Free sample questions

Question 1 part a (sorting)

🧮 **Question (Part A): Partially Sorted Heap Levels**

🧠 Problem Statement

We are given a binary heap (complete binary tree) that contains n elements.
The heap has the following special property:

- **All even-numbered levels (level 0, 2, 4, …) are sorted from left to right.**
- **All odd-numbered levels (level 1, 3, 5, …) are not sorted in any way.**

It is also known that the average value of all elements in the heap equals `n^100`.

🎯 Task

Design the most efficient algorithm possible to sort all n elements of the heap into non-decreasing order, given the above structure.

⏱️ Required Analysis

1. **Describe the time complexity of your algorithm using asymptotic notation (`O`, `Θ`, `Ω`).**
2. **Justify why the chosen algorithm is asymptotically optimal under the given conditions.**

📋 Output

Provide only the algorithm design and asymptotic analysis — no implementation code is required.

---

solution question 1 part a

🧩 Solution (Part A): Using the Bounded Average for Linear-Time Sorting

🧠 Given Conditions

- **The heap has n elements.**
- **All even levels are sorted; odd levels are not.**
- **The average value of all elements equals n^{100}.**

Therefore, if all values are non-negative integers, the total sum is

$$\sum a_i = n \cdot n^{100} = n^{101}$$

- **implying that the maximum possible value satisfies $\max \le n^{101}$.**

Hence, all keys lie within a polynomial range

$$0 \le a_i \le n^{101}$$

which allows the use of linear-time integer sorting.

⚙️ Algorithm: Radix (LSD) or Counting Sort

We ignore the internal heap structure and exploit the bounded domain.

1. **Representation.**
   Represent each integer a_i in base $B = n$ (or equivalently, $B = 2^{\lceil \log_2 n \rceil}$).

Number of digits.
Since $U \le n^{101}$, the number of base-B digits is

$$\log_B(U) = \log_n(n^{101}) = 101$$

2. — a fixed constant.

3. Sorting process.
Perform LSD Radix Sort, where each pass uses a stable Counting Sort on one digit.
   - Each Counting Sort costs $O(n + B) = O(n)$ (since $B = n$).
   - Total number of passes = 101 (constant).
   - Total running time: $O(101 \cdot n) = O(n)$.

4. Memory usage.
Each pass uses auxiliary arrays of size $O(B) = O(n)$, so total space is $O(n)$.

🧮 Complexity Summary

Operation	Time	Space	Notes
Counting Sort (per pass)	$O(n)$	$O(n)$	Stable
101 passes total	$O(n)$	$O(n)$	Constant factor
Final merged result	$O(n)$	—	Sorted array

🧾 Correctness

- Counting Sort guarantees stable sorting within each digit.
- LSD Radix Sort preserves global nondecreasing order across all 101 digits.

- **Since `U` is polynomially bounded and the number of passes is constant, the final array is globally sorted in linear time.**

⚠️ Edge Cases

- **If negative integers exist, shift all keys by `+|min|` to make them non-negative, or sort positive and negative parts separately, each in linear time, and merge.**
- **If keys are non-integers, Radix/Counting does not apply directly — revert to the `O(n log n)` comparison-based method.**

✅ Final Result

Under the assumption of non-negative integer keys with an average value of `n^100`,
the key domain is polynomially bounded (`U ≤ n^101`), so:

```
Total sorting time:  Θ(n)
Total space:         Θ(n)
```

This is asymptotically optimal for integer sorting and strictly better than the `Θ(n log n)` bound of comparison-based methods.

Question 1 part b (sorting)

🧮 Question: Sorted Dynamic Array with Partial Sorting

🧠 Problem Statement

We maintain a sorted dynamic array that supports element insertions.
Initially, the array is empty and has a fixed capacity of 1.

Each insertion works as follows:

1. If there is free space, the new element is inserted into the array in its correct sorted position.
2. If the array becomes full after the insertion:
   - Sort the newly inserted right half of the array.
   - Merge the right half with the already sorted left half.
   - Double the array's capacity.
   - Copy all elements into the new larger array.

⚙️ Operations

- `insert(x)` – inserts element `x` into the sorted dynamic array, following the above rules.

🎯 Required Analysis

1. Find the worst-case time complexity of a single insertion operation.
2. Determine the total time complexity for performing `n` insertions starting from an empty array.

⏱️ Expected Output

Express both results using asymptotic notation (`Θ`, `O`, `Ω`) and provide clear justification for the growth rate.

Solution question 1 part b

🧩 Solution: Sorted Dynamic Array with Partial Sorting

✅ Assumptions & Invariant

- We maintain an array with capacity `C` (starting at `1`), size `t` (`0 ≤ t ≤ C`).
- Invariant between rebuilds:
  - The left half `[0 .. ⌊C/2⌋-1]` is sorted.
  - The right half `[⌊C/2⌋ .. t-1]` contains the recently appended elements (not necessarily sorted).
- When the array becomes full (`t = C`), we finish sorting the current block and rebuild.

