left	2, b, left	3, a, right	3.5, b, right	4, c, left	5, d, left	6, c, right	6.5, e, left	7, e, right	7.5, f, left	8, d, right	10, f, right
<i>V</i>	a: 5	b: 7	c: 12	d: 19	e: 14	f:					<i>max</i>	12



# One-dimensional Algorithm



Set **max** to zero.

For  $i$  from 1 to  $2r$  do

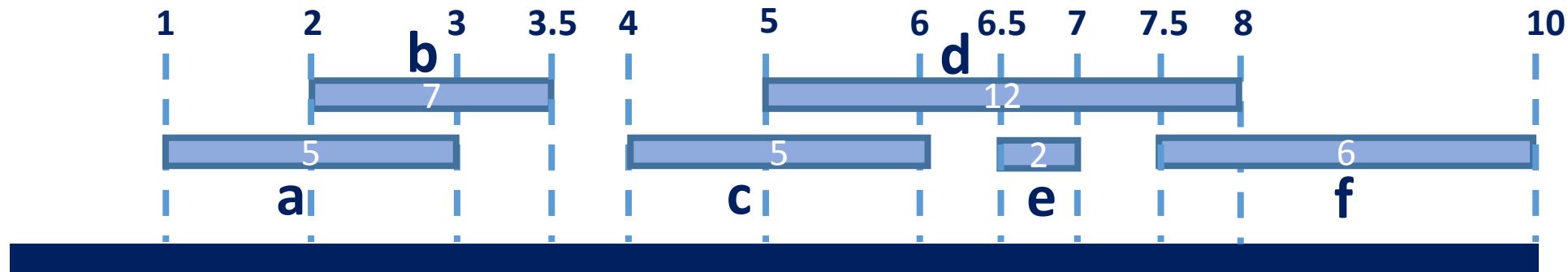
Begin

If  $I[i]$  represents the left end of an interval, say interval  $j$ , then set  $V[j]$  to  $v(j) + \text{max}$ .

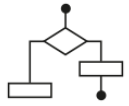
If  $I[j]$  represents the **right** end of interval  $j$ , then set **max** to the maximum of **max** and  $V[j]$ .

End.

<i>I</i>	1, a, left	2, b, left	3, a, right	3.5, b, right	4, c, left	5, d, left	6, c, right	6.5, e, left	7, e, right	7.5, f, left	8, d, right	10, f, right
<i>V</i>	a: 5	b: 7	c: 12	d: 19	e: 14	f:					<i>max</i>	14



# One-dimensional Algorithm



Set **max** to zero.

For  $i$  from 1 to  $2r$  do

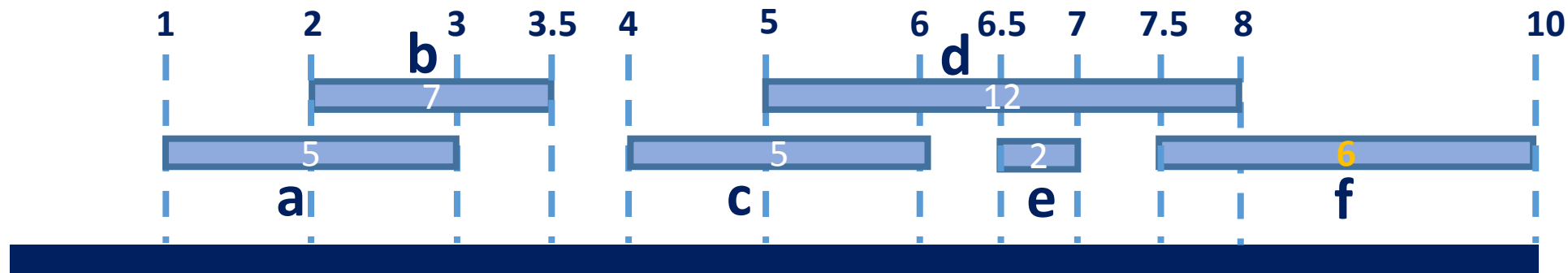
Begin

If  $I[i]$  represents the **left** end of an interval, say interval  $j$ , then set  $V[j]$  to  $v(j) + \text{max}$ .

If  $I[j]$  represents the right end of interval  $j$ , then set **max** to the maximum of **max** and  $V[j]$ .

End.

$I$	1, a, left	2, b, left	3, a, right	3.5, b, right	4, c, left	5, d, left	6, c, right	6.5, e, left	7, e, right	7.5, f, left	8, d, right	10, f, right
$V$	5	7	12	19	14	20						
											<b>max</b>	14



For  $i$  from 1 to  $2r$  do

If  $l[j]$  represents the left end of an interval, say interval  $j$ , then set  $V[j]$  to  $v(j) + \max$ .

End.

# One-dimensional Algorithm



Set *max* to zero.

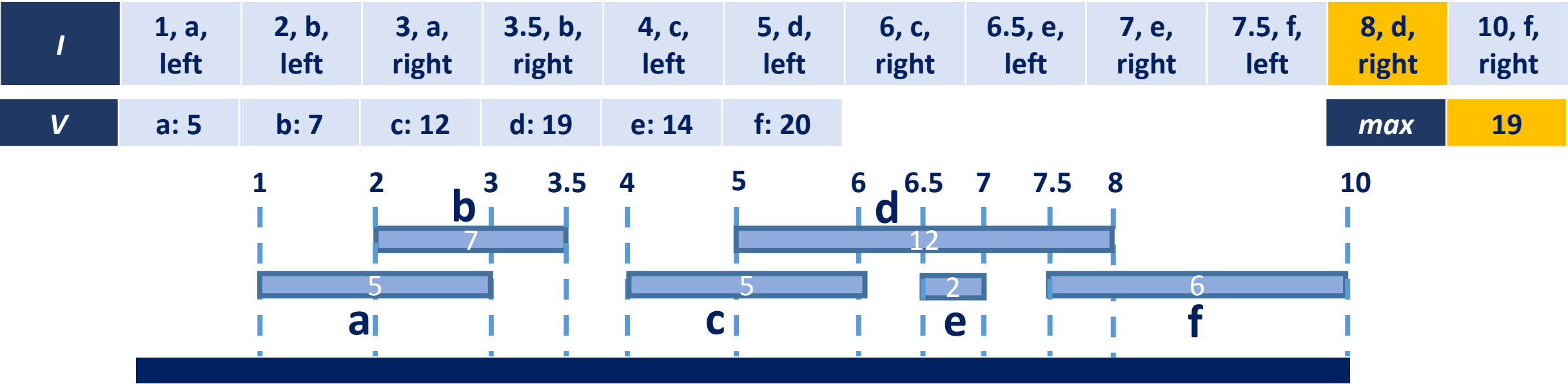
For *i* from 1 to 2*r* do

Begin

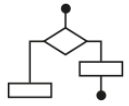
If *I*[*i*] represents the left end of an interval, say interval *j*, then set *V*[*j*] to *v*(*j*) + *max*.

If *I*[*j*] represents the **right** end of interval *j*, then set *max* to the maximum of *max* and *V*[*j*].

End.



# One-dimensional Algorithm



Set **max** to zero.

For  $i$  from 1 to  $2r$  do

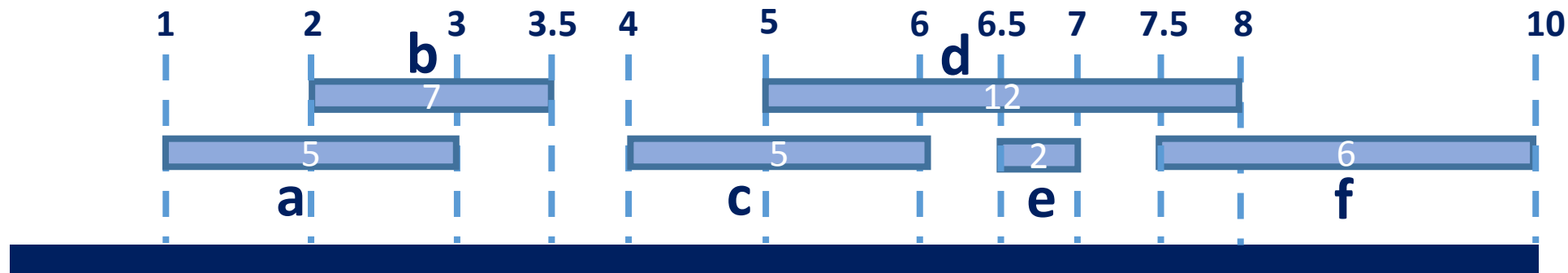
Begin

If  $I[i]$  represents the left end of an interval, say interval  $j$ , then set  $V[j]$  to  $v(j) + \text{max}$ .

If  $I[j]$  represents the **right** end of interval  $j$ , then set **max** to the maximum of **max** and  $V[j]$ .

End.

<i>I</i>	1, a, left	2, b, left	3, a, right	3.5, b, right	4, c, left	5, d, left	6, c, right	6.5, e, left	7, e, right	7.5, f, left	8, d, right	10, f, right
<i>V</i>	a: 5	b: 7	c: 12	d: 19	e: 14	f: 20					<i>max</i>	19





# One-dimensional Algorithm



Set *max* to zero.

For *i* from 1 to 2r do

Begin

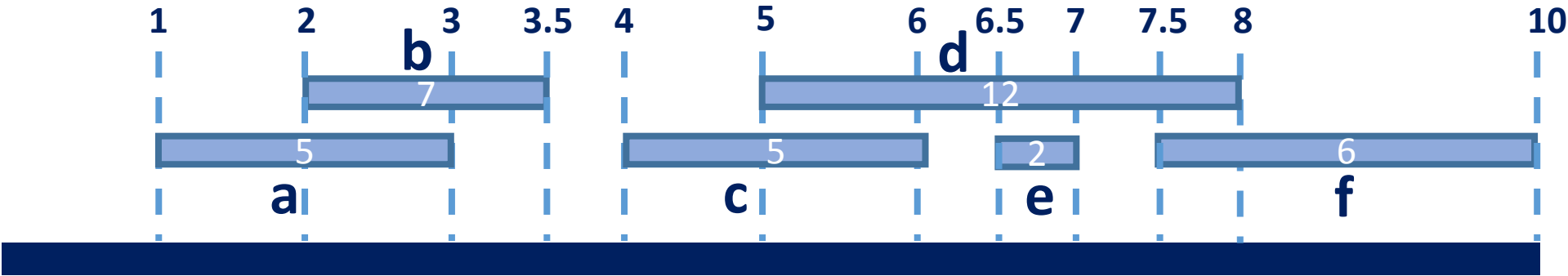
If *I*[*i*] represents the left end of an interval, say interval *j*, then set *V*[*j*] to *v*(*j*) + *max*.

If *I*[*j*] represents the **right** end of interval *j*, then set *max* to the maximum of *max* and *V*[*j*].

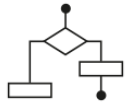
End.

<i>I</i>	1, a, left	2, b, left	3, a, right	3.5, b, right	4, c, left	5, d, left	6, c, right	6.5, e, left	7, e, right	7.5, f, left	8, d, right	10, f, right
----------	------------	------------	-------------	---------------	------------	------------	-------------	--------------	-------------	--------------	-------------	--------------

<i>V</i>	a: 5	b: 7	c: 12	d: 19	e: 14	f: 20					<i>max</i>	20
----------	------	------	-------	-------	-------	-------	--	--	--	--	------------	----



# One-dimensional Algorithm



Set **max** to zero.

For  $i$  from 1 to  $2r$  do

Begin

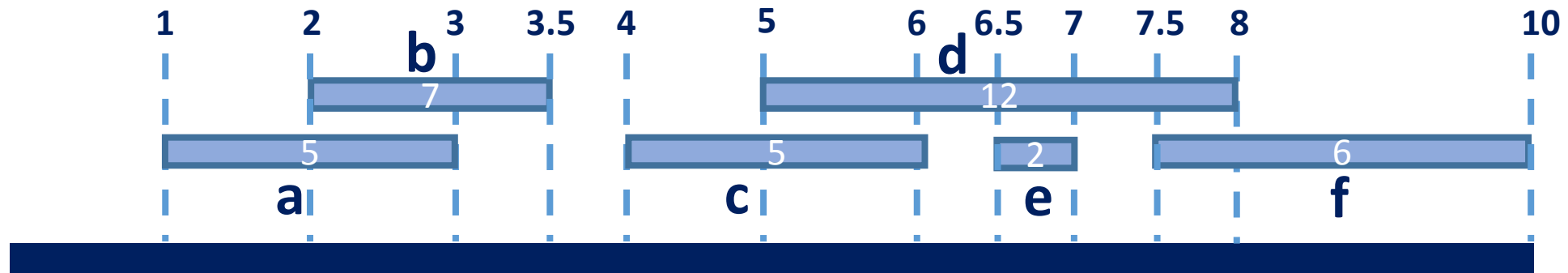
If  $I[i]$  represents the left end of an interval, say interval  $j$ , then set  $V[j]$  to  $v(j) + \text{max}$ .

