

## Inf 2B: Introduction to Algorithms

### Lecture 1 of ADS thread

Kyriakos Kalorkoti

School of Informatics  
University of Edinburgh

Taught by [Kyriakos Kalorkoti \(KK\)](#), IF5.26, [kk@inf.ed.ac.uk](mailto:kk@inf.ed.ac.uk).

#### Topics:

- 1: Algorithms, analysing algorithms, Asymptotic notation (for talking about running-times), Sequential Data Structures, Tree data structures, Hashing, Priority Queues, Advanced sorting.
- 2: Algorithms for the WWW: indexing, searching.
- 3: Algorithms for searching graphs, applications to graph problems.

1 / 23

2 / 23

## Textbooks

For Algorithms and Data Structures (recommended, not required.):

- ▶ [GT] *Data Structures and Algorithms in Java*, by Goodrich & Tamassia (4th or 3rd ed), Wiley.  
Gentle textbook, *best* for this course (doesn't have WWW stuff).  
Java.
- ▶ [CLRS] *Introduction to Algorithms*, by Cormen, Leiserson, Rivest & Stein, MIT Press.  
Lots of Algorithms & Data Structures.  
Technical.  
No Java (or any other programming language).  
Course text for 3rd year Algorithms and Data Structures course.

If you will not take 3rd year ADS, choose [GT].

3 / 23

## Our Ingredients

**Algorithms** Step-by-step procedure (a “recipe”) for performing a task.

**Data Structures** Systematic way of organising data and making it accessible in certain ways.

We are interested in the design and analysis of “good” algorithms and data structures.

4 / 23

## Data Structures

Arrays, linked lists, stacks, trees.

## Algorithm design principles

Recursive algorithms.

## Searching and Sorting Algorithms

Linear search and Binary search. Insertion sort, selection sort.

Other prerequisites:

Maths-for-Inf1 and Maths-for-Inf-2 (especially the material on counting and graphs).

▶ *Correctness*

▶ *Efficiency* w.r.t.

{ running time,  
space (=amount of memory used),  
network traffic,  
number of times secondary storage is accessed.

▶ *Simplicity*

5/23

6/23

# Measuring Running time

The running time of a *program* depends on a number of factors such as:

1. The *input*.
2. The *running time of the algorithm*.
3. The *quality of the implementation* and the *quality of the code generated by the compiler*.
4. The *machine used to execute the program*.

We will rarely be concerned with the *implementation quality*, the *code quality* or the *machine*.

# Example 1: Linear Search in JAVA

```
public static int linSearch(int[] A,int k) {  
    for(int i = 0; i < A.length; i++)  
        if ( A[i] == k )  
            return i;  
    return -1;  
}
```

This is Java.

- ▶ We want to ignore implementation details, so we map this to *pseudocode*.

7/23

8/23

## Linear Search in Pseudocode

**Input:** Integer array  $A$ , integer  $k$  being searched.

**Output:** The least index  $i$  such that  $A[i] = k$ ; otherwise  $-1$ .

**Algorithm** linSearch( $A, k$ )

1. **for**  $i \leftarrow 0$  **to**  $A.length - 1$  **do**
2.     **if**  $A[i] = k$  **then**
3.         **return**  $i$
4. **return**  $-1$

Suppose  $A = \langle 19, 5, 6, 77, 2, 1, 90, 3, 4, 22, 1, 5, 6 \rangle$  and  $k = 1$ .  
What happens?

9/23

## Average Running Time

In general worst-case seems overly pessimistic.

### Definition

The *average running time* of an algorithm  $A$  is the function  $AVT_A : \mathbb{N} \rightarrow \mathbb{R}$  where  $AVT_A(n)$  is the *average* number of computation steps performed by  $A$  on an input of size  $n$ .

Problems with average time

- ▶ What precisely does *average* mean? What is meant by an “average” input depends on the application.
- ▶ Average time analysis is mathematically very difficult and often infeasible (in general, OK for linSearch).

11/23

## Worst Case Running Time

Assign a *size* to each possible input.

### Definition

The (*worst-case*) *running time* of an algorithm  $A$  is the function  $T_A : \mathbb{N} \rightarrow \mathbb{N}$  where  $T_A(n)$  is the *maximum* number of computation steps performed by  $A$  on an input of size  $n$ .

**Example:** linSearch.

- ▶ Suppose the *size* is the length of the array  $A$ .
- ▶ Worst-case running time is a linear function of size.

**Note:**

- ▶ Implicit assumption that array entries are of bounded size.
- ▶ Otherwise we could take sum of all array entry sizes as measure of input size.

10/23

## Analysis of Algorithms

A nice approach would be to combine:



We will aim for this *but*

- ▶ Java's **Garbage Collection** hampers the quality of our experiments.

