Data structures & algorithms

Martín Escardó

- Kinds: Lists, trees, tables, graphs, ...
- Algorithms: Sort, insert, delete, find, ...
- Efficiency: How fast? How much memory?
- Abstraction: $\underbrace{\text{How to use?}}_{\text{abstract}}$ $\underbrace{\text{How to implement?}}_{\text{concrete}}$ (c.f. Java interfaces)

  deliberately forget details — need to consider details explicitly
- Specification: what we intend the algorithm to do.

Warming-up example

- Data structure : array
- Problem specification (imprecise): find an element in the array.
- Algorithms :
  - linear search (slower)
  - binary search (faster)

what is this ?

And this ?

As we shall see, more precisely :

- linear search is $O(n)$
- binary search is $O(\log n)$

what is this ?

Linear search specification (precise this time)

Given an array a of integers, and given an integer x,
find an integer i such that

1. if there is no j such that $a[j]$ is x,
   then i is -1,
2. otherwise, $a[i]$ is equal to x.

Question: If there is more than one i with $a[i]$ equal to x,
which one should we return?

Answer: According to the above (ambiguous!) specification any such i
is fine. (For example, we can return the first from left to right.)

Examples

Consider $a =$

	0	1	2	3	4	5	6	← positions
a	17	13	100	3	2	100	20	← contents

1. If $X = 1001$, the specification says we should return -1, indicating that 1001 is not in the array.
2. If $X = 2$, the specification says we should return 4.
3. If $X = 100$, the specification says we may return 2 or 5 (and nothing else). Our algorithm is free to choose which one.

Algorithm 1 : linear search

```
int linearSearch (int [] a, int x) {
  for (int i = 0; i < a.length; i++)
    if (a[i] == x)
      return i; // we disambiguate the specification
                // by choosing the first i
                // from left to right.
  // not found. Indicate this with -1, as per specification.
  return -1;
}
```

Algorithm writing routine

1. Think what you want it to do and write this down.
   This is the specification.
2. Think, come up with an idea, and write down your algorithm.
3. Check that your algorithm does indeed satisfy the specification
   - By testing (imperfect)
   - By writing a convincing argument (aka proof)
4. If necessary, reason about its run-time complexity and/or its space complexity. (If they are bad, you may need a better algorithm)

The case of linear search

- We have done 1 (specification) and 2 (algorithm writing).
- What about 3 (checking correctness)? This algorithm is so simple that it is immediately clear that it works.
  (But we will meet many algorithms whose correctness is not obvious, in fact soon.)
- So it remains to think about 4 (run-time and space).

Linear search run-time

Call n the length of the array a.

- If x is not found by the algorithm, it will loop n times. This is the worst case and is indicated by $O(n)$.

- On average the algorithm takes

unlucky case

$$\frac{1 + 2 + 3 + \cdots + n}{n} = \frac{n+1}{2}$$

lucky case

loop iterations. (As we shall see, this is still $O(n)$.)

Linear search – practical considerations

- When n is small, and when we only make a few searches, linear search is good enough.

- But if n is large and/or we use the algorithm repeatedly (maybe in a loop), this will be problematic.
  So we look for faster algorithms.

Binary search

- Same specification.
- But we assume that the array a is sorted.

  (we will learn how to sort later.)

E.g.

$a =$	10	100	20	13	20	17

not sorted

$a =$	10	13	17	20	20	100

sorted

This is called a precondition for the algorithm.

