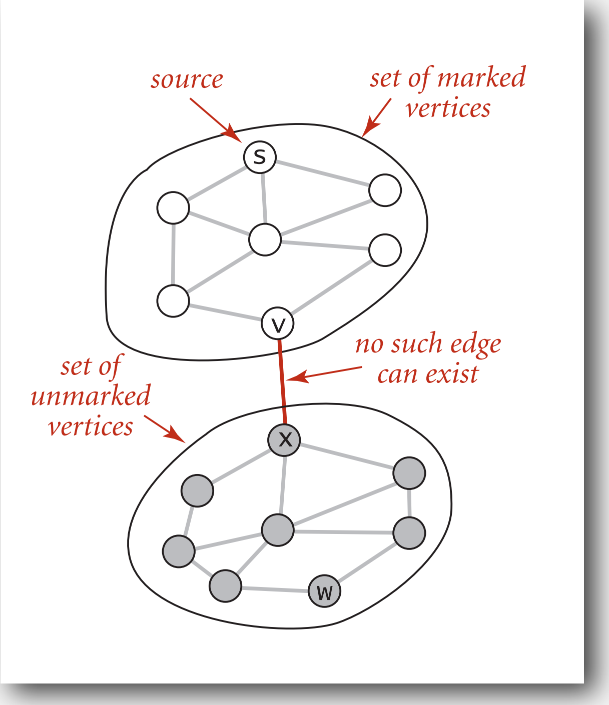
Final review

Depth-first search vs. Breadth-first search

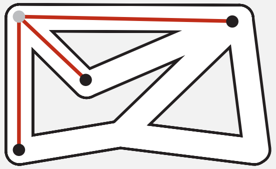
Depth-first search. Put unvisited vertices on a stack.

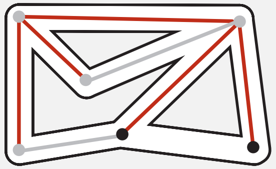
Breadth-first search. Put unvisited vertices on a queue.

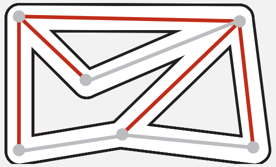


Connected component: maximal set of connected vertices.

DFS marks all vertices connected to s in time proportional to the sum of their degrees.







BFS finds path from s to t that uses fewest number of edges.

Intuition. BFS examines vertices in increasing distance from s.

‣ directed graphs

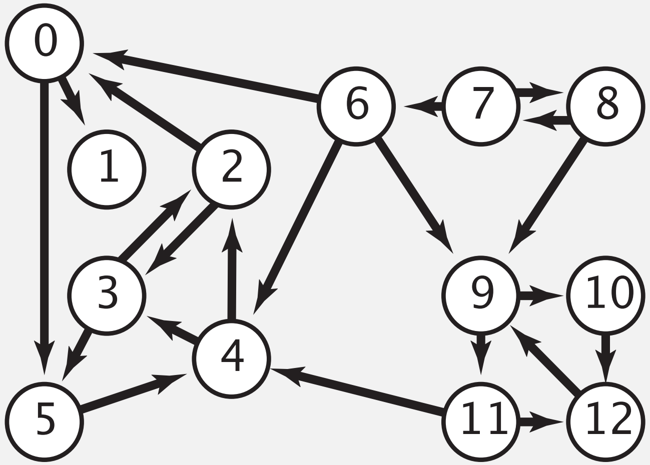
Depth-first search in digraphs

Same method as for undirected graphs.

- Every undirected graph is a digraph (with edges in both directions).
- DFS is a digraph algorithm.

DFS (to visit a vertex v)

```
Mark v as visited.
Recursively visit all unmarked
        vertices w adjacent from v.
```



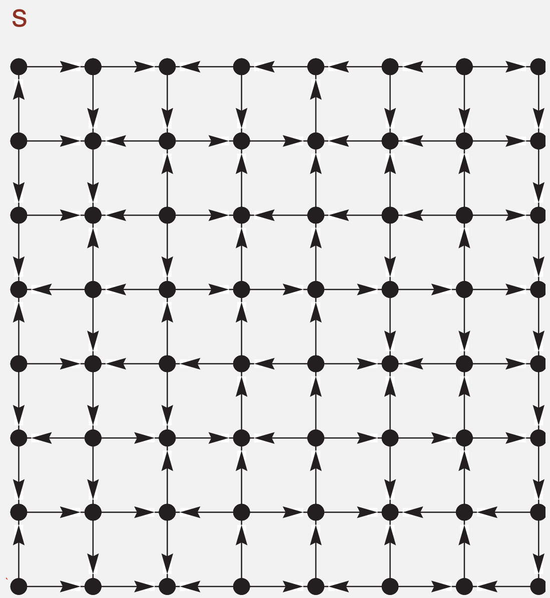
Breadth-first search in digraphs

Same method as for undirected graphs.

- Every undirected graph is a digraph (with edges in both directions).
- BFS is a digraph algorithm.

```
BFS (from source vertex s)

Put s onto a FIFO queue, and mark s as visited.
Repeat until the queue is empty:
  - remove the least recently added vertex v
  - for each unmarked vertex adjacent from v:
      add to queue and mark as visited..
```



Proposition. BFS computes shortest paths (fewest number of edges).

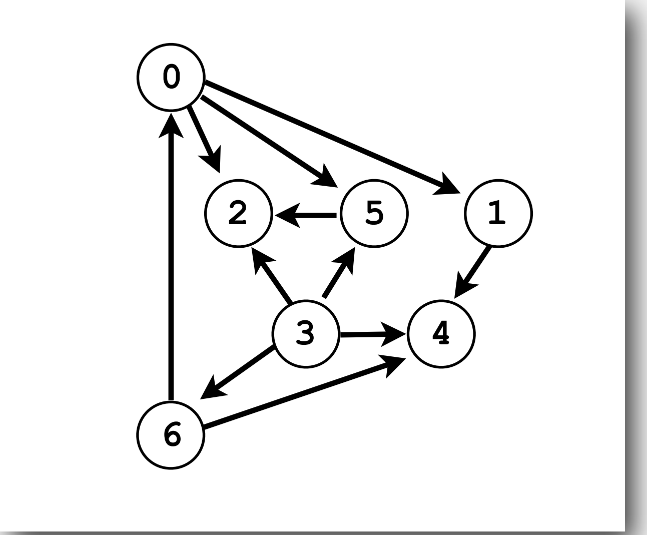
Topological sort

DAG. Directed acyclic graph.

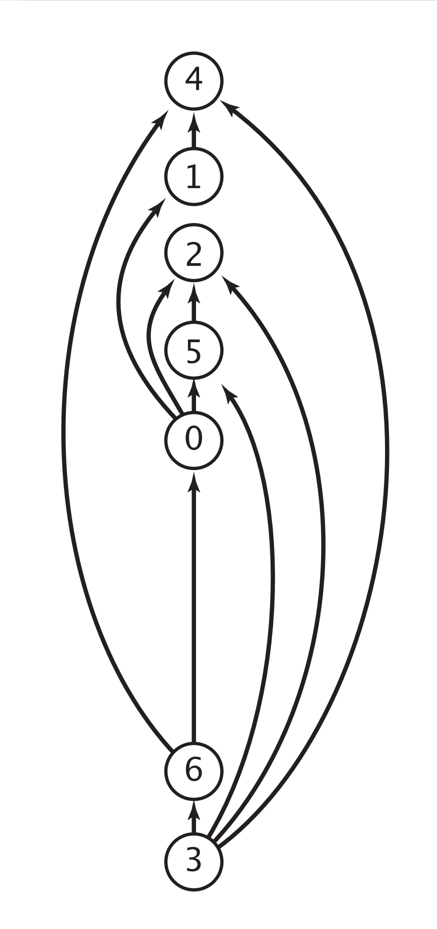
Topological sort. Redraw DAG so all edges point up.

0→5
0→1
3→5
5→4
6→0
1→4
0→2
3→6
3→4
6→4
3→2

directed edges



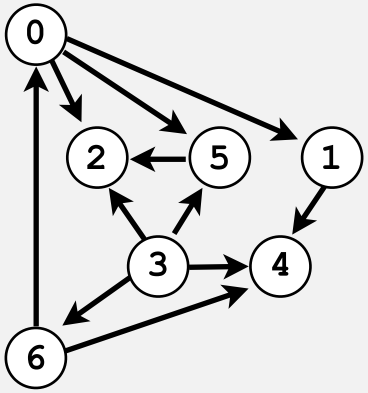
DAG



topological order

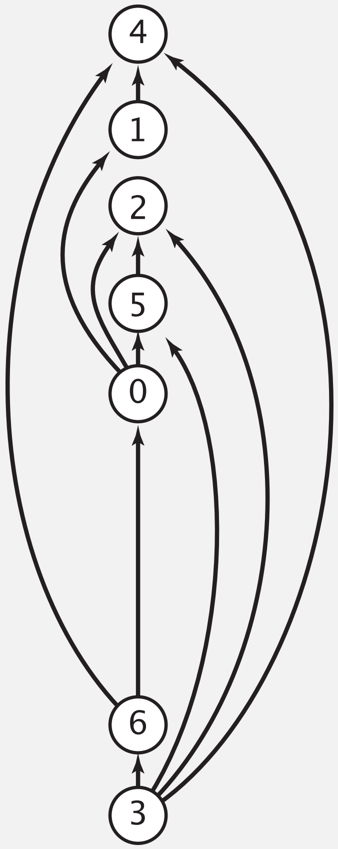
Solution. Reverse DFS Postorder!

Reverse DFS postorder in a DAG



0→5
0→2
0→1
3→6
3→5
3→4
5→4
6→4
6→0
3→2
1→4

```
                     marked[]        reversePost
dfs(0)               1 0 0 0 0 0 0   -
  dfs(1)             1 1 0 0 0 0 0   -
    dfs(4)           1 1 0 0 1 0 0   -
    4 done           1 1 0 0 1 0 0   4
  1 done             1 1 0 0 1 0 0   4 1
  dfs(2)             1 1 1 0 1 0 0   4 1
  2 done             1 1 1 0 1 0 0   4 1 2
  dfs(5)             1 1 1 0 1 1 0   4 1 2
    check 2          1 1 1 0 1 1 0   4 1 2
  5 done             1 1 1 0 1 1 0   4 1 2 5
0 done               1 1 1 0 1 1 0   4 1 2 5 0
check 1              1 1 1 0 1 1 0   4 1 2 5 0
check 2              1 1 1 0 1 1 0   4 1 2 5 0
dfs(3)               1 1 1 1 1 1 0   4 1 2 5 0
  check 2            1 1 1 1 1 1 0   4 1 2 5 0
  check 4            1 1 1 1 1 1 0   4 1 2 5 0
  check 5            1 1 1 1 1 1 0   4 1 2 5 0
  dfs(6)             1 1 1 1 1 1 1   4 1 2 5 0
  6 done             1 1 1 1 1 1 1   4 1 2 5 0 6
3 done               1 1 1 1 1 1 1   4 1 2 5 0 6 3
check 4              1 1 1 1 1 1 0   4 1 2 5 0 6 3
check 5              1 1 1 1 1 1 0   4 1 2 5 0 6 3
check 6              1 1 1 1 1 1 0   4 1 2 5 0 6 3
done                 1 1 1 1 1 1 1   4 1 2 5 0 6 3
```



reverse DFS postorder is a topological order!

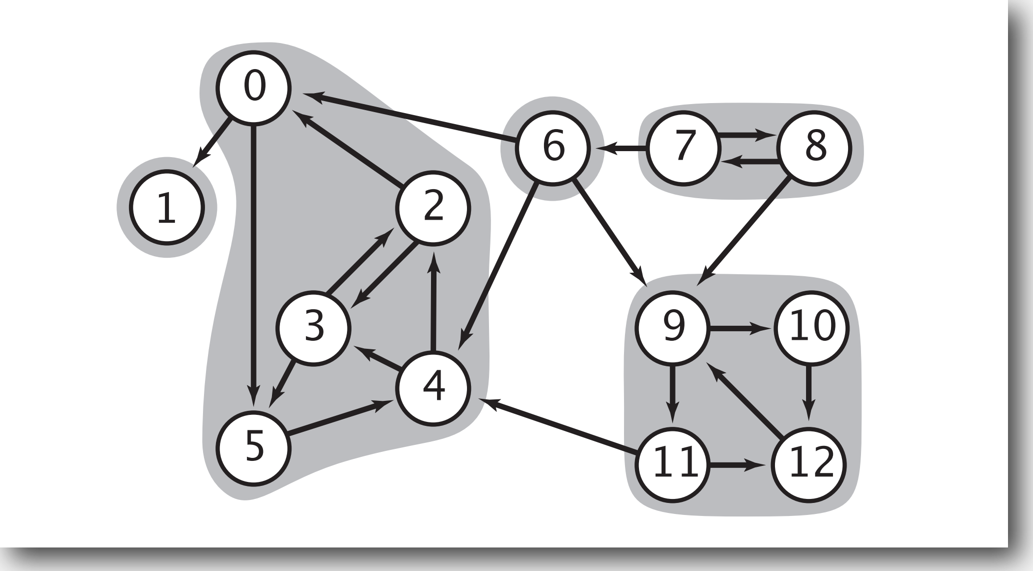
Strongly-connected components

Def. Vertices v and w are **strongly connected** if there is a directed path from v to w **and** a directed path from w to v.

Key property. Strong connectivity is an equivalence relation:

- v is strongly connected to v.
- If v is strongly connected to w, then w is strongly connected to v.
- If v is strongly connected to w and w to x, then v is strongly connected to x.

Def. A strong component is a maximal subset of strongly-connected vertices.

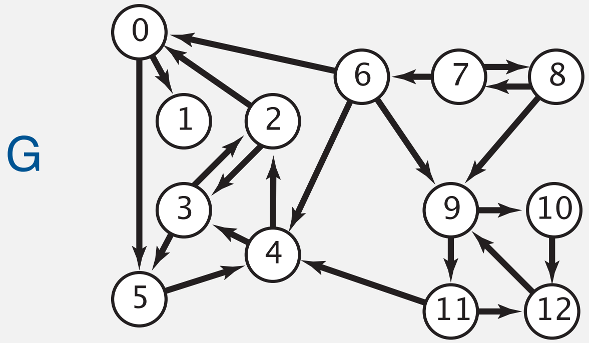


Kosaraju's algorithm

Simple (but mysterious) algorithm for computing strong components.

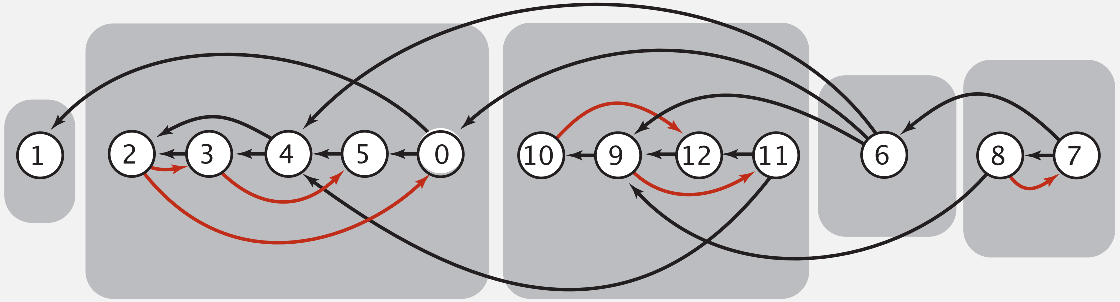
- Run DFS on G^R to compute **reverse postorder**.
- Run DFS on G, considering vertices in order given by first DFS.

DFS in original digraph



check unmarked vertices in the order

1 0 2 4 5 3 11 9 12 10 6 7 8



```
dfs(1)
1 done
```

```
dfs(0)
  dfs(5)
    dfs(4)
      dfs(3)
        check 5
        dfs(2)
          check 0
          check 3
        2 done
      3 done
      check 2
    4 done
  5 done
  check 1
0 done
check 2
check 4
check 5
check 3
```

```
dfs(11)
  check 4
  dfs(12)
    dfs(9)
      check 11
      dfs(10)
        check 12
      10 done
    9 done
  12 done
11 done
check 9
check 12
check 10
```

```
dfs(6)
  check 9
  check 4
  check 0
6 done
```

```
dfs(7)
  check 6
  dfs(8)
    check 7
    check 9
  8 done
7 done
check 8
```

Second DFS gives strong components. (!!)

