Fundamentals of CS

WEEK 1

1.101 Intro to propositional logic

A proposition is a statement that can be either true or false. it must be one or the other, and it cannot be both.

eg. : 2 is a prime number

NOT a proposition : X is a prime number.

Syntaxes: propositions are denoted by capital letters, P, Q, ...
general statements are denoted by lowercase letters.
any on a logical arguments used in proofs
called propositional variables

Connectives :

- NOT $\neg$, negation
- OR $\vee$, disjunction
- AND $\wedge$, conjunction
- IF... THEN : $\rightarrow$, conditional / implication

$p \rightarrow q$, p : premise
q : conclusion

IF and only IF $\leftrightarrow$

$p \leftrightarrow q$, bi-conditional

Exclusive OR : XOR . $\oplus$

Operator precedence : $\neg$ $\wedge$ $\vee$ $\rightarrow$ $\leftrightarrow$

1.202 Tautology and consistency

Tautology : a ~~fa~~ formula that is always true

Consistent : a formula that is true at least for one scenario
eg : $p \wedge q$

Not consistent or inconsistent is a formula that is never true
eg : $p \wedge \neg p$

(also called contradiction)

WEEK 2

Propositional Equivalences

Formula A and B are equivalent if they have the same truth tables.

$\equiv$ (not a connective) but a relation

De Morgan's Laws

$\neg(p \wedge q) \equiv \neg p \vee \neg q$

$\neg(p \vee q) \equiv \neg p \wedge \neg q$

Important equivalence:

$(p \to q) \equiv (\neg p \vee q)$

$(\neg p \vee q) \equiv \neg(\neg\neg p \wedge \neg q) \equiv \neg(p \wedge \neg q)$

Contrapositive:

$p \to q \equiv \neg q \to \neg p$

First-order logic

Important notions

Predicates that describe properties of objects

eg: odd(3)

Predicates take arguments and become propositions.

Quantifiers allow us to reason about multiple objects

Existential quantifier, $\exists$

Universal quantifier, $\forall$

All Ps are Qs: $\forall x(P(x) \to Q(x))$

Some Ps are Qs: $\exists x(P(x) \wedge Q(x))$

Quantifiers to connectives

$\exists x, P(x)$ and domain is $D = \{x_1, \dots x_n\}$

means $P(x_1) \vee P(x_2) \vee \cdots \vee P(x_n)$

$\forall x, P(x)$ and domain is $D\{x_1, \dots, x_n\}$

means: $P(x_1) \wedge P(x_2) \wedge \cdots \wedge P(x_n)$

De Morgan's Law:

$\neg \exists x, P(x) \equiv$ ~~$\neg P(x_1) \wedge \neg P(x$~~

$\neg((P(x_1) \vee P(x_2) \cdots \vee P(x_n))$

$\equiv \neg P(x_1) \wedge \neg P(x_2) \wedge \cdots \wedge \neg P(x_n)$

$\equiv \forall x, \neg P(x)$

$\neg \forall x, P(x) \equiv \neg P(x_1) \vee \neg P(x_2) \vee \cdots \vee \neg P(x_n)$

$\equiv \exists x, \neg P(x)$

$\forall x(P(x) \to q(x))$

$\neg \forall x(P(x) \to q(x))$

$\equiv \exists x\ \neg[P(x) \to q(x)]$

$\equiv \exists x\ \neg(\neg P(x) \vee q(x))$

$\equiv \exists x(\neg\neg p(x) \wedge \neg q(x))$

WEEK 3

Direct Proof:

easy because no particular technique is used.

not easy — starting point is not obvious

know your definitions

allowed to use any theorem, axiom, logic, etc.

Proof by Contradiction — Indirect proof

Example: $\sqrt{2}$ is irrational.

Assume $\sqrt{2}$ is rational

- A rational number is a number that can be made by dividing two integers
- It means $\sqrt{2} = \frac{p}{q}$ for a $p, q \in N$, $q \neq 0$, $(p, q) = 1$
- $\sqrt{2} = \frac{p}{q} \rightarrow 2 = \frac{p^2}{q^2}$

$\rightarrow \cancel{2p^2 = q^2}$ $2q^2 = p^2$

$\rightarrow p$ is even $\rightarrow p = 2k$

- $2q^2 = p^2 \rightarrow 2q^2 = (2k)^2 = 4k^2$

$\rightarrow q^2 = 2k^2 \rightarrow q$ is even

Proof by Contrapositive

- Prove $A \rightarrow B$ is true
- Prove $\neg B \rightarrow \neg A$ is true

Proving $A \rightarrow B$

Assume A is true, show B is true

Assume not B is true, show not A is true

WEEK 4

Proof by induction

The principle of mathematical induction

IF

$P(0)$ is true (If it starts true)

AND

$\forall k \in N, P(k) \to P(k+1)$ (and it remains true)

THEN

$\forall n \in N, P(n)$ is true

Three steps of Induction

Prove $P(0)$ is true

— Basis

Prove If $P(k)$ then $P(k+1)$

— inductive step

The assumption that $P(k)$ is true is called inductive hypothesis.

Conclude, by induction, that $P(n)$ is true for all n.

WEEK 5

Product rule for more tasks

- Suppose there is a job with k tasks
- If there are n_i ways of completing task i
- Then there are $n_1 \cdot n_2 \cdots n_k$ ways of completing this job.

The sum rule:

If a job can be done either in n ways OR in m ways, then the job can be completed in $m+n$ ways

The pigeonhole principle

If there are $k+1$ or more objects to be placed in k boxes, there is at least one box containing more than one object.

(Dirichlet Drawer Principle)

The generalised pigeonhole principle

If there are N objects to be placed in k boxes, there is at least one box containing at least $\lceil N/k \rceil$ ~~&~~ objects.