12/23

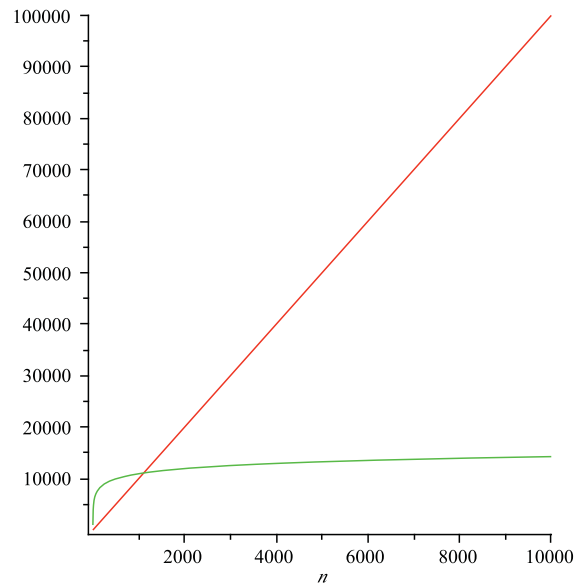
## Example 2: Binary Search

**Input:** Integer array  $A$  in increasing order, integers  $i_1, i_2, k$ .

**Output:** An index  $i, i_1 \leq i \leq i_2$  with  $A[i] = k$ , if such an  $i$  exists,  $-1$  otherwise.

**Algorithm** `binarySearch(A, k, i1, i2)`

1. **if**  $i_2 < i_1$  **then return**  $-1$
2. **else**
3.      $j \leftarrow \lfloor \frac{i_1 + i_2}{2} \rfloor$
4.     **if**  $k = A[j]$  **then**
5.         **return**  $j$
6.     **else if**  $k < A[j]$  **then**
7.         **return** `binarySearch(A, k, i1, j - 1)`
8.     **else**
9.         **return** `binarySearch(A, k, j + 1, i2)`



$$T_{\text{linSearch}}(n) = 10n + 10,$$

$$T_{\text{binarySearch}}(n) = 1000 \lg(n) + 1000.$$

## Running-time of Binary search

Input array of size  $n$ .

- ▶ Do at most a constant  $c$  amount of work.
- ▶ If  $k$  found done else recurse on array of size about  $n/2$ .
- ▶ Do a constant  $c$  amount of work.
- ▶ If  $k$  found done else recurse on array of size about  $n/2^2$ .
- ⋮
- ▶ Do a constant  $c$  amount of work.
- ▶ If  $k$  found done else recurse on array of size about  $n/2^r$ .

Base case:  $n/2^r = 1$ , i.e.,  $r = \lg(n)$ . Then one more call.

Total work done (time) no more than

$$c(\lg(n) + 2).$$

Better than `linSearch`?

13/23

14/23

## $\lg n$ versus $n$

Put

$$m = \lg n.$$

By definition

$$n = 2^m.$$

Now:

$$\begin{array}{ll} m \rightarrow m + 1 & n \rightarrow 2n \\ m \rightarrow m + 5 & n \rightarrow 32n \\ m \rightarrow m + 10 & n \rightarrow 1024n \\ \\ m \rightarrow m + c & n \rightarrow 2^c n \end{array}$$

15/23

16/23

## Some Statistics

Jan 2008 on a DICE machine.

size	wc linS	avc linS	wc binS	avc binS
10	≤ 1 ms	≤ 1 ms	≤ 1 ms	≤ 1 ms
100	≤ 1 ms	≤ 1 ms	≤ 1 ms	≤ 1 ms
1000	≤ 1 ms	≤ 1 ms	≤ 1 ms	≤ 1 ms
10000	≤ 1 ms	≤ 1 ms	≤ 1 ms	≤ 1 ms
100000	≤ 1 ms	≤ 1 ms	≤ 1 ms	≤ 1 ms
200000	≤ 1 ms	≤ 1 ms	≤ 1 ms	≤ 1 ms
400000	3 ms	≤ 1 ms	≤ 1 ms	≤ 1 ms
600000	3 ms	1.3 ms	≤ 1 ms	≤ 1 ms
800000	3 ms	1.5 ms	≤ 1 ms	≤ 1 ms
1000000	5 ms	2.1 ms	≤ 1 ms	≤ 1 ms
2000000	7 ms	3.7 ms	≤ 1 ms	≤ 1 ms
4000000	12 ms	6.9 ms	≤ 1 ms	≤ 1 ms
6000000	24 ms	11.6 ms	≤ 1 ms	≤ 1 ms
8000000	24 ms	15.6 ms	≤ 1 ms	≤ 1 ms

200 repetitions for each size.

17/23

## Why not just do experiments?

- ▶ Consider sorting arrays of the integers 1, 2, ..., 100 held in some order.
- ▶ Just take a 1% sample of all possible inputs.
- ▶ How many experiments?

18/23

## Why not just do experiments?

- ▶ Consider sorting arrays of the integers 1, 2, ..., 100 held in some order.
- ▶ Just take a 1% sample of all possible inputs.
- ▶ How many experiments?

99! = 9332621544394415268169923885626670049071596826438  
 162146859296389521759999322991560894146397615651  
 82862536979208272237582511852109168640000000000  
 0000000000.

Assume algorithm can sort  $10^{50}$  instances per second(!).  
 How long do we need to wait?

19/23

## Why not just do experiments?

- ▶ Consider sorting arrays of the integers 1, 2, ..., 100 held in some order.
- ▶ Just take a 1% sample of all possible inputs.
- ▶ How many experiments?

99! = 9332621544394415268169923885626670049071596826438  
 162146859296389521759999322991560894146397615651  
 82862536979208272237582511852109168640000000000  
 0000000000.

Assume algorithm can sort  $10^{50}$  instances per second(!).  
 How long do we need to wait?

$$\frac{99!}{60 \times 60 \times 24 \times 366 \times 10^{50}} \approx 2.951269209 \times 10^{98} \text{ years.}$$

Be seeing you!

20/23