date: 2025-10-22

1 algorithms

an **algorithm** is a finite sequence of precise instructions for solving a **problem**. an algorithm should have these important features:

- finite: in order to execute, it must be finite
- sequence: the steps needs to be in the correct order
- precise: each step must be unambiguous
- instructions: each step can be carried out

the **problem** should be general enough to be broadly useful, but specific enough to dictate strategies. for example: the problem "sort the numbers (6, 8, 5, 3)" is too specific, and "return the numeric solution to the input query" is too general.

1.1 searching problem

problem: given a sequence of values $a_1, a_2, a_3, ..., a_n$, and a target value x, return an index i such that $a_i = x$ (or 0 if no such i exists).

one algorithm that might solve this problem is by going through each number in the sequence, and return the first i such that $a_i = x$.

1.2 pseudocode

pseudocode is a semi-structured set of notations. it is more precise than describing in plain words, and it have less overhead than a programming language. if a program (executed by computers, very precise) is analogous to a formal proof, then pseudocode is analogous to an informal proof, where it is meant to read by humans.

we can use pseudocode to solve the searching problem above:

algorithm for the searching problem

```
1 procedure linear search(x: integer, a_1, a_2, ..., a_n: distinct integers)
2 |  i := 1
3 |  while ((i ≤ n) ∧ (x ≠ a_i))
4 |  |  i := i + 1
5 |  |  if i ≤ n then
6 |  |  |  location := i
7 |  |  else
8 |  |  |  location := 0
9 |  return location
```

1.3 bindary searching

enumerating through each number in the sequence isn't very efficient, however. instead, we can use the binary search algorithm, since a sequence is always in ascending order. a binary search algorithm searches from both sides, and compares the midpoint against the target value; see pseudocode below:

binary search through sequence

```
1  procedure binary_search(x: integer, a_1, a_2, ..., a_n: integers in non-decreasing order)
2  | start := 1
3  | end := n
4  | while start < end:
5  | | mid := floor((start + end) / 2)
6  | | if x > a_mid then:
7  | | | start := mid + 1
8  | | else if x < a_mid then:
9  | | | end := mid - 1
10 | | else:
11 | | | start := mid
12 | | | end := mid
13 | if x = a_start then:
14 | | location := start
15 | else:
16 | | location := 0
17 | return location
```