Contents

I Logic 3

1 Introduction 5

2 Propositional Logic 7

2.1 Syntax . . . 7
2.2 Semantic . . . 9
2.3 Proofs . . . 11
2.4 Sequent Calculus . . . 12

II Computability Theory 13

3 Lecture Notes 15

3.1 Computation . . . 15
3.2 Formal Languages . . . 17
3.3 Computation Machines . . . 17
3.4 Complexity of the languages . . . 18
3.5 Gap problems . . . 18
3.6 Constraining Turing machines . . . 18

III Advanced Algorithms 19

4 Introduction 21

4.1 Computation Variants . . . 21

5 Amortized Analysis 23

5.1 Table insertion . . . 23

Part I

Logic

Chapter 1

Introduction

Mathematical logic is about formalizing mathematic thinking and considers inferences and proofs. The goal in mathematical logic is to construct mathematical models and systems.

It must be noted that mathematical logic is a part of mathematics. In fact it is akin to a Persian grammar book in Persian. It talks about the rules in the language in the same language. In the same way, logic explains various models, mathematically.

To formalize a thought we need a language. In other words, the materalization of a thought is a set of words in a language. There are two aspects two languages; syntax and semantic. The syntax of a language is the rules and principles of the language. On the other hand, the semantic is about the meaning of words in a language.

Mathematical logic is typically divided into four parts;

1. Set theory.

2. Model theory.

3. Proof theory.

4. Recursion theory.

Chapter 2

Propositional Logic

A language is formed from an alphabet and a set of principles to construct words and sentences.

Definition: The language $\mathcal{L}$ of a propositional logic contains

1. A countable alphabet, also known as propositional symbol, $Q = \{p_0, p_1, p_2, \ldots\}$.
2. Logical connectives $\wedge, \vee, \neg, \rightarrow, \top$, and $\bot$.
3. Paranthesis (,).

2.1 Syntax

The words in a mathematical language are called expressions.

Definition: An expression is a finite sequence of propositional symbols, logical connectives, and paranthesis. The set of all expressions in $\mathcal{L}$ is denoted by $\mathrm{EXP}_{\mathcal{L}}$, however, we usually omit the subscript.

However, not all strings are acceptable in a language. In propositional logic, propisitions or well-formed formulea are the acceptable expression.

Definition: The set of well-formed formulea, $\mathrm{PR}_{\mathcal{L}} \subset \mathrm{EXP}_{\mathcal{L}}$ is the smallest subset with the following properties.

1. $\top, \bot \in \mathrm{PR}$,
2. The atomic propisition p_i are in PR, $P \subset \mathrm{PR}$.
3. For all propositions $A, B \in \mathrm{PR}$, the expressions $(A \wedge B)$, $(A \vee B)$, and $(A \rightarrow B)$ are propositions.
4. For all propositions $A, \in \mathrm{PR}$, the expression $(\neg A) \in \mathrm{PR}$.

The set of $P = Q \cup \{\top, \bot\}$ is called the atmoic symbols.

The majority of results in propsitional logic are proved with induction.

Theorem 2.1 (Principle of Induction). *Suppose $\mathcal{P}$ is a property of propositions that;*

1. *$\mathcal{P}$ holds all propositional symbol p_i and $\top, \bot$.*

2. *If $\mathcal{P}$ holds propositions A and B, then $(A \vee B), (A \wedge B)$, and $(A \to B)$ satisfy $\mathcal{P}$.*

3. *If $\mathcal{P}$ holds for a proposition A, then it holds for $(\neg A)$.*

Then, $\mathcal{P}$ holds for all propsitions.

Proof. Suppose $X \subset \mathrm{PR}$ is the set of all propositions that satisfy $\mathcal{P}$. Then, by definition, $P, \bot \subset X$ and thus X is non-empty. Moreover, from the second and third statement, we conclude that $\mathrm{PR} \subset X$ hence, $X = \mathrm{PR}$. ■

In these notes, we consider the symbol $\bot$ as an atomic proposition.

