Efficient Algorithms and Problem Complexity

– Introduction to Problem Complexity –

Frank Drewes
Department of Computing Science
Umeå University
Today’s Menu

1. About Problem Complexity

2. Decision Problems and the Class P
What is Problem Complexity?

- Obviously (?) some problems can be solved more efficiently than others.
- As always, we measure efficiency depending on input size.
- Efficiency can refer to very different aspects:
  - solving the problem \textit{quickly},
  - solving it in \textit{little memory space},
  - solving it on a parallel computer using \textit{few processors},
  - solving it in a distributed way with \textit{little communication},
  - solving it with a \textit{small circuit},
  - solving it using a \textit{short program},
  - \ldots
- There are at least \textbf{3 versions} of each of these:
  - worst case
  - average case
  - best case
But What Is a Problem?

A problem is a function \( A : \mathcal{I} \rightarrow \mathcal{O} \) from a set \( \mathcal{I} \) of inputs to a set of outputs \( \mathcal{O} \).

- If \( \mathcal{O} = \{ \text{yes}, \text{no} \} \), then the problem is a decision problem.
- A decision problem \( A \) can be identified with \( \{ I \in \mathcal{I} \mid A(I) = \text{yes} \} \).
- If we want to stress that \( A \) is not necessarily a decision problem, we call it a function problem.
- More about both types of problems later...
And, by the Way, What Is an Algorithm?

When studying problem complexity, we must be somewhat more precise about what constitutes an algorithm:

- We do not discuss concrete algorithms, but *algorithms we do not know* ⇒ it is important to agree on a suitable *model of computation*.
- The model must allow us to *measure input size and running time*.
- We may use pseudocode, but must be aware of the underlying model.
- Traditional choice:
  - Model: *Turing machine*
  - Input/output: strings
  - Size (of input): length of string
  - Time consumed: number of steps
- The *random access machine* (RAM) is more similar to a modern computer.
Deterministic Random Access Machines (RAMs)

A deterministic random access machine $M$ has a program counter $pc$ and registers $R_0, R_1, R_2 \ldots$ storing non-negative integers $R_0, R_1, R_2, \ldots$. A program is a sequence of instructions of finite length $m \in \mathbb{N}$. The available instructions are:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_i \leftarrow j$ or $R_i \leftarrow R_j$ or $R_i \leftarrow RR_j$</td>
<td>store $j$, $R_j$ or $RR_j$ in $R_i$</td>
</tr>
<tr>
<td>$RR_i \leftarrow j$ or $RR_i \leftarrow R_j$ or $RR_i \leftarrow RR_j$</td>
<td>store $j$, $R_j$ or $RR_j$ in $RR_i$</td>
</tr>
<tr>
<td>$R_i \leftarrow R_j + R_k$</td>
<td>store $R_j + R_k$ in $R_i$</td>
</tr>
<tr>
<td>$R_i \leftarrow R_j - R_k$</td>
<td>store $\max(0, R_j - R_k)$ in $R_i$</td>
</tr>
<tr>
<td>if $i$ then goto $j$ ($1 \leq j \leq m + 1$)</td>
<td>set $pc$ to $j$ if $R_i &gt; 0$</td>
</tr>
</tbody>
</table>

In all cases except the last (if $R_i > 0$), increment $pc$ by 1.

Note that there is no multiplication instruction!
Deterministic Random Access Machines (RAMs)

For $d \in \mathbb{N}$, let $||d||$ denote the length of $d$ written in binary notation. A random access machine $M$ works as follows:

- The input is a sequence $x = a_1 \cdots a_d \in \mathbb{N}^*$ which is stored in registers $R_1, \ldots, R_d$. $R_0$ initially contains $2^d - 1$ (thus, $||R_0|| = d$ unless $d = 0$).
- $R_i = 0$ for all $i > d$, and the program counter $pc$ is set to 1.
- $M$ repeatedly executes $I_{pc}$, and terminates when $pc = m + 1$.
- The output is $R_1 \cdots R_{||R_0||}$, which is denoted by $M(x)$.