WEEK 6

$p(n,r) = \frac{n!}{(n-r)!}$ r-permutation

$c(n,r) = \frac{n!}{r!(n-r)!}$, r-combination
no order
$c(n,r) = c(n,n-r)$

WEEK 7

4.1 Finite Automata

Definitions:

- An Alphabet, Σ, is a non-empty set of symbols
  eg, $\Sigma = \{0,1\}$ binary alphabet
  $\Sigma = \{a,b,\dots,z\}$ lowercase letters
- A string or word is a finite sequence of letters drawn from an alphabet
- Empty strings denoted by ε are strings with zero occurrences of letters. Empty strings can be from any alphabet.
- Length of a string $|x|$
- The set of all strings composed from letters in Σ is denoted by Σ^*
  eg. if $\Sigma = \{0,1\}$, then $\Sigma^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, \dots\}$
- The set of all non-empty strings composed from letters in Σ is denoted by Σ^+
- The set of all strings of length k composed from letters in Σ is denoted by Σ^k
  eg. $\Sigma = \{0,1\}$, then $\Sigma^2 = \{00, 01, 10, 11\}$
  $\Sigma^3 = \{000, 001, \_\_\_\_, 111\}$
- Size of $\Sigma^k = |\Sigma|^k$

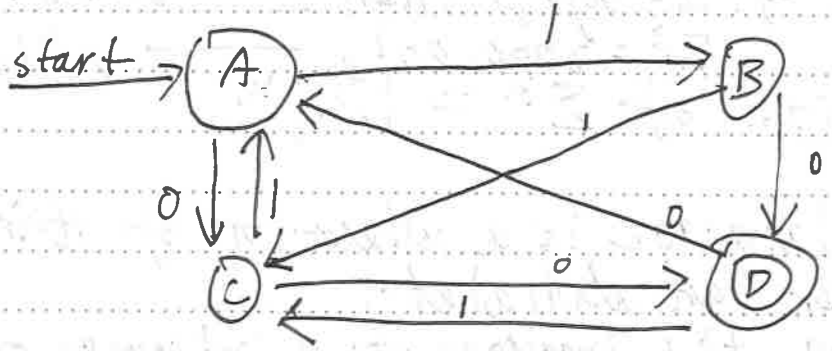
- A language is a collection of strings over an alphabet.
  eg. the language of palindromes over the binary alphabet is
  $\{\varepsilon, 0, 1, 00, 11, 000, 010, 101, 111, \dots\}$

- Σ^1 is not Σ
  Elements in Σ are called symbols and they are letters, whereas the elements in Σ^1 are strings.

- A finite automata is a simple mathematical machine; it is a representation of how computations are performed with limited memory space.
- It is a model of computation, which consists of a set of states that are connected by transitions
- input: ——

  output: Reject or Accept

eg.



A: read '1' → B

read '0' → C

◎D: accept state

An automaton M is 5-tuple $(Q, \Sigma, \delta, q_0, F)$;

Q is a finite set called the states

Σ is a finite set called the alphabet

$\delta: Q \times \Sigma \to Q$ is the transition function

$q_0 \in Q$ is the start state

$F \subseteq Q$ is the set of accept state

eg: $Q = \{A, B, C, D\}$

$\Sigma = \{0, 1\}$

$q_0 = A$

$F = \{D\}$

δ:	0	1
A	C	B
B	D	C
C	D	A
*D	A	C

4.2 Deterministic Automata

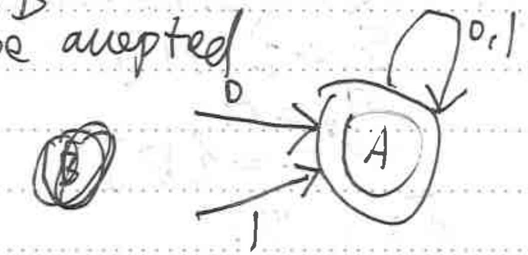
Only accept if the input ends in an accepting state

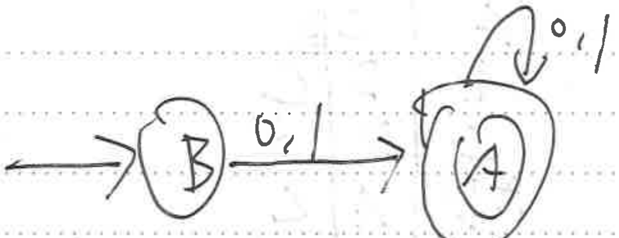
Backward computation

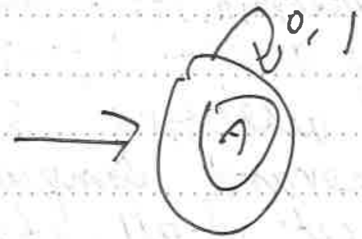
- The set of all strings accepted by an automaton is called the language of that automaton
- If M is an automaton on alphabet Σ, then $L(M)$ is the language of M

  $L(M) = \{x \in \Sigma^* \mid M \text{ accept } x\}$

Automaton accepting every binary
- a final state A
- the incoming arrows have no restriction
- the outgoing arrows from A end at A
- initial state, B
- 0 and 1 must be accepted







WEEK 8

DFA: deterministic finite automata

1. For each state in DFA, there is exactly one transition for each letter of alphabet.
2. There is a unique starting state

If 1 or 2 are not met: Non-deterministic

NFA: nondeterministic finite automata

There may be many choices at one particular point

There may be no path spelling the input.

An input is accepted if at least one sequence of choice leads to an accepting state.

5.1 Formal Languages.

Regular operations:

Let L_1 and L_2 be languages.

- Union: $L_1 \cup L_2 = \{x \mid x \in L_1 \text{ or } x \in L_2\}$
- Concatenation: $L_1 \circ L_2$

  $= \{xy \mid x \in L_1 \text{ and } y \in L_2\}$
- Star: $L_1^* = \{x_1 x_2 \cdots x_m \mid m \geq 0, \text{ each } x_i \in L_1\}$

Properties of regular operations

Union

- commutative: $A \cup B = B \cup A$

- associative: $(A \cup B) \cup C = A \cup (B \cup C)$
- Taking union with $\emptyset$: $A \cup \emptyset = A$
- idempotent: $A \cup A = A$

Concatenation
- associative: $(A \circ B) \circ C = A \circ (B \circ C)$
- $A \circ \varepsilon = \varepsilon \circ A = A$
- $A \circ \emptyset = \emptyset$

Concatenation and union
- $(A \cup B) \circ C = (A \circ C) \cup (B \circ C)$
- $A \circ (B \cup C) = (A \circ B) \cup (A \circ C)$

Kleene star
- $\emptyset^* = \{\varepsilon\}$
- $\varepsilon^* = \varepsilon$
- $(A^*)^* = A^*$
- $A^* A^* = A^*$
- $(A \cup B)^* = (A^* B^*)^*$

Atomic regular expressions
- The empty language, $\emptyset$, is a regular expression, which is the empty regular language.
- Any letter a in Σ is a regular expression, and it's language is $\{a\}$
- Empty string, ε, is a regular expression representing the regular language $\{\varepsilon\}$

Compound regular expressions
The regular operations preserve the regularity:
- Concatenation: If R_1 and R_2 are regular expressions, so is $R_1 \circ R_2$.
- Union. If R_1 and R_2 are regular expressions, so is $R_1 \cup R_2$.
- Kleene star. If R is a regular expression, so is R^*.