Theorem 2.2 (Uniqueness of Function Extension). *Suppose there are functions, $f_P : P \to S$, $f_\wedge, f_\vee, f_\to : S^2 \to S$ and $f_\neg : S \to S$, where S is an arbitrary set, then there exists a unique function $F : \mathrm{PR} \to S$ such that;*

1. *$F(p) = f_p(p)$ for all $p \in P$.*

2. *$F((A \circ B)) = f_\circ(F(A), F(B))$ for all propisitions A and B, where $\circ \in \{\wedge, \vee, \to\}$.*

3. *$F((\neg A)) = f_\neg(F(A))$ for all propositions A.*

Proof. We call a function $f_P \subset F \subset \mathrm{PR} \times S$ a good function if it satisfies the following conditions.

1. When $(A \circ B)$ is in its domain, then A and B are in its domain as well, and $F((A \circ B)) = f_\circ(F(A), F(B))$.

2. When $(\neg A)$ is in its domain, then A is in its domain as well, and $F((\neg A)) = f_\neg(F(A))$.

Consider the set of such good function. This set is non-empty as f_P is a good function. Let F be the union of all good functions. Obviously, such a set F exists.

1. First we must show that F is a function. That is, for each A in its domain there is at most one $s \in S$. Let X be the set of all propositions that have at most one value s such that $(A, s) \in F$. By construction, all atomic propositions are in X. Suppose $A, B \in X$ and $(A \circ B)$ is in the domain F. Then, there must some good functions such as $G_0, G_1, \ldots$ that have $(A \circ B)$ in their domain. Since $A, B \in X$, then we must have $G_i(A) = G_0(A)$ and $G_i(B) = G_0(B)$ for any i. As a result,

$$G_i((A \circ B)) = f_\circ(G_i(A), G_i(B)) = f_\circ(G_0(A), G_0(B)) = G_0((A \circ B)) \tag{2.1}$$

   hence, $(A \circ B) \in X$. Similarly, when $A \in X$ and $(\neg A)$ is in the domain F, there must some good functions such as $G_0, G_1, \ldots$ that have $(\neg A)$ in their domain. Since $A \in X$, then we must have $G_i(A) = G_0(A)$ for any i. As a result,

$$G_i((\neg A)) = f_\neg(G_i(A)) = f_\neg(G_0(A)) = G_0((\neg A)) \tag{2.2}$$

   hence, $(\neg A) \in X$. By the principle of induction $X = \mathrm{PR}$ and thus, F is a function.

2. Moreover, F must be a good function. This, follows immediately from the definition.

3. The domain F is PR. Let X be the domain of F. By definition, $P \subset X$. Moreover, if $A, B \in X$, then there must exists a good function G with A and B in its domain. Let $G^* = G \cup \{((A \circ B), f_\circ(G(A), G(B)))\}$. Clearly, G^* is a good function, hence $(A \circ B) \in X$. A similiar argument can be for $(\neg A) \in X$ when $A \in X$. Thus, $X = \mathrm{PR}$.

4. Lastly, we must show that F is unique. Suppose, there is another function G that satisfies the criteria and let X be the set of propositions A such that $F(A) = G(A)$. From the principle of induction we must have $F \equiv G$ thus, F is unique. ∎

– priority of operations. – examples of induction

For all proposition A we can define the set of all its sub-proposition, SPA, and it can be defined inductively. Order of operation.

2.2 Semantic

The semantic of a propositional logic considers the truth and falsity of propositions.

Definition: An interpretation is a function $I : \mathrm{PR} \to \{0, 1\}$ such that;

1. $I(\bot) = 0$ and $I(\top) = 1$.
2. $I(A \wedge B) = 1$ if and only if $I(A) = 1$ and $I(B) = 1$.
3. $I(A \vee B) = 0$ if and only if $I(A) = 0$ and $I(B) = 0$.
4. $I(A \to B) = 0$ if and only if $I(A) = 1$ and $I(B) = 0$.
5. $I(\neg A) = 1$ if and only if $I(A) = 0$.

The last two properties are points of contention among logicians. we say I models A, denoted by $I \models A$, when $I(A) = 1$ and $I \not\models A$ when $I(A) = 0$, for some proposition A. Generally, if Γ is a set of propositions, $I \models \Gamma$ means that I models every proposition $B \in \Gamma$.

Definition: A valuation is an interpretation function restricted to the atoms, $v : P \to \{0, 1\}$, with $v(\top) = 1$ and $v(\bot) = 0$.

Theorem 2.3. *For each valuation function v there is unique extension to an interpretation function I.*

Proof. Let I be defined as follows.

