Computer Scientist's View of Cantor's Diagonalization

CIS 300 Fundamentals of Computer Science

Brian C. Ladd

Computer Science Department
SUNY Potsdam
Spring 2023

Sunday 23rd April, 2023

Algorithms

Definition

Definition

An algorithm is a *finite* series of *precise instructions* for performing a computation or solving a problem that terminates with the *correct* answer in a finite amount of time.

Algorithms

Definition

Definition

An algorithm is a *finite* series of *precise instructions* for performing a computation or solving a problem that terminates with the *correct* answer in a finite amount of time. "*precise*" here will mean written in `Java`.

Algorithms

Definition

Definition

An algorithm is a *finite* series of *precise instructions* for performing a computation or solving a problem that terminates with the *correct* answer in a finite amount of time. "*precise*" here will mean written in `Java`.

Example (Algorithm: Maximum element in finite sequence)

```
// @precondition: A is not empty
int maxValue(int A[]) {
  int max = A[0];
  for (int i = 1; i < A.length; i++)
    if (A[i] > max) max = A[i];
    // max is maximum vaule on A[0-i] inclusive
  // i == A.length on exit; max on A[0-A.length − 1] inclusive
  return max;
}
```

Properties of an Algorithms

Input An algorithm has input values from a specified set

Properties of an Algorithms

Input An algorithm has input values from a specified set

Output From each set of input values, an algorithm produces output values, the solution, from a specified set

Properties of an Algorithms

Input An algorithm has input values from a specified set

Output From each set of input values, an algorithm produces output values, the solution, from a specified set

Definiteness The steps of the algorithm are defined precisely

Properties of an Algorithms

- **Input** An algorithm has input values from a specified set
- **Output** From each set of input values, an algorithm produces output values, the solution, from a specified set
- **Definiteness** The steps of the algorithm are defined precisely
- **Correctness** The algorithm should produce the *correct* output for each input value

Properties of an Algorithms

Input An algorithm has input values from a specified set

Output From each set of input values, an algorithm produces output values, the solution, from a specified set

Definiteness The steps of the algorithm are defined precisely

Correctness The algorithm should produce the *correct* output for each input value

Finiteness The solution must be produced in a finite number of steps

Properties of an Algorithms

Input An algorithm has input values from a specified set

Output From each set of input values, an algorithm produces output values, the solution, from a specified set

Definiteness The steps of the algorithm are defined precisely

Correctness The algorithm should produce the *correct* output for each input value

Finiteness The solution must be produced in a finite number of steps

Effectiveness It must be possible to perform each step in the algorithm precisely and in a finite amount of time

Properties of an Algorithms

Input An algorithm has input values from a specified set

Output From each set of input values, an algorithm produces output values, the solution, from a specified set

Definiteness The steps of the algorithm are defined precisely

Correctness The algorithm should produce the *correct* output for each input value

Finiteness The solution must be produced in a finite number of steps

Effectiveness It must be possible to perform each step in the algorithm precisely and in a finite amount of time

Generality The procedure should apply to all problems of the given form, not just a single input value

Algorithms

Linear Search

Example (Algorithm: Linear Search of Finite Sequence)

```
int indexOfMatch(int x, int A[]) {
  int match = -1; // no match yet found
  int i = 0;
  while ((match < 0)           // no match yet
         && (i < A.length)) {  // still list to check
    if (A[i] == x) match = i;  // remember the match
    i++;
  }
  // return index of match or -1 if no match
  return match;
}
```

Algorithms

Binary Search

Example (Algorithm: Binary Search of Sorted Finite Sequence)

```
// @precondition A is sorted and non-empty
int binaryMatch(int x, int A[]) {
  int low = 0;
  int high = A.length;
  // search interval half-open: [low, high)

  while (low < high - 1) { // while range > 1 element
    int mid = (low + high)/2; // mid = ⌊(low+high)/2⌋
    if (x > A[mid]) low = mid + 1;
    else high = mid;
  }
  if (A[low] == x) return low;
  else return -1;
}
```

Algorithms

Binary Search

Theorem

binary always terminates

Proof.