If  $I[j]$  represents the right end of interval  $j$ , then set **max** to the maximum of **max** and  $V[j]$ .

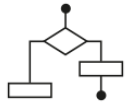
End.

$I$	1, a, left	2, b, left	3, a, right	3.5, b, right	4, c, left	5, d, left	6, c, right	6.5, e, left	7, e, right	7.5, f, left	8, d, right	10, f, right
-----	------------	------------	-------------	---------------	------------	------------	-------------	--------------	-------------	--------------	-------------	--------------

$V$	a: 5	b: 7	c: 12	d: 19	e: 14	f: 20							max	20
-----	------	------	-------	-------	-------	-------	--	--	--	--	--	--	-----	----



# One-dimensional Algorithm



Set **max** to zero.

For  $i$  from 1 to  $2r$  do

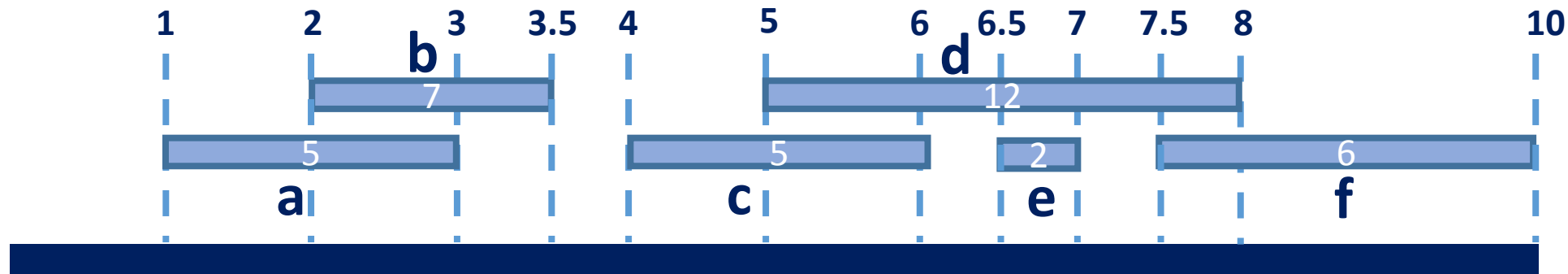
Begin

If  $I[i]$  represents the left end of an interval, say interval  $j$ , then set  $V[j]$  to  $v(j) + \text{max}$ .

If  $I[j]$  represents the right end of interval  $j$ , then set **max** to the maximum of **max** and  $V[j]$ .

End.

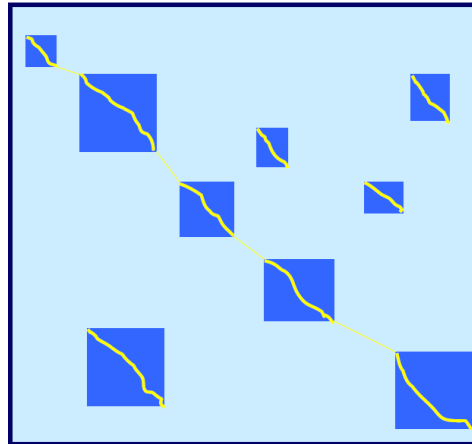
<i>I</i>	1, a, left	2, b, left	3, a, right	3.5, b, right	4, c, left	5, d, left	6, c, right	6.5, e, left	7, e, right	7.5, f, left	8, d, right	10, f, right
<i>V</i>	a: 5	b: 7	c: 12	d: 19	e: 14	f: 20					<i>max</i>	20



# The two-dimensional chain problem

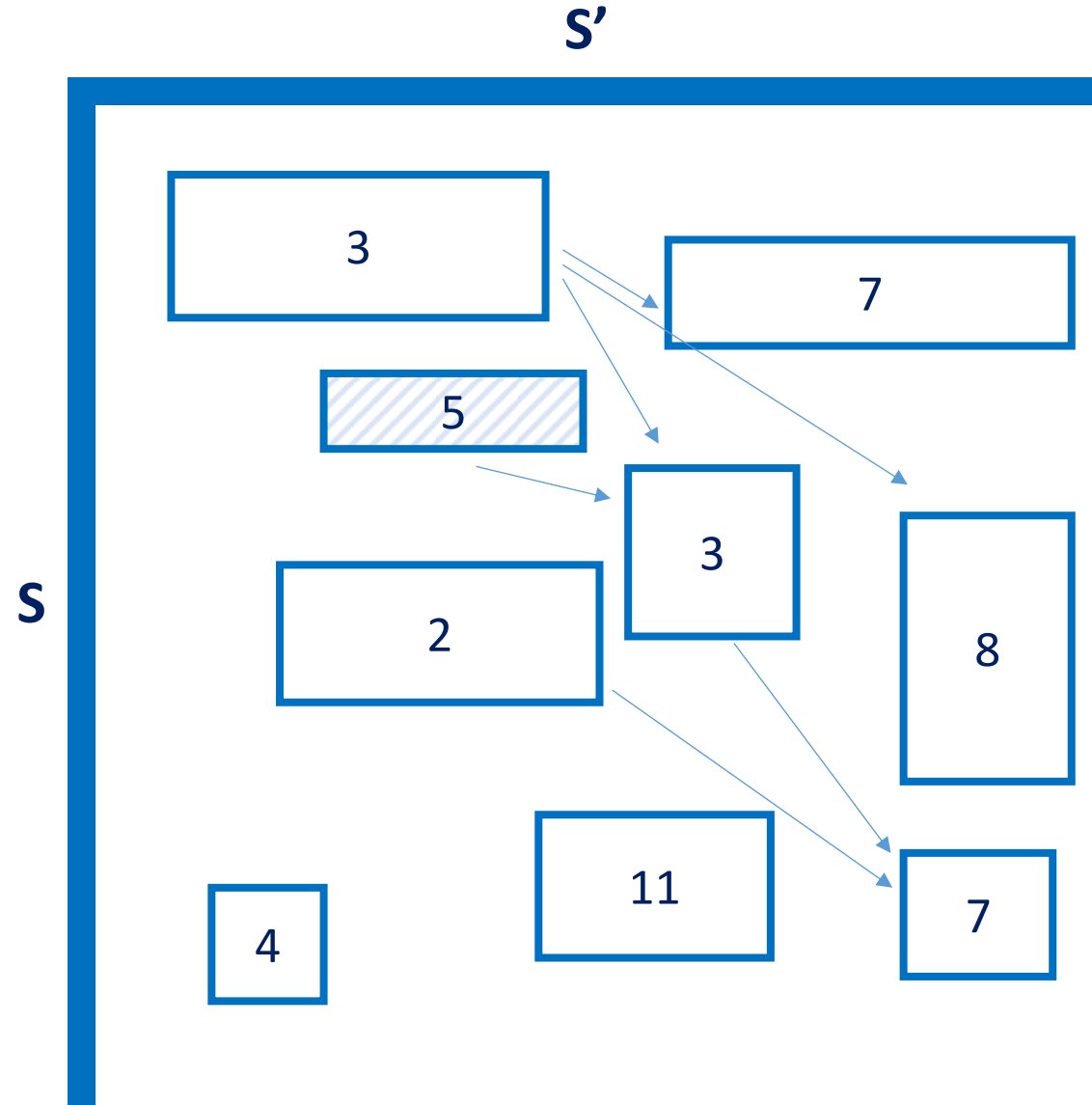
---

- Now return to the original problem of selecting pairs of highly similar substrings.
- The substrings may overlap each other, and the problem is **to select a “good” subset of substring pairs** so that none of two strings,  $S$  and  $S'$  overlap each other.



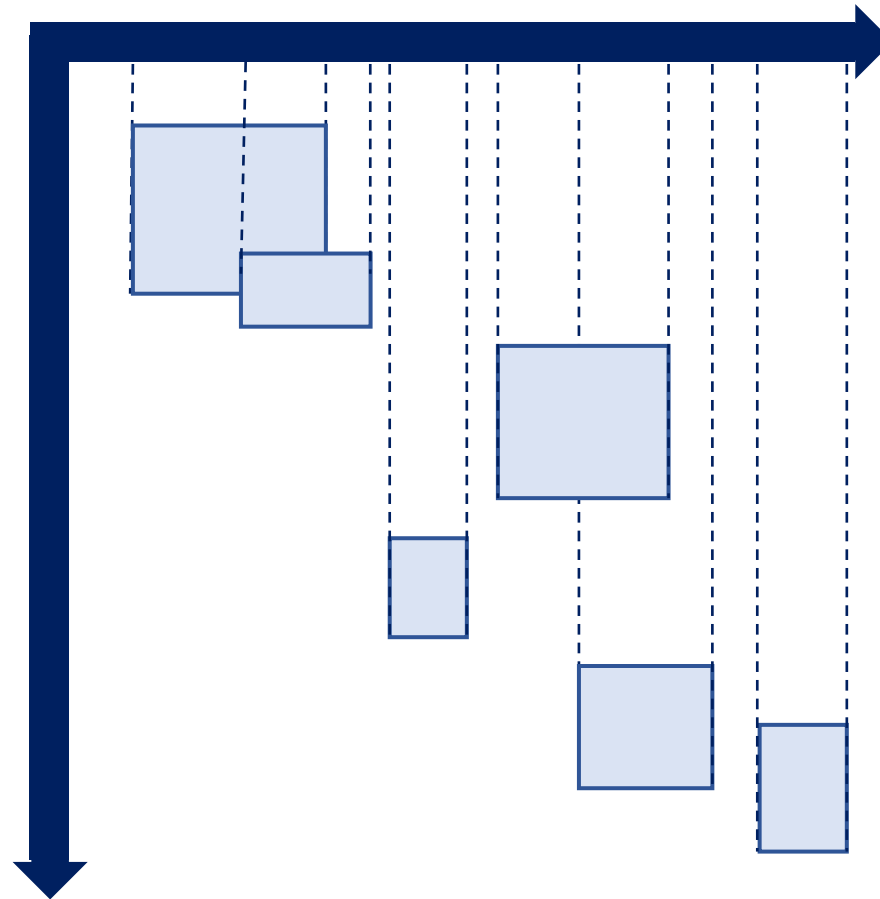
# The two-dimensional chain problem

- Each **directed line** points from a rectangle to a rectangle that can succeed it in a chain.
- The **value of each rectangle** can be the similarity value of the two substrings.
- The **optimal chain**?