⚙️ Algorithm: `insert(x)`

1. If `t < C` (has room):
   - `append x` to position `t`.
   - `t ← t + 1`.
   - *(No shifting; we do not maintain global sortedness continuously—only at rebuilds.)*
2. If `t = C` (trigger rebuild):
   - Let `L := A[0 .. C/2-1]` (already sorted by the previous rebuild), `R := A[C/2 .. C-1]` (unsorted block just filled).
   - Sort `R` by any comparison sort: `Θ((C/2) log(C/2)) = Θ(C log C)`.

- Merge `L` and sorted `R` into a single sorted array of size `C`: `Θ(C)`.
- Allocate new array of capacity `2C` and copy the `C` sorted elements: `Θ(C)`.
- Set `C ← 2C`, `t` remains `C/2 + C/2 = C` after the merge-copy (array is now globally sorted again, left half will serve as the next “frozen” sorted prefix).

After each rebuild, the entire prefix of length `t` is sorted; between rebuilds, only the left half is sorted, and we accumulate unsorted elements in the right half until the next rebuild.

🧠 Correctness Sketch

- Base: Initially `C=1`, after first rebuild (if any), the array of length `C` is fully sorted.
- Maintenance: Between rebuilds, the left half remains untouched (thus sorted). New items are appended to the right half.
- Rebuild step: Sorting `R` and merging with `L` yields a globally sorted array of length `C`.
- Progress: Capacity doubles; hence rebuilds are finite and happen at sizes `1,2,4,8,…`.

⏱️ Complexity Analysis

- Single insertion (non-rebuild): `O(1)` (pure append).
- Single insertion that triggers rebuild at capacity `C`:
  - Sorting the right half: `Θ(C log C)`
  - Merging halves: `Θ(C)`

- Alloc+copy to capacity 2C: $\Theta(C)$
- Worst case: $\Theta(C \log C)$ (dominated by the sort).

Total for n insertions (starting empty):
Rebuilds occur at capacities 1, 2, 4, …, 2^k ≈ n. The cumulative cost is

$\Sigma_{i=0}^{\lfloor \log_2 n \rfloor} \Theta(2^i \cdot i) = \Theta(n \log n)$.

- **Non-rebuild inserts contribute $O(n)$ and are dominated.**

Amortized per insertion:

$\Theta(n \log n) / n = \Theta(\log n)$.

-

📌 Final Results (Asymptotics)

- **Worst-case time of a single insert: $\Theta(C \log C)$ when it triggers a rebuild at capacity C.**
- **Total time for n inserts: $\Theta(n \log n)$.**
- **Amortized time per insert: $\Theta(\log n)$.**

Question 1 part c (sorting)

🧮 Question (Part C): Dynamic Sorted Array – Alternative Construction

---

🧠 Problem Statement

We wish to build a dynamic sorted array using the following algorithm:

1. **Each time a new element x is inserted, it is simply appended to the end of the array.**

2. **The left half of the array is always assumed to be sorted, while the right half contains the newly inserted unsorted elements.**

3. **For each newly inserted element x, we perform a binary search on the left (sorted) half to find the position where x should appear in sorted order.**

4. **We then store an auxiliary field in x that records this target position.**

5. **Once the array becomes full, we double its capacity, and for each element in the right half, we move it to the position indicated by its stored index.**

6. **Any remaining empty locations in the left half are filled in place with the corresponding elements during the copy process.**

🎯 Required Tasks

1. **Determine whether this algorithm always produces a correctly sorted dynamic array once the array becomes full.**

2. **If it is correct, prove its correctness formally and analyze the total time complexity for n insertions.**

3. **If it is not correct, provide a counterexample that shows the failure, explain why it occurs, and propose a modification that fixes the algorithm while keeping it as efficient as possible.**

⏱️ Expected Output

Provide a clear proof or counterexample, supported by asymptotic analysis (Θ, O, Ω), and discuss the resulting time complexity of all n insertions.

Solution question 1 part c

🧩 **Solution (Part C): Dynamic Sorted Array – Alternative Construction**

❌ Claim Check: The Proposed Algorithm Is *Not* Correct

Counterexample. Consider capacity `C = 4` at the moment it becomes full.
Left (sorted) half: `[2, 100]`.
Right (unsorted) half (in arrival order): `[60, 50]`.

Both `60` and `50` fall—by binary search over the left half—into the *same* target interval `(2, 100)`. If we only "store" that interval/position and later place both according to their stored positions *without ordering them relative to each other*, we may realize the final layout `[2, 60, 50, 100]`, which is not sorted.
Therefore, the algorithm does not guarantee a sorted array upon expansion.

🛠️ Fix That Makes It Correct

To ensure correctness while keeping the same spirit:

1. **Interval assignment:** For each element in the right half, compute its target interval (via binary search over the left half).

2. **Intra-interval ordering:** Before the rebuild, sort the elements of the right half (globally, or per-interval buckets).