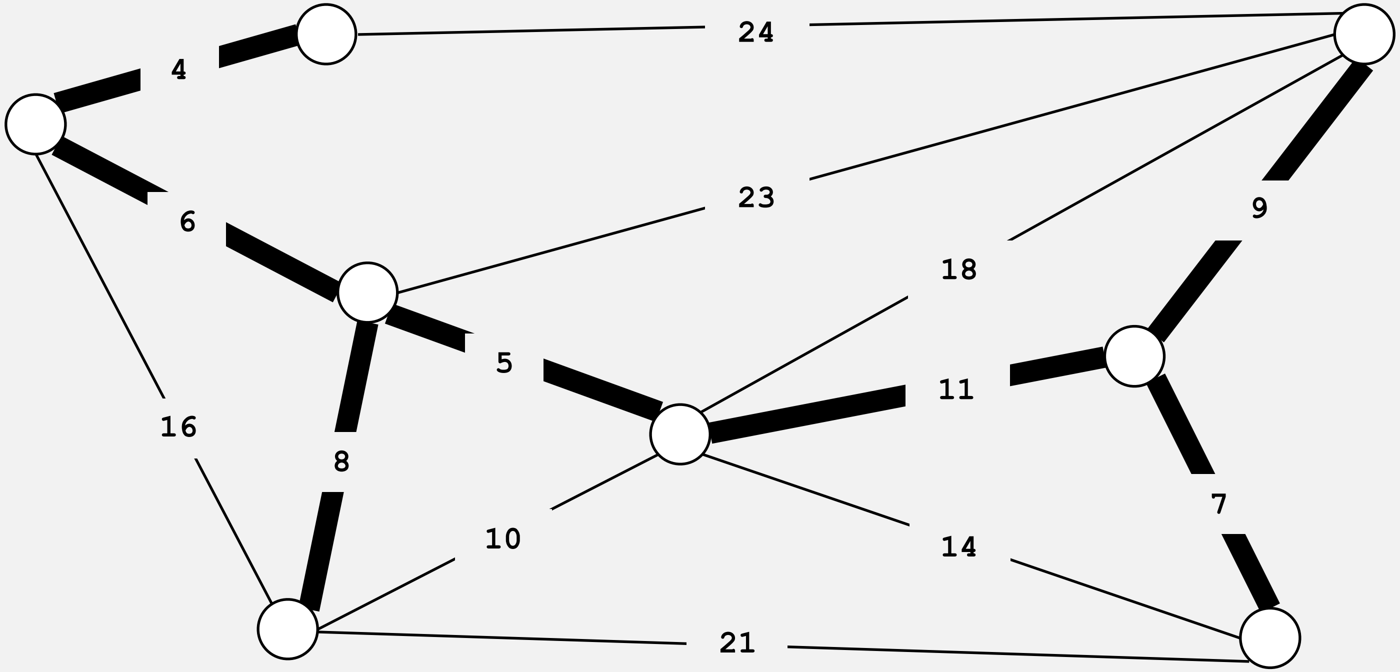
‣ minimum spanning trees

Minimum spanning tree

Given. Undirected graph G with positive edge weights (connected).

Def. A **spanning tree** of G is a subgraph T that is **connected** and **acyclic**.

Goal. Find a min weight spanning tree.



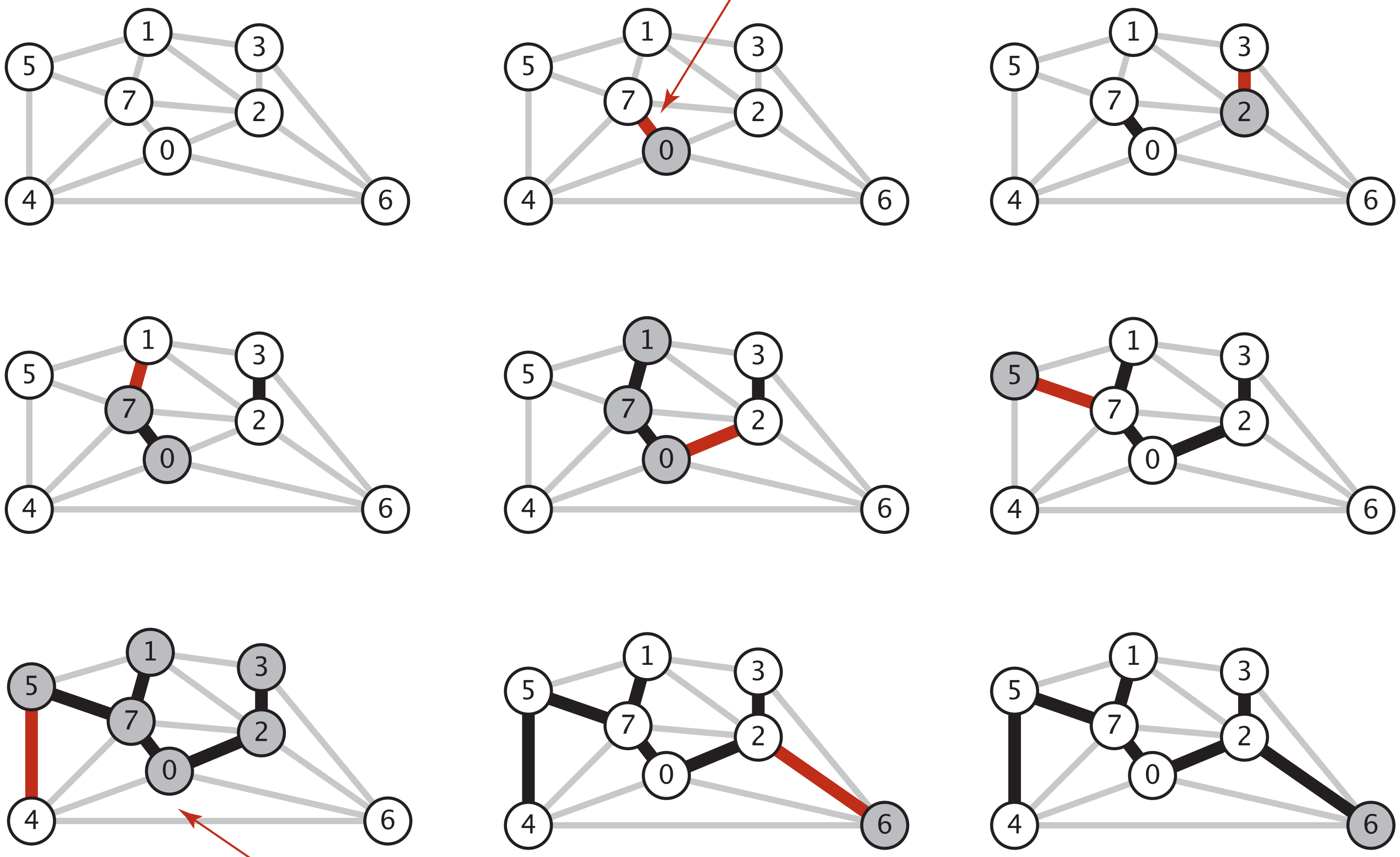
spanning tree T: cost = 50 = 4 + 6 + 8 + 5 + 11 + 9 + 7

Brute force. Try all spanning trees?

Kruskal's algorithm

Kruskal's algorithm. [Kruskal 1956] **Consider edges in ascending order of weight.**

Add the next edge to the tree *T* unless doing so would create a cycle.



next MST edge is red

grey vertices are a cut defined by the vertices connected to one of the red edge's vertices

graph edges sorted by weight

MST edge (black)

0-7	0.16
2-3	0.17
1-7	0.19
0-2	0.26
5-7	0.28
1-3	0.29
1-5	0.32
2-7	0.34
4-5	0.35
1-2	0.36
4-7	0.37
0-4	0.38
6-2	0.40
3-6	0.52
6-0	0.58
6-4	0.93

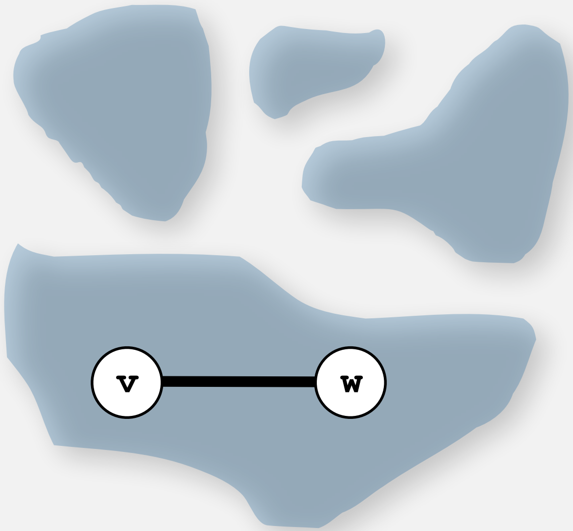
obsolete edge (gray)

Kruskal's algorithm: implementation challenge

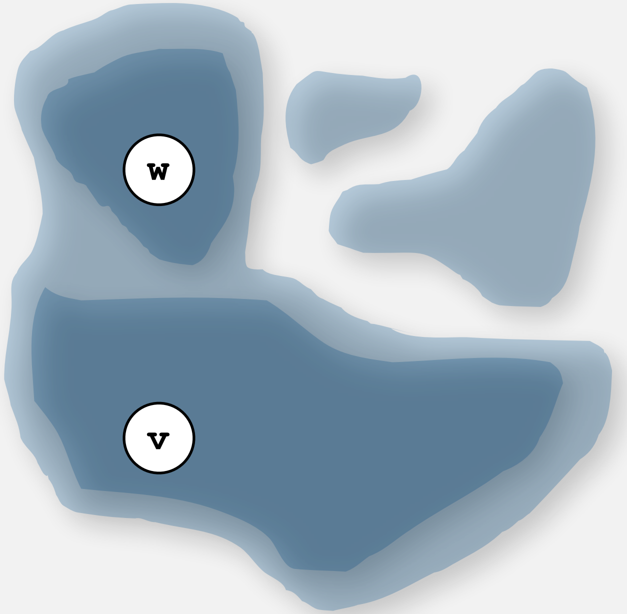
Challenge. Would adding edge v–w to tree T create a cycle? If not, add it.

Efficient solution. Use the **union-find** data structure.

- Maintain a set for each connected component in T.
- If v and w are in same set, then adding v–w would create a cycle.
- To add v–w to T, merge sets containing v and w.



Case 1: adding v–w creates a cycle

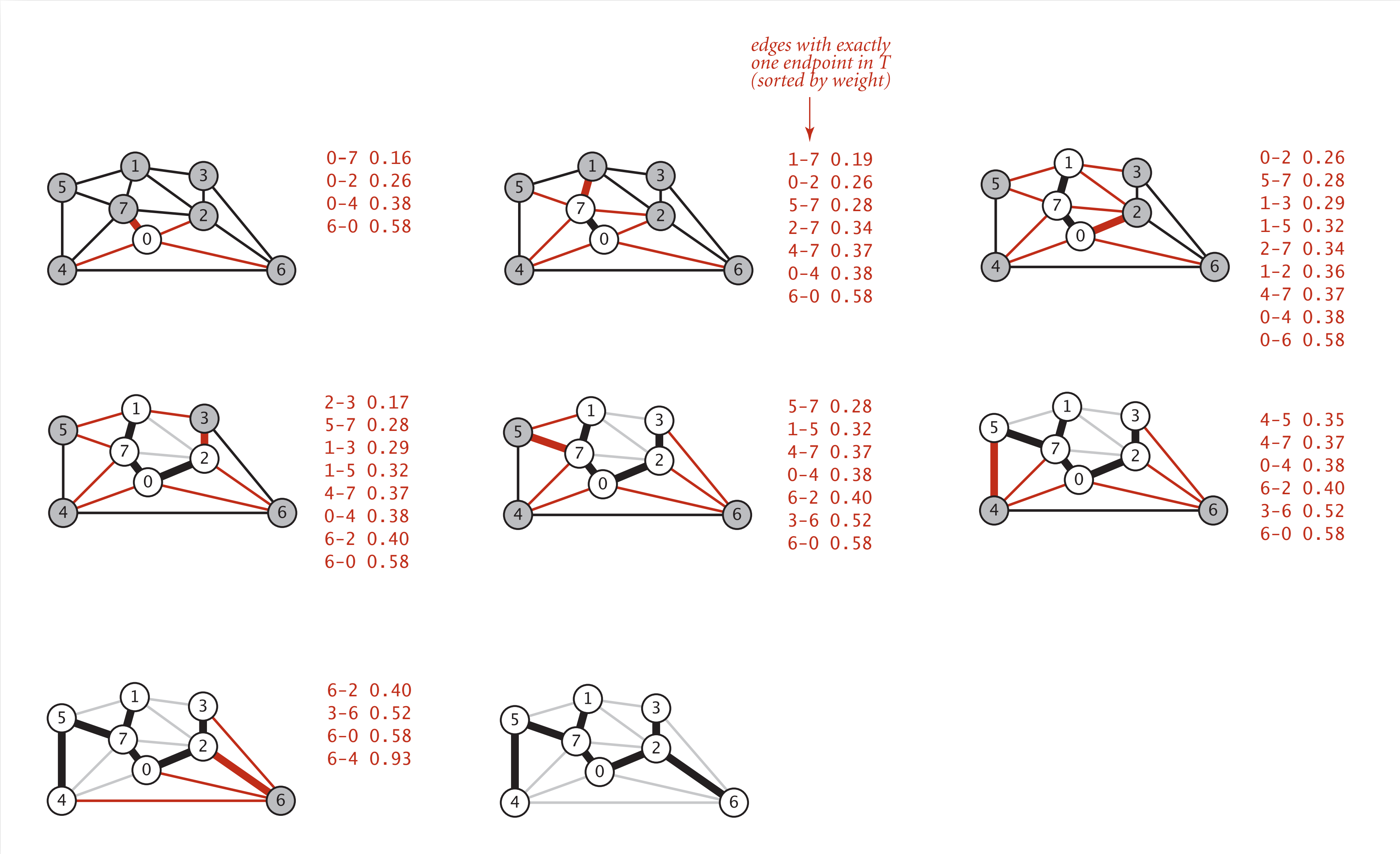


Case 2: add v–w to T and merge sets containing v and w

Prim's algorithm

Prim's algorithm. [Jarník 1930, Dijkstra 1957, Prim 1959]

Start with vertex 0 and **greedily grow tree** T. At each step, add to T the min weight edge with exactly one endpoint in T.



edges with exactly one endpoint in T (sorted by weight)

0-7 0.16
0-2 0.26
0-4 0.38
6-0 0.58

1-7 0.19
0-2 0.26
5-7 0.28
2-7 0.34
4-7 0.37
0-4 0.38
6-0 0.58

0-2 0.26
5-7 0.28
1-3 0.29
1-5 0.32
2-7 0.34
1-2 0.36
4-7 0.37
0-4 0.38
0-6 0.58

2-3 0.17
5-7 0.28
1-3 0.29
1-5 0.32
4-7 0.37
0-4 0.38
6-2 0.40
6-0 0.58

5-7 0.28
1-5 0.32
4-7 0.37
0-4 0.38
6-2 0.40
3-6 0.52
6-0 0.58

4-5 0.35
4-7 0.37
0-4 0.38
6-2 0.40
3-6 0.52
6-0 0.58

6-2 0.40
3-6 0.52
6-0 0.58
6-4 0.93

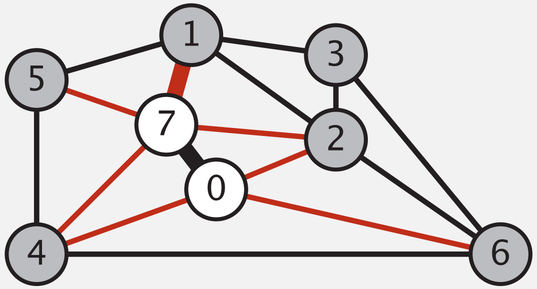
Prim's algorithm: lazy implementation

Challenge. Find the min weight edge with exactly one endpoint in T.

Lazy solution. Maintain a **PQ of edges** with (at least) one endpoint in T.

- Delete min to determine next edge $e = v–w$ to add to T.
- Disregard if both endpoints v and w are in T.
- Otherwise, let v be vertex not in T :
  - add to PQ any edge incident to v (assuming other endpoint not in T)
  - add v to T

1-7 is min weight edge with exactly one endpoint in T



priority queue of crossing edges

1-7	0.19
0-2	0.26
5-7	0.28
2-7	0.34
4-7	0.37
0-4	0.38
6-0	0.58

Prim's algorithm: running time

Proposition. Lazy Prim's algorithm computes the MST in time proportional to $E \log E$ in the worst case.

Pf.