# The two-dimensional chain algorithm

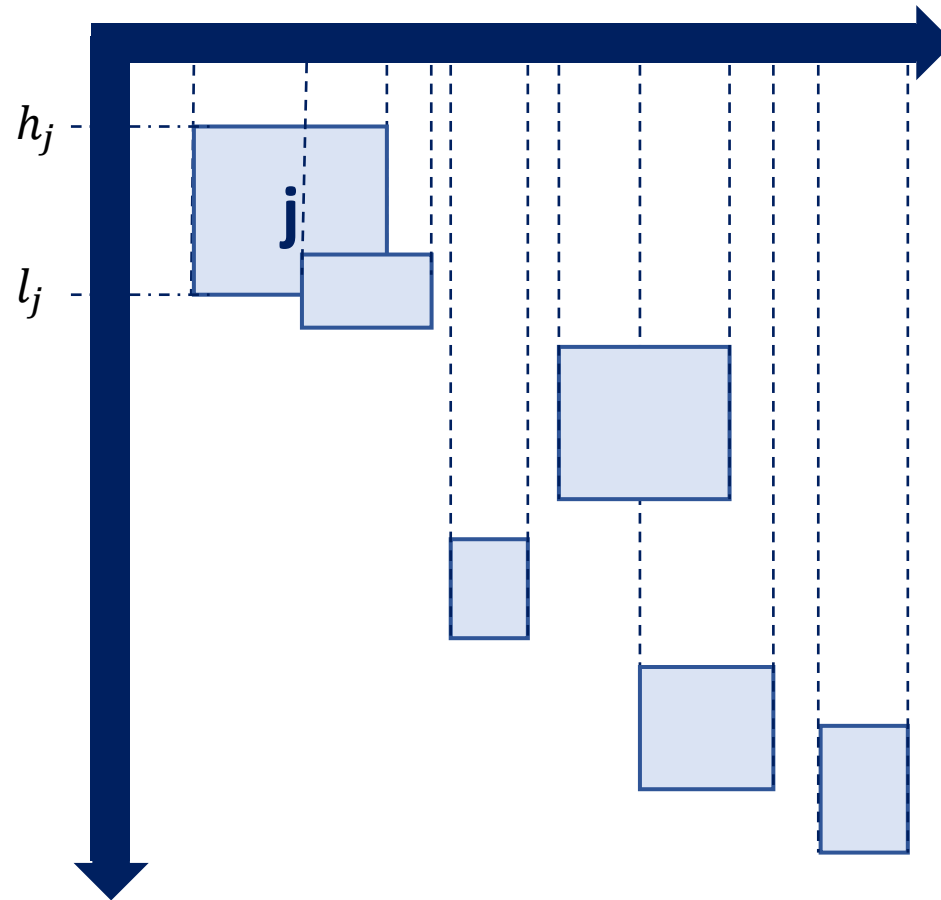
- Let  $I$  be a list of all the  **$2r$  numbers** representing the **locations of the endpoints** of the rectangles of  $x$  coordinate. (sorted left to right order)



- The number of rectangles:  $r$

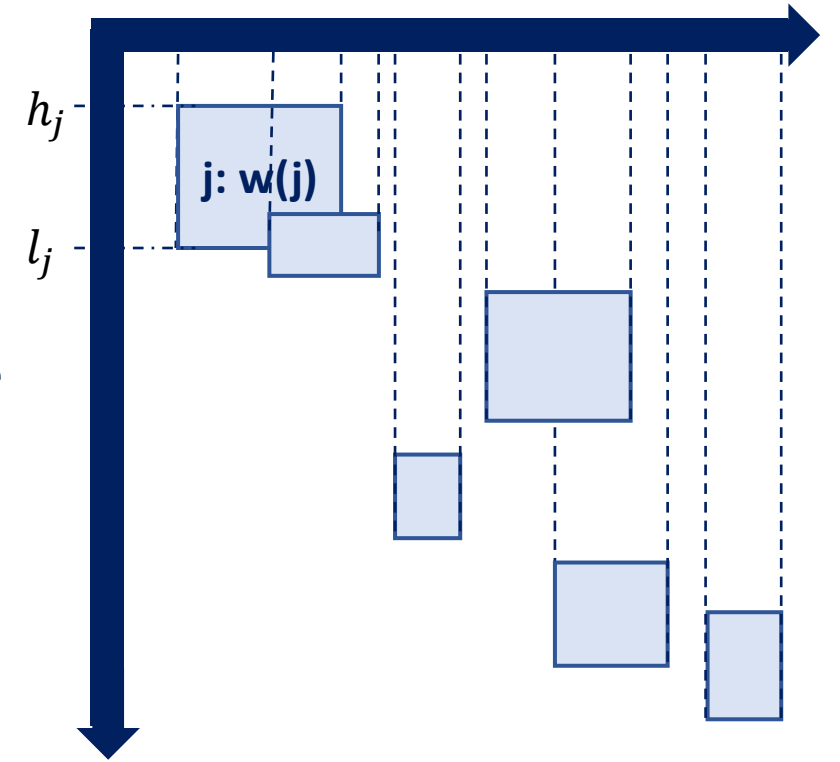
# The two-dimensional chain algorithm

- For each rectangle  $j$ , let  $h_j$  be the  $y$  coordinate of its **highest point**, and let  $l_j$  be the  $y$  coordinate of its **lowest point**.



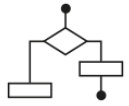
# The two-dimensional chain algorithm

- $w(j)$  is the weight of rectangle  $j$
- $V(j)$  is optimal score of chain ending in  $j$
- **L is a list of triplets  $(l_j, V(j), j)$** 
  - L is sorted by  $l_j$ : smallest(North) to largest (South) value
  - L is implemented as a balanced binary tree





# The two-dimensional chain algorithm



List L begins empty.

For  $i$  from 1 to  $2r$  do

Begin

If  $I[i]$  is the left end of a rectangle, say rectangle  $k$ , then

search L for the first triple (largest  $l_j$ ) where  $l_j < h_k$

set  $V(k) = w(k) + V(j)$

Else

If  $I[i]$  is the right end of a rectangle  $k$ , then

search L for the last triple where  $l_j \leq l_k$

if  $V(j) < V(k)$ , then

Insert the triple  $(l_k, V(k), k)$  into L, in the proper location to keep the triples sorted by their  $l$  values.

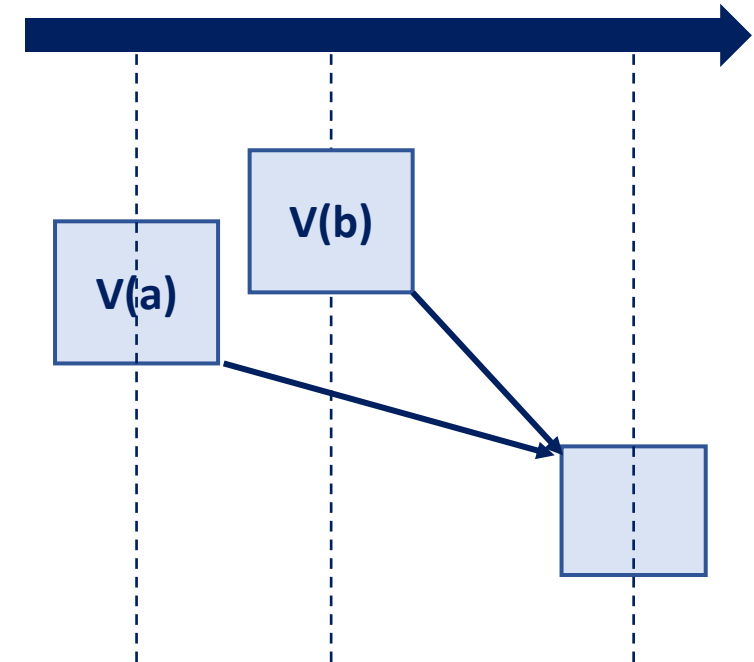
Delete from L the triple for every rectangle  $j'$  where  $l_{j'} \geq l_k$  and  $V(j') < V(k)$ .

End.

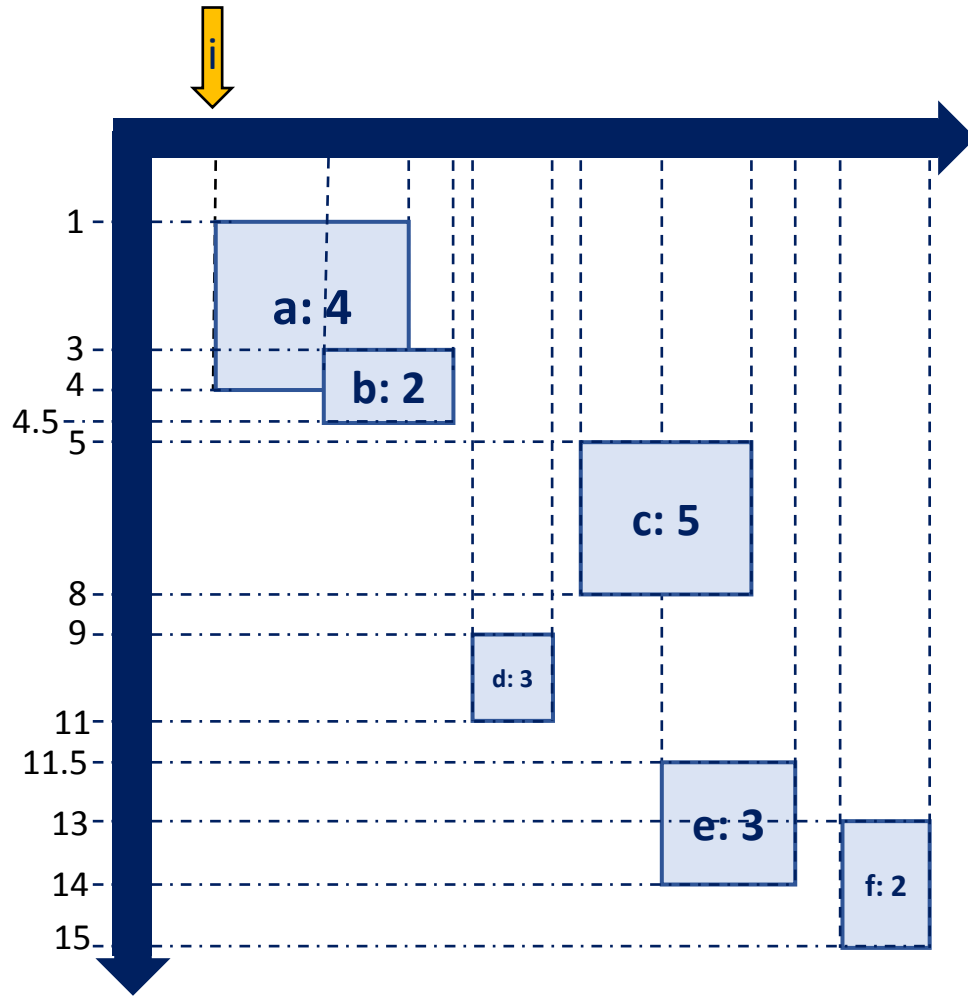
# The two-dimensional chain algorithm

## Main Idea

- Sweep through x-coordinates
- To the right of b, anything chainable to a is chainable to b
- Therefore, if  $V(b) > V(a)$ , rectangle a is “useless” for subsequent chaining.
- In L, keep rectangles j sorted with increasing  $l_j$  coordinates → sorted with increasing  $V(j)$  score.



# The two-dimensional chain algorithm



$V$	a	b	c	d	e	f
	4					

$L$	$l_j$	
	$V(j)$	
	$J$	

For  $i$  from 1 to  $2r$  do

If  $I[i]$  is the left end of a rectangle, say rectangle  $k$ , then

search  $L$  for the first triple (largest  $l_j$ ) where  $l_j < h_k$

set  $V(k) = w(k) + V(j)$

If  $I[i]$  is the right end of a rectangle  $k$ , then

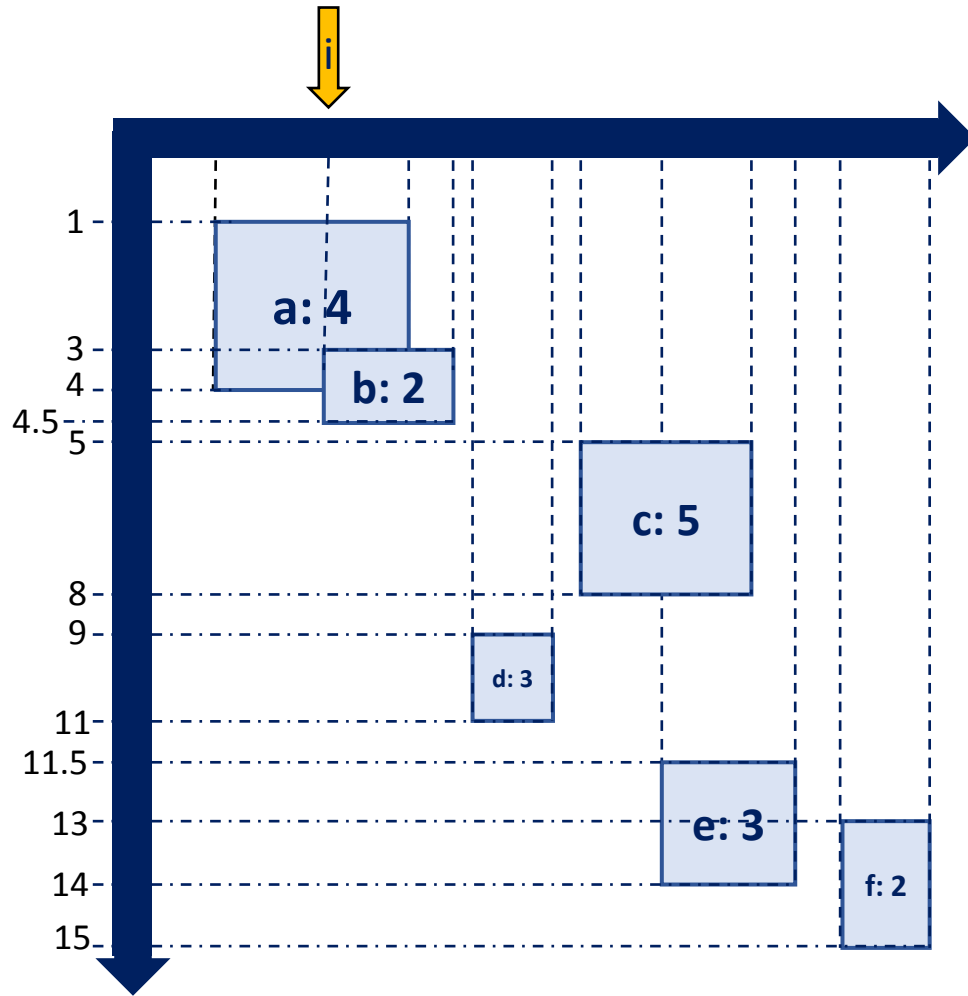
search  $L$  for the last triple where  $l_j \leq l_k$

if  $V(j) < V(k)$ , then

Insert the triple  $(l_k, V(k), k)$  into  $L$ , in the proper location to keep the triples sorted by their  $l$  values.

