Arrays, Binary Search, Dynamic Arrays and Elementary Sorting

Lecture Script, Worked Traces, Worksheet and Solutions
Z5007 Programming and Data Structures, MTech DSAI | Week 2, Oct 12–16, 2026

1. Contiguous Arrays and Binary Search

> In a contiguous array storing elements of width w bytes starting at base address B, element $a[i]$ sits at $B + i \cdot w$. Computing that byte offset takes one multiplication and one addition, so random reads cost $O(1)$. Inserting at index 0, however, requires shifting all n elements one slot right in $O(n)$ time.

When an array is kept sorted, binary search compares against the midpoint $(low + high)//2$ and discards half of the candidate range per iteration, visiting at most $\lfloor \log_2 n \rfloor + 1$ elements.

```
def binary_search(arr, target):
    low, high = 0, len(arr) - 1
    while low <= high:
        mid = low + (high - low) // 2
        if arr[mid] == target:
            return mid
        if arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1
    return -1
```

Check question. In `binary_search`, what fails if the loop condition is changed from `low <= high` to `low < high`?

Answer. When `low == high` holds on a single remaining element, the loop exits without checking that final element and falsely returns `-1`.

2. Dynamic Arrays and Amortized Doubling

Python `list` keeps an internal capacity $C \geq n$. Whenever `append(x)` arrives at a full array ($n = C$), it allocates a block of capacity $2C$, copies the n items in $O(n)$ steps, and writes x.

Over n appends starting from capacity 1, resize copies happen at powers of two ($1 + 2 + 4 + \cdots + n/2 < n$), plus n direct writes, giving total cost $< 3n$, or $O(1)$ amortized cost per append.

Table 1: Elementary sorting comparison across worst-case, best-case, and stability.

Algorithm	Best	Average / Worst	Max Swaps	Stable
Bubble Sort (early exit)	$O(n)$	$O(n^2)$	$n(n-1)/2$	Yes
Selection Sort	$O(n^2)$	$O(n^2)$	$n-1$	No

Check question. Selection sort always takes $\Theta(n^2)$ comparisons even on sorted input. When can it still beat bubble sort?

Answer. When swaps are very expensive (such as moving large structs or writing flash memory), because selection sort performs at most $n - 1$ swaps total.

3. Worksheet (Students Complete Individually)

1. Trace `binary_search([4, 9, 13, 19, 24, 31, 40], 24)` by listing $(low, mid, high)$ at every step.
2. A dynamic array starts at capacity 1 and triples capacity ($3C$) on overflow. Prove whether `append` remains $O(1)$ amortized.
3. Trace Selection Sort pass by pass on $[29, 10, 14, 37, 13]$.

4. Worksheet Solutions

1. Step 1: $(0, 3, 6)$, $a[3] = 19 < 24 \implies low = 4$; Step 2: $(4, 5, 6)$, $a[5] = 31 > 24 \implies high = 4$; Step 3: $(4, 4, 4)$, $a[4] = 24$, returns index 4.
2. Total resize copies over n pushes are $\sum_{k=0}^{\lfloor \log_3 n \rfloor} 3^k < \frac{3}{2}n$, so amortized cost is still $O(1)$.
3. Pass 1: swap 29, 10 $\rightarrow [10, 29, 14, 37, 13]$; Pass 2: swap 29, 13 $\rightarrow [10, 13, 14, 37, 29]$; Pass 3: 14 stays; Pass 4: swap 37, 29 $\rightarrow [10, 13, 14, 29, 37]$.

References

1. Canning, J., Broder, A. and Lafore, R. (2022). *Data Structures and Algorithms in Python*. Addison-Wesley. Chapters 2–3.
2. Cormen, T. H. et al. (2022). *Introduction to Algorithms*, 4th ed. MIT Press. Chapter 17 (Amortized Analysis).