3. **Stable merge by intervals:** Rebuild by stable merging each left-half segment with its corresponding (now sorted) right-half elements.

This guarantees that within every interval the relative order is ascending, and concatenating the intervals yields a globally sorted array.

✅ Correctness (Sketch)

- The left half is sorted by invariant.

- After sorting the right half (globally or per-interval), all elements mapped to a given interval are in nondecreasing order.
- A stable merge of each interval's left-half elements with its right-half elements preserves sortedness within the interval.
- Concatenating intervals in left-to-right order yields a fully sorted array. ✓

⏱️ Time Complexity for n Insertions

We analyze rebuilds at capacities `1, 2, 4, 8, …, 2^k ≈ n`.

- At capacity `C`:
  - Sorting the right half (≈ `C/2` items): `Θ((C/2) · log(C/2)) = Θ(C log C)`
  - Stable merging halves: `Θ(C)`
  - Allocating new array of size `2C` and copying: `Θ(C)`
  - Total per rebuild: `Θ(C log C)` (dominated by sorting)
- Summation over rebuilds: $\sum_{i=0 \text{ to } \lfloor \log 2n \rfloor} \Theta(2i \cdot i) = \Theta(n \log n)$..
- Non-rebuild insertions: `O(1)` each; total `O(n)` and dominated.

Final bounds:

- n insertions: `Θ(n log n)`
- Amortized per insertion: `Θ(log n)`
- Worst case for an insertion that triggers rebuild at capacity `C`: `Θ(C log C)`

📌 Summary

- As stated, the algorithm is incorrect (counterexample above).
- With the fix (sorting the right half and stable interval-wise merging), it becomes correct with total insertion cost `Θ(n log n)` and amortized `Θ(log n)`.

Question 2 part a (heaps)

🧮 Section A — Formal Question

You are given a static binomial min-heap containing `n` elements.
No insertions or deletions are allowed.

Design a data structure that supports the following two operations efficiently:

1. `increase(x, k)` —
   Given a pointer to a node `x`, add the value `k` to every key in the subtree rooted at `x`.

2. `Return(x)` —
   Given a pointer to a node `x` whose original key was `x`, return its current key after all previous `increase` operations have been applied.

The goal is to achieve the lowest possible time complexity for both operations while keeping the total memory usage within `O(n)`.

Solution question part a

🧠 Formal Solution — Using Only the Heap Structure

We are given a static binomial min-heap (no insertions or deletions).
We will not use any external data structures** — the solution works entirely within the heap itself.

⚙️ Data Fields

Each node `v` in the heap stores:

- `base[v]`: its original key.
- `parent[v]`: pointer to its parent node (naturally available in a binomial heap).
- `add[v]`: a local increment field, initially `0`.

➕ Operation — `increase(u, k)`

Add `k` to every node in the subtree rooted at `u`.

We simply record this increment locally at `u`:

```
increase(u, k):

    add[u] += k
```

Time: `O(1)`

🔎 Operation — `Return(x)`

Return the current key of node `x` after all previous `increase` operations.

We accumulate the increments of all ancestors (including `x` itself):

```
Return(x):

    sum ← 0

    v ← x

    while v ≠ null:

        sum ← sum + add[v]

        v ← parent[v]
```

```
    return base[x] + sum
```

Time: $O(height) = O(\log n)$ in a binomial heap.
Space: $O(n)$ total (one add field per node).

✅ Correctness

Each `increase(u, k)` conceptually adds k to all descendants of u.
For any node x, its logical key equals:

```
key(x) = base[x] + Σ add[a]   over all ancestors a on the path
root → x (including x)
```

The `Return(x)` operation explicitly sums these contributions, producing the exact updated key.

⏱️ Complexity Summary

Operation	Time	Space	Description
`increase(u,k)`	$O(1)$	$O(n)$	Adds k to all descendants lazily.
`Return(x)`	$O(\log n)$	$O(n)$	Sums increments along path to root.

🧩 Notes

- This solution uses only the parent pointers already available in a binomial heap.

- No additional trees, segment structures, or index arrays are needed.
- Optional optimization: during traversal (e.g., inside `Return`), you may "push" the accumulated add down to children to reduce future path costs — without affecting asymptotic bounds.

Question 2 part b (heaps)

➖ **Section B — Formal Question (`decrease`)**

You are given the same static binomial min-heap from Section A (no insertions, deletions, or melds).
The topology of the forest is fixed, and each node can be referenced by a pointer.

Add support for the following operation, while keeping `Return(x)` from Section A:

1. `decrease(x, k)` —
 Given a pointer to a node `x` and a non-negative number `k`, subtract `k` from the key of every node in the subtree rooted at `x`.

Requirements (no solution requested):

- Maintain correctness of `Return(x)` (it should return the current key of `x` after any sequence of `increase`/`decrease` operations).
- Aim for the best possible asymptotic time bounds per operation under `O(n)` total space.
- The binomial min-heap property must remain logically consistent with the updated keys (you do not need to restructure the heap since the topology is static).
- Clearly state the time and space complexities you achieve for `decrease` and `Return`.

