# EFFICIENCY OF COMPUTATION AND ANALYSIS OF ALGORITHMS

by

David Toman

University of Waterloo

# List of Slides

# Algorithm Analysis Overview

- why do we analyze algorithms?

  $\Rightarrow$ and why can't we just use a stop watch?

- how do we describe the results of the analysis?

  $\Rightarrow$ succinctly?

- what is the relation between an algorithm and the problem it tries to solve?

  $\Rightarrow$ better vs. best possible solution?

- how do data structures help?

# How do we measure “goodness”?

$$\textit{input} \longrightarrow \boxed{\text{Algorithm}} \longrightarrow \textit{output}$$

**Questions:**

How long does it take?
How much *additional* storage did we use?

**in terms of**

size of the input (e.g., in bits)
size of the output

general observation: longer input $\Rightarrow$ more effort...

# Analysis vs. Benchmarking

**Experimental Methods:**

$\Rightarrow$ limited to the test cases we try

no guarantees for really big inputs

$\Rightarrow$ need for controlled test environment

quite costly to set up

**Analytic Methods:**

$\Rightarrow$ takes into account all possible inputs

$\Rightarrow$ abstracts hardware/software

$\Rightarrow$ the result is less precise

usually in terms of “order-of” notation

# Average vs. Worst case

An Algorithm may run faster on *certain* inputs (of the same size) than on others . . . and almost always will

$\Rightarrow$ we shoot for some "generic" characterization:

**Average case complexity**

- $\Rightarrow$ (very) difficult
- $\Rightarrow$ depends on distribution of inputs

**Worst case complexity**

- $\Rightarrow$ more common approach
- $\Rightarrow$ (relatively) simpler analysis
- $\Rightarrow$ crucial to real-time systems (air-traffic control)

# Hardware Independence

**IDEA:** notion of **primitive operation**

$\Rightarrow$ execution bounded (above) by a constant
**AND** independent of particular implementation
$\Rightarrow$ instead of measuring time

we can *count* primitive operations

. . . and we'll be off by at most a multiplicative constant

**Examples of primitive operations:**

- an assignment statement (for ordinal types only!)
- comparing two numbers (**`integers`**)
- looking up a value in an array
- call/return of a procedure (parameters?)

# The “Order of . . . ” Approach

**IDEA1:** characterize time complexity of an algorithm

as a **function** $f : N \to R^+$

that maps an integer (size of the input)
to the (largest possible) number of primitive operations.

**IDEA2:** group **similarly growing** functions together

$\Rightarrow$ asymptotic behaviour of functions
$\Rightarrow$ functions of the same **order** (like rounding)

**Definition:** Let $f, g : N \to R^+$. Then $f \in O(g)$ if

$$\exists c \in R^+ . \exists n \in N . \forall n' \geq n . f(n') \leq cg(n')$$

$\Rightarrow$ $O(g)$ is a **SET** of functions that are “more expensive” than $g$ by at most a constant factor.

# Asymptotic Notation (cont.)

Common classes of functions:

| | | |
|---|---|---|
| constant | $O(1)$ | 1,2,6,... |
| logarithmic | $O(\log n)$ | $2\log n$, $1+\log n$, ... |
| linear | $O(n)$ | $2n$, $5n+4$, $n+\log n$, ... |
| quadratic | $O(n^2)$ | $(4n)^2$, $n^2+4n$, $n^2+logn$, ... |
| polynomial | $O(n^k)$ | for $k \geq 0$ |
| exponential | $O(c^n)$ | |

# Asymptotic Notation: warning(s)

- in general:

  algorithm $A$ is *no worse than* $B$ if $f_A \in O(f_B)$
  algorithm $A$ is *better than* $B$ if $O(f_A) \subset O(f_B)$

  $\Rightarrow$ again, up to a constant

- in practice: we need to be careful:

  $\Rightarrow 10^{10}n > 2n^2$ for $n < 5 \cdot 10^9$
  $\Rightarrow$ important when designing *recursive* algorithms
  often different algorithm for the "small" cases

# How do we analyze an Algorithm?

We look at the commands in the language:

- **skip**
  takes one "step", $O(1)$
- $loc(\tau) := exp(\tau)$
  the size of $\tau$'s representation; $O(f_{exp} + |\tau|)$
  commonly $O(1)$ for atomic/simple record types
- $cmd_1 \ ; \ cmd_2$
  composition adds complexities; $O(f_{cmd_1} + f_{cmd_2})$
- **if** $exp(bool)$ **then** $cmd_1$ **else** $cmd_2$ **fi**
  max of the cases; $O(f_{exp} + \max(f_{cmd_1}, f_{cmd_2}))$
- **while** $exp(bool)$ **do** $cmd$ **od**
  . . . this is the hard case

