What Is Computation?

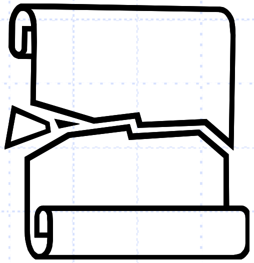
Locating the Computational Dynamics of Nature

Within the Mathematics of Computation

Algorithms

- An algorithm is a procedure or sequence of steps for computing outputs from given inputs
- Algorithms can be implemented in programming languages like Java and C
- We will study techniques for creating algorithms, measuring their efficiency, and refining their performance

Foundational Questions



- In this introductory lesson, we ask What Is Computation?
- Important to know the answer so we can manage computation efficiently
  - for use in technology
  - in our own lives

How Does Nature Compute?

- Physics uses mathematical models to describe how Nature operates. But how does Nature actually do it? How is the computation of orbits of the planets actually computed by Nature? Not likely that our mathematical formulas are being used…
- Important to know the answer if we wish to harness the power of Nature for our own use

Magician Analogy

- Watch an illusion at a magic show – like a woman being sawed in half. How is it done?
- The procedure that you witness, the sequence of steps you are able to reproduce, is not the procedure that the magician used.
- *There is more to the intelligence behind the performance than can be found in the performance itself*

Analogy To Daily Life

- How to build a successful career? How to make a lot of money?
- One approach: Watch and imitate what successful people do in America… Will this work? Does the success lie in the obvious steps of the performance? Maybe partial success is possible this way…but is the magician really sawing the lady in half? Maybe there is something more to it…

Modern Physics

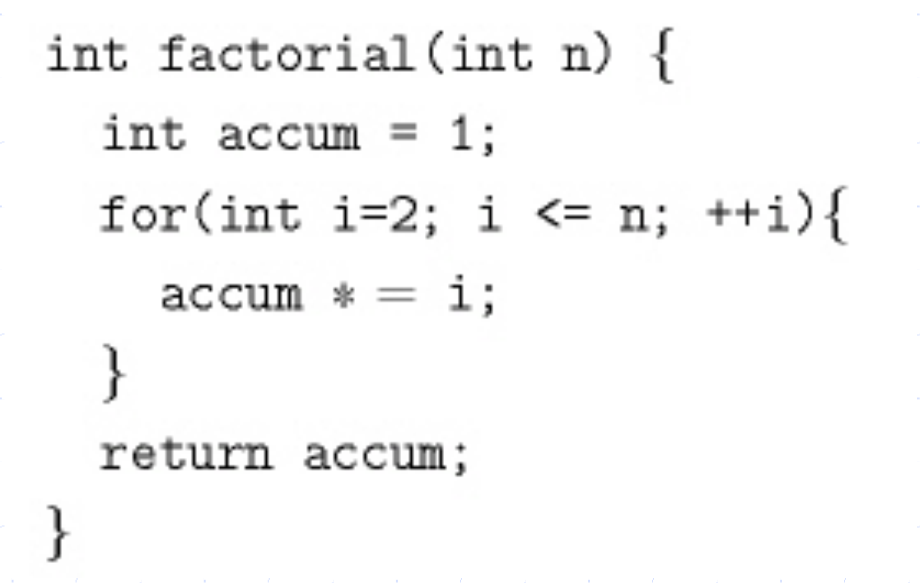
- Quest to understand how nature computes.
- Early understanding of physics about the world: there are particles (objects) and forces that act upon them.
- Deeper understanding: forces and particles are excitations of quantum fields. Examples: Electron field, electromagnetic field, gravitational field
- Next step: At small time and distance scales (Planck scale), different matter and force fields are seen to be identical.
- Deepest reality: the "show" of existence is the manifestation of unseen self-referral dynamics of a single unified field
- This is very unexpected based on ordinary experience of the world – the world is not as it appears; self-referral dynamics are the underlying reality of all that we see and hear

Self-Referral Dynamics at the Heart of Computation?