Solution question 2 part b

➖ Formal Solution — Using a Reduction Field and Subtree Splitting

We are given a static binomial min-heap (a forest of binomial trees).
We extend the previous solution to support the `decrease(x, k)` operation, where we must subtract `k` from all nodes outside the subtree of `x`, while keeping the structure itself unchanged.

⚙️ Key Idea

Instead of explicitly visiting all nodes outside the subtree, we:

1. **Split the heap into two parts:**
    - `H_in`**: the subtree rooted at** `x` **(the "protected" part — no decrease).**
    - `H_out`**: the remaining forest (all other trees).**
2. **Maintain for each heap component (tree root) a reduction field** `red[root]` **that represents the global decrease applied to that entire component.**
3. **When we later merge the two parts, we correct the key of the protected subtree's root to preserve the real minimum order.**

🏗️ Data Fields

Each root in the binomial forest stores:

- `offset[root]` **— the total global shift (reduction or addition) applied to all keys in its component.**
- **Each node keeps its usual:**
    - `base[v]` **— original key,**

- `add[v]` — lazy increment field (from Section A),
- `parent[v]` — pointer to parent node.

➕ Operation — `decreaseOutside(x, k)`

1. Split the heap into:
   - the subtree `H_in` rooted at `x`;
   - the remaining forest `H_out`.

Apply the global decrease:

```
offset[H_out] -= k
```

2. (this conceptually subtracts `k` from all nodes outside the subtree).
3. When re-merging `H_in` and `H_out`:
   - the root of `H_in` preserves its original offset (since it was not decreased);

when linking two roots `r1` and `r2`, always compare

```
key(r) = base[r] + add[r] + offset[root_of_tree(r)]
```

- so that comparisons remain consistent even under different offsets.

🔎 Operation — `Return(x)`

To return the current key of node `x`:

1. Start from `x`, follow parent pointers to its root `r`.

2. Accumulate along the path:
    - all `add[v]` values (as in Section A),
    - plus the `offset[r]` of the root.

Return:

```
key(x) = base[x] + Σ add[a]  (ancestors a from root→x) + offset[root(x)]
```

3.

✅ Correctness

- The split ensures that only the complement of the subtree receives the global decrease.
- Each tree root stores an offset that affects all its descendants uniformly.
- During merges, we perform a rebase step: when a tree with offset α becomes a child of another tree with offset β, we keep the parent's offset as the unified value and store a lazy difference `(α - β)` at the losing root, preserving logical consistency.

⏱️ Complexities

Operation	Time	Description
`increase(u, k)`	`O(1)`	As before — add to `add[u]`.
`decreaseOutside(x, k)`	`O(log n)`	Split + offset update + merge.

`Return(x)`	`O(log n)`	Sum of local adds + root offset.
Space	`O(n)`	One `offset` per root, one `add` per node.

💡 **Summary**

This approach:

- **Keeps all logic inside the heap structure (no auxiliary trees or arrays).**
- **Handles both `increase` and `decrease` through small constant-size fields (`add`, `offset`).**
- **Preserves min-heap order automatically, since every comparison uses `base + add + offset`.**
  Thus, `decreaseOutside` is supported in `O(log n)` time with `O(n)` total space.

Question 3 part a (binary search trees)

📘 Section A — Problem Statement (Sequence with `insert`, `get`, `shift`)

🎯 Goal

Design a data structure that maintains an ordered sequence of elements and supports the following operations efficiently.

🧩 Universe & Notation

- **The element domain is arbitrary (denote it by Σ).**

- Sequence length at any time is `n ≥ 0`.
- Indices are 1-based unless stated otherwise.
- Let `A[1..n]` denote the current sequence.

🔧 Operations (to be supported)

- `insertLast(x)`
  - Effect: Append element `x ∈ Σ` to the end of the sequence.
  - Post-state: Sequence becomes `A[1..n]·x` (length `n ← n+1`).
- `get(i)`
  - Input: Index `i`.
  - Precondition: `1 ≤ i ≤ n`.
  - Output: Return the element `A[i]`.
  - No modification to the sequence.
- `shift(i, x)`
  - Input: Index `i`, element `x ∈ Σ`.
  - Precondition: `1 ≤ i ≤ n+1`.
  - Effect: Insert `x` at position `i`, shifting the current suffix `A[i], A[i+1], …, A[n]` one step to the right.
  - Post-state:
    - If `i ≤ n`: new sequence is `A[1..i-1] · x · A[i..n]`.
    - If `i = n+1`: equivalent to `insertLast(x)`.

- Length update: `n ← n+1`.

⏱️ Required Performance Targets

- Each operation `insertLast`, `get`, `shift` must run in `O(log n)` time (worst-case or amortized; specify your chosen model).
- Space usage over `n` elements is `O(n)`.
- The interface must handle up to `Q` operations with the above bounds.

