## **Janna**

## **Problem (Language):**

A problem Q is a binary relation on a set I of instances and a set S of solutions Q ⊆ I x S

* every problem can be expressed as deciding whether an object is a member of a set.

[w ∈ L]?

## **There are three types of problems:**

### **1- General problem:**

A general problem G is a total relation

G ⊆ I x S

One or more solutions can be produced for each problem instance. It also called search problem or optimal solution problem.

It can be expressed as deciding whether an object is a member of a set

[(i, s1, s2,...., sn) ∈ {(i,s1,s2,...., sn)}:∀i,i∊I Ǝsn,sn ∈ S(i, sn) ∈ G}]?

* inG = lambada i,ss: all((i,s) in G for s in ss)
* PS: one input can have several outputs
* Also called optimization problem

### **2- Function problem:**

A function problem F is a function F ⊆ I x S

F: I 🡪 S

One solution is produced for each problem instance. It also called computation problem or optimal value problem.

It can be expressed as deciding whether an abject, (i,s) where i ∈ I, s ∈ Sis a member of the set F

[(i,s) ∈ F]?

### **3- Decision Problem**

A decision problem Q is a function Q ⊆ I x {no, yes}

Q: I 🡪 {no, yes}

It can be expressed as deciding whether an object is a member of a set

Y = {i ∈ I: Q[i] == yes}

[i∈Y]?

💁🏻‍♂️ Since every problem can be expressed as deciding whether an object is a member of a set, every problem can be casted as a decision problem.

### **Encoding:**

An encoding of a set A of abstract objects is a mapping 'e' from A to a set L of strings.

e: A 🡪 L

### **Semantics Encoding:**

An encoding of a language A into language B is a mapping of all terms of A into B

e: A 🡪 B

### **Language:**

set of words

### **Alphabet:**

An alphabet is a finite nonempty set of symbols (characters).

Binary alphabet

𝛴 = {0,1}

Lower-case English letters

𝛴 = {1,b,c,...,z}

### **String**

A string (word\_ is a finite sequence of symbols chosen from the some alphabet.

"101"

The length of a string is the number of positions for symbols in the string, |w|.

|"101"|

3

The empty string, ε, is the string with zero occurrences of symbols.

|''|

0

* Zk = {string of length k: each of whose symbols is in 𝛴}
* 𝛴3 = {000, 001, 010, 011, 100, 101, 110, 111}
* where 𝛴 = {0,1}
* 𝛴0 = {ε}

The set of all strings over an alphabet 𝛴 is denoted as 𝛴\*

𝛴\* = 𝛴0 ⋃ 𝛴1 ⋃ 𝛴2 ⋃....

𝛴+ = 𝛴1 ⋃ 𝛴2 ⋃ 𝛴3 ⋃....

### **Language:**

#### **A language is a set of strings, all of which are chosen from 𝛴\***

L is a language over 𝛴

L ⊆ 𝛴\*

𝛴 is a set of alphabet 𝛴\* is a set of all possible combination of the alphabet (Some of them could be empty or one character)

L is a set of 𝛴\*

### **Encoding a decision Problem:**

A decision problem can be encoded in a language as a function from the set 𝛴\* to the set {0,1}

Q: 𝛴\* 🡪 {0,1}

0: no

1: yes

The decision problem Q is entirely characterized by those problem instances that produce a 1 (yes) answer

L = {i ∈ 𝛴\* : Q[i] == 1}

Assign semantics to L, such as a set of prime numbers, even integers, logical expressions, or graphs.

The decision problem Q is: Given a string W ∈ 𝛴\*, decide whether or not w ∈ L

For w ∈ 𝛴\* [w ∈ L]?

### **The problem Q can be viewed as the language L**

## **Computational Model:**

### **Turing Machine:**

A Turing machine is a 7-tuple,

(Q, 𝛴, 𝝘, 𝞭, q0, qaccept, qreject)

Q: is the set of states \*(a state is like a picture -snap shoot- everything is changing over time. Here we take a snap shoot at a step level)

𝛴: is the input alphabet not containing the special blank symbol\_ \*(blank symbol is also a character)

𝝘: (Tao) is the alphabet, where \_ ∈ T and 𝛴 is ⊆ of T

𝞭: (delta) is the transition function

𝞭: Q x 𝝘 🡪 Q x 𝝘 x {L, R}

\*(delta is a function mapping Q x 𝝘 to Q x 𝝘 x {L:left, R:right})

q0 ∈ Q is the start state

qaccept ∈ Q is the accept halt state

qreject: ∈ Q is the reject halt state, where qreject != qaccept

Function 𝞭 is the program of the machine.