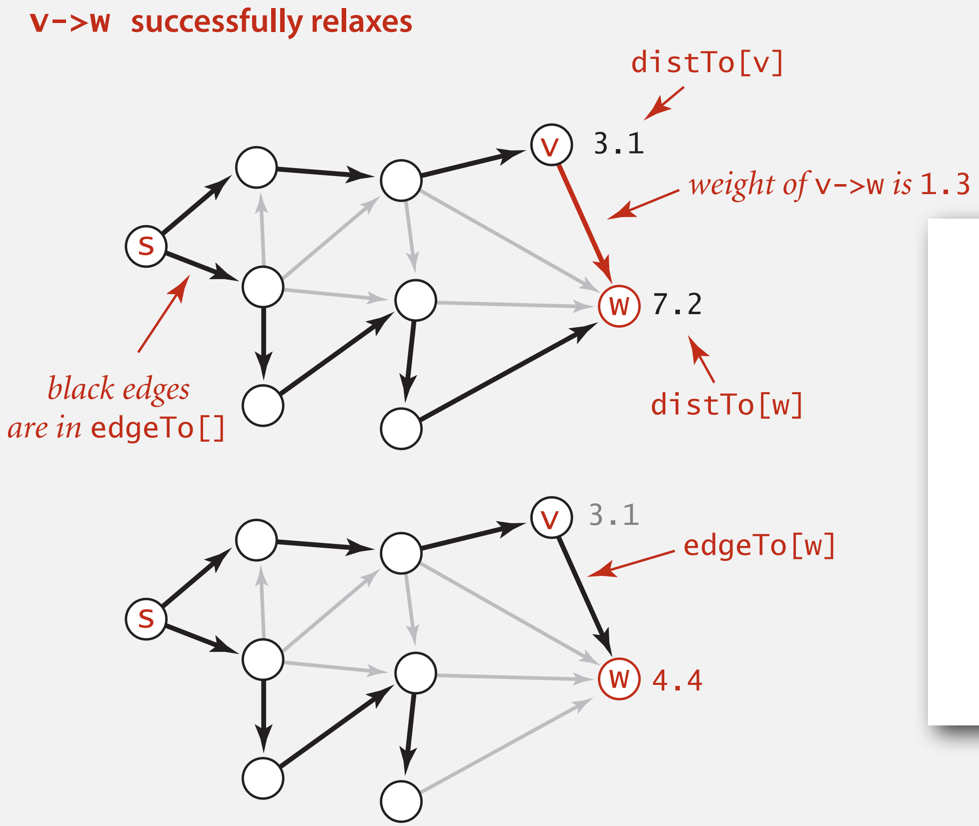
operation	frequency	binary heap
delete min	E	log E
insert	E	log E

▸ shortest path

Edge relaxation

Relax edge $e = v \to w$.

- `distTo[v]` is length of shortest known path from s to v.
- `distTo[w]` is length of shortest known path from s to w.
- `edgeTo[w]` is last edge on shortest known path from s to w.
- If $e = v \to w$ gives shorter path to w through v, update `distTo[w]` and `edgeTo[w]`.

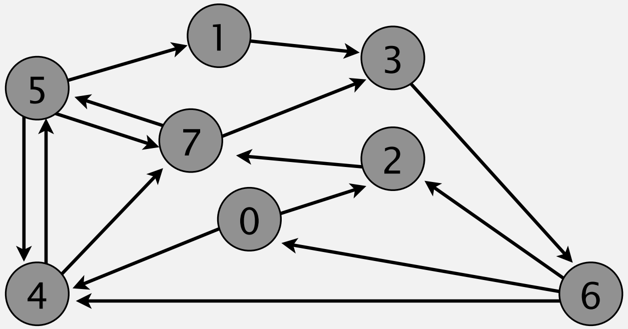


```
private void relax(DirectedEdge e)
{
   int v = e.from(), w = e.to();
   if (distTo[w] > distTo[v] + e.weight())
   {
       distTo[w] = distTo[v] + e.weight();
       edgeTo[w] = e;
   }
}
```

Dijkstra's Algorithm

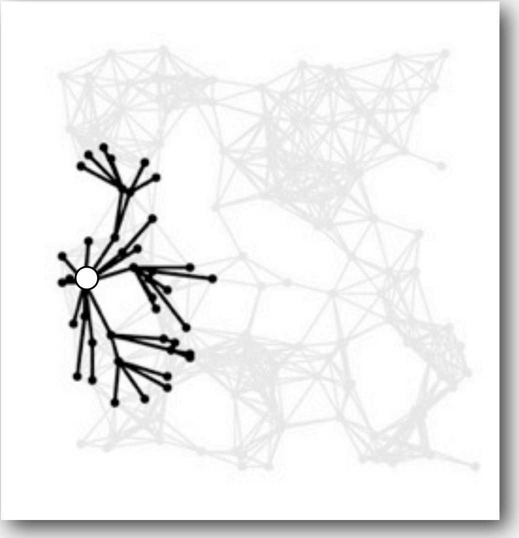
- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges incident from that vertex.

4->5	0.35
5->4	0.35
4->7	0.37
5->7	0.28
7->5	0.28
5->1	0.32
0->4	0.38
0->2	0.26
7->3	0.39
1->3	0.29
2->7	0.34
6->2	0.40
3->6	0.52
6->0	0.58
6->4	0.93

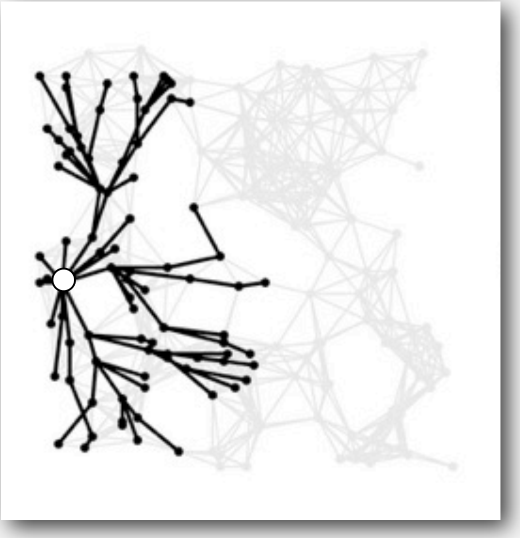


v	distTo[v]	edgeTo[v]
0	0.00	-
1		
2		
3		
4		
5		
6		
7		

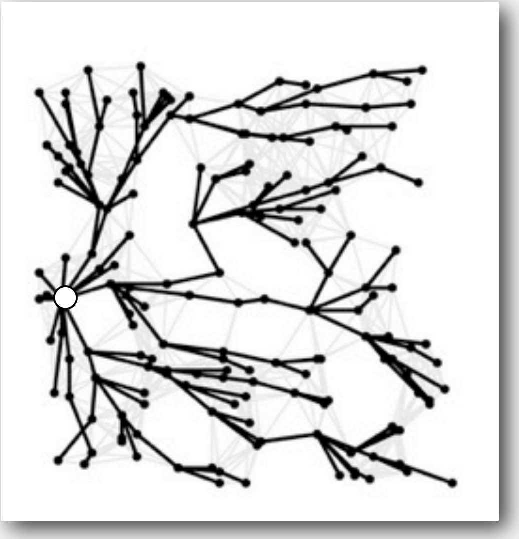
25%



50%



100%



‣ string sorts

Key-indexed counting

Goal. Sort an array `a[]` of N integers between 0 and $R - 1$.

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move records.
- Copy back into original array.

```
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
   count[a[i]+1]++;

for (int r = 0; r < R; r++)
   count[r+1] += count[r];

for (int i = 0; i < N; i++)
   aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
   a[i] = aux[i];
```

copy back

i	a[i]
0	a
1	a
2	b
3	b
4	b
5	c
6	d
7	d
8	e
9	f
10	f
11	f

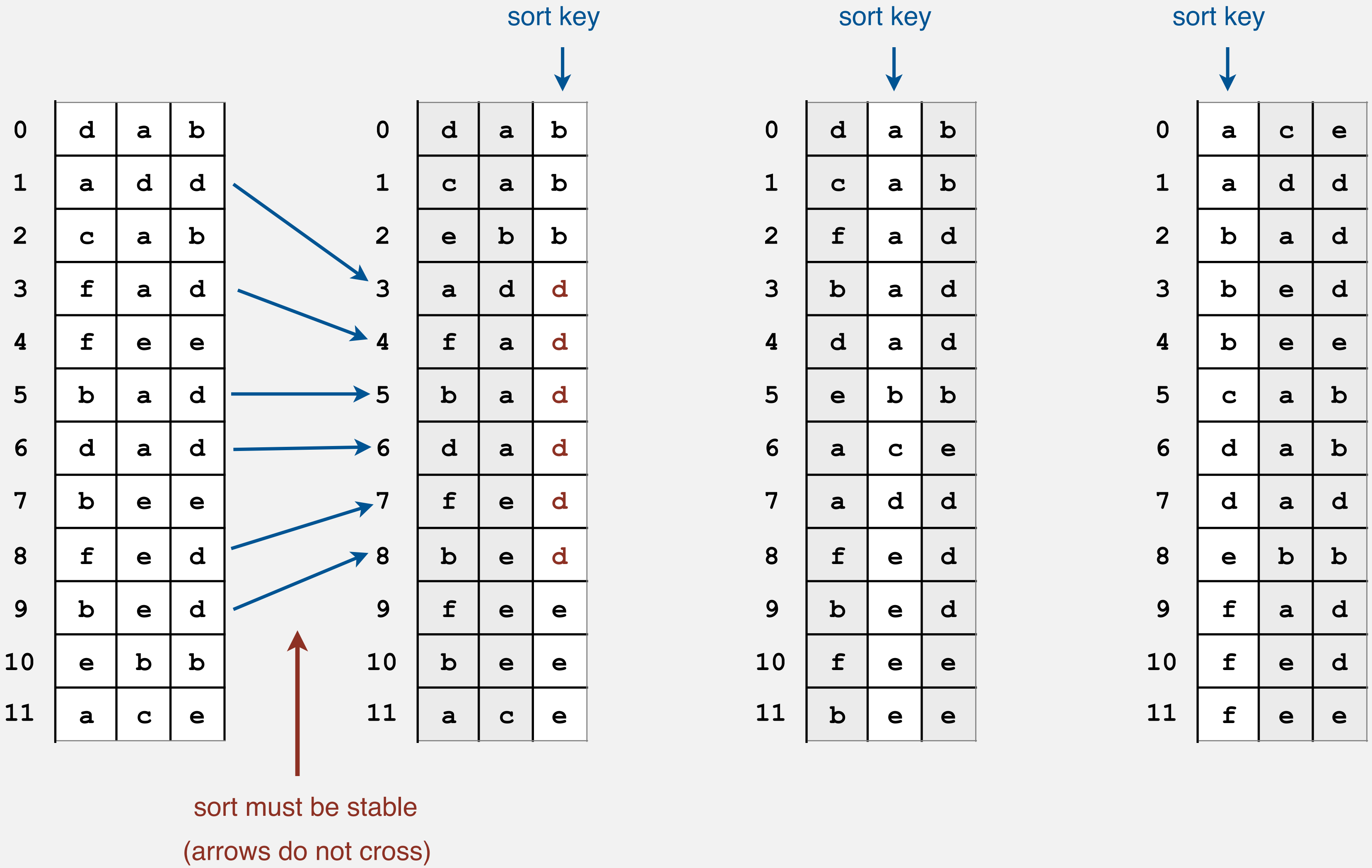
r	count[r]
a	2
b	5
c	6
d	8
e	9
f	12
-	12

i	aux[i]
0	a
1	a
2	b
3	b
4	b
5	c
6	d
7	d
8	e
9	f
10	f
11	f

Least-significant-digit-first string sort

LSD string sort.

- Consider characters from right to left.
- Stably sort using d^{th} character as the key (using key-indexed counting).



0	d	a	b
1	a	d	d
2	c	a	b
3	f	a	d
4	f	e	e
5	b	a	d
6	d	a	d
7	b	e	e
8	f	e	d
9	b	e	d
10	e	b	b
11	a	c	e

sort key

0	d	a	b
1	c	a	b
2	e	b	b
3	a	d	d
4	f	a	d
5	b	a	d
6	d	a	d
7	f	e	d
8	b	e	d
9	f	e	e
10	b	e	e
11	a	c	e

sort key

0	d	a	b
1	c	a	b
2	f	a	d
3	b	a	d
4	d	a	d
5	e	b	b
6	a	c	e
7	a	d	d
8	f	e	d
9	b	e	d
10	f	e	e
11	b	e	e

sort key

0	a	c	e
1	a	d	d
2	b	a	d
3	b	e	d
4	b	e	e
5	c	a	b
6	d	a	b
7	d	a	d
8	e	b	b
9	f	a	d
10	f	e	d
11	f	e	e

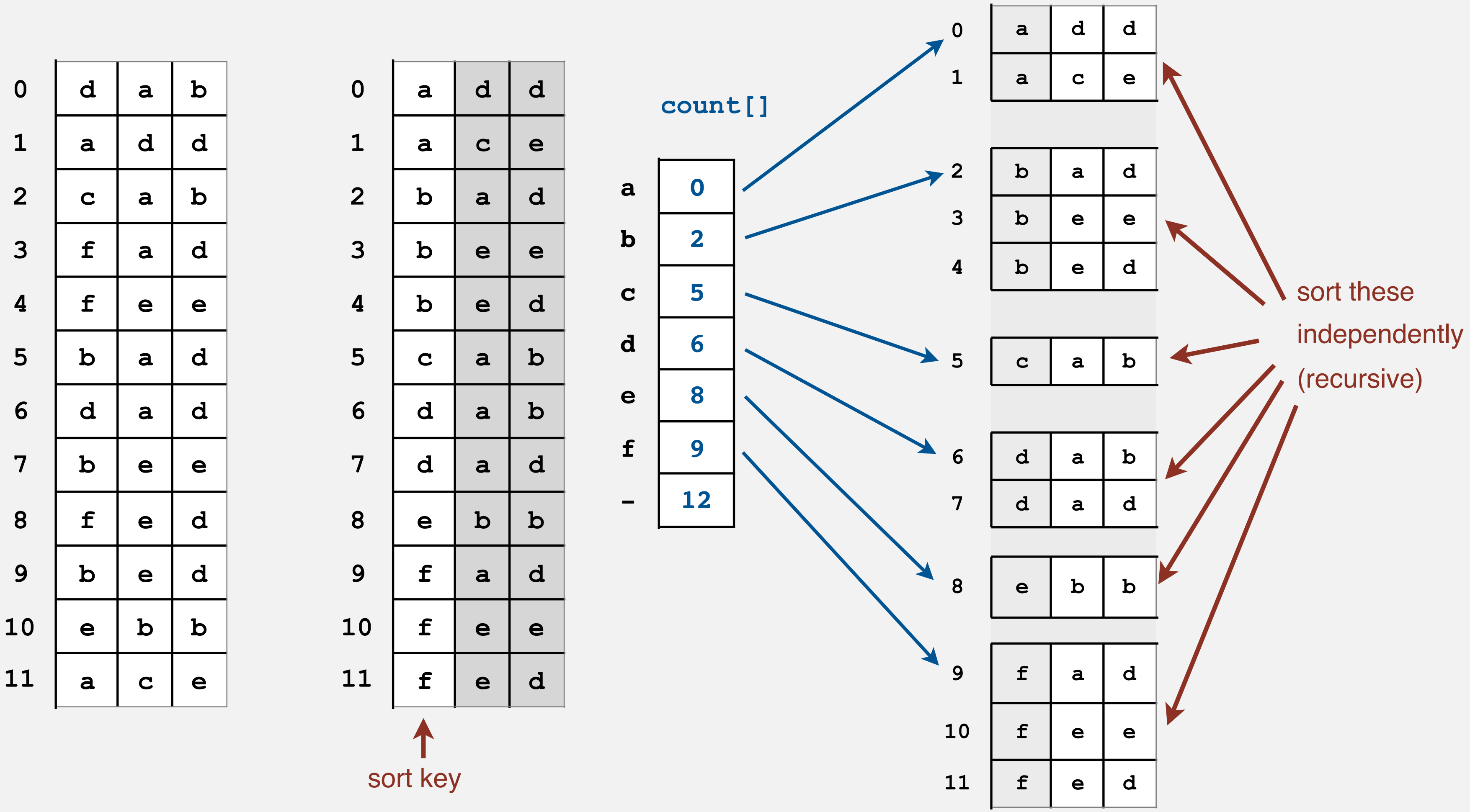
sort must be stable

(arrows do not cross)

Most-significant-digit-first string sort

MSD string sort.

- Partition file into R pieces according to first character (use key-indexed counting).
- Recursively sort all strings that start with each character (key-indexed counts delineate subarrays to sort).