The language of regular expression
- the language of ab^*:
  $\{a, ab, abb, abbb, \cdots\}$
- $ab^* \cup b^*$
  $\{a, ab, abb, \cdots\} \cup \{\varepsilon, b, bb, bbb\}$
  $= \{\varepsilon, a, b, ab, bb, \cdots\}$
- $ab^+ \cup b^+b$
  $\{ab, abb, abbb, \cdots\} \cup \{bb, bbb, \cdots\}$
  $= ab^* \cup b^* / \{a, \varepsilon, b\}$

Examples on binary alphabet, $\Sigma = \{a, b\}$
- $\Sigma^* a$
  $\{a, aa, ba, aaa, aba, baa, bba, \ldots\}$
  all binary strings ending with a.
- $\Sigma^* a \Sigma^*$
  all binary words with at least one a

Read regular expressions
- Precedence of operations
  $*$, concatenation, union
  eg. $a \cup bc^* = a \cup (b(c^*))$
  $= \{a, b, bc, bcc, bccc, \ldots\}$

Design regular expressions
All binary words containing bb
- $(a \cup b)^* bb (a \cup b)^*$
- $\Sigma^* bb \Sigma^*$

All binary words ending with ab or ba
$\Sigma^* ab \cup \Sigma^* ba$
Binary strings with at most one a
$b^* a b^* \cup b^*$
Binary strings of length 3
$\Sigma \Sigma \Sigma$

Binary strings of length at least 3
$\Sigma \Sigma \Sigma \Sigma^*$
$\Sigma \Sigma \Sigma^+$
Binary strings of length 0, $\{\varepsilon\}$
- Regular expression: ε

Binary strings of length 1, $\{a, b\}$
- Regular expression: Σ

Binary strings of length 2 $\{aa, ab, bb, ba\}$
- Regular expression: $\Sigma \Sigma$

5.2 Regular Languages
Regular language and regular expression
- Kleene's theorem: a language is regular if and only if it can be described by a regular expression
- If and only if means we need a two-way theorem
  1. If a language is described by a regular expression, then it is regular.
  2. If a language is regular, it can be described by a regular

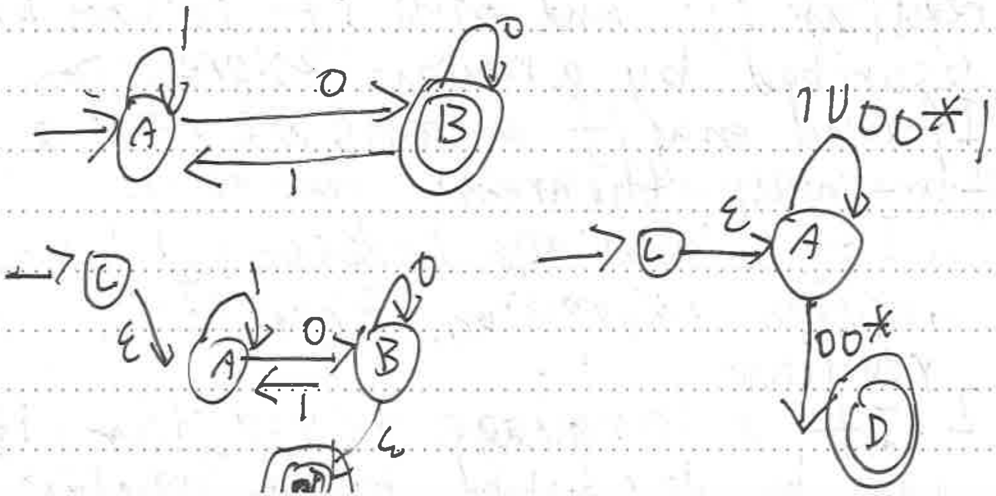
expression

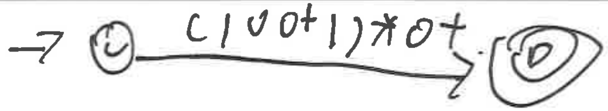
Two main theorems — Part 1

1. If $L = L(A)$ for some finite automaton A, then there is a regular expression R, such that $L(R) = L$

Converting a FA to RE

1. Create a new initial state
   - The transition is ε
2. Create a new final state
   - Connected to final states with transition ε
3. Remove states and transitions
4. Remove state B
5. Remove state A





Part 2

2. If $L = L(R)$ for some regular expression R, then there is a finite automaton A such that $L(A) = L$.

5-3 Pumping Lemma

Closure Properties

If L_1 and L_2 are regular languages on alphabet Σ, then the following languages are also regular.

- $U - L_1$. This means $\Sigma^* - L_1$ or the complement of L_1.
- $L_1 \cup L_2$
- $L_1 \cap L_2$
- $L_1 L_2$
- L_1^*

Examples of non-regular language

- $L = \{a^n b^n \mid n \in N\}$
- $L = \{xx \mid x \in \{a, b\}^*\}$
- $L = \{a^{n!} \mid n \in N\}$
- $L = \{xx^R \mid x \in \{a, b\}^*\}$
- $L = \{a^{n^2} \mid n \in N\}$

- $L = \{a^n \mid n \in N, n \text{ is prime number}\}$

Using closure properties — intersection

- Prove $L = \{x \in \{a,b\}^* \mid \#a \text{ in } x = \#b \text{ in } x\}$ is not regular
- $L = \{ab, aabb, abab, abba, baab, \ldots\}$
  Let's assume L is regular
  $L' = \{x \in a^*b^*\}$ is regular
  $L \cap L' = \{a^n b^n \mid n \in N\}$: not regular

Pumping Lemma

If length of the input $\geq$ number of states, then there are repeating states

If L is a regular language, then there is a number p (the pumping length) where, if s is any string in L of length at least p, then s may be divided into three pieces $s = xyz$, satisfying the following conditions:

- for each $i \geq 0$, $xy^iz \in L$
- $|y| > 0$ and
- $|xy| \leq p$ |xy|<=P

- If the language is finite, it is regular.
- We choose p to be the number of states in the FA representing L
- If $|s| \geq p$, s must have a repeated state (Pigeonhole Principle)

How it work?

- Assume the given language is regular
- Assume p is the pumping length
- Construct a string, s, whose length is at least p.
- Pumping Lemma: $s = xyz$ and for all $i \geq 0$, $xy^iz \in L$
- Condition 2 of the lemma: There is a y such that $|xy| \leq p$
- Investigate this y
- Prove that for one $i \geq 0$, $xy^iz \notin L$
- Contradiction!
- L is not regular.

eg. $L = \{a^n b^n \mid n \in N\}$
Let $s = a^p b^p$, $|s| \geq p$
$s = xyz$

for any i, $xy^iz \in L$. Let's try $i = 2$
1) y is only a's, $xyyz$ will have more a's than b's

2) y is only b's. $xyyz$ will have more b's than a's.