1. $I(p) = v(p)$ for atomic propositions.
2. $I(A \wedge B) = I(A) \cdot I(B)$.
3. $I(A \vee B) = I(A) + I(B) - I(A) \cdot I(B)$.
4. $I(A \to B) = I(A) \cdot I(B) + 1 - I(A)$.
5. $I(\neg A) = 1 - I(A)$.

The existence and uniqueness of I is proved from 2.2. ∎

Definition: If for all interpretation functions I that models Γ, I models A as well, we write $\Gamma \models A$. When $\Gamma = \emptyset$, we say A is a tautology, denoted by $\models A$. Moreover, two propositions A and B are equivalent, denoted by $A \equiv B$, if $\models (A \to B) \wedge (B \to A)$.

We introduce the logical connective $A \leftrightarrow B$ as a shorthand for $(A \to B) \wedge (B \to A)$.

Let $A[p/B]$ denote the substituition of every propositional symbol p in A with the proposition B. For example, if $A = p_0 \wedge p_1$ and $B = p_2 \vee p_0$, then $A[p_0/B] = (p_2 \vee p_0) \wedge p_1$.

Theorem 2.4. *If $\models A$, then $\models A[p/B]$ for all choices of p and B.*

Proof. Suppose the interpretation function I is generated by valuation v. Define the valuation function u such that $u(p) = I(B)$ and $u(q) = v(q)$ for all propositional symbols q other than p. Let J be the interpretation function generated by u. Then, $J \models A$ if and only if $I \models A[p/B]$. Since $\models A$, then $\models A[p/B]$ ■

Lemma 2.5. *We have $\models A \to B$ if and only if $I(A) \leq I(B)$ for all interpretations I.*

Proof. For any interpretation I, we must have $I(A \to B) = 1$. Thus,

$$I(A \to B) - 1 = I(A)I(B) - I(A) \tag{2.3}$$
$$= I(A)(I(B) - 1) \tag{2.4}$$

If $I(A) = 0$, then the lemma follows trivially. If $I(A) = 1$, then we must have $I(B) = 1$ and hence, $I(A) \leq I(B)$. ■

Lemma 2.6. *Suppose $\models A \to B$, then if $\models A$, then $\models B$.*

Proof. An immediate corrolary of the previous lemma. ■

Lemma 2.7. *For all choices of p, $I(B_1 \leftrightarrow B_2) \leq I(A[p/B_1] \leftrightarrow A[p/B_2])$.*

Proof. Without loss of generality suppose $I(B_1 \leftrightarrow B_2) = 1$ and suppose A contains the propositional symbol p. Then, $A[p/B_1] = B_1$ and $A[p/B_2] = B_2$ and thus, the substituition holds trivially, $I(A[p/B_1] \leftrightarrow A[p/B_2]) = 1$. Let $A = A_1 \circ A_2$, then $I(A_i[p/B_j]) = I(A_i[p/B_k])$ for $j \neq k$ and as a result, $I(A[p/B_j]) = I(A[p/B_k])$. A similiar argument holds for $\neg$. ■

Theorem 2.8 (Substituition). *If $\models B_1 \leftrightarrow B_2$, then $\models A[p/B_1] \leftrightarrow A[p/B_2]$.*

Proof. Immediately follows from the preceeding lemmas and theorems. ■

2.2.1 Truth Table

The simplest method determining whether a proposition is a tautology is constructing a truth table. In each row, we consider a valuation function on the propsitional symbols of that proposition. As a result, for proposition containing n symbols, we need a table of 2^n rows. We thus have the following theorem.

Theorem 2.9. *The set of tautologies in the set of propositions is decidable.*

– some tautologies + deMorgans

From the above tautologies and the deMorgan's ralations, it is apparent that we describe all connectives with $\{\wedge, \neg\}, \{\vee, \neg\}$, $\{\wedge, \rightarrow\}$, etc. We call a set of connective functionally complete if all other connectives can be written in terms that set. The Sheffer connective $|$, defined as $A|B = \neg(A \wedge B)$, is complete by itself. One might ask if there is another connective that can not be written with Sheffer connective. Suppose O is an n-ary functional connective, that is there exists a function f_O such that for all interpretations I and symbols $p_1, \ldots, p_n$.

$$I(O(p_1, \ldots, p_n)) = f_O(I(p_1), \ldots, I(p_n))$$

The following theorem states that all functional connectives can be written in terms of Sheffer or any other complete set of connectives.