✅ Correctness Requirements

- `get(i)` must return exactly the element at logical position `i` after all prior updates.
- `shift(i, x)` must preserve the relative order of all pre-existing elements.
- Edge conditions must be validated (e.g., index bounds).

📥 Inputs & 📤 Outputs (abstract API)

- Inputs: A sequence of operation calls of the forms
  - `insertLast(x)`
  - `get(i)`
  - `shift(i, x)`
- Outputs: For each `get(i)` call, output exactly one element `∈ Σ`. Other operations produce no output.

📝 Notes & Conventions

- Duplicates are allowed: elements of `Σ` need not be distinct.
- The data structure should be generic over `Σ` (no assumptions on value range).
- If you adopt amortized bounds, clearly state the potential argument or accounting method (outside of this section).

🔒 Robustness (Index Policy)

- If a call violates the precondition (e.g., `i` out of range), the behavior is undefined or should raise an explicit error (choose and document one policy).

Solution question 3 part a

🧠 Solution — Sequence with `insertLast`, `get`, `shift`

📦 Data Structure (High-Level)

Maintain the sequence in an implicit balanced binary tree (e.g., AVL or 2–3 tree) where inorder yields the current order of elements.
Each node stores:

- `val` — the element,
- `left`, `right` — child pointers,
- `size` — number of elements in the subtree (order-statistics key).

> **Any worst-case balanced option is fine (AVL or 2–3). We describe with AVL terminology; the same logic holds for a 2–3 tree.**

🧰 Invariants & Helpers

Invariants

- `size(u) = size(u.left) + 1 + size(u.right)`
- **Inorder traversal equals the current sequence.**

Helper primitives (both `O(log n)` worst case):

- `split(T, k)` ⇒ **returns `(L,R)` where `L` holds the first `k` elements (positions `1..k`), and `R` holds the rest (`k+1..`). Structure remains balanced.**
- `join(A,B)` ⇒ **concatenation preserving order: inorder is exactly `inorder(A) · inorder(B)`; structure remains balanced.**

Order-statistics query (`kth`) in `O(log n)`

```
kth(T, k):
  let L = size(T.left)
  if k == L+1: return T.val
  if k <= L:   return kth(T.left, k)
  else:        return kth(T.right, k - L - 1)
```

➕ Operation `insertLast(x)` — Append

Idea: Concatenate a single-node tree at the end.

```
T ← join(T, node(x))
```

- **Correctness: By `join`, the inorder becomes previous sequence followed by `x`.**

- Time: `O(log n)`.

🔎 Operation `get(i)` — Access by index

Idea: Order-statistics descent using subtree sizes.

```
return kth(T, i)
```

- **Correctness:** By the definition of `kth`, we return exactly the element at logical position `i`.
- **Time:** `O(log n)`.

📍 Operation `shift(i, x)` — Insert at position `i`

Goal: Insert `x` before the current element at position `i` (1-based). If `i = n+1`, this is exactly `insertLast(x)`.

Using `split/join`:

```
(L, R)  ← split(T, i-1)      // L: positions 1..i-1,   R:
positions i..n
X       ← node(x)            // single-node tree
T       ← join( join(L, X), R )
```

- **Correctness:**
    - `split` isolates the prefix `A[1..i-1]` from the suffix `A[i..n]`.

- Concatenating `L`, `X`, then `R` yields the sequence `A[1..i-1] · x · A[i..n]`.

- Time: `O(log n)` for one `split` and two `join` calls.

✅ Correctness Argument (Sketch)

- Structure: The implicit tree stores only local sizes; inorder order is preserved by construction.

- `split` soundness: For any `k`, the inorder of the left result is the first `k` elements; the right result is the remaining suffix. No elements are duplicated or lost.

- `join` soundness: Inorder of `join(A,B)` is exactly concatenation of their inorders; no reordering occurs.

- Operations:

  - `insertLast` is a direct concatenation → append is correct.

  - `get` follows the unique path determined by subtree sizes → returns the element at index.

  - `shift` uses `split` at `i-1` then concatenates a singleton before the old suffix → inserts at the desired position and preserves relative order.

⏱️ Complexity

- Each primitive `split` / `join` / `kth` runs in `O(log n)` worst case (AVL height or 2–3 height is Θ(log n)).

- Therefore:

- `insertLast` — `O(log n)`
- `get` — `O(log n)`
- `shift` — `O(log n)` (one `split` + two `join`)

Space is `O(n)` for n elements (balanced tree nodes), plus `O(1)` auxiliary per operation.
If persistent (copy-on-write) nodes are used, each update allocates only `O(log n)` new nodes while sharing the rest.

🧪 Edge Policy

- Indices are 1-based.
- Precondition checks (recommended):
  - `get(i)`: require 1 `≤ i ≤ n`.
  - `shift(i,x)`: require 1 `≤ i ≤ n+1`.
  - On violation: raise a well-defined error.