3) y has a's and b's. $xyyz$ will have a's and b's jumbled up.

eg. $L = \{xx \mid x \in \{a,b\}^*\}$

Let $s = a^p b a^p b$, $|s| > b$

$s = xyz$

For any i, $xy^iz \in L$, let's try $i = 2$

The third condition: $|xy| \leq p$

$\cdot$ So $y = a^q$, $q \leq p$

$\cdot$ $xyyz = a^{p+q} b a^p b \notin L$

WEEK 11

6.1 Gramma

Context-free gramma

- Gramma: set of rules for connecting strings together
- Another way of representing languages
- It works by recursively describing the structure of the strings

Formal definition

A context-free gramma is a 4-tuple (V, Σ, R, S), where

- Variables: a finite set of symbols, denoted by V
- Terminals: a finite set of letters, denoted by Σ; it is disjoint from V
- Rules: a finite set of mappings, denoted by R, with each rule being a variable and a string of variables and terminals
- Start variables: a member of V, denoted by S. It is usually the variable on the left-hand side of the top rule.

eg: $S \to bSa$

$S \to ba$

Two production rules

S is the non-terminal

Terminals: a, b

S is the start variable

Generating rules
1. Start from the starting symbol, read its rule.
2. Find a variable in the rule of the starting symbol and replace it with a rule of that variable.
3. Repeat step 2 until there are no variables left.

- A derivation is a sequence of substitutions in generating a string
- There may be more than one rule for a variable. Then we can use "|" symbol to indicate or.
- eg. $S \rightarrow bSa | ba$

eg. $S \Rightarrow bSa \Rightarrow bbaa$
$S \Rightarrow bSa \Rightarrow bbSaa \Rightarrow bbbaaa$

We say u derives v, or $u \Rightarrow^* v$, if there is a ~~devit~~ derivation from u to v

eg. $S \rightarrow aS | T$
$T \rightarrow b | \varepsilon$

- Variables: S, T; Terminals: a, b; Starting: S
- Find 3 strings derived from S

$S \Rightarrow$ ~~@P~~ $aS \Rightarrow aT \Rightarrow a$
$S \Rightarrow aS \Rightarrow aT \Rightarrow ab$
$S \Rightarrow T \Rightarrow \varepsilon$
bb: $S \Rightarrow T \Rightarrow b$
abb: $S \Rightarrow aS \Rightarrow aT \Rightarrow ab$
aba: $S \Rightarrow aS \Rightarrow aT \Rightarrow ab$

6.2 Regular languages

Language of a gramma

- All the strings that can be derived from the starting symbol using the rules of the gramma.
- Formal definition:
  If $G = (V, \Sigma, R, S)$ then $L(G) = \{w \in \Sigma^* | S \Rightarrow^* w\}$
- The language of any context-free gramma is context-free

eg. G_1: $S \rightarrow bSa$
$S \rightarrow ba$
S: ba, bbaa, bbbaaa, —
$L(G_1) = \{b^n a^n | n \geq 1\}$
$G_1 = (S, \{a, b\}, R, S)$,
$R = \{S \rightarrow bSa, S \rightarrow ba\}$

eg. G_2: $S \rightarrow aS | T$
$T \rightarrow b | \varepsilon$
$L(G_2)$: a, ab, b, ε, ~~ab~~ aa, aab

not in $L(G_2)$: ba, abb, aabb

$L(G_2) = a^* \cup a^* b = \{a^i b^j \mid 0 \le i, 0 \le j \le 1\}$

eg. G_3: $U \to XUX \mid 0Y$

$X \to Y \mid U$

$Y \to 1 \mid \varepsilon$

$L(G_3)$: $U \Rightarrow XUX \Rightarrow YUX \Rightarrow Y0YX$

$\Rightarrow Y0YY$

$\Rightarrow \{0, 01, 011, 10, 101, 1011\}$

$U \Rightarrow XUX \Rightarrow YUX \Rightarrow Y0YX \Rightarrow Y0YU$

$\Rightarrow Y0Y0Y \Rightarrow 1$ }

not in $L(G_3)$: ε, 1, 11, 111

- If the string contains no 0, then it is 1^n. This must have derived from Y.
- It is not from 0Y, so it is derived from X
- X occurs only in the rule for U, in the form of XUX.
- The stopping step is 0Y, so eventually we see one 0.

p.203 Design a gramma

Decompose and build recursively

- eg1: language of Palindromes over binary alphabet
- First letter is the same as the last
  - a: $S \to aAa$
  - b: $S \to bAb$), if A is a palindrome
  - $S \to \varepsilon$
  - A would be S
- $S \to aSa \mid bSb \mid \varepsilon$

Checklist:

- After designing a grammar:
- Consistency: if all strings generated by the grammar fit the description
- Completeness: all strings in the description can be generated by the grammar
- Terminating recursion: all recursions used in the grammar terminate.

eg. Binary strings with even number of 0's

- First letter is 1: $S \to 1A$, if A has even ~~nu~~ numbers of 0's
- First letter is 0:
  After some characters there must be another zero, so $W = 0A0B$

where A and B have even number of 0's

eg $S \to 0S0S$

· Empty string has an even number of 0's

· $S \to 1S \mid 0S0S \mid \varepsilon$

eg. 0^*1^*

· $S \to UV$

· $U \to 0U \mid U$

· $V \to 1V \mid 1$

eg: Design a CFG for $\{a^m b^n \mid n \geq m\}$

$a^m b^n = a^m b^{n-m} b^m$

if $n-m=0$: $a^m b^m$: $S \to aSb \mid \varepsilon$

if $n-m>0$: $i = n-m$, b^i: $U \to bU \mid b$

· $S \to aSb \mid U \mid \varepsilon$

$U \to bU \mid b$

or: $S \to aSb \mid U$

$U \to bU \mid \varepsilon$

eg: $L = \{a^n b^m \mid n+m \text{ is even}\}$

· n, m even

$A \to aaA \mid \varepsilon$

$B \to bbB \mid \varepsilon$

$a^n b^m$ $S \to AB$

· n, m odd:

$S \to aAbB$

· $S \to AB \mid aAbB$

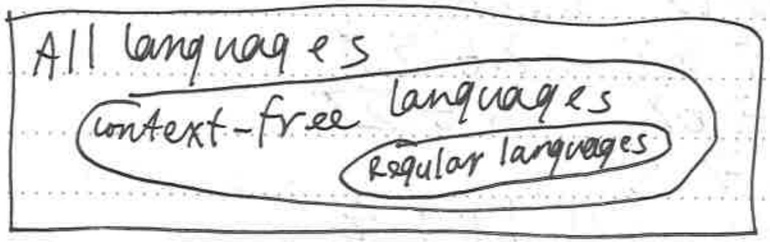
$A \to aaA \mid \varepsilon$

$B \to bbB \mid \varepsilon$

WEEK12, 6.301 RE to CFG

Regular languages can be expressed by REs.	Every regular language is also a context-free language	Context-free language can be generated by CFGs

All regular expressions can be written as context-free grammar



eg. Convert ab^* to CFG

· b^* : $U \to bU \mid \varepsilon$

· ab^* : $S \to aU$

· CFG is: $S \to aU$

CFG IS : s-> aU ,
U-> bU| ε

eg. convert $ab^* \cup b^*$ to CFG

- b^*: $U \to bU \mid \varepsilon$
- ab^*: $S \to aU$
- CFG: $S \to aU \mid U$
  $U \to bU \mid \varepsilon$

eg. ~~ab~~ $ab^+ \cup b^+b$

- b^+: $U \to bU \mid b$

ab^+ can be written as $S \to aU$

b^+b: $S \to bU$

- CFG: $S \to aU \mid bU$
  $U \to bU \mid b$

eg. $\Sigma^* a \Sigma^*$, $\Sigma = \{a, b\}$

- Σ^*: all strings on Σ
  - Strings starting with a
  - strings starting with b
  - Empty string
  - $U \to aX$, $X \in \Sigma^*$
  - $U \to bX$, $X \in \Sigma^*$
  - $U \to \varepsilon$
  - $U \to aU \mid bU \mid \varepsilon$
- $\Sigma^* a \Sigma^*$: $S \to UaU$
- CFG: $S \to UaU$
  $U \to aU \mid bU \mid \varepsilon$

eg Binary strings of length at least three, $\Sigma\Sigma\Sigma^+$

- $\Sigma^+ = (a \cup b)^+$: $U \to aU \mid bU \mid a \mid b$
- $\Sigma\Sigma^+$: $V \to aU \mid bU$
- $\Sigma\Sigma\Sigma^+$: $S \to aV \mid bV$
- Or $S \to aaU \mid abU \mid baU \mid bbU$
- CFG is: $S \to aV \mid bV$
  $V \to aU \mid bU$
  $U \to aU \mid bU \mid a \mid b$
  OR: $S \to aaU \mid abU \mid baU \mid bbU$
  $U \to aU \mid bU \mid a \mid b$

6.3.04 Chomsky Normal Form

- A context-free grammar is in Chomsky Normal Form if every rule is of the form
  $S \to XU$
  $S \to a$
- a is any terminal
- X, U, and S are non-terminals
- X and U are not the start variable
- $S \to \varepsilon$ is permitted if S is the start variable

Not Chomsky

- $S \to 1S \mid 0S0S \mid \varepsilon$: S appears on the right-hand side

- $V \to \varepsilon$, where V is not a start variable
  - ε-rules
- $U \to V$, where V is a variable
  - Unit-rules
- $X \to UV$, the length of the rule is ≥ 3
  - Improper rules

Convert to Chomsky Normal Form

1. Add a new start variable, S_0, make a rule $S_0 \to S$
   - guarantees that S will not occur in the right-hand side of any rule
2. Eliminate ε-rules, $U \to \varepsilon$, where U is not a start variable.
   For each occurrence of U on the right-hand side of a rule:
   - add a new rule with that particular occurrence of U deleted
   - example: $S \to Xb$, $X \to UVU \mid bX$, $V \to aU$, $U \to a \mid \varepsilon$
   - $U \to a$
   - $V \to aU \mid a$
   - $X \to UVU \mid bX \mid VU \mid UV \mid V$
3. Remove the ~~unit~~ unit rules, $A \to B$
   - Add $A \to X$ for every $B \to X$; X is a string of variables and terminals
   - Example: $S \to YU$, $U \to Y \mid a$, $Y \to UY \mid b$
   - $U \to a \mid UY \mid b$
4. Convert to proper forms
   - $U \to X_1 X_2 \cdots X_n$, where $n \geq 2$ to $U \to X_1 U_1$, $U_1 \to X_2 U_2$, ..., $U_{n-3} \to X_{n-2} U_{n-2}$, $U_{n-2} \to X_{n-1} X_n$
   - Example. $S \to YUVa \mid Sb$
   - $X \to YZ$, $Z \to UV$
   - $S \to Xa \mid Sb$

7.1 Turing machine

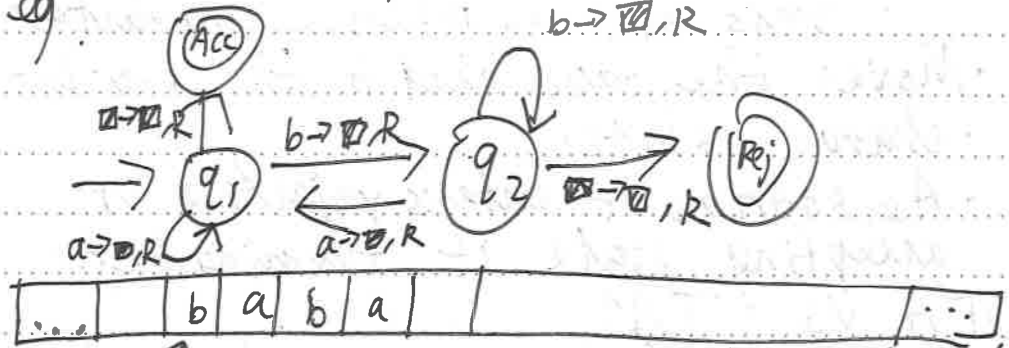
- a finite automaton with random access memory
- assumed with unbounded memory
- a finite state automation that provides ~~insuran~~ instructions on an infinite tape, where the input is given and can also be the working space working space

· every cell contains one character
· some cells are empty
· a tape head that reads and writes according to the instructions given by FSA

Formal definition
A TM: $(Q, \Sigma, \Gamma, \delta, q_1, q_{Acc}, q_{Rej})$
· the tape alphabet Γ that includes the blank symbol.
· the input alphabet Σ, $\Sigma \subseteq \Gamma$
· tape head
· a finite set of states Q
· a start state $q_1 \in Q$
· an accepting and rejecting state q_{Acc}, q_{Rej}.
· a transition function: $\delta : Q \times \Gamma \to (Q \times \Gamma \times \{L, R\})$

Transition function:
· $\delta : Q \times \Gamma \to (Q \times \Gamma \times \{L, R\})$
· It takes one state and one letter from Γ
· It returns:
  · a state of the automation
  · a letter to be written on the current cell of the tape
  · the direction, instructing the tape head where to go, L for left, R for right.

eg.



initially, the tape head → start of input

— transitions
· Each transition:
  1. read — write
     ■ means blank
  2. dir: R right, L left

a→b, R means if the letter under the tape head is a, then write b instead and go right.

