Week 5: Heaps, intervals, and graph traversal

Priorities and spreading failures

Maintain the top k measurements as values arrive, and merge overlapping maintenance intervals. On a sensor map, identify separate failure regions and the time needed for failures to spread.

Full ordering, partial ranking, and connectivity need different state. Track a heap boundary, the end of a sorted interval, and traversal visits.

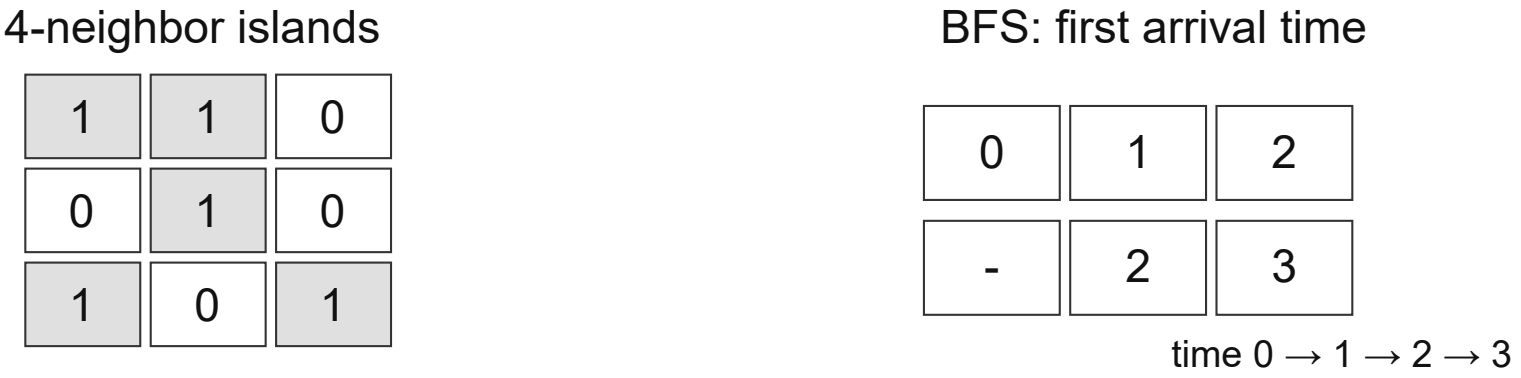
Learning outcomes

Sessions 9 and 10: Heap boundaries, interval unions, visited sets, and BFS levels

- Check contracts and boundary conditions.
- Trace state and implement in Python.
- Explain correctness, time, auxiliary space, and output space.

What counts as one step on a grid?

Treat cells as vertices and four-directional adjacencies as edges. Islands ask for connected components; propagation asks for distance from the nearest source.



Diagonal land cells on the left belong to separate islands. Numbers on the right are first infection times for `[[2,1,1],[0,1,1]]`; `-` marks an empty cell.

Goal	Processing order	Retained information
Find all connected land	Stack: depth first	Visited coordinates and pending neighbors
Spread simultaneously by minute	Queue: breadth first	Current sources and remaining fresh cells

Mark cells when discovered so different neighbors cannot enqueue or push the same cell repeatedly.

215 · Idea and execution trace

LeetCode 215: Kth Largest Element in an Array · Medium

Return the kth value in descending order, counting duplicates.

Sorting everything finds the kth value, but only the largest k values need to be retained. In a min-heap containing those values, the root is their minimum and hence the current kth largest. Replace it only when a larger value arrives.

`nums=[3,1,5,2,4], k=2`. The table shows the actual heap list, not a fully sorted sequence.

Operation	heap	Current kth value
Heapify first two	[1,3]	1
5 > 1: replace	[3,5]	3
2 <= 3: retain	[3,5]	3
4 > 3: replace	[4,5]	4

215 · Turning the idea into code

```
import heapq

def kth_largest(nums, k):
    heap = nums[:k]
    heapq.heapify(heap)
    for index in range(k, len(nums)):
        value = nums[index]
        if value > heap[0]:
            heapq.heapreplace(heap, value)
    return heap[0]
```

`nums[:k]` copies k entries before heapify. A min-heap is a complete binary tree whose parents are no larger than their children. Capacity doubles per level, giving O(log k) height; heapreplace may descend that height. Simply reading the root `heap[0]` costs O(1).

