Heaps and Heap Sort

A Series of Exercises

Learn by implementing: from basic heapify to optimized algorithms

Contents

Exercise 1: Understanding the Heap Property

Concept

A binary heap is a complete binary tree stored in an array, where each node satisfies the *heap property*.

For a **max-heap**: Every parent node is ≥ its children.

For a **min-heap**: Every parent node is ≤ its children.

Key insight: Because it's a complete binary tree stored in an array, we can navigate parent-child relationships using arithmetic:

```
Given a node at index i (0-indexed):

  parent(i)      → (i - 1) // 2

  left_child(i)  → 2*i + 1

  right_child(i) → 2*i + 2
```

Your Task

Write a function `heapify_down(arr, n, i)` that takes:

• `arr`: the array representing the heap

• n: the size of the heap (only consider elements 0 to n-1)

• `i`: the index of a node that might violate the max-heap property

The function should restore the heap property for the subtree rooted at i by "sinking" the element down to its correct position.

Hints

1. Find which of three values is largest: the node itself, its left child, or its right child.
2. Be careful with bounds — check that children exist before comparing.
3. If the largest isn't the current node, swap and recurse on the affected subtree.
4. Don't swap during comparisons — first find which index has the largest value, then do a single swap.

Test Case

`arr = [1, 5, 3, 4, 2]`, call `heapify_down(arr, 5, 0)`

Expected result: `[5, 4, 3, 1, 2]` (valid max-heap)

Exercise 2: Building a Heap from an Arbitrary Array

Concept

To transform any array into a valid heap, we need to heapify multiple nodes. But we don't need to heapify all of them.

Key insight: Leaf nodes (the bottom half of the array) are trivially valid heaps of size 1. We only need to heapify the internal nodes, working from the bottom up.

Your Task

Implement `build_heap(arr)` that converts an arbitrary array into a max-heap in-place.

Hints

1. The last internal (non-leaf) node is at index (n // 2) - 1.
2. Work backwards from the last internal node to the root (index 0).
3. Call your heapify_down function on each internal node.
4. Think about why bottom-up order matters: children must be valid heaps before their parent is heapified.

Test Case

```
arr = [3, 1, 6, 5, 2, 4]
```

Expected result: `[6, 5, 4, 1, 2, 3]` (valid max-heap)

Exercise 3: Heap Sort

Concept

Heap sort works in two phases: first build a max-heap, then repeatedly extract the maximum.

Key insight: After building a max-heap, the largest element is at index 0. We swap it to the end (its final sorted position), shrink the heap size by 1, and heapify the root to restore the heap property. Repeat until done.

Visual intuition:

```
[  heap portion  | sorted portion ]
      ↓ shrinks       ↓ grows
```

Your Task

Implement `heap_sort(arr)` that sorts the array in ascending order in-place.

Hints

1. First call build_heap to create a valid max-heap.
2. Loop from the last index down to 1.
3. Each iteration: swap arr[0] with arr[i], then heapify_down with size i (not n).
4. Using a max-heap produces ascending order because you're placing elements from the end.

Test Case

`arr = [3, 1, 6, 5, 2, 4]`

Expected result: `[1, 2, 3, 4, 5, 6]`

Exercise 4a: Optimization — Iterative Heapify

Concept

Your current heapify_down is recursive. Each recursive call has overhead (stack frames, function calls). For large arrays or deep heaps, this adds up.

Your Task

Convert your recursive `heapify_down` to an iterative version using a while loop.

Hints

1. Replace the recursive call with a while True loop.
2. Instead of recursing with heapify_down(arr, n, largest), just update i = largest and continue the loop.
3. Break out of the loop when largest == i (heap property satisfied).
4. The logic inside the loop is identical to the recursive version.

Test Case

Use the same test cases as Exercise 1 — behavior should be identical.

Exercise 4b: Optimization — The Hole Technique

Concept

The problem with swaps: Each swap involves 3 operations. As an element sinks down multiple levels, we're doing redundant writes.

Consider sinking a value down 3 levels with swaps: that's 9 writes total.

The insight: We only care about the initial value's final position. Instead of swapping, we can: (1) save the initial value, (2) move children up into the "hole" as we descend, (3) place the saved value once at the end. This reduces 9 writes to just 4.

Your Task

Implement `heapify_down_hole(arr, n, i)` using the hole technique.