Delete from  $L$  the triple for every rectangle  $j'$  where  $l_{j'} \geq l_k$  and  $V(j') < V(k)$ .

# The two-dimensional chain algorithm



$V$	a	b	c	d	e	f
	4	2				

$L$	$l_j$	
	$V(j)$	
	$J$	

For  $i$  from 1 to  $2r$  do

If  $I[i]$  is the left end of a rectangle, say rectangle  $k$ , then

search  $L$  for the first triple (largest  $l_j$ ) where  $l_j < h_k$

set  $V(k) = w(k) + V(j)$

If  $I[i]$  is the right end of a rectangle  $k$ , then

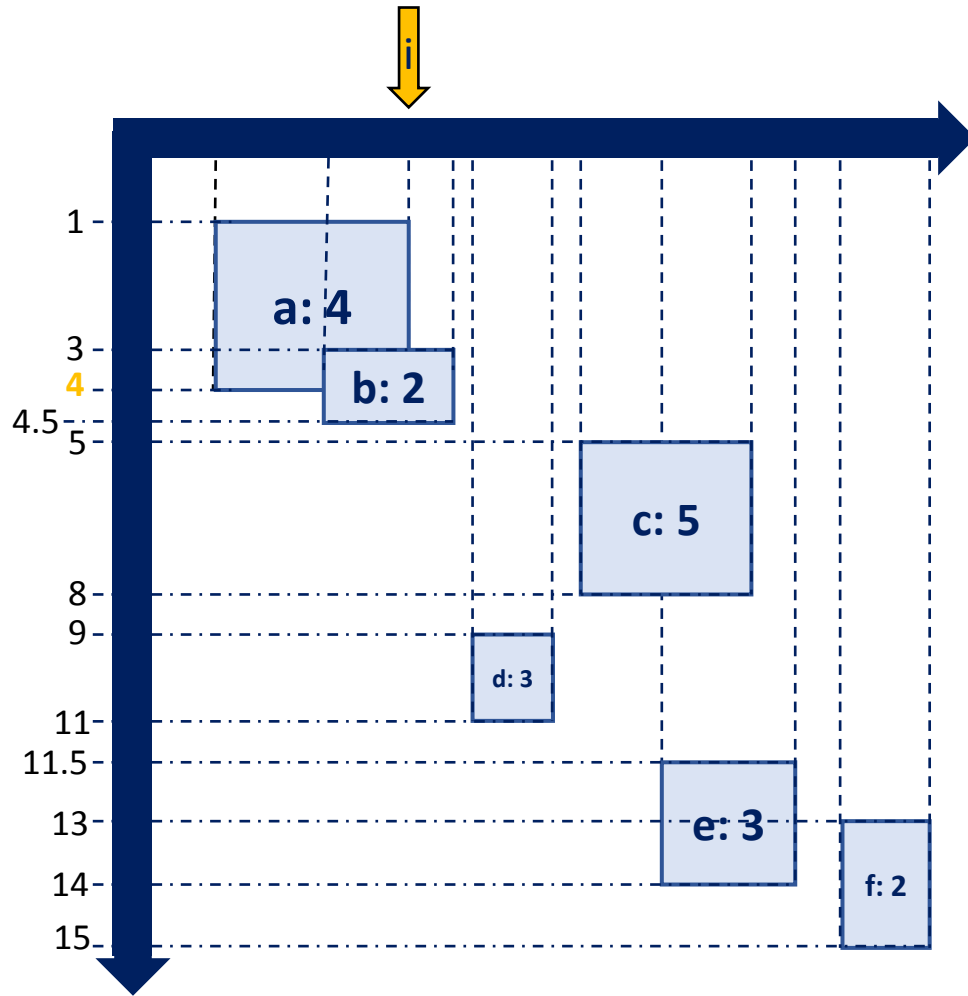
search  $L$  for the last triple where  $l_j \leq l_k$

if  $V(j) < V(k)$ , then

Insert the triple  $(l_k, V(k), k)$  into  $L$ , in the proper location to keep the triples sorted by their  $l$  values.

Delete from  $L$  the triple for every rectangle  $j'$  where  $l_{j'} \geq l_k$  and  $V(j') < V(k)$ .

# The two-dimensional chain algorithm



$V$	a	b	c	d	e	f
	4	2				

$L$	$l_j$	4
	$V(j)$	4
	$j$	a

For  $i$  from 1 to  $2r$  do

If  $I[i]$  is the left end of a rectangle, say rectangle  $k$ , then

search  $L$  for the first triple (largest  $l_j$ ) where  $l_j < h_k$

set  $V(k) = w(k) + V(j)$

If  $I[i]$  is the right end of a rectangle  $k$ , then

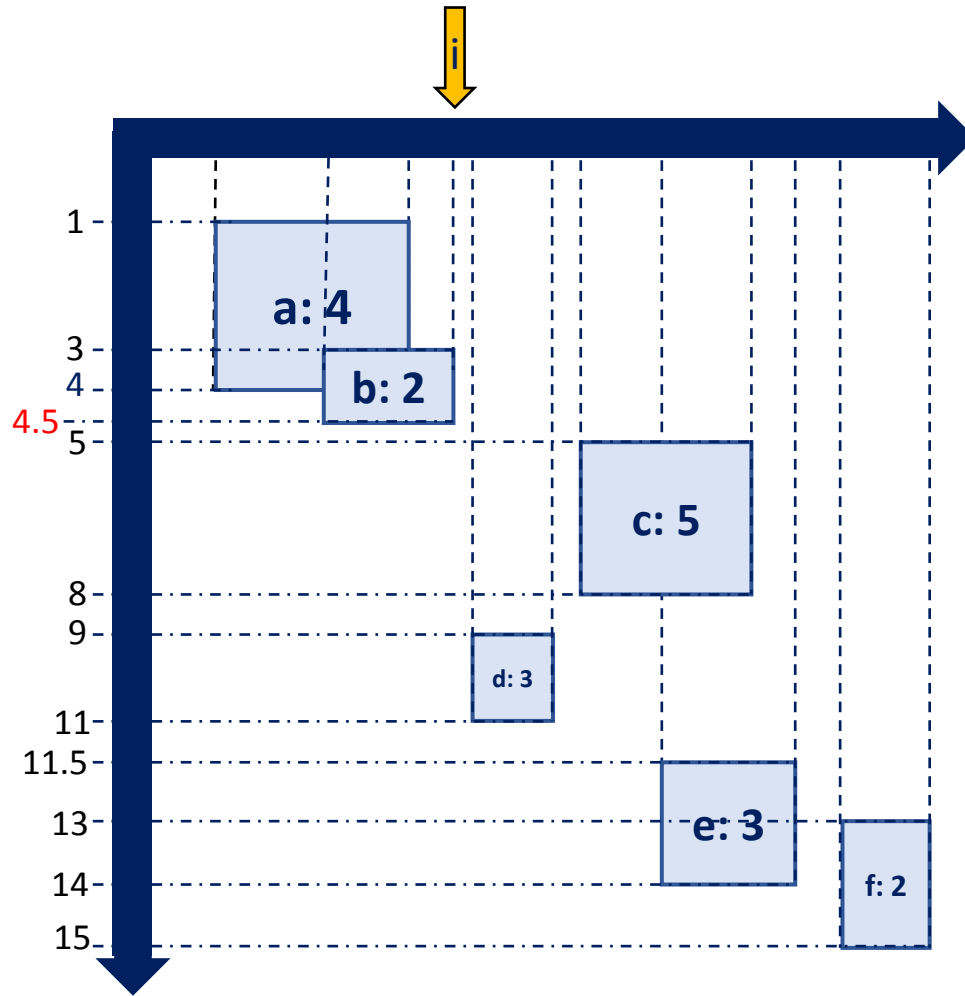
search  $L$  for the last triple where  $l_j \leq l_k$

if  $V(j) < V(k)$ , then

Insert the triple  $(l_k, V(k), k)$  into  $L$ , in the proper location to keep the triples sorted by their  $l$  values.

Delete from  $L$  the triple for every rectangle  $j'$  where  $l_{j'} \geq l_k$  and  $V(j') < V(k)$ .

# The two-dimensional chain algorithm



$V$	a	b	c	d	e	f
	4	2				

$L$	$l_j$	4
	$V(j)$	4
	$j$	a

For  $i$  from 1 to  $2r$  do

If  $i$  is the left end of a rectangle, say rectangle  $k$ , then

search  $L$  for the first triple (largest  $l_j$ ) where  $l_j < h_k$

set  $V(k) = w(k) + V(j)$

If  $i$  is the right end of a rectangle  $k$ , then

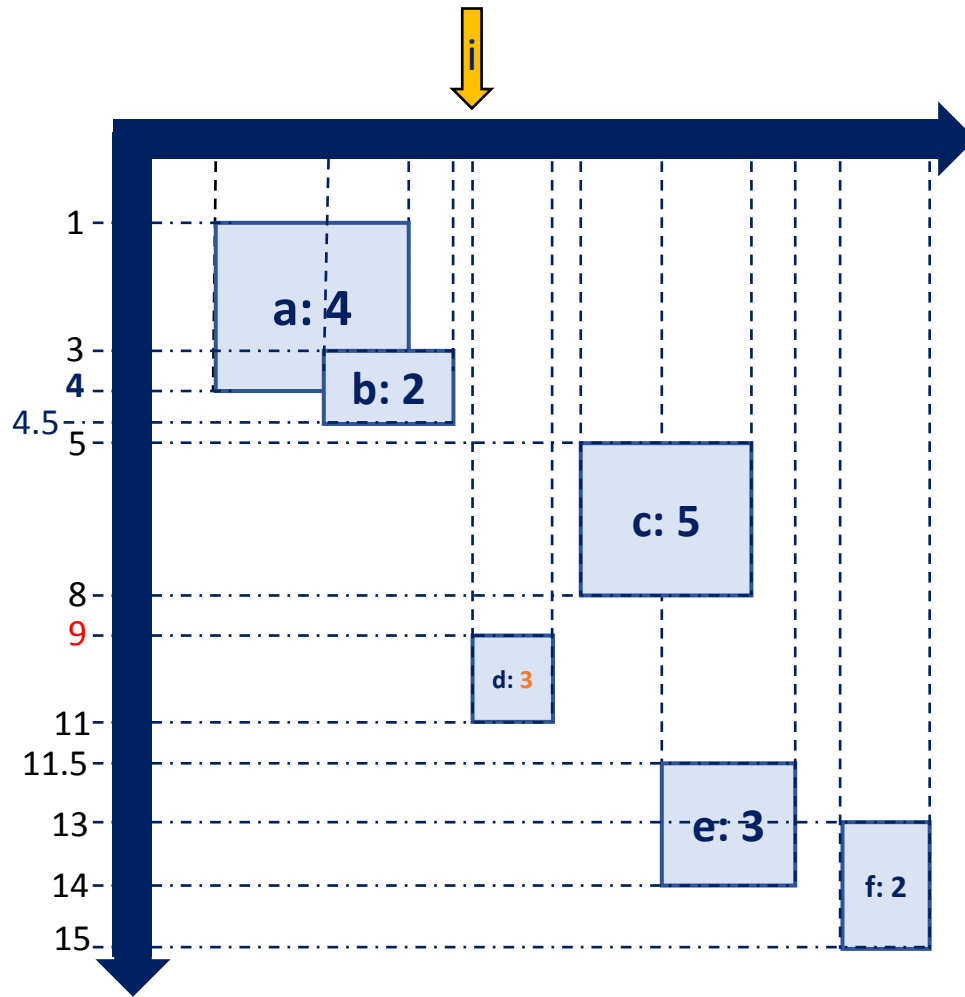
search  $L$  for the last triple where  $l_j \leq h_k$

if  $V(j) < V(k)$ , then

Insert the triple  $(l_k, V(k), k)$  into  $L$ , in the proper location to keep the triples sorted by their  $l$  values.