0	d	a	b
1	a	d	d
2	c	a	b
3	f	a	d
4	f	e	e
5	b	a	d
6	d	a	d
7	b	e	e
8	f	e	d
9	b	e	d
10	e	b	b
11	a	c	e

0	a	d	d
1	a	c	e
2	b	a	d
3	b	e	e
4	b	e	d
5	c	a	b
6	d	a	b
7	d	a	d
8	e	b	b
9	f	a	d
10	f	e	e
11	f	e	d

sort key

count[]

a	0
b	2
c	5
d	6
e	8
f	9
-	12

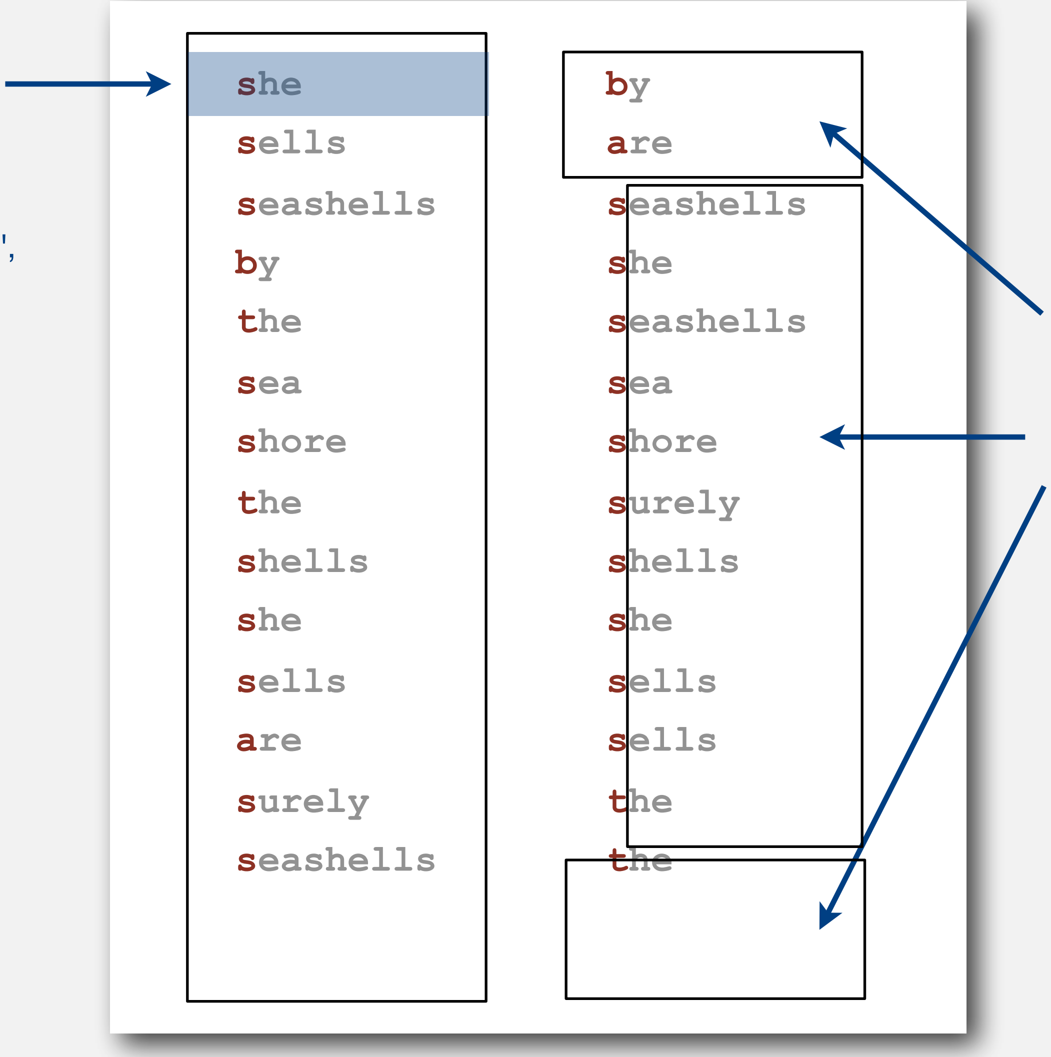
0	a	d	d
1	a	c	e
2	b	a	d
3	b	e	e
4	b	e	d
5	c	a	b
6	d	a	b
7	d	a	d
8	e	b	b
9	f	a	d
10	f	e	e
11	f	e	d

sort these independently (recursive)

3-way string quicksort (Bentley and Sedgewick, 1997)

Overview. Do 3-way partitioning on the d^{th} character.

- Cheaper than R-way partitioning of MSD string sort.
- Need not examine again characters equal to the partitioning char.



partitioning element

use first character value
to partition into "less", "equal",
and "greater" subarrays

she
sells
seashells
by
the
sea
shore
the
shells
she
sells
are
surely
seashells

by
are
seashells
she
seashells
sea
shore
surely
shells
she
sells
sells
the
the

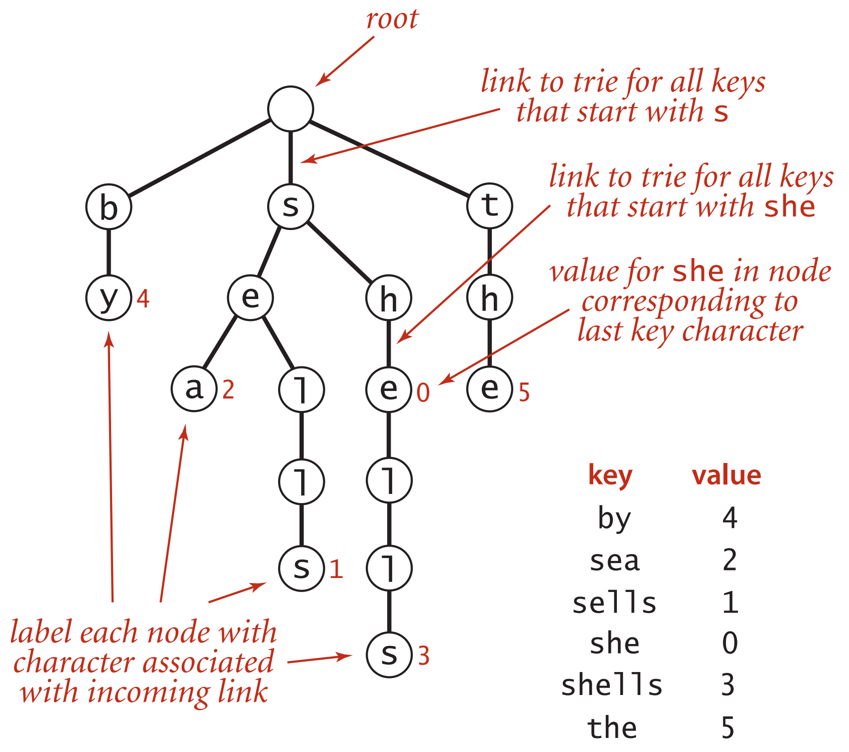
recursively sort subarrays,
excluding first character
for "equal" subarray

▸ tries

R-Way Tries

- Store characters and values in nodes (not keys).
- Each node has R children, one for each possible character.
- For now, we do not draw null links.

Ex. `she sells sea shells by the`



root

link to trie for all keys that start with s

link to trie for all keys that start with she

value for she *in node corresponding to last key character*

label each node with character associated with incoming link

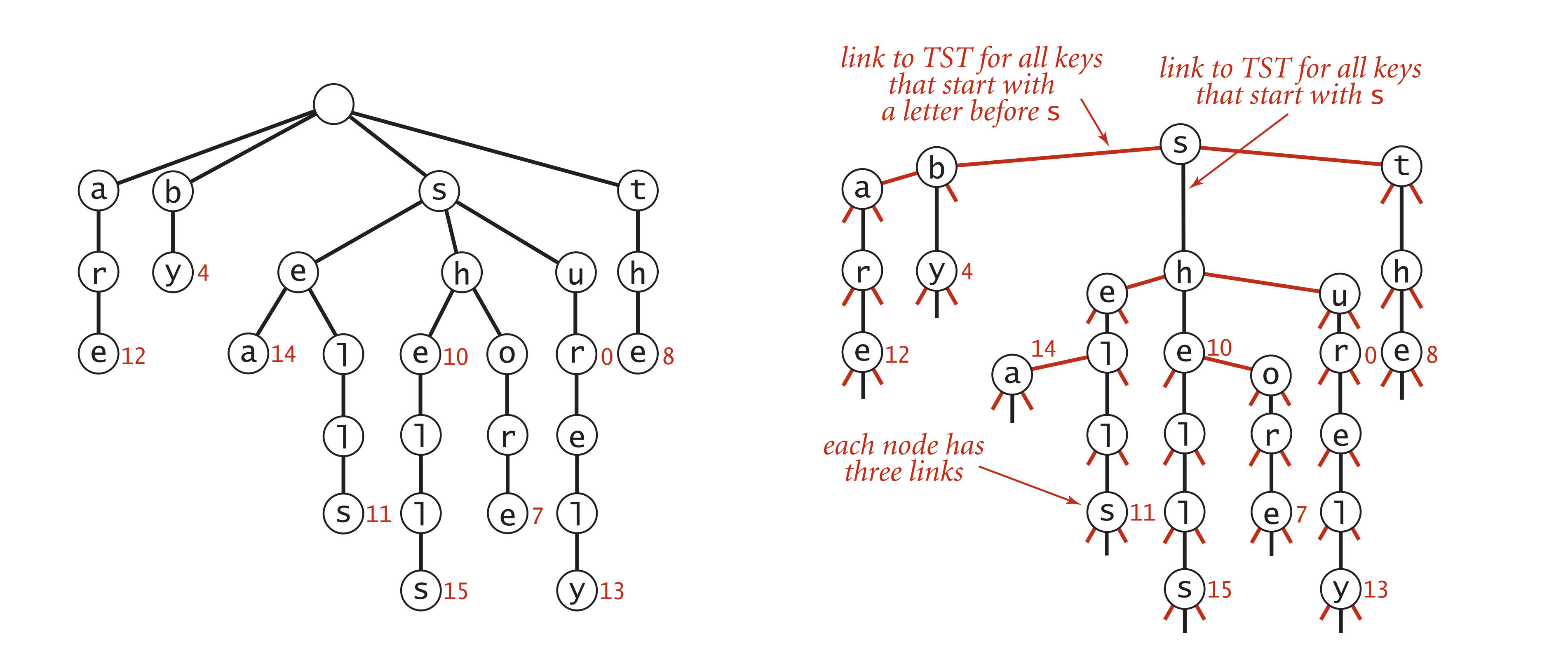
key	value
by	4
sea	2
sells	1
she	0
shells	3
the	5

Anatomy of a trie

Ternary search tries

TST. [Bentley-Sedgewick, 1997]

- Store characters and values in nodes (not keys).
- Each node has three children: smaller (left), equal (middle), larger (right).



link to TST for all keys that start with a letter before s
link to TST for all keys that start with s
each node has three links
a
b
s
t
r
y
4
e
h
u
e
12
a
14
l
e
10
o
r
0
e
8
l
l
r
e
s
11
l
e
7
l
s
15
y
13

TST representation of a trie

‣ compression

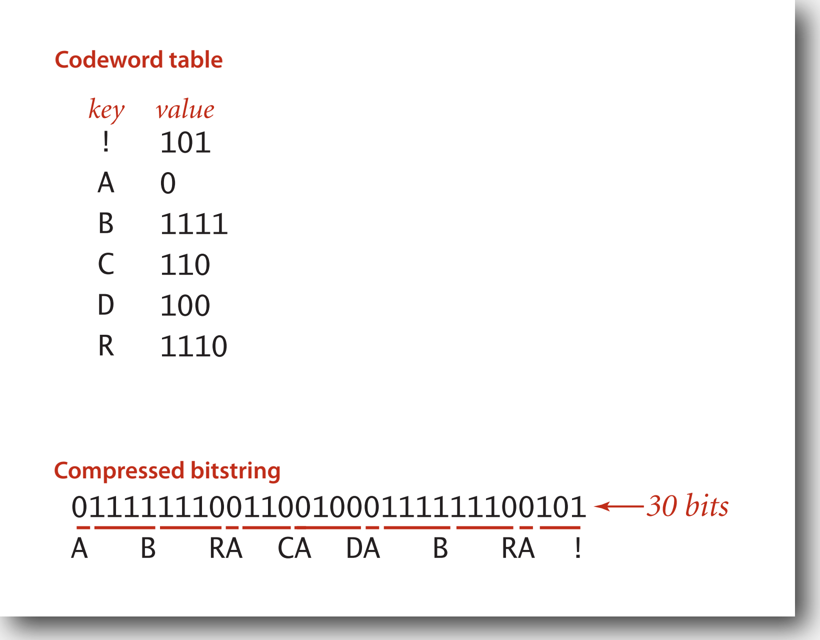
Variable-length codes

Need **prefix-free code** to prevent ambiguities: Ensure that no codeword is a prefix of another.

Ex 1. Fixed-length code.

Ex 2. Append special stop char to each codeword.

Ex 3. General prefix-free code.



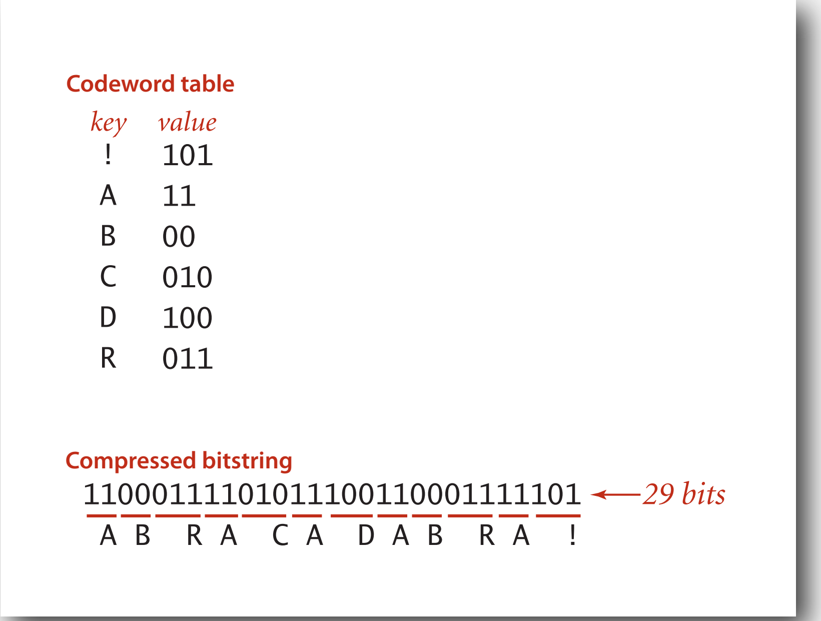
Codeword table

key	*value*
!	101
A	0
B	1111
C	110
D	100
R	1110

Compressed bitstring

011111110011001000111111100101 ← *30 bits*

A B RA CA DA B RA !



Codeword table

key	*value*
!	101
A	11
B	00
C	010
D	100
R	011

Compressed bitstring

11000111101011100110001111101 ← *29 bits*

A B R A C A D A B R A !

Huffman codes

David Huffman

Goal: Find **best prefix-free code**

Huffman algorithm:

- **Count frequency** `freq[i]` **for each char** `i` in input.
- Start with one node corresponding to each char `i` (with weight `freq[i]`).
- Repeat until single trie formed:
  - select two tries with min weight `freq[i]` and `freq[j]`
  - merge into single trie with weight `freq[i] + freq[j]`
- Implementation: **Use minPQ** on trie weights

Applications. JPEG, MP3, MPEG, PKZIP, GZIP, PDF, …

Prefix-free codes: trie representation

Q. How to represent the prefix-free code?

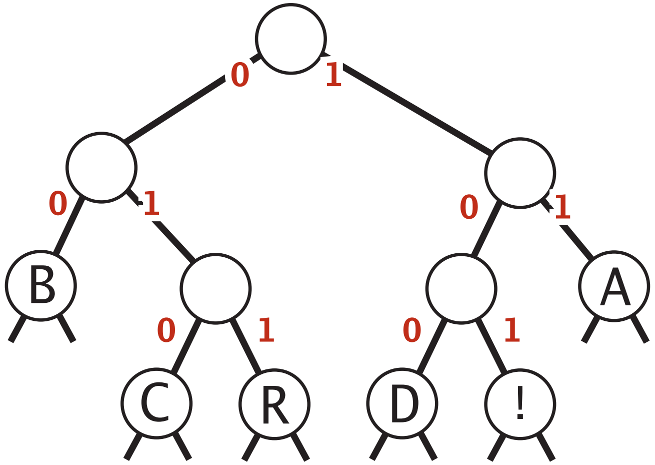
A. A binary trie!

- Chars in leaves.
- Codeword is path from root to leaf.