📝 Notes

- You may implement `split/join` directly for AVL or leverage a 2–3 tree where concatenation and splitting are particularly natural; both give the same asymptotic guarantees.
- This section provides the solution design only (no code).

Question 3 part b (binary search trees)

📘 Section A — Problem Statement (Sequence with `insertLast`, `get`, `Duplicate`)

🎯 Goal

Design a data structure that maintains an ordered sequence and supports efficient append, random access by index, and interval duplication.

🧩 Universe & Notation

- **Element domain: arbitrary** `Σ`**.**
- **Sequence length:** `n ≥ 0`**.**
- **Indices are 1-based.**
- **Current sequence:** `A[1..n]`**.**
- **For** `1 ≤ i ≤ j ≤ n`**, denote the contiguous block by** `A[i..j]`**.**

🔧 Operations to Support

- `insertLast(x)`
  - **Effect: Append** `x ∈ Σ` **to the end of the sequence.**
  - **Post-state:** `A ← A · x`**, length** `n ← n+1`**.**
- `get(k)`
  - **Input: index** `k`**.**
  - **Precondition:** `1 ≤ k ≤ n`**.**
  - **Output: Return** `A[k]`**.**

- No modification to the sequence.

- `Duplicate(i, j)`
  - Input: indices `i, j` with `1 ≤ i ≤ j ≤ n`.
  - Effect: Insert a second copy of the block `A[i..j]` immediately after position `j`.
  - Formally: After the operation,
    `A ← A[1..i-1] · A[i..j] · A[i..j] · A[j+1..n]`
    and `n ← n + (j - i + 1)`.

✅ Correctness Requirements

- `get(k)` returns exactly the element located at logical position `k` after all prior updates.
- `Duplicate(i, j)` places two consecutive copies of the pre-state block `A[i..j]` at positions `i..j` and `j+1..j+(j-i+1)` in the post-state, while preserving the relative order of all other elements.
- Multiple operations must compose correctly on the evolving sequence.

⏱️ Performance Targets

- Each operation must run in `O(log n)` time (worst-case or amortized; specify the chosen model elsewhere).
- Space over `n` elements: `O(n)`.
- Additional space per update call (metadata / restructuring): `O(log n)`.

🔒 Preconditions & Edge Policy

- `insertLast(x)`: always valid.

- `get(k)`: require `1 ≤ k ≤ n`.
- `Duplicate(i, j)`: require `1 ≤ i ≤ j ≤ n`.
- On violation: behavior is undefined or raise a well-specified error (choose one policy).
- Corner cases to be handled: `i = 1`, `j = n`, `i = j`.

📥 Inputs & 📤 Outputs (abstract API)

- Inputs:
  - `insertLast(x)`
  - `get(k)`
  - `Duplicate(i, j)`
- Outputs: For each `get(k)`, output exactly one element `∈ Σ`. Other operations produce no direct output.

Solution question 3 part b

🧠 Solution — Sequence with `insertLast`, `get`, `Duplicate`

🧱 Data Structure (Implicit Balanced Tree)

Maintain the sequence in an implicit balanced search tree (AVL or 2–3 tree) where inorder equals the sequence order.
Each node stores:

- `val` — element from Σ.
- `left, right` — child pointers.

- `size` — number of elements in the subtree (order statistics).
  (For AVL also keep `height`; for 2–3, node degree invariants.)

All updates use persistent path-copy (copy-on-write) so that large blocks can be reused structurally without element-wise copying. This guarantees that `Duplicate(i, j)` runs in `O(log n)` time and uses only `O(log n)` new nodes.

🧩 Core Primitives (Both `O(log n)` worst-case)

- `split(T, k)` → returns `(L, R)` where `L` contains the first `k` elements (positions `1..k`) and `R` the rest (`k+1..n`).
  Implementation: descend by comparing `k` to `size(left)`; rebuild/rotate (AVL) or split nodes (2–3) on the way back, updating `size` (and `height` for AVL).

- `join(A, B)` → returns the concatenation whose inorder is exactly `inorder(A) · inorder(B)`.
  Implementation:

  - AVL: if heights differ by ≥2, descend along the taller spine (right of `A` or left of `B`), attach, then rebalance on the way up; if heights are close, create a pivot root and rebalance.

  - 2–3: standard `concat`: bubble a separator upward, perform local splits/merges to maintain degrees 2–3.

- `kth(T, k)` → order-statistics search using `size(left)` to return the element at position `k`.

All three primitives run in `O(log n)` and preserve balance invariants.