# Simple FOR Loops

- Single loop

  **for** $i := 1$ **to** $n$ **do** $cmd$ **od**

  $O(n \cdot f_{cmd})$ provided $cmd$ doesn't change value of $i$

- Two nested loops

  **for** $i := 1$ **to** $n$ **do for** $j := 1$ **to** $f(i)$ **do** $cmd$ **od od**

  $O(\sum_{i=1}^{n} f(i))$ provided $cmd$ doesn't change $i$ and $j$

- etc.

# Tricky Loops and Recursion

- General loops

  $\Rightarrow$ special case of recursion
  $\Rightarrow$ see definition of $F($**while** $E$ **do** $C$ **od**$)$

- Recursion

  $\Rightarrow$ recurrence relations
  $\Rightarrow$ characteristic equations

**Example:** Merge-Sort:

$$f(n) = 2f(n/2) + O(n)$$

Solution: $f \in O(n \log n)$.

# Value vs. Representation

- given two integers (values $M$) compute their sum:

```
function s(i:integer,j:integer):integer
  if (i=0) then s := j;
           else s := s(i-1,j+1);
```

$\Rightarrow$ linear in $M$.

- given two integers $M$ and $N$ *represented by arrays of bits* of size $m$:

```
function s(i:bits, j:bits):bits
  c := 0;
  for k:=length(i) downto 1 do
    s[k] := (i[k]+j[k]+c) mod 2;
    if (i[k]+j[k]+c>1) then c:=1;
                       else c:=0;
```

$\Rightarrow$ linear in $m$ which is *logarithmic* in $M$!

# Algorithms vs. “problems”

**Question:** What is a complexity of a “problem”?

$\Rightarrow$ complexity of the **best** algorithm that solves it

**Example 1:** Sorting $n$ elements (integers)

$\Rightarrow$ can be done in $O(n \log n)$ using, e.g., merge sort
$\Rightarrow$ **in addition** there is no algorithm that
can sort $n$ elements faster: $\Omega(n \log n)$

**Example 2:** Hamiltonian Circuit on $n$ nodes

$\Rightarrow$ can be done in $O(2^n)$
$\Rightarrow$ lower bound not known (at least $\Omega(n)$)

# Big-$O$ Cousins

**Definition:** Let $f, g : N \to R^+$. Then $f \in \Omega(g)$ if

$$\exists c \in R^+ . \exists n \in N . \forall n' \geq n . f(n') \geq cg(n')$$

$\Rightarrow \Omega(g)$ is a **SET** of functions that are
"at least as expensive" as $g$.

**Definition:** $\Theta(f) = O(f) \cap \Omega(f)$.

$\Rightarrow \Theta(g)$ is a **SET** of functions that are "exactly like" $g$

# Sub-linear Algorithms?

**Question:** can we have sub-linear algorithms?

$\Rightarrow$ don't we have to look at all of the input at least once?

**Answer**: We may avoid looking at the whole input
if we have *additional information* about it

$\Rightarrow$ "find the least element in a $n$-element array":
$\Theta(n)$ in general
$O(1)$ if we know that the array is *sorted*

# Data Structures

**IDEA:** organizing *inputs* so that algorithms can take advantage of their *structural* properties

$\Rightarrow$ avoids “looking at data”

**Example:** A Database:

$\Rightarrow$ absolutely infeasible to “read” the whole database
$\Rightarrow$ organized in a “data structure” that allows
queries to find answers “quickly”

# Other Complexity Measures

- Space Complexity

  $\Rightarrow$ How many “units” of store does an algorithm use?

  “unit allocations” instead of “primitive operations”

  $\Rightarrow$ What do we need to account for:

  space for local variables (in recursion!)
  **new**$(\tau)$s

  $\Rightarrow$ what about “unit deallocations”?

- Circuit Complexity

  $\Rightarrow$ how easy is it to parallelize?

# Summary

- Complexity analysis

  ⇒ measures performance **for all inputs**
  especially **very large** inputs

- Things to measure

  — time in *primitive operations*
  — space in *primitive units*

- Asymptotic approach

  ⇒ abstracts from actual implementation/hardware
  ⇒ studies behaviour for large inputs

- better algorithm can solve much bigger problems

  ⇒ performance boost **not achievable**
  by speeding up hardware!