Delete from  $L$  the triple for every rectangle  $j'$  where  $l_{j'} \geq l_k$  and  $V(j') < V(k)$ .

# The two-dimensional chain algorithm



$V$	a	b	c	d	e	f
	4	2		7		

$L$	$l_j$	4
	$V(j)$	4
	$j$	a

For  $i$  from 1 to  $2r$  do

If  $l[i]$  is the left end of a rectangle, say rectangle  $k$ , then

search  $L$  for the first triple (largest  $l_j$ ) where  $l_j < h_k$

set  $V(k) = w(k) + V(j)$

If  $l[i]$  is the right end of a rectangle  $k$ , then

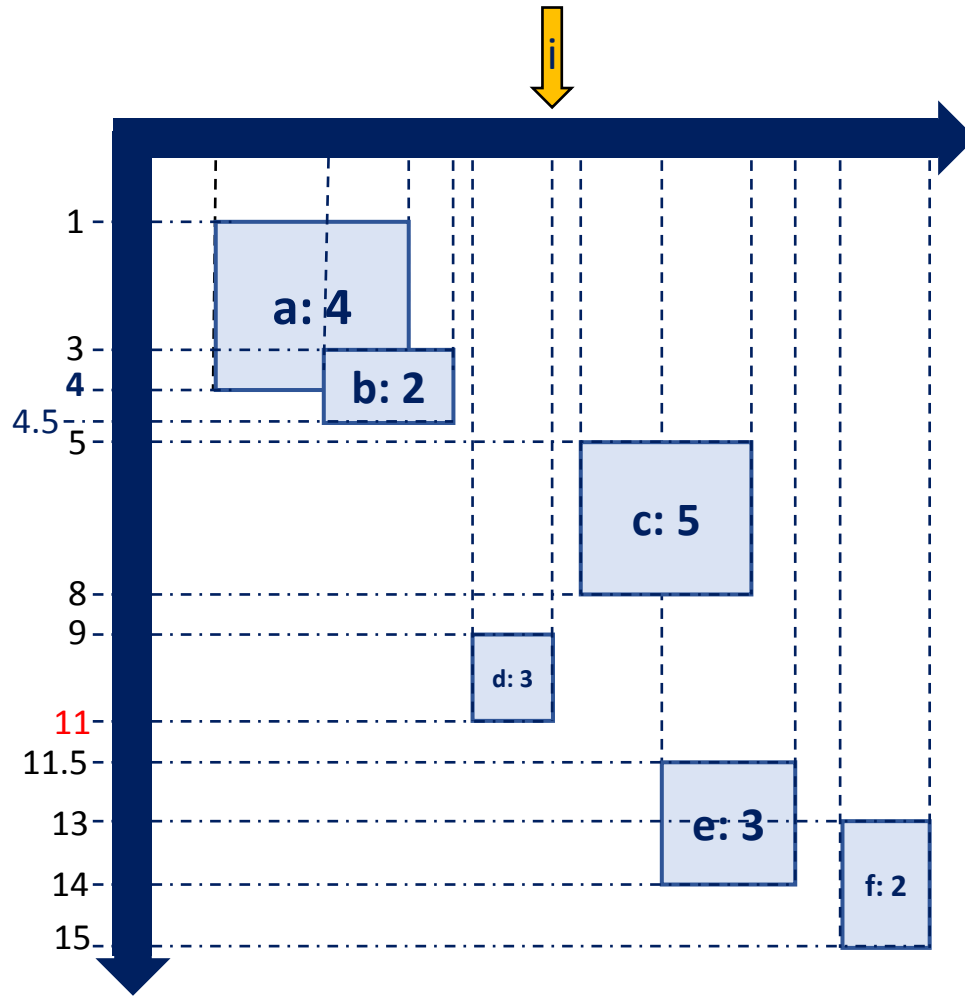
search  $L$  for the last triple where  $l_j \leq l_k$

if  $V(j) < V(k)$ , then

Insert the triple  $(l_k, V(k), k)$  into  $L$ , in the proper location to keep the triples sorted by their  $l$  values.

Delete from  $L$  the triple for every rectangle  $j'$  where  $l_{j'} \geq l_k$  and  $V(j') < V(k)$ .

# The two-dimensional chain algorithm



$V$	a	b	c	d	e	f
	4	2		7		

$L$	$l_j$	4
	$V(j)$	4
	$j$	a

For  $i$  from 1 to  $2r$  do

If  $I[i]$  is the left end of a rectangle, say rectangle  $k$ , then

search  $L$  for the first triple (largest  $l_j$ ) where  $l_j < h_k$

set  $V(k) = w(k) + V(j)$

If  $I[i]$  is the right end of a rectangle  $k$ , then

search  $L$  for the last triple where  $l_j \leq l_k$

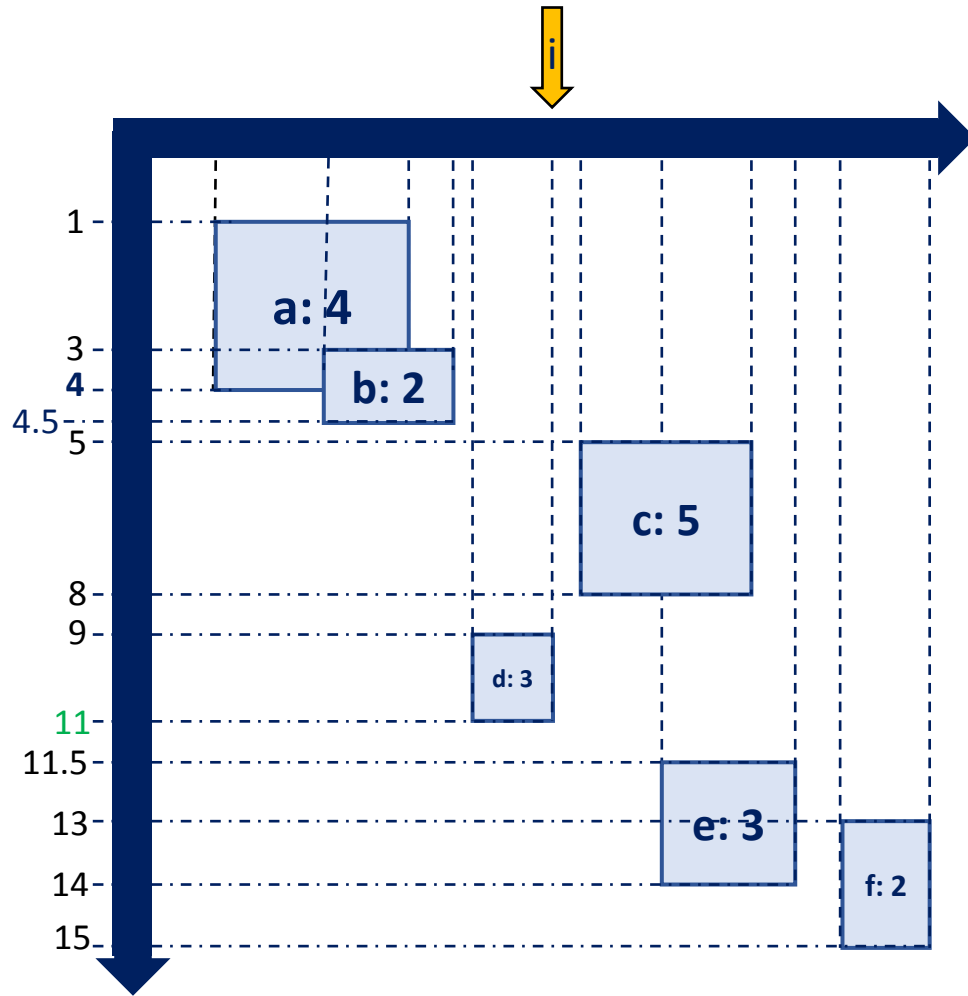
if  $V(j) < V(k)$ , then

Insert the triple  $(l_k, V(k), k)$  into  $L$ , in the proper location to keep the triples sorted by their  $l$  values.

Delete from  $L$  the triple for every rectangle  $j'$  where  $l_{j'} \geq l_k$  and  $V(j') < V(k)$ .



# The two-dimensional chain algorithm



$V$	a	b	c	d	e	f
	4	2		7		

$L$	$l_j$	4	11
	$V(j)$	4	7
	$j$	a	d

For  $i$  from 1 to  $2r$  do

If  $I[i]$  is the left end of a rectangle, say rectangle  $k$ , then

search  $L$  for the first triple (largest  $l_j$ ) where  $l_j < h_k$

set  $V(k) = w(k) + V(j)$

If  $I[i]$  is the right end of a rectangle  $k$ , then

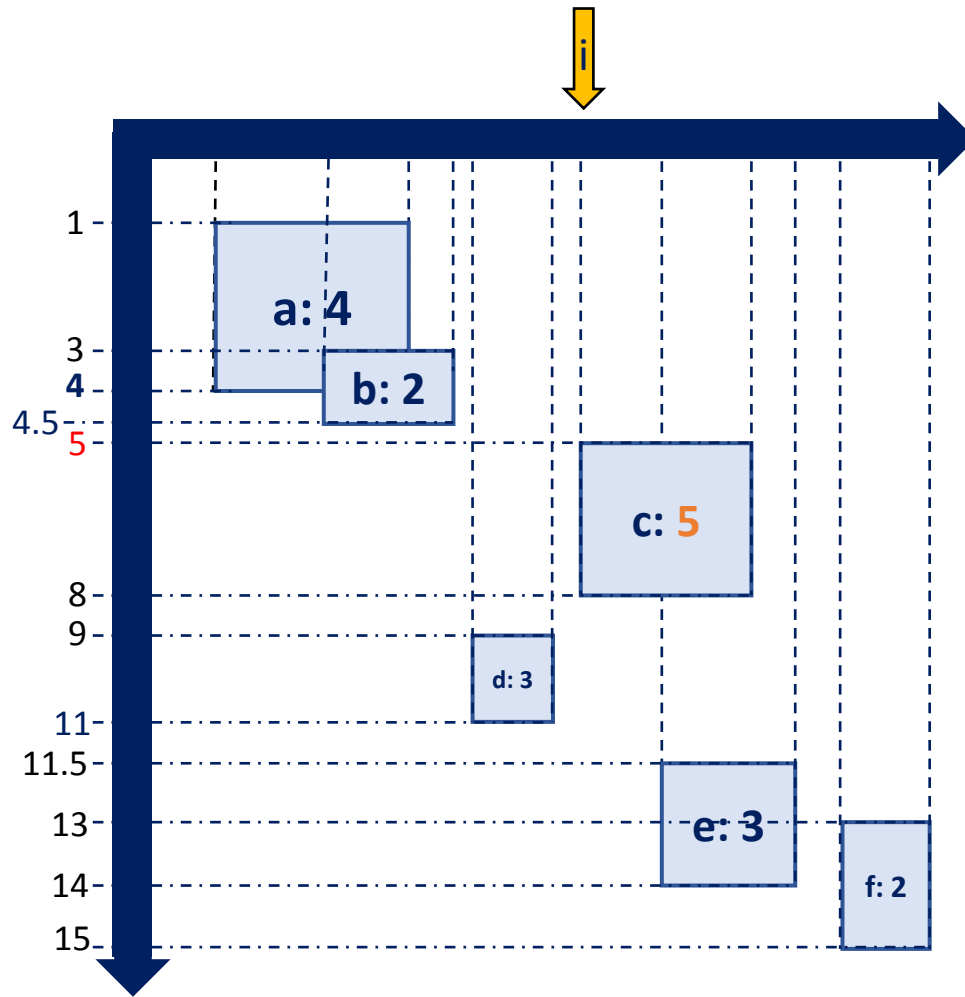
search  $L$  for the last triple where  $l_j \leq l_k$

if  $V(j) < V(k)$ , then

Insert the triple  $(l_k, V(k), k)$  into  $L$ , in the proper location to keep the triples sorted by their  $l$  values.

Delete from  $L$  the triple for every rectangle  $j'$  where  $l_{j'} \geq l_k$  and  $V(j') < V(k)$ .