The only way it can **not** terminate is to be stuck in the `while` loop.
In the loop, $low \leq mid \leq high - 1$
high - *low* is the size of the range to be searched
(half-open so no extra $+$ 1)

Algorithms

Binary Search

Theorem

binary always terminates

Proof.

The only way it can **not** terminate is to be stuck in the `while` loop.
In the loop, $low \leq mid \leq high - 1$
high - *low* is the size of the range to be searched
(half-open so no extra $+ 1$)
high' - *low'*, the value after the loop, is smaller:

Algorithms

Binary Search

Theorem

binary always terminates

Proof.

The only way it can **not** terminate is to be stuck in the `while` loop.
In the loop, $low \leq mid \leq high - 1$
high - *low* is the size of the range to be searched
(half-open so no extra $+ 1$)
high′ - *low*′, the value after the loop, is smaller:
If `x > A[mid]`, $low' > low$
Otherwise, $high' < high$ because
$low \neq high$; $high' = \lfloor$ average $\rfloor < high$
Range to be searched is smaller on each iteration of loop; range initially finite so value must cross 1 terminating the while loop. □

Algorithms

Making Change

Example (Problem)

Given: Amount of change to make, $\mathtt{n} \in \mathbb{Z}^+$
Sequence of *r* coins: $\texttt{coins[0]} > \texttt{coins[1]} < \ldots < \texttt{coins[r-1]}$

Describe an *algorithm* to solve this problem.

Algorithms

Making Change

Example (Algorithm: Greedy Algorithm for Making Change)

```
// @precondition coins in decreasing order
// @precondition n >= 0
// @precondition any value n >= 0 can be made with coins
public static List<Integer> makeChange(int n, int coins[]) {
  List<Integer> change=new ArrayList<Integer>();
  for (int i = 0; i < coins.length; i++) {
    while (n >= coins[i]) {
      change.add(coins[i]);
      n -= coins[i];
    }
    // n < coins[i]: no more coins[i] can be part of change
  }
  return change;
 }
```

Algorithms

Change Making Correctness

Lemma

*$\forall\ n \in \mathbb{Z}^{\geq 0}$, n¢ using the **fewest** American coins possible can contain at most one 50¢ piece, one quarter, two dimes, one nickel, and four pennies. Further, it cannot contain two dimes **and** a nickel.*
The amount of change, excluding dollar coins, cannot exceed 99¢ = (50 + 25 + 10 + 10 + 1 + 1 + 1 + 1)¢

Algorithms

Change Making Correctness

Theorem

The greedy algorithm produces correct change in the fewest number of coins using American coins.

Deciders as Functions

A **language** is a *set* of *strings* across some *alhpabet*:

- *string* a *sequence* of zero or more symbols
- *alphabet* a *set* of symbols

Deciders as Functions

A **language**, L_d is a *set* of *strings* across some *alphabet*, Σ.
A decider is a predicate function, $d : \Sigma^* \to \{0, 1\}$ (if it is a *predicate*, how are we interpreting the result bits?)
$d(s) ::= 1$ if and only if $s \in L_d$.

Deciders as Algorithms

A decider can also be thought of as an *algorithm*:

Input Σ^*

Output $\{0, 1\}$

Correctness Returns 1 $\iff$ input $\in L_d$.

The other properties must be held so that d works for every string in Σ^*, finishes in finite time, is expressed in a finite number of steps, and so on.