Codeword table

key	*value*
!	101
A	11
B	00
C	010
D	100
R	011

Trie representation



Compressed bitstring

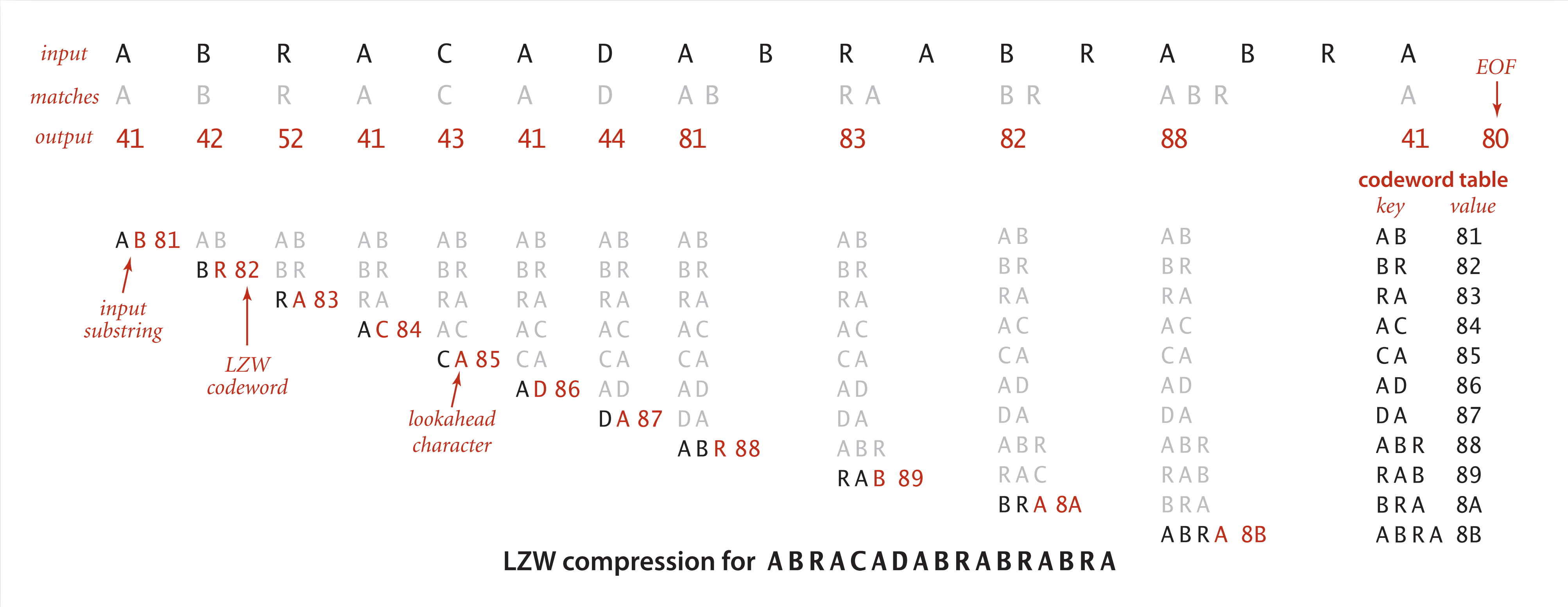
11000111101011100110001111101 ← *29 bits*

A B R A C A D A B R A !

Lempel-Ziv-Welch compression

LZW compression.

- Create ST associating W-bit codewords with string keys.
- Initialize ST with codewords for single-char keys.
- Find longest string s in ST that is a prefix of unscanned part of input.
- Write the W-bit codeword associated with s.
- Add s + c to ST, where c is next char in the input.



input	A	B	R	A	C	A	D	A	B	R	A	B	R	A	B	R	A	EOF
matches	A	B	R	A	C	A	D	A B		R A		B R		A B R			A	
output	41	42	52	41	43	41	44	81		83		82		88			41	80

codeword table

key	*value*
A B	81
B R	82
R A	83
A C	84
C A	85
A D	86
D A	87
A B R	88
R A B	89
B R A	8A
A B R A	8B

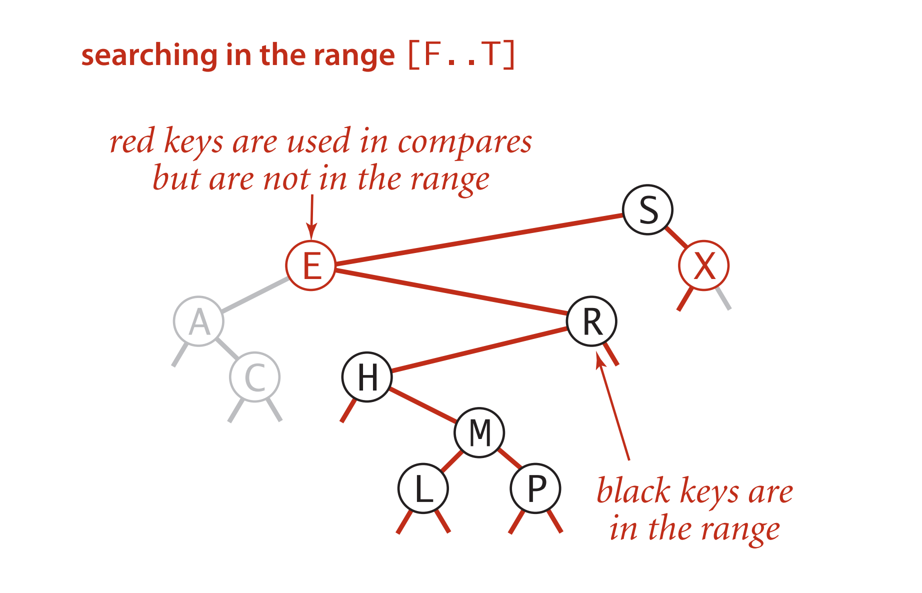
LZW compression for A B R A C A D A B R A B R A B R A

‣ Geometric search

1d range search: BST implementation

Range search. Find all keys between k_1 and k_2.

- Recursively find all keys in left subtree (if any could fall in range).
- Check key in current node.
- Recursively find all keys in right subtree (if any could fall in range).


searching in the range [F..T]
red keys are used in compares but are not in the range
S
E
X
A
R
C
H
M
L
P
black keys are in the range

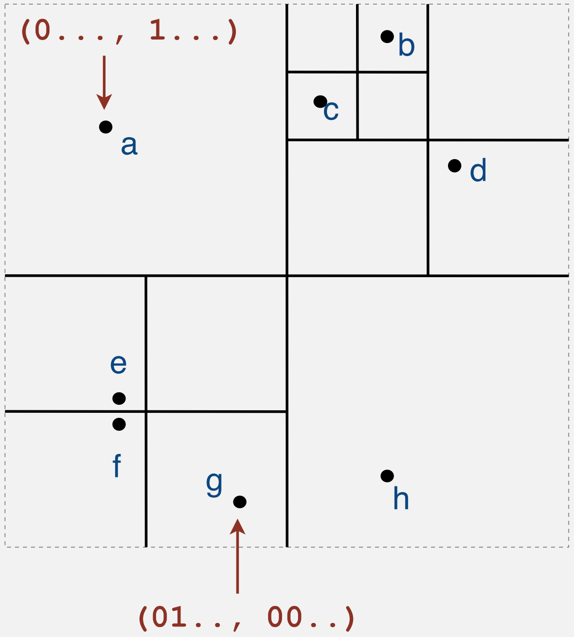
Range search in a BST

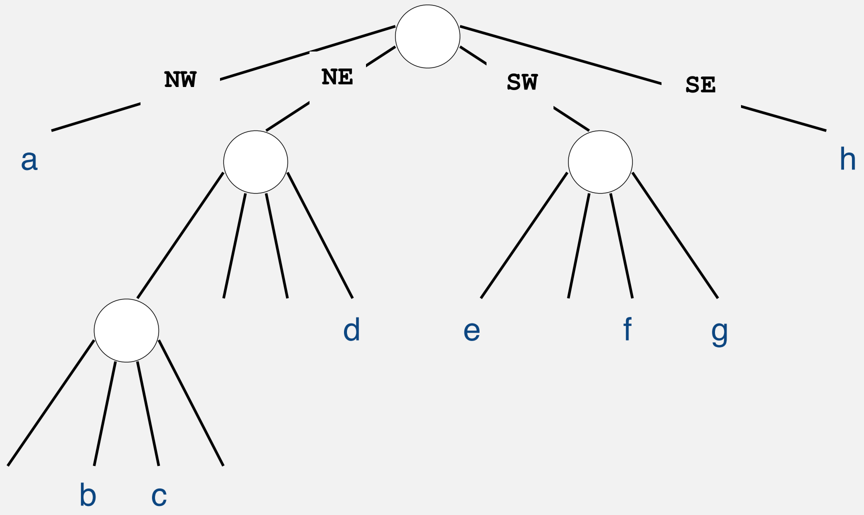
Proposition. Running time is proportional to $R + \log N$ (assuming BST is balanced).

Quadtree

Idea. Recursively divide space into 4 quadrants.

Implementation. 4-way tree (actually a trie).





```
public class QuadTree
{
   private Quad quad;
   private Value val;
   private QuadTree NW, NE, SW, SE;
}
```

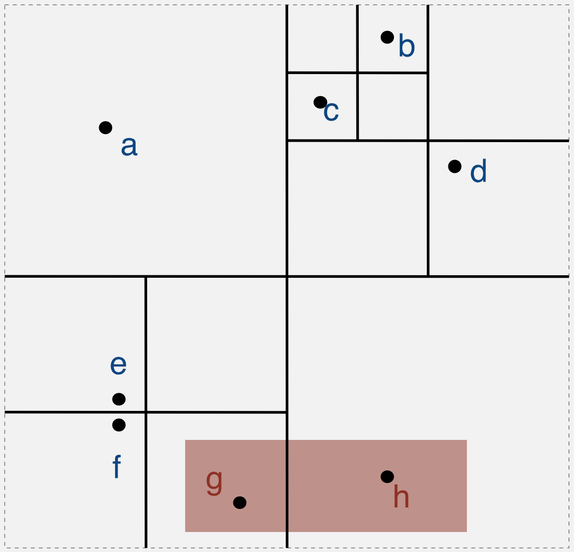
Benefit. Good performance in the presence of clustering.

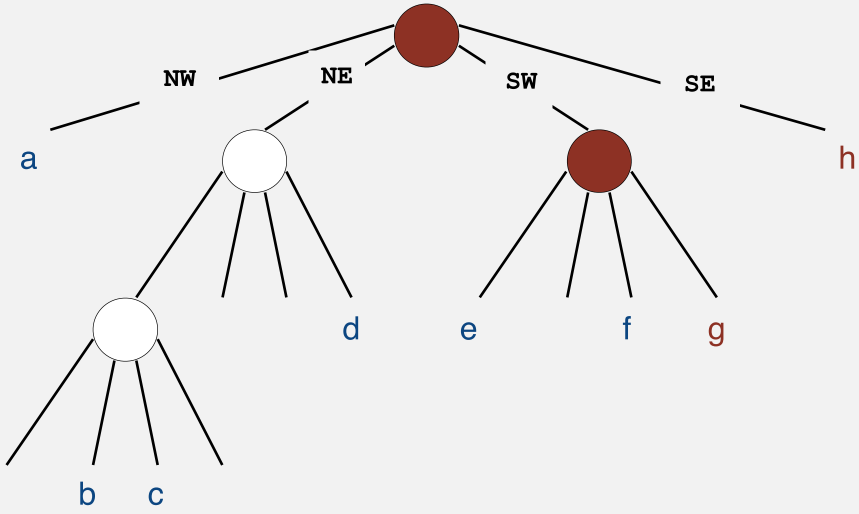
Drawback. Arbitrary depth!

Quadtree: 2d orthogonal range search

Range search. Find all keys in a given 2d range.

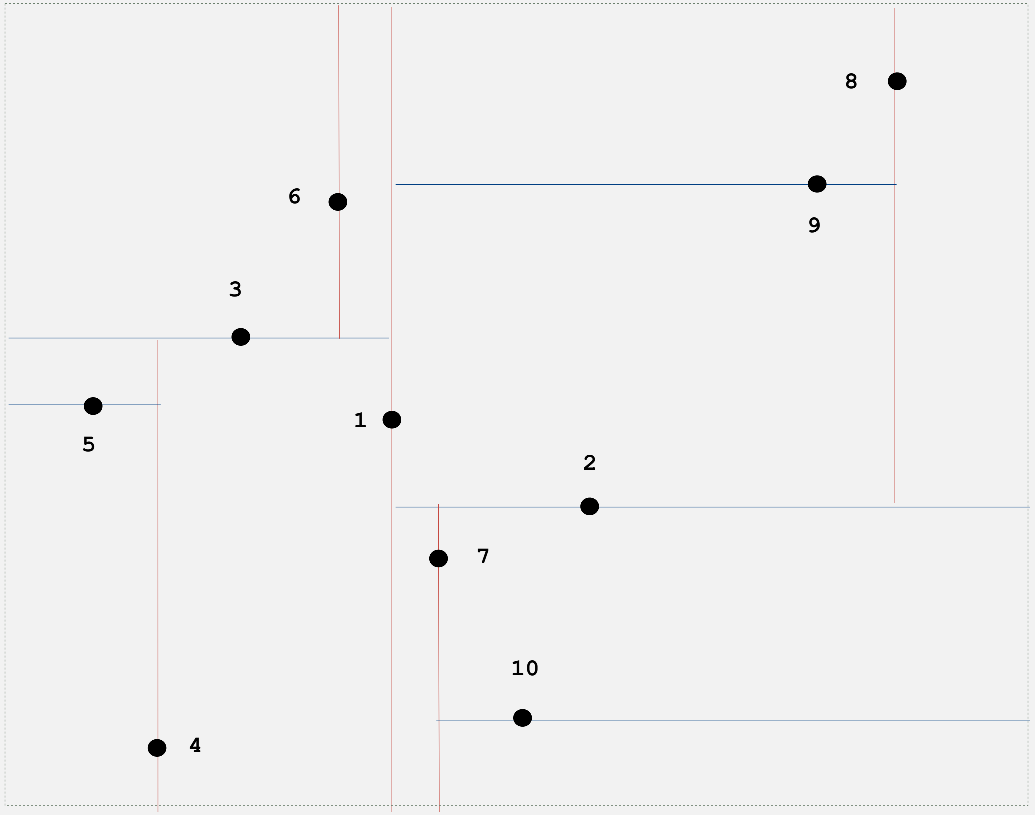
- Recursively find all keys in NE quadrant (if any could fall in range).
- Recursively find all keys in NW quadrant (if any could fall in range).
- Recursively find all keys in SE quadrant (if any could fall in range).
- Recursively find all keys in SW quadrant (if any could fall in range).

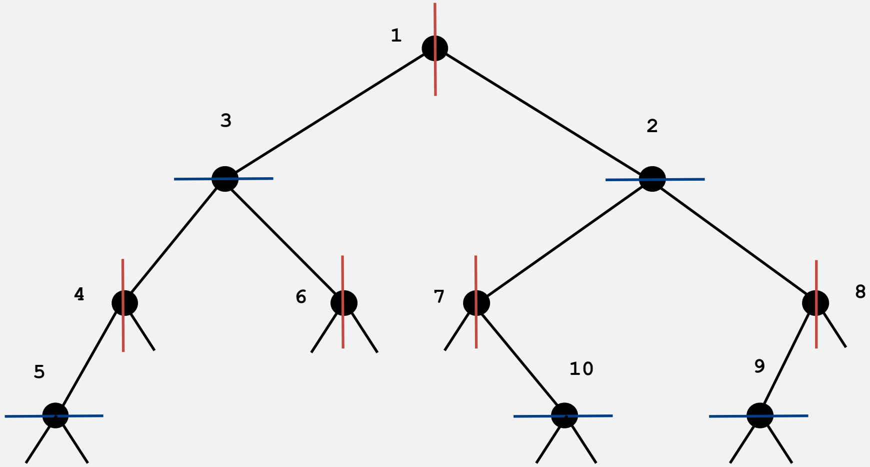




Typical running time. $R + \log N$.

Recursively partition plane into two halfplanes.

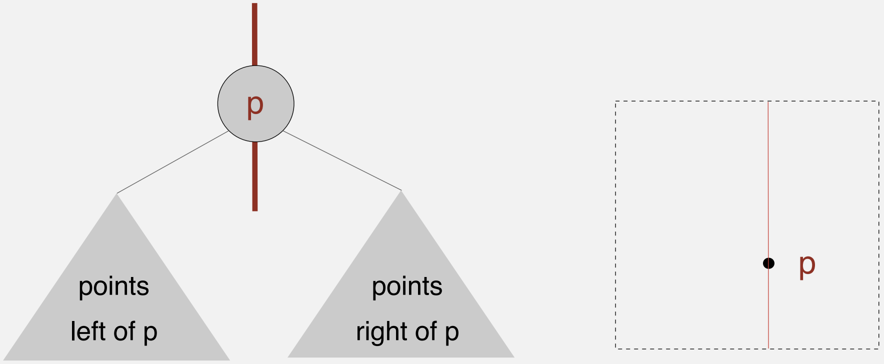




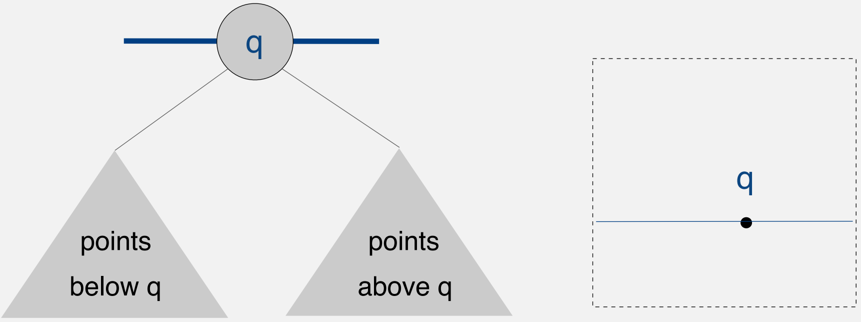
2d tree implementation

Data structure. BST, but alternate using x- and y-coordinates as key.