- Just as we want to discover the secret of Nature’s computing power so we can harness it, so likewise do we want to discover the secret underlying computation itself so we can likewise master it. Are self-referral dynamics in some way central to computation as they are for Nature’s functioning?
- Advantages:
  - Improve technology
  - Improve life

The Surface Value of Algorithms

- On the surface, an algorithm is a sequence of steps, a procedure, for computing outputs from given inputs
- Typical implementation: a piece of procedural code not involving recursion:

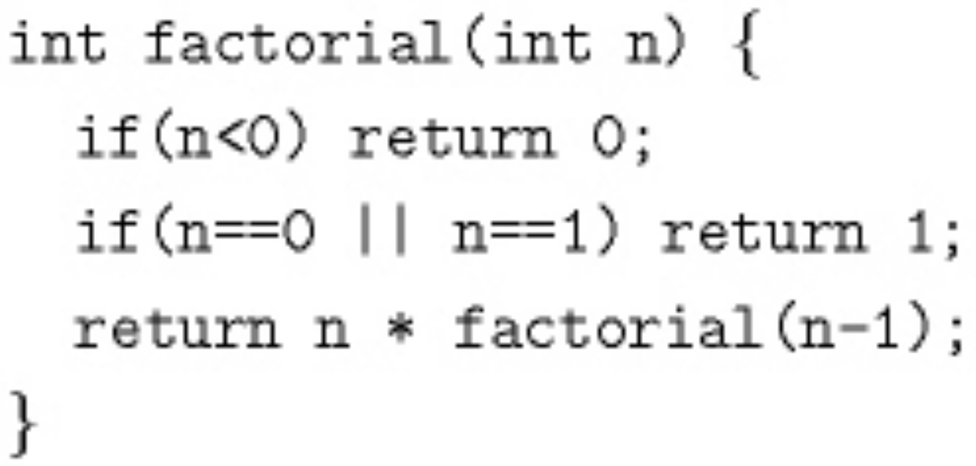


```
int factorial(int n) {
  int accum = 1;
  for(int i=2; i <= n; ++i){
    accum *= i;
  }
  return accum;
}
```

Recursion In Algorithms and Programs

- Sometimes recursion is used in the design and/or impelmentation of an algorithm. Recursion exhibits the flavor of self-referral dynamics:

  factorial(n) = n * factorial(n-1)



```
int factorial(int n) {
  if(n<0) return 0;
  if(n==0 || n==1) return 1;
  return n * factorial(n-1);
}
```

Deeper Layers of Self-Referral

- **Fact #1**: Every computable function can be represented (and implemented) as a recursively defined function. (See "**The Self-Referral Dynamics Of Computation**" **article.)**

- **Fact #2**: Recursion allows us to define the factorial function fact like this:

  fact(n) = F(n, fact(n-1))

  A deeper analysis shows that it can be defined like this, as a purely self-referral expression:

  fact = F(fact)

Deeper Layers (cont.)

- In this example fact is a *fixed point* of the operator F, which is a function from the set S of partial functions on the natural numbers to itself.
- Roughly, F is defined as follows: Given a function g defined on a subset of N, F(g) is the function h that behaves as follows:

$$h(n) = \begin{cases} 1 & \text{if } n = 0, 1 \\ n * g(n-1) & \text{otherwise} \end{cases}$$

Deeper Layers (cont.)

- **Fact #3**: Every computable function arises as the fixed point of an operator like F; that is, of an operator mapping functions in S to functions in S. Here is the well-known theorem:
- **Kleene Recursion Theorem**: For every recursively defined function f there is an operator F: S -> S so that f is the least fixed point of F; that is, f = F(f)
- This shows that the deeper reality about the nature of computation is self-referral dynamics; sequential computation arises from self-interacting dynamics.
- This discovery is only possible by *expanding the context*: We needed to look at the domain of operators from S to S – this represents an infinity much vaster than that of the computable functions from N to N. The point is that these self-referral dynamics are not obvious on the surface, where applied values of computing dominate.