Deciders as Algorithms

For ease of writing, a decider is a single boolean Java function.
So, to decide $L_{\text{even length}}$ across $\{0,1\}^*$, the following would work:

```java
boolean decide(String bin) {
  return (bin.length() % 2) == 0;
}
```

Deciding other Binary Languages

$\{\}$
$\{0, 1\}^*$
$\{\omega :$ an even number of 1s $\}$
$\{\omega :$ as a binary number, is divisible by 4$\}$

Deciding other Binary Languages

$\{\}$

```
boolean decide(String bin) {
  return false;
}
```

$\{0,1\}^*$

$\{\omega$: an even number of 1s $\}$

$\{\omega$: as a binary number, is divisible by 4$\}$

Deciding other Binary Languages

$\{\}$

$\{0, 1\}^*$

```
boolean decide(String bin) {
  return true;
}
```

$\{\omega$: an even number of 1s $\}$

$\{\omega$: as a binary number, is divisible by 4$\}$

Deciding other Binary Languages

$\{\}$
$\{0,1\}^*$
$\{\omega$: an even number of 1s $\}$

```
boolean decide(String bin) {
  int ones = 0;
  for (int c = 0; c < bin.length(); c++)
    if (bin.charAt(c) == '1') ones++;
  return (ones % 2) == 0;
}
```

$\{\omega$: as a binary number, is divisible by 4$\}$

Deciding other Binary Languages

$\{\}$
$\{0,1\}^*$
$\{\omega :$ an even number of 1s $\}$
$\{\omega :$ as a binary number, is divisible by 4$\}$

```
boolean decide(String bin) {
  return bin.endsWith("00");
}
```

The General Halting Problem

Is it possible to write an **algorithm** that when run on a *program*, *input* pair, determines if that *program* halts after a finite amount of time when run on that *input*.

How can we express the Halting Problem in terms of a *binary language*?

Encoding

An encoding is a way of representing some set of objects as bit strings.
For example, the integer range [0-255] (inclusive) could be encoded into a bit string of length 8 (a byte) with the sequence of bits interpreted as a *base 2 number*.
A set of printable and control characters might similarly be encoded into a bit string of length 8, each pattern mapped to a specific character.

Encoding a Java Program

A Java program is encoded (before compiling) as a string of characters from some character set. That character set, in turn, can encode each character as a string of 8 bits (or 16 or 32 bits, depending on the size of the character set).

Apply both encodings in turn and a Java program can be encoded into a bit string.

Deciding Java Programs

$L_{Java} = \{\omega : \omega \in \{0,1\}^* \wedge \omega \text{ encodes a valid Java program}\}$
The *Java decider* then looks like this:

```
boolean decide(String bin) {
  return validCharString(bin)
         && validJava(decodeCharString(bin));
}
```

Halting Problem

As a Language

Definition

Let $< P, I >$ be the encoding (into binary) of a program, P and input for that program I. The split between them must also be encoded.
Let $L = \{< P, I >\}$ be the set of all binary strings that encode a program followed by input for that program.
Then let $L_H = \{< P, I >: P(I)\text{only runs for a finite amount of time}\}$
L_H is the collection of binary strings representing programs that do **not** loop forever on a given input.
Deciding this language is the same as solving the general Halting Problem.

Halting Problem

As a Language

Definition

Let $< P, I >$ be the encoding (into binary) of a program, P and input for that program I. The split between them must also be encoded.
Let $L = \{< P, I >\}$ be the set of all binary strings that encode a program followed by input for that program.
Then let $L_H = \{< P, I >: P(I)\text{only runs for a finite amount of time}\}$
L_H is the collection of binary strings representing programs that do **not** loop forever on a given input.
Deciding this language is the same as solving the general Halting Problem.
The Halting Problem is the problem of constructing a program, H, that takes two parameters: P, another computer program and I, input for P. H should report "halts" or "loops forever" depending on whether or not P halts on input I.

$$H(P, I) = \begin{cases} \text{"halts"} & \text{if } P(I) \text{ halts} \end{cases}$$

Programs as Data

A program is encoded in some way (Fortran, pseudo-code, Java, bytecode). An encoding can be expressed as a sequence of symbols across some alphabet.