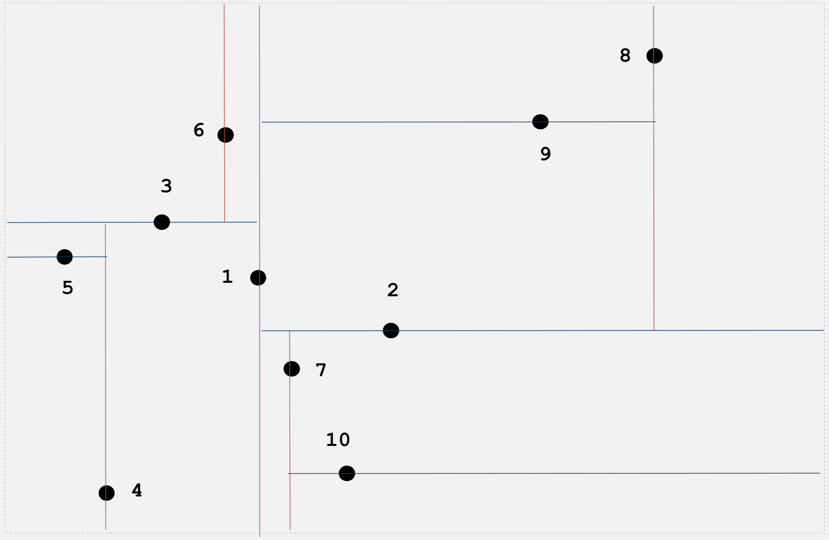
- Search gives rectangle containing point.
- Insert further subdivides the plane.

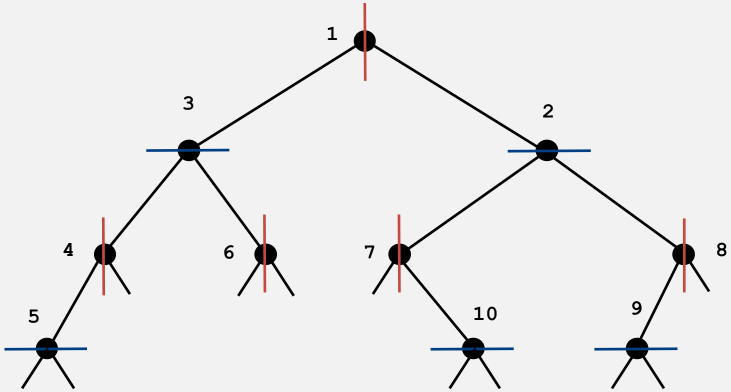


even levels



odd levels





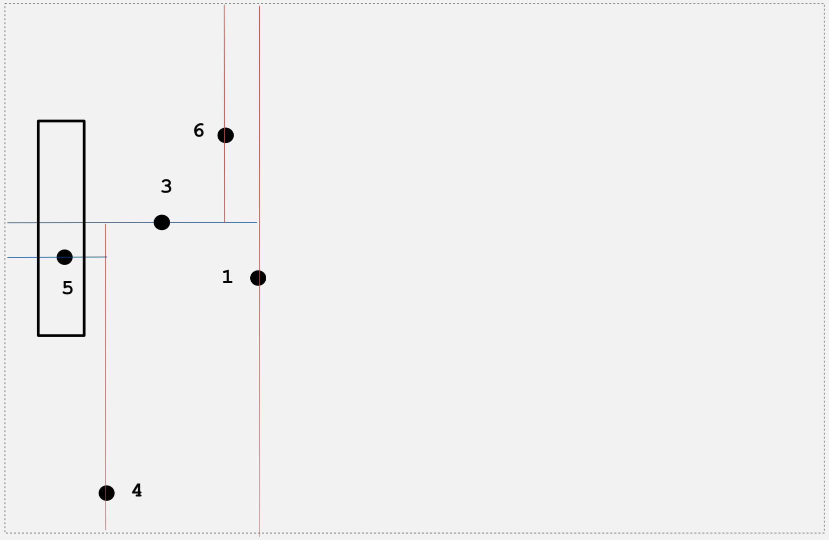
2d tree: 2d orthogonal range search

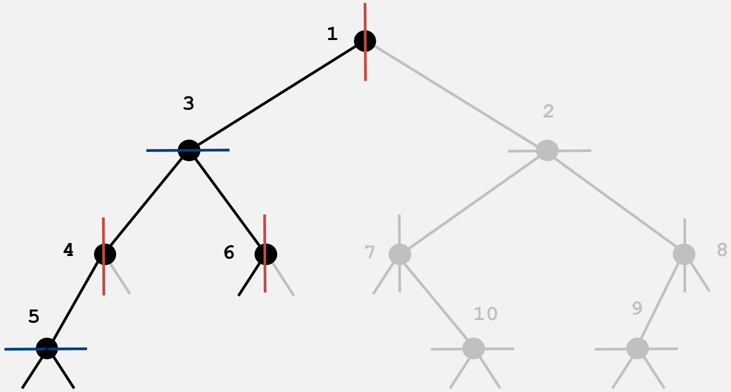
Range search. Find all points in a query axis-aligned rectangle.

- Check if point in node lies in given rectangle.
- Recursively search left/top subdivision (if any could fall in rectangle).
- Recursively search right/bottom subdivision (if any could fall in rectangle).

Typical case. $R + \log N$.

Worst case (assuming tree is balanced). $R + \sqrt{N}$.





2d tree: nearest neighbor search

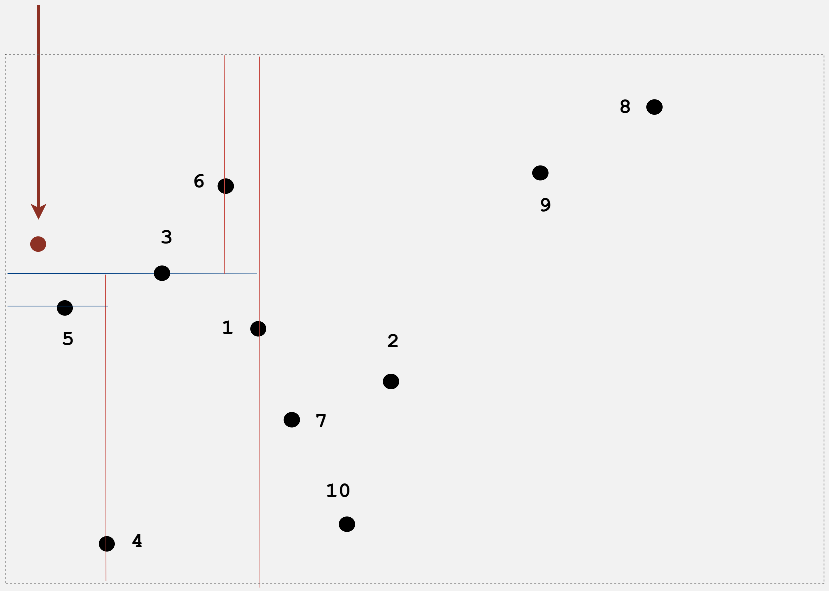
Nearest neighbor search. Given a query point, find the closest point.

- Check distance from point in node to query point.
- Recursively search left/top subdivision (if it could contain a closer point).
- Recursively search right/bottom subdivision (if it could contain a closer point).
- Organize recursive method so that it begins by searching for query point.

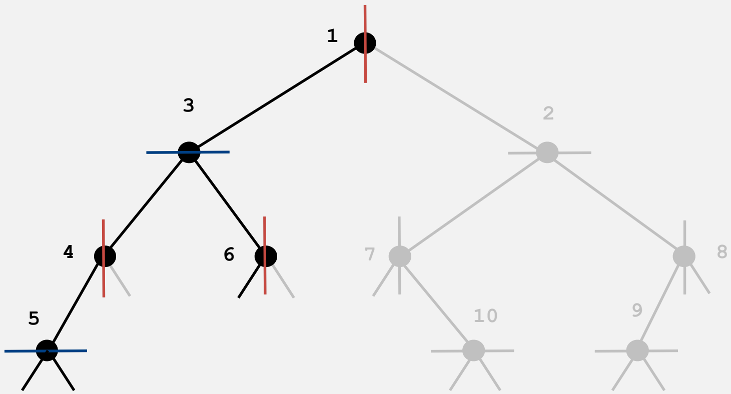
Typical case. $\log N$.

Worst case (even if tree is balanced). N.

query point



closest point = 5

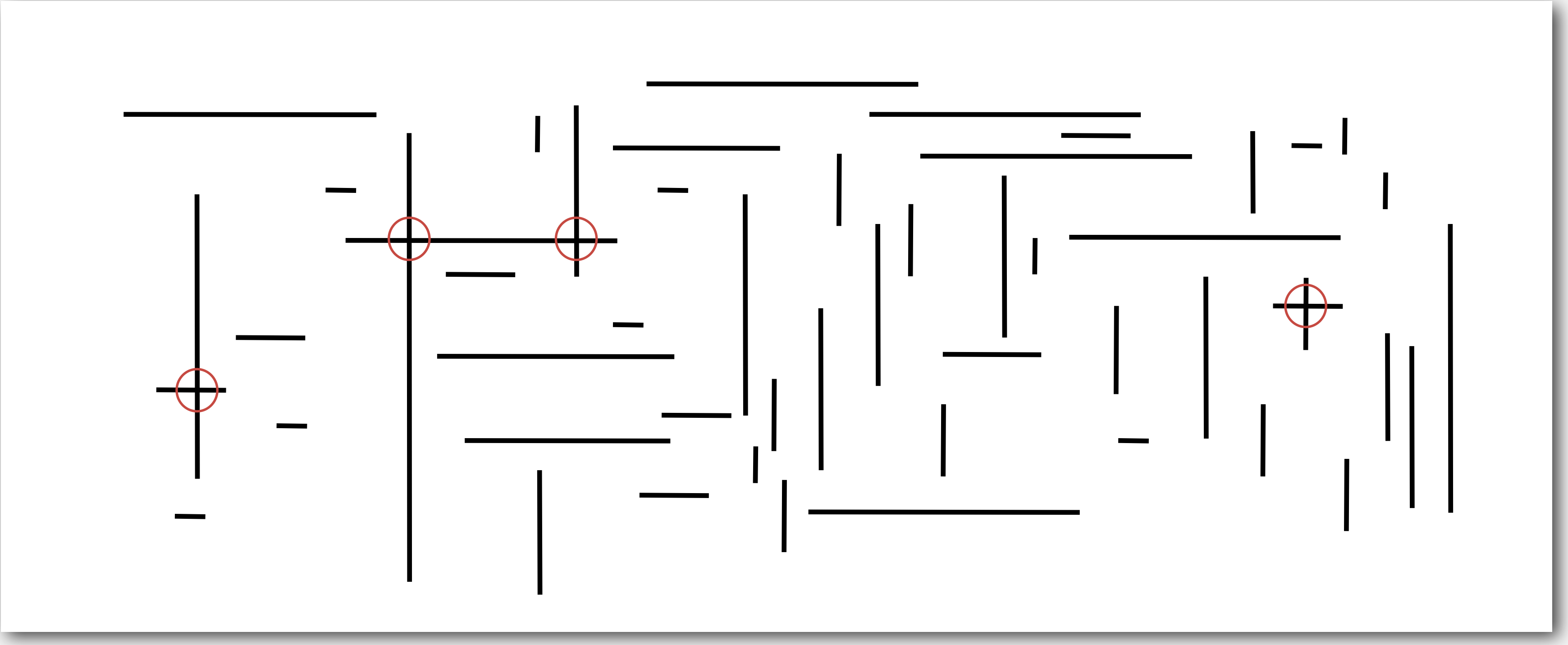


Search for intersections

Problem. Find all intersecting pairs among N geometric objects.

Applications. CAD, games, movies, virtual reality, ...

Simple version. 2d, all objects are horizontal or vertical line segments.

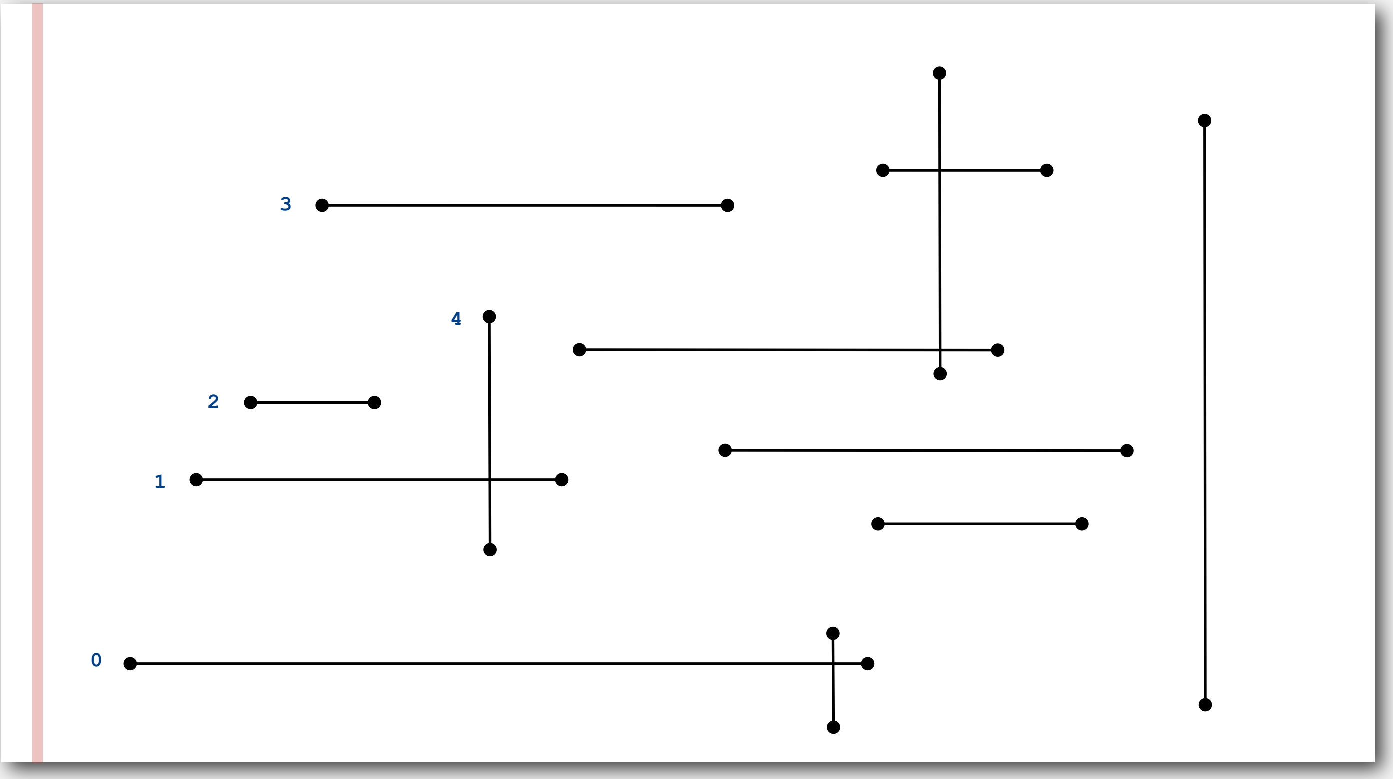


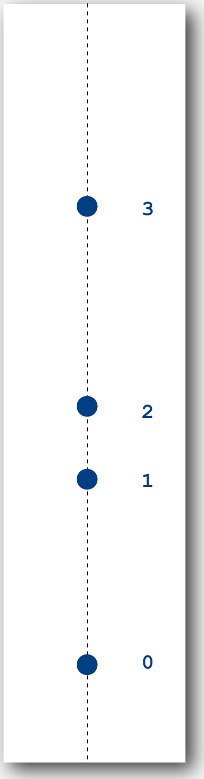
Brute force. Test all $\Theta(N^2)$ pairs of line segments for intersection.

Orthogonal line segment intersection search: sweep-line algorithm

Sweep vertical line from left to right.