δ	a	b	■
q_1	q_1, ■, R	q_2, ■, R	Accept
q_2	q_1, ■, R	q_2, ■, R	Reject

Turing machine at each step
- Tape head is on the first character of the input
- Reads what is under the tape head
- Writes a character under the tape head
  - this would be blank character
- Moves the tape head to the R or L
- change states
- As soon as it enters rejecting or accepting state, it terminates

DFA vs TM
- TM may not terminate when the input is completely processed
- It may process the input several times
- Only terminate at accepting or rejecting states
- They may enter an infinite loop
- Manipulate input
- TMs are deterministic

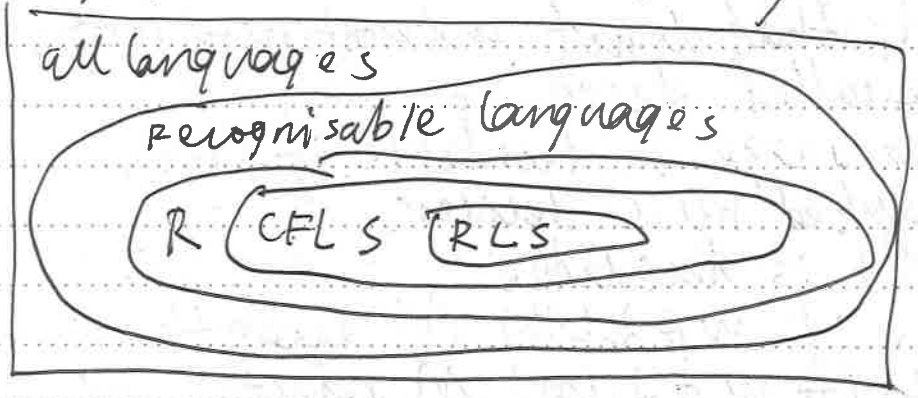
7.303
- Language of TM is $\mathcal{L}(M)$
  $\mathcal{L}(M) = \{w \in \Sigma^* \mid M \text{ accepts } w\}$
1. If $w \in \mathcal{L}(M)$, M reaches accept state
2. If $w \notin \mathcal{L}(M)$, M does not reach accept state:
   - either reaches reject state
   - or enters a loop (infinite)
- A language is recognisable if it is accepted by a TM
- The TM, M that is called the recogniser of $\mathcal{L}(M)$
- RE is a class of all recognisable languages.

- What if we want to avoid infinite loops?
- TMs that do not enter infinite loops are called deciders
- A language is decidable if it is accepted by a decider.
- $\mathcal{L}(M)$ is decidable
  1. if $w \in \mathcal{L}(M)$, M accepts w
  2. if $w \notin \mathcal{L}(M)$, M rejects w
- R is a class of all decidable languages

Halting problem
- Every decider is a TM
- $R \subset RE$

· Halting problem: TM halts? or not?
· Church-Turing thesis: solving halting problem is undecidable problem
· We cannot determine if the TM enters an infinite loop or not.

Every regular language is context-free
FSA or RE
Every context-free language is Turing-decidable CFGs.
Every decidable language is recognisable Decider Turing machine



Chomsky Hierarchy

Grammer	languages	Automaton	eg
Type-0	recursively enumerable	Turing Machine	
Type-1	context-sensitive	TM with bounded Tape	$a^n b^n c^n$
Type-2	context-free	push-down	$a^n b^n$
Type-3	Regular	Finite state	$a^* b^*$

8.

An algorithm is a set of steps required to complete a task.
A receipt is an example of such steps
Informal definition
Algorithm: steps ~~eg~~ required to take the input and achieve the outcome
A formal definition
· An algorithm can be defined as an ordered set of unambiguous executable steps that form a terminating process.
Ordered: after every step, we know what to do next — does not mean sequential
Unambiguous: each ~~opt~~ operation is sufficiently clear
Executable: each operation must be doable. eg. dividing by zero is not doable.

Terminating: there is finite number of executions. The process should halt eventually.

Insertion sort:
- pick the first item in the unordered part of the list
- compare with previous items (ordered)
- move item if necessary
- i=2
- while i in the length
  - select the ith entry of the list as a pivot entry.
  - move the pivot entry to a temporary position, leave a hole in the list.
  - While there is an item above the hole and item > pivot entry
    - move the item to the hole, leaving a hole above the item
  - Move the pivot entry into the ~~the~~ hole ·i++

Bubble sort
- end = length − 1
  while end > 1:
      i = 1
      while i does not exceed end
          if list[i] > list[i+1]:
              swap
          i = i + 1
      end = end − 1
  return list

8.3 Binary search def search(list, item)
If List is empty
    Report failed
Else:
    Pivot = the middle entry of list
    If ~~Pro~~ pivot = item:
        Report succeeded
    Else if Pivot > item:
        List = items preceding the pivot
        Report the list, search(list, item)
    Else:
        List = item following the pivot
        Report the list, Search(list, item)

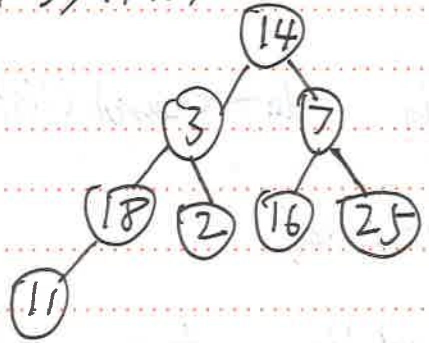
Heap sort

binary tree: a rooted tree; every vertex has no more than two children

a complete binary tree:

- Each node has two children, except possibly the nodes in the last level.
- All leaves in the last level are placed as far to the left as possible