Any alphabet can be re-encoded using strings of bits.

Any program can be expressed as a sequence of bits.

A program, encoded as a sequence of bits, can be given as input to a program. The receiving program may or may not respect the “right” interpretation.

Programs as Data

A program is encoded in some way (Fortran, pseudo-code, Java, bytecode). An encoding can be expressed as a sequence of symbols across some alphabet.

Any alphabet can be re-encoded using strings of bits.

Any program can be expressed as a sequence of bits.

A program, encoded as a sequence of bits, can be given as input to a program. The receiving program may or may not respect the "right" interpretation.

Any program that takes a single input parameter can be passed itself (or rather, its own encoding) as its input.

Halting Problem

Definition

$$H(P, I) = \begin{cases} \text{"halt"} & \text{if } P(I) \text{ halts} \\ \text{"loop"} & \text{if } P(I) \text{ does not halt} \end{cases}$$

Halting Problem

Definition

$$H(P, I) = \begin{cases} \text{"halt"} & \text{if } P(I) \text{ halts} \\ \text{"loop"} & \text{if } P(I) \text{ does not halt} \end{cases}$$

Definition

FSOC Assume H exists. Construct D

$$D(P) = \begin{cases} \text{loop forever} & \text{if } H(D, P) \text{ halts} \\ \text{return} & \text{if } H(D, P) \text{ does not halt} \end{cases}$$

Halting Problem

Definition

$$H(P, I) = \begin{cases} \text{"halt"} & \text{if } P(I) \text{ halts} \\ \text{"loop"} & \text{if } P(I) \text{ does not halt} \end{cases}$$

Definition

FSOC Assume H exists. Construct D

$$D(P) = \begin{cases} \text{loop forever} & \text{if } H(D, P) \text{ halts} \\ \text{return} & \text{if } H(D, P) \text{ does not halt} \end{cases}$$

What does $H(D, D)$ return?

Languages

alphabet A finite set of **symbols**; *e.g.* the *binary alphabet* is $\{0, 1\}$.

string A sequence of zero or more symbols from an alphabet; *e.g.* λ (the empty string), 01011100010, 0, 101
Σ is used to represent the alphabet as a whole. λ or ε stand for the empty string.

language A set of **strings**; *e.g.* $\{w | w \text{ starts with } 1\}$, $\{00, 01, 10, 11\}$.

Σ^* The star (Kleene's star operator) means zero or more copies of the symbol before the star. This is short hand for the set of **all** the strings across the alphabet Σ.
Note: that means Σ^* is a *set*.

Cardinality of $\{0, 1\}^*$

$\{0, 1\}^*$ is infinite. Consider the set of just strings containing only '1' symbols. The lengths of different strings in this language range across non-negative integers. That is infinite.

Is Σ^* **countable**?

If so, how to prove it.

Cardinality of $\{0,1\}^*$

$\{0,1\}^*$ is infinite. Consider the set of just strings containing only '1' symbols. The lengths of different strings in this language range across non-negative integers. That is infinite.

Is Σ^* **countable**?

If so, how to prove it.

Find bijection $f\colon \Sigma^* \to \mathbb{Z}^+$.

$f: \mathbb{Z}^+ \to \{0,1\}^*$

Define the *length-then-value* ordering for binary strings: w_1 comes before w_2 if $|w_1| < |w_2|$ or $|w_1| = |w_2| \wedge$ the unsigned number represented by w_1 is less than the number represented by w_2.
So, Σ^* can be put in order by the above ordering:

$$\{\lambda, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, 110, 111, ...\}$$

$\forall z \in \mathbb{Z}$ let $s = \lfloor \log_2(z) \rfloor$ and $p = 2^s$. Then $f(z) = z - p$ written as a binary number s characters long.