A value no larger than the root cannot improve the retained k values. Duplicates remain separate entries: the second largest in `[5,5,1]` is 5.

Heap positions and root replacement

In min-heap `[2,4,3,8,6,5,7]`, the children of position i are 2i+1 and 2i+2. Root 2 has children 4 and 3. Siblings need not be sorted; each parent must be no larger than its children.

Replace the root with 9 and trace a repair path that promotes the smaller child.

Step	Array	Next comparison
Replace root	[9,4,3,8,6,5,7]	choose 3 over 4
Promote 3	[3,4,9,8,6,5,7]	choose 5 over 7
Promote 5	[3,4,5,8,6,9,7]	9 is a leaf; stop

Reading root 3 takes O(1); repair follows a path of at most the tree height. heapify repairs parents from the bottom. With seven nodes, four leaves have height 0, two parents have height 1, and the root has height 2. This allows 2×1+1×2=4 downward levels in total. In general, the number of tall subtrees decreases geometrically, giving O(k) total work.

Python heapq representation and operations

215 · Why it works; time and memory

The heap retains the largest k values seen. A new value at most its minimum cannot improve that set; a larger one replaces the minimum. The final root is kth largest.

Copying and heapify cost O(k). Each of the remaining n-k values requires one root comparison and, only on replacement, a path of height O(log k). Time is O(k+(n-k)(1+log k)), commonly O(n log k) for k>=2. For k=1 it is O(n), like tracking a maximum.

The heap always has k entries, so auxiliary space is O(k); the result is one value. Heapify is linear because most nodes are near the bottom and have only short downward paths.

56 · Idea and execution trace

LeetCode 56: Merge Intervals · Medium

Return disjoint closed intervals covering the input union.

After sorting by start, the next interval begins no earlier than processed intervals. It can therefore overlap only the last merged interval. Extend that interval's end on overlap, or begin a new result interval when there is a gap.

Input: `[[1,4],[2,3],[4,6],[8,9]]`. Each row shows result after processing that interval.

Interval	Test	result
[1,4]	First interval	[[1,4]]
[2,3]	2 <= 4	[[1,4]]
[4,6]	4 <= 4	[[1,6]]
[8,9]	8 > 6	[[1,6],[8,9]]

56 · Turning the idea into code

```
def merge_intervals(intervals):
    result = []
    for start, end in sorted(intervals, key=lambda item: item[0]):
        if result and start <= result[-1][1]:
            result[-1][1] = max(result[-1][1], end)
        else:
            result.append([start, end])
    return result
```

The equality in `start <= result[-1][1]` merges closed intervals touching at an endpoint. Update with max so a nested `[2,3]` cannot shorten `[1,4]`.

Create each result interval as `[start,end]` rather than modifying an input interval's list.

56 · Why it works; time and memory

result equals the union of the processed prefix. Increasing starts mean a new interval can overlap only the last result; a gap makes a new disjoint component.

Sorting n intervals costs O(n log n). The subsequent scan runs n times and compares only one last result interval each time, so it costs O(n). Sorting dominates the total O(n log n).

`sorted` creates a list of n references and uses sorting workspace, giving O(n) auxiliary space. If z merged intervals remain, output space is O(z), with z<=n. Each output interval stores its own two endpoints.

200 · Idea and execution trace

LeetCode 200: Number of Islands · Medium

Count land components connected horizontally or vertically in a character grid.

Starting from one land cell and following all four-directional connections reaches its whole island. Start a traversal and increase the count only at unvisited land, preventing duplicate counting. Diagonals do not connect cells.

Grid: `[["1","1","0"],["0","1","0"],["1","0","1"]]`. Coordinates start at zero; stack top is on the right.

Action	stack	Newly added to seen	count
Start (0,0)	[(0,0)]	(0,0)	1
Pop (0,0)	[(0,1)]	(0,1)	1
Pop (0,1)	[(1,1)]	(1,1)	1
Pop (1,1)	[]	None	1
Traverse (2,0), (2,2) separately	[]	(2,0), (2,2)	3

200 · Turning the idea into code

```
def islands(grid):
    rows, cols = len(grid), len(grid[0])
    seen, count = set(), 0
    for row in range(rows):
        for col in range(cols):
            if grid[row][col] != "1" or (row, col) in seen:
                continue
            count += 1
            stack = [(row, col)]
            seen.add((row, col))
            while stack:
                r, c = stack.pop()
                for nr, nc in ((r-1,c),(r+1,c),(r,c-1),(r,c+1)):
                    if (0 <= nr < rows and 0 <= nc < cols
                            and grid[nr][nc] == "1" and (nr,nc) not in seen):
                        seen.add((nr,nc))
                        stack.append((nr,nc))
    return count
```

The outer loops locate traversal starts; the inner while finishes one connected island. Mark a cell in seen just before pushing it, preventing another neighbor from pushing it again.

