Computational Complexity From A Programming Perspective

Kai Brünnler

presenting an approach developed by Neil Jones et. al.

Outline

1. Turing Machines disregard constant factors
   - linear speedup theorem
   - time hierarchy theorem

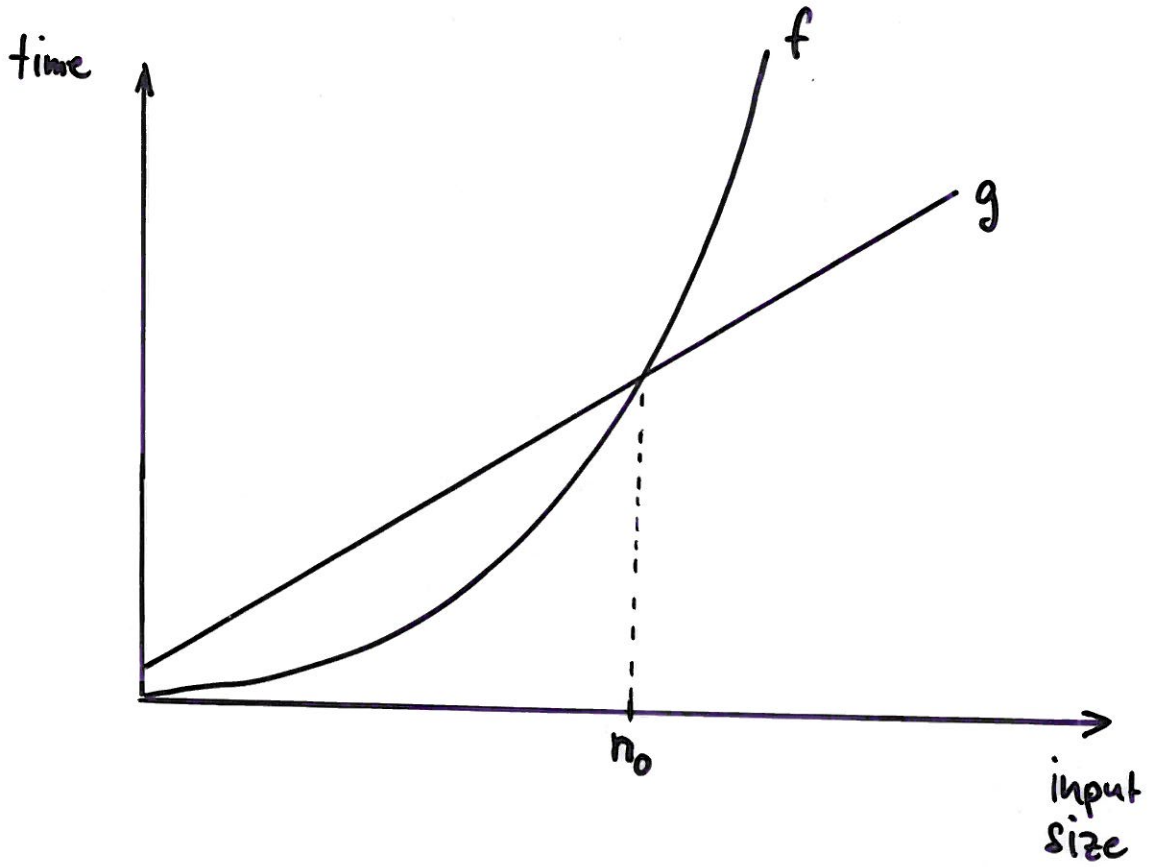
2. A programming language approach that respects constant factors
   - programs, data
   - an efficient self-interpreter
   - a linear-time hierarchy theorem

The Rôle of Constants

Def. $g(n)$ is $O(f(n))$ if

$\exists c, n_0 \; \forall n \geq n_0 \; g(n) \leq c \cdot f(n)$.

Ex. g is $O(n)$, f is $O(n^2)$



Linear Speedup

Theorem $\forall L, f, s \in \mathbb{N}$

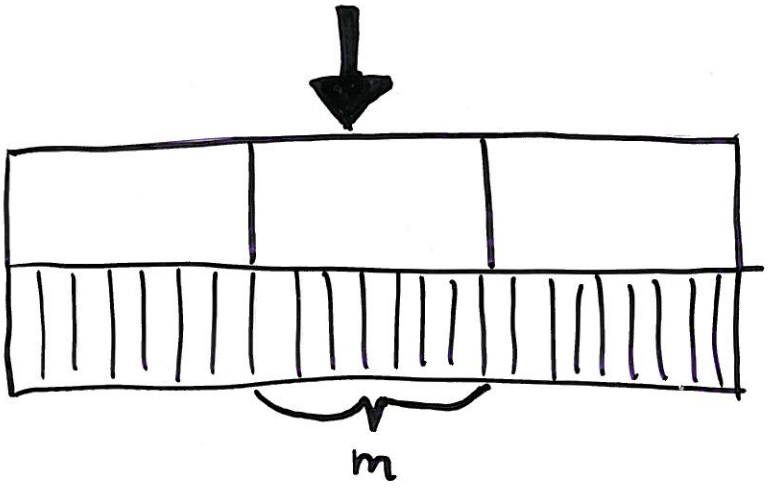
$$L \in \text{TIME}(f(n)) \Rightarrow L \in \text{TIME}\left(\frac{f(n)}{s} + n + 2\right)$$