➕ Operation `insertLast(x)` — Append

Rule:
`T ← join(T, node(x))`

- **Correctness:** `join` **preserves order and places** `x` **after all current elements.**
- **Time / Space:** `O(log n)` **time;** `O(log n)` **new nodes by path-copy.**

🔎 Operation `get(k)` — Random Access

Rule (order statistics):

```
get(k):
  let L = size(T.left)
  if k == L+1: return T.val
  if k <= L:   descend into T.left with k
  else:        descend into T.right with k-L-1
```

- **Correctness: unique path determined by subtree sizes.**
- **Time:** `O(log n)`**.**

📀 Operation `Duplicate(i, j)` — Interval Duplication

Goal: transform A into
`A[1..i-1] · A[i..j] · A[i..j] · A[j+1..n]`**.**

Construction with `split/join` **(no element copying):**

1. `(A1, C) ← split(T, j)` **//** `A1 = A[1..j]`**,** `C = A[j+1..n]`

2. `(L, M) ← split(A1, i-1)` // `L = A[1..i-1]`, `M = A[i..j]`

3. `T ← join( join(L, M), join(M, C) )` // same tree `M` used twice

- **Correctness: by the `join` invariant, inorder becomes**
  `inorder(L) · inorder(M) · inorder(M) · inorder(C)`,
  i.e., exactly two consecutive copies of the pre-state block `A[i..j]`.

- **Persistence / Structural Sharing: `M` is a reused subtree (same pointer) in both places; only `O(log n)` nodes are newly created along the touched paths.**

- **Time / Space: `O(log n)` time (two `split`, three `join`); `O(log n)` extra nodes.**

✅ Correctness Sketch

- **Split soundness: for any `k`, `split(T,k)` partitions inorder into prefix `1..k` and suffix `k+1..n` without reordering or loss.**

- **Join soundness: `join(A,B)` preserves both internal orders and places all of `A` before all of `B`.**

- **`insertLast`: direct concatenation with a singleton preserves append semantics.**

- **`get`: subtree-size descent pinpoints position `k`.**

- **`Duplicate`: by composing the two invariants above, the post-state equals**
  `A[1..i-1] · A[i..j] · A[i..j] · A[j+1..n]`.

- **Persistence: since nodes are not mutated in-place, reusing `M` twice is safe; later updates affect only their own top-to-leaf paths.**

⏱️ Complexity

- `insertLast`, `get`, `Duplicate`: `O(log n)` time each.
- Space: `O(n)` for the stored elements plus `O(log n)` new nodes per update (persistent path-copy).
- Bounds hold in the worst case for AVL/2–3 trees.

🔒 Edge Handling

- Indices are 1-based.
- Preconditions:
  - `get(k)` requires 1 `≤ k ≤ n`.
  - `Duplicate(i, j)` requires 1 `≤ i ≤ j ≤ n`.
- Corner cases:
  - `i = 1` (duplicate from start),
  - `j = n` (duplicate to end),
  - `i = j` (duplicate a single element).
    All handled uniformly by the same `split/join` pipeline.

📝 Notes

- Either AVL (with explicit rotations) or 2–3 tree (degree-balanced) can be used; both give the same asymptotic guarantees.

Question 3 part c (binary search tree)

📗 Section C — Problem Statement (`halve(i, j)`)

🎯 Goal

Extend the sequence data structure (from Sections A–B) with an operation that removes one copy when a block appears twice consecutively.

🧩 Universe & Notation

- **Element domain: arbitrary Σ.**
- **Sequence at any time: `A[1..n]`, indices are 1-based.**
- **For `1 ≤ i ≤ j ≤ n`, let `A[i..j]` be the contiguous block from `i` to `j` (inclusive).**
- **Let `L = j - i + 1` denote the block length.**

🔧 Operation to Support

- `halve(i, j)`
    - **Input: indices `i, j` with `1 ≤ i ≤ j ≤ n`.**
    - **Effect:**

If the block `A[i..j]` is immediately followed by an identical block of the same length, i.e.

```
j + L ≤ n  and  A[i..j] = A[j+1 .. j+L],
```

- **then replace the two consecutive copies by a single copy of `A[i..j]`.**

Formally, in this case the post-state is

`A ← A[1..j] · A[j+L+1 .. n]`

- and `n ← n - L`.
- Otherwise (no immediate duplicate), no change is made to `A`.

○ No reordering or modification of any other elements occurs.

✅ Correctness Requirements

- When the precondition `A[i..j] = A[j+1..j+L]` holds, the subsequence at positions `i..j` in the pre-state appears exactly once at positions `i..j` in the post-state; elements originally at positions `j+L+1..n` shift left by `L` and preserve their relative order and values.
- When the precondition does not hold, the sequence remains bit-wise identical to the pre-state.
- Equality of blocks is element-wise equality over `Σ`.

⏱️ Performance Targets

- Each call to `halve(i, j)` must run in `O(log n)` time (worst-case or amortized; specify your chosen model elsewhere).
- Additional space per operation: `O(log n)` (for auxiliary structure maintenance).
- Total space over `n` elements: `O(n)`.

🔒 Preconditions & Edge Policy

- Require `1 ≤ i ≤ j ≤ n`.

- The duplicate-check uses `L = j - i + 1` and is meaningful only if `j + L ≤ n`; otherwise duplication cannot hold.
- Corner cases to handle explicitly:
  - `i = j` (single-element duplication),
  - `j = n` (duplication impossible; no change),
  - `i = 1` (duplication starting at the first element).
- If an index precondition is violated, behavior is undefined or should raise a well-specified error (choose one consistent policy).