- x-coordinates define events.
- h-segment (left endpoint): insert y-coordinate into ST.



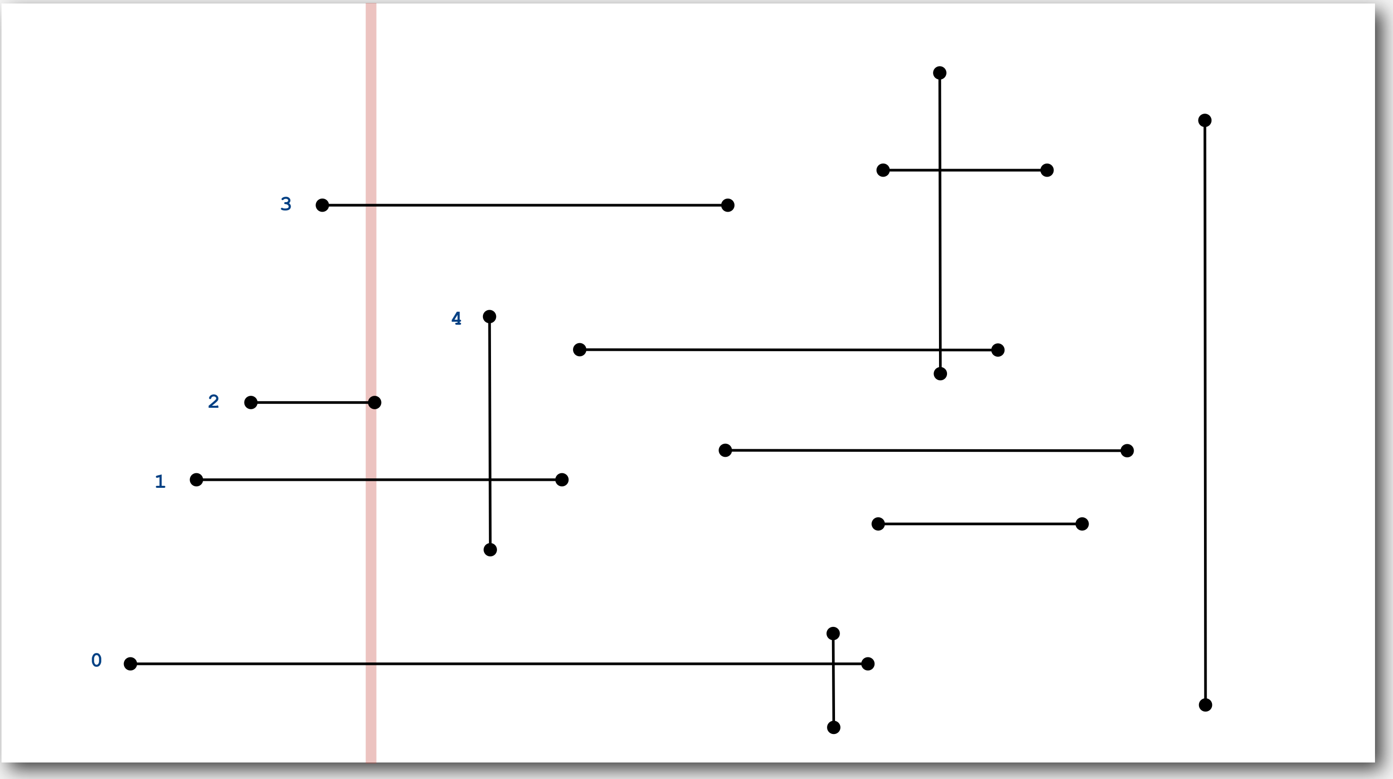


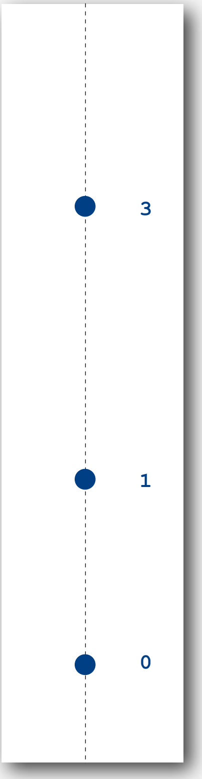
y-coordinates

Orthogonal line segment intersection search: sweep-line algorithm

Sweep vertical line from left to right.

- x-coordinates define events.
- h-segment (left endpoint): insert y-coordinate into ST.
- h-segment (right endpoint): remove y-coordinate from ST.



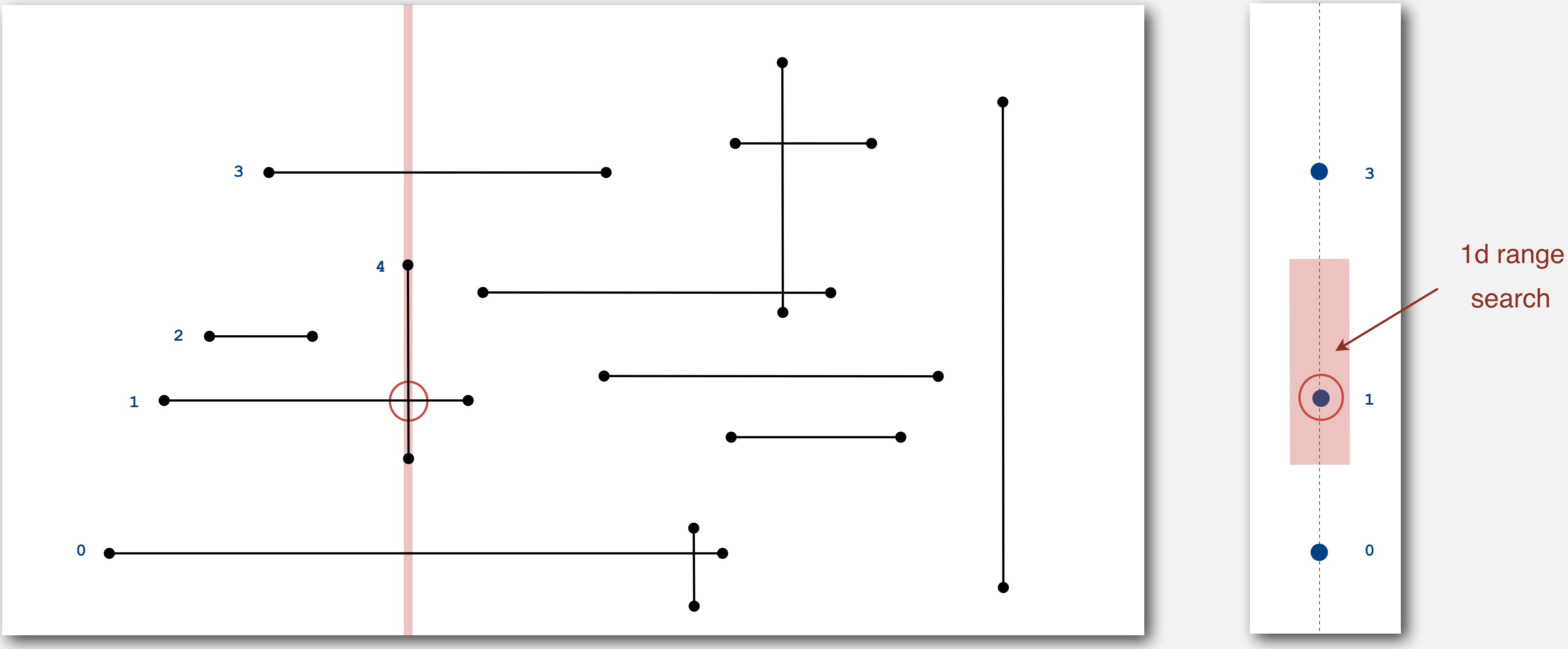


y-coordinates

Orthogonal line segment intersection search: sweep-line algorithm

Sweep vertical line from left to right.

- x-coordinates define events.
- h-segment (left endpoint): insert y-coordinate into ST.
- h-segment (right endpoint): remove y-coordinate from ST.
- v-segment: range search for interval of y-endpoints.

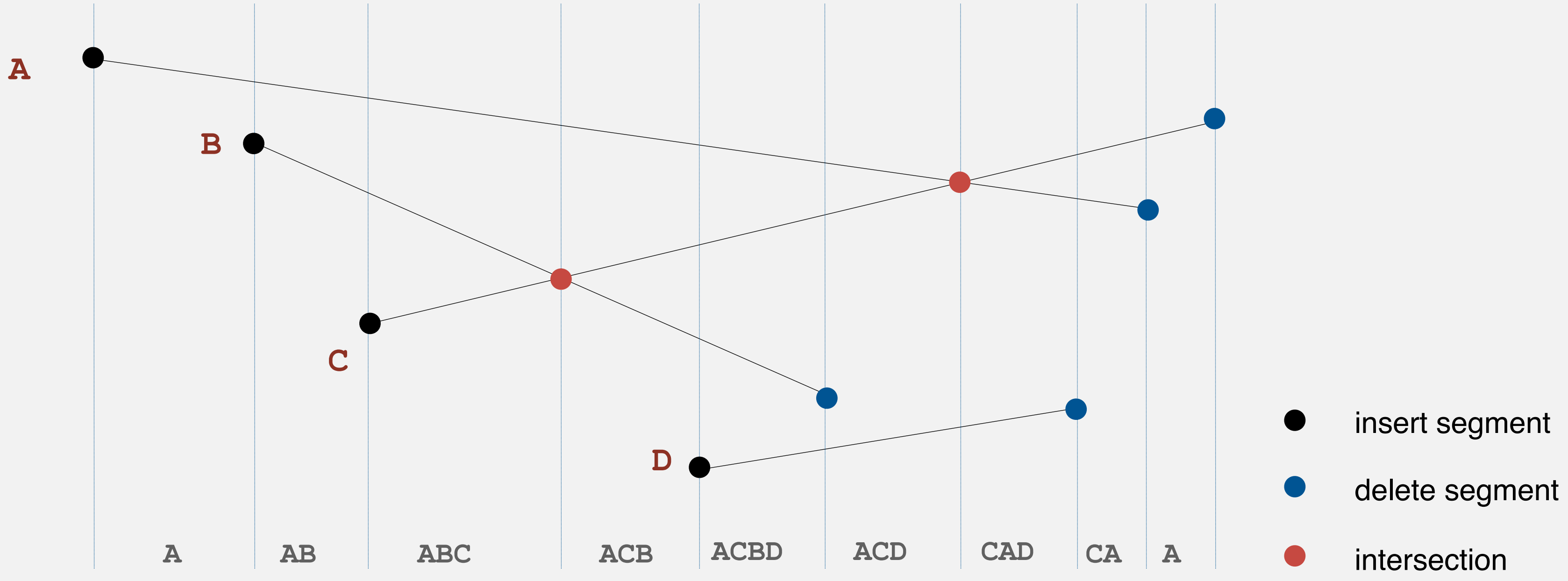


y-coordinates

General line segment intersection search

Extend sweep-line algorithm.

- Maintain segments that intersect sweep line ordered by y-coordinate.
- Intersections can only occur between adjacent segments.
- Add line segment ⇒ one new pair of adjacent segments.
- Delete line segment ⇒ two segments become adjacent
- Intersection ⇒ swap adjacent segments.



order of segments that intersect sweep line

Line segment intersection: implementation

Efficient implementation of sweep line algorithm.

- Maintain PQ of important x-coordinates: endpoints and intersections.
- Maintain set of segments intersecting sweep line, sorted by x.
- Time proportional to $R \log N + N \log N$.

↑ to support "next largest" and "next smallest" queries

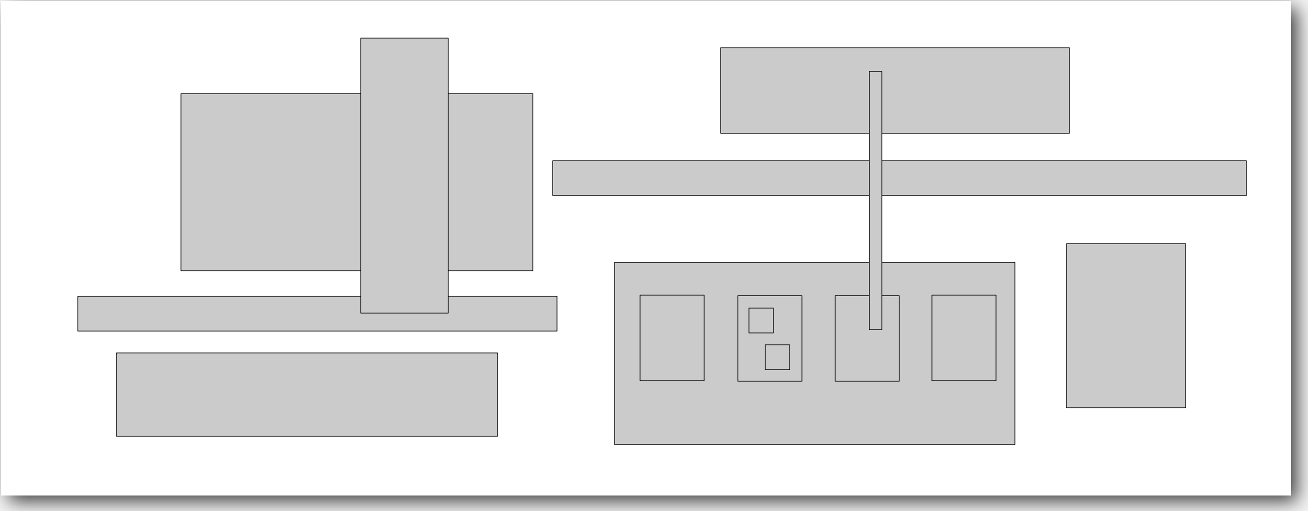
Implementation issues.

- Degeneracy.
- Floating-point precision.
- Must use PQ, not presort (intersection events are unknown ahead of time).

Orthogonal rectangle intersection search

Goal. Find all intersections among a set of N orthogonal rectangles.

Non-degeneracy assumption. All x- and y-coordinates are distinct.

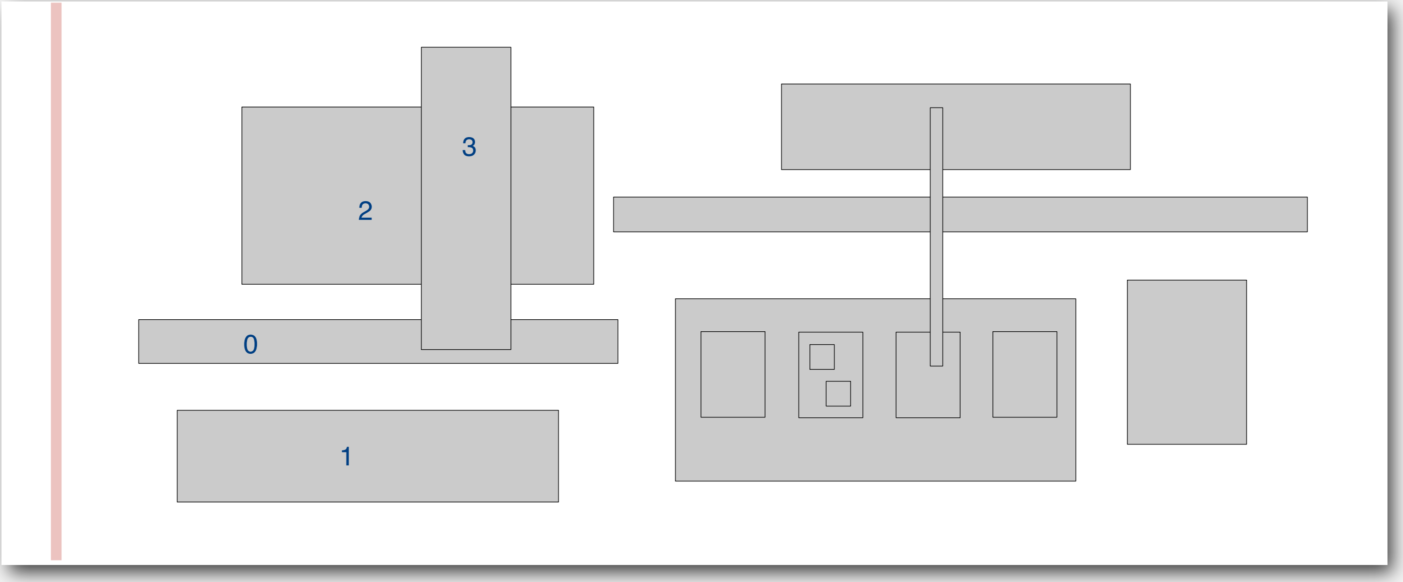


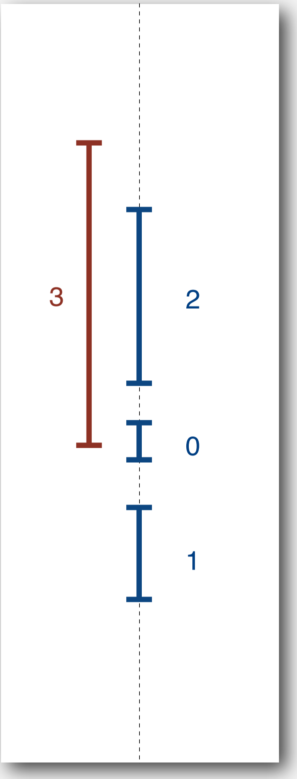
Application. Design-rule checking in VLSI circuits.