Theorem 2.10. *For all functional connectives O, there exists a proposition only composed of $p_1, \ldots, p_n$ and connectives $\{\vee, \neg\}$ such that $\models O(p_1, \ldots, p_n) \leftrightarrow A$.*

When the constrained to $\{\wedge, \vee, \not\}$ we can writte every proposition in a conjunctive or distinjuctive normal form.

1. A proposition A is in the conjunctive normal form if A is

   $$(A_{1,1} \vee \cdots \vee A_{1,i_1}) \wedge \cdots \wedge (A_{m,1} \vee \cdots \vee A_{m,i_m})$$

   where all $A_{i,j}$ are either a propositional symbol or its negate.

2. A proposition A is in the disjunctive normal form if A is

   $$(A_{1,1} \wedge \cdots \wedge A_{1,i_1}) \vee \cdots \vee (A_{m,1} \wedge \cdots \wedge A_{m,i_m})$$

   where all $A_{i,j}$ are either a propositional symbol or its negate.

Theorem 2.11. *For a propisition A with symbols $p_1, \ldots, p_n$;*

1. *there exists an equivalent conjunctive normal form with the same symbols.*
2. *there exists an equivalent disjunctive normal form with the same symbols.*

2.3 Proofs

2.3.1 Inference rules

– Hilbert's method

2.3.2 Natural Deduction

Everything is a rule. Two types of rules, introduction rules and elimination rules.

2.4 Sequent Calculus

Suppose Γ and Δ are finite sets of propositions. By $\Gamma \Rightarrow \Delta$ we mean

$$(\gamma_1 \wedge \gamma_2 \wedge \cdots \wedge \gamma_n) \to (\delta_1 \vee \delta_2 \vee \cdots \vee \delta_m) \tag{2.5}$$

If $\Gamma = \emptyset$ or $\Delta = \emptyset$, then $\Gamma \Rightarrow \Delta$ is $\top$.

Definition (Axiom system): The axiomatic principles is

1. $A \Rightarrow A$.
2. $\perp \Rightarrow A$.

The structural principles

1. If $\Gamma \Rightarrow \Delta$, then $\Gamma \Rightarrow \Delta, A$.
2. If $\Gamma \Rightarrow \Delta$, then $A, \Gamma \Rightarrow \Delta$.
3. If $\Gamma \Rightarrow \Delta, A, A$, then $\Gamma \Rightarrow \Delta, A$.
4. If $A, A, \Gamma \Rightarrow \Delta$, then $A, \Gamma \Rightarrow \Delta, A$.

The logical principles.

1. If $\Gamma \Rightarrow \Delta, A$ and $\Gamma \Rightarrow \Delta, B$, then $\Gamma \Rightarrow \Delta, A \wedge B$.
2. If $A, \Gamma \Rightarrow \Delta$ and $B, \Gamma \Rightarrow \Delta, B$, then $A \wedge B, \Gamma \Rightarrow \Delta$.
3. If $\Gamma \Rightarrow \Delta, A$, then $\Gamma \Rightarrow \Delta, A \vee B$.
4. If $A, \Gamma \Rightarrow \Delta$ and $B, \Gamma \Rightarrow \Delta, B$, then $A \vee B, \Gamma \Rightarrow \Delta$.
5. If $A, \Gamma \Rightarrow \Delta, B$, then $\Gamma \Rightarrow \Delta, A \to B$.
6. If $\Gamma \Rightarrow \Delta, A$ and $B, \Gamma \Rightarrow \Delta, B$, then $A \to B, \Gamma \Rightarrow \Delta$.

– two properties : subformula and cut elimination.

Part II

Computability Theory

Chapter 3

Lecture Notes

3.1 Computation

Abstractly, a *computation* is the "work done" by a *computation machine.* For example, we can regard our brains as computation machines and whatever they do is considered a computation. A computation machine M can be viewed as a black box that takes an input w and acts on the input in a step by a step manner. Suppose a brain is tasked with calculating a formula. This formula is the input and at each step the brain may calculate or simplify a certain part of the formula.

A *configuration* is a description of the internals and the state of the machine. Note that the state of the machine may be described in many ways, however, we usually fix a method of describing so that the resulting configurations are unique. A configuration of a brain computing a formula might be the state of neurons in the brain and the steps of the calculation as written on the piece of paper.