Computation According to the Ancients

- *According to the ancients…*According to ancient texts the answer to “How does Nature compute?” and “What is computation, ultimately?” is simply: the self-interacting dynamics of pure intelligence. Nature’s computation is at the depth a self-referral performance, effortless creation.

- *The nature of pure intelligence to flow.* The self-interacting dynamics of pure intelligence arise by the nature of intelligence as *pure wakefulness.* Intelligence is awake to itself. This results in a relationship within intelligence of rishi-devata-chhandas. Each is awake to each of the others. This results in a flow within intelligence, a flow of self-knowing, resulting in sequential unfoldment of creation itself.

- *Sequential dynamics arise from self-referral dynamics.*The sequential nature of computation as it is expressed in physics and computer science are, from the point of view of Vedic science, expressions of the eternal self-referral performance of Nature’s intelligence

Practical Value

- *Summary*: The dynamics at the basis of Nature's enormous power and at the root of computation itself are
  - Hidden from ordinary perception
  - Self-referral
- *Practical application:* Because hidden, we need to take a little time to step into this deeper field – this is the purpose of TM. Because self-referral, stepping into this field serves simply to wake us up to our true nature, to our own self-interacting dynamics, beneath the surface

Advice from the Ancient Texts

From the Bhagavad Gita:

II.45. *Nistraigunyo bhavarjuna*: Transcend, go beyond the field of change, step into the field of Being

II.48 *Yogastah kuru karmani*: Established in Being, perform action

II.49 *Durena hy avaram karma buddhi yogad dhananjaya*: Far away indeed from the balanced intellect is action devoid of greatness

Engaging Self-Referral Dynamics Directly

- TM allows the awareness to gain the benefit of self-referral dynamics of pure consciousness
- In the TM-Sidhi program, these self-referral dynamics are engaged directly
- Patanjali spoke of three elements to produce any effect at will: dharana, dhyana, samadhi (intent, flow of awareness, transcendence). When all three are present, the intent is manifested.

More self-referral in computation

- These dynamics can be seen at work in the foundations of computation
- Consider F: S -> S once again.
  - F = intent
  - empty function = = transcendence
  - $\perp \subseteq F(\perp) \subseteq F(F(\perp)) \subseteq \ldots$ = flow
  - togetherness = union of all pieces to produce factorial function `fact`

  `fact` = $\perp$ **U** $F(\perp)$ **U** $F(F(\perp))$ **U** …

Main Point

The computing behavior of computer programs can in every case be represented as *recursive* computation. The Kleene Recursion Theorem states that every recursive function f: N -> N may be "defined" by the self-referral expression

$$f = F(f)$$

for some recursive operator F: S -> S. Likewise, we learn from Maharishi Vedic Science (and also from modern quantum field theory) that the underlying reality of all computational dynamics in Nature is the self-referral performance of pure intelligence.

