CS50 Week 3 – Searching, Sorting, and Recursion

Animesh Garg and ChatGPT

Introduction

In this set of notes for **Week 3 of CS50**, we dive deeper into:

- **Searching algorithms** in arrays: linear and binary search, plus their running times.
- **Sorting algorithms**, including *selection sort*, *bubble sort*, and *merge sort*.
- **Asymptotic notation** ($O(\cdot)$, $\Omega(\cdot)$, $\Theta(\cdot)$) to categorize running times.
- **Recursion** as a fundamental technique for solving smaller subproblems repeatedly.
- **Structs in C**, allowing us to create custom data types (e.g. a `person` with name and number).

We also consider **efficiency** and design trade-offs:

- Searching in sorted vs. unsorted data
- Using extra memory (space) to speed up runtime
- Recursion vs. iteration

Contents

1 Linear Search

1.1 Concept and Pseudocode

- Problem: Given an unsorted array (e.g. of integers), determine if a specific value (e.g. 50) is present.
- **Linear search** (also called *sequential search*) inspects each element in turn, from left to right.

Pseudocode:

```
for each door from left to right:
    if door's value == 50:
        return true
return false
```

This directly translates to code using a for-loop and an if-statement.

1.2 Example in C: `search.c`

```c
// Implements linear search for integers

#include <cs50.h>
#include <stdio.h>

int main(void)
{
    // An array of integers
    int numbers[] = {20, 500, 10, 5, 100, 1, 50};

    // Search for a number
    int n = get_int("Number: ");
    for (int i = 0; i < 7; i++)
    {
        if (numbers[i] == n)
        {
            printf("Found\n");
            return 0; // success
        }
    }
    printf("Not found\n");
    return 1; // failure
}
```

Notes:

- We `return 0` on success, `return 1` on failure.
- The linear search has a worst-case scenario: we check *all* elements.
- $\Rightarrow O(n)$ in the worst case, $\Omega(1)$ in the best case (if we get lucky at the first element).

1.3 Strings and `strcmp`

In C, you cannot simply do `if (strings[i] == s)` to compare strings. Instead, you call:

```
strcmp(s1, s2)
```

which returns:

- 0 if `s1 == s2` (same string),
- < 0 if `s1` comes *before* `s2` in ASCII order,
- > 0 if `s1` comes *after* `s2`.

Hence, we do `if (strcmp(strings[i], s) == 0) {...}`.

2 Binary Search

2.1 Concept and Pseudocode

If an array is **sorted**, we can do better than linear search. **Binary search**:

- Repeatedly divide the array in half.
- Compare the middle element to the search target (like 50).
- Either we found it, or we know it must lie to the left or right half.

Pseudocode:

```
if no doors left:
    return false

if middle door has 50:
    return true
else if 50 < middle door:
    search left half
else:
    search right half
```

This improves worst-case from $O(n)$ to $O(\log n)$. However, we *require* the data to be sorted to apply binary search effectively.

3 Running Time and Asymptotic Notation

3.1 Big O ($O(\cdot)$), Big Omega ($\Omega(\cdot)$), and Big Theta ($\Theta(\cdot)$)

- $\mathbf{O}(\cdot)$ describes the **upper bound** on the running time for an algorithm as input grows large.
- $\mathbf{\Omega}(\cdot)$ describes the **lower bound** (best-case) on the running time.
- $\mathbf{\Theta}(\cdot)$ means the two bounds coincide: the running time grows at that rate in both best- and worst-cases.

Typical complexities:

$$O(1), \quad O(\log n), \quad O(n), \quad O(n \log n), \quad O(n^2), \quad O(n^3), \quad O(2^n), \ldots$$

- *Linear search*: $O(n)$ worst, $\Omega(1)$ best.
- *Binary search*: $O(\log n)$ worst, $\Omega(1)$ best.

4 `phonebook.c` and the Case for Structs

4.1 Without Structs

Recall an example:

```
// phonebook.c (version 1)

#include <cs50.h>
#include <stdio.h>
#include <string.h>

int main(void)
{
    // Parallel arrays
    string names[] = {"Yuliia", "David", "John"};
    string numbers[] = {"+1-617-495-1000",
                        "+1-617-495-1000",
                        "+1-949-468-2750"};

    // Search for a name
    string name = get_string("Name: ");
    for (int i = 0; i < 3; i++)
    {
        if (strcmp(names[i], name) == 0)
        {
            printf("Found %s\n", numbers[i]);
            return 0;
        }
    }
    printf("Not found\n");
    return 1;
}
```

Drawbacks:

- We must ensure parallel arrays remain in sync (index `i` must map the same person).
- Hard-coded array length.
- If a name changes position, must also move the corresponding number.