Measuring input length and running time:

- The input length is $\sum_{j=1}^{d} ||a_i||$.
- The running time is the number of steps executed until $pc = m + 1$.

Question: Why do we set $R_0 = 2^d - 1$ rather than $R_0 = d$?
Polynomial-time RAMs

- A RAM runs in polynomial time if there is some \( k \in \mathbb{N} \) such that the running time of \( M \) is \( O(n^k) \) for all inputs \( x \) of length \( n \).
- Easy exercise: Show that, in this case, \( M \) runs in time \( n^l + c \) for some \( l, c \in \mathbb{N} \) (i.e., we may assume that the constant factor is 1).
- A RAM \( M \) decides a decision problem \( A \) if, for all inputs \( x \),

\[
M(x) = \begin{cases} 
1 & \text{if } x \in A \\
0 & \text{otherwise.}
\end{cases}
\]

- The class of all decision problems that can be decided in polynomial time by a RAM is denoted by \( P \).
A Few Remarks about Encodings

- Encodings are important to be aware of, even though they are usually not made explicit.
- A RAM that computes $f(n)$ in time $O(n^k)$ is not polynomial.
- Such a RAM is called pseudopolynomial
  ⇒ be careful when encoding numbers in unary (or don’t do that)!
- All decision problems can be encoded as sets of strings (languages).
- Encoding each symbol $\sigma_i$ in an alphabet $\Sigma = \{\sigma_1, \ldots, \sigma_k\}$ as $i$, strings can be given as input to a RAM. (Think ASCII or UTF.)
- For problems involving numbers, other encodings may be more natural.
- Reasonable encodings can be transformed into each other in polynomial time (usually $O(n)$ or $O(n^2)$).
A Few Remarks about Encodings

But we usually talk about instances of a problem, disregarding non-instances. So, what about inputs that are not valid instances at all?

**Theorem (disregarding invalid inputs)**

Let $A$ be a decision problem. If there is a RAM such that
- for all inputs $x$, $M(x) = 1$ if and only if $x \in A$, and
- $M$ runs in polynomial time if the input is in $A$,

then $A$ can be decided in polynomial time (i.e., $A \in P$).

**Proof sketch:** Suppose $M$ runs in time less than $n^k + c$ on inputs in $A$. Construct a new RAM $M'$ that works as follows:

1. With input $x$, start by computing a “yardstick” $R_Z = |x|^k + c$ in some register $R_Z$ (easy to do in time $O(n^k)$).
2. Continue precisely like $M$, but decrease $R_Z$ by 1 in each step.
3. Moreover, if $R_Z$ ever reaches 0, stop with output 0.
Composition of polynomial-time RAMs

Another important thing is to be able to combine algorithms from already existing ones. For $P$ to be a robust class, it should be closed under such constructions.
Composition of polynomial-time RAMs

Composition Theorem

If $M_1$ and $M_2$ are polynomial-time RAMs, the composition $M_2 \circ M_1$, where $M_2$ is applied to the output of $M_1$, can be computed by a polynomial-time RAM as well.

Proof sketch: Suppose $M_1$ and $M_2$ run in time $O(n^k)$ and $O(n^l)$, for some $k, l \geq 1$. Construct a new RAM $M$ that works in the obvious way:

1. With input $x$, start by computing $M_1(x)$.
2. When $M_1$ has terminated, continue by working like $M_2$.

Clearly, phase 1 runs in time $O(n^k)$

$\Rightarrow$ the length of $M_1(x)$ is $O(n^{2k})$ [WHY?]

$\Rightarrow$ phase 2 runs in time $O((n^{2k})^l) = O(n^{2kl})$

$\Rightarrow$ the total time is $O(n^{2kl})$. 