Unbounded Source of Computable Functions

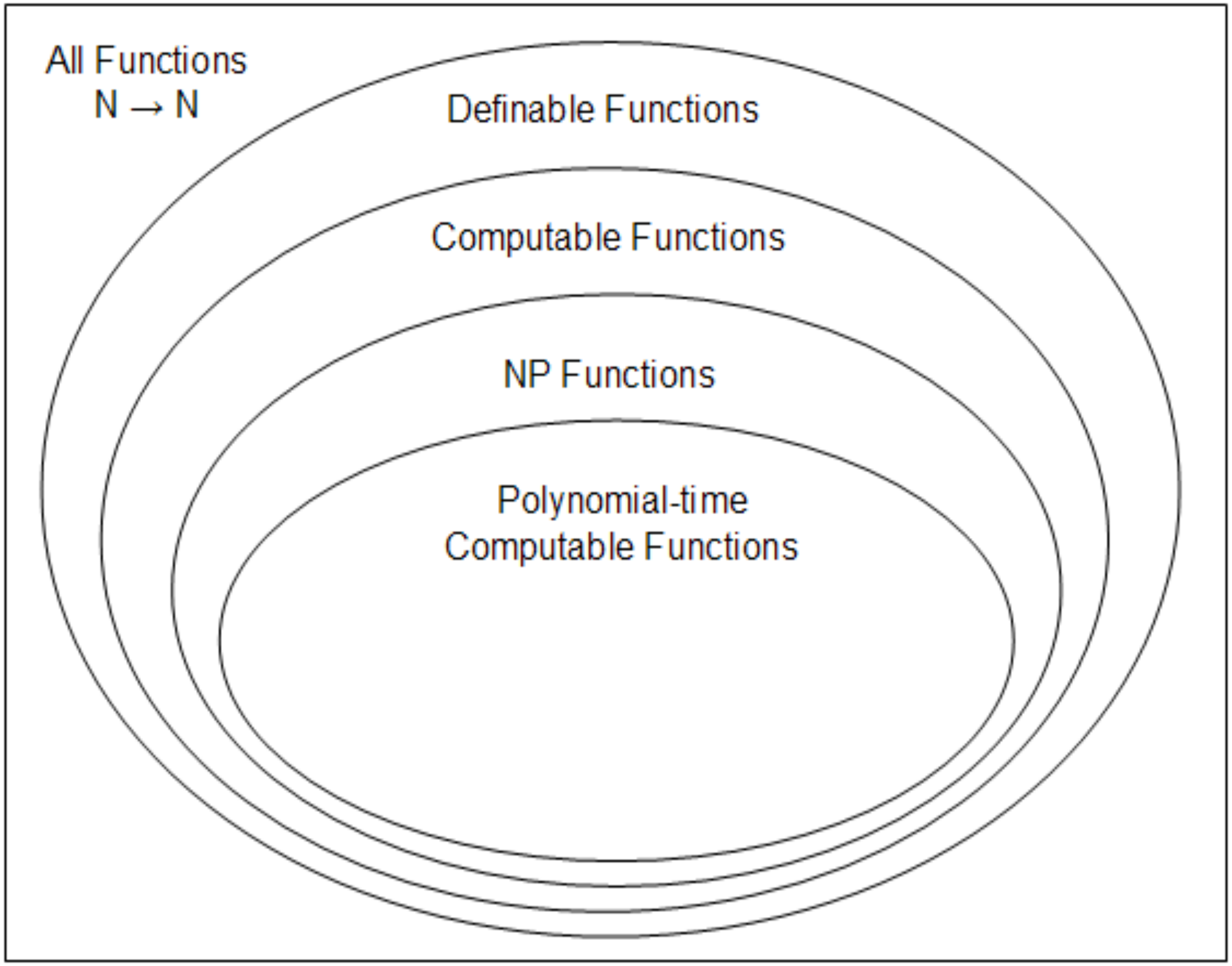
- All computations can be viewed as functions from N to N. Examples: factorial, sorting, graph algorithms
- A function from N to N is said to be *computable* if its input/output behavior can be represented in a computer program (here, one assumes a computer has unbounded memory)

- Example: The function f(n) = 2n is computable since we can represent the function in a program

```
int fcn(int n) {
        return n + n;
}
```

- The functions used in Computer Science for building software and models of the world – a small subset of the computable functions -- represent a tiny speck in the full range of all functions from N to N -- a “point” in the unbounded source of all functions.

The Collapse of Unboundedness to a Point



The Definable Functions

- "Definable" means we have some way of formally and unambiguously describing the input/output behavior of the function, using natural numbers as parameters. Examples:

  f(n) = 2n

  f(n, c) = 1 if n > c, 0 otherwise
- <u>Nearly All</u> functions from N to N are <u>Not</u> definable! There is no way to "grasp" the vast majority of functions N -> N.

