Final review
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.

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

**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.
Topological sort

DAG. Directed acyclic graph.

Topological sort. Redraw DAG so all edges point up.

Solution. Reverse DFS Postorder!
Reverse DFS postorder in a DAG

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!
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.

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.

Brute force. Try all spanning trees?

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

- **grey vertices** are a cut defined by the vertices connected to one of the red edge's vertices.
- **obsolete edge** (gray)
- **next MST edge** is red
- **MST edge** (black)
- **graph edges** sorted by weight

```
<table>
<thead>
<tr>
<th>Edge</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-7</td>
<td>0.16</td>
</tr>
<tr>
<td>2-3</td>
<td>0.17</td>
</tr>
<tr>
<td>1-7</td>
<td>0.19</td>
</tr>
<tr>
<td>0-2</td>
<td>0.26</td>
</tr>
<tr>
<td>5-7</td>
<td>0.28</td>
</tr>
<tr>
<td>1-3</td>
<td>0.29</td>
</tr>
<tr>
<td>1-5</td>
<td>0.32</td>
</tr>
<tr>
<td>2-7</td>
<td>0.34</td>
</tr>
<tr>
<td>4-5</td>
<td>0.35</td>
</tr>
<tr>
<td>1-2</td>
<td>0.36</td>
</tr>
<tr>
<td>4-7</td>
<td>0.37</td>
</tr>
<tr>
<td>0-4</td>
<td>0.38</td>
</tr>
<tr>
<td>6-2</td>
<td>0.40</td>
</tr>
<tr>
<td>3-6</td>
<td>0.52</td>
</tr>
<tr>
<td>6-0</td>
<td>0.58</td>
</tr>
<tr>
<td>6-4</td>
<td>0.93</td>
</tr>
</tbody>
</table>
```
**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. [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
- 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
- 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
- 6-2 0.40
- 3-6 0.52
- 6-0 0.58
- 6-4 0.93
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 $e = v \rightarrow w$ 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 $T$

![Diagram showing connection weights and Prim's algorithm](image_url)

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

priority queue of crossing edges

<table>
<thead>
<tr>
<th>Edge</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-7</td>
<td>0.19</td>
</tr>
<tr>
<td>0-2</td>
<td>0.26</td>
</tr>
<tr>
<td>5-7</td>
<td>0.28</td>
</tr>
<tr>
<td>2-7</td>
<td>0.34</td>
</tr>
<tr>
<td>4-7</td>
<td>0.37</td>
</tr>
<tr>
<td>0-4</td>
<td>0.38</td>
</tr>
<tr>
<td>6-0</td>
<td>0.58</td>
</tr>
</tbody>
</table>
Proposition. Lazy Prim's algorithm computes the MST in time proportional to $E \log E$ in the worst case.

Pf.

<table>
<thead>
<tr>
<th>operation</th>
<th>frequency</th>
<th>binary heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>delete min</td>
<td>$E$</td>
<td>$\log E$</td>
</tr>
<tr>
<td>insert</td>
<td>$E$</td>
<td>$\log E$</td>
</tr>
</tbody>
</table>
› shortest path
Edge relaxation

Relax edge $e = v \rightarrow w$.

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

```java
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 \( \text{distTo}[\cdot] \) value).
- Add vertex to tree and relax all edges incident from that vertex.

<table>
<thead>
<tr>
<th>( v )</th>
<th>( \text{distTo}[v] )</th>
<th>( \text{edgeTo}[v] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.00</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
egin{align*}
4\to5 & : 0.35 \\
5\to4 & : 0.35 \\
4\to7 & : 0.37 \\
5\to7 & : 0.28 \\
7\to5 & : 0.28 \\
5\to1 & : 0.32 \\
0\to4 & : 0.38 \\
0\to2 & : 0.26 \\
7\to3 & : 0.39 \\
1\to3 & : 0.29 \\
2\to7 & : 0.34 \\
6\to2 & : 0.40 \\
3\to6 & : 0.52 \\
6\to0 & : 0.58 \\
6\to4 & : 0.93 \\
\end{align*}
\]
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.

```java
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];
```
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).

<table>
<thead>
<tr>
<th></th>
<th>sort key</th>
<th></th>
<th>sort key</th>
<th></th>
<th>sort key</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>d a b</td>
<td>0</td>
<td>d a b</td>
<td>0</td>
<td>a c e</td>
</tr>
<tr>
<td>1</td>
<td>a d d</td>
<td>1</td>
<td>c a b</td>
<td>1</td>
<td>a d d</td>
</tr>
<tr>
<td>2</td>
<td>c a b</td>
<td>2</td>
<td>e b b</td>
<td>2</td>
<td>b a d</td>
</tr>
<tr>
<td>3</td>
<td>f a d</td>
<td>3</td>
<td>a d d</td>
<td>3</td>
<td>b e d</td>
</tr>
<tr>
<td>4</td>
<td>f e e</td>
<td>4</td>
<td>f a d</td>
<td>4</td>
<td>b e e</td>
</tr>
<tr>
<td>5</td>
<td>b a d</td>
<td>5</td>
<td>b a d</td>
<td>5</td>
<td>c a b</td>
</tr>
<tr>
<td>6</td>
<td>d a d</td>
<td>6</td>
<td>d a d</td>
<td>6</td>
<td>d a b</td>
</tr>
<tr>
<td>7</td>
<td>b e e</td>
<td>7</td>
<td>f e d</td>
<td>7</td>
<td>d a d</td>
</tr>
<tr>
<td>8</td>
<td>f e d</td>
<td>8</td>
<td>b e d</td>
<td>8</td>
<td>e b b</td>
</tr>
<tr>
<td>9</td>
<td>b e d</td>
<td>9</td>
<td>f e e</td>
<td>9</td>
<td>f a d</td>
</tr>
<tr>
<td>10</td>
<td>e b b</td>
<td>10</td>
<td>b e e</td>
<td>10</td>
<td>f e d</td>
</tr>
<tr>
<td>11</td>
<td>a c e</td>
<td>11</td>
<td>a c e</td>
<td>11</td>
<td>f e e</td>
</tr>
</tbody>
</table>

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).
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.

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

Partitioning element

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

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...
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).
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**

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td>101</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>1111</td>
</tr>
<tr>
<td>C</td>
<td>110</td>
</tr>
<tr>
<td>D</td>
<td>100</td>
</tr>
<tr>
<td>R</td>
<td>1110</td>
</tr>
</tbody>
</table>

**Compressed bitstring**