14, 3, 7, 18, 2, 16, 25, 11

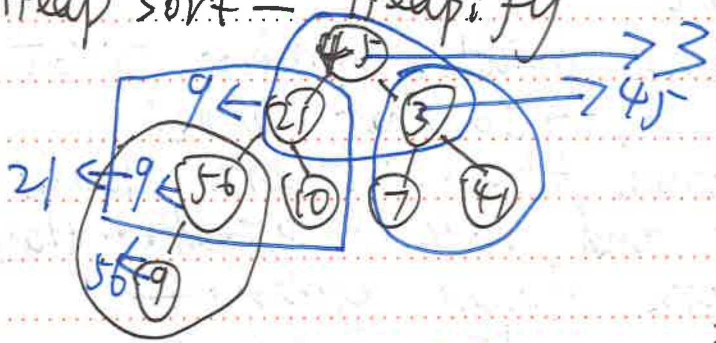


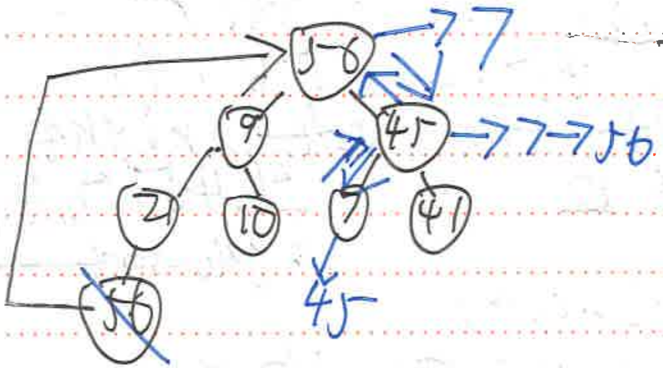
Max Heap

- a complete binary tree
- each internal node has a value greater than or equal to its children

Min Heap. → smaller than or equal

Heap sort — Heapify





3

3, 7

3, 7, 9, —

9.1 Recursion

```
def gcd (a,b):
    if b=0:
        return a
    else:
        r=a mod b
        return gcd (b,r)
```

Quick sort

- choose the middle number pivot.
- Record the list so that values smaller than the pivot are placed before it and values greater than the pivot are placed after it

45, 21, 3, 10, 56, 7, 41, 9

3, 7, 9, 10,   45, 21, 56, 41

- Quick sort the left, right

3 7 9 10   21, 45, 56, 41

↓ quicksort

```
def quickSort(List)
    If the list has one item
        Return List
    ELSE:
        Pivot = the middle entry
        Delete pivot from the list
        for item in list
            if Pivot > item
                ListLeft.append(item)
            ELSE:
                ListRight.append(item)
        C = QuickSort(ListLeft) +
            Pivot + QuickSort(ListRight)
        Return C
```

9.2 Merging list

```
def Merge (A, B):
    While (A, B have elements):
        if (A[1] > B[1]):
            Append B[1] to the end of C
            Remove B[1] from B
        else:
            Append A[1] to the end of C
            Remove A[1] from A.
    While (A has elements):
        Append A[1] to the end of C
        Remove A[1] from A
    While (B has elements):
        Append B[1] to the end of C
        Remove B[1] from B
    Return C
```

Merge sort:

keep splitting

$\boxed{2, 13}$, 45, 21, 3 | $\boxed{10, 56}$, 7, 41, 9

$\boxed{2, 13, 45}$, $\boxed{3, 21}$

$\boxed{2, 3, 13, 21, 45}$

```
def MergeSort(List)
    N = size of list
    if N=1:
        return List
    ListLeft = List[1 ... ⌈N/2⌉]
    ListRight = List[⌈N/2⌉+1 ... N]
    ListLeft = MergeSort(ListLeft)
    ListRight = MergeSort(ListRight)
    Return Merge(ListRight, ListRight)
```

9.3 The algorithm of happiness
- There are n hospitals and n medical students
- There are lists of preferences for students and the hospitals.
- Pair hospitals and students such that the matching is stable
- Unstable pair student s and hospital h:
    - s prefers h
    - h prefers s
    - They may make a side deal
- Stable matching:
    - There is no unstable pair

Input:
- List of hospitals
- List of students
- Numbers of students and hospitals are equal
- One students per hospital
- Each hospital provides a list of preferences (order of students)
- Each students provides a list of preferences (order of hospitals)

Definitions
- H is the list of hospitals, S — students
- A matching M is a set of pairs, (h, s) where $h \in H$, $s \in S$
    - Each hospital appears at most in one pair of M
    - Each student appears at most in one pair of M
- A matching is perfect if:
    - each hospital appears at least in one pair of M
    - each students appears at least in one pair of M
    - $|M| = |H| = |S|$

Gale-Shapley Algorithm, 1962

- Step1
  - Each unmatched hospital offers a place to a student on the top of its list
- Step2
  - students with one offer:
    - accept the offer
  - Students with more than one offer.
    - accept the top hospitals that made them an offer
- Repeat until all hospitals are matched

Pseudocode:

- M empty set

While (there is h unmatched and it has not been rejected by all students):

    s = first student on h's list
      who have not rejected h
    if s is unmatched:
        Add (h, s) to M
    else if s prefers h to current
        hospital h':
            Delete(h', s) from M
            Add(h, s) to M
    else:
        s rejects h

Return M

Proof of correctness
  terminates
return perfect matching
  match is stable

10.1 Efficiency

worse case: guarantees the time needed for any input

average case:

best case: rarely used

Insertion sort: average case:

$\frac{1}{4} \cdot n(n-1)$

- average over all cases
- on average, every item is compared to the half of the sorted items
- half of the worse case

Worst case: $\frac{n(n-1)}{2}$

Best case: $n-1$

Bubble sort: best case: $n-1$

worse case: $\frac{n(n-1)}{2}$

average case: n^2

Binary search: best case: 1

worse case: $\log n$

average case: $\log n$

10.2 Asymptotic complexity

- asymptotic behavior is the growth of the function when n is large.
- estimate how the algorithms become with large input
- ignore small expressions.
- eg. $f(n) = 2n^2+5n$, $g(n) = 4n^2-7n$
  same asymptotic behavior