Proof:

- "Descriptions" are finite-length strings of symbols (like English characters or math symbols). Though the symbols themselves may form an infinite collection, they can all be arranged in a long list. It follows that all descriptions can also be arranged in a long (infinite) list.

- Therefore all descriptions of functions can be arranged in a sequence like the following

  $d_1, d_2, d_3, \ldots$

  where d_1 describes f_1, d_2 describes f_2, …
- We therefore are able to give a complete list of all definable functions $f_1, f_2, f_3, \ldots$

- We can build a new function g like this:

  $g(n) = f_n(n) + 1$

  g is guaranteed to be different from every function on our list of definable functions. So g is *not* definable. This shows that *some* function is not definable. But *how many* undefinable functions are there?

- If we could arrange all the undefinable functions in a list (as we did for the definable functions), we could combine this list with the list of definable functions and find a new function h not in either list. This shows that the undefinable functions cannot be arranged in a list – they form a *bigger* class of functions.

$$\begin{tikzcd}
\text{ALL FUNCTIONS} \arrow[d] \\
\text{DEFINABLE FUNCTIONS} \arrow[d] \\
\text{COMPUTABLE FUNCTIONS}
\end{tikzcd}$$

Collapsing Further To Computable Functions

- Functions that are computable are definable in a special way – we know not only the behavior of the function but *how* to obtain outputs from given inputs in a step-by-step fashion (i.e. via a computer program)
- Classic example of a definable function that is not computable is known as the *Halting Problem.* One way to state it is: Consider the function H defined as follows. It takes as input a Java program P together with a positive integer n. If P outputs a value on input n, H(P,n) =1; if not, H(P,n) = 0. Is H computable?

- To understand the answer, we observe that every Java program may be encoded as a positive integer in such a way that, from the integer code, the program can be reconstituted.

Encoding Java programs

- Assume < 512 distinct characters are ever used in a Java program, so each character can be represented by a bit string of length 9
- Encode a program by eliminating white space, start with ‘1’, and concatenate the encoded characters one-by-one

```
BigInteger f(BigInteger[] n){
    return n[0].add(n[0]);
}
```

Char	Code	Char	Code	Char	Code
a	000000001	d	000000010	e	000000011
f	000000100	g	000000101	i	000000110
n	000000111	r	000001000	t	000001001
u	000001010	B	000001011	I	000001100
.	000001101	{	000001110	}	000001111
(	000010000	)	000010001	;	000010010
⟨space⟩	000010011	[	000010100	]	000010101
0	000010110				

```
1000001011000000110000000101000001100000000111000001001000000011000000101
 000000011000001000000001001100000001000000100000000010110000000110000000101
 000001100000000111000001001000000001100000010100000000110000010000000010100
 000010101000010011000000011100001000100000111000000010000000000110000001001
 000001010000000100000000011100001001100000011100001010000000101100000010101
 000001101000000000100000001000000000100000100000000000111000010100000010110
 0000101010000100010000100100000001111
```

- We also need the precise definition of H:

  H(e,n) = 1 if the program encoded by e halts when run on input n; else H(e,n) = 0.

- Suppose H is computable. Define another function G as follows:

  G(e) = 1 if H(e,e) = 0

  G(e) is undefined if H(e,e) = 1

  If H is computable, so is G: Let P be a program that computes values of H. Create a program Q that computes G like this: On input e, Q runs P on e. If P outputs 0, Q outputs 1 and halts. If P outputs 1, Q goes into an infinite loop.