$f: \mathbb{Z}^+ \to \{0,1\}^*$

$\forall z \in \mathbb{Z}$ let $s = \lfloor \log_2(z) \rfloor$ and $p = 2^s$. Then $f(z) = z - p$ written as a binary number s characters long.

z	s	$z-p$	$f(z)$
1	0	0	"" $= \lambda$
2	1	0	"0"
3	1	1	"1"
4	2	0	"00"
5	2	1	"01"
6	2	2	"10"
7	2	3	"11"
8	3	0	"000"
9	3	1	"001"
10	3	2	"010"
11	3	3	"011"

$f : \mathbb{Z}^+ \to \{0,1\}^*$

$\forall z \in \mathbb{Z}$ let $s = \lfloor \log_2(z) \rfloor$ and $p = 2^s$. Then $f(z) = z - p$ written as a binary number s characters long.

$f(23) =$

$f : \mathbb{Z}^+ \to \{0, 1\}^*$

$\forall z \in \mathbb{Z}$ let $s = \lfloor \log_2(z) \rfloor$ and $p = 2^s$. Then $f(z) = z - p$ written as a binary number s characters long.

$f(23) =$0111

$f(32) =$

$f : \mathbb{Z}^+ \to \{0,1\}^*$

$\forall z \in \mathbb{Z}$ let $s = \lfloor \log_2(z) \rfloor$ and $p = 2^s$. Then $f(z) = z - p$ written as a binary number s characters long.

$f(23) =$0111

$f(32) =$00000

$f^{-1}(\lambda) =$

$f : \mathbb{Z}^+ \to \{0,1\}^*$

$\forall z \in \mathbb{Z}$ let $s = \lfloor \log_2(z) \rfloor$ and $p = 2^s$. Then $f(z) = z - p$ written as a binary number s characters long.

$f(23) =$0111
$f(32) =$00000
$f^{-1}(\lambda) =$0
$f^{-1}(1000) =$

$f : \mathbb{Z}^+ \to \{0, 1\}^*$

$\forall z \in \mathbb{Z}$ let $s = \lfloor \log_2(z) \rfloor$ and $p = 2^s$. Then $f(z) = z - p$ written as a binary number s characters long.

$f(23) =$0111
$f(32) =$00000
$f^{-1}(\lambda) =$0
$f^{-1}(1000) =$24
$f^{-1}(00010) =$

$f : \mathbb{Z}^+ \to \{0, 1\}^*$

$\forall z \in \mathbb{Z}$ let $s = \lfloor \log_2(z) \rfloor$ and $p = 2^s$. Then $f(z) = z - p$ written as a binary number s characters long.

$f(23) =$0111
$f(32) =$00000
$f^{-1}(\lambda) =$0
$f^{-1}(1000) =$24
$f^{-1}(00010) =$34

Counting Deciders

- A *decider* is an *algorithm*.
- An *algorithm* can be expressed in `Java` (or another programming language).
- Any `Java` program can be encoded as a sequence of *characters* that are, in turn, encoded as sequences (strings) of bits.
- Any `Java` program can be represented by a *bit string*.

Counting Deciders

$\{\text{deciders}\} \subset \{\text{Java programs}\} \subset \{0,1\}^*$
Remember: $A \subset B \rightarrow |A| \leq |B|$. This will be important.

An Uncountable Infinity

A Mathematician's View

- $\mathbb{Z}$ in *countably* infinite. ($\mathbb{Z}$ is often used as the canonical countably infinite set.)
- $|\mathbb{R}| > |\mathbb{Z}|$ There are **more** real numbers than there are integers.
- Proof: By contradiction. Assume they are the same size; show that the resulting *bijective* function between them cannot map *onto* $\mathbb{R}$.