📥 Inputs & 📤 Outputs (abstract API)

- Input call: `halve(i, j)`
- Output: No direct output (the sequence may be updated in place).

Solution question 3 part c

🧠 Deterministic Solution — `insertLast`, `get`, `Duplicate`, `halve(i, j)`

📦 Data Structure (Deterministic, Persistent)

Maintain the sequence as an implicit balanced tree (AVL or 2–3 tree) whose inorder equals the sequence order.
Each node stores:

- `val` — element in Σ (only at leaves or single-element nodes, per your variant),
- `left, right` — child pointers,
- `size` — number of elements in the subtree (order statistics),

- (AVL: also `height`; 2–3: node degree invariants).

Persistence (copy-on-write): updates never mutate existing nodes; along each update, only the `O(log n)` nodes on the touched path are reallocated, and all untouched subtrees are *structurally shared*.
We rely on node identity (pointer/reference equality of subtree roots) as an exact, deterministic notion of "the same block".

🔧 Core Primitives (all worst-case `O(log n)`)

- `split(T, k) → (L, R)`
  `L` contains positions `1..k`, `R` contains `k+1...` Implement by descending with `size(left)` and (AVL) rotations / (2–3) local splits; recompute `size` (and `height`) on the way up.

- `join(A, B) → T`
  Concatenation preserving order: inorder is exactly `inorder(A) · inorder(B)`. Implement by height/degree–aware glueing and rebalancing.

- `kth(T, k)`
  Order-statistics search via `size(left)` to return the element at index `k`.

These are deterministic and preserve balance invariants.

➕ `insertLast(x)` — Append (`O(log n)`)

`T ← join(T, node(x))`

Correctness: `x` appears after all existing elements.

🔎 `get(k)` — Random Access (`O(log n)`)

Standard order-statistics descent using subtree sizes.

🧯 `Duplicate(i, j)` — Interval Duplication (`O(log n)`)

Let `1 ≤ i ≤ j ≤ n`, `L = j - i + 1`. Build using `split/join` with structural sharing:

```
(A,   C)  = split(T, j)          // A = A[1..j],   C = A[j+1..n]

(LFT, M)  = split(A, i-1)        // LFT = A[1..i-1], M = A[i..j]

T         = join( join(LFT, M), join(M, C) )
```

Key property (deterministic): because the structure is persistent, the *second* copy of `A[i..j]` is the exact same subtree `M` (same root identity). No elementwise copying occurs.

✂️ `halve(i, j)` — Deterministic Version (`O(log n)`)

Intent: If the block `A[i..j]` is *immediately* followed by an *identical* block of the same length, keep one copy; otherwise, do nothing.
Deterministic criterion: use subtree identity (pointer/reference equality) — *no hashing*.

Let `L = j - i + 1`. If `j + L > n`, duplication cannot hold ⇒ no-op.
Otherwise:

1. Isolate the two candidate blocks (three splits):

```
(A,    R)  = split(T, j + L)      // A = [1..j+L], R =
[j+L+1..n]

(P,   Q2)  = split(A, j)          // P = [1..j],  Q2 =
[j+1..j+L]  (2nd block)
```

```
(LFT, M1)  = split(P, i-1)       // LFT = [1..i-1], M1 =
[i..j]   (1st block)
```

Now the sequence is factored as: `LFT | M1 | Q2 | R`.

2. Deterministic equality test (no probability):

```
IF  root(M1) === root(Q2)        // pointer/reference equality

    // They are the exact same (shared) subtree created by
Duplicate

THEN

    T ← join( join(LFT, M1), R ) // remove the second copy

ELSE

    T ← join( join( join(LFT, M1), Q2 ), R ) // restore
original
```

Because nodes are immutable (persistent), `root(M1) === root(Q2)` iff the two blocks are *structurally the same object*, which in this design arises exactly when they were produced by a prior `Duplicate` and have not been modified.

Correctness (deterministic)

- Splits preserve order and partition the sequence into four consecutive segments without loss.
- Pointer-equality is an exact, deterministic predicate for "same block" under persistence.
- Joins preserve order; thus:
    - If equal: `M1 | Q2` replaced by `M1` ⇒ post-state is `A[1..j] · A[j+L+1..n]`.

- If not equal: rejoining `LFT | M1 | Q2 | R` restores the original sequence.

Complexity

- 3× `split` + 2× (or 3×) `join` ⇒ `O(log n)` worst-case time;
- `O(log n)` new nodes (path-copy) in persistence;
- Total space remains `O(n)`.

Scope of the deterministic predicate

This `halve(i, j)` collapses *exact duplicates that were created by the data structure itself* (via `Duplicate`) because only then the two blocks share the same persistent subtree. Identical blocks formed by coincidental edits elsewhere will not be collapsed (by design), preserving worst-case `O(log n)` deterministically and avoiding any probabilistic hashing.