- Use computability of G to obtain a contradiction. Suppose u is the integer that encodes the program Q. We run Q on input u and ask: Does Q output a value?
- Suppose Q does eventually halt on input u; then it outputs 1 on this input. By definition H (u,u) = 0. But this means that Q when run on input u, does *not* halt. Impossible.
- Suppose Q does *not* halt on input u. Then H (u,u) =1, which means that when Q is run on input u, it *does* halt. Impossible
- This shows that the assumption that H is computable leads to absurd conclusions – H is not computable.

$$\begin{tikzcd}
\text{ALL FUNCTIONS} \arrow[d] \\
\text{DEFINABLE FUNCTIONS} \arrow[d] \\
\text{COMPUTABLE FUNCTIONS} \arrow[d] \\
\text{FEASIBLY COMPUTABLE FUNCTIONS}
\end{tikzcd}$$

Collapsing Further To Feasibly Computable Functions

- Knowing that a function is computable does not guarantee that it is computable *in practice.*
- The Sorting problem. One algorithm for sorting an array of integers MinSort (also known as SelectionSort). If the input array has n elements, MinSort requires roughly n*n steps to return a sorted array.
- For small to medium sized input arrays, MinSort can be used in practice for sorting.
  Example: An array with 1000 elements can be sorted by MinSort in roughly 1 million steps.

- The Knapsack Problem You have a knapsack (=bag) that holds a maximum weight W and you have a target "value" (in dollars) V. You want to put into the bag some or all of n items s_1, s_2, …, s_n, where for each i, the value of s_i (in dollars) is v_i and the weight of s_i is w_i. Is there a subset T of {1,2,3,…n} for which the sum of the weights w_i, $i \in T$, is $\leq$ W and the sum of the values s_i, $i \in T$, is $\geq$ V?

- A procedure for solving the Knapsack problem is to examine every possible subset of {1,2,3…n} and, if possible, pick one that gives the desired result.
- However, examining all subsets of a set with n elements requires (at least) 2^n steps. For even small values of n, 2^n steps exceeds feasible computational power of modern computers.

Example: if n = 100, 2^n equals approximately 10^{30}.

"Feasibly Computable" = "Polynomial bounded"

- If an algorithm requires exponentially many steps, relative to input size, it is considered an *infeasible solution*
- If an algorithm requires only polynomially many steps (n^2, n^3, n^4, etc), it is considered a *feasible solution.* Functions that represent feasible solutions are called *polynomial bounded.*