# The two-dimensional chain algorithm



$V$	a	b	c	d	e	f
	4	2	9	7		

$L$	$l_j$	4	11
	$V(j)$	4	7
	$j$	a	d

For  $i$  from 1 to  $2r$  do

If  $l[i]$  is the left end of a rectangle  $k$ , then

search  $L$  for the first triple (largest  $l_j$ ) where  $l_j < h_k$

set  $V(k) = w(k) + V(j)$

If  $l[i]$  is the right end of a rectangle  $k$ , then

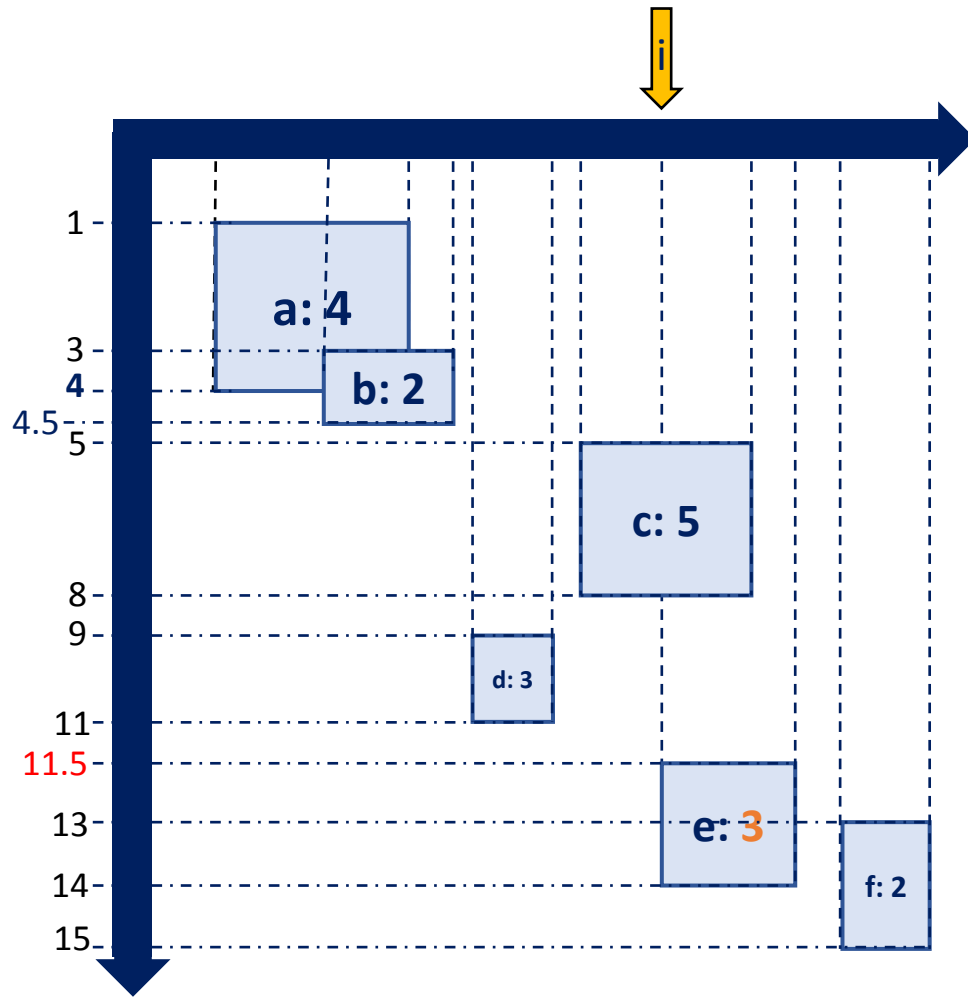
search  $L$  for the last triple where  $l_j \leq l_k$

if  $V(j) < V(k)$ , then

Insert the triple  $(l_k, V(k), k)$  into  $L$ , in the proper location to keep the triples sorted by their  $l$  values.

Delete from  $L$  the triple for every rectangle  $j'$  where  $l_{j'} \geq l_k$  and  $V(j') < V(k)$ .

# The two-dimensional chain algorithm



$V$	a	b	c	d	e	f
	4	2	9	7	10	

$L$	$l_j$	4	11
	$V(j)$	4	7
	$j$	a	d

For  $i$  from 1 to  $2r$  do

If  $l[i]$  is the left end of a rectangle, say rectangle  $k$ , then

search  $L$  for the first triple (largest  $l_j$ ) where  $l_j < h_k$

set  $V(k) = w(k) + V(j)$

If  $l[i]$  is the right end of a rectangle  $k$ , then

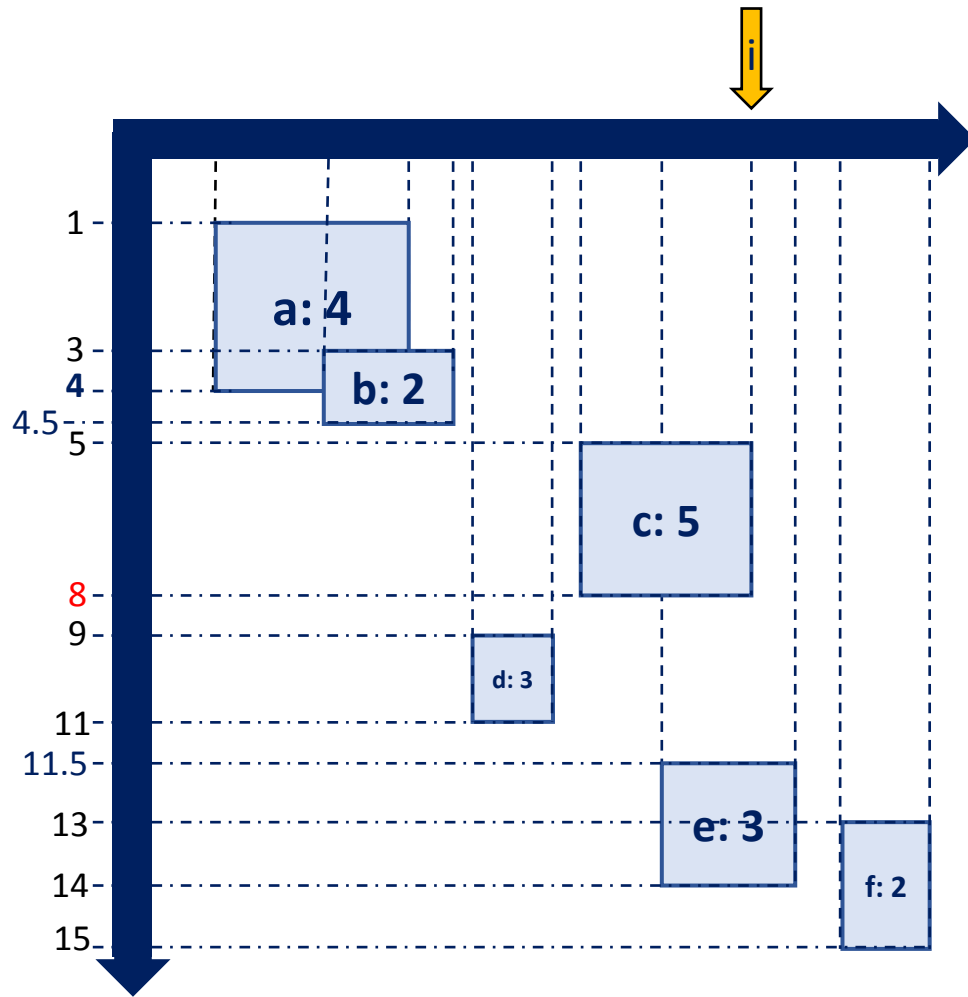
search  $L$  for the last triple where  $l_j \leq l_k$

if  $V(j) < V(k)$ , then

Insert the triple  $(l_k, V(k), k)$  into  $L$ , in the proper location to keep the triples sorted by their  $l$  values.

Delete from  $L$  the triple for every rectangle  $j'$  where  $l_{j'} \geq l_k$  and  $V(j') < V(k)$ .

# The two-dimensional chain algorithm



$V$	a	b	c	d	e	f
	4	2	9	7	10	

$L$	$l_j$	4	11
	$V(j)$	4	7
	$j$	a	d

For  $i$  from 1 to  $2r$  do

If  $l[i]$  is the left end of a rectangle, say rectangle  $k$ , then

search  $L$  for the first triple (largest  $l_j$ ) where  $l_j < h_k$

set  $V(k) = w(k) + V(j)$

If  $l[i]$  is the right end of a rectangle  $k$ , then

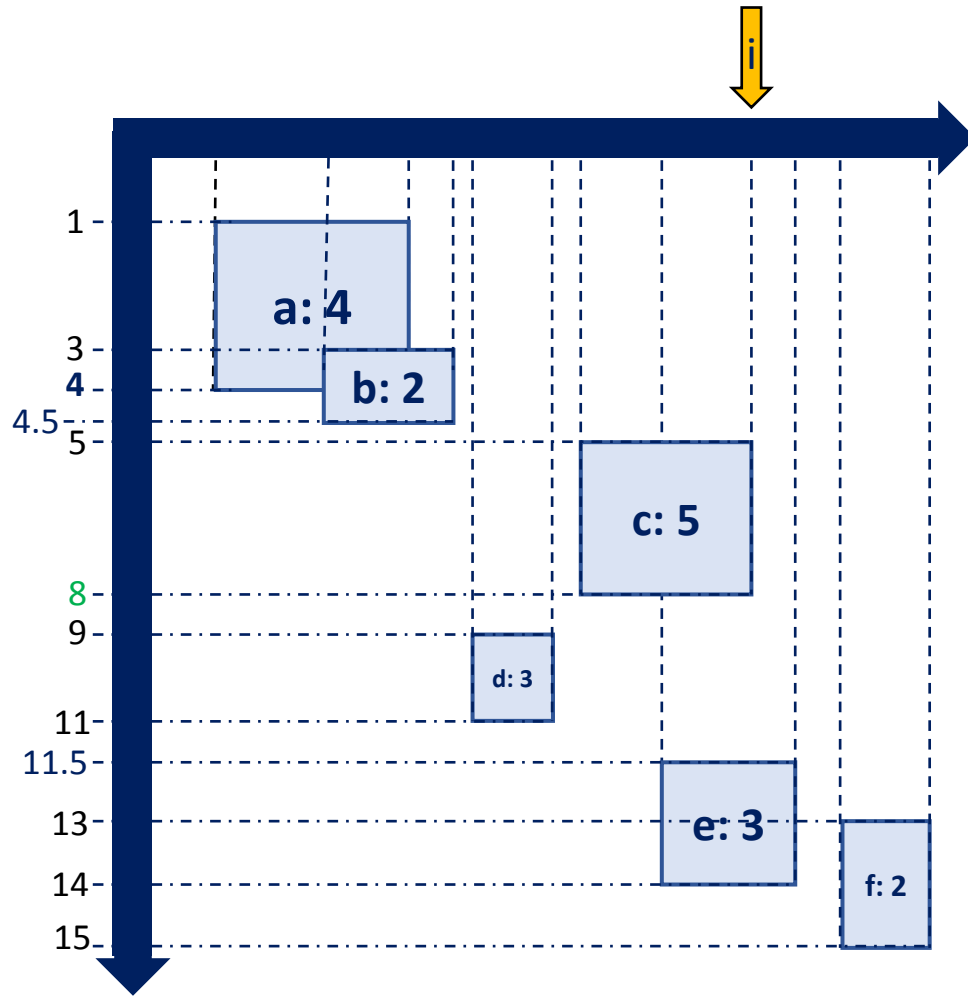
search  $L$  for the last triple where  $l_j \leq l_k$

if  $V(j) < V(k)$ , then

Insert the triple  $(l_k, V(k), k)$  into  $L$ , in the proper location to keep the triples sorted by their  $l$  values.

Delete from  $L$  the triple for every rectangle  $j'$  where  $l_{j'} \geq l_k$  and  $V(j') < V(k)$ .