An Uncountable Infinity

A Computer Scientist's View

- $\{0,1\}^*$ in *countably* infinite. (The language of all *binary strings* is countably infinite.)
- $|\mathbb{P}(\{0,1\}^*)| > |\{0,1\}^*|$ There are **more** *binary languages* than there are *binary strings*.
- Proof: By contradiction. Assume they are the same size; show that the resulting *bijective* function between them cannot map *onto* $\mathbb{P}(\{0,1\}^*)$.

There is more!

$|\{\text{deciders}\}| \leq |\{\text{Java programs}\}| \leq |\{0,1\}^*|$

There are **more** binary languages than there are *deciders* for languages: there **must be** undecidable languages.

Cardinality of $\mathbb{P}(()\Sigma^*)$

Review of Terms

- $\Sigma = \{0, 1\}$ = the *binary* alphabet
- $\Sigma^* =$

Cardinality of $\mathbb{P}(()\Sigma^*)$

Review of Terms

- $\Sigma = \{0, 1\}$ = the *binary* alphabet
- $\Sigma^* = \{$ all *binary* strings$\}$
  $\{\epsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, 110, 111, ...\}$
- $\mathbb{P}(()\Sigma^*)$ = Set of all *subsets* of Σ^*

Cardinality of $\mathbb{P}(()\Sigma^*)$

Review of Terms

- $\Sigma = \{0, 1\}$ = the *binary* alphabet
- $\Sigma^* = \{$ all *binary* strings$\}$
  $\{\epsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, 110, 111, ...\}$
- $\mathbb{P}(()\Sigma^*)$ = Set of all *subsets* of Σ^*
  $\{$ all *binary* **languages**$\}$

Cardinality of $\mathbb{P}(()\Sigma^*)$

$\mathbb{P}(()\Sigma^*)$ is *uncountable*.
$|\mathbb{P}(()\Sigma^*)| > |\mathbb{Z}^+|$
$|\mathbb{P}(()\Sigma^*)| > \aleph_0$

$\mathbb{P}(()\Sigma^*$ is Uncountable

TBP: $\mathbb{P}(()\Sigma^*)$ is *uncountable*.

TBP:	$\lvert\mathbb{P}(()\Sigma^*)\rvert \neq \lvert\mathbb{Z}^+\rvert$	Countably Infinite
FSOC:	$\lvert\mathbb{P}(()\Sigma^*)\rvert = \lvert\mathbb{Z}^+\rvert$	
1.	bijection $\exists\, f : \mathbb{Z}^+ \to \mathbb{P}(()\Sigma^*)$	Same cardinality
2.	f can be represented as a table	

$\square$

Looking at f

	ϵ	0	1	00	01	10	11	000	001	...
1	0	1	0	1	0	1	0	1	0	...
2	0	1	0	1	1	1	0	0	0	...
3	0	0	1	0	0	0	0	0	0	...
4	1	0	1	0	1	0	0	0	1	...
5	1	0	0	0	1	0	0	0	1	...
$\vdots$										

Each row represents a *subset* of Σ^* or an element of $\mathbb{P}(()\Sigma^*)$. A sequence of Boolean values whether the string atop the column is/is not in the language in that row.

Looking at f

	ϵ	0	1	00	01	10	11	000	001	...
1	0	1	0	1	0	1	0	1	0	...
2	0	1	0	1	1	1	0	0	0	...
3	0	0	1	0	0	0	0	0	0	...
4	1	0	1	0	1	0	0	0	1	...
5	1	0	0	0	1	0	0	0	1	...
$\vdots$										

Let $f(i) = f(i)_1 f(i)_2 f(i)_3 f(i)_4 f(i)_5 \ldots$

Looking at f

	ϵ	0	1	00	01	10	11	000	001	...
1	0	1	0	1	0	1	0	1	0	...
2	0	1	0	1	1	1	0	0	0	...
3	0	0	1	0	0	0	0	0	0	...
4	1	0	1	0	1	0	0	0	1	...
5	1	0	0	0	1	0	0	0	1	...
$\vdots$										