Hints

1. Save arr[i] to a variable called 'value' at the start.
2. Instead of swapping, just do: arr[i] = arr[larger_child] (move child up).
3. Compare against 'value' (the saved element), not arr[i] — because arr[i] now contains something else.
4. After the loop, place the saved value: arr[i] = value.
5. Add a break condition: if arr[larger_child] <= value, we've found the right spot.

Test Case

Use the same test cases — behavior should be identical, just more efficient.

Exercise 4c: Optimization — Floyd's Bottom-Up Heapify

Concept

The observation: In standard heapify, we do 2 comparisons per level (left child vs current, right child vs current). But during heap sort, the element we're sinking is usually small and will likely sink all the way to the bottom.

Floyd's insight: Instead of comparing the sinking element at each level, we: (1) descend first — follow the path of larger children all the way down (1 comparison per level), (2) bubble up — then walk back up to find where our saved value actually belongs. This reduces comparisons from 2 per level to ~1 per level on average.

Your Task

Implement `floyds_heapify(arr, n, i)` for a max-heap.

Hints

1. Phase 1 (descend): Save arr[i], then follow larger children to a leaf, moving each child up.
2. In Phase 1, only compare children against each other (1 comparison), not against the value.
3. Phase 2 (bubble up): Walk back toward the root, comparing value against parent.
4. Remember: parent(i) = (i - 1) // 2
5. Recalculate left_child and right_child inside the loop — they change as i changes!

Test Case

Use the same test cases — behavior should be identical.

Exercise 5: Implementing a Priority Queue

Concept

A priority queue is an abstract data type where each element has a priority, and you can efficiently: (1) insert a new element, (2) extract the highest priority element. A heap is the classic implementation — it gives O(log n) for both operations.

Operation	**Description**	**Time**
insert(value)	Add element, maintain heap property	O(log n)
extract_max()	Remove and return largest element	O(log n)
peek()	Return largest without removing	O(1)

Key Insight: True "Bubble Up"

For insertion, we need a different operation than heapify_down. We append the new element to the end, then "bubble it up" by comparing with its parent and swapping if larger. This is the opposite direction of sinking.

Your Task

Implement a `PriorityQueue` class with insert, extract_max, and peek methods.

Hints

1. Use a list as the underlying heap storage.
2. insert: append to end, then bubble_up from the last index.
3. bubble_up: compare with parent, swap if larger, move to parent index, repeat until heap property satisfied.
4. extract_max: save root, move last element to root, pop the last, then sink_down from index 0.
5. Handle edge cases: empty heap for extract_max and peek.
6. You can reuse your Floyd's heapify for sink_down, but bubble_up is new — it only uses Phase 2 logic.

Test Case

```
pq = PriorityQueue()
```

```
pq.insert(3); pq.insert(1); pq.insert(6)
pq.insert(5); pq.insert(2); pq.insert(4)
pq.peek()        # → 6
pq.extract_max() # → 6
pq.extract_max() # → 5
pq.extract_max() # → 4
```

Exercise 6: Time Complexity Analysis

Concept

A common interview question: Why is build_heap O(n) instead of O(n log n)?

At first glance, it seems like it should be O(n log n): we call heapify_down on n/2 nodes, and each heapify_down is O(log n), so... O(n/2 × log n) = O(n log n)?

But the actual complexity is **O(n)**. Your task is to figure out why.

Key Insight

Not all nodes sink the same distance. Most nodes are near the bottom of the tree, where they have very little distance to sink.

Your Task — Part A

Calculate the total work for a heap of size n = 15. Fill in this table:

Level	Nodes at level	Max sinks per node	Total work
0	1	3	?
1	2	2	?
2	4	1	?
3	8	0	?
		TOTAL:	?

Your Task — Part B

For a general heap of height h, the total work is:

`Σ (n / 2^(k+1)) × k` for k from 0 to h

Simplify this sum and show it's O(n).

Hints

1. Part A: Multiply nodes × sinks for each level, then add up.
2. Part A: Compare your answer to the naive upper bound (n × log n = 15 × 3 = 45).

3. Part B: Factor out n/2 from the sum.
4. Part B: You'll get a sum of the form $\Sigma k \times x^k$ — this is a known series.
5. Part B: The formula for $\Sigma k \times x^k$ (from $k=0$ to ∞) is $x / (1-x)^2$ when $|x| < 1$.
6. Part B: With $x = 1/2$, what does this converge to?

The Intuition

The "bottom-heavy" nature of binary trees saves us. Most nodes are leaves that do nothing, and the few nodes near the top that sink far are vastly outnumbered by the many nodes near the bottom that sink only a little.