# The two-dimensional chain algorithm



$V$	a	b	c	d	e	f
	4	2	9	7	10	

$L$	$l_j$	4	8	11
	$V(j)$	4	9	7
	$j$	a	c	d

For  $i$  from 1 to  $2r$  do

If  $I[i]$  is the left end of a rectangle, say rectangle  $k$ , then

search  $L$  for the first triple (largest  $l_j$ ) where  $l_j < h_k$

set  $V(k) = w(k) + V(j)$

If  $I[i]$  is the right end of a rectangle  $k$ , then

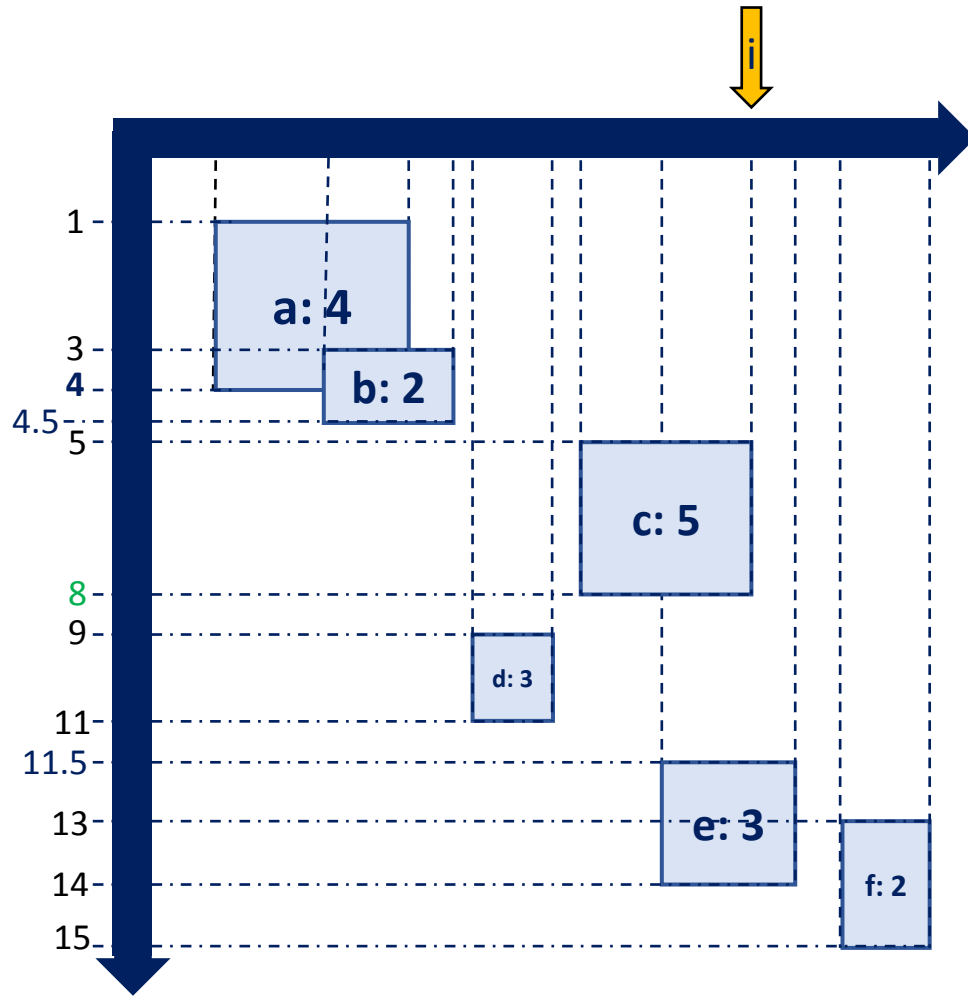
search  $L$  for the last triple where  $l_j \leq l_k$

if  $V(j) < V(k)$ , then

Insert the triple  $(l_k, V(k), k)$  into  $L$ , in the proper location to keep the triples sorted by their  $l$  values.

Delete from  $L$  the triple for every rectangle  $j'$  where  $l_{j'} \geq l_k$  and  $V(j') < V(k)$ .

# The two-dimensional chain algorithm



$V$	a	b	c	d	e	f
	4	2	9	7	10	

$L$	$l_j$	4	8
	$V(j)$	4	9
	$j$	a	c

For  $i$  from 1 to  $2r$  do

If  $I[i]$  is the left end of a rectangle, say rectangle  $k$ , then

search  $L$  for the first triple (largest  $l_j$ ) where  $l_j < h_k$

set  $V(k) = w(k) + V(j)$

If  $I[i]$  is the right end of a rectangle  $k$ , then

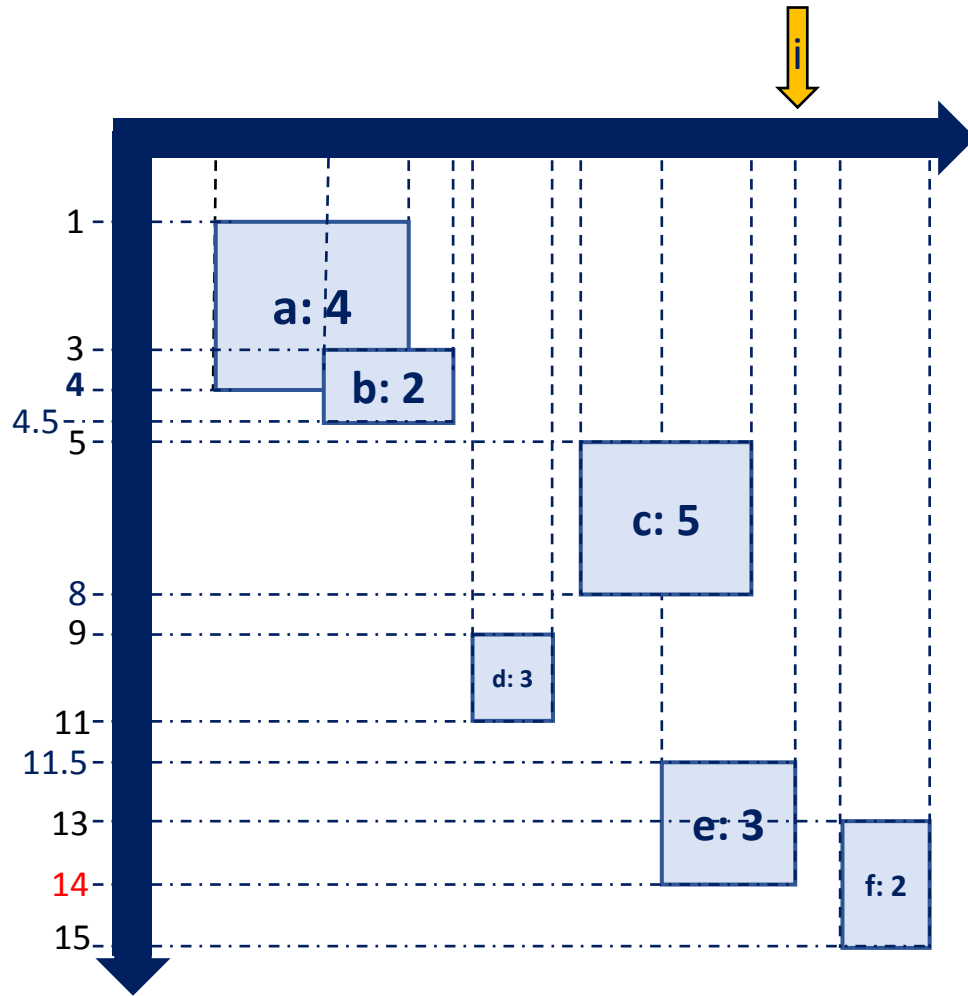
search  $L$  for the last triple where  $l_j \leq l_k$

if  $V(j) < V(k)$ , then

Insert the triple  $(l_k, V(k), k)$  into  $L$ , in the proper location to keep the triples sorted by their  $l$  values.

Delete from  $L$  the triple for every rectangle  $j'$  where  $l_{j'} \geq l_k$  and  $V(j') < V(k)$ .

# The two-dimensional chain algorithm



$V$	a	b	c	d	e	f
	4	2	9	7	10	

$L$	$l_j$	4	8
	$V(j)$	4	9
	$j$	a	c

For  $i$  from 1 to  $2r$  do

If  $I[i]$  is the left end of a rectangle, say rectangle  $k$ , then

search  $L$  for the first triple (largest  $l_j$ ) where  $l_j < h_k$

set  $V(k) = w(k) + V(j)$

If  $I[i]$  is the right end of a rectangle  $k$ , then

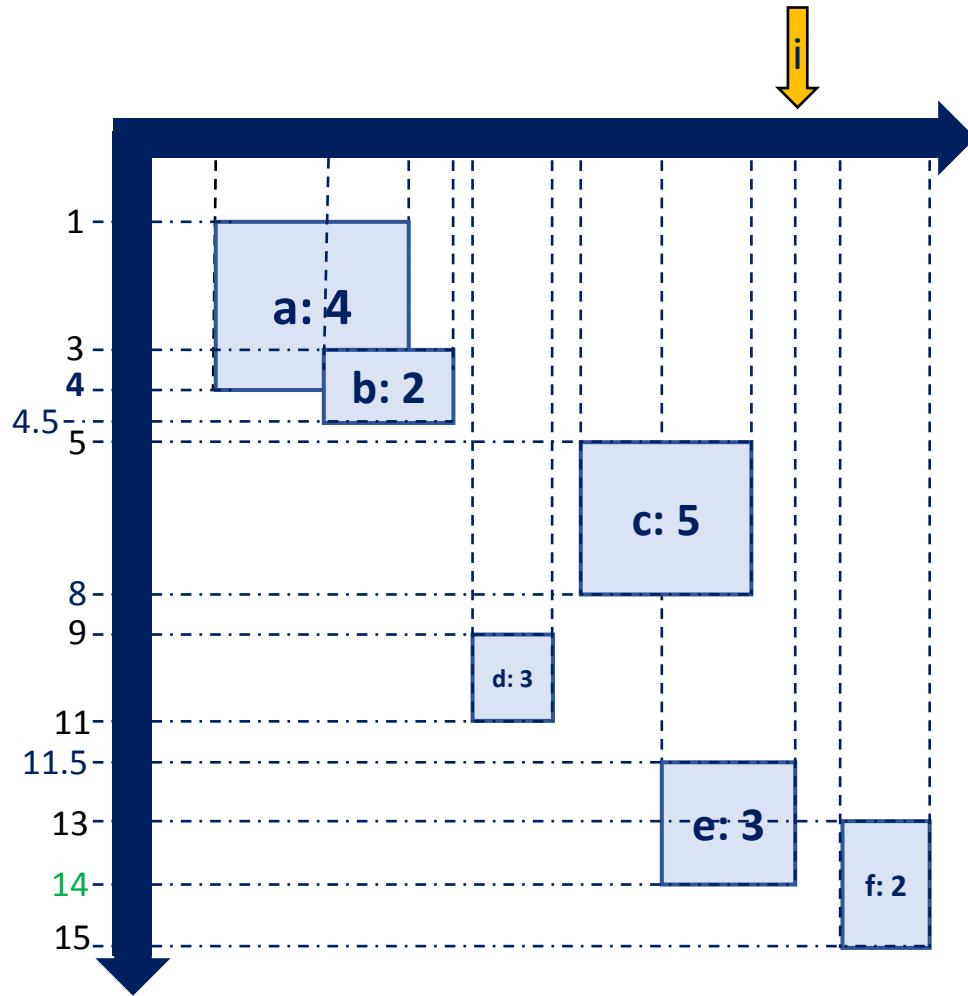
search  $L$  for the last triple where  $l_j \leq l_k$

if  $V(j) < V(k)$ , then

Insert the triple  $(l_k, V(k), k)$  into  $L$ , in the proper location to keep the triples sorted by their  $l$  values.

Delete from  $L$  the triple for every rectangle  $j'$  where  $l_{j'} \geq l_k$  and  $V(j') < V(k)$ .