4.2 With Structs

C allows definition of a custom type with `struct`:

```
typedef struct
{
    string name;
    string number;
} person;
```

Hence:

```
// phonebook.c (version 2 with structs)

#include <cs50.h>
#include <stdio.h>
#include <string.h>

// Define a new person type with name and number
typedef struct
{
    string name;
    string number;
} person;

int main(void)
{
    // Create an array of people (size 3)
    person people[3];

    // Initialize
    people[0].name   = "Yuliia";
    people[0].number = "+1-617-495-1000";

    people[1].name   = "David";
    people[1].number = "+1-617-495-1000";

    people[2].name   = "John";
    people[2].number = "+1-949-468-2750";

    // Search for a name
    string name = get_string("Name: ");
    for (int i = 0; i < 3; i++)
    {
        if (strcmp(people[i].name, name) == 0)
        {
            printf("Found %s\n", people[i].number);
            return 0;
        }
    }
    printf("Not found\n");
```

```
    return 1;
}
```

Notes:

- We define `person`, a composite type with `name` and `number`.
- Use `dot` notation, e.g. `people[0].number`, to access fields.

5 Sorting

5.1 Motivation

- Many searching algorithms (like binary search) require data to be sorted.
- Sorting is thus an important first step if frequent searches are expected.
- But sorting itself can be costly. We'll see trade-offs in time and space.

We will consider three sorts:

- **Selection sort**
- **Bubble sort**
- **Merge sort**

5.2 Selection Sort

Idea:

1. For each index i from 0 to $n-1$:
2. Find the smallest value from the sub-array `numbers[i]` ... `numbers[n-1]`.
3. Swap that smallest element with `numbers[i]`.

Analysis:

- On pass #1: $n-1$ comparisons (search entire array).
- On pass #2: $n-2$ comparisons.
- ...
- On pass #(n-1): 1 comparison.

Sum of comparisons:

$$(n-1)+(n-2)+(n-3)+\cdots+1 \quad = \quad \frac{n(n-1)}{2}$$

- $\Rightarrow O(n^2)$ worst-case.
- No optimization if array is already sorted, so $\Omega(n^2)$ in best-case.
- So $\Theta(n^2)$ overall.

5.3 Bubble Sort

Idea:

1. Repeatedly traverse the array.
2. Compare each adjacent pair of elements.
3. If out of order, swap them. (This gradually "bubbles" larger elements rightward.)
4. Repeat $(n-1)$ times or until no swaps needed.

Pseudocode:

```
Repeat n-1 times:
    For i from 0 to n-2:
        if numbers[i] > numbers[i+1]:
            swap(numbers[i], numbers[i+1])
    If no swaps in this pass, array is sorted; quit
```

Analysis:

- Worst-case: $O(n^2)$.
- If array is already sorted, we can detect no swaps in first pass, so best-case: $\Omega(n)$.

5.4 Recursion

We can define certain processes in a *recursive* way, meaning a function calls itself:

- Base case: a minimal scenario (like an empty list).
- Recursive case: break problem into smaller subproblems, solve them, combine results.

For example, a pyramid of height n can be drawn by:

1. Draw pyramid of height $(n-1)$.
2. Then print a row of width n.

5.5 Merge Sort

Idea:

If list has 1 or 0 elements, it's sorted (base case).

Otherwise:

1. Sort left half (recursively).
2. Sort right half (recursively).
3. **Merge** the two sorted halves into one sorted whole.

Merging is done by:

- Compare the first elements of each half (left pointer vs. right pointer).
- Move the smaller into a "temp" buffer or array.

- Repeat until all elements consumed.

Analysis:

- We split the array into halves repeatedly: $\log n$ *levels* of splitting.
- At each level, merging all sub-lists together touches each element once: $\sim n$ steps per level.
- $\Rightarrow O(n \log n)$ in worst-case.
- Also $\Omega(n \log n)$ in best-case (no easy early-out).

Hence merge sort is $\Theta(n \log n)$. More efficient than our earlier n^2 sorts for large n.

6 Summary and Takeaways

- **Linear Search**:
  - On unsorted data
  - $O(n)$ worst-case
  - Simple to implement
- **Binary Search**:
  - On sorted data
  - $O(\log n)$ worst-case
  - Powerful if data is pre-sorted or can be sorted ahead
- **Selection Sort, Bubble Sort**:
  - Both in $\Theta(n^2)$
  - Straightforward to code
- **Merge Sort**:
  - $\Theta(n \log n)$ best- and worst-case
  - Requires extra space for merging
  - Illustrates power of recursion
- **Recursion** is a common technique where a function calls itself on smaller subproblems. A base case (or cases) ensures termination.
- **Structs** in C let you define custom data types (e.g. a `person` with `name` and `number`).

Big ideas for Week 3:

- Efficiency and **asymptotic notation** are vital to good algorithmic design.
- Sorting can help speed up searching but sorting itself can be expensive.
- **Recursion** can simplify logic and mirror how many problems are naturally subdivided.