Proof

$m = 6 \cdot s$

$\Sigma_{new} = \Sigma \cup \Sigma^m$

$Q_{new} = Q \times \{1, \ldots, m\} \times \Sigma^{3m}$



Time Hierarchy

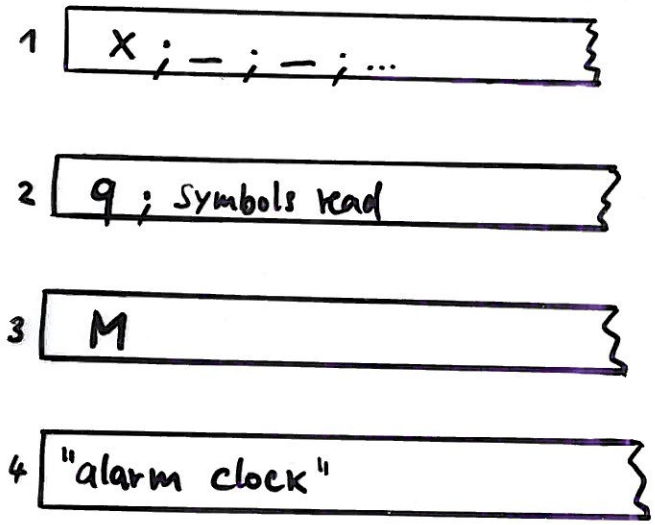
Theorem

$$\forall \text{ proper } f.\ \ \mathrm{TIME}(f(n)) \subsetneq \mathrm{TIME}(f(2n+1)^3)$$

Proof

$$L_f = \{ (M,x) \mid M \text{ accepts } x \text{ after at most } f(|x|) \text{ steps} \}$$

1) $L_f \in \mathrm{TIME}(f(n')^3)$ $\qquad n' = 2n+1$



one step of M : $O(f(n')^2)$

total : $O(f(n')^3)$

2) $L_f \notin \mathrm{TIME}(\lfloor \frac{n'}{2} \rfloor)$: diagonalisation.

DATA

$\mathbb{D} ::= \text{nil} \mid (\mathbb{D}.\mathbb{D})$

$\text{size}(\text{nil}) = 1$

$\text{size}((d_1.d_2)) = 1 + \text{size}(d_1) + \text{size}(d_2)$

$(d_1\ d_2\ \ldots\ d_n)$ is short for $(d_1.(d_2.(\ldots(d_n.\text{nil})\ldots)))$

an integer $i \in \mathbb{N}$ is short for $\underbrace{(\text{nil}\ \text{nil}\ \ldots\ \text{nil})}_{i \text{ times}}$

false is short for nil,

true is short for non-nil

PROGRAMS

	informal	formal
C ::=	$X_i := E$	(:= i E)
	$C_1 ; C_2$	(; C_1 C_2)
	if E then C_1 else C_2	(if E C_1 C_2)
	while E do C	(while E C)

E ::=	X_i	(var i)
	D	(quote D)
	cons E_1 E_2	(cons E_1 E_2)
	hd E	(hd E)
	tl E	(tl E)

P ::=	read X_i ; C ; write X_j	C

A SELF-INTERPRETER

$$[\![\text{int}]\!](p.d) = [\![p]\!](d)$$

```
read PD;
    CodeStack := cons (hd PD) nil;
    Value := tl PD;
    DataStack := nil;
    while CodeStack do STEP;
write Value;
```

State : (CodeStack, DataStack, Value)

$$\text{state} \xrightarrow{\text{STEP}} \text{next state}$$

THE STEP MACRO

CodeStack	DataStack	Value	CodeStack	DataStack	Value
(while E C). CS			E. dowhile. (while E C). CS		
dowhile. (while E C). CS	(T.U). DS		C. (while E C). CS	DS	
dowhile. C. CS	nil. DS		CS	DS	

Code for the STEP Macro

```
if hd hd CodeStack = while then
    CodeStack := cons (hd tl hd CodeStack) (cons do_while CodeStack)
                                else
    if hd CodeStack = dowhile then if hd DataStack then
                { CodeStack := cons (hd tl tl tl CodeStack) (tl CodeStack);
                  DataStack := tl DataStack }
                                else
                { CodeStack := tl tl CodeStack;
                  DataStack := tl DataStack }
                                else
                                ...
```