# The two-dimensional chain algorithm



$V$	a	b	c	d	e	f
	4	2	9	7	10	

$L$	$l_j$	4	8	14
	$V(j)$	4	9	10
	$j$	a	c	e

For  $i$  from 1 to  $2r$  do

If  $I[i]$  is the left end of a rectangle, say rectangle  $k$ , then

search  $L$  for the first triple (largest  $l_j$ ) where  $l_j < h_k$

set  $V(k) = w(k) + V(j)$

If  $I[i]$  is the right end of a rectangle  $k$ , then

search  $L$  for the last triple where  $l_j \leq l_k$

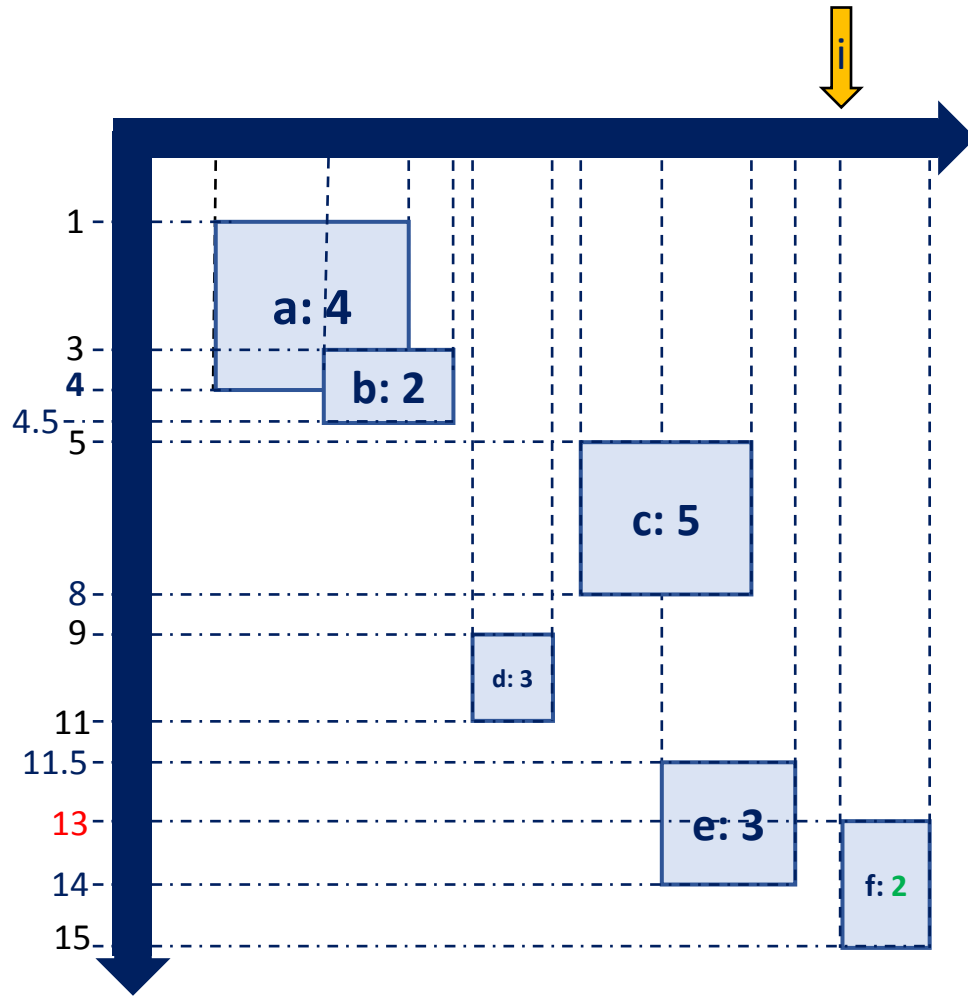
if  $V(j) < V(k)$ , then

Insert the triple  $(l_k, V(k), k)$  into  $L$ , in the proper location to keep the triples sorted by their  $l$  values.

Delete from  $L$  the triple for every rectangle  $j'$  where  $l_{j'} \geq l_k$  and  $V(j') < V(k)$ .



# The two-dimensional chain algorithm



$V$	a	b	c	d	e	f
	4	2	9	7	10	11

$L$	$l_j$	4	8	14
	$V(j)$	4	9	10
	$j$	a	c	e

For  $i$  from 1 to  $2r$  do

If  $l[i]$  is the left end of a rectangle  $k$ , then

search  $L$  for the first triple (largest  $l_j$ ) where  $l_j < h_k$

set  $V(k) = w(k) + V(j)$

If  $l[i]$  is the right end of a rectangle  $k$ , then

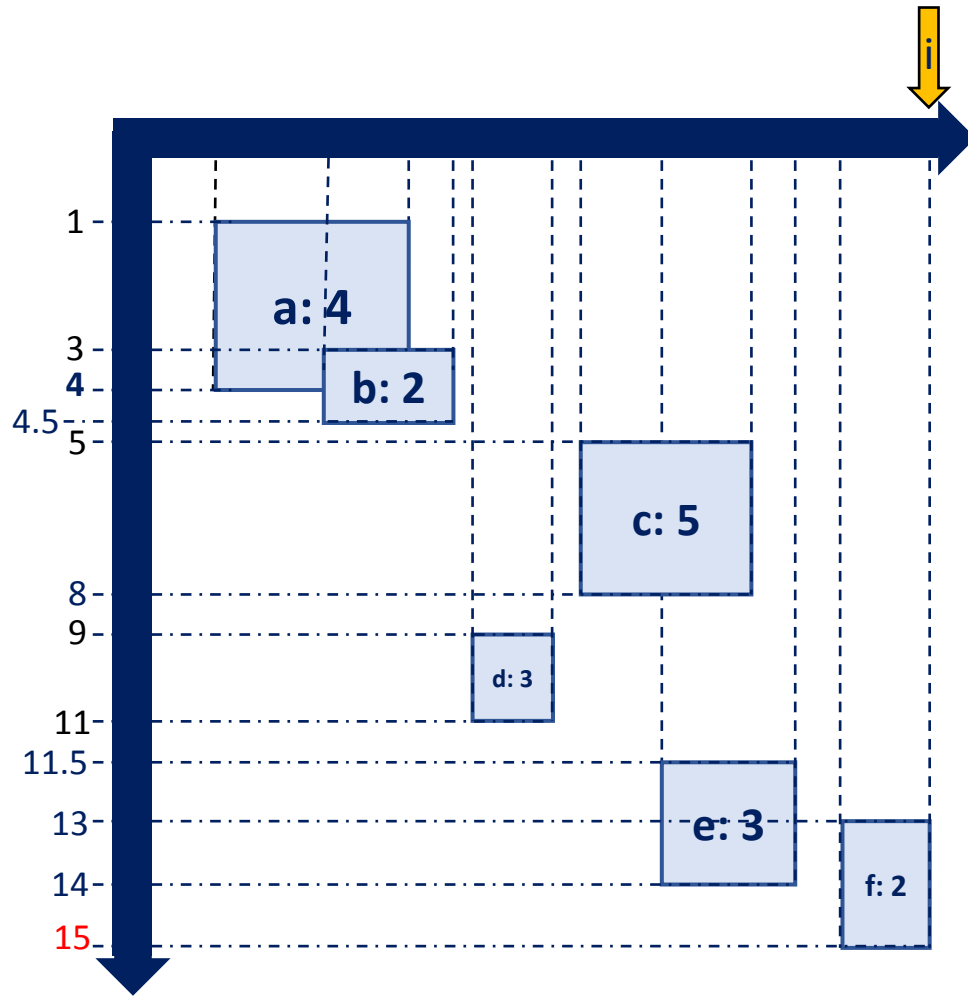
search  $L$  for the last triple where  $l_j \leq l_k$

if  $V(j) < V(k)$ , then

Insert the triple  $(l_k, V(k), k)$  into  $L$ , in the proper location to keep the triples sorted by their  $l$  values.

Delete from  $L$  the triple for every rectangle  $j'$  where  $l_{j'} \geq l_k$  and  $V(j') < V(k)$ .

# The two-dimensional chain algorithm



$V$	a	b	c	d	e	f
	4	2	9	7	10	11

$L$	$l_j$	4	8	14
	$V(j)$	4	9	10
	$j$	a	c	e

For  $i$  from 1 to  $2r$  do

If  $l[i]$  is the left end of a rectangle, say rectangle  $k$ , then

search  $L$  for the first triple (largest  $l_j$ ) where  $l_j < h_k$

set  $V(k) = w(k) + V(j)$

If  $l[i]$  is the right end of a rectangle  $k$ , then

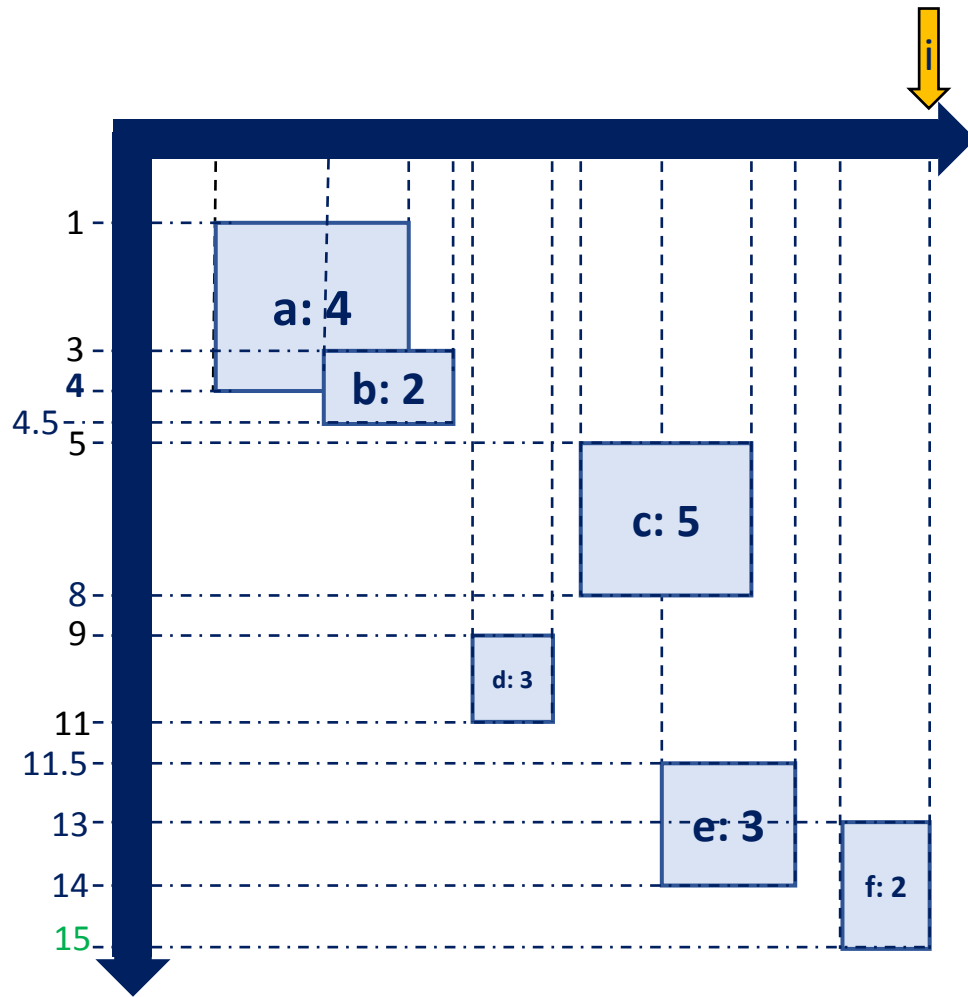
search  $L$  for the last triple where  $l_j \leq l_k$

if  $V(j) < V(k)$ , then

Insert the triple  $(l_k, V(k), k)$  into  $L$ , in the proper location to keep the triples sorted by their  $l$  values.

Delete from  $L$  the triple for every rectangle  $j'$  where  $l_{j'} \geq l_k$  and  $V(j') < V(k)$ .

# The two-dimensional chain algorithm



$V$	a	b	c	d	e	f
	4	2	9	7	10	11

$L$	$l_j$	4	8	14	15
	$V(j)$	4	9	10	11
	$j$	a	c	e	f

For  $i$  from 1 to  $2r$  do

If  $l[i]$  is the left end of a rectangle, say rectangle  $k$ , then

search  $L$  for the first triple (largest  $l_j$ ) where  $l_j < h_k$

set  $V(k) = w(k) + V(j)$

If  $l[i]$  is the right end of a rectangle  $k$ , then

search  $L$  for the last triple where  $l_j \leq l_k$

if  $V(j) < V(k)$ , then

Insert the triple  $(l_k, V(k), k)$  into  $L$ , in the proper location to keep the triples sorted by their  $l$  values.

Delete from  $L$  the triple for every rectangle  $j'$  where  $l_{j'} \geq l_k$  and  $V(j') < V(k)$ .

# Time Analysis

---

- Sorting list / takes  $O(r \log r)$
- Keep  $L$  as a balanced binary search tree sorted by  $l_j$  e.g.) AVL tree
- Searching  $L$ :
  - Either for  $l_j < h_k$  or for  $l_j \leq l_k$  takes  $O(\log r)$  time
  - The total of search is  $O(r \log r)$
- Inserting in  $L$ :
  - Insertion operation takes  $O(\log r)$  time
  - For all insertions,  $O(r \log r)$
- Deleting from  $L$ :
  - Deletion operation takes  $O(\log r)$  time
  - For all deletions,  $O(r \log r)$

# Theorem 13.3.1

---

- An optimal chain can be found in  $O(r \log r)$  time.

**The End**

**Thank you**