Binary search pictorially

$x = 100$

$a =$

2	4	7	13	17	20	20	100	200	220	300	400

1. Here ✗ or here ✓

2. Here ✓ or here ✗

3. ✗ ✓

4. ✓ ✗

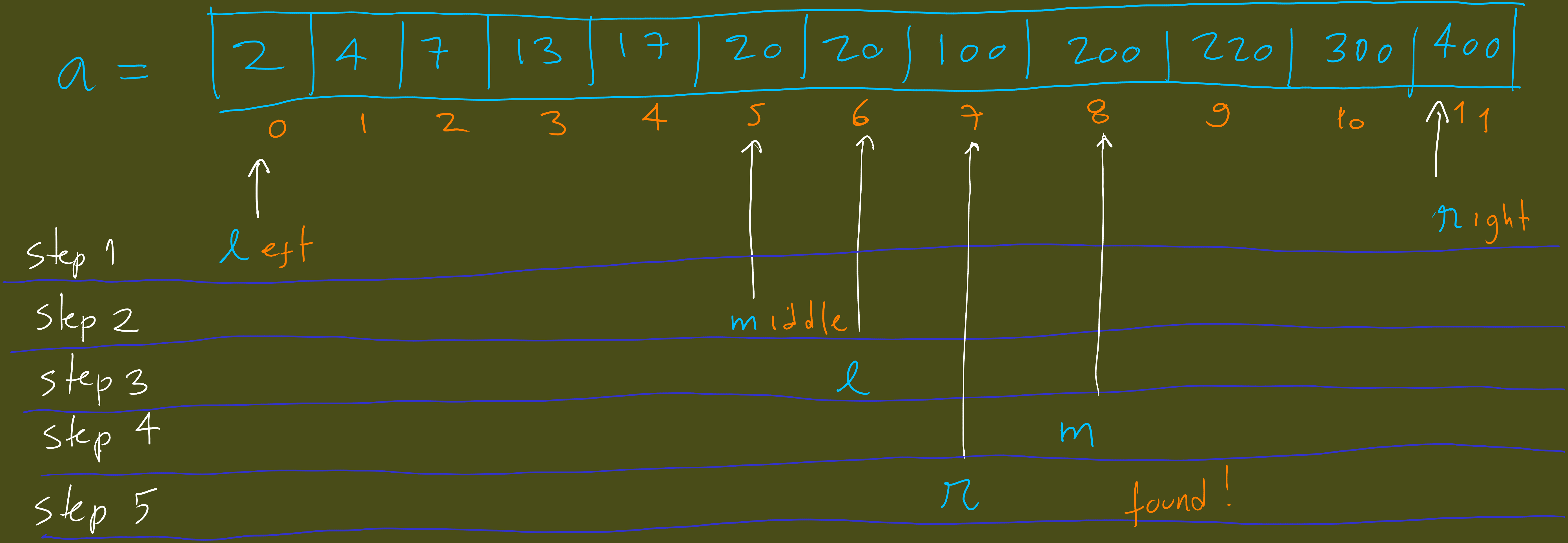
found

Idea: split the search space in two (approximately) equal parts at each stage.

Like when looking for a word in a dictionary.

Binary search pictorially

$x = 100$



```
int binarySearch(int[] a, int x) {
    assert(isSorted(a)); // Precondition
    int l = 0, r = a.length-1;
    while (l ≤ r) {
        int m = (l+r)/2;
        if (a[m] < x)
            l = m+1;
        else if (a[m] > x)
            r = m+1;
        else return m; // success
    }
    return -1; // failure
}
```

Wrong! should be

r = m-1

Binary search

We want to examine:

- Run time

  Does it really get faster?

  Significantly so?

  By how much?

- Correctness

  Understand why the algorithm really works

  (i.e. satisfies the intended specification)

Binary search run time (Recall that $n = a.length$)

Suppose $n = 4096$ The binary search takes at most 12 steps:

splitting a into two equal parts, each part gets size

1 $4096/2 = 2048$

splitting each part into two equal parts gives

2 $2048/2 = 1024$

And then

3 $1024/2 = 512$

4 $512/2 = 256$

5 $256/2 = 128$

6 $128/2 = 64$

7 $64/2 = 32$

8 $32/2 = 16$

9 $16/2 = 8$

10 $8/2 = 4$

11 $4/2 = 2$

12 $2/2 = 1$

At this point the algorithm ends

The relation between 4096 and 12

$$12 = \log_2 4096$$

because

$$4096 = 2^{12} = \underbrace{2 * 2 * \cdots * 2 * 2}_{12 \text{ times}}$$

The run time of binary search is $\log_2 n$ loop iterations in the worst case.

Powers of 2

$1 = 2^0$

$2 = 2^1$

$4 = 2^2 = 2*2$

$8 = 2^3 = 2*2*2$

$16 = 2^4 = 2*2*2*2$

$32 = 2^5 = 2*2*2*2*2$

$64 = 2^6 = 2*2*2*2*2*2$

$128 = 2^7 = 2*2*2*2*2*2*2$

$256 = 2^8 = 2*2*2*2*2*2*2*2$

$512 = 2^9 = 2*2*2*2*2*2*2*2*2$

$1024 = 2^{10}$

$2048 = 2^{11}$

$\log_2(1048576) = 20$

$4096 = 2^{12}$

$8192 = 2^{13}$

$16384 = 2^{14}$

$32768 = 2^{15}$

$65536 = 2^{16}$

$131072 = 2^{17}$

$262144 = 2^{18}$

$524288 = 2^{19}$

$1048576 = 2^{20}$

big

small

Logarithm in base 2

Definition $\log_2(n)$ is the number k such that $2^k = n$.