Cells contain strings “1” and “0”. Check coordinate bounds before reading a cell. The input grid remains unchanged.

200 · Why it works; time and memory

A traversal reaches exactly its starting component. Marked components cannot restart; the number of starts equals the component count.

The outer scan reads R·C cells. Each land cell is pushed and popped once because of seen, and each pop checks four neighbors. Four is independent of input size, giving expected O(RC) time under expected hash-set costs. The inner traversal is not restarted fully for every cell.

Both seen and the pending stack can hold up to RC coordinates, so auxiliary space is O(RC). Marking the grid itself could replace seen, but would mutate the input and would not eliminate stack storage.

994 · Idea and execution trace

LeetCode 994: Rotting Oranges · Medium

Return the minimum minutes for all fresh cells to change under simultaneous spread, or -1.

All initially rotten cells are sources at time zero. Put them all in one queue and expand by levels. The first infection time of a cell is its distance from the nearest source. Processing sources sequentially would misrepresent simultaneous spread.

Input: `[[2,1,1],[0,1,1]]`. Each queue contains cells newly rotten at that time, awaiting processing.

Time minutes	queue	fresh
0	[(0,0)]	4
1	[(0,1)]	3
2	[(1,1),(0,2)]	1
3	[(1,2)]	0

Once fresh is zero, return 3 without processing the final queue.

994 · Turning the idea into code

```
from collections import deque
def rot_minutes(grid):
    rows, cols = len(grid), len(grid[0])
    board = [row[:] for row in grid]
    queue, fresh, minutes = deque(), 0, 0
    for r in range(rows):
        for c in range(cols):
            if board[r][c] == 2:
                queue.append((r,c))
            fresh += board[r][c] == 1
    while queue and fresh:
        for _ in range(len(queue)):
            r, c = queue.popleft()
            for nr, nc in ((r-1,c),(r+1,c),(r,c-1),(r,c+1)):
                if 0 <= nr < rows and 0 <= nc < cols and board[nr][nc] == 1:
                    board[nr][nc] = 2
                    fresh -= 1
                    queue.append((nr,nc))
        minutes += 1
    return minutes if fresh == 0 else -1
```

`range(len(queue))` freezes the source count for the current time. Mark newly infected cells as 2 immediately to avoid duplicate infection, but append them for processing in the next while round.

994 · Why it works; time and memory

BFS levels are minimum edge distances from any source. Equal edge durations make first discovery earliest; remaining fresh cells after exhaustion are unreachable.

Copying and the initial scan each cost O(RC). Every cell enters the queue once and checks four neighbors when removed, so propagation also costs O(RC). Adding these stages gives O(RC); do not multiply by the number of minutes again.

The board copy stores RC cells to preserve the input. The queue may also hold O(RC) cells, giving O(RC) auxiliary space. Constant-size fresh and minutes counters do not make the whole algorithm constant-space.

`board=[row[:] for row in grid]` copies each row as well, preserving the original cell values. `fresh += board[r][c] == 1` adds 1 for True and 0 for False. Scanning [2,1,1] therefore makes fresh go 0→1→2.

An empty queue with fresh cells remaining means unreachable cells and returns -1. If fresh is initially zero, no time step is needed and the result is 0.

Checkpoint questions

1. Find the answers for k=2 and k=5 in `[8,3,8,2,5]`.
2. Merge `[[1,8],[2,3]]` and, separately, `[[1,3],[3,6]]`.
3. Count components in `[["1","0"],["0","1"]]`.
4. Find spread times for `[[2,0,1]]` and `[[0,2]]`.

Exercises and sources

LeetCode · 215, 56, 200, 994 · Worked solutions · Python ZIP

```
python demo_09.py
python demo_10.py
python verify.py
```

After implementing the student file, check it with `python verify.py --student`.

Erik Demaine, Srini Devadas. MIT 6.006, Introduction to Algorithms (2011).