In each unit of time the configuration of the machine changes based on its *configuration graph.* This is like saying that in each time unit the brain does an algebraic step such as addition or multiplication and then changes the content of the paper to a new intermediatary result. The configuration graph is a directed graph with all the possible configurations of M as nodes. There is an edges from configuration C to C' if the machine can go from C to C' in one unit of time. In our case, the brain can go from C to C' if the corresponding formula of C' can be arrived by doing one algebraic step from the corresponding formula of C. This graph might be infinite, and in fact, finite configuration graphs can be dealt with tools from graph theory and thus do not provide a rich theory of computation.

In this way, we can formalize the notion of computation machines as a discrete dynamical system.

Definition: A discrete dynamical system is a system whose state varies with a discrete variable, time, according to a directed graph $G = (\mathcal{V}, \mathcal{E})$ such that $\mathcal{V}$, the set of states of the system, is a discrete set. The set $\mathcal{E}$ determines how the state of the system varies with time. Let $V_t \in \mathcal{V}$ be the state of the system at time t, then the state of the system a unit of time later, V_{t+1}, must be a neighbor of V_t. That is,

$$V_{t+1} \in N(V_t) = \{V \in \mathcal{V} \,|\, (V_t, V) \in \mathcal{E}\}$$

Remark 1. In general, $N(V_t)$ might have zero, one, or more than one states. When $N(V_t)$ then the system can not progess and it stops. When V_t has exactly one neighbouring state, the next state is uniquely determined. However, when it has more than one neighbouring states, then the next state is selected non-determinstically. Such selections are not based on some probability distribution or any other rules.

Definition: A **computation machine** is a discrete dynamical system $M = (\mathcal{C}, \mathcal{E})$. The state $C \in \mathcal{C}$ is called a **configuration** of the machine and $E \in \mathcal{E}$ is called a **transition**. On an input w, the machine will be placed in an *initial configuration* $C_0^w \in \mathcal{C}$. At each unit of time, a neighbouring configuration is chosen if it exists. If no neighbouring configurations exists, then the machine *crashes.* A computation machine M is deterministic for any $C \in \mathcal{C}$ there is at most one neighbour.

The possible configurations that are attained by an input w, form a tree. If the computation is deterministic, then the tree is a path.

Remark 2. In computation, we usually distinguish between halting and crashing. Loosely defined, halting is an expected stopping and crashing is an unexpected stopping. To distinguish between these two scenarios, we may consider $\mathcal{C}_h \subset \mathcal{C}$ as the set of halting configuration and add it to the description of M. Therefore, if a non-halting configuration has no neighbour, the computation machine crahses.

When viewing our current definition of the computation machines as computer programs, we see certain similarities. For example, a computer program is a set of variables and instructions. At each step, an instruction is executed which will change the value of the some the variables– e.g. the instruction pointer. The input of the program is considered as one of the variables. Then, the values of the variables is the configuration of the program. The instruction are the transition between different configurations.

However, there is a major difference between computer programs and computation machines. Computer programs are finite, however, computation machines are not. As stated above, the set of configurations of any useful machine is infinite. As a result, the set of transitions is also infinite. On the other hand, a practical theory of computation requires a certain finite limitation. This finite limitation is placed on the description of the computation machine. That is, there exists a finite set symbols such that the computation machines can be described by a finite string of those symbols.

Therefore, if we can describe a machine by a string, it makes sense to assume that its configurations and transitions correspond to some strings. As a result, a computation machine can also be viewed as a string processor. We want our string processor to have a finite description.

A string processor's configurations are strings and its transition describe how these strings change. There are infinitely many possible strings. Thus, the challange is to give the correct result/appropriate action for infinitely many input by a finitely describable entity. Hence, finite description is the same as saying that configuration graph can be constructed by finitely many rules. Finitely many rules implies that configurations change locally. Local property is necessary for finite description. Basic operations are changes in bounded number of memory.

3.1.1 Problems

A problem is a query on some given data and constants. To answer the query by a computation machine, we need to encode the data of the problem into some string. This string is then given to the computation machine. The computation machine takes into consideration the constants and query of the problem so that on any valid input it returns the answer to the query.

The coding of the problem must not have any pre-processing. We enforce this by making the coding essentially trivial. The reason being that different coding may result in better or faster computation.

