Part I

Background

Chapter 0

Computability

Notion 1. *A problem is "computable" if it can be solved by transforming a mathematical object over a finite amount of time, without ingenuity.*

Any attempt at a more definite notion of computability seems to arrive at a philosophical impasse, where the notions of "transformation", "mathematical object", and "ingenuity" form a philosophical conundrum. The indefinite notion however, is sufficient to state the following theorem:

Theorem 1. *The class of computable problems is closed under concatenation.*

That is, if a problem P can be solved by solving a computable problem Q, followed by solving a computable problem R, then P itself is computable.

Proof. Since both Q and R are computable, and no transformations are performed, other than to solve the problems Q and R, P itself is computable. $\square$

Thus we arrive at the folklore notion of an algorithm:

Notion 2. *An "algorithm" is a specification of how a problem can be solved by solving a finite sequence of computable problems.*

Such indefinite notions are useful for little else. We are left to take a philosophical leap of faith and define some notion of mathematical object, and transformation without ingenuity.

0.1 Function Algebras

A formal system is a system of mathematical symbols and rules for employing them. We'll refer to formal systems as algebras.

An algebra is a set of symbols and rules for manipulating them. In this sense, and algebra

Definition 1. *A **function algebra** $\mathcal{A}$ is a set of functions, including a set of basic functions $\mathcal{A}_{\mathrm{B}}$ and a set of operations $\mathcal{A}_{\mathrm{O}}$.*

0.2 Machines

TODO:

- The above definition is useful for little else - provide some historical perspective on defining computability beyond this notion, and provide a characterization using function algebras (similar to Church) and Turing machines. Draw parallels to type theory an constructivism.
- The classical result that primitive recursive functions are computable. To argue for this, we probably need to argue for a type theoretic approach, in that, what we can construct, we can compute. Function algebras should also be introduced here.
- General recursion (due to Kleene wrt. definition) and it's undecidability (due to Church).
- A different approach to computability: Post and Turing machines. Prove their equivalence to general recursion above.

Chapter 1

Complexity

This may be a good point to mention that, although I have so far been tacitly equating computational difficulty with time and storage requirements, I don't mean to commit myself to either of these measures. It may turn out that some measure related to the physical notion of work will lead to the most satisfactory analysis; or we may ultimately find that no single measure adequately reflects our intuitive concept of difficulty.

— ALAN COBHAM, *Logic, Methodology and Philosophy of Science* (1964)

In practice, the length of computer computations must be restricted, otherwise the cost in time and money would be prohibitive.

— H. E. ROSE, *Subrecursion: functions and hierarchies* (1984)

Definition 2. *The computational complexity of a function* f*, wrt. a particular resource, quantifies the use of that resource as a function of the length of the input string.*

1.1 Time

1.1.1 Polynomial Time

- Recursive characterization of polytime functions in [**?**], proving certain claims by [**?**]. Both question the relation to the Grzegorczyk hierarchy [**?**].
- Leivant's paper - A Foundational Delineation of Computational Feasibility.
- Bellantoni and Cook paper - A NEW RECURSION-THEORETIC CHARACTERIZATION OF THE POLYTIME FUNCTIONS
- Niel Jones paper.
- Caporaso
- Upper bounds (algorithms) can be produced by expressing the property of interest in one of our languages. Lower bounds proven elsewhere can be used as a proof that the language is expressive enough.

1.1.2 Subpolynomial Time

In what follows, we delineate a hierarchy of complexity classes, strictly under polynomial time. That is, we present a sequence $C_1(n)$, $t(n) = o\left(n^{O(1)}\right)$. For each complexity class C, we present a representative problem P_C. We aim to find problems which are known to be $t(n) = \Theta\left(C\left(n\right)\right)$.

- Some problems, although computable in polynomial time, are still hard to compute in practice (ICALP'2014, Amir Abboud).
- Remind of the definitions of O, Ω, etc.
- For each of the below show that every subsequent class is distinct from the proceeding, and exhibit some "complete" problems for these classes.

$O\left(1\right)$ **— Constant**

$O\left(\alpha\left(n\right)\right)$ **— Inverse Ackermann**

$O\left(\log^*\left(n\right)\right)$ **— Log star**

$O\left(\log\log\left(n\right)\right)$ **— Log-log**

$O\left(\log\left(n\right)\right)$ **— Log**

$O\left(\log\left(n\right)^{O(1)}\right)$ **— Polylog**

$O\left(n^c\right)$**, for** $0 < c < 1$ **— Fractional power**

$O\left(n\right)$ **— Linear time**

$O\left(n\log^*\left(n\right)\right)$ **— n log star**

$O\left(n\log\log\left(n\right)\right)$ **— n log-log**

$O\left(n\log\left(n\right)\right)$ **— n log n** Comparison-based sorting $o\left(n\log n\right)$.