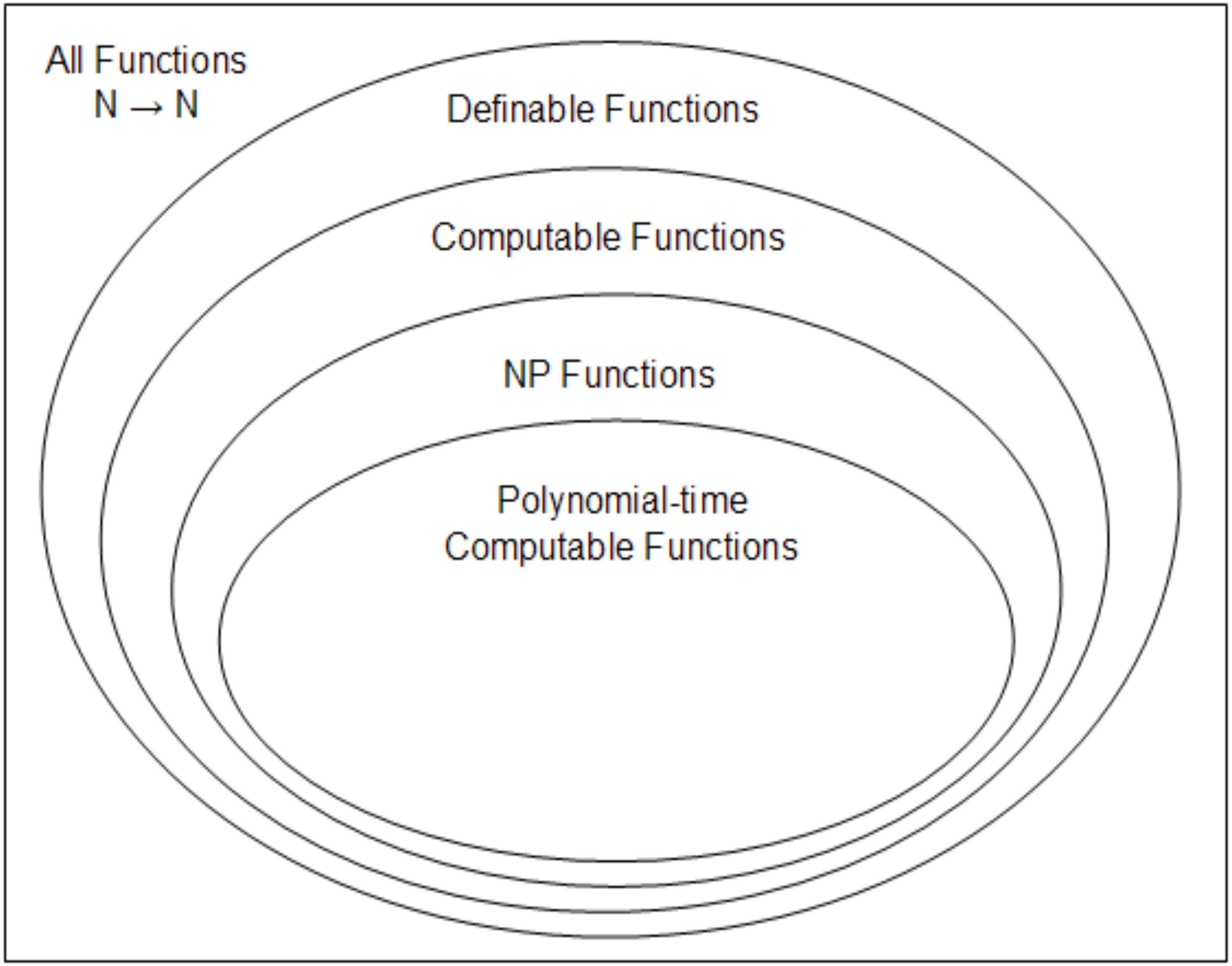
NP Functions / Problems

- Problems whose only known solutions are exponentially slow can be classified further by asking this question: For such a problem, how long does it take to *verify* that a solution is correct?

- Example: Suppose you are given a solution T to an instance of the Knapsack problem. How many steps are required to verify that T really is a solution?
- Steps:
  - Add up w_i for i in T and compare sum with W (approximately n steps)
  - Add up v_i for I in T and compare sum with V (approximately n steps)
  - Therefore, approximately 2n steps are required.

Definition of NP

- An algorithmic solution to a problem whose correctness can be checked in polynomial time is said to belong to NP ("nondeterministic polynomial time").
- It can be shown that every polynomial bounded function / algorithm belongs to NP. However, many NP functions/algorithms are not known to be polynomial bounded.


All Functions
N → N
Definable Functions
Computable Functions
NP Functions
Polynomial-time
Computable Functions

Main Point

The functions that are used in practice, at the basis of all the creative software applications of the IT industry, are the polynomial bounded functions, which represent only a tiny “point value” of the full range of number-theoretic functions N -> N. This fact illustrates the MVS theme that creation itself emerges in the collapse of unboundedness to a point.

Connecting the Parts of Knowledge With The Wholeness of Knowledge

1. Computation in computer science is represented by sequential procedures
2. The Kleene Recursion Theorem shows that every sequential procedure f can be *defined* by a self-referral expression of the form f = F(f) for some recursive operator F: S ->S

3. *Transcendental Consciousness* is the field of pure unbounded silence, beyond the active field of Nature’s computation.
4. *Impulses Within The Transcendental Field*. The hidden self-referral dynamics within the field of pure intelligence, on the ground of pure silence, give rise to the perfectly orderly unfoldment of creation
5. *Wholeness Moving Within Itself.* In Unity Consciousness, one appreciates the flawless unfoldment of life and existence as the lively impulse of one’s own pure consciousness.