Orthogonal rectangle intersection search

Move a vertical "sweep line" from left to right.

- Sweep line: sort rectangles by x-coordinates and process in this order, stopping on left and right endpoints.
- Maintain set of y-intervals intersecting sweep line.
- Left endpoint: search set for intersecting y-intervals; insert y-interval.
- Right endpoint: delete y-interval.





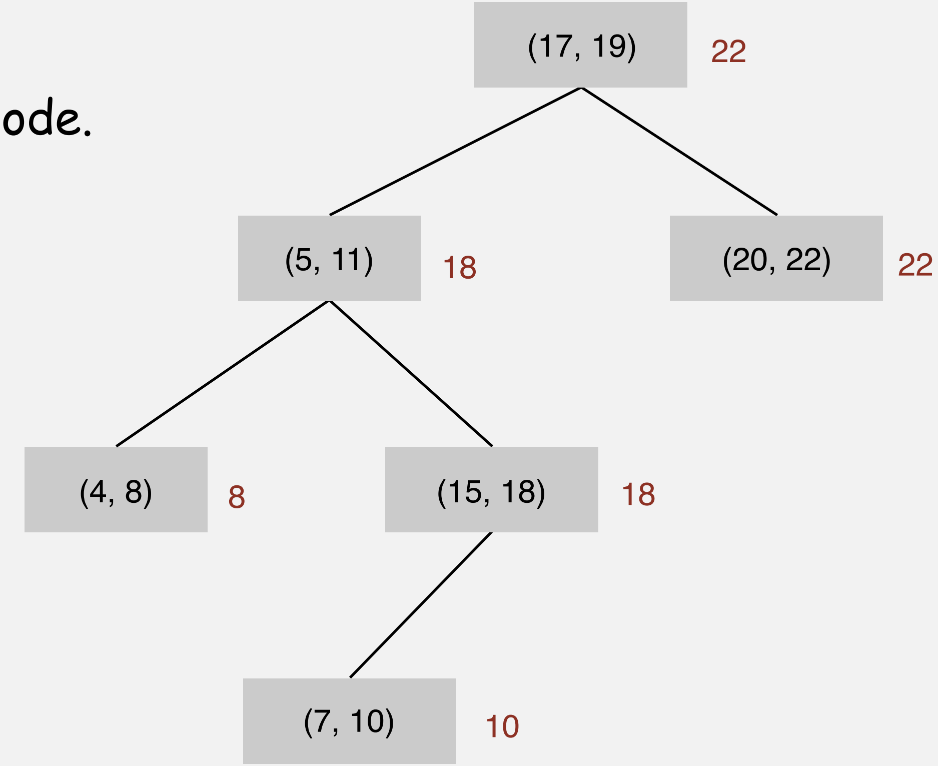
y-coordinates

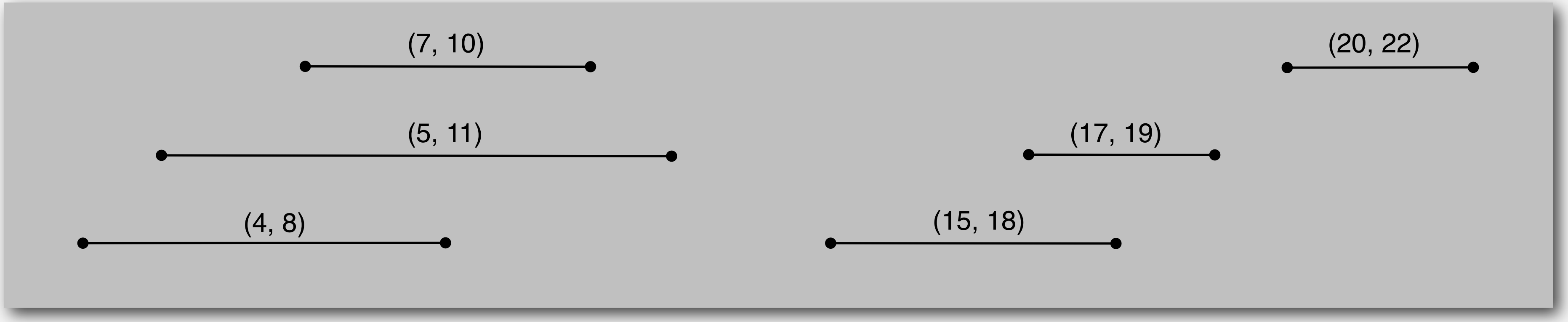
Interval search trees

Create BST, where each node stores an interval `(lo, hi)`.

- Use left endpoint as BST key.
- Store max endpoint in subtree rooted at node.

Suffices to implement all ops efficiently!



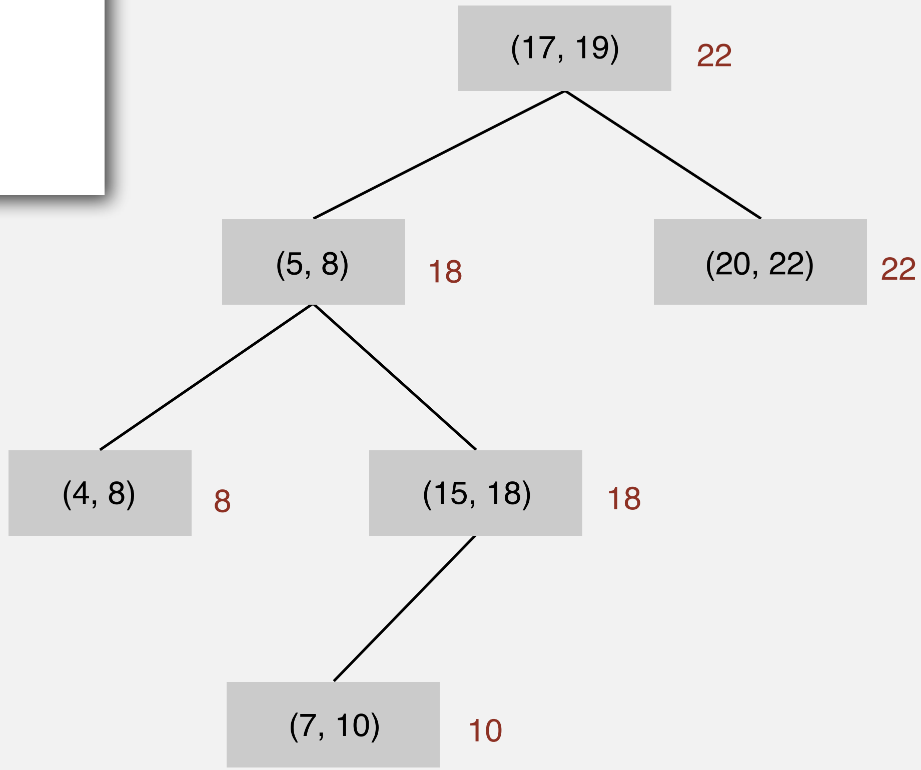


Finding an intersecting interval

To search for any interval that intersects query interval (lo, hi) :

```
Node x = root;
while (x != null)
{
   if (x.interval.intersects(lo, hi))
      return x.interval;
   else if (x.left == null)   x = x.right;
   else if (x.left.max < lo) x = x.right;
   else                       x = x.left;
}
return null;
```

Ex. Search for $(9, 10)$.



Interval search tree: analysis

Implementation. Use a red-black BST to guarantee performance.

can maintain auxiliary information
using log N extra work per op

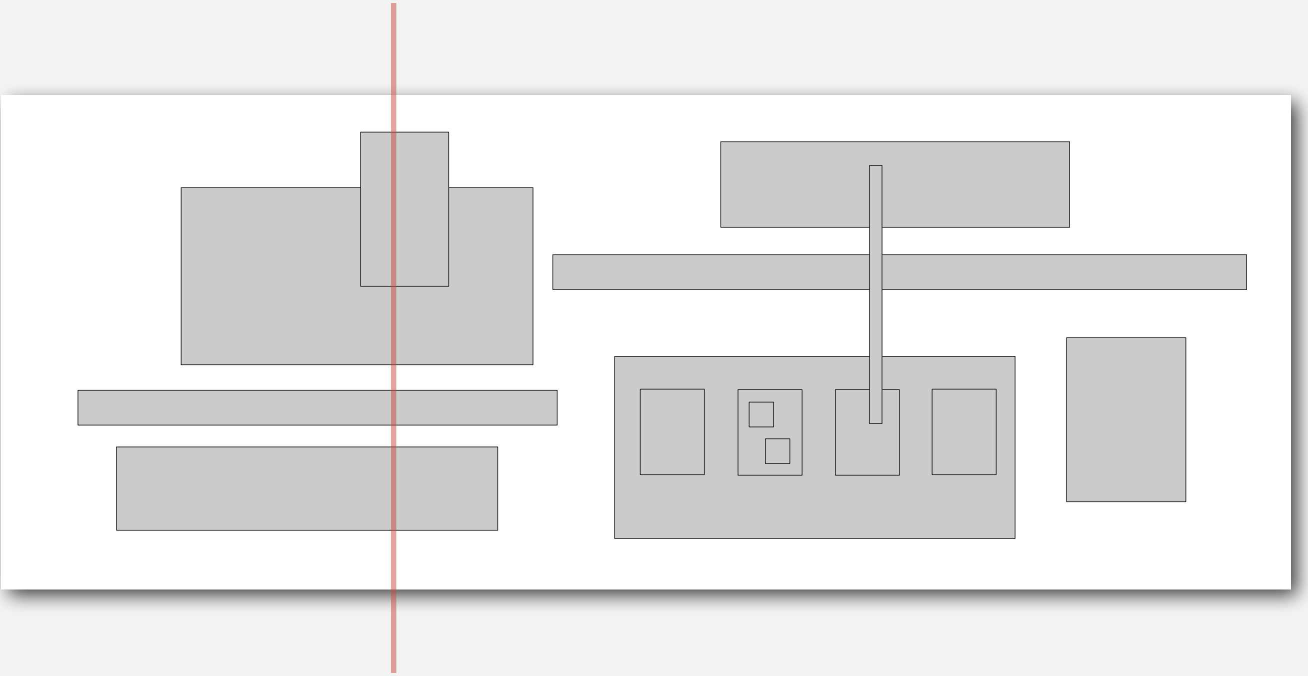
operation	brute	interval search tree	best in theory
insert interval	1	log N	log N
find interval	N	log N	log N
delete interval	N	log N	log N
find any interval that intersects (lo, hi)	N	log N	log N
find all intervals that intersects (lo, hi)	N	R log N	R + log N

order of growth of running time for N intervals

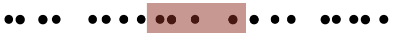
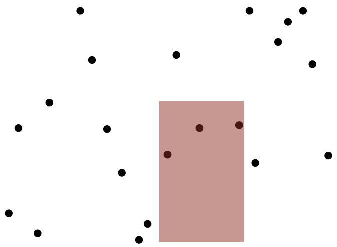
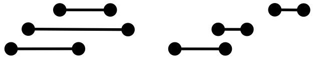
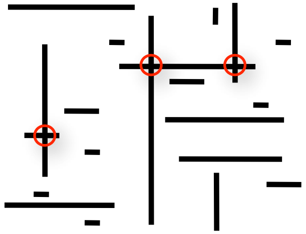
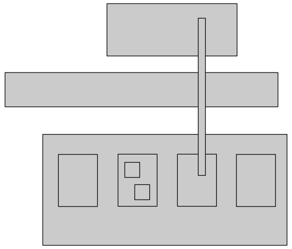
Rectangle intersection sweep-line algorithm: review

Move a vertical "sweep line" from left to right.

- Sweep line: sort rectangles by x-coordinates and process in this order, stopping on left and right endpoints.
- Maintain set of rectangles that intersect the sweep line in an interval search tree (using y-intervals of rectangle).
- Left endpoint: interval search for y-interval of rectangle; insert y-interval.
- Right endpoint: delete y-interval.



Geometric search summary: algorithms of the day

problem	example	solution
1d range search		BST
kd orthogonal range search		kd tree
1d interval search		interval search tree
2d orthogonal line segment intersection		sweep line reduces to 1D range search
2d orthogonal rectangle intersection		sweep line reduces to 1D interval search

‣ dynamic programming

General dynamic programming technique

Applies to a problem that at first seems to require a lot of time (possibly exponential), provided we have:

- Simple subproblems: the subproblems can be defined in terms of a few variables, such as j, k, l, m, and so on.
- Subproblem optimality: the global optimum value can be defined in terms of optimal subproblems
- Subproblem overlap: the subproblems are not independent, but instead they overlap (hence, should be constructed bottom-up).

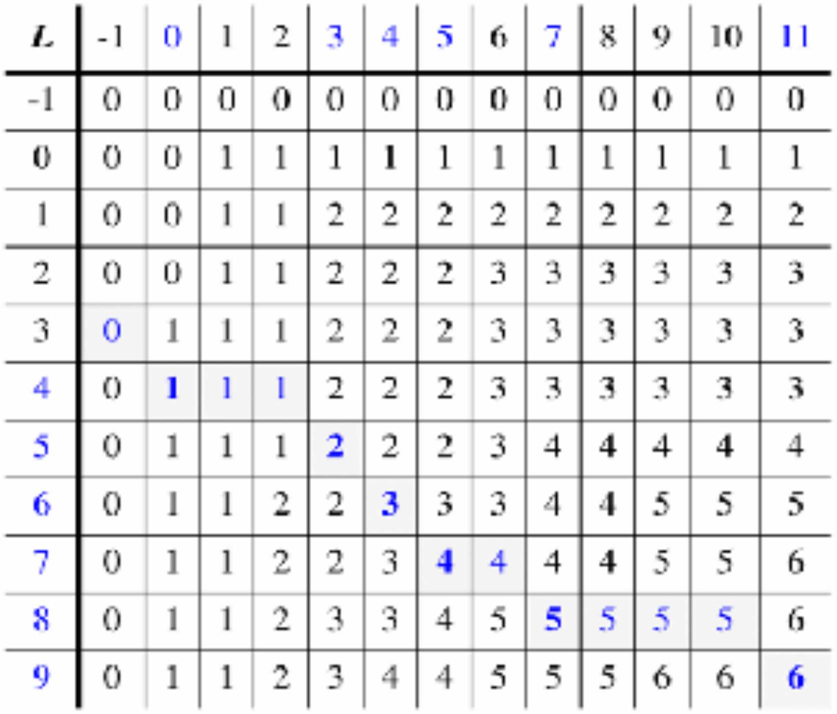
Analysis of the LCS algorithm

Define L[i,j] to be the length of the longest common subsequence of two strings X[0..i] and Y[0..j].

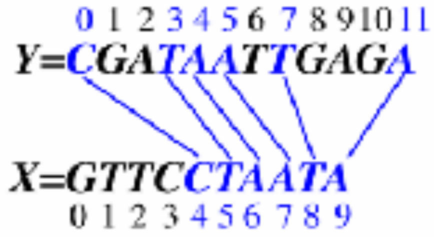
Then we can define L[i,j] in the general case as follows:

- If $x_i=y_j$, then L[i,j] = L[i-1,j-1] + 1 (we can add this match)
- If $x_i \neq y_j$, then L[i,j] = max{L[i-1,j], L[i,j-1]} (we have no match here)

Answer is contained in L[n,m] (and the subsequence can be recovered from the L table).



L	-1	0	1	2	3	4	5	6	7	8	9	10	11
-1	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	1	1	1	1	1	1	1	1	1	1
1	0	0	1	1	2	2	2	2	2	2	2	2	2
2	0	0	1	1	2	2	2	3	3	3	3	3	3
3	0	1	1	1	2	2	2	3	3	3	3	3	3
4	0	**1**	1	1	2	2	2	3	3	3	3	3	3
5	0	1	1	1	**2**	2	2	3	4	4	4	4	4
6	0	1	1	2	2	**3**	3	3	4	4	5	5	5
7	0	1	1	2	2	3	**4**	4	4	4	5	5	6
8	0	1	1	2	3	3	4	5	**5**	5	5	5	6
9	0	1	1	2	3	4	4	5	5	5	6	6	**6**


0 1 2 3 4 5 6 7 8 9 10 11
Y=CGATAATTGAGA
X=GTTCCTAATA
0 1 2 3 4 5 6 7 8 9