```
011111110011001000111111100101
A B RA CA DA B RA !
```

---

**Codeword table**

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td>101</td>
</tr>
<tr>
<td>A</td>
<td>11</td>
</tr>
<tr>
<td>B</td>
<td>00</td>
</tr>
<tr>
<td>C</td>
<td>010</td>
</tr>
<tr>
<td>D</td>
<td>100</td>
</tr>
<tr>
<td>R</td>
<td>011</td>
</tr>
</tbody>
</table>

**Compressed bitstring**

```
11000111101011100110001111101
A B RA CA DA B RA !
```

---

---
Huffman codes

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, ...
Q. How to represent the prefix-free code?
A. A binary trie!
• Chars in leaves.
• Codeword is path from root to leaf.
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.

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

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

**Codeword table**

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>A B</td>
<td>81</td>
</tr>
<tr>
<td>B R</td>
<td>82</td>
</tr>
<tr>
<td>A C</td>
<td>83</td>
</tr>
<tr>
<td>A D</td>
<td>84</td>
</tr>
<tr>
<td>R A</td>
<td>85</td>
</tr>
<tr>
<td>A B R</td>
<td>86</td>
</tr>
<tr>
<td>A C A</td>
<td>87</td>
</tr>
<tr>
<td>A D A</td>
<td>88</td>
</tr>
<tr>
<td>A B R</td>
<td>89</td>
</tr>
<tr>
<td>R A B</td>
<td>8A</td>
</tr>
<tr>
<td>B R A</td>
<td>8B</td>
</tr>
<tr>
<td>A B R</td>
<td>88</td>
</tr>
<tr>
<td>A B R</td>
<td>8B</td>
</tr>
</tbody>
</table>

**LZW compression**

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 8B

LZW compression for A B R A C A D A B R A B R A B R A
Geometric search
**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).

**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).

**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. $R + \log N$. 

$\begin{array}{cccc}
a & b & c & d \\
e & & c & d \\
f & & g & h
\end{array}$

$\begin{array}{cccc}
a & & & \\
& d & e & f \\
& & b & c
\end{array}$

$\begin{array}{cccc}
& & & \\
& & & h
\end{array}$
Recursively partition plane into two halfplanes.
Data structure. BST, but alternate using $x$- and $y$-coordinates as key.

- Search gives rectangle containing point.
- Insert further subdivides the plane.
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). $R + \sqrt{N}$.
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). $N$. 
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.
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.
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.
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 $\Rightarrow$ one new pair of adjacent segments.
- Delete line segment $\Rightarrow$ two segments become adjacent
- Intersection $\Rightarrow$ swap adjacent segments.
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 $R \log N + N \log N$.

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.
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 \(x = \text{root}\);
while \((x \neq \text{null})\)
{
    if \((x.\text{interval}.\text{intersects}(lo, hi))\)
        return \(x.\text{interval}\);
    else if \((x.\text{left} == \text{null})\) \(x = x.\text{right}\);
    else if \((x.\text{left}.@max < lo)\) \(x = x.\text{right}\);
    else \(x = x.\text{left}\);
}
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 operation

<table>
<thead>
<tr>
<th>operation</th>
<th>brute</th>
<th>interval search tree</th>
<th>best in theory</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert interval</td>
<td>1</td>
<td>log N</td>
<td>log N</td>
</tr>
<tr>
<td>find interval</td>
<td>N</td>
<td>log N</td>
<td>log N</td>
</tr>
<tr>
<td>delete interval</td>
<td>N</td>
<td>log N</td>
<td>log N</td>
</tr>
<tr>
<td>find any interval that intersects (lo, hi)</td>
<td>N</td>
<td>log N</td>
<td>log N</td>
</tr>
<tr>
<td>find all intervals that intersects (lo, hi)</td>
<td>N</td>
<td>R log N</td>
<td>R + log N</td>
</tr>
</tbody>
</table>

**order of growth of running time for N intervals**
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

<table>
<thead>
<tr>
<th>problem</th>
<th>example</th>
<th>solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>1d range search</td>
<td><img src="image1.png" alt="Example 1D Range Search" /></td>
<td>BST</td>
</tr>
<tr>
<td>kd orthogonal range search</td>
<td><img src="image2.png" alt="Example KD Orthogonal Range Search" /></td>
<td>kd tree</td>
</tr>
<tr>
<td>1d interval search</td>
<td><img src="image3.png" alt="Example 1D Interval Search" /></td>
<td>interval search tree</td>
</tr>
<tr>
<td>2d orthogonal line segment intersection</td>
<td><img src="image4.png" alt="Example 2D Orthogonal Line Segment Intersection" /></td>
<td>sweep line reduces to 1D range search</td>
</tr>
<tr>
<td>2d orthogonal rectangle intersection</td>
<td><img src="image5.png" alt="Example 2D Orthogonal Rectangle Intersection" /></td>
<td>sweep line reduces to 1D interval search</td>
</tr>
</tbody>
</table>
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).
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).