For a k multitape Turing machine,

𝞭: Q x 𝝘k 🡪 Q x 𝝘k x {L, R}k

For a non-deterministic Turing machine (non-deterministic: means the delta function maps to the powerset Q x 𝝘 x {L, R}) -- Not specified

𝞭: Q x 𝝘 🡪 P[Q x 𝝘 x {L, R}}

### **Algorithm**

🧲 Any algorithm can be coded in terms of a Turing machine.

🧲 Any Turing machine can be described by an algorithm.

Don't forget the graph (screen shot) Writhe the Turing machine equation. in the exam

### **Universal Turing Machine:**

A Universal Turning machine can simulate any Turing machine form the description of the machine.

Let U be three-tape univerival Turing machine

U = (Q, 𝛴, 𝝘, 𝞭, q0, qaccept, qrejects)

𝞭: Q x 𝝘3 🡪 Q x 𝝘3 x {L,R}3

Let M be a one-tape Turing machine to be simulated by U

M = (Q', 𝛴', 𝝘', 𝞭', q0', q'accept,qreject')

𝞭': Q' x 𝝘' 🡪 q' x 𝝘' x {L',R'}

where 𝝘' ⊆ 𝝘

The definition of 𝞭' is stared on tape p as program code. Initially head p is on the leftmost non-blank.

The q0' is stared on tape q. initially head q is on the leftmost non- blank.

The input string of M is stored on tape t. Initially head t is on the left most input symbol.

### **The 𝞭 of U is defined as:**

#### **Repeat unit halt:**

* the following macrostep of U to simulate one state transition of M
* Instruction format for each state transition M:d(pa,in)=(ns,out,move)

1. read present state and input of M

* head q moves to the leftmost non-blank to read present state of M.
* If the present state is qaccept' or q'reject
* then U goes to the corresponding qaccept or q reject
* head t reads the present input of M

1. Search program for the present state and input

* head p moves to leftmost non-blank and search right the pair of present state and input of M until found.

1. Copy next state

* head p moves right to read the next state.
* head q moves back to leftmost no-bank and write the next state.

1. Copy output

* head p moves right to read the present output.
* head t writes the present output.

1. move head as specified

* head p moves right to read the movement
* head t moves as specified by the movement

### **Computer:**

A computer can be simulated on a Universal Turing machine. A Universal Turing machine can be programmed to run on a computer.

### **Most powerful model:**

Turing machine (and equivalent formalisms) is the most powerful formalism to model and mechanical calculus.

### **Beyond Turing Machine:**

Exploring theoretical computational models beyond the limit of Turing machine and the possibility for hypercomputer.

See S:pp.125-147 MG;pp.151-162

**Computability Theory**

**Solvable Problems**

A problem is decidable if it has either a **yes** or a **no** answer

computable, Decidable (or recursive), Semidecidable (recursively enumerable, or acceptable)

**Computable function**

A function f:𝛴\* ⇒ 𝛴\* is computable if there exist Turing machine, on every input *w*, halts with just f(w) on its tape.

The class of computable functions is equivalent to the class of functions defined by

* Recursive function, lambda calculus, or Markov algorithms.

**Recursive functions** are precisely the function that can be computed by Turing machines

**Membership Question**

[x ∈ S]?

The characteristic function 𝑐𝑠: 𝛴\* ⟹ {0,1} of a set *S*, is defined by

𝑐𝑠 (x):= 1 if x ∈ *S* else 0

A set S is **decidable** (or recursive) ⇔ its characteristic function 𝑐𝑠 is computable

A set S is **semidecidable** (or recursive enumerable) ⇔ S = ⦰ or it is the image of a total and computable function gs, that is

S = {y: ∀x ∈ 𝛴\* (y == gs(x))}

When a problem P is represented as a set *Sp*, then

(P is **solvable**) if (*Sp* is recursive)

(P is **partially solvable**) if (*Sp* is recursively enumerable)

**Decidable Languages:**

∀𝑥∈Σ∗[𝑥∈𝐿] ∀x ∈ Σ∗ [x ∈ L]

Let L be a regular language and 𝑥∈Σ\*, it is decidable whether x is a member of L

* Regular language is decidable.
* Context free language is decidable.
* Language accepted by pushdown automata is decidable.

**Enumeration**

An enumeration of a set S of abstract objects is a mapping g from S to the set natural number N.

A **godel numbering** is a function g form a countable se *S* to N

With both g and the inverse of g a computable function

g: S ⟹ N

The set g: {TM𝛴} of all the Turing Machines over a given alphabet 𝛴 can be enumerated algorithmically. That is, an algorithmic bijection g: {TM𝛴} ⟹ N can always be stated

g: {TM𝛴} ⟹ N

g: g is bijective

a and g-1 are computable

The number of *all* possible TM based on Z is:

|{TM𝛴}| = N0

* All algorithmically computable function can be algorithmically enumerated.
* All algorithmically recognizable language can be algorithmically enumerated.

Turing Machines can be referred as devices computing function form N ⟹ N

**All computable function f: N ⟹ N can be enumerated**

**Rice's Theorem (solvable case)**

Let *F* be any set of computable functions.

The set *S* = {*x*: *fx* F} is decidable ⇔ *F* = ⦰ or *F* is the set of all computable functions.

**Only trivial property of computable functions are decidable.**

EG: all function for one binary variable

EG2: all functions for two binary variables

### **Unsolvable Problems:**

Algorithmic problem **undolvability may** occur only in the case where **infinitely** many cases (arguments of a function, strings of a language) are to be considered.

* e.g. One machine may not predicts the behavior of *any* other machines.

When a problem *P* is represented as a function *f*, then:

*P* is unsolvable if *f* is nonrecursive

When a problem *P* is represented as a set *Sp* then:

*P* is unsolvable if *Sp* in nonrecursively enumerable

### **Uncomputable function**

A vast majority of function on N cannot be computed.

Since all computable function *f*: N ⇒ N can be enumerated, the cardinality of the class of **all computable** functions is at most

*N*0

However, the cardinality of the class *P* of **all definable function** on N, where *P* = {*f*:(*f*: N⇒N)} is

2*N*0

### **Non-recursively enumerable set**

A set S is **not recursively enumerable** if

((i *S*) ⟹ (*fi* is total)) ∧ (∀*f, f* is total and computable ∃*i* ∈ *S* (*f* == *fi*))

* Total computable function are not recursively enumerable

(*S*tot = {*i*: *fi* is total}) is not recursively enumerable

### **Rice's Theorem (non-solvable case)**

Let F be any set of computable function.

The set *S* = {*x*: *fx* ∈ *F*} is **undecidable** ⇔ ((F != ⦰) ∧ (F != {the set of all computable functions}))

* For any non-trivial property of partial function, the question of whether a given algorithm computes a partial function with this property is undecidable.

### **Halting problem is undecidable**

The *halting* problem is a decision problem for determining, when given a description of a program and its initial input, whether **any** program executing on the input will eventually *halt*.

**Halting problem is undecidable**

Proof sketch: Proof by contradiction. Assume, to the contrary, that there exit a program halt(progStr, inStr), which takes any program encoded as progStr and any input string *inStr*, is able to decide whether or not the program will halt when given the input string.

halt(progStr, inStr) ::= True if prog(inStr) is halt else False

trouble(xSr ::= Ture if halt(xStr, xStr) == False else loopForever

tStr ::= encoding(trouble program)

Dose trouble(tStr) halts?

Assume that trouble(tStr) halts:

Then, halt(tStr, tStr)==False,. That is, tStr program given tStr (trouble(tStr)) does not halts, resulting in a contradiction.

Assume that trouble(sStr) does not halts:

Then, trouble(tStr) program must reach loopForever. Then, halt(tStr, tStr) != False, that is, halt(tSt, tStr) is true. Then, tStr program given tStr (trouble(tStr)) halts, resulting in a contradiction.

Therefore, there does not exist such a halt(progStr, inStr) program.

HaltTM = {<M,w>: (M is a Turing Machine) and (M halts on input w)}

HaltTM is undecidable

### **Undecidable Languages**

ATM = {<M,w>:(M is a Turning Machine and (M accepts w)}

ATM is undecidable

EQ*TM* = {<*M1,M2*> : (*M1* and *M*2 are TMs) and L(*M*1) == L(*M*2)}

EQTM is undecidable

Turing machine M is **minimal** if there is not Turing machine equivalent to M that has a shorter description.

The length of the description M of a Turing machine M is the number of symbols in the string describing M.

MinTM = {<M>: is a minimal Tring machine}

MinTM is not *Turing recognizable*

### **Super-Turing computation**

Super-Tring computation is any form of computation that cannot be performed by a Turing machine.

**No physical examples of super-Turing computers are currently known.**

### **See**

W, computability theory, hypercomputation, digital physics, Turing completeness, Super-Turing computation

S, 206, 217-231

MG, 147-191

### **Complexity Theory**

Each problem has inherent complexity.

There is some minimum amount of work required to solve it.

### **Reducibility Ordering**

Reducibility expresses that a set A of problems is less complex than a set B of problems, by transforming each instance of A into a instance of B.

A B

A set A is reducible under F to a set B, given A and B be subsets N, and a set of functions F: N⇒N which is closed under composition, if and only there exists a *f* ∈ F such that for all *x* ∈ N, *x* ∈ A if and only if f(*x*) ∈ B

(A F B) ⇔ (∃*f*∈F ∀*x*∈N((*x* ∈ A) ⇔ (*f*(*x*) ∈ *B*)))

If F is a set of polynomial function, then the reduction is called polynomial reduction.

To define reduction in term of language, the domain 𝛴\* is used instead of N, and F is a set of computable functions.

A set *S* ⊆ *P*(N) is closed under reduction

(S is closed under ) ⇔ (∀*A∈P*(N)∀*s*∈*S*((*A*⇐ *s*) ⇔ (*A* ∈ *S*))

A ⊆ H of N is hard for *S* ⊆ *P*(N)

(H is hard for *S*) ⇔ (∀s∈S(*s* H))

A subset *C* of N is complete for *S* ⊆ *P*(N) if *C* is hard for *S* and *C* is in *S*

(*C* is complete for *S*) ⇔ ((∀*s*∈*S*(*s* *C*) ∧ (*C* ∈ *S*))

The set *C* is the **most complex** (maximal) problem in the class *S*

Any solution to C can, in combination with the reductions, be used to solve every problem in the class S.

**Properties of reduction**

The reduction relation is Reflexive and Transitive

(A B) 🡪 (( ))

(A B) ∧ (B *is recursive*)) 🡪 (*A is recursive*)

(A B) ∧ (B *is recursively enumerable*)) 🡪 (A *is recursively enumerable*)

**Complexity degree**

A set A of problem has the **same complexity** under reduction r as a set B of problems if and only if A is reducible to B and B is reducible to A

(*A* =r *B*) ⇔ ((*A* r *B*) ∧ (*B* r *A*))

A complexity degree is an equivalence class under =r

**Descriptive Ordering**

Descriptive complexity seeks to order problem complexity by the type of logic needed to express the problem.

* Frist-order logic:

FO == *AC*0

the languages recognized by polynomial-sizecircuits of bounded depth,

which equals the languages recognized by concurrent random access machine in constant time.

* First-order logic added with a commutative, transitive closure operator:

FO(CTS) == L

Which equals SL problems solvable in logarithmic space.

* First-order logic with a transitive closure operator:

FO(TC) == NL

The problem solvable in nondeterministic logarithmic space

* First-order logic extended by the ability to define new relations by induction formalized by Least fixed point operator:

FO(LFP) == P

The problem solvable in deterministic polynomial time

* Existential second-order logic (excluding universal second-order quantification over relation, functions, and subsets):

SO∃ == NP

The problem solvable in nondeterministic polynomial time

* Universal second-order (excluding existential second-order quantification):

SO∀ == co-NP

The complement of the class NP

* Second-order logic:

SO == PH

The union of all complexity classes in the polynomial hierarchy

* Second-order logic with a transitive closure (commutative or not):

SO(TC) == PSPACE

The problem solvable in polynomial space

* Second-order logic extended by the ability to define new relations by induction formalized by Least Fixed Point Operator:

SO(LFP) == EXPTIME

The problems solvable in exponential time

**Resource Bound Ordering**

Resource bound ordering classifies problems in terms of computational resources required to solve the problems.

Determine a function *f* mapping g input sized to resources,

*r* = *f*(*n*)

Usually, decision problem are analyzed and the resources required, such as running time and space used, may depends on algorithms and machine models.

**Complexity measure**

Measure relative complexity of computable function, without referring to concrete computational model.

**Complexity measure** assigns an appropriate resource bound *fe* to each computable function Ae

It does not specify the machine model or the resource but define the constraints with (Blum) axioms:

(A1) for each *e*, *fe* is defined on precisely the same domain as *Ae*

(A2) it is decidable form *e*, *n* and *r*, whether *fe*(*n*) == *r*.

**Complexity classes** of computable function can be define in terms of a total computable function g.

*C*(g) := {Ae : *n*f*e*(*n*) *g*(*n*)}

*C*(*g*) is the set of all computable functions with a complexity less than or equal to g.

*C*0 (*g*) := {*h* ∈ *C*(*g*): codom(*h*) {0,1}}

*C*0(*g*) is the set of all Boolean-valued functions with complexity less than *g*.

Using them as characteristic function on sets, *C0(g)* can be considered of as a complexity class of sets.

Ordering functions

*f* *g*

f grows no faster than g

where f and g be function from nonnegative integers to positive real numbers, and for some real constant c > 0, and some nonnegative integer constant *n*0, define

*f* ∈ *O*(*g*), (*f* *g*) f is upper bounded by g

*f* ∈ Ω(*g*), (*f* *g*) f is lower bounded by g

*f* ∈ Θ(*g*), (*f* = *g*) f is order g

*O*(*g*) := {*f* : ∀*n*>=*n0*(*f*(*n*) (*c* \* *g*(*n*)))}

*o*(*g*) := {*f* : ∀*n*>=*n0*(*f*(*n*) < (*c* \* *g*(*n*)))}

Ω (*g*) := {*f* : ∀*n*>=*n0*(*f*(*n*) (*c* \* *g*(*n*)))}

(*g*) := {*f* : ∀*n*>=*n0*(*f*(*n*) > (*c* \* *g*(*n*)))}

*Θ(g) := O(g) Ω(g)*

Each set (theta)(g) is an equivalence class of functions, a complexity class.

Ordering of types of functions:

log(*n*) < *n* < *nK* < *kn*

**Time complexity**

* Number of steps required

The time complexity, running, of a deterministic Turing machine M that halts on all inputs, is the function *f*: N 🡪 N, where *f*(*n*) is the maximum number of steps that *M* uses on any input of length n.

M runs in time *f*(*n*). M is an *f*(*n*) time Turing machine.

Time complexity class, *TIME*(t(*n*)), is the collection of all languages that are decidable by an *O*(*t*(*n*)) time Truing machine, where t: N 🡪 +

P is the class of languages that are decidable in polynomial time on a deterministic single-tape Turing machine.

*P* =  TIME(*nk*)

The running time of nondeterministic Turing machine N, that is a decider, is the function f: N🡪 N, where *f*(*n*) is the maximum number of steps that N uses on any branch of its computation on any input of length n.

*NTIME*(*t*(*n*)) := {L:L is a language decided by a *O*(*t*(*n*)) time nondeterministic Turing machine}

NP is the class of languages that have polynomial time verifiers.

*NP* =  NTIME(*nk*)

A verifier for a language L is an algorithm V, where L = {*w*: Vaccepts < *w,e* > for some string *c*}

A polynomial time verifier runs in polynomial time in the length of w. A language A is polynomial verifiable if it has a polynomial time verifier.

A language B is NP-complete if (B ∈ NP) and (every A ∈ NP is polynomial time reducible to B)

(*B* is *NP*complete) ⇔ ((*B* ∈ *NP*) ∧ (∀A ∈ *NP* (*A* *p* *B*)))

Proving NP-complete of a problem Q, choose a known NP-complete problem P, reduce P to Q

*P* *r* *Q*

CIRCIT-SAT is NP-complete:

CIRCIT-SAT := {<C>: C is a satisfiable Boolean combinational circuit}

Size(Boolean combinational circuit) := number of logic gates + number of wires

**Proof outline:**

* Proving CIRCIT-SAT ∈ NP

Construction a two-input polynomial-time algorithm A, that can verify CIRCIT-SAT

A(encoding of a circuit, certification of an assignments for each wires)

* FOR each logic gate, check its output match the value provided by the certification.
* IF entire circuit output 1, return 1, ELSE return 0 Algorithm A runs in polynomial time.
* Proving every L ∈ NP is polynomial time reducible to CIRCIT-SAT

Constructing a polynomial-time reduction algorithm F, computing a reduction function f,

that maps every binary string w to a circuit *C* = *f*(*w*), such that (*w* ∈ *L*) ⇔ (*C* ∈ *CIRCIT-SAT*)

Since *L* ∈ *NP*, there must exist an algorithm A that verifies L in polynomial tim. A(w,x)=1 iff w ∈ L

* F produces a circuit C = *f*(*w*) that is satisfiable iff there exists a certificate x such that A(w,x)=1
* F uses A(w,x) to produce the circuit C, that takes input c, and gives 0 or 1 as output. Considering algorithm A runs on a computer, each step produces one configuration of all memory.

A step from one configuration to another is done by a copy of combinational circuit.

Now take away all memory, simply link all copy of the combinational circuits to form a complete circuit, that is C.

* Showing F is correct: Assume A(w,x)=1, then apply x to input C, which will output 1. Assume C is satisfiable, there exist input x such that C output 1, then A(w,x)=1
* Showing F runs in time polynomial in n =|w| First, the number of bits required to represent a configuration is polynomial in n. Since algorithm A is polynomial time, there are polynomial copies of combinational circuits. The construction of C form w can be accomplished in polynomial time by the reduction algorithm F.

QED

**Space Complexity**

Number of memory cells required

The **space complexity** of a deterministic Turing machine M that halts on all inputs is the function *f*: N 🡪 N, where f(n) is the number of tape cells that M scans on any input length n.

* M runs in space f(n).

The **space complexity** of a nondeterministic Turing machine N were in all branches halt on all inputs is the function f: N 🡪 N, where f(n) is the maximum number of tape cells that N scans on any branch of its computation for any input length n.

The space complexity classes, where *f* : N 🡪 R+

SPACE(f(n)) := {L: L is a language decided by an O(f(n)) space deterministic Turing machine}

NSPACE(f(n)) := {L: L is a language decided by an O(f(n)) space nondeterministic Turing machine}

* L is the class of language that are decidable in logarithmic space on a deterministic Turing machine.

*L* = SPACE(log(*n*))

* NL is the class of language that are decidable in logarithmic space on a nondeterministic Turing machine.

NL = NSPACE(log(*n*))

* SPACE is the class of languages that are decidable in polynomial space on a deterministic Turing machine.

PSPACE =  SPACE(*nk*)

A language B is **PSAVE-complete** if (B ∈ PSACE) and (every A ∈ PSACE, (A, *p* B)

**Complexity Hierarchy**

Diagram

Description automatically generated

[**https://people.cs.umass.edu/~immerman/descriptive\_complexity.html**](https://people.cs.umass.edu/~immerman/descriptive_complexity.html)

**Algorithm**

Problem solving using computers: problem, strategy, algorithm (input, output, steps), Analysis (correctness, Time, Space, Optimality), Implementation, verification.

**Math:** A problem P is a binary relation on a set I of instances and a set S of solutions: P IxS

**System:** A problem T is a requirement for transforming a set I of inputs to a set of O outputs: I (\to\_T)

**Spatial:** A problem G is a requirement from a location A to location B: A (\to\_G) B

**Time:** A problem T is a requirement to start from the initial state (S\_0) to reach a final state (S\_f):(S\_0\to\_TS\_f)

**Human:** A problem D is the desire or need form what currently have to reach to what eventually want: H (\to\_D) W

**Algorithm specification**

**Data specification**

Encoding the set I of \pmb{input} input and the set O \pmb{output}

Encoding the set S of \pmb{state} or the intermediate results

\pmb{Data structures form the basis for computer algorithm}

**Step specification**

A step transforms an input to an output

A step can be an operation an instruction, a statement, an ordered-pair, a transition, a function, a subroutine, or an action.

Methods for specifying a step of an algorithm depend on the underline computational model:

Truing Machine:

State diagram, State table; (Transition form one state to another, Move head, Write a symbol)

Digital circuit: Truth table, Boolean equation; (Gate, Memory state change)

Microprocessor: Machine code, Assembly language; (Instruction)

Imperative programming: BASIC, C; (Statement)

Object-oriented programming: C++, java; (object)

Functional programming: LISP; (Function)

Logic programming: Prolog(rule)

Human: outline, pseudocode; (Step)

**An Algorithm:**

An algorithm consists of sequence of steps (operations, instructions, statements, order-pairs, action) for transforming inputs (preconditions) to outputs (postconditions).

Search for an algorithm:

Search Space, Find a path, A\* search, Generalize form the tried examples.

**Complexity classes**

Linear time

Polynomial time

Exponential time

Non-terminate