Let $f(i) = f(i)_1 f(i)_2 f(i)_3 f(i)_4 f(i)_5 \dots$

Let d, the diagonal language, be $f(1)_1 f(2)_2 f(3)_3 f(4)_4 f(5)_5 \dots = 01101\dots$

Looking at f

	ϵ	0	1	00	01	10	11	000	001	...
1	0	1	0	1	0	1	0	1	0	...
2	0	1	0	1	1	1	0	0	0	...
3	0	0	1	0	0	0	0	0	0	...
4	1	0	1	0	1	0	0	0	1	...
5	1	0	0	0	1	0	0	0	1	...
$\vdots$										

Let $f(i) = f(i)_1 f(i)_2 f(i)_3 f(i)_4 f(i)_5 \ldots$

Let d, the diagonal language, be $f(1)_1 f(2)_2 f(3)_3 f(4)_4 f(5)_5 \ldots = 01101\ldots$

Let $\overline{d}$, the *compliment* of the diagonal language be $\overline{f(1)_1}\,\overline{f(2)_2}\,\overline{f(3)_3}\,\overline{f(4)_4}\,\overline{f(5)_5} \ldots = 10010\ldots$

Looking at f

	ϵ	0	1	00	01	10	11	000	001	...
1	0	1	0	1	0	1	0	1	0	...
2	0	1	0	1	1	1	0	0	0	...
3	0	0	1	0	0	0	0	0	0	...
4	1	0	1	0	1	0	0	0	1	...
5	1	0	0	0	1	0	0	0	1	...
$\vdots$										

Let $f(i) = f(i)_1 f(i)_2 f(i)_3 f(i)_4 f(i)_5 \ldots$
Let d, the diagonal language, be $f(1)_1 f(2)_2 f(3)_3 f(4)_4 f(5)_5 \ldots = 01101\ldots$
Let $\overline{d}$, the *compliment* of the diagonal language be
$\overline{f(1)_1}\,\overline{f(2)_2}\,\overline{f(3)_3}\,\overline{f(4)_4}\,\overline{f(5)_5} \ldots = 10010\ldots$
$\overline{d} \notin range(f)$.

Looking at f

TBP: $\overline{d} \notin range(f)$.

$$
\begin{array}{lll}
\text{FSOC:} & \overline{d} \in range(f). & \\
1. & \exists z \in \mathbb{Z}^+ \ni f(z) = \overline{d} & \text{Definition of range} \\
2. & \overline{d}_z = f(z)_z = b & \overline{d_z} \text{ from the table} \\
3. & \overline{d}_z = \overline{f(z)_z} = \overline{b} & \text{by construction} \\
\Rightarrow\Leftarrow & \overline{d}_z = \overline{b} \wedge \overline{d}_z = b & \text{Combine 2, 3} \\
\therefore & f(z) \neq \overline{d} & \\
\therefore & \overline{d} \notin range(f) &
\end{array}
$$

$\square$

f is **not** surjective; f is not bijective

$\mathbb{P}(()\{0,1\}^*)$ is Uncountable

TBP: $\mathbb{P}(()\Sigma^*)$ is *uncountable*.

TBP:	$\lvert\mathbb{P}(()\Sigma^*)\rvert \neq \lvert\mathbb{Z}^+\rvert$	Countably Infinite
FSOC:	$\lvert\mathbb{P}(()\Sigma^*)\rvert = \lvert\mathbb{Z}^+\rvert$	
1.	bijection $\exists\, f : \mathbb{Z}^+ \to \mathbb{P}(()\Sigma^*)$	Same cardinality
2.	*f* can be represented as a table	
3.	The *f* in the table is not **onto**.	
$\Rightarrow\Leftarrow$	*f* both is and is not onto	1. and 3.
$\therefore$	$\lvert\mathbb{P}(()\Sigma^*)\rvert \neq \lvert\mathbb{Z}^+\rvert$	

$\square$