3.2 Formal Languages

- Suppose we want to compute a function $f : \Sigma^* \to \Pi^*$ with yes/no answer. That is, $\forall w \in \Sigma^*,\ f(w) \in \{0, 1\}$. Then, computing $f(w)$ is equivalent to determining $w \in L_f$ where $L_f = \{w \mid f(w) = 1\}$.
- Generally a problem type is described by a triple of constants, given data, and a query. Yes/no problems have queries which are either true or false.
- To have a well-defined yes/no problem, the set L must have a finite description.

3.2.1 Finite description of formal languages

1. Logical: $L = \{x \mid P(x)\}$ for some proposition P on strings.
2. Grammar: $L = \langle A, \tau_1, \ldots, \tau_n\rangle$ where A is a set of axioms and τ_i are the rules that allow us to combine the axoim, inference rules.
3. Computation Machine: $L =$ is the language of a computation machine, a program in a compiler.
4. Enumerator: $L =$ is the language of a an enumerator, a generator in a compiler.
5. Formal Game: $L =$.?
6. Transducer?

In general, the classes of languages that can be described by several of the above's methods are interesting.

3.3 Computation Machines

Computation is a finite description for the formal language and an evaluation process. Types of machines:

- Deterministic vs Nondetermninistic. Technically, any deterministic machine is also nondetermninistic.

Machines may have some halting configurations. Based on that we can convene the certain halting configuration are accepting configuration and others are rejecting configuration. Based on that we can define certain languages for the machines, for example, the input that are accepted. Then, we have two further classifications.

- Acceptors vs Deciders.

Therefore, a computation is described by its finite description and its standard language. Deciders are closed under inversion of output.

The main problem in theory of computation is determining the relationships of the subclasses of a model. For example, whether DetDec are a proper subset of DetAcc or are they equal.

3.3.1 Finite automaton

3.3.2 Pushdown automaton

3.3.3 Turing machine

3.3.4 Enumerators

3.4 Complexity of the languages

We want to measure the complexity of the problems wrt some model. One way is to reduce the problems to each other.

many-to-one reduction. B must be serializable. hardness and completeness. isomorphism conjecture. Berman-Hartmanis conjecture

3.5 Gap problems

Gap problems between NATM and DATM: No. Gap problems between DATM and DDTM: No. Gap problems between NATM and DDTM: Yes. Gödel's theorem.

Halting problem. Complete problem.

1. Fix a coding for a TMs in $\{0, 1\}$.
2. It is like numbering them.
3. Determining if a number is TM are not is a simple syntax checking.
4. Construct a language that does not have a DTM. Done by the Cantor's diagonalization.
5. If lucky, the language has a acceptor.

3.6 Constraining Turing machines

3.6.1 Time constraints

redefine the TM as binary and single taped. define a the TIME as the length of the computation. Define $f(n)$-boundedness of a machine. Now limit by choosing f.

Write the Gödel's proof for P vs NP.

$NP \subset DDTM$.

Combination of two poly TM is another poly TM.

NP is like having a poly certifier.

3.6.2 Space constraints

redefine the TM as 3-taped: input RO, worktap RW, output WO. Argue that this does not affect our computation capability and results in the same class of languages (almost).

LBAs and the context sensitive grammers.

Part III

Advanced Algorithms

Chapter 4

Introduction

The topics are advanced data structures – Heap, Splay tree, suffix tree and suffix array– and advanced algorithms – string mathcing and network flow. There is also an overview of complexity classes, reducibility, and completeness. Approximation and randomized algorithms are also considered.

4.1 Computation Variants

Computation models include main memory, external memory, and streaming memory. Consider the sequence $x = x_1 x_2 \ldots x_n$ where each $x_i \in \mathcal{U} = \mathbb{N}_m$, count the number of unique elements in x, $D(x)$.

1. naive approach.
2. hash approach.
3. Sample based approach. relative error $\sqrt{\frac{n-r}{2r} \ln \frac{1}{\delta}}$. proof is wtf
4. Streaming model. deterministic memory $M = \Omega(n \log m)$.
5. Randomized approximation. with a factor of $1 + \epsilon$ and with probability of δ, the order of some shit is $O\left(\log \frac{n}{\epsilon^2} \log \frac{1}{\delta}\right)$.
6.

– Distinct sampling.

Chapter 5

Amortized Analysis

5.1 Table insertion

some stuff about potential with deletion we could have an obvious strategy and a simple startegy.