"EFFICIENCY"

int is "efficient" in the following sense:

usually (e.g. Turing Machines):

$$\text{time}(\text{int}, p.d) \leq K_p \cdot \text{time}(p, d)$$

we have:

$$\exists K \, \forall p \quad \text{time}(\text{int}, p.d) \leq K_0 \cdot \text{time}(p, d)$$

A Timed Interpreter

tint "runs a program up to a time limit"

$$[\![\text{tint}]\!](p\ d\ \text{nil}^n) = \begin{cases} ([\![p]\!](d).\text{nil}) & \text{if } \text{time}(p,d) \leq n \\ \text{nil} & \text{otherwise} \end{cases}$$

$$\exists k \forall p\ \text{time}(\text{tint}, (p\ d\ \text{nil}^n)) \leq k \cdot \min(n, \text{time}(p,d))$$

DIAGONALISATION

```
diag_a :   read X;
           Timebound := nil^(a·|x|);
           X := tint cons x(cons X (cons Timebound nil));
           if hd X then X := false
                   else X := true;
           writ X;
```

"take a program, run it on itself, do the opposite"

The Linear-Time Hierarchy

Theorem

$$\exists b \forall a \geq 1 \quad \mathrm{TIME}(a \cdot n) \subsetneq \mathrm{TIME}(a \cdot b \cdot n)$$

Proof

$$A = \{ p \mid [\![\mathrm{diag}_a]\!](p) = \mathrm{true} \}$$

Claim 1 : $A \in \mathrm{TIME}(a \cdot b \cdot n)$

Claim 2 : $A \notin \mathrm{TIME}(a \cdot n)$

Proof of Claim 1

diag$_a$ runs in time:

$$\mathrm{nil}^{a\cdot|p|} : c\cdot a\cdot|p|$$

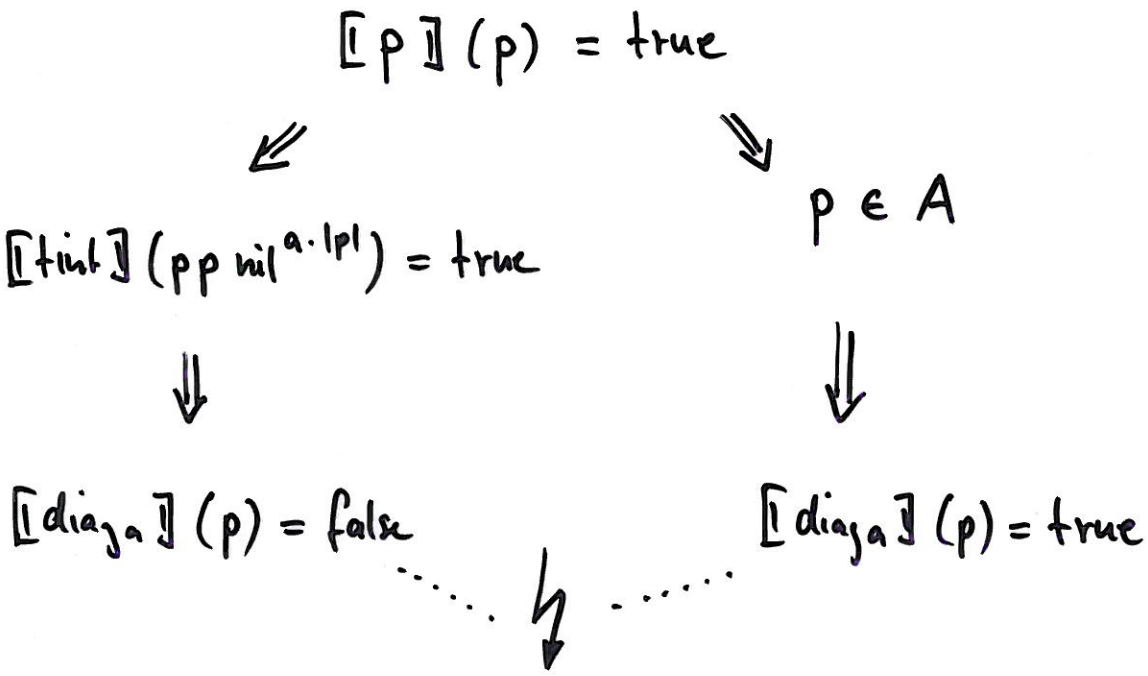
$$\mathrm{tint}\ (p\ p\ \mathrm{nil}^{a\cdot|p|}) : k\cdot\min(a\cdot|p|, \mathrm{time}(p,p))$$
$$\leq k\cdot a\cdot|p|$$

total:

$$c\cdot a\cdot|p| + k\cdot a\cdot|p| + e$$
$$\leq a\cdot\underbrace{(c+k+e)}_{b}\cdot|p|$$

Proof of Claim 2

Suppose there is a p that decides A in $\mathrm{TIME}(a \cdot n)$



Conclusion

1. Constant factors matter, from a programming perspective.

2. Turing Machines disregard constant factors.

3. We have seen a programming language approach to computational complexity, which
   - respects constant factors
   - is closer to programming practice