How do we calculate it?

- Use the table of the previous page. E.g. $\log_2(250000)$
  - The table says $2^{17} = 131072$ $\quad 250000$
    $2^{18} = 262144$
  - So $17 < \log_2(250000) < 18$.
  - We take the integer part for counting number of loop iterations.

Binary search algorithm correctness

- We use
  - preconditions
  - invariants
  - post conditions

  to reason about the algorithm correctness.

- We also use
  - assertions

  to help <u>testing</u> the algorithm.

Definition of big-O notation

We write $f(n) = O(g(n))$

Starting point

to mean there are numbers $M > 0$ and $n_0 > n$ such that

$$f(n) \leq M \cdot g(n)$$

proportionality constant

not always, but when $n \geq n_0$

concept 1 Constant Time

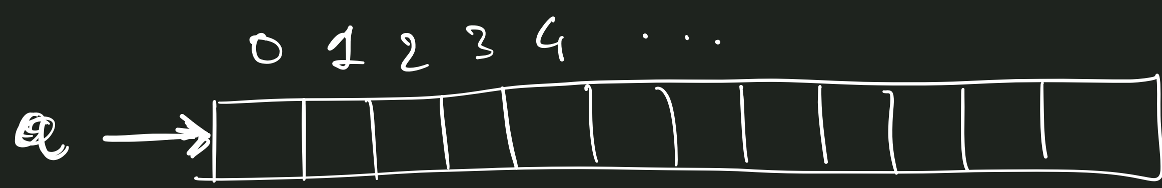
$$f(n) \le M \cdot g(n) \text{ when } n \ge n_0$$

Example:

array

```
function swap(a, i, j)
    aux = a[i]        1 step
    a[i] = a[j]       1 step
    a[j] = aux        1 step
    return
```

array



$$f(n) = 3 = O(1)$$

$$3 \le M \cdot 1$$

concept 2 Linear Time

$$f(n) \leq M \cdot g(n) \text{ when } n \geq n_0$$

Example:

linked list

the complexity of this operation depends on the data structure

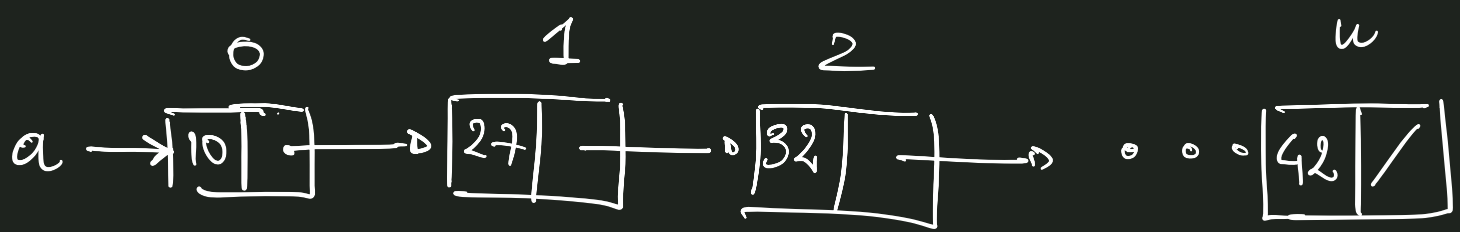
```
function swap(a, i, j)
    aux = a[i]
    a[i] = a[j]
    a[j] = aux
    return
```

i steps $\leq n$ steps

$i+j$ steps $\leq 2n$ steps

j steps $\leq n$ steps

linked list



$$f(n) \leq 4n = O(n)$$

concept 3

Log Time

$$f(n) \leq M \cdot g(n) \text{ when } n \geq n_0$$

when input size n grows 2^K with the number of step K, then

$$K = \log_2 n = O(\log n)$$

- what if $n = 3^K$?

$$K = \log_3 n = O(\log n)$$

Fact : $\log_a(n) = O(\log_b n)$

Because :

$$\log_a(n) = \frac{1}{\log_b(a)} \log_b(n)$$

M

this is why the base does not matter, and we just say $O(\log n)$