$O\left(n^2\right)$ **— quadratic**

$O\left(n^3\right)$ **— cubic**

$O\left(n^O(1)\right)$ **— polynomial**

1.1.3 Space

Chapter 2

Implicit Characterizations of P

2.1 Primitive Recursion on Notation

When dealing with the manipulation of symbolic strings, there is a natural total order on the values of the formal parameters — the length of their representing string. Primitive recursion on notation utilizes this order, requiring that the length of the input string be decreased before a recursive call.

Definition 3. *A function* f *is defined by primitive recursion on notation from functions* $g, h_1, h_2, \ldots, h_{|\Sigma|}$ *iff*

$$f\left(\varepsilon, \bar{t}\right) = g\left(\bar{t}\right) \tag{2.1}$$

$$f\left(s_i \cdot s, \bar{t}\right) = h_i\left(s, \bar{t}, f\left(s, \bar{t}\right)\right) \qquad \forall s_i : \Sigma \tag{2.2}$$

We say that a function is primitive recursive on notation (PRN), if it is defined by primitive recursion on notation from non-recursive or PRN functions.

Unfortunately, not all PRN functions take polynomial time.

Theorem 2. *There exist PRN functions which do a superpolynomial amount of work.*

Proof. Consider a function g, which duplicates every symbol in the input string:

$$g\left(\varepsilon\right) = \varepsilon \tag{2.3}$$

$$g\left(0 \cdot s\right) = 0 \cdot 0 \cdot g\left(s\right) \tag{2.4}$$

Consider furthermore a function h, which calls g iteratively, the same number of times as the length of its input string:

$$h\left(\varepsilon\right) = 0 \tag{2.5}$$

$$h\left(0 \cdot s\right) = g\left(h\left(s\right)\right) \tag{2.6}$$

Calling g with a string of length n, we obtain a string of length 2^n due to iterated duplication. It follows that g does a superpolynomial amount of work. □

Example 1. *We illustrate the above proof with an example:*

$$\begin{aligned}
h(0 \cdot 0) &\rightsquigarrow g(h(0)) && (2.7)\\
&\rightsquigarrow g(g(h(\varepsilon))) && (2.8)\\
&\rightsquigarrow g(g(0)) && (2.9)\\
&\rightsquigarrow g(0 \cdot 0 \cdot g(\varepsilon)) && (2.10)\\
&\rightsquigarrow g(0 \cdot 0) && (2.11)\\
&\rightsquigarrow 0 \cdot 0 \cdot g(0) && (2.12)\\
&\rightsquigarrow 0 \cdot 0 \cdot 0 \cdot 0 \cdot g(\varepsilon) && (2.13)\\
&\rightsquigarrow 0 \cdot 0 \cdot 0 \cdot 0 && (2.14)
\end{aligned}$$

2.2 Bounded Primitive Recursion on Notation

With reference to the "extended rudimentary functions" of [**?**], [**?**] defined bounded primitive recursion on integers in decimal notation. For a similar definition on binary notation, see [**?**, p. 127].

We generalize this to an arbitrary alphabet Σ, building upon PRN above:

Definition 4. *A function* f *is defined by bounded primitive recursion on notation from functions* $g, h_1, h_2, \ldots, h_{|\Sigma|}$, *and* k *iff*

$$\begin{aligned}
f(\varepsilon, \overline{y}) &= g(\overline{y}) && && (2.15)\\
f(s_i \cdot x, \overline{y}) &= h_i(x, \overline{y}, f(x, \overline{y})) && \forall s_i : \Sigma && (2.16)\\
f(x, \overline{y}) &\leq k(x, \overline{y}) && && (2.17)
\end{aligned}$$

We say that a function is bounded primitive recursive on notation (BPRN) if it is defined by bounded primitive recursion on notation from non-recursive, or BPRN functions. The scheme is also known as limited primitive recursion on notation.

The addition that we make to PRN is that the function f must be bounded from above by function i. Letting i characterize polynomial time functions, we obtain perhaps the earliest implicit characterization of P due to [**?**].

This straight-forward approach has an obvious limitation: it requires defining an ordering relation on our functions — a problem that is undecidable in general. Furthermore

2.3 Finite Model Theory

Finite model theory is

2.4 Ramification

2.4.1 Safe Recursion

2.4.2 Tiering