Big O notation

- Let $f(n)$, $g(n)$, be two functions, $f(n)$ is $O(g(n))$, if there are constants C and K such that:
  $f(x) \le c \cdot g(x)$, $x > K$
- $f(n)$ is Big O of $g(n)$
- This means $f(n)$ grows slower than some multiples of $g(n)$, when n grows.
- C and K are called the witnesses to the relation of $f(n)$ and $g(n)$
- $g(n) = c \cdot n^2$
- $f(n) = 2n^2+4n+40$
- choose $c=3$, $k=9$

eg: show that $7n+n^3+1 = O(n^3)$

$7n+1 < n^3$ if $n > 2$

$7n+n^3+1 < 2n^3$

$c=2$, $k=2$

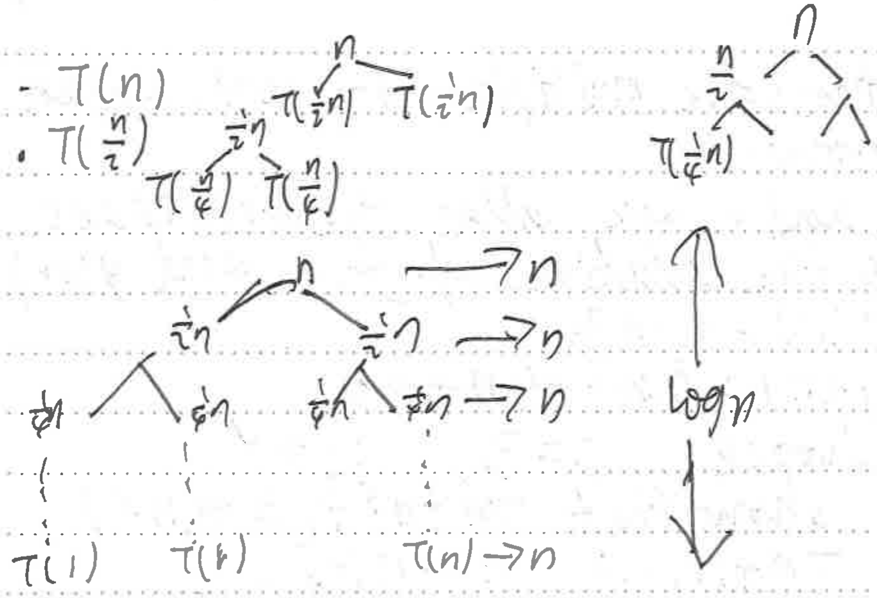
10.3 Recursion complexity

1. prove by induction

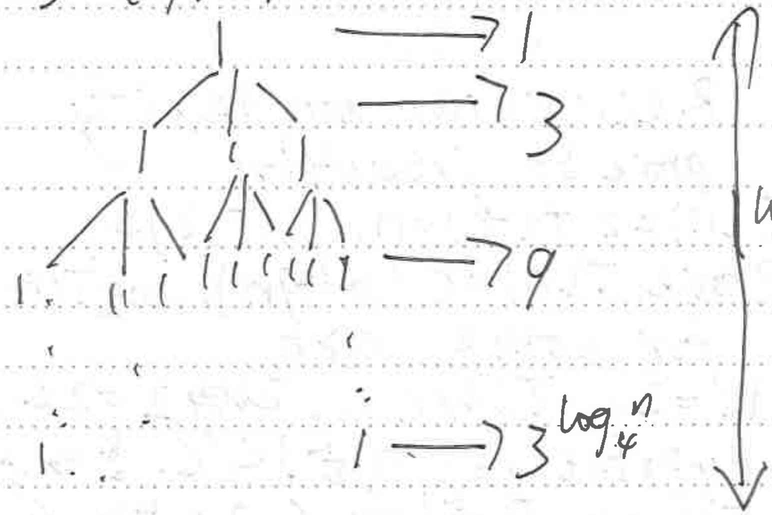
- $T(n) = 2T(\frac{n}{2})+n$, $T(2)=1$
- Prove $T(n) = O(n\log n)$, or $T(n) < c \cdot n\log n$ for some $c > 0$
- $k=2$, $T(2) < c$, $2\log 2 = 2c$, TRUE
- Hypothesis: $T(\frac{k}{2}) < c \cdot \frac{k}{2}\log\frac{k}{2}$
- $T(k) = 2T(\frac{k}{2})+k < 2c\frac{k}{2}\log\frac{k}{2}+k$
  $= ck\log k - ck\log 2 + k$
  $= ck\log k - ck + k \le ck\log k$

2. Tree method

- $T(n) = 2T(\frac{n}{2})+n$.



· $T(n) = 3T(\frac{n}{4}) + 1$



3. Master Theorem

· $T(n) = T(\frac{n}{2}) + O(1)$, $T(n) = O(\log n)$

$T(n) = 2T(\frac{n}{2}) + O(n)$, $T(n) = O(n \log n)$

$T(n) = 4T(\frac{n}{2}) + O(n)$, $T(n) = O(n^2)$

• The recurrence $T(n) = aT(\frac{n}{b}) + O(n^d)$,

$a \geq 1$, $b > 1$, $d \geq 0$

$d < \log_b a \rightarrow T(n) = O(n^{\log_b a})$

$d = \log_b a \rightarrow T(n) = O(n^d \log n)$

$d > \log_b a \rightarrow T(n) = O(n^d)$

eg. $T(n) = T(\frac{n}{2}) + O(1)$,

$a = 1$, $b = 2$, $d = 0$

$0 = \log_2 1$

$T(n) = O(\log n)$

· $T(n) = 3T(\frac{n}{4}) + O(1)$, $a = 3$, $b = 4$, $d = 0$

$0 < \log_4 3$

$T(n) = O(n^{\log_4 3})$

Quicksort: average case.

· $T(n) = n + T(i) + T(n-1-i)$

· average: $\frac{1}{n} \sum_{i=0}^{n-1} (T(i) + T(n-1-i) + n)$

$= \frac{1}{n} (2 \sum_{i=0}^{n-1} T(i) + n(n))$

· $T(n) = \frac{2}{n} (T(n-1) + T(n-2) + \cdots + T(0)) + (n)$

Worst case: $1 + 2 + \cdots + (n-1) = \frac{1}{2} n(n-1)$

best case:

$T(n) \approx 2T(\frac{n}{2}) + n$

$\approx 2(2T(\frac{n}{4}) + \frac{n}{2}) + n = 4T(\frac{n}{4}) + 2n$

$T(n) \approx nT(1) + ?n \approx n + (\log n) n$

$a = 2$, $b = 2$, $d = 1$, $\log 2 = d$, $\Rightarrow T(n) = n \log n$

Merge sort:

best/worst case

- $\log n$ levels
- at every level, merging takes $O(n)$ times
- if the list is sorted, we still have to read the items
- $T(n) \approx O(n \log n)$

average case:

- if $n=1$, return the list
- else:
  - $\text{merge}(\underset{\Downarrow\, T(\frac{n}{2})}{\text{mergesort}}, \underset{\Downarrow\, T(\frac{n}{2})}{\text{mergesort}})$
- $T(n) = 2T(\frac{n}{2}) + cn$, $a=2, b=2, d=1$,

  $d = \log_b a$,

$\Rightarrow T(n) = O(n \log n)$

	Worst-case Time	Space
Bubble	$O(n^2)$	$O(1)$
Insertion	$O(n^2)$	$O(1)$
Quick	$O(n^2)$ *	$O(\log n)$
Merge	$O(n \log n)$	$O(n